第８回
情報科学ワークショップ

The 8th Workshop on Theoretical Computer Science
Kobe, Hyogo, September 2012
(WTCS2012)

Toshimitsu Masuzawa
Hirotsugu Kakugawa
Fukuhito Ooshita

主担当： 大阪大学

目次

第8回情報科学ワークショップ　プログラム

みのたにグリーンスポーツホテル（〒651-1252 神戸市北区山田町原野）

1日目（9月1日）　　送迎バス 12:30谷上駅発

13:10-13:15　開会

13:15-14:50　セッション1（並列・分散システム）　座長：和田幸一

中村 純哉（阪大）	M	A method of Parallelizing Consensuses for Accelerating Byzantine Fault Tolerance
金 鎔煥（阪大）	M	高可用性Hadoopシステム実現のためのNameNode分散化
金 鎔煥（阪大）	M	仮想化技術を導入したHadoopシステムの実践的評価
上野 健次郎（名工大）	S	MapReduce上での最小全域木アルゴリズムに対する入力データ分割に関する考察
周 昕（広大）	S	Fast Hough Transform Using DSP blocks and block RAMs on the FPGA

15:10-16:40　セッション2（GPU1）　座長：金鎔煥

伊藤 靖朗（広大）	L	Accelerating Dynamic Programming for the Optimal Polygon Triangulation on the GPU
内田 晃裕（広大）	L	An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem
満 都呼（広大）	L	Accelerating Computation of Euclidean Distance Map using the GPU with Efficient Memory Access

17:00-18:00　セッション3（分散アルゴリズム）　座長：泉朋子

佐々木 徹（九大）	L	完全k部グラフにおける移動ビザンチン合意問題アルゴリズムの提案
神崎 裕信（名工大）	S	動的ネットワークにおける関数監視問題の定式化について
服部 雄輔（名工大）	S	弱安定アルゴリズムに対する遷移グラフに関する考察

18:30 夕食
19:30 自由討論会

2日目（9月2日）
07:30 朝食
09:00-10:00 セッション4（アルゴリズム・モバイルシステム）　座長：中村純哉

泉 泰介（名工大）	L	A New Direction for Counting Perfect Matchings
鈴木 大輔（名工大）	S	2次元軌跡データの比較における文字列アルゴリズムの応用について
栗林 大輔（名工大）	S	軌跡データベースにおけるk点連結最良軌跡問い合わせのキャッシュを利用した高速化

10:20-11:35 セッション5（GPU2）　座長：大下福仁

中野 浩嗣（広大）	LL	Memory Machine Models for GPUs
笠置 明彦（広大）	S	Implementation of Data Permutation on the GPU

12:00 自由討論会　　送迎バス 12:15ホテル発 18:00谷上駅発
18:30 懇親会・自由討論会

3日目（9月3日）
07:30 朝食
09:00-10:20　セッション6（自己安定）　座長：伊藤靖朗

八木 渉（名工大）	M	マルチキャストツリーを構成する自己安定アルゴリズムに関する研究
瀧元 友也（阪大）	L	無線ネットワークにおけるエネルギー効率に優れた自己安定プロトコル
山内 由紀子（九大）	L	乱択スケジューラの下での乱択弱自己安定アルゴリズム

10:40-11:40　セッション7（エージェント）　座長：山内由紀子

柴田 将拡（阪大）	L	非同期リング上におけるモバイルエージェント部分集合アルゴリズム
妻鹿 敏也（阪大）	L	同期リング上におけるモバイルエージェント均一配置アルゴリズム

11:40-11:50　閉会　　送迎バス 12:15ホテル発

A method of Parallelizing Consensuses for Accelerating Byzantine Fault Tolerance

Junya Nakamura, Tadashi Araragi, Toshimitsu Masuzawa, and Shigeru Masuyama

Abstract

We propose a new method that accelerates asynchronous Byzantine Fault Tolerant (BFT) protocols designed on the principle of state machine replication. State machine replication protocols ensure consistency among replicas by applying operations in the same order to all of them. A naive way to determine the application order of the operations is to repeatedly execute the BFT consensus to determine the next executed operation, but this may introduce inefficiency caused by wait for the completion of the previous execution of the consensus protocol. To reduce this inefficiency, our method allows parallel execution of the consensuses with keeping consistency of the consensus results at the replicas. In this paper, we also prove the correctness of our method and experimentally compare the existing method in terms of latency and throughput. The evaluation results show that our method makes a BFT protocol three or four times faster than the existing one when some machines or message transmissions are delayed.

Byzantine fault tolerance; asynchronous distributed system; agreement; consensus; state machine replication;

1 Introduction

Byzantine failures, which have no restriction on behavior of faulty machines, are the most malicious failures. Such failures can model any kind of malfunction caused by hardware faults, infection by a virus, intrusion of crackers and so on. In the services provided on open networks like the Internet, these failures cause serious damage, and thus, robust fault tolerance against them is strongly demanded.

One of the most robust approaches for implementing Byzantine fault tolerant services is state machine replication [1], where a server is modeled as a *state machine* and replicated on different host machines. The behavior of a state machine is determined by its current state and the set of received messages. In the state machine replication of a server, some server replicas are arranged and execute the same tasks to tolerate Byzantine faults. To maintain consistency among the replicas, they communicate with each other and agree on the order of processing the received requests, which may arrive in different order at different replicas. By processing the requests in an order common to all the replicas, non-faulty server replicas behave identically. Even if a minority of replicas malfunction and return wrong or forged results, clients receive the same and correct results from a majority of replicas (or non-faulty ones) and can ignore such wrong or forged results from the faulty replicas. Here, we assume that the clients are non-faulty and multicast identical requests to all the server replicas. Thus a main technical issue of state machine replication is to develop a Byzantine consensus protocol for achieving the above agreement in the presence of Byzantine faults.

This paper targets Byzantine fault tolerance for huge distributed systems working on open networks like the Internet. Such systems are generally asynchronous. That is, we cannot guarantee that messages are received in expected time intervals after being sent. As is well known, no deterministic Byzantine consensus protocol exists in asynchronous systems [2]. There are two main approaches for circumventing that impossibility. One is based on randomization [3, 4] and the other is based on a rotating coordinator [5, 6]. Our acceleration method is based on the randomization approach, which is less efficient but more robust than the coordinator approach and is suitable for open networks.

In the randomization approach, randomized actions are introduced to avoid critical damage from attackers. However, the approach is likely to be inefficient, since a number of rounds must be repeated until the correct replicas reach agreement. To improve efficiency, a request set agreement is employed rather than an agreement on a sequential number (the order to be processed) of each request. Once agreement on a request set is achieved, the requests in the set are processed in a predefined order (e.g., the order of the IDs of the clients submitting requests) among them. This request set agreement is repeated sequentially, and all requests are arranged in a common order. However, if some replicas work very slowly or some requests reach very late, a request set agreement may take a long time, which seriously delays the next invocation of the consensus protocol. This paper presents a method of solving this problem by parallelizing the request set agreements.

Next we will explain more details of the randomization approach and the involved problem. Many randomized protocols based on request set agreement have been already proposed [7, 8, 4, 9]. The consensus protocol is invoked periodically with a given time interval, which is measured by a local clock of each replica. When an execution of the protocol is finished by agreeing on a request set, the requests in the set are arranged in a predefined order. By this series of arrangements, all the requests are arranged in a common order among the replicas. At each invocation of the consensus protocol, each replica proposes a set of the requests that were received so far but not included in the previous agreements. Of course, these proposals can be different among the replicas because of the delay of the request arrival or the machine behavior. But the set agreement protocol guarantees that all non-faulty replicas agree on a subset of the union of the request sets proposed by non-faulty replicas.

The length of the local time interval between invocations of the set agreement affects the efficiency, but it is difficult to decide a suitable one. If it is short, the number of invocations of the consensus protocol will increase. If it is long, requests have to wait long for the invocation of the agreement protocol, and the agreement may take a long time because the size of the proposal grows. When an execution of the consensus protocol does not terminate within the local time interval, a big delay might occur. In this case, the invocation of the consensus protocol is kept waiting until the termination of the previous consensus, even if the local time interval passes, to prevent inconsistency of the total order of requests among the replicas. Such blocking of the invocation makes the following invocations of agreement move backward. As a result, the number of unprocessed requests grows and the efficiency of the replication method is reduced. When request arrivals or machine behaviors are delayed, the validity check becomes very time consuming in the agreement, and the termination is easily delayed over the local time interval. Here, the validity check is a process in the agreement for excluding forged requests.

1.1 Contributions

To solve the above problem, we introduce a method that parallelizes the agreement so that executions of the set consensus protocol are not blocked by delayed requests or machines. Our experimental results show that our parallelization method greatly improves the efficiency compared with a sequential method, especially three or four times faster when some requests are delayed or some replicas work slowly.

We solved the following two technical issues:

Safety problem: The parallel executions of the set consensus protocol may terminate in different orders among the replicas. For example, on one replica, the execution of the agreement initiated first terminates after the one initiated second, and on another replica, the one initiated first terminates first. When the replicas are restricted to process the requests in the invocation order of the agreements, they have to wait until the delayed agreement is completed, which may reduce the efficiency achieved by parallelization. Therefore, we have to consistently arrange the outputs (or request sets) of the parallel executions among the replicas.

Liveness problem: A request contained in the proposal made by a replica is not necessarily included in the output of the corresponding agreement. Therefore, to guarantee the liveness that a request is eventually processed, a replica has to keep proposing the request until it is included in an output of the agreements. Therefore, a request that delays agreement can commonly be included in the proposals of the parallel executions of the agreement. This reduces the positive effects of parallelization.

To solve the safety problem, we introduce another agreement process in the replication protocol that identically arranges the output of the parallel executions among the replicas. We show that this additional agreement's overhead is small by experimentally evaluating the performance.

To solve the liveness problem, we introduce randomization to decide the proposals of each execution of the consensus protocol. The requests in the proposal are chosen randomly from the requests that have already been received but have not been processed. A request that causes a delay in a previous execution may be missed in this choice, and a new execution can have no delay. We experimentally show that this randomization brings a reasonable advantage of response time.

1.2 Related work

As stated above, there are two main approaches for replications based on Byzantine agreement in asynchronous distributed systems: randomization [3, 4] and a rotating coordinator [5, 6].

In the rotating coordinator approach, a special replica (a rotating coordinator) determines a sequence number (the processing order) for each received request and announces it to all the other replicas. Therefore, all the replicas can process the requests in the same order and maintain consistency.

If the coordinator is faulty, its role is taken over by another replica. From the impossibility result of FLP [2], this approach needs some assumptions on synchrony (weak synchrony) to guarantee termination. On the other hand, the randomization approach guarantees termination with probability 1 and needs no additional assumption, and it is more robust but less efficient.

Among the protocols in the coordinator approach, the Castro-Liskov protocol [5] achieves very high performance and is considered a practical replication method. Under the above assumption, it terminates in a few rounds and executions of the consensus protocol are executed in parallel. Although the original Castro-Liskov protocol executes the consensus protocol for each request, it is not hard to modify the protocol to allow each process to propose a request set like the randomization approach. However, parallel execution of the agreements for request sets in the coordinate approach is essentially different from that in the randomization approach. Actually, the modification of the Castro-Liskov protocol reduces the number of agreement executions and, consequently improves efficiency in ordinary situations. But it worsens when requests or replicas are delayed. Because of the delay, a coordinator is suspected to be faulty and coordinator alternation often happens. At each alternation, a heavy load procedure must be done to maintain this protocol's integrity.

For the existing protocols in the randomization approach, to the best of our knowledge, our parallelization proposal is the first.

1.3 Organization

This paper is organized as follows. The next section defines the system model. State machine replication is defined in Sect. 3. Section 4 briefly describes an existing replication approach using consensus protocols and specifies what requirements such protocols must satisfy. Our parallelizing method is proposed in Sect. 5, and Sect. 6 proves its correctness. The performance of our proposed method is evaluated and compared with an existing method to show its advantages in Sect. 7. Finally, Sect. 8 concludes this paper.

2 System Model

A distributed system consists of *processes* and *communication links*. We assume that the system is *asynchronous*, i.e., no assumptions are made about the bounds of processing time or communication delays. Every pair of processes is directly connected by a communication link, and a process can exchange information only by exchanging messages. We assume that communication links are *reliable channels*, i.e., messages sent by correct processes must eventually be received by the destination processes without corruption or loss. A process can identify the sender process of each delivered message, for example, by the signature, and no process (even a malicious one) can impersonate other processes when sending messages. A process has a local clock, but it is not synchronized; clocks of different processes may be running at different speeds.

Some processes may fail during the protocol execution. Here, we adopt *Byzantine* failure (also called arbitrary failure) as a failure model. Byzantine failure allows processes to arbitrarily deviate from protocol specifications, e.g., to stop processing, omit messages, and send fabricated messages. A process is called *faulty* if its behavior deviates from the protocol specification, otherwise it is called *correct*.

3 State Machine Replication

In state machine replication [1], a server is modeled by a state machine, which is a process that, on receipt of a message, changes its state and sends messages to other processes (if necessary). The server's role is replicated to n replicas that independently operate the role on distinct hosts and interact with clients by request and response messages. A client submits a request to all replicas to request the servers to execute certain commands. Even though the arrival orders of the requests at different replicas may differ because of differences in communication delays, the replicas must process the requests in the same order to keep consistency among the replicas.

More formally, a state machine replication method must satisfy the following two requirements:

Safety All correct replicas process the requests submitted by clients in the same order.

Liveness A client eventually accepts the response to any request it submitted.

To realize identical processing order of requests, the replicas execute a consensus protocol. After a replica processes a request, it replies to the client with the execution result. The client accepts the result when it receives the same result from $f + 1$ replicas. Here f is the upper bound of the number of faulty replicas. A client can confirm that at least one correct result was received from a correct replica when it collects $f + 1$ identical results. Since n must be greater than or equal to $3f + 1$ to realize Byzantine consensus by randomized protocols [10], we assume that $f \leq \lfloor (n-1)/3 \rfloor$.

Figure 1 shows an example of state machine replication. There are two clients and four replicas, and the clients broadcast requests r_1 and r_2. Since its network is asynchronous, the arrival orders of the requests are different among the replicas who execute a Byzantine consensus protocol to agree with the processing order of the requests. As a result, the replicas agree with processing order $r_1 \rightarrow r_2$, process the requests in the order, and send their responses to the clients.

4 Replication by Request Set Consensus (RSC)

We introduce a state machine replication method based on Byzantine consensus on a set of requests (called *request set consensus* (RSC)), which is commonly used in replications in completely asynchronous distributed systems to accelerate replication execution.

In this replication method, a replica periodically initiates RSC with a predefined interval. We denote the sequence of RSC executions by $RSC^1, RSC^2, \cdots$. A replica maintains the *arrived request set* to store the set of the

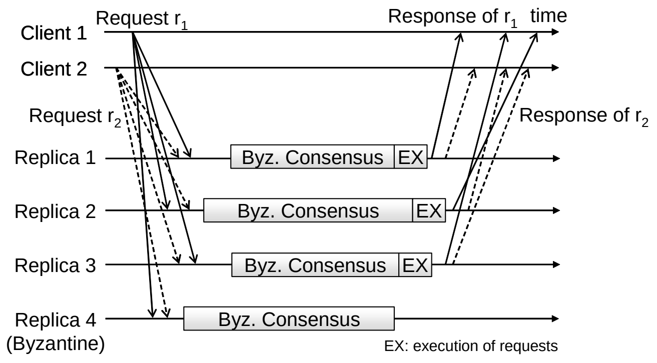


Figure 1: Example of state machine replication

requests that have already been received but have not yet been processed; a request is added to the set when it is received, and it is removed when it is processed. When a replica initiates RSC^k, its proposal is the set of the requests stored in the arrived request set. Let the output (a set of requests) of RSC^k be V_k. Requests are processed in the order of $V_1, V_2, \dots$, and the requests in each V_i are serialized in a deterministic order shared among the replicas. In the existing methods, the initiation of RSC^{k+1} must be delayed until RSC^k is finished to maintain the consistency of the processing order of requests, even if it passes the scheduled initiation time of RSC^{k+1} (*Initiation Condition*).

To ensure the safety and liveness requirements for state machine replication, the RSC protocol must satisfy the following requirements. Hereafter we denote an execution of RSC^i at a replica with proposal v by $RSC^i(v)$ or RSC^i if the proposal does not matter.

RSC agreement No distinct correct replicas output different sets of requests.

RSC validity The output set is a subset of the union of the proposals of all correct replicas.

RSC termination Every correct replica eventually outputs a set of requests.

RSC integrity A request contained in the proposals of all correct replicas is also contained in the output.

RSC agreement, *validity*, and *termination* are standard requirements for Byzantine consensus protocols. *RSC integrity* suffices to guarantee the *liveness requirement* of state machine replication.

Figure 2 illustrates replication behavior using RSC. There are four replicas, and replica 4 fails. The replicas initiate the i th execution of RSC with the proposals of the arrived request sets. Since the system is asynchronous, the arrival orders of the requests may be different among the replicas and the RSC^i proposals may be different. Actually, in the example, replica 1 proposes $\{r_1, r_2, r_4\}$ and replica 2 proposes $\{r_2\}$ for RSC^i. Faulty replica 4 makes

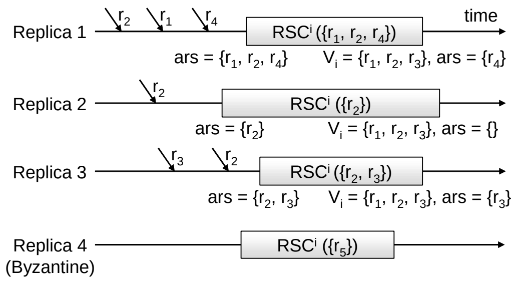


Figure 2: Example of replication by Request Set Consensus (RSC)

forged request r_5. When RSC^i is finished, the replicas obtain common request set $V_i = \{r_1, r_2, r_3\}$ as an agreed on value. Although the contents of the agreed set depend on the message delivery and process execution schedules, r_2 must be contained in the agreed set by RSC integrity. On the other hand, RSC validity guarantees that forged request r_5 is not contained in the agreed set. Arrived request set ars of each replica is modified by removing the requests in this agreed set.

5 Parallelizing Executions of RSC

5.1 Problem with parallelization

Executions of existing replication methods can be very slow due to the initiation condition mentioned in Sect. 4, especially when the behaviors of some replicas are delayed or requests reach some replicas late. One idea to improve the efficiency of the replication method is parallelizing the executions of RSC by consistently removing the initiation condition. To achieve this, we have to solve the following two problems.

Safety problem: Since the delays of the communication links among replicas and clients are different from each other in asynchronous systems, the order of finishing the RSC executed in parallel can be different among the replicas. In Fig. 3, replica p finishes RSC^1 first, while replica q finishes RSC^2 first. If a replica immediately executes requests after the agreements, then the processing orders of the requests are not the same among replicas p and q, and the safety condition is not guaranteed.

This problem can be simply resolved by waiting for the terminations of all RSC^j $(j < i)$ before processing V_i. However, the method can cause great overhead (Fig. 3), where replica q has to wait for the termination of RSC^1 to process V_2. If a RSC takes a long time, all requests already agreed by the following RSCs have to wait to be processed until the previous RSC is terminated.

Liveness problem: Even if we reduce the overhead of waiting for the termination of other RSC executions, inefficiency remains, caused by the delayed replicas or the

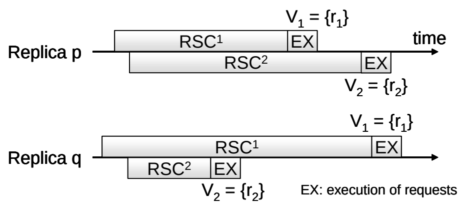


Figure 3: Invalid parallel executions of RSC

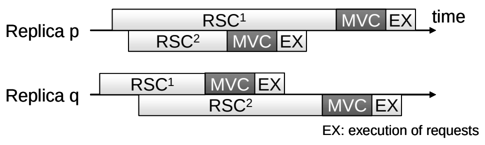


Figure 4: Execution of our proposed parallelizing method

delayed requests. A request included in a proposal may not be included in the output. Therefore, the replica must keep proposing the request until it is included in an output of RSC to guarantee the liveness requirement for state machine replication.

In such a naive parallelization, the proposal of RSC^{j+1} is likely to contain a request in that of RSC^{j}. However, if the request is greatly delayed for some replicas, the validity check in the protocol commonly takes a long time for both executions RSC^{j} and RSC^{j+1}. Therefore, a few delayed requests may cause big delays in the parallel execution of RSC.

5.2 Our approach

To solve the safety problem, we introduce a multi-valued consensus (MVC) in the parallelization. When an execution of RSC^{j} is finished in a replica with output rs_j, the replica initiates MVC with proposal (j, rs_j) (Fig. 4). If MVC outputs agreed value (id, rs_{id}), the replicas process the requests in rs_{id} in an arbitrary predefined order. All correct replicas clearly process the same requests in the same order. Note that MVC is itself executed sequentially on each replica. An important point of this method is that the replica does not have to wait for the termination of RSC^{i} ($i < id$). In addition, even if RSC^{id} has not finished at the replica, it can process the requests in rs_{id} since the replica can learn the requests from the MVC output.

To solve the liveness problem, we introduce randomization for deciding the proposal. We decide the RSC proposal by probabilistically choosing requests from the set of requests already received but not yet processed. With this simple modification, we can decrease the probability that the terminations of successive RSCs executed in parallel are delayed by the same request. At the same time, we can guarantee the liveness requirement with probability 1.

5.3 Multi-valued consensus protocol

We show the requirements for the multi-valued consensus protocol used to determine the request set to be processed first. The MVC proposal at a correct replica is a pair of ID of a terminated RSC execution and its agreed request set. The MVC protocol is, of course, the randomized protocol, because the targeted distributed system is asynchronous. The MVC protocol must satisfy the following requirements:

MVC agreement No distinct correct processes output different values.

MVC validity If the proposals of all correct processes are the same, the agreed value is the proposal.

MVC termination Every correct process eventually outputs an agreed value.

MVC extra validity The output of a correct process must be a proposal of some correct process.

MVC agreement, *validity*, and *termination* are the common requirements for MVC in general. *MVC extra validity* speeds up state machine replication while avoiding forged requests, which is explained in Sect. 5.4. MVC extra validity is feasible using a signature scheme on an existing MVC protocol. Each replica repeatedly executes MVC, and we denote the i th execution of MVC by MVC^{i}.

5.4 Protocol

Our proposed parallelizing method is shown in Protocol 1. The value of *input_rs* is a set of requests, which is given to RSC as a proposal. The value of *old_rs* is a set of requests that were received before the last RSC initiation and remain unprocessed. The value of *new_rs* is a set of requests that were received after the last RSC initiation. The value of *agreed_rs* is a set of requests that belong to RSC output. *rsc_id_queue* is a queue of pairs (j, rs_j) of RSC ID j and agreed set rs_j output by the execution of the RSC with ID j, whose element is a proposal of MVC. *wait_queue* is a queue of agreed request sets, and a thread $T_{process}$ processes them in order. *mvc_id* is a counter that gives a sequence number to each execution of MVC, allowing replicas to recognize a common execution of MVC.

We assume that each replica has its own special scheduler PS, which employs a local clock of the replica. PS periodically outputs positive integers 0, 1, 2, ... in this order with a predefined interval. When PS outputs number k, the replica initiates the k th execution of RSC with ID k. The shorter the PS interval is, the more frequently RSC is initiated.

A replica initiates MVC with a proposal of a pair of an RSC ID and its agreed set. If MVC^{j} outputs the agreed value (id, V), the replica processes V at the j th turn. The MVC proposal includes the corresponding agreed set as well as the RSC ID to improve the efficiency. If the proposal is only RSC ID, when MVC outputs RSC ID *id* and the replica has not finished the execution of the RSC of *id*, it has to wait for the termination of the RSC before processing the requests in the agreed set. With the agreed value in the output of MVC and *MVC extra validity*, which means that the agreed value is not forged, a replica can process the correct request set immediately after the MVC outputs.

Our method starts from initialization in which a replica creates a new thread $T_{process}$. $T_{process}$ dequeues a request set from *wait_queue* and processes the elements in a deterministic order shared with all replicas.

Our protocol has four *when* clauses:

- When a new request arrives from a client, it is added to *new_rs*.
- When scheduler *PS* outputs value j, first, the already agreed requests are removed from *old_rs* and *new_rs*, and the proposal for a new RSC is calculated using given function *choose*, which randomly selects requests from its input *old_rs* and *new_rs* in a predefined manner. Then a new RSC with ID j is initiated, and the elements in *new_rs* are moved to *old_rs*.
- When an RSC execution with ID *id* is finished with output *rs*, a replica updates *agreed_rs* and enqueues a pair (id, rs) to *rsc_id_queue* if *id* is not in *agreed_rsc_id*. If a previously invoked MVC is running, it waits for the termination. Then the replica chooses the first element, a pair of an RSC ID and an agreed set (id', rs') from *rsc_id_queue* (without deleting it from the queue), initiates a new MVC with ID *mvc_id* with proposal (id', rs') and increments the value of *mvc_id*.
- When MVC outputs value (id, rs), a replica removes the pair whose first element is *id* from *rsc_id_queue* and enqueues *rs* into *wait_queue* and *id* into *agreed_rsc_id*.

6 Correctness

We prove that our proposed protocol, which parallelizes RSC in state machine replication, satisfies the safety and liveness requirements of state machine replication.

Safety We have to show that requests are processed in the same order among the non-faulty replicas and that no forged requests are included in them.

To show that requests are processed in the same order, it is sufficient to show that RSC outputs are enqueued to *wait_queue* in the same order among the replicas under RSC agreement since thread *T_process* processes the requests in the order in which they are stored in *wait_queue* (line 39). On the other hand, enqueuing is executed only in the event of MVC output, and MVC is executed sequentially (lines 26–30), and then the desired result follows from the MVC agreement. A non-forged requirement immediately follows from RSC validity and MVC extra validity. □

Algorithm 1 Proposed parallelizing method

```
 1: Variables
 2:     $input\_rs := \emptyset$; {input of RSC}
 3:     $old\_rs := \emptyset$; {requests received before the last RSC}
 4:     $new\_rs := \emptyset$; {requests received after the last RSC}
 5:     $agreed\_rs := \emptyset$; {agreed requests}
 6:     $agreed\_rsc\_id := \emptyset$; {RSC IDs agreed by MVC}
 7:     $prs := \emptyset$; {processed requests}
 8:     $mvc\_id := 1$; {counter for MVC IDs}
 9:     $rsc\_id\_queue$ := empty; {queue of pairs of RSC ID and a set of requests }
10:     $wait\_queue$ := empty; {queue of agreed sets waiting to be processed}
11: Initialization
12:     start task $T_{process}$;
13: When a request $r$ arrives do
14:     $new\_rs := new\_rs \cup \{r\}$;
15: When $PS$ outputs $j$ do
16:     $old\_rs := old\_rs \setminus agreed\_rs$;
17:     $new\_rs := new\_rs \setminus agreed\_rs$;
18:     $input\_rs := choose(old\_rs, new\_rs)$;
19:     invoke $RSC^{j}(input\_rs)$;
20:     $old\_rs := old\_rs \cup new\_rs$;
21:     $new\_rs := \emptyset$;
22: When $RSC^{id}$ outputs its agreed value $rs$ do
23:     $agreed\_rs := agreed\_rs \cup rs$
24:     if $id \notin agreed\_rsc\_id$ then
25:         enqueue $(id, rs)$ into $rsc\_id\_queue$;
26:     if MVC is running then
27:         wait until it terminates;
28:     let $(id', rs')$ be the first element of $rsc\_id\_queue$;
29:     invoke $MVC^{mvcid}(id', rs')$;
30:     $mvcid := mvcid + 1$;
31: When $MVC^{i}$ outputs its agreed value $(id, rs)$ do
32:     if $rsc\_id\_queue$ contains $(id, *)$ then
33:         remove $(id, *)$ from $rsc\_id\_queue$;
34:     enqueue $rs$ into $wait\_queue$;
35:     $agreed\_rsc\_id := agreed\_rsc\_id \cup \{id\}$;
36: Task $T_{process}$
37:     loop
38:         wait until $wait\_queue$ is not empty;
39:         dequeue $rs$ from $wait\_queue$;
40:         for all $r \in (rs \setminus prs)$ in some deterministic order do
41:             execute $r$ and send the result to the client;
42:         $prs := prs \cup rs$;
```

Liveness Assume that there exists a request rq that has never been processed. Such a request is eventually delivered to all correct replicas and stored in their *new_rs* or *old_rs*. Hence, there must be an RSC execution with some probability in which every correct replica contains rq in its proposal. Let the ID of the execution be k. By RSC termination, the execution must terminate, and by RSC integrity, agreed set V_k must contain rq. Then every correct replica enqueues (k, V_k) into *rsc_id_queue*. Assume that (k, V_k) has never been chosen as an output of any MVC execution. *rsc_id_queue* is a queue, so if (k, V_k) is not removed for a long time, (k, V_k) moves to the front of the *rsc_id_queue*. If the front of the *rsc_id_queue* of every correct replica gets (k, V_k), by MVC validity, the agreed value of the next MVC execution must be (k, V_k), and the execution must terminate by MVC termination. Therefore, request *rq* is eventually processed and contradicts the assumption. □

7 Performance Evaluation

In this section, we experimentally compare the performance of state machine replication employing our proposed parallelizing method with an existing one based on sequential agreements. In particular, we show how the delay of request message delivery and machine behavior affects the response time of the requests. We also evaluate the throughput of the two methods in ordinary and delayed situations.

7.1 Experiment environment

For our experiments, we use five machines completely connected by one network switch. On each of four machines, a replica is running individually. On the other machine, several clients are simulated, and their requests are issued from it. The machines have a Core i3 540 3.07 GHz CPU and 2 GB RAM and run Linux 2.6.18. The network is 1 Gbps LAN. In experiments of performance evaluation, we did not model Byzantine failure because it has thousands of varieties and seldom occurs.

Through the experiments, we fix the *choose* function so that it uniformly chooses every element as an element of a proposal with 0.25 probability. This value is empirically preferable for the parallelization as shown in Sect. 7.2.2.

We used the RSA protocol proposed in [9, 11] as an underlying RSC protocol and the M_V_Consensus protocol proposed in [7] as the MVC protocol. These protocols and our proposed parallelizing method were implemented by C++ language with POSIX socket library for the evaluation. Note that the M_V_Consensus protocol may output a special value, $\perp$, which is different from any proposed value. To cope with this exceptional value, we slightly modified our protocol. When this value is output, we reinvoke M_V_Consensus protocol with a different proposal: the element of *rsc_id_queue* whose RSC ID is the smallest. If the repetition of this reinovocation continues, the proposals finally coincide among the replicas, and the invocation terminates by outputting the proposal of a normal value by the M_V_Consensus property stated in Theorem 3 in [7]. Then, the repetition is finished.

7.2 Latency

7.2.1 Evaluation model

From the machine that simulates clients, 50 requests are multicast to the replicas in total. Let $r_1, r_2, \ldots, r_{50}$ be the requests issued from the clients. To realize delayed delivery of the requests, we change the order of sending the requests. For example, if the delivery of request r_1 is delayed for replica R_1, we send the requests to the replicas other than R_1 in the order $r_1, r_2, \ldots, r_{50}$ and the requests to R_1 in the order $r_2, r_3, \ldots, r_{25}, r_1, r_{26}, \ldots, r_{50}$. To realize delayed behavior of the replicas, we delay the timing to start sending the requests to replicas. For example, if the behavior of replica R_1 is delayed, we start sending the requests to R_1 after sending 25 requests to the other replicas.

We introduce the following parameters and values to configure this model:

#*d_req*: number of delayed requests whose values are {1,2,3}.

#*d_rcv*: number of machines that receive delayed requests. Their values are {2*,3*}, where we attach "*" to distinguish them from the values of #*d_req*.

***ed_req*:** extent of how much requests are delayed. The values are {*middle*, *end*}, in short, {m, e}.

***ed_mac*:** extent of how much a machine's behavior is delayed. The values are {0%, 50%, 100%}.

The values, *middle* and *end*, of *ed_req* mean that the first requests are moved backward to the middle and to the end of the order of the sequence of requests, respectively. For example, if #*d_req* = 2 and *ed_req* = *middle*, r_1 and r_2 are moved between r_{25} and r_{26}, and if *ed_req* = *end*, they are moved after r_{50}. We assume that at most one machine can be delayed, which is called a *delayed replica*. The value of 0% of *ed_mac* means that there is no machine delay. 50% and 100% mean that the sending of the requests to the delayed replica starts when the sending of the requests for the other machines has progressed 50% and 100%, respectively. Machine delay *ed_mac* implies delays of all the requests, and request delay *ed_req* does delay some requests.

Each request is issued to a replica every 100 ms. The local time interval for invoking RSC is 100 ms. For each combination of the parameter values, we execute the experiments 50 times and average the response times. The response time of a request is the time from sending it until receiving the same results from $f + 1$ replicas.

7.2.2 Experimental results and analysis

The average response times of the sequential and parallel executions for each parameter configuration are shown in Fig. 5. On the horizontal axis, each configuration is depicted in the form $x1$-$x2$-$x3$-$x4$, meaning that the values

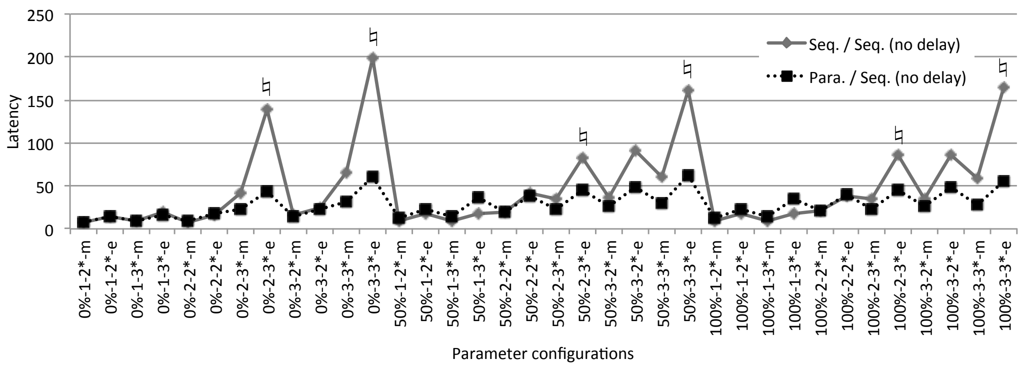


Figure 5: Results for individual parameter configurations

of *ed_mac*, #*d_rcv*, #*d_req*, and *ed_req* are $x1$, $x2$, $x3$, and $x4$, respectively. On the vertical axis, the average response times are measured in the ratio to the average response time of the sequential executions with *no delay* of the delivery of requests or the behavior of replicas.

We clearly observed that at configurations of #*d_rcv* = 3* and *ed_req* = *end*, (i.e., when the number of replicas receiving delayed requests is large and these requests arrive very late, the peaks were marked with ♮ in Fig. 5), the response time of the sequential executions is 150 or 200 times longer than the no delay case, and the efficiency becomes very low. On the other hand, the response time of the parallel executions is at most around 50 times longer than the no delay case. Especially, when the efficiency of the sequential executions is terrible, the good effect of parallel executions is remarkable for the following reason. Multiple replicas that receive many delayed requests cannot indirectly verify the validity of the requests received from other replicas until they receive them directly from clients. This greatly delays the termination of the involved agreement and shifts the following agreements afterward. However, in parallel executions, a new RSC can be started without waiting for termination of the agreement, and the delayed messages have no effect on the following agreements.

Although at configurations of 50-1-3*-e or 100-1-3*-e the efficiency of the parallel executions is worse than that of the sequential executions, the difference is small. This means that the overhead of additional MVC in parallel executions does not have much effect on the whole response time.

Next we focus on the randomization of the RSC proposal. Fig. 6 shows the average response times of parallel executions with different probabilities employed in the *choose* function: 0.25, 0.5, and 1.0. The case of probability 1.0 corresponds to the naive approach without randomization in RSC proposals. As we presumed, the response time is almost the same as the sequential executions, and no advantage of parallelization appears. On the other hand, probabilities 0.25 and 0.5 equally and positively affect parallelization, proving the usefulness of our idea of randomization.

7.3 Throughput

We conduct experiments on throughput to evaluate the amount of resource consumption by parallelization. First, we explain how we evaluate the throughput because reasonably evaluating throughput is a subtle problem at loads exceeding the resource bound of systems. To evaluate the throughput at a given load of request frequency, we execute the protocol for 25 seconds at the load. Here, *request frequency* means the number of requests received by a replica every second. Then we divide the execution into five successive sections of five-second long intervals. For each section, we calculate the number of processed requests and divide it by five seconds to obtain a tentative throughput value. Finally we choose the maximum value among the five tentative throughput values as the throughput value at the load. If the load does not exceed the resource bound, then the tentative throughput value increases and becomes stable. On the other hand, if it exceeds the resource bound, the value first increases and then decreases. Thus, we choose the maximum of the tentative values to commonly characterize the throughput value for both cases. The result for each request frequency listed below is an average value of ten executions. Through the experiments, there is no delay on the delivery of requests or the behavior of replicas, because controlling the delay is difficult in heavy loads.

In the throughput graph, the request frequency at which the throughput peaks corresponds to the load where the system reaches the resource bound. By our calculation, the angle of inclination after the peak shows how fast the resource will be exhausted after reaching the resource bound. A larger angle means faster exhaustion.

In Figs. 7 and 8, we show the throughput of the sequential and parallel executions. In Fig. 7, parallel exe-

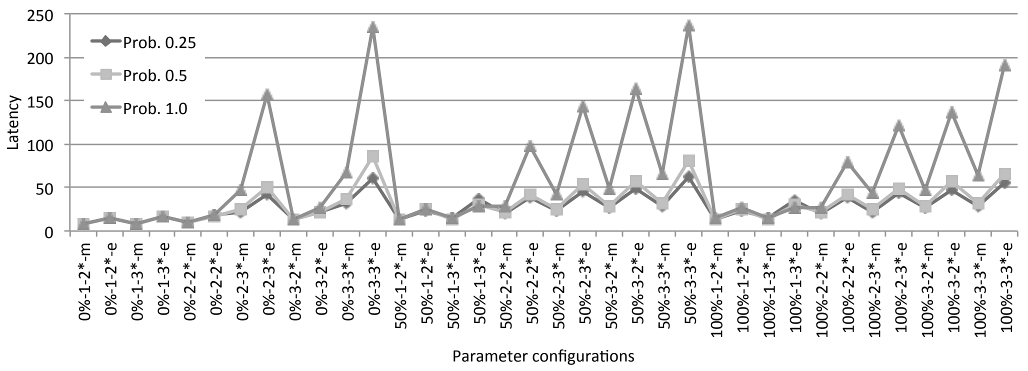


Figure 6: Average response times of parallel executions with probabilities: 0.25, 0.5 and 1.0

cutions are controlled by restricting the number of parallel agreements at a time, denoted by $\#para$. For example, $\#para = 2$ means that if two RSC are being executed in parallel and timing for a new RSC is being invoked, the invocation must wait until one of the executions is terminated. In Fig. 8, we add another restriction on the frequency of the parallel executions of RSC, denoted by $freq$. For example, $\#para = 2$ and $freq = 5$ mean that if two RSCs are executed in parallel and one terminates, parallel RSC execution is not allowed until five newly invoked sequential executions of RSC have been completed.

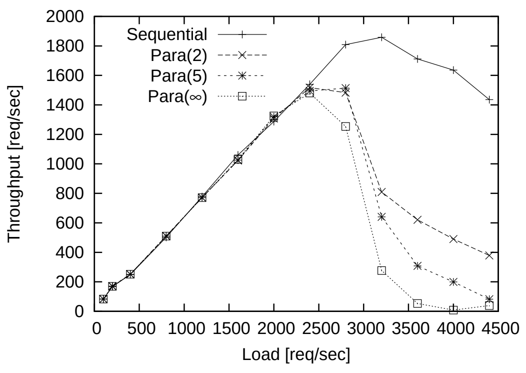


Figure 7: Throughput in restriction on number of parallel RSC executions. Para(x) means "Parallel execution with $\#para = x$".

In Fig. 7, when the value of $\#para$ is large, the parallel execution reaches the resource bound with a smaller load. At loads that fail to reach the resource bound, parallel executions show the same throughput values as the sequential execution. At loads beyond the resource bound, parallel executions exhaust the resource more rapidly. On the other hand, in Fig. 8, if we control the frequency of the parallel executions of RSC, the resource consumption is greatly reduced for $\#para = 2$. Especially if $freq = 10$, the execution reaches the resource bound at the same load as the sequential execution and the speed of exhausting the resource is not so different from the sequential one at loads beyond the resource bound.

From these observations, we conclude that parallel executions consume resources in proportion to the number of consensus protocol instances executed in parallel. When we restrict the number, the executions still exhaust the resources rapidly when the load exceeds the bound, and the speed slows down when we restrict the frequency of RSC because time is required for parallel executions to release the resource. For the practical use of the parallelizing method, when the load is heavy, we should dynamically control the number of parallel executions and their frequency to avoid resource exhaustion.

8 Conclusion

In this paper, we proposed a method to accelerate state machine replication for Byzantine fault tolerance by parallelizing the executions of request set consensus and adding an extra multi-valued consensus for deciding the processing order of agreed sets. We also show the correctness of the protocol for parallelizing agreements. Parallelization has a good advantage in spite of an additional agreement, especially when some replicas work slowly or some requests are delivered late. We showed this property by an experimental evaluation. In this evaluation, our parallelizing method accelerates the latency of replication three or four times more than the existing sequential method in delayed situations.

Acknowledgement

The authors would like to thank Profs. Kakugawa and Ooshita of Osaka University for their helpful discussions about improving this paper.

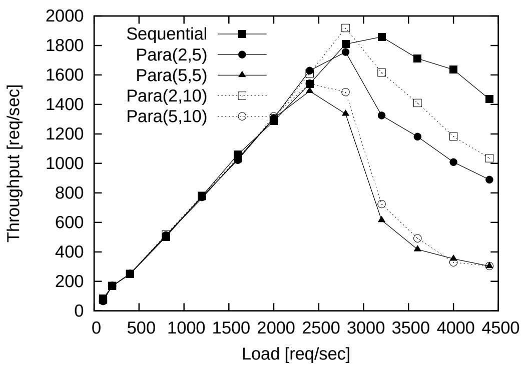


Figure 8: Throughput in additional restriction on frequency of parallel RSC executions. Para(x,y) means "Parallel execution with $\#para = x$ and $freq = y$".

References

[1] F.B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," ACM Computing Surveys, vol.22, no.4, pp.299–319, 1990.

[2] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of distributed consensus with one faulty process," J. ACM, vol.32, no.2, pp.374–382, 1985.

[3] H. Moniz, N. Neves, M. Correia, and P. Verissimo, "Ritas: Services for randomized intrusion tolerance," Dependable and Secure Computing, IEEE Transactions on, vol.8, no.1, pp.122–136, 2011.

[4] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," Lecture Notes in Computer Science, vol.2139, pp.524–541, 2001.

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance," OSDI '99: Proceedings of the third symposium on Operating systems design and implementation, Berkeley, CA, USA, pp.173–186, USENIX Association, 1999.

[6] J.P. Martin and L. Alvisi, "Fast byzantine consensus," IEEE Trans. Dependable Secur. Comput., vol.3, no.3, pp.202–215, 2006.

[7] M. Correia, N.F. Neves, and P. Verissimo, "From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures," The Computer Journal, vol.49, no.1, pp.82–96, 2006.

[8] H. Moniz, N.F. Neves, M. Correia, and P. Veríssimo, "Randomized intrusion-tolerant asynchronous services," DSN '06: Proceedings of the International Conference on Dependable Systems and Networks, pp.568–577, 2006.

[9] J. Nakamura, T. Araragi, and S. Masuyama, "Asynchronous byzantine request-set agreement algorithm for replication," Proceedings of the 1st AAAC Annual Meeting, p.35, 2008.

[10] G. Bracha and S. Toueg, "Resilient consensus protocols," PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing, New York, NY, USA, pp.12–26, ACM Press, 1983.

[11] J. Nakamura, T. Araragi, and S. Masuyama, "A fast randomized byzantine fault tolerance method for the system replication (japanese)," Proceedings of Electronics, Information and Systems Conference, pp.102–107, TC4, 2007.

高可用性 Hadoop システム実現のための NameNode 分散化

金 鎔煥 †　　中村 純哉 †　　櫟 粛之 ‡　　増澤 利光 †

† 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻
‡NTT コミュニケーション科学基礎研究所

概要

近年，クラウドコンピューティングや SNS などの普及により，扱うデータの量が急増している．BigData と呼ばれるこのような大規模なデータは，既存のデータ処理手法では時間的および空間的資源の激しい消費が予想されるため，効率的な処理のために新たなシステム又は手法が必要となる．このような BigData を効率良く管理及び処理するために提案されたフレームワークとして Hadoop が注目を浴びている．Hadoop は，Google File System をベースにして開発された HDFS(Hadoop Distributed File System) と，分散処理フレームワークである MapReduce で構成される．Hadoop は，コモディティ性の高い計算機だけでも高いデータ処理能力を持ち，大規模のデータも安全に管理し，効率的な処理を可能にする．しかし，Hadoop システムはたった 1 台だけのマスタノードによってクラスタ全体が管理されるため，様々な問題が生じる．マスタノードが単一障害点 (Single Point of Failure) であること，負荷がマスタノードに集中されることなどが問題としてあげられる．本研究では，Hadoop のマスタノードを複数の計算機に分散させたシステムを提案する．この分散化により，単一障害点の解除，負荷分散などの効果を期待できる．

1　はじめに

近年，クラウドコンピューティングや SNS などの普及により，扱うデータの量が急増している．IDC [7] は，2011 年のデジタル世界 (Digital universe) の大きさは $1.8ZB(1.8 \times 10^{21}B)$ だと予測している．このように急増している膨大なデータを保存および管理することは，既存のデータ処理手法では時間的および空間的資源の激しい消費が予想される．そのため，効率的な処理のために新たなシステム又は手法が必要となる．このように既存のシステムや手法によって処理することが困難である膨大なデータを BigData [6] と呼び，近年の IT 業界での重要な課題として挙げられている．このような BigData を効率良く管理及び処理するために提案されたフレームワークとして Hadoop が注目を浴びている．Hadoop は，Google File System [3] をベースにして開発された HDFS(Hadoop Distributed File System) [5] と，分散処理フレームワークである MapReduce [4] で構成される．

Hadoop システムにおいて，分散ファイルシステムを提供するフレームワークである HDFS は，Master/Worker 構造になっていて，Master ノードは NameNode，Worker ノードは DataNode と呼んでいる．Master ノードである NameNode は，HDFS において，1 台だけ存在する．HDFS では，NameNode が全てのファイルのメタデータを管理し，実際のデータは多数の DataNode に分散されて保存される．保存される各データは一定の大きさで分割されて，この分割されたブロックを多数の DataNode に分散及び複製して保存する．複製ブロックも保存しておくことで，HDFS では一部の DataNode に障害が生じても安定的にデータを保存することを可能にしている．

NameNode は全てのファイルの情報を管理する．HDFS に保存された各ファイルの名前，パス，権限などの一般のファイルシステムで使われている情報はもちろん，該当ファイルの複製が何回行われたのか，複製ブロックを含めて全てのファイルブロックがどの DataNode に存在しているのかも保持している．HDFS を利用するクライエントは NameNode のアドレスだけを把握することで，ファイルシステムにアクセスすることが可能となる．また，NameNode が保持しているメタデータを用いて，ファイルの分割ブロックの場所や複製ブロックの状況などを気にする必要なく，一般のファイルシステムのような木構造の NameSpace としてアクセスすることができる．

しかし，NameNode は HDFS において，Master ノードとして 1 台にしか存在しない．このことにより，次のような問題が生じる [11]．

- **単一障害点 (SPOF) の問題**：HDFS に NameNode は 1 台だけ存在し，HDFS の全てのファイルのメタデータを保持する．よって，NameNode に不具合が生じた場合，ファイルシステム全体が利用不可能となる．
- **NameSpace 制限**：NameNode はクライアントのリクエストを迅速に処理するために，全てのメタデータを NameNode のローカルなメモリに保存している．しかし，計算機のメモリは有限資源であるため，上限が存在する．1GB のメモリに保存可能

なメタデータは約 5 万個であり，メモリを増やすことである程度上限を増やすこともできるが，億単位のファイルは確実に保存できなくなる問題がある．

- **負荷の集中**： Hadoop システムは計算機を追加するだけで性能が線形的に上がる優れた拡張性 (Scalability) を持つ．しかし，大規模のシステムになってしまうと，クライアントからのリクエストを処理する NameNode に負荷がかかり，性能向上に限界が生じる．

このような問題を解決するため，本研究では NameNode を単一 Master ノードではなく，メタデータを分割することで分散化することを提案する．

2　既存研究

NameNode の単一障害点問題は，Hadoop システムの可用性において最も致命的な問題であるため，様々な研究が行われている．公式リリースされた Hadoop 1.0 の場合，Secondary NameNode という別のノードに定期的に NameNode の情報をバックアップする方法を適用している [1, 2]．しかし，NameNode が故障した場合，回復に時間がかかる上，最新の情報で復帰するということが保証できない．

別の Hadoop Project のバージョン 0.23 [8] では，完全に同期された 2 つの NameNode を動作させることで耐故障性を実現している．2 つの NameNode は Active と Standby に別れ，故障などによる切り替えを準備する．しかし，このシステムでは共有ディスクが必要な上，NameSpace 制限と負荷集中の問題を解決することができない．Facebook [10]　でも，AvatarNode [9] と呼ばれる Standby ノードを用意することで耐故障性を実現しているが，同じく NameSpace 制限や負荷の問題が残る．

3　提案モデル

図 1 の左の図は NameNode が保持しているメタデータを NameSpace として表せている．実際はメタデータのリストとして NameNode のメモリに保存されているが，クライアントから見えるファイルのシステムは図の

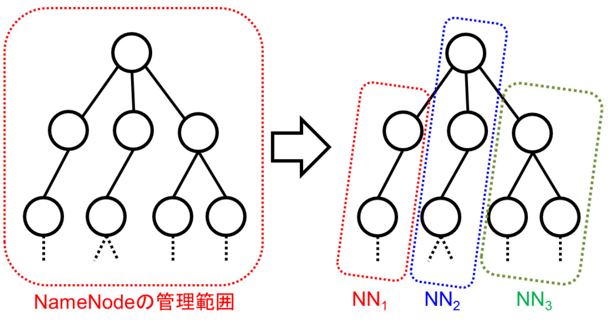


図 1　NameSpace の分割

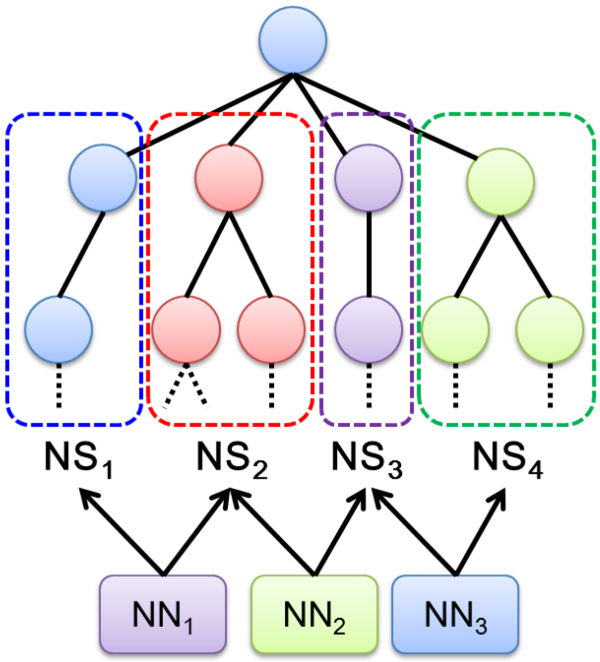


図 2　分割された NameSpace の多重化

ような論理的な木構造となっている．また，NameNode か管理してメタデータ情報の範囲が点線で示されている．**図 1** の左の図のように，一般の Hadoop システムの HDFS では 1 台の NameNode が全体 NameSpace を管理することとなっている．提案するモデルでは，右側の図のように NameSpace を分割，すなわちメタデータの幾つかの集合に分割する．分割された NameSpace は複数の NameNode に分かれて保存される．このように NameSpace を分割して保存することにより，NameSpace 制限の問題を解決される．NameSpace を拡張するためには，追加の NameNode 用の計算機を追加するだけで可能となる．

図 2 は分割された NameSpace を保存する NameNode(図では NN と表記) を指定した例を示している．$NameNode_1$ は $NameSpace_1$ と $NameSpace_2$ を保存し，$NameNode_2$ は $NameSpace_2$ と $NameSpace_3$ を保存すると仮定する．このように指定した場合，各 NameSpace は複数の NameNode によって管理されることになる．しかし，複数の NameNode が同じ NameSpace を管理している場合，各自必要なタイミングで NameSpace を修正すると，同一 NameSpace の内容が異なる問題が生じ，NameSpace の一貫性が保たれない問題がある．それで，本研究では，クライアントからのリクエストを処理する NameNode を指定している．例えば，$NameSpace_1$ に保存されているデータと関連するリクエストは $NameNode_1$ が処理するように指定する．ある $NameSpace_i$ のリクエストを担当する $NameNode_j$ を，$NameSpace_i$ の **PrimaryNameNode** と呼ぶ．ある NameSpace に対する Primary NameNode のみが，NameSpace を変更する権限を持つ．Primary NameNode ではない他の

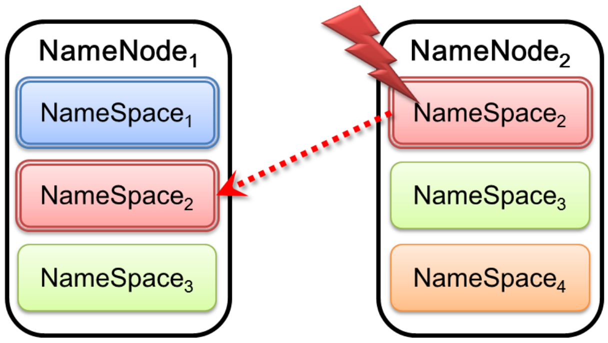


図 3　Primary NameNode の変更

NameNode は，分割 NameSpace を保存はするが，修正の権限は持たない．また，Primary NameNode が NameSpace を修正した場合，同じ NameSpace を管理する NameNode は NameSpace の情報を同期させる．このように複数の NameNode が同一の分割 NameSpace を管理することで，ある NameNode が故障などの不具合が生じた場合，他の NameNode が保存している同一 NameSpace 情報を利用することで，耐故障性を持たせることが可能となる．このような NameSpace 多重化を行うことで，Hadoop システムが持つ Master ノードの単一障害点の問題を解決できる．

図 3 は NameSpace の分割管理を NameNode の観点から表せている．図の通り，分割 $NameSpace_2$ は $NameNode_1$ と $NameNode_2$ の両方で多重化して管理されている．また，$NameSpace_1$ の Primary NN は $NameNode_1$，$NameSpace_2$ の Primary NN は $NameNode_2$ とする．この場合，$NameNode_2$ が故障すると $NameSpace_2$ を持っていた $NameNode_1$ が $NameSpace_2$ の Primary NN として動作することで問題を解決する．このような Failover 動作を行うことで，提案モデルは NameNode の耐故障性を保つ．また，**図 3** の Failover の後の状況を考えると，$NameNode_1$ は 2 つの NameSpace の Primary NameNode となっている．このような状況で $NameNode_1$ にアクセスが集中され，負荷が上がることが考えられる．Primary NameNode が変更できる特徴を利用することで，故障に応じた Failover だけではなく，システムが負荷分散のために Primary NameNode を変更することも可能となる．このような負荷分散のための動作を行うことで，本来単一計算機に集中されていた NameNode の負荷を状況に応じて動的に分散されることが可能となる．

4　まとめと今後の課題

本研究では，Hadoop システムの HDFS における単一 NameNode によって生じる問題を紹介し，その解決策として，NameNode の分散化を提案している．HDFS では NameNode が 1 台だけ存在するため，単一障害点による可用性の低下，メモリ容量の限界による NameSpace の制限，クライアントからのリクエストが 1 台に集中されるための負荷などの問題があげられる．このような問題を解決させるため，NameSpace の分散化を提案した．本来 1 台の NameNode か管理していたメタデータの集合である NameSpace を，幾つかに分割し，複数の NameNode がその分割された NameSpace を多重化して保持することで，高い可用性を提供し，NameSpace の制限問題を解決した．また，NameSpace の担当ノードを動的に変更することで，故障耐性とともに負荷分散を実現した．NameNode の分散化はまだいくつかの課題を残している．NameSpace の一貫性問題がその一つで，Primary NameNode が担当の分割 NameSpace に修正を加えた場合，他の同一 NameSpace を持つ NameNode と同期を行うが，多重化が 3 以上の場合，同期の途中に NameNode が故障する場合も考えられる．それにより，Failover によって選択された次の NameNode によって該当 NameSpace の内容がことなる場合が考えらる．このような問題は信頼性のあるブロードキャストなどで解決できるが，常にクライアントからのリクエストを処理する NameNode ではもっと効率的な解決方法が望まれる．今後は実際 NameNode の分散化を実現させるため，このような問題に実践的に取り込み，分散 NameNode の実装し，評価を行う．

参考文献

[1] Apache Hadoop Project, `http://hadoop.apache.org/`

[2] Tom White: Hadoop The Definitive Guide, O'Reilly (2009).

[3] Ghemawat S., Gobioff H. and Leung S.T.: The Google file system, ACM SIGOPS Operating Systems Review (2003).

[4] Dean J. et al.: MapReduce: Simplified data processing on large clusters, Communications of the ACM (2004).

[5] K. Shvachko et al.: The Hadoop Distributed File System, IEEE 26th Symposium on Mass Storage Systems and Technologies (2010).

[6] Big data, `http://en.wikipedia.org/wiki/Big_data`

[7] IDC, `http://www.idc.com/`

[8] Apache Hadoop 0.23 Release, `http://hadoop.apache.org/common/docs/current/index.html`

[9] Hadoop AvatarNode High Availability, `http://hadoopblog.blogspot.jp/2010/02/hadoop-namenode-high-availability.html`

[10] facebook, `http://www.facebook.com/`

[11] Konstantin S.: Warm HA NameNode going Hot,

Apache Hadoop Issues, HDFS-2064, (2011).

マルチコア CPU 環境における仮想計算機を用いた Hadoop システムの評価

石井 朝葉　金 鎔煥　中村 純哉　大下 福仁　角川 裕次　増澤 利光

大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻

概要

Hadoop とは, 大規模データの並列分散処理フレームワークである．Hadoop システムは，大規模データを効率的かつ安全に保持するだけではなく，複数の計算機が効率的に並列処理を行うことで，膨大なデータの高速処理を実現している．近年, 仮想化技術が注目を浴びており，Hadoop システムが仮想計算機で構成される場合が想定される．仮想計算機による Hadoop システムの運用は，拡張性，耐故障性，運用コストの面で利点を持つ．その一方で，仮想計算機を管理するための仮想マシンモニタがリソースを必要とするため，計算機本来のリソースを仮想計算機が使いきることはできず，性能低下が懸念される．本報告では，仮想計算機から構成される Hadoop システムと実際の物理マシンから構成される Hadoop システムとの比較実験を行い，性能の評価・分析を行った．比較実験の結果として，仮想計算機を用いた Hadoop システムは，小さいファイルを大量に扱う場合に有利であること，仮想計算機のリソースで扱える程度の負荷の処理では物理マシンとの処理性能の差が少ないことを確認した.

1　はじめに

仮想化とは，CPU やメモリ等のハードウェア内のリソースを，物理的構成にとらわれず統合，分割する技術のことである．近年，この仮想化技術に対する注目が高まっており，仮想化技術に関する IDC の発表 [9] によると，2014 年までにサーバの 70 %が仮想化環境で運営されると予想されている．仮想化技術は，計算機のリソースを論理的に分割させて複数の計算機として利用することができるため，構成に必要な計算機の数より少ない計算機でクラスタを構成することが可能となる．この特徴は，実際の計算機を購入する費用を節約するだけではなく，計算機を設置する場所のスペースコスト，また計算機運用の電力コストの削減も可能にするため，仮想化技術を取り入れる企業が急増している．更に，仮想化を行うことで各計算機のバックアップ，移動などが容易になり，かつ計算機のリソースを動的に変更させることも可能となる.

また，近年クラウドコンピューティングの浸透が進み，Web コンテンツや IT システムにより生成されるデータ量が急増している．情報社会においてあらゆる形で生成される膨大な量のデータは，既存のデータ処理方法では，保存及び管理が非常に困難である．このように，既存の方法での処理が難しいほどの膨大なデータは BigData[10] と呼ばれ，近年の研究課題として注目を浴びている．Google では BigData の処理に対応するため，大量のデータを格納できる分散ファイルシステム [2] と分散処理を自動化するフレームワーク [3] を独自開発した．この Google のファイルシステムとフレームワークの概念を基に開発されたのが, 並列分散処理フレームワーク **Hadoop**[1] である.

Hadoop システムの特徴としては，コモディティなマシンで高性能の分散システムを構成可能であること, 台数を増やすことで線形的に性能向上すること, データの複製により高い耐故障性を持つことがあげられる. よって仮想化技術を用いると，少数の物理計算機上で大規模な Hadoop クラスタが構成可能であり，コスト削減やマシン管理の容易さなど仮想化の利点を享受することができる.

しかし，仮想計算機 (VM) を稼働させるためには仮想マシンモニタ (VMM) が介在するため，VMM によるオーバーヘッド [11, 12, 13] の発生は避けられない．このオーバーヘッドの発生により，VM で構築された Hadoop クラスタの性能低下が考えられる．VMware の発表 [5] では，VMware を用いて複数の CPU コアと対応づけられた VM を作成し，その VM で構築した Hadoop システムの性能評価を行なっている．その結果，1 台の物理マシン上に 1 台の VM を構築した場合は平均 4% の性能低下，1 台の物理マシン上に複数の VM を構築した場合は最大 14% の性能向上が見られた．しかし、VMware の実験で用いられた計算機は，複数の高性能プロセッサと膨大なメモリを搭載していて、仮想化技術のためのリソース消費の割合が非常

に少ない環境である．また性能評価の際には，タスク数が少なく，ファイルサイズが大きい処理を実行していた．本報告では，よりコモディティなマシンで，ジョブあたりのファイルサイズ，タスク数を変化させた場合の実験を試みる．仮想化ソフトウェア Xen[7] を用いて 1 台の VM が 1 コアを持つ Hadoop システムを構築し，物理マシンのみで構成された Hadoop システムとの比較実験を行った．その結果，ファイルサイズが小さい場合，最大 25% の性能向上が見られた．

本報告の構成は以下のとおりである．まず，2 章において Hadoop について説明する．次に 3 章で評価実験環境について説明する．4 章では, 評価実験結果について述べる．最後に 5 章で本報告の結果をまとめる．

2　Hadoop の概要

Hadoop とは，大規模データを効率的に処理するためのオープンソースの並列分散処理基盤である．複数の計算機で並列的に処理を行い，結果を集約することで高速処理を実現している．Apache[6] のプロジェクトとして開発が進められており, 分散コンピューティングに関連するサブプロジェクトの集合体である．主な構成要素は，Hadoop Distributed File System(以下 HDFS)[4] と，MapReduce[3] である．

Hadoop システムは，Master・Worker 型のシステム構成であり，システム全体の管理は 1 台の Master ノード，実際の並列処理は複数の Worker ノードが担っている．

2.1　HDFS

HDFS とは，大規模データを効率良く管理・処理するために設計された論理的な分散ファイルシステムである．巨大なファイルを複数の計算機に分割して保持することにより，複数の計算機のストレージを 1 つの巨大なストレージとして扱うことを可能にしている．HDFS では Master ノードの役割を果たすものを NameNode，Worker ノードの役割を果たすものを DataNode と呼ぶ．NameNode はファイルシステム全体の管理を行い，DataNode は実際のデータの格納先となる．HDFS 内では，ファイルは固定長のブロックに分割され管理されており，ブロックを複数の DataNode に複製配置することにより耐故障性を高めている．

2.2　MapReduce

MapReduce とは，大規模なシステムにおいて，膨大なデータを高速で並列分散処理するアプリケーションを作成するためのプログラミングモデルおよびソフトウェアフレームワークである．MapReduce では Master ノードの役割を果たすものを JobTracker，Worker ノードの役割を果たすものを TaskTracker と呼ぶ．JobTracker は，分散処理の指示，管理を行い，TaskTracker は実際に処理を行う．MapReduce による分散処理では，まず JobTracker がクライアントからジョブと呼ばれる分散処理要求を受け取る．JobTracker はクライアントから受け取ったジョブをタスクと呼ばれる小さい単位の処理に分割させて，複数の TaskTracker にタスクの処理を要求する．以降，JobTracker はクライアントからのジョブが全て完了されるまで，ジョブやタスクの処理の進捗，タスクの割り当て，TaskTracker の死活などの状況を監視し，管理する．計算処理は Map フェーズと Reduce フェーズに分かれている．Map フェーズでは，map タスクを割り当てられたノードが HDFS に格納された入力データに対して処理を行い，必要なデータを抽出する．抽出されたデータは Reduce フェーズを実行するノードへ転送される．Reduce フェーズでは Map フェーズで抽出された情報の集約を行い，処理結果を得る．

3　評価実験環境

本実験では, 21 台の計算機を用いた．1 台は Master ノード，残りの 20 台は Worker ノードの役割を担っている．この 21 台の物理マシンから構成される Hadoop システムと，20 台の物理マシン上に構築した 40 台の VM を Worker ノードとして持つ Hadoop システムの比較実験を行った．また，各物理マシンの計算機環境を表 1 に示す．各マシンはそれぞれ 1Gigabit Ethernet で接続されている．

表 1　計算機環境

	Master ノード	Worker ノード
CPU	Intel Core i3(3.07GHz) 2 Cores(HT 対応)	Intel Core i3(3.10GHz) 2 Cores(HT 対応)
RAM	2GB(DDR3)	4GB(DDR3)
HDD	500GB(S-ATA Ⅱ)	500GB(S-ATA Ⅱ)
OS	CentOS 5.7 (Linux Kernel 2.6)	CentOS 5.7 (Linux Kernel 2.6)
台数	1 台	20 台

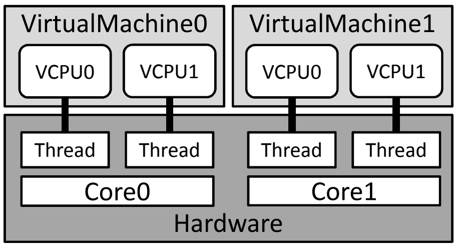


図 1　仮想計算機の CPU 設定

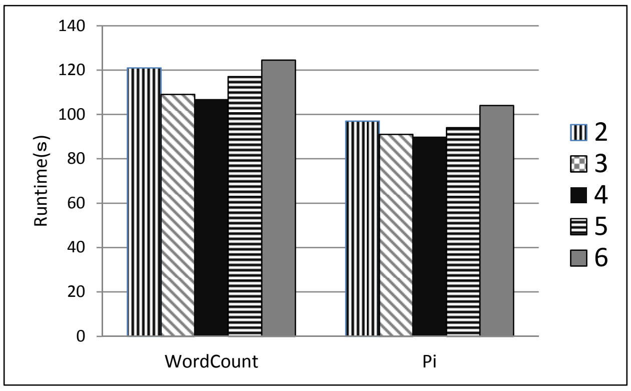


図 2　物理クラスタ実行結果

3.1　仮想化環境

仮想化は，仮想化ソフトウェア Xen[7] を用いて準仮想化方式で行った．使用した Xen のバージョンは Xen3.1.2 である．各 Worker ノード用計算機に対して，計算機のコア数と同数になるよう 2 台ずつ VM を設置した．本実験に用いる計算機は Hyper-Threading 技術 [14] に対応しており，1 つの物理 CPU コアを仮想的に 2 つの CPU コアのように扱える．よって図 1 のように VM の仮想 CPU 数を 2 つにし，それぞれを同一コアの異なるスレッドに対応づけた．VM の計算機環境は表 2 に示す．

表 2　仮想計算機環境

RAM	1.5GB
HDD	50GB
OS	CentOS 5.7 (Linux Kernel 2.6)

3.2　Hadoop 環境

実験に用いた Hadoop のバージョンは 1.0.3, Java のバージョンは 1.7.0_ 01 である．Hadoop システムは耐故障性向上のためデータを異なるマシンに複製配置する．本実験では複製数を 2 とした．しかし仮想化 Hadoop システムでは，同じ物理マシン上の異なる VM にデータが複製配置される可能性がある．その場合，物理マシンの故障により元のデータと複製データが共に失われてしまう．Hadoop では，ラック設定 [8] を行うと 2 つ目の複製は別のラックに配置されるようになるため，仮想化 Hadoop システムでは，同一物理マシン上の VM2 台を 1 つのラックに設定することにより，データの複製が同一物理マシン上に配置されないようにした．また本実験では，HDFS にデータを保存する際のブロックサイズはデフォルト値の 64MB である．Hadoop で管理者は MapReduce 処理において，各 Worker ノードに同時に割り当てるタスクの数を設定することが可能である．割り当てるタスクの数はデフォルトでは 2 に設定されているが，map 数を変化させた予備実験の結果 (図 2) より物理マシン 1 台あたり 4 つの map タスクを割り当てることにする．よって，単一コアで構成された仮想化クラスタの実験環境では，ノードあたりの map タスク数を 2 つにした．これにより，物理クラスタ，仮想化クラスタ共に，同時に 80 個の map タスクを受け入れることが可能となる．

4　評価実験

4.1　Hadoop ベンチマーク

評価実験では以下の 3 つのベンチマークを用いた．

1. Pi
   モンテカルロ法によって円周率を求めるプログラムである．計算処理が多く，入出力処理はほとんど無い．計算サンプル数と map タスク数の 2 つの引数をもつ．Map フェーズで平面にサンプル数の数だけランダムにプロットした点のうち円の中に入る数を数え，円周率を求める．Reduce フェーズでは，各 Map フェーズの出力値の平均を算出する．
2. RandomTextWriter
   HDFS にランダムなテキストデータを書き込むプログラムである．RandomTextWriter は，(1) の Pi とは異なり，計算処理は殆ど行わず大量の書き込み作業を行う．各ノードに書き込まれるデータ量は等しい．Map フェーズでは，データを生成しシーケンシャルファイルとして HDFS に書き込む．Reduce フェーズは存在しない．
3. WordCount
   入力ファイル中の各単語の出現頻度を数えるプログラム．Map フェーズでは，HDFS に格納されてい

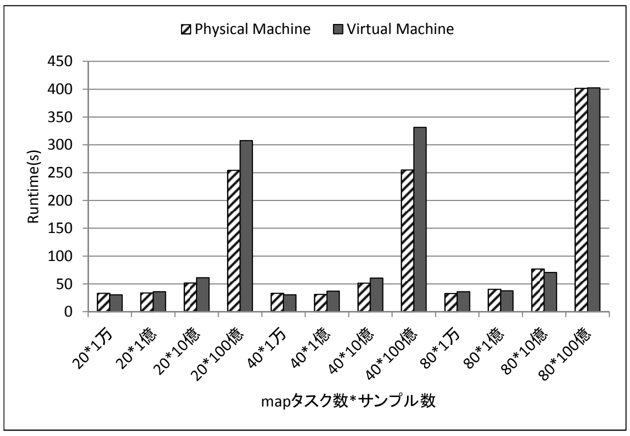


図 3　Pi の実行結果 (1)

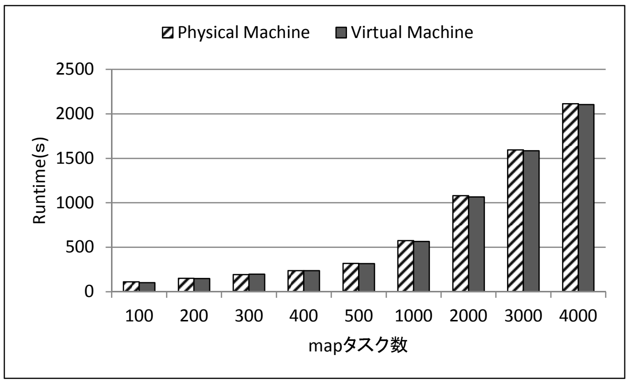


図 4　Pi 実行結果 (2)

る入力ファイルから指定ブロックサイズごとにファイル中に出現する単語を抽出し，Reduce 処理を行うノードに送信する．Reduce フェーズでは，Map フェーズから送信されたファイルを基に単語ごとの出現回数をカウントし結果を HDFS に格納する．WordCount では，少しの計算処理と大量の読み出し作業が行われる．

4.2 実験結果

1. Pi

サンプル数,map タスク数を変化させて実験を行った．サンプル数を増加させることにより，1 タスクあたりの計算処理による負荷が増加する．map タスク数を増加させることにより，タスクの切り替えが頻発し，ネットワーク転送量が増加する．

図 3 は，map タスク数 20，40，80 に対してサンプル数を変化させて Pi を実行した際の結果である．map タスク数がコア数と同数の 40 以下の場合，サンプル数の大きな実行では，物理マシンの方が処理時間が短い．map タスク数が 20 の場合，全ての map タスクは異なる計算機で実行される．物理マシンによるクラスタでは計算機が 20 台のため，全ての map タスクが別の物理マシン上で実行されることになる．一方 VM によるクラスタでは，VM が 40 台のため，同一物理マシン上の VM2 台に map タスクが割り当てられる場合，map タスクを 1 つも割り当てられない場合が起こりうる．このように VM によるクラスタでは，クラスタ内の計算機資源を全て利用出来ないため，処理性能の低下につながったと考えられる．また map タスクが 40 の場合，1 つのコアが 1 つのタスクを処理することになり，マシン性能が高い物理マシンの方が有利であるため，性能差が生じたと考えられる．しかし map タスク数が 80 の場合，サンプル数によらず処理時間の差がほぼ無くなっている．

図 4 は，サンプル数を 10 億に固定し，map タスク数を変化させた場合の結果である．map タスク数によらず処理時間の差は殆ど見られない．これにより，1 つのノードに対し演算処理多いタスクを複数割り当てるような実行では，仮想化によるオーバーヘッドがほぼ無いと考えられる．

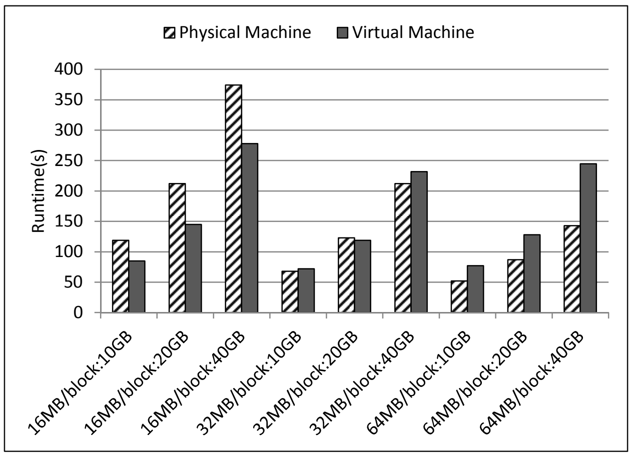


図 5　RandomTextWriter の実行結果

2. RandomTextWriter

RandomTextWriter では 1 つの Map で指定サイズのシーケンシャルファイルを書き込む．よって，あるサイズのデータを書き込む場合，シーケンシャルファイルあたりのサイズが小さいほど, プログラム全体で実行される Map タスク数は増加する．各 Map タスクでシーケンシャルファイルを生成し，HDFS に書き込んでおり，入力処理が多いプログラムである．

図 5 は，16MB，32MB，64MB のサイズのシーケンシャルファイルを合計入力ファイルサイズが，10GB,20GB,40GB となるように書き込んだ場合の実行結果である．ファイルサイズがブロックサイズと等しい 64MB の場合，物理マシンの方が処理時間が短く，ファイルサイズがブロックサイズに比べ小さくなるにつれて，VM の方が処理時間が短くなっている．

図 6，図 7，図 8 は，64MB，32MB，16MB のサイズのシーケンシャルファイルを 20GB 書き込んだ際の，Worker ノードである計算機の 1 秒あたりの転送数 (デバイスに対する IO リクエスト数) の合計値である．緑の線が物理マシン，青の線が仮想化されている計算機の実際の転送量を示している．このグラフより，ファイルサイズが小さい場合，物理マシンでは次のファイル書き込みまで idle 状態が存在し，ファイル数が多いため idle 状態が頻発している．一方，VM では 2 台の VM が休みなく書き込むことで idle 状態がほぼ発生していない．またファイルサイズが大きい場合，物理マシンはブロック数が少ないため idle 状態になる頻度が少なく，かつファイルサイズが大きく連続したディスク領域に書き込み可能なため最大転送速度が速くなっていることがわかる．以上より，ファイルサイズが小さい場合は VM が，ファイルサイズが大きい場合は物理マシンが優れていると考えられる．Hadoop システムでは，一般的なファイルシステムと比べデフォルトで 64MB と，非常に大きなブロックサイズを採用しており，大きなファイルの処理に適した設計がされている．よって小さなファイルを扱う場合，処理性能が低下する問題 [15] がある．この問題に対する対策として，小さなファイルを統合して 1 つのファイルとして扱う方法などがあり，本実験が示す VM の利用も有用な方法のひとつであると考えられる．

3. WordCount

WordCount はデータの読み出し，処理，データの格納というワークフローを持っており，データの入出力が頻繁に発生するプログラムである．　図 9 は，RandomTextWriter の出力に対し，Reduce タスクを 1 つにして WordCount を行った際の実行結果である．Reduce タスクが 1 つのため，1 台のマシンで全ての Map タスクの処理結果を集約し，処理を行っている．図 9 の結果より，VM に比べ物理マシンの方が処理時間が短くなっている．40GB の入力ファイルを処理する際の Map フェーズの実行時間，Reduce フェーズの実行時間を図 10 に示す．図 10 より，VM と物理マシンでは，Map フェーズよりも Reduce フェーズの処理時間に差が出ていることがわかる．Reduce タスクが 1 つの場合，1 台のマシンで全ての Map タスクの処理結果を集約し，処理を行うため，1 台あたりのマシン性能の高い物理マシンの方が実行時間が短くなったと考えられる．そこで Reduce タスクを 2 つに増やし，Reduce タスク 1 つあたりの処理負荷を減らして再度同じデータに対して WordCount を実行した．その結果を図 11 に示す．結果として，VM と物理マシン共に処理時間が

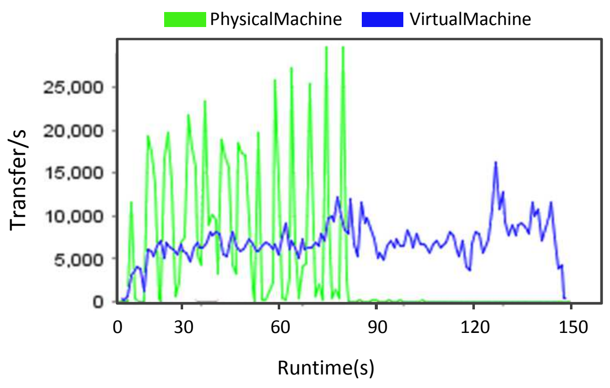


図 6　RandomTextWriter:64MB/block:20GB の実行結果

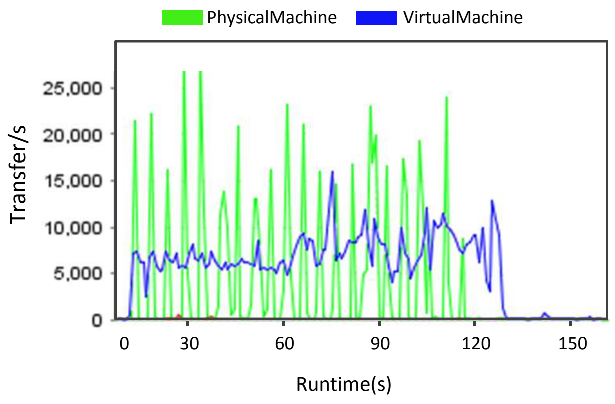


図 7　RandomTextWriter:32MB/block:20GB の実行結果

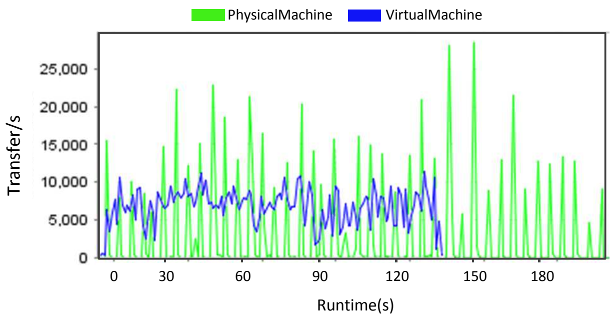


図 8　RandomTextWriter:16MB/block:20GB の実行結果

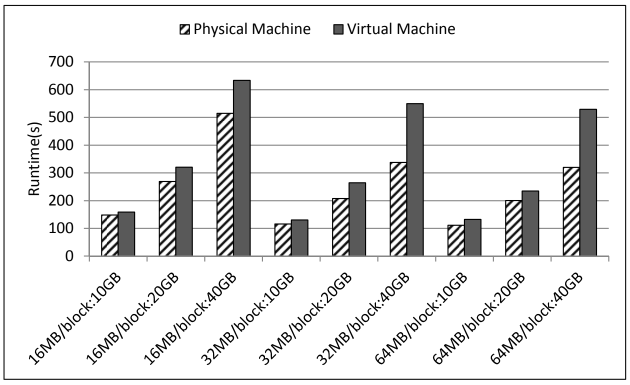


図 9　WordCount の実行結果:Reduce1

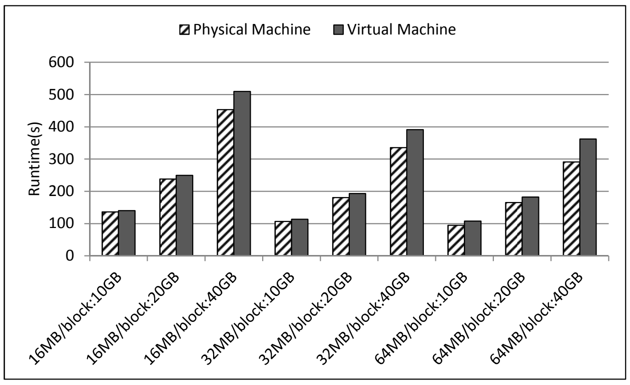


図 11　WordCount の実行結果:Reduce2

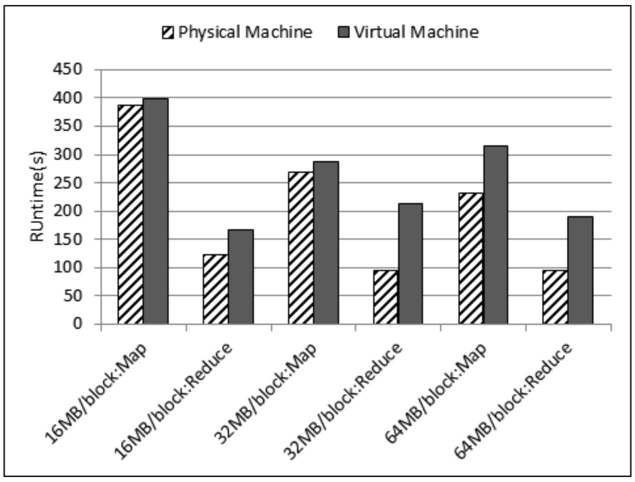


図 10　入力ファイルサイズ 40GB Map フェーズ，Reduce フェーズの実行時間:Reduce1

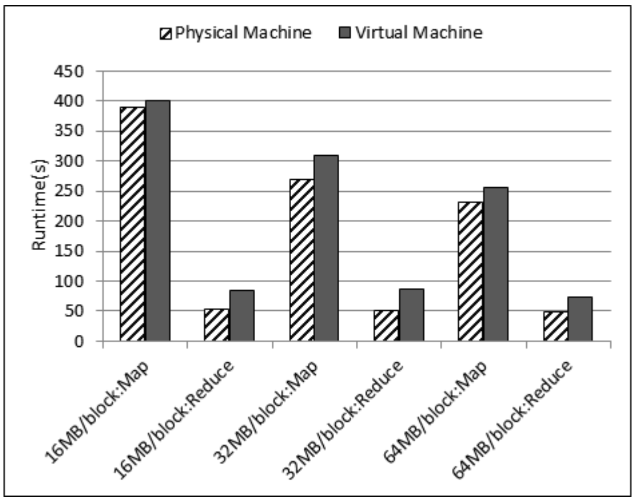


図 12　入力ファイルサイズ 40GB Map フェーズ，Reduce フェーズの実行時間:Reduce2

短くなり，VM と物理マシンの差が小さくなっている．この時の 40GB の入力ファイルを処理する際の Map フェーズの実行時間，Reduce フェーズの実行時間を図 12 に示す．Reduce タスクが 1 つの場合と比べて，VM と物理マシンでの Reduce フェーズの実行時間の差が短くなっている．これより，タスクあたりの負荷を軽減することで，VM での性能低下を防ぐことができることがわかる．

5　まとめ

本報告では，マルチコア CPU を持つ物理マシン上にコア数と同数の VM を作成し，その VM を用いて Hadoop システムを構築した．仮想化 Hadoop システムと，物理マシンから成る Hadoop システムの比較実験を行った結果，VM は小さいファイルの頻繁なアクセスには有利であること，VM の負荷を上回らない限り，物理マシンと比べてオーバーヘッドは少ないことがわかった．

参考文献

[1] Apache Hadoop Project,
`http://hadoop.apache.org/`

[2] Ghemawat S., Gobioff H. and Leung S.T.: The Google file system, ACM SIGOPS Operating Systems Review (2003).

[3] Dean J. et al.: MapReduce: Simplified data processing on large clusters, Communications of the ACM (2004).

[4] K. Shvachko et al.: The Hadoop Distributed File System, IEEE 26th Symposium on Mass Storage Systems and Technologies (2010).

[5] VMware.: A Benchmarking Case Study of Virtualized Hadoop Performance on VMware vSphere 5 (online),
`http://www.vmware.com/files/pdf/techpaper/`

VMW-Hadoop-Performance-vSphere5.pdf (2011).

[6] The Apache Software Foundation.: http://www.apache.org/

[7] Barham P., Dragovic B., Fraser K., Hand S., Harris T., Ho A., Neugebauer R., Pratt I. and Warfield A.: Xen and the art of virtualization, ACM SIGOPS Operating Systems Review (2003).

[8] Tom White: Hadoop The Definitive Guide, O'Reilly (2009).

[9] IDC: Worldwide Market for Enterprise Server Virtualization to Reach $19.3 Billion by 2014, http://www.idc.com/about/viewpressrelease.jsp?containerId=prUS22605110 (2011).

[10] Big data, http://en.wikipedia.org/wiki/Big_data

[11] Cherkasova L. and Gardner R.: Measuring CPU overhead for I/O processing in the Xen virtual machine monitor,Proceedings of the annual conference on USENIX Annual Technical Conference (2005).

[12] Menon A., Santos J.R., Turner Y., Janakiraman G.J. and Zwaenepoel W.: Diagnosing performance overheads in the Xen virtual machine environment, Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (2005).

[13] Oracle Corporation.: Oracle VM, Performance Evaluation of Oracle VM Server Virtualization Software(online), http://www.oracle.com/us/026997.pdf (2008).

[14] Intel Hyper-Threading Technology, http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[15] Tom White: The Small Files Problem, cloudera, http://www.cloudera.com/blog/2009/02/the-small-files-problem/ (2009).

MapReduce 上での最小全域木アルゴリズムに対する入力データ分割に関する考察

上野健次郎 † 和田幸一 †† 泉泰介 † † †
† 名古屋工業大学大学院創成シミュレーション工学専攻
†† 法政大学大学院工学研究科情報電子工学専攻
† † † 名古屋工業大学大学院情報工学専攻

1　はじめに

近年，インターネットの普及や記憶装置の大容量化に伴い，多くの情報システムにおいて蓄積・処理されるデータ量が爆発的に増大している．例えば，Facebook 社では一日当たり 80TB のデータ処理をおこなっていると言われている．このような大規模データ処理において，並列化による高速化はほぼ不可避な状況となっているが，特に近年大規模並列計算のプラットフォームとして MapReduce が注目を集めている．MapReduce は google 社により開発された，大量のデータを大規模な並列計算機環境で処理するためのプログラミングモデルおよびその実装であり，ユーザは Map および Reduce というふたつの関数を記述することで，並列化処理を簡便に記述できるという特徴がある．実際に Google 社では Web 検索のためのインデックス作成などに使用している．さらに，MapReduce のオープンソース実装である Hadoop のリリースにより，多くの商用システムにおいて大規模並列処理用プラットフォームとしての地位を急速に確立しつつある．

MapReduce を用いたプログラミングにおいては，Map 処理および Reduce 処理において計算機間の大規模なデータ交換を必要とし，これは MapReduce を用いたプログラムの実行において大きな影響を及ぼす．そのため，MapReduce 上の Map および Reduce の回数を少なく抑えることが効率的なアルゴリズム設計の上では重要になる．

本研究では大規模グラフに対する最小全域木問題を解くための MapReduce 上のアルゴリズムについて検討する．

文献 [2] においては，各計算機がメモリ領域 $O(n^{1-\epsilon})$ ビットを使える状況において，$O(n^{\epsilon})$ 個の計算機を用いた最小全域木構成アルゴリズムを提案している．同アルゴリズムは漸近的には $O(1)$ 回の MapReduce 処理で MST を出力することが示されている．

本研究では，このアルゴリズムのさらなる高速化を目指す．同アルゴリズムでは計算のために入力グラフの辺集合を各計算機に分配する必要がある．文献 [2] ではハッシュ関数を用いたランダム分割により分配をおこなっているが，本研究では分配方法を工夫することで，さらなる高速化を狙う．我々の提案手法を実験した結果，ランダム分割に比べて MapReduce 回数が少なく抑えられることが確認された．

2　MapReduce

MapReduce は，Google 社によって開発された分散処理のためのプログラミングモデルである．MapReduce では，データ処理のプロセスは Map，Shuffle，Reduce の 3 つのフェーズに分解される．ひとつのジョブをタスクと呼ばれる複数の小さな処理に分割し，各計算機に割り当て，各計算機は割り当てられたタスクを実

行し応答を返す．図 1 に処理の流れを示す．

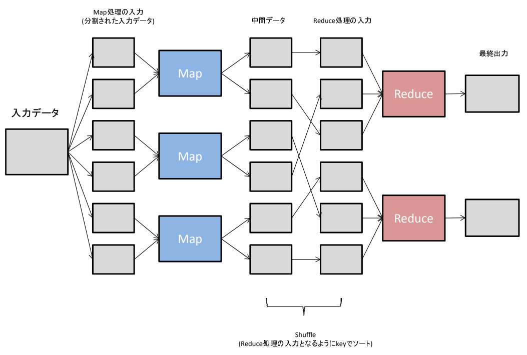


図 1　MapReduce 処理の流れ

Map フェーズ

Map フェーズでは，入力データを分割しそれぞれを Map タスクに割り当てる．各 Map タスクは分割された入力データからひとつの < key,value> ペアを取り出し，ユーザが記述した Map 処理をおこなう．処理されることで生成された中間データもまた < key,value> ペアの形で格納される．

Shuffle フェーズ

Shuffle フェーズでは, 中間データをソートし，同じ key を持つデータが 1 か所にまとまるように value のリスト (values) を生成し、Reduce フェーズに < key,values> ペアを渡す．この Shuffle フェーズの中で大量のデータのネットワーク転送が発生するため、処理全体のボトルネックとなることがあり得る．

Reduce フェーズ

Reduce フェーズでは，Shuffle フェーズで生成された < key,values> ペアにユーザが記述した Reduce 処理をおこなう．Reduce 処理の結果として，新たに < key,value> ペアを生成し出力する．

2.1　MapReduce 計算の理論モデル

前述のとおり，MapReduce を用いた計算において焦点となるのは Map および Reduce 処理の回数である．しかしながら，MapReduce の処理回数を評価尺度とした場合，何も制約を課さない計算モデルにおいては単一計算機において全処理をおこなうという自明な最適解を持つこととなる．そこで，MapReduce 計算の理論モデルにおいては，通常単一計算機の利用可能なメモリ量が $o(N)$ ビットである，すなわち問題全体を単一計算機にロードすることが不可能であるという仮定を置く [1]．本研究では既存のアルゴリズムの仮定に従い，$O(N^{1-\epsilon})$ ビットのメモリを利用可能な計算機が $O(N^{\epsilon})$ 台利用可能であるとする．ここで N は入力のグラフ $G = (V, E)$ の複雑度 (すなわち $N = max|V|, |E|$) である．また $N^{1-\epsilon} > |V|$ を仮定する．

3　定義

まず，はじめにいくつかの定義をおこなう．

定義 3.1（最小全域木（MST））

重み付き無向グラフ $G = (V, E, w)$ に対して，頂点数 $|V|$ で閉路を持たない部分グラフのことを G の全域木と呼ぶ．全域木のコストとは，全域木を構成する辺のコストの総和である．G の全域木でコストが最小となるものを最小全域木 (MST) と呼ぶ．

4　MST アルゴリズム

アルゴリズムはサイズ η となるように入力グラフを辺集合の分割によって複数の部分グラフに分ける．それぞれの部分グラフの MST を求めることにより不要な辺を削除する．不要な辺を削除されたグラフのサイズが η 以下であるならば一台の計算機でそのグラフの MST を求めアルゴリズムを終了する．そうでなければ再帰的に処理をおこなう．図 2 にアルゴリズムを示す．

このアルゴリズムは $\omega(n)$ 本の辺を持つグラフに対して，1 回の mapreduce 処理で辺の本数を $\Omega(n^{\epsilon})$ 本だけ削減する．初期状態での辺の本数は明らかに高々 $O(n^2)$ 本なので，このアルゴリズムは定数回の mapreduce 処理を実

```
1. if $|E| < \eta$ then
2. Compute $T^* = MST(E)$
3. return $T^*$
4. end if
5. $l \leftarrow \Theta(|E|/\eta)$
6. Partition $E$ into $E_1, E_2, ..., E_l$
   where $|E_i| < \eta$ using a universal
   hash function $h : E \rightarrow \{1, 2, ..., l\}$.
7. In parallel: Compute $T_i$, the mini-
   mum spanning tree on $G(V, E_i)$.
8. return $MST(V, \bigcup_i T_i)$
```

図 2　論文 [2] の MST アルゴリズム

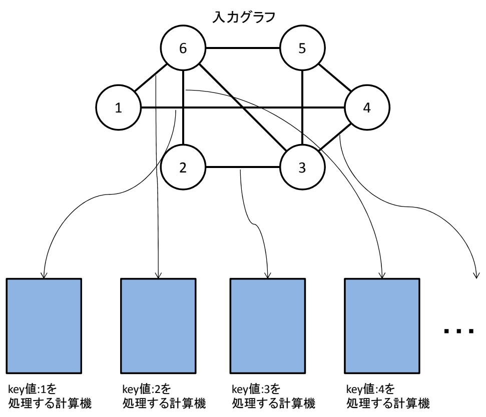


図 3　ランダム分割の例

行して停止する．

補題 4.1　図 2 のアルゴリズムは $O(1)$ 回の mapReduce 処理で終了し，正しく MST を求める [2].

5　研究内容

5.1　MST アルゴリズムの実装

図 2 で示した MST アルゴリズムを実装していく．具体的な処理は以下のようになる．

入力グラフの辺の数を数え，一台の計算機に格納できるかを判断する．格納できるならば，一台のみの MapReduce 処理をおこない入力グラフの MST を求めプログラムを終了する．格納できないならば，入力グラフを Map フェーズにて分割し，いくつかの部分グラフを出力する．Reduce フェーズではその部分グラフを入力として受け取り，MST を求める．求めた部分グラフの MST を集め，次の入力として出力する．

5.2　入力グラフの分割手法

5.2.1　既存の分割 (ランダム分割)

各計算機に送信される辺の数が均等になるように分割をおこなう [2].

5.2.2　ヒストグラムを利用した分割 (提案手法)

ユーザが指定した本数分コストの小さいものから順に選んでいき，選ばれた辺の集合をすべての計算機に送信する．まず，MapReduce 処理により入力グラフの辺のコストに対するヒストグラムを作成する (図 4)．そして，作成したヒストグラムを利用し，コストの小さいものから O(n) 本の辺を選択する．この選択された辺についてはすべての計算機に送信される．残りの辺については，既存手法と同様にランダム分割により分配する (図 5)．この手法では一部の辺集合は重複して計算機に送信されるため同一の計算機台数であれば各計算機が必要とするメモリの量は若干増加する（ただし漸近的な差はない)．また，初めにヒストグラムを作成する必要があるため MapReduce 処理回数が 1 増える．しかしながら，重みの小さい辺は最終的にもとめる MST に現れやすい傾向があるため，上記の分割をおこなうことで，複数の計算機の計算結果として返される辺集合の交わりが大きくなりやすい（すなわち，計算結果の和集合が小さくなりやすい)．よって 1 回の MapReduce 処理による辺の枝刈り効果が大きくなることが期待される．

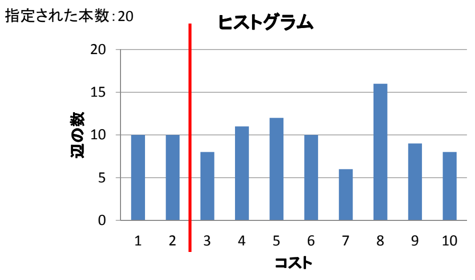


図 4　提案手法のヒストグラムの例

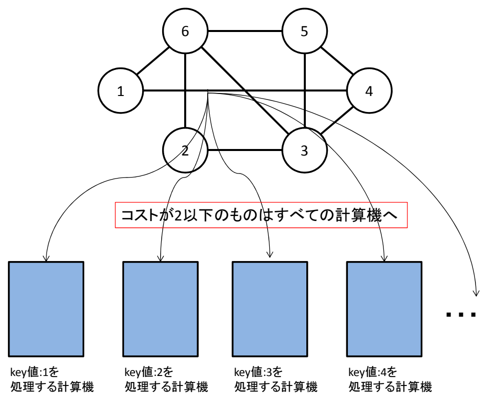


図 5　提案手法の例

6　実験および考察

一台のローカル環境にてふたつの実験をおこなった．実験においては，MST を計算するアルゴリズムとしてはクラスカルのアルゴリズムを利用し，また，仮想的に計算機のメモリ制約を導入している．
台数の想定：$O(|E|/\eta)$
実験 1 のメモリの想定：各頂点数の二倍
実験 2 のメモリ想定：各頂点数の四倍
次のグラフに対し実験をおこない, MapReduce 処理の回数の比較をおこなった．グラフのコストはランダムで生成される．

1.　完全グラフ．
$|V| = 3,000, |E| = 4,498,500$

実験 1 はランダム分割と提案手法の比較をおこなう．表 1 に完全グラフを入力した際のそれぞれの MapReduce 処理の回数を示す．MapReduce 処理の回数が 1+x となっているのはヒストグラム作成の一回と MST アルゴリズムの回数をわかりやすくするためである．それぞれの結果からまず MapReduce 処理の回数が変化することがわかった．
提案手法ではすべての計算機がコストの小さいものを持つ．また，MST はコストの小さい辺で構成されている可能性が高い．つまり，コストの小さい辺を持つことで MST を構成している辺を持ちやすくなる．よって，一度の MapReduce 処理でより多くの辺を削除することができる．

実験 2 は提案手法の送信する辺の本数の変化による違いを比較する．表 2 に完全グラフを入力した際のそれぞれの MapReduce 処理の回数を示す．今回の実験結果ではわかりづらいが 3000 本から 4500 本に変化させたときの結果から送信する辺を増やすことで MapReduce 処理の回数を減らすことができることがわかる．
送信する辺の本数を増やすことで計算機はより多くの MST を構成している辺を持つことになる．よって，一回の MapReduce 処理でより多くの辺を削除することができる．

このアルゴリズムでは各計算機で求められた MST のすべてが入力グラフの MST またはその一部となれば一回の MapReduce 処理で終了する．そのため，入力グラフの MST を意識した分割をおこなえば，処理回数を減らすことができる．

	ランダム分割	提案手法
MapReduce 回数	11	1+8

表 1　実験 1 の結果

7　まとめと今後の課題

MST アルゴリズムの入力データの分割手法についてふたつの手法を実装し，同じ入力データであってもグラフ構造を意識した分割をおこなうことで MapReduce 処理の回数を減らすこ

送信する辺の本数	MapReduce 回数
3000	1+5
4500	1+3
6000	1+3
7500	1+3

表 2　実験 2 の結果

とが可能であることを示した．

今回の研究では MapReduce の処理回数のみに注目しているため実行時間での評価をおこなっていない．今後の課題としては実際の環境での実行時間の比較が必要である．また，ヒストグラムの作成を一度だけでなく，毎回おこなうなどの複数回おこなった場合の処理回数の変化も実験すべきである．また，グラフの構造を意識した分割をおこなうことで処理回数を減らすことができることを示したが，MapReduce の都合上グラフ全体の構造を知ることが難しい．したがって，MapReduce 処理中にグラフの構造を得る高速な手法を考える必要がある．

参考文献

[1] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’10, pp. 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[2] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA ’11, pp. 85–94, New York, NY, USA, 2011. ACM.

[3] 山下真一 猿田浩輔 藤井達朗太田一樹. Hadoop　徹底入門. 翔泳社, 2011.

Fast Hough Transform Using DSP blocks and block RAMs on the FPGA

Xin Zhou, Yasuaki Ito, and Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—Since FPGA chips maintain relatively low price and its programmable features, it is widely used in those fields which need to update architecture or functions frequently such as communication and education areas. Especially, in mobile devices that recently require the ability to perform computation such as real-time image processing, FPGAs are promising devices. The main contribution of this paper is to present a new FPGA architecture for the Hough transform that identifies straight lines in a binary image. Recent FPGAs have hundreds of embedded DSP blocks and block RAMs. For example, Xilinx Virtex-6 Family FPGAs have a DSP48E1 block, which is a configurable logic block equipped with fast multipliers, adders, pipeline registers, and so on. They also have a dual-port memory with 18Kbits as a block RAM. One of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP blocks and block RAMs. Our new architecture for the Hough transform uses 178 DSP48E1 blocks and 180 block RAMs with 18Kbits that work in parallel. As far as we know, there is no previously published work that fully utilizes DSP blocks and block RAMs for the Hough transform. Roughly speaking, a conventional sequential implementation performs $180m$ voting operations for m edge points. Our architecture performs voting operations in parallel, and outputs identified straight lines in $m+97$ clock cycles. Since $180m$ voting operations are performed using 178 DSP48E1 blocks, the lower bound of the computing time is m clock cycles. Hence our implementation is close to optimal. The implementation results show that the Hough transform for a 512×512 image with 33232 edge points can be done in only 135.75μs.**

***Keywords*-Image processing, Line detection, Hough transform, FPGA, Embedded DSP blocks, Embedded block RAMs**

I. INTRODUCTION

A Field Programmable Array (FPGA) is a programmable logic device designed to be configured by the customer or designer by hardware description language after manufacturing. The most common FPGA architecture consists of an array of logic blocks, I/O pads, block RAMs and routing channels. Furthermore, recent FPGAs have embedded DSP blocks that make a higher performance and a broader application.

The Xilinx Virtex-6 series FPGAs have DSP48E1 blocks that are equipped with a multiplier, adders, logic operators, etc [1]. More specifically, the DSP48E1 block has a two-input multiplier followed by multiplexers and a three input adder/subtractor/accumulator. The DSP48E1 multiplier can perform multiplication of an 18bit and a 25bit two's complement numbers and produces one 48bit two's complement production. Programmable pipelining of input operands, intermediate products, and accumulator outputs enhances throughput and improves frequency. The DSP48E1 also has pipeline registers between operators to reduce the delay. The block RAM in the Virtex-6 FPGA is an embedded memory supporting synchronized read and write operations. In the Virtex-6 FPGA, it can configured as a 36Kbit dual port block RAMs, FIFOs, or two 18Kbit dual port RAMs. In our architecture, it is used as a 1K$\times$18bit dual port RAM.

Since FPGA chips maintain relatively low price and its programmable features, it is widely used in those fields which need to update architecture or functions frequently such as communication and education areas. They are widely used in consumer and industrial products for accelerating processor intensive algorithms [2], [3], [4], [5], [6], [7], [8].

Recently, mobile devices increasingly require the ability to perform computation that is performed on desktop platforms. To support the embedded processors in mobile devices, FPGAs will be used to implement coprocessors for applications such as signal processing, image processing, data encryption/decryption, etc. Especially, to perform real-time image processing such as object tracking and augmented reality with embedded video cameras, an FPGA is a promising device on mobile devices in the future.

Hough transform is a technique to find shapes in images [9]. In particular, it has been utilized to extract lines, circles, ellipses and arbitrary shapes. The Hough transform defines a mapping from an image into a parameter space represented by an accumulate array. The parameter space is defined by parameterizing detected shapes. Based on each edge point of the image, the mapping adds a vote to corresponding elements in the accumulate array. The elements that are increased represent associated parameters based on detected shapes. Therefore, the elements that are voted intensively correspond to the parameters of shapes in the image space.

The Hough transform can be used to extract straight lines in a binary image [10]. The idea of this method is to exploit the duality between points of a line and parameters of that line. A point in the image is represented by a curve in the parameter space and lines of collinear points intersect in the parameter space at one point. These intersections are counted in an array of accumulators that quantizes the parameter space appropriately. In the followings, we call this counting to the accumulators *voting*. More specifically, for each edge point (x, y) in a 2-dimentional image, the voting is performed along a curve $\rho = x \cos\theta + y \sin\theta$ ($0 \leq \theta < 180$). Possible lines can be detected by searching points that are voted intensively. Figure 1 shows an example of straight line detection using Hough transform. For an input image (Figure 1(a)), the binary edge image (Figure 1(b)) is obtained by the edge detector such as Sobel filter. The result of voting to the parameter space is shown in Figure 2. In this figure, darker points show points that are voted intensively, that is, represent probable lines. According to the result of voting, the principal lines are detected (Figure 1(c)).

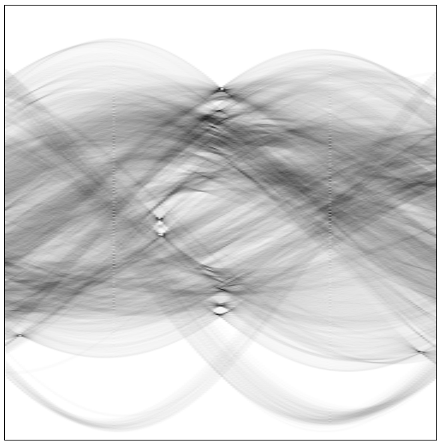


Figure 2. Hough parameter space

The main contribution of this paper is to present a new FPGA architecture for the Hough transform that fully utilizes embedded DSP blocks and block RAMs. Our new idea includes:

Voting Space Partitioning:
Polar coordinate voting space (θ, ρ) is partitioned and arranged into block RAMs. This enables us to perform voting operations in parallel. Also, the function of dual-port of block RAMs are fully used to accumulate the voting value instantly.

Efficient Usage of DSP blocks:
DSP blocks are used to compute $x \cos\theta$ and $y \sin\theta$ in parallel for each edge pixel (x, y). We compute $x \cos\theta$ and $y \sin\theta$ for θ such that $0 \leq \theta < 90$ instead of computing them for θ such that $0 \leq \theta < 180$. Also, we avoid the computation of the values of $\cos\theta$ and $\sin\theta$ by pre-loading them in the DSP blocks.

Fully Pipelined Architecture:
We take into account a layout of DSP blocks and block RAMs in Virtex-6 FPGA architecture, and design our Hough transform architecture as a fully pipelined one. For example, in the Virtex-6 FPGA XC6VLX240T has 768 DSP48E1 blocks arranged in 8 columns of 96 adjacent DSP48E1 blocks. Neighboring DSP48E1 blocks are connected directly through pipeline registers. Our Hough transform architecture uses 2 columns to compute $x \cos\theta$ and $y \sin\theta$ each, and uses a pipeline technique to maximize the clock frequency.

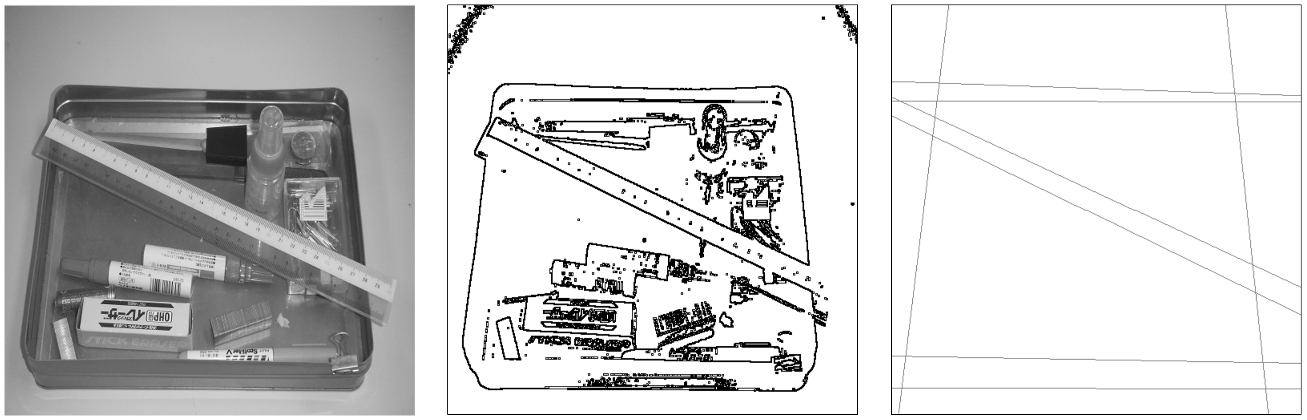
Using these ideas, our new architecture for the Hough transform uses 178 DSP48E1 blocks and 180 block RAMs with 18Kbits that work in parallel. One of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP blocks and block RAMs. Nevertheless, as far as we know, there is no previously published work that fully utilizes DSP blocks and block RAMs for the Hough transform. Roughly speaking, a conventional sequential implementation performs $180m$ voting operations for m edge points. Our architecture performs voting operations in parallel, and outputs identified straight lines in $m+97$ clock cycles. Since $180m$ voting operations are performed using 178 DSP48E1 blocks, the lower bound of the computing time is m clock cycles. Hence our implementation is close to optimal. We have implemented our new architecture on a Virtex-6 family FPGA XC6VLX240T-1. The circuit runs in 245.519MHz and outputs identified straight lines in $m + 97$ cycles. For example, Figure 1 includes 33232 edge points. Therefore, the circuit can perform the Hough transform in $135.75\mu s$.

Many hardware algorithms for FPGA implementation of the Hough transform for lines have been proposed in past. As far as we know, however, there is no published hardware algorithm using embedded DSP blocks or multipliers in the FPGA. In the existing researches, instead of circuits of multiplication with DSP blocks or multipliers, they introduced incremental Hough transform [11], [12], [13], CORDIC [14], [15], and hybrid-log arithmetic [16] to the computation of Hough transform. Since most of recent FPGAs produced by principal vendors equip embedded DSP blocks [17], [18], [19], one of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP blocks and block RAMs.

This paper is organized as follows. Section II introduces the Hough transform algorithms for lines. We show the FPGA architecture for the Hough transform in Section III. Section IV shows the experimental results. Finally, Section V concludes the paper.

II. Hough Transform

The main purpose of this section is to review Hough transform algorithms for straight lines. Suppose that we have an image of size $n \times n$. We assume that $n \times n$ pixels are arranged in two dimensional xy-space such that the origin is in the center of the image as illustrated in Figure 3.



(a) Input image
(b) Binary edge image by Sobel filter
(c) Line detection using Hough transform

Figure 1. Example of straight line detection using Hough transform

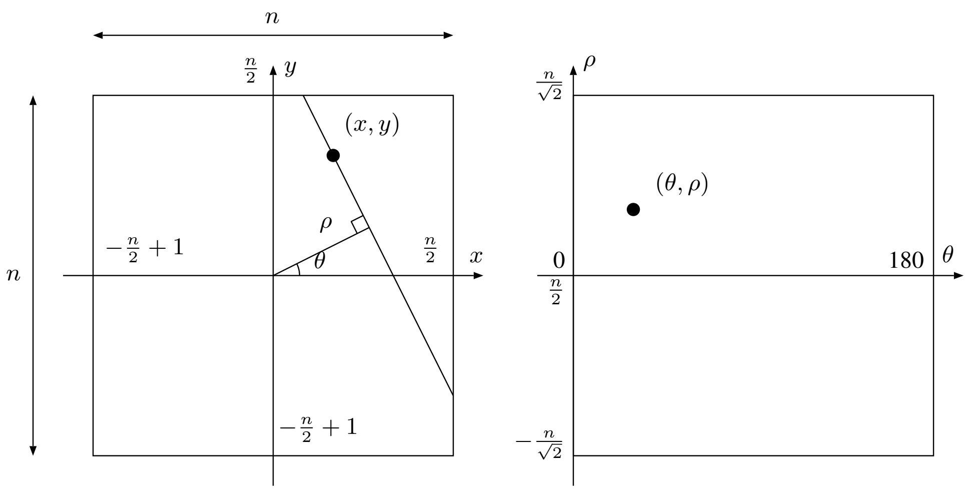


Figure 3. Two dimensional Spaces xy and $\theta\rho$ used in the Hough transform

Hence, both coordinates x and y take integers in the range $[-\frac{n}{2}+1, \frac{n}{2}]$.

A pixel (x, y) $(-\frac{n}{2}+1 \leq x, y \leq \frac{n}{2})$ in the xy-space is converted to a curve in the $\theta\rho$-space by the following formula:

$$\rho \quad = \quad x\cos\theta + y\sin\theta \quad (0 \leq \theta < 180) \tag{1}$$

Clearly, the double inequality $-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}}$ is satisfied. The values of θ and ρ can also be obtained geometrically. Suppose that we draw a line going through the origin with angle θ as illustrated in Figure 3. For such line, we can draw the orthogonal line going through a pixel (x, y). The value of ρ corresponds to the distance to the line. In other words, a point (θ, ρ) of $\theta\rho$-space corresponds to a line of xy-space.

The key idea of the Hough transform is to vote in $\theta\rho$-space for every pixel in the xy-space. Let $(x_0, y_0), (x_1, y_1), \ldots, (x_{k-1}, y_{k-1})$ be the k pixels in xy-space. The Hough transform is spelled out as follows:

[Straight Forward Hough Transform]

```
for $i \leftarrow 0$ to $k-1$
    for $\theta \leftarrow 0$ to 179
        begin
            $\rho \leftarrow x_k \cos\theta + y_k \sin\theta$
            $v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$
        end
for $\theta \leftarrow 0$ to 179 do in parallel
    for $\rho \leftarrow -\frac{n}{\sqrt{2}}$ to $\frac{n}{\sqrt{2}}$ do in parallel
        output $(\theta, \rho)$ if $v[\theta][\rho] \geq threshold$
```

For simplicity, we assume that the value of ρ is automatically rounded to an integer. In the Straight Forward Hough Transform, for each point (x_k, y_k), the values of $x_k \cos\theta$ and $y_k \sin\theta$ are computed for $\theta = 0, 1, \ldots, 179$. If $v[\theta][\rho]$ is storing a large value, many points in the k input pixels lie in the line in xy-space corresponds to a point (θ, ρ) in $\theta\rho$-space.

We will show that, it is sufficient to compute these values for $\theta = 0, 1, \ldots, 90$. From the addition theorem of trigonometric functions, we have

$$\begin{aligned} \rho &= x_k \cos(180-\theta) + y_k \sin(180-\theta) \\ &= -x_k \cos(\theta) + y_k \sin(\theta). \end{aligned} \qquad (2)$$

Using Formula (2), the Hough transform can also be done by partitioning the range $[0, 179]$ of θ into two ranges $[0, 89]$ and $[90, 179]$. Also, we avoid going through array v for finding elements larger than a threshold. Thus, our new Hough transform, called the Circuit-oriented Hough Transform is be spelled out as follows:

[Circuit-oriented Hough Transform]

```
for $i \leftarrow 0$ to $k-1$ do
    begin
        for $\theta \leftarrow 0$ to 89 do
            begin
                $\rho \leftarrow x_k \cos\theta + y_k \sin\theta$
                $v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$
                output $(\theta, \rho)$ if $v[\theta][\rho] = \textit{threshold}$
            end
        for $\theta \leftarrow 1$ to 90 do
            begin
                $\rho \leftarrow -x\cos(\theta) + y\sin(\theta)$
                $v[180-\theta][\rho] \leftarrow v[180-\theta][\rho] + 1$
                output $(\theta, \rho)$ if $v[\theta][\rho] = \textit{threshold}$
            end
    end
```

In the following section, we show an efficient implementation of the Circuit-oriented Hough Transform.

III. Our FPGA architecture for the Hough transform

This section describes our FPGA architecture for the Hough transform using DSP blocks and block RAMs in Xilinx Virtex-6 FPGA. We use Xilinx Virtex-6 Family FPGA XC6VLX240T-1 as the target device [20].

A. Structure of our architecture for the Hough transform

Figure 4 illustrates our architecture for the Hough transform. We use 178 DSP blocks $X_1, X_2, \ldots X_{89}$ and $Y_1, Y_2, \ldots, Y_{89}$. For each θ $(0 \leq \theta \leq 90)$ X_θ and Y_θ compute $x_k \cos\theta$ and $y_k \cos\theta$ for given x_k and y_k, respectively. Since $x_k \cos 0 = x_k$, $x_k \cos 90 = 0$, $y_k \sin 0 = 0$, and $y_k \cos 90 = y_k$, DSP blocks X_0, X_{90}, Y_0, and Y_{90} are not necessary. Using an adder and a subtractor for each pair X_θ and Y_θ, $\rho_\theta = x_k \cos\theta + y_k \cos\theta$ and $\rho_{180-\theta} = -x_k \cos\theta + y_k \cos\theta$ are computed. We also use 180 block RAMs $V_0, V_1, \ldots V_{179}$ to store the voting value. Address ρ of each V_θ $(0 \leq \theta \leq 179)$ is used to store the value of $v[\theta][\rho]$.

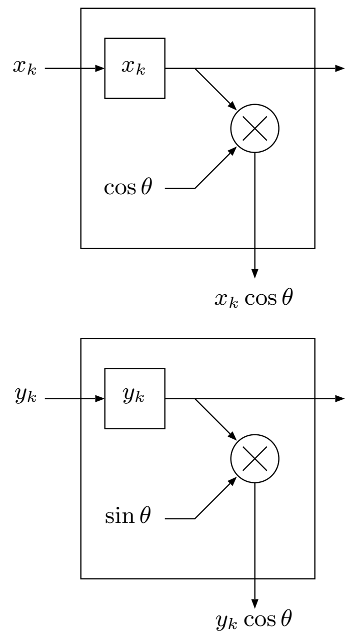


Figure 5. Two DSP blocks X_θ and Y_θ with an adder and subtracter to compute ρ

To minimize the delay between registers, DSP blocks $X_1, \ldots, X_{90}$ are connected in a pipeline fashion as illustrated in Figure 4. Each X_θ has a register to store the value of x_k. In every clock cycle, the value is transferred from X_θ to $X_{\theta+1}$. Similarly, DSP blocks $Y_0, Y_1, \ldots, Y_{90}$ are connected in a pipeline fashion.

Figure 5 illustrates two DSP blocks X_θ and Y_θ with an adder and subtracter to compute ρ. In X_θ, the value of x_k is loaded in an internal register. Also, the value of $\cos\theta$ is pre-computed. Note that the value of $\cos\theta$ used in X_θ is a fixed value. The product of x_k and $\cos\theta$ is computed in a multiplier of the DSP block X_θ. Similarly, the value of $\sin\theta$ used in Y_θ is a fixed value and the product of y_k and $\sin\theta$ is computed in a multiplier of the DSP block Y_θ.

In the Virtex-6 FPGA XC6VLX240T, that is our target device, has DSP48E1 blocks are arranged in 8 columns of 96 adjacent DSP48E1 blocks. Neighboring DSP48E1 blocks are connected directly through pipeline registers. Our Hough transform architecture uses 2 columns to compute $x_k \cos\theta$ and $y_k \sin\theta$ each, and uses a pipeline technique to maximize the clock frequency (Figure 6).

Figure 7 illustrates the architecture of V_θ using a block RAM. A block RAM in the FPGA is dual port architecture. Xilinx Virtex-6 Family has 18Kbit dual-port block RAMs, which have two sets of ports operated independently. Two sets of ports are:

Port Set A *ADDRA* (ADDRess A), *DOA* (Data Output A), *DIA* (Data Input A), and

Port Set B *ADDRB* (ADDRess B), *DOB* (Data Output

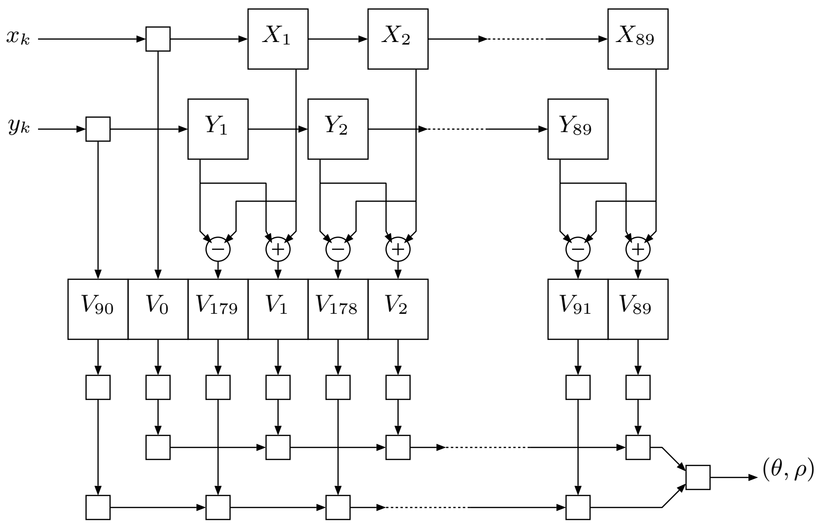


Figure 4. The outline of our FPGA architecture for the Hough transform

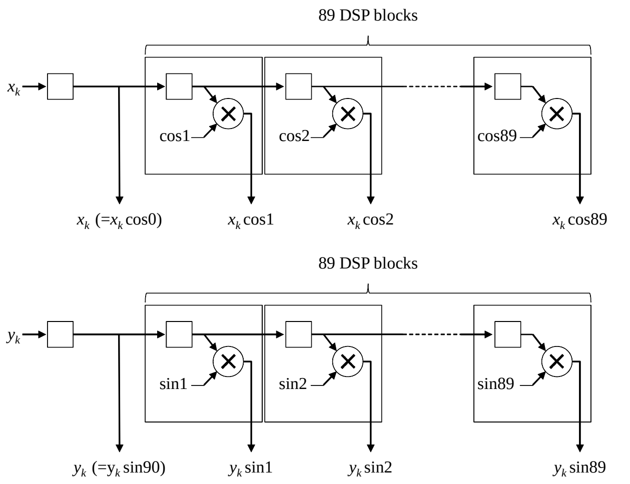


Figure 6. Pipeline architecture to compute $x_k \cos\theta$ and $y_k \sin\theta$ with DSP blocks

B), *DIB* (Data Input B).

Let $M[i]$ denote a data of address i of the block RAM. In read operation of Port Set A, $M[ADDRA]$ is output from *DOA* after the rising clock edge. In write operation of Port Set A, the data given to DIA is written in $M[ADDRA]$ at the rising clock edge. Read/write operations of Port Set B are the same as Port Set A. Port Set A and Port Set B work independently. In the block RAMs in the target device of this work, read/write operations can be configured as either RF (Read First) mode or WF (Write First) mode. In the RF mode, if reading and writing operations are performed to the same address, reading operation is performed before the reading operation. Hence the reading data is the data before writing data. On the other hand, in the WF mode, since the writing performed before the reading, the reading data is the updated data. However, when a dual port is used, there is a restriction that if read and write operation to the same address are performed for each port, the setting of block RAMs must be RF [21].

We use the block RAM to store the values of $v[\theta][\rho]$ ($-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}}$). Let $v_\theta[i]$ denote the data of address i in block RAM V_θ. Since ρ is given to it ADDRA, $v_\theta[\rho]$ is output from *DOA* after the rising clock edge as illustrated in Figure 7. After that, $v_\theta[\rho]+1$ is computed and it is given to *DOB*. Since ρ is given to *ADDB*, $v_\theta[\rho]+1$ is written in $v_\theta[\rho]$. In other words, $v_\theta[\rho] \leftarrow v_\theta[\rho]+1$ is performed. At that time, according to the restriction stated in the above, since the same value of ρ may be input continuously, the setting of block RAMs must be RF. Namely, when the same value of ρ is input continuously, the former voted value is not read from the block RAM. To avoid this situation, we use an additional register to store the latest voted value and if the same value of ρ is input continuously, the stored value is used instead of the value read from the block RAM.

In the same time, a comparator is used to determine if $v_\theta[\rho]+1 = threshold$. If so, the value of ρ is written in a register. After that, a pair (θ, ρ) is written into a next register. The pair (θ, ρ) represents a probable line. It moves toward the output of the circuit using series of shift registers one by one shown in Figure 4. In order to reduce the number of clock cycles necessary to move data to the output, we use two series of shift registers. One is used for output data of $V_0, \ldots, V_{89}$. The other is used for output data of $V_{90}, \ldots, V_{179}$. Therefore, the number of clock cycles necessary to move data to the output is reduced to at most 90 clock cycles.

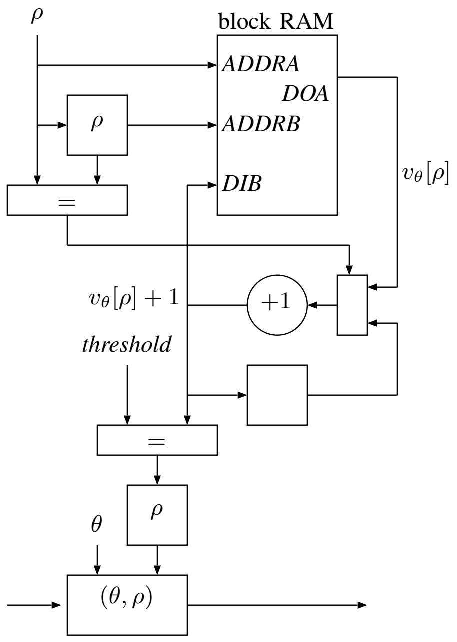


Figure 7. A block RAM V_θ to store $v[\theta][\rho]$

B. Data representation

The choice of data precision is guided by the implementation cost in terms of area, simplicity of design, speed and power consumption. Higher precision will lead to less quantization error in the final implementation. On the other hand, lower precision will produce more compaction and faster designs with less power consumption. A trade-off choice needs to be made depending on the given application and available FPGA resources.

In our work, in order to minimize chip space and computation time, short fixed point representation of numbers are used. Considering the structure of DSP blocks and block RAMs, we choose the data presentation in our implementation, as follows. The data format of inputs that are pairs of coordinates x_k and y_k are 10bit two's complement integer each. Also, the data format of $\cos\theta$ and $\sin\theta$ is 16bit fixed point number, which consists of 1bit sign, 1bit integer and 14bit fraction based on two's complement. On the other hand, the data format of ρ is 10bit two's complement integer. The data format of the voted value is 18bit integer. Namely, the number of the vote is at most $2^{18}-1$. Since the range of the value of θ is 0 to 180, the data format of θ is 8bit integer.

IV. Experimental Results

We have implemented the proposed architecture for Hough transform and evaluated it on the Xilinx Virtex-6 FPGA XC6VLX240T-1. Table I shows the experimental results using Xilinx ISE 13.1. In the implementation, to reduce the delay of the circuit, some pipeline registers are inserted into between circuit elements. It takes 3 clock cycles to compute the values of ρ for given x_k and y_k. Also, 4 clock cycles are necessary to output a pair (θ,ρ) that represents a probable line. Moreover, the number of clock cycles necessary to move data to the output is reduced to at most 90 clock cycles. Therefore, this circuit can output identified straight lines represented by (θ,ρ) in $m+97$ cycles, i.e., $\frac{m+97}{245.519}\mu s$. For example, Figure 1(b) includes 33232 edge points. Therefore, the circuit can perform the Hough transform in $135.75\mu s$. If the input image is worst case in terms of the computing time, that is, if all the points of an image of size $512\times512(=262144)$ are edge points, it takes $1068.11\mu s$ to complete to output the results. Of course, it is not possible that all points are edge points, however, this fact guarantees that our Hough transform implementation for any 512×512 image terminates in less than $1068.11\mu s$.

Table I
PERFORMANCE EVALUATION OF THE PROPOSED ARCHITECTURE FOR HOUGH TRANSFORM

DSP48E1 blocks (out of 768)	178 (23.1%)
18Kbit block RAMs (out of 832)	180 (21.6%)
Slices (out of 301440)	14493 (4.81%)
Clock frequency [MHz]	245.519

For the purpose of estimating the speed up of our FPGA implementation, we have also implemented a conventional software approach of Hough transform using GNU C. We have used Intel Xeon X7460 running in 2.66GHz and 128GB memory to run the sequential algorithm for Hough transform. For the image shown in Figure 1(b) that includes 33232 edge points, the software implementation can perform the Hough transform in $413.98ms$. Also, if all the points of an image of size $512\times512(=262144)$ are edge points, it takes $3266.75ms$ to complete to output the results. Therefore, our FPGA implementation attains a speed-up factor of more than 3000 over the sequential implementation on the CPU.

There are a number of literatures reported to implement Hough transform for lines using the FPGA shown in Section I. Performances such as device, logic blocks, DSP blocks, frequency and throughput are compared in Table II. It is difficult to directly compare to other works because utilized FPGAs and supported size of images differ. Considering the throughput, however, it is clear that the performance of our FPGA implementation is better than that of other works.

V. Conclusions

We have presented a new architecture of the Hough transform for the straight lines using DSP blocks and block RAMs in the Virtex-6 Family FPGA. Partitioning the parameter space to vote, the 180 voting operations are performed in parallel with 178 DSP48E1s and 180 18Kbit block RAMs.

Table II
COMPARISON WITH RELATED WORKS FOR HOUGH TRANSFORM

	Karabernou [14]	Deng [15]
Device	XC4010EPC84	XC4010XL
Logic blocks	205 CLBs	333 CLBs
DSP blocks	—	—
Frequency	23.166MHz	40MHz
Throughput	10.368Mpixel/s	0.623Mpixel/s

	Lee [16]	This work
Device	Virtex 4	XC6VLX240T-1
Logic blocks	314 CLBs	14493 Slices
DSP blocks	—	178 DSP48E1s
Frequency	132MHz	245.519MHz
Throughput	32.768Mpixel/s	245.428Mpixel/s

We have implemented our architecture on the Virtex-6 Family FPGA XC6VLX240T-1. The experimental results show that this implementation runs in 245.519MHz and given m coordinates of edge points, it can output identified straight lines in $m + 97$ cycles, i.e., $\frac{m+97}{245.519}\mu s$.

REFERENCES

[1] Xilinx Inc., *Virtex-6 FPGA DSP48E1 Slice User Guide (v1.3)*, 2011.

[2] J. L. Bordim, Y. Ito, and K. Nakano, "Accelerating the CKY parsing using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 803–810, May 2003.

[3] ——, "Instance-specific solutions to accelerate the CKY parsing for large context-free grammars," *International Journal on Foundations of Computer Science*, pp. 403–416, 2004.

[4] Y. Ito and K. Nakano, "Efficient exhaustive verification of the Collatz conjecture using DSP blocks of Xilinx FPGAs," *International Journal of Networking and Computing*, vol. 1, no. 1, pp. 49–62, 2011.

[5] Y. Ito, K. Nakano, and S. Bo, "The parallel FDFM processor core approach for CRT-based RSA decryption," *International Journal of Networking and Computing*, vol. 2, no. 1, pp. 56–78, 2012.

[6] K. Nakano and E. Takamichi, "An image retrieval system using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, May 2003.

[7] K. Nakano and Y. Yamagishi, "Hardware n choose k counters with applications to the partial exhaustive search," *IEICE Trans. on Information & Systems*, 2005.

[8] Y. Ago, Y. Ito, and K. Nakano, "An FPGA implementation for neural networks with the FDFM processor core approach," *International Journal of Parallel, Emergent and Distributed Systems*, pp. 1–13, 2012.

[9] P. V. C. Hough, "Method and means for recognizing complex patterns," U.S. Patent 3,069,654, 1962.

[10] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, 1972.

[11] S. Tagzout, K. Achour, and O. Djekoune, "Hough transform algorithm for FPGA implementation," *Signal Processing*, vol. 81, no. 6, pp. 1295–1301, 2001.

[12] H. Bessalah, S. Seddiki, F. Alim, and M. Bencherif, "On line mode incremental Hough transform implementation on Xilinx fpga's," in *Proc. of the 8th conference on Signal, Speech and image processing*, 2008, pp. 176–179.

[13] O. Djekoune and K. Achour, "Incremental Hough transform: an improved algorithm for digital device implementation," *Real-Time Imaging*, vol. 10, no. 6, pp. 351–363, 2004.

[14] S. M. Karabernou and F. Terranti, "Real-time FPGA implementation of Hough transform using gradient and CORDIC algorithm," *Image and Vision Computing*, vol. 23, no. 11, pp. 1009–1017, 2005.

[15] D. D. S. Deng and H. ElGindy, "High-speed parameterisable Hough transform using reconfigurable hardware," in *Proc. of the Pan-Sydeny area workshop on Visual information processing*, vol. 11, 2001, pp. 51–57.

[16] P. Lee and A. Evagelos, "An implementation of a multiplierless Hough transform on an FPGA platform using hybrid-log arithmetic," in *Proc. of Real-Time Image Processing 2008*, vol. 6811, 2008, pp. 68 110G–1.

[17] Xilinx Inc., *Virtex-4 FPGA User Guide(v2.6)*, 2008.

[18] ——, *Virtex-5 FPGA User Guide(v5.2)*, 2009.

[19] Altera Corp., *Stratix V Device Handbook*, 2012.

[20] Xilinx Inc., *Virtex-6 Family Overview(v2.4)*, 2012.

[21] ——, *Virtex-6 FPGA Memory Resources User Guide (v1.6)*, 2011.

Accelerating Dynamic Programming for the Optimal Polygon Triangulation on the GPU

Kazufumi Nishida, Koji Nakano, and Yasuaki Ito
Department of Information Engineering, Hiroshima University,
Kagamiyama 1-4-1, Higashi Hiroshima 739-8527, Japan

***Abstract*—Modern GPUs (Graphics Processing Units) can be used for general purpose parallel computation. Users can develop parallel programs running on GPUs using programming architecture called CUDA (Compute Unified Device Architecture). The optimal polygon triangulation problem for a convex polygon is an optimization problem to find a triangulation with minimum total weight. It is known that this problem can be solved using the dynamic programming technique in $O(n^3)$ time using a work space of size $O(n^2)$. The main contribution of this paper is to present an efficient parallel implementation of this $O(n^3)$-time algorithm on the GPU. In our implementation, we have used two new ideas to accelerate the dynamic programming. The first idea (granularity adjustment) is to partition the dynamic programming algorithm into many sequential kernel calls of CUDA, and to select the best size and number of blocks and threads for each kernel call. The second idea (sliding and mirroring arrangements) is to arrange the temporary data for coalesced access of the global memory in the GPU to minimize the memory access overhead. Our implementation using these two ideas solves the optimal polygon triangulation problem for a convex 16384-gon in 69.1 seconds on the NVIDIA GeForce GTX 580, while a conventional CPU implementation runs in 17105.5 seconds. Thus, our GPU implementation attains a speedup factor of 247.5.**

***Keywords*-Dynamic programming; parallel algorithms; coalesced memory access; GPGPU; CUDA**

I. Introduction

The GPU (Graphical Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [3], [4], [5]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1], [6]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [7], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [8], since they have hundreds of processor cores running in parallel.

Dynamic programming is an important algorithmic technique to find an optimal solution of a problem over an exponential number of solution candidates [9]. A naive solution for such problem needs exponential time. The key idea behind dynamic programming is to:

- partition a problem into subproblems,
- solve the subproblems independently, and
- combine the solution of the subproblems

to reach an overall solution. Dynamic programming enables us to solve such problems in polynomial time. For example, the longest common subsequence problem, which requires finding the longest common subsequence of given two sequences, can be solved by the dynamic programming approach [10]. Since a sequence have an exponential number of subsequences, a straightforward algorithm takes an exponential time to find the longest common subsequence. However, it is known that this problem can be solved in $O(nm)$ time by the dynamic programming approach, where n and m are the lengths of two sequences. Many important problems including the edit distance problem, the matrix chain product problem, and the optimal polygon triangulation problem can be solved by the dynamic programming approach [9].

The main contribution of this paper is to implement the dynamic programming approach to solve *the optimal polygon triangulation problem* [9] on the GPU. Suppose that a convex n-gon is given and we want to triangulate it, that is, to split it into $n-2$ triangles by $n-3$ non-crossing chords. Figure 1 illustrates an example of a triangulation of an 8-gon. In the figure, the triangulation has 6 triangles separated by 5 non-crossing chords. We assume that each of the $\frac{n(n-3)}{2}$ chords is assigned a weight. The goal of the optimal polygon triangulation is to select $n-3$ non-crossing chords that triangulate a given convex n-gon such that the total weight of selected chords is minimized. This problem is applied to matrix chain multiplication that is an optimization problem. Matrix chain multiplication is a special case of optimal polygon triangulation problem, i.e., instances of matrix chain multiplication can be computed as optimal polygon triangulation problem [9]. A naive approach, which evaluates the total weights of all possible $\frac{(2n-4)!}{(n-1)!(n-2)!}$ triangulations, takes an exponential time. On the other hand, it is known that the dynamic programming technique can be applied to solve the optimal polygon triangulation in $O(n^3)$ time [9], [11], [12] using work space of size $O(n^2)$. As far as we know, there is no previously published algorithm running faster than $O(n^3)$ time.

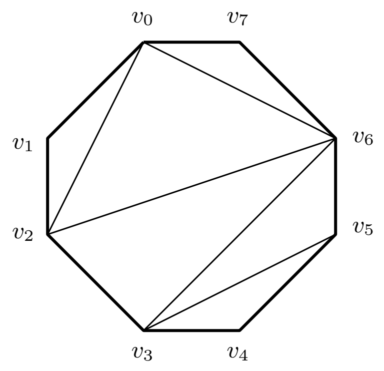


Figure 1. An example of a triangulation of a convex 8-gon

In our implementation, we have used two new ideas to accelerate the dynamic programming algorithm. The first idea is to partition the dynamic programming algorithm into a lot of sequential kernel calls of CUDA, and to select the best method and the numbers of blocks and threads for each kernel calls (*granularity adjustment*). The dynamic programming algorithm for an n-gon has $n-1$ stages, each of which involves the computation of multiple temporary data. Earlier stages of the algorithm are *fine grain* in the sense that we need to compute the values of a lot of temporary data but the computation of each temporary data is light. On the other hand, later stages of the algorithm are *coarse grain* in the sense that few temporary data are computed but the computation is heavy. Thus, in earlier stages, a single thread is assigned to the computation of each temporary data and its value is computed sequentially by the thread (*OneThreadPerEntry*). In middle stages, a block with multiple threads is allocated to the computation for each temporary data and the value of the temporary data is computed by threads of a block in parallel (*OneBlockPerEntry*). Multiple blocks are allocated to compute each temporary data in later stages (*BlocksPerEntry*). Also, the size of each block (i.e. the number of threads), and the number of used blocks affects the performance of algorithms on the GPU. We have tested all of the three methods for various sizes of each block and the number of blocks for every stage, and determined the best way, one of the three methods and the size and the number of blocks for computing the temporary data in each stage.

The second idea is to arrange temporary data in a 2-dimensional array of the global memory using two types of arrangements: *sliding arrangement* and *mirroring arrangement*. The temporary data used in the dynamic programming algorithm are stored in a 2-dimensional array in the global memory of the GPU. The bandwidth of the global memory is maximized when threads repeatedly performs coalesced access to it. In other words, if threads accessed to continuous locations of the global memory, these access requests can be completed in minimum clock cycles. On the other hand, if threads accessed to distant locations in the same time, these access requests need a lot of clock cycles. We use the sliding arrangement for OneThreadPerEntry and the mirroring arrangement for OneBlockPerEntry and BlocksPerEntry. Using these two arrangements, the coalesced access is performed for the temporary data.

Our implementation using these two ideas solves the optimal polygon triangulation problem for a convex 16384-gon in 69.1 seconds on the NVIDIA GeForce GTX 580, while a conventional CPU implementation runs in 17105.5 seconds. Thus, our GPU implementation attains a speedup factor of 247.5.

The rest of this paper is organized as follows; Section II introduces the optimal polygon triangulation problem and reviews the dynamic programming approach solving it. In Section III, we show the GPU and CUDA architectures to understand our new idea. Section IV proposes our two new ideas to implement the dynamic programming approach on the GPU. The experimental results are shown in Section V. Finally, Section VI offers concluding remarks.

II. The optimal polygon triangulation and the dynamic programming approach

The main purpose of this section is to define the optimal polygon triangulation problem and to review an algorithm solving this problem by the dynamic programming approach [9].

Let $v_0, v_1, \ldots, v_{n-1}$ be vertices of a convex n-gon. Clearly, the convex n-gon can be divided into $n-2$ triangles by a set of $n-3$ non-crossing chords. We call a set of such $n-3$ non-crossing chords *a triangulation*. Figure 1 shows an example of a triangulation of a convex 8-gon. The convex 8-gon is separated into 6 triangles by 5 non-crossing chords. Suppose that a weight $w_{i,j}$ of every chord v_iv_j in a convex n-gon is given. The goal of *the optimal polygon triangulation problem* is to find an optimal triangulation that minimizes the total weights of selected chords for the triangulation. More formally, we can define the problem as follows. Let T be a set of all triangulations of a convex n-gon and $t \in T$ be a triangulation, that is, a set of $n-3$ non-crossing chords. The optimal polygon triangulation problem requires finding the total weight of a minimum weight triangulation as follows:

$$\min\{ \sum_{v_iv_j \in t} w_{i,j} \mid t \in T\}.$$

We will show that the optimal polygon triangulation can be solved by the dynamic programming approach. For this purpose, we define *the parse tree* of a triangulation. Figure 2 illustrates the parse tree of a triangulation. Let l_i $(1 \leq i \leq n-1)$ be edge $v_{i-1}v_i$ of a convex n-gon. Also, let r denote edge v_0v_{n-1}. The parse tree is a binary tree

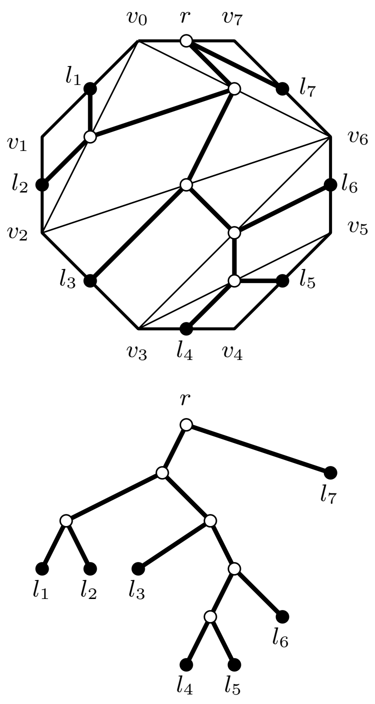


Figure 2. The parse tree of a triangulation

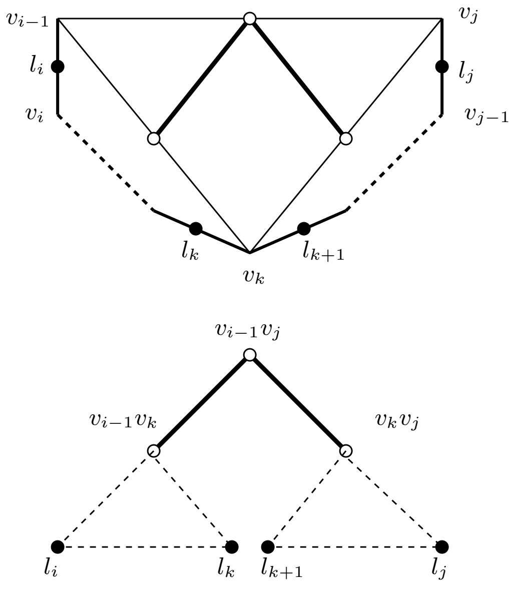


Figure 3. A $(j-i)$-gon is partitioned into a $(k-i)$-gon and a $(j-k)$-gon

of a triangulation, which has the root r and $n-1$ leaves $l_1, l_2, \ldots, l_{n-1}$. It also has $n-3$ internal nodes (excluding the root r), each of which corresponds to a chord of the triangulation. Edges are drawn from the root toward the leaves as illustrated in Figure 2. Since each triangle has three nodes, the resulting graph is a full binary tree with $n-1$ leaves, in which every internal node has exactly two children. Conversely, for any full binary tree with $n-1$ leaves, we can draw a unique triangulation. It is well known that the number of full binary trees with $n+1$ leaves is the Catalan number $\frac{(2n)!}{(n+1)!n!}$[13]. Thus, the number of possible triangulations of convex n-gon is $\frac{(2n-4)!}{(n-1)!(n-2)!}$. Hence, a naive approach, which evaluates the total weights of all possible triangulations, takes an exponential time.

We are now in position to show an algorithm using the dynamic programming approach for the optimal polygon triangulation problem. Suppose that an n-gon is chopped off by a chord $v_{i-1}v_j$ $(0 \leq i < j \leq n-1)$ and we obtain a $(j-i)$-gon with vertices $v_{i-1}, v_i, \ldots, v_j$ as illustrated in Figure 3. Clearly, this $(j-i)$-gon consists of leaves $l_i, l_{i+1}, \ldots, l_j$ and a chord $v_{i-1}v_j$. Let $m_{i,j}$ be the minimum weight of the $(j-i)$-gon. The $(j-i)$-gon can be partitioned into the $(k-i)$-gon, the $(j-k)$-gon, and the triangle $v_{i-1}v_kv_j$ as illustrated in Figure 3. The values of k can be an integer from i to $j-1$. Thus, we can recursively define $m_{i,j}$ as follows:

$$
\begin{aligned}
m_{i,j} &= 0 \qquad \text{if } j-i \leq 1, \\
m_{i,j} &= \min_{i \leq k \leq j-1}(m_{i,k} + m_{k+1,j} + w_{i-1,k} + w_{k,j}) \quad \text{otherwise.}
\end{aligned}
$$

The figure also shows its parse tree. The reader should have no difficulty to confirm the correctness of the recursive formula and the minimum weight of the n-gon is equal to $m_{1,n-1}$.

Let $M_{i,j} = m_{i,j} + w_{i-1,j}$ and $w_{0,n-1} = 0$. We can recursively define $M_{i,j}$ as follows:

$$
\begin{aligned}
M_{i,j} &= 0 \qquad \text{if } j-i \leq 1, \\
M_{i,j} &= \min_{i \leq k \leq j-1}(M_{i,k} + M_{k+1,j}) + w_{i-1,j} \quad \text{otherwise.}
\end{aligned}
$$

It should be clear that $M_{1,n-1} = m_{1,n-1} + w_{0,n-1} = m_{1,n-1}$ is the minimum weight of the n-gon.

Using the recursive formula for $M_{i,j}$, all the values of $M_{i,j}$ can be computed in $n-1$ stages by the dynamic programming algorithm as follows:

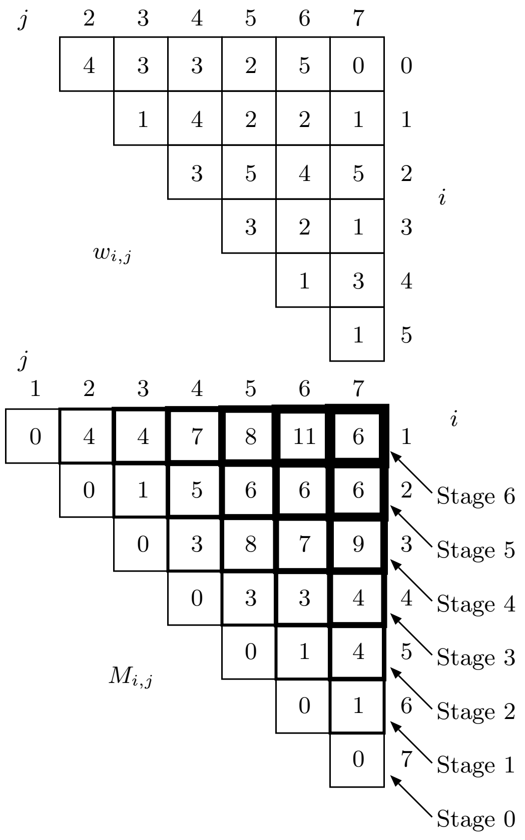
Stage 0 $M_{1,1} = M_{2,2} = \cdots = M_{n-1,n-1} = 0$.
Stage 1 $M_{i,i+1} = w_{i-1,i+1}$ for all i $(1 \leq i \leq n-2)$
Stage 2 $M_{i,i+2} = \min_{i \leq k \leq i+1}(M_{i,k} + M_{k+1,i+2}) + w_{i-1,i+2}$ for all i $(1 \leq i \leq n-3)$

$\vdots$

Stage p $M_{i,i+p} = \min_{i \leq k \leq i+p-1}(M_{i,k} + M_{k+1,i+p}) + w_{i-1,i+p}$ for all i $(1 \leq i \leq n-p-1)$

$\vdots$

Stage n − 3 $M_{i,n+i-3} = \min_{i \leq k \leq n+i-4}(M_{i,k} + M_{k+1,n+i-3}) + w_{i-1,n+i-3}$ for all i $(1 \leq i \leq 2)$



j	2	3	4	5	6	7
$i=0$	4	3	3	2	5	0
1		1	4	2	2	1
2			3	5	4	5
3				3	2	1
4					1	3
5						1

j	1	2	3	4	5	6	7
$i=1$	0	4	4	7	8	11	6
2		0	1	5	6	6	6
3			0	3	8	7	9
4				0	3	3	4
5					0	1	4
6						0	1
7							0

Figure 4. Examples of $w_{i,j}$ and $M_{i,j}$

Stage $n-2$: $M_{1,n-1} = \min_{1\leq k\leq n-2}(M_{i,k} + M_{k+1,n-1}) + w_{0,n-1}$

Figure 4 shows examples of $w_{i,j}$ and $M_{i,j}$ for a convex 8-gon. It should be clear that each stage computes the values of table $M_{i,j}$ in a particular diagonal position. Let us analyze the computation performed in each Stage p ($2 \leq p \leq n-2$).

- $(n-p-1)$ $M_{i,j}$'s, $M_{1,p+1}, M_{2,p+2}, \ldots, M_{n-p-1,n-1}$ are computed, and
- the computation of each $M_{i,j}$'s involves the computation of the minimum over p values, each of which is the sum of two $M_{i,j}$'s.

Thus, Stage p takes $(n-p-1)\cdot O(p) = O(n^2 - p^2)$ time. Therefore, this algorithm runs in $\sum_{2\leq p\leq n-2} O(n^2-p^2) = O(n^3)$ time.

From this analysis, we can see that earlier stages of the algorithm is *fine grain* in the sense that we need to compute the values of a lot of $M_{i,j}$'s but the computation of each $M_{i,j}$ is light. On the other hand, later stages of the algorithm is *coarse grain* in the sense that few $M_{i,j}$'s are computed but its computation is heavy.

III. GPU and CUDA Architectures

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [7]. The

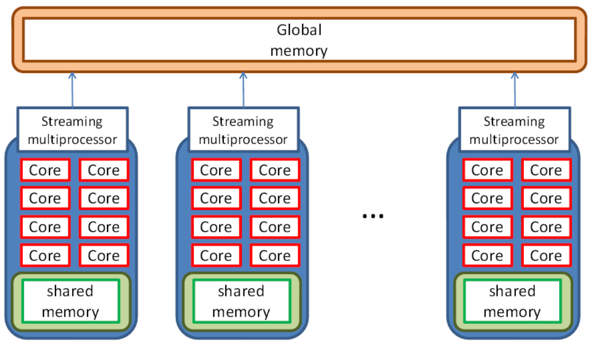


Figure 5. CUDA hardware architecture

global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [14], [3], [8]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory. Figure 5 illustrates the CUDA hardware architecture.

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, as we can see in Figure 5, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories.

CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each block are executed by processor cores in a single streaming processor. The kernel calls terminates, when threads in all blocks finish the computation. Since all threads in a single block are executed by a single streaming processor, the barrier synchronization of them can be done by calling CUDA C `syncthreds()` function. However, there is no direct way to synchronize threads in different blocks. One of the indirect methods of inter-block barrier synchronization is to partition the computation into kernels. Since continuous kernel calls can be executed such that a kernel is called after all blocks of the previous kernel terminates, execution of blocks is synchronized at the end of kernel calls. Thus, we arrange a single kernel call to each of $n-1$ stages of the dynamic programming algorithm for the optimal polygon triangulation problem.

As we have mentioned, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 6, when threads access to continuous locations in a row of a two-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed in the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed in the same time (*stride access*). From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus, we should avoid the stride access (or the vertical access) and perform the coalesced access (or the horizontal access) whenever possible.

IV. Our implementation of the dynamic programming approach for the optimal polygon triangulation

The main purpose of this section is to show our implementation of dynamic programming for the optimal polygon triangulation in the GPU. We focus on our new ideas, granularity adjustment and sliding and mirroring arrangements for accelerating the dynamic programming algorithm.

A. Granularity adjustment technique

Recall that each Stage p ($2 \leq p \leq n-2$) consists of the computation of $(n-p-1)$ $M_{i,j}$'s each of which involves the computation of the minimum of p values. We consider three methods, *OneThreadPerEntry*, *OneBlockPerEntry*, and *BlocksPerEntry* to perform the computation of each of the $n-2$ stages. In OneThreadPerEntry, each $M_{i,i+p}$ is computed sequentially by one thread. In OneBlockPerEntry, each $M_{i,i+p}$ is computed by one block with multiple threads in parallel. In BlocksPerEntry, each $M_{i,i+p}$ is computed by multiple blocks in parallel.

Let t be the number of threads in each block and b be the number of blocks. In our implementation of the three methods, t and b can be the parameters that can be changed to get the best performance. The details of the implementation of the three methods are spelled out as follows:

OneThreadPerEntry(t): Each $M_{i,i+p}$ is computed by a single thread sequentially. Thus, we use $(n-p-1)$ threads totally. Since each block has t threads, $\frac{n-p-1}{t}$ blocks are used.

OneBlockPerEntry(t):Each $M_{i,i+p}$ is computed by a block with t threads. The computation of $M_{i,i+p}$ involves the p sums $M_{i,k} + M_{i+p,k+1}$ ($i \leq k \leq i+p-1$). The t threads compute p sums in parallel such that each thread computes $\frac{p}{t}$ sums and their local minimum of the $\frac{p}{t}$ sums is computed. The resulting local t minima are written into the shared memory. After that, a single thread is used to compute the minimum of the t local minima.

BlocksPerEntry(b,t):Each $M_{i,i+p}$ is computed by b blocks with t threads each. The computation of p sums is arranged b blocks equally. Thus, each block computes the $\frac{p}{b}$ sums and their minimum is computed in the same way as OneBlockPerEntry(t). The resulting b minima are written to the global memory. The minimum of the b minima is computed by a single thread.

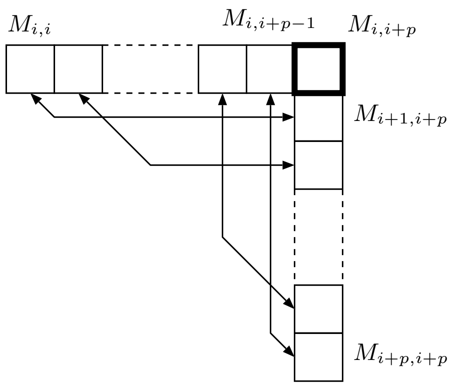


Figure 7. The computation of $M_{i,i+p}$

For each Stage p ($2 \leq p \leq n-2$), we can choose one of the three methods OneThreadPerEntry(t), OneBlockPerEntry(t), and BlocksPerEntry(b,t), independently.

B. Sliding and mirroring arrangement

Recall that, each Stage p ($2 \leq p \leq n-2$) of the dynamic programming algorithm involves the computation

$$M_{i,i+p} = \min_{i \leq k \leq i+p-1} (M_{i,k} + M_{k+1,i+p}) + w_{i-1,i+p}.$$

Let us first observe *the naive arrangement* which allocates each $M_{i,j}$ to the (i,j) element of the 2-dimensional array, that is, the element in the i-th row and the j-th column. As illustrated in Figure 7, to compute $M_{i,i+p}$ in Stage p

- p temporary data $M_{i,i}, M_{i,i+1}, \ldots, M_{i,i+p-1}$ in the same row and
- p temporary data $M_{i+1,i+p}, M_{i+2,i+p}, \ldots, M_{i+p,i+p}$ in the same column

are accessed. Hence, the naive arrangement involves the vertical access (or the stride access), which decelerates the computing time.

For the coalesced access of the global memory, we present two arrangements of $M_{i,j}$s in a 2-dimensional array, *the sliding arrangement* and *the mirroring arrangement* as follows:

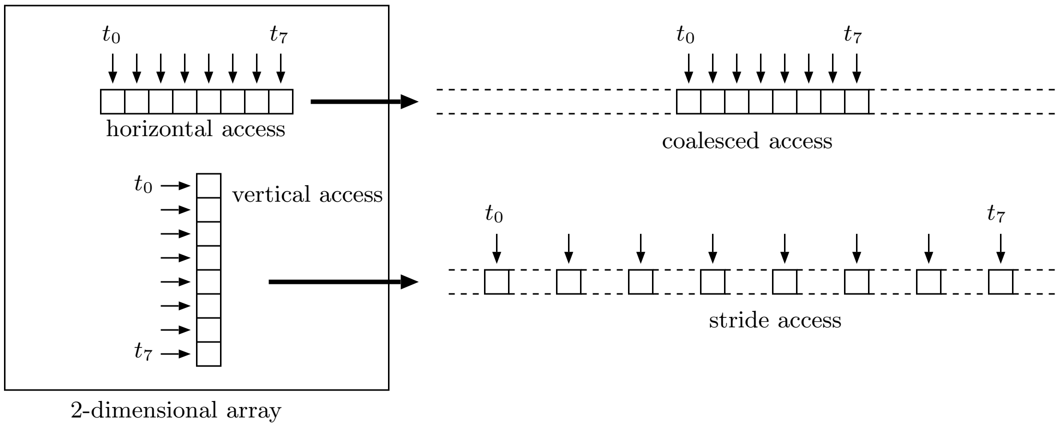


Figure 6. Coalesced and stride access

Sliding arrangement: Each $M_{i,j}$ $(0 \le i \le j \le n-1)$ is allocated to $(i - j + n, j)$ element of the 2-dimensional array of size $n \times n$.

Mirroring arrangement: Each $M_{i,j}$ $(0 \le i \le j \le n-1)$ is allocated to (i, j) element and (j, i) element.

The reader should refer to Figure 8 for illustrating the sliding and mirroring arrangements. We will use sliding arrangement for OneThreadPerEntry and the mirroring arrangement for OneBlockPerEntry and BlocksPerEntry.

We will show that the vertical access can be avoided if we use the sliding arrangement for OneThreadPerEntry. Suppose that each thread i computes the value of $M_{i,i+p}$. First, each thread i reads $M_{i,i}$ in parallel and then read $M_{i+1,i+p}$ in parallel. Thus, $M_{0,0}, M_{1,1}, \ldots$ are read in parallel and then $M_{1,1+p}, M_{2,2+p}, \ldots$ are read in parallel. Clearly, $M_{0,0}, M_{1,1}, \ldots$ are in the same row of the sliding arrangement. Also, $M_{1,1+p}, M_{2,2+p}, \ldots$ are also in the same row. Thus, the coalesced read is performed. Similarly, we can confirm that the remaining read operations by multiple threads perform the coalesced read.

Next, we will show that the vertical access can be avoided if we use the mirroring arrangement for OneBlockPerEntry and BlocksPerEntry. Suppose that a block computes the value of $M_{i,i+p}$. Threads in the block read $M_{i,i}, M_{i,i+1}, \ldots, M_{i,i+p-1}$ in parallel, and then read $M_{i+1,i+p}, M_{i+2,i+p}, \ldots, M_{i+p,i+p}$ in parallel. Clearly, $M_{i,i}, M_{i,i+1}, \ldots, M_{i,i+p-1}$ are stored in $(i, i), (i, i + 1), \ldots, (i, i + p - 1)$ elements in the 2-dimensional array of the mirroring arrangement, and thus, threads perform the coalesced read. For the coalesced read, threads read $M_{i+1,i+p}, M_{i+2,i+p}, \ldots, M_{i+p,i+p}$ stored in $(i + p, i + 1), (i + p, i + 2), \ldots, (i + p, i + p)$ elements in the 2-dimensional array of the mirroring arrangement. Clearly, these elements are in the same row and the threads perform the coalesced read.

C. Our algorithm for the optimal polygon triangulation

Our algorithm for the optimal polygon triangulation is designed as follows: For each Stage p $(2 \le p \le n - 2)$, we execute three methods OneThreadPerEntry(t), OneBlockPerEntry(t), and BlocksPerEntry(b, t) for various values of t and b, and find the fastest method and parameters. As we are going to show later, OneThreadPerEntry is the fastest in earlier stages. In middle stages, OneBlockPerEntry is fastest. Finally, BlocksPerEntry is the best in later stages. Thus, we first use the sliding arrangement in earlier stages computed by OneThreadPerEntry. We then convert the 2-dimensional array with the sliding arrangement into the mirroring arrangement. After that, we execute OneBlockPerEntry and then BlocksPerEntry in the remaining stages. Note that the computing time of our algorithm depends only on the number of vertices, i.e., it is independent from the weights of edges. Therefore, given the number of vertices, we can find and determine the fastest method and parameters.

V. Experimental results

We have implemented our dynamic programming algorithm for the optimal polygon triangulation using CUDA C. We have used NVIDIA GeForce GTX 580 with 512 processing cores (16 Streaming Multiprocessors which has 32 processing cores) running in 1.544GHz and 3GB memory. For the purpose of estimating the speedup of our GPU implementation, we have also implemented a conventional software approach of dynamic programming for the optimal polygon triangulation using GNU C. We have used Intel Core i7 870 running in 2.93GHz and 8GB memory to run the sequential algorithm for dynamic programming.

Table I shows the computing time in seconds for a 16384-gon. Table I (a) shows the computing time of OneThreadPerEntry(t) for t = 32, 64, 128, 256, 512,

					$M_{0,5}$
				$M_{0,4}$	$M_{1,5}$
			$M_{0,3}$	$M_{1,4}$	$M_{2,5}$
		$M_{0,2}$	$M_{1,3}$	$M_{2,4}$	$M_{3,5}$
	$M_{0,1}$	$M_{1,2}$	$M_{2,3}$	$M_{3,4}$	$M_{4,5}$
$M_{0,0}$	$M_{1,1}$	$M_{2,2}$	$M_{3,3}$	$M_{4,4}$	$M_{5,5}$

Sliding arrangement

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{0,4}$	$M_{0,5}$
$M_{0,1}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{1,4}$	$M_{1,5}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{2,3}$	$M_{2,4}$	$M_{2,5}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$	$M_{3,4}$	$M_{3,5}$
$M_{0,4}$	$M_{1,4}$	$M_{2,4}$	$M_{3,4}$	$M_{4,4}$	$M_{4,5}$
$M_{0,5}$	$M_{1,5}$	$M_{2,5}$	$M_{3,5}$	$M_{4,5}$	$M_{5,5}$

Mirroring arrangement

Figure 8. Sliding and Mirroring arrangements

1024. The computing time is evaluated for the naive arrangement and the sliding arrangement. For example, if we execute OneThreadPerEntry(64) for all stages on the naive arrangement, the computing time is 854.8 seconds. OneThreadPerEntry(64) runs in 431.8 seconds on the sliding arrangement and thus, the sliding arrangement can attain a speedup of factor 1.98.

Table I (b) shows the computing time of OneBlockPerEntry(t) for $t = 32, 64, 128, 256, 512, 1024$. Suppose that we select t that minimizes the computing time. OneBlockPerEntry(128) takes 604.7 seconds for the naive arrangement and OneBlockPerEntry(128) runs in 73.5 seconds for the mirroring arrangement. Thus, the mirroring arrangement can attain a speedup of factor 8.23.

Table I (c) shows the computing time of BlocksPerEntry(b, t) for $b = 2, 4, 8$ and $t = 32, 64, 128, 256, 512, 1024$. Again, let us select b and t that minimize the computing time. BlocksPerEntry(2,128) takes 610.9 seconds for the naive arrangement and BlocksPerEntry(2,128) runs in 97.8 seconds for the mirroring arrangement. Thus, the mirroring arrangement can attain a speedup of factor 6.25.

Figure 9 shows the running time of each stage using the three methods. For each of the three methods and for each of the 16382 stages, we select best values of the number t of threads in each block and the number b of blocks. Also, the sliding arrangement is used for OneThreadPerEntry and the mirroring arrangement is used for OneBlockPerEntry and BlocksPerEntry. Recall that we can use different methods with different parameters can be used for each stage independently. Thus, to attain the minimum computing time we should use

- OneThreadPerEntry for Stages 0-49,
- OneBlockPerEntry for Stages 50-16350, and
- BlocksPerEntry for Stages 16351-16382.

Note that if we use three methods for each stage in this way, we need to convert the sliding arrangement into the mirroring arrangement. This conversion takes only 0.21 mseconds. Including the conversion time, the best total computing time of our implementation for the optimal polygon triangulation problem is 69.1 seconds. The sequential implementation used Intel Core i7 870 runs in 17105.5 seconds. Thus, our best GPU implementation attains a speedup factor of 247.5. Recall that the computing time does not depend on edge weights shown in the above section. Therefore, for another 16384-gon whose weights are different, we can obtain almost the same speedup factor as that of the above experiment.

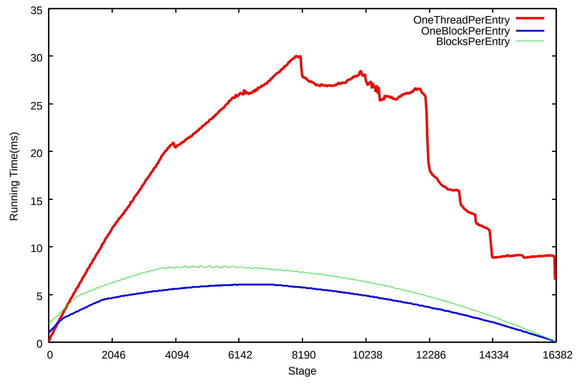


Figure 9. The running time of each stage using three methods

VI. Concluding remarks

In this paper, we have proposed an implementation of the dynamic programming algorithm for an optimal polygon triangulation on the GPU. Our implementation selects the best methods, parameters, and data arrangement for each stage to obtain the best performance. The experimental results show that our implementation solves the optimal polygon triangulation problem for a convex 16384-gon in 69.1 seconds on the NVIDIA GeForce GTX 580, while a

Table I
THE COMPUTING TIME (SECONDS) FOR A 16384-GON USING EACH OF THE THREE METHODS

(a) The computing time of OneThreadPerEntry(t)

t	32	64	128	256	512	1024
naive arrangement	596.8	854.8	863.3	889.2	1202.0	1614.2
sliding arrangement	312.8	431.8	442.2	541.0	668.3	1023.2

(b) The computing time of OneBlockPerEntry(t)

t	32	64	128	256	512	1024
naive arrangement	631.8	606.8	604.7	612.3	678.7	1286.5
mirroring arrangement	169.5	98.5	73.5	80.4	225.0	824.8

(c) The computing time of BlocksPerEntry(b, t)

t		32	64	128	256	512	1024
naive arrangement	$b = 2$	650.2	614.6	610.9	627.3	828.8	2007.8
	$b = 4$	650.5	617.5	624.9	673.1	1174.9	3585.0
	$b = 8$	655.6	630.5	670.0	815.1	1917.8	6779.5
mirroring arrangement	$b = 2$	176.3	110.8	97.8	129.1	422.6	1611.7
	$b = 4$	188.5	136.2	148.2	229.8	820.3	3188.6
	$b = 8$	216.0	189.9	250.5	433.6	1613.7	6337.9

conventional CPU implementation runs in 17105.5 seconds. Thus, our GPU implementation attains a speedup factor of 247.5.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 313–319.

[3] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *International Workshop on Advances in Networking and Computing*, Nov. 2010, pp. 279–280.

[5] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 153–159.

[6] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 320–326.

[7] *NVIDIA CUDA C Programming Guide Version 4.1*, NVIDIA Corp., 2011.

[8] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, 1st ed. MIT Press, 1990.

[10] L. Bergroth, H. Hakonen, and T. T. Raita, "A survey of longest common subsequence algorithms," in *Proc. of International Symposium on String Processing and Information Retrieval*, 2000.

[11] P. D. Gilbert, "New results on planar Triangulations," in *M.Sc. thesis*, July 1979, pp. Report R–850.

[12] G. T. Klincsek, "Minimal triangulations of polygonal domains," *Annals of Discrete Mathematics*, vol. 9, pp. 121–123, July 1980.

[13] G. Pólya, "On picture-writing," *Amer. Math. Monthly*, vol. 63, pp. 689–697, 1956.

[14] *CUDA C Best Practice Guide Version 4.1*, NVIDIA Corp., 2012.

An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem

Akihiro Uchida, Yasuaki Ito, Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima, 739–8527 Japan
{uchida, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—Graphics Processing Units (GPUs) are specialized microprocessors that accelerate graphics operations. Recent GPUs, which have many processing units connected with an off-chip global memory, can be used for general purpose parallel computation. Ant Colony Optimization (ACO) approaches have been introduced as nature-inspired heuristics to find good solutions of the Traveling Salesman Problem (TSP). In ACO approaches, a number of ants traverse the cities of the TSP to find better solutions of the TSP. The ants randomly select next visiting cities based on the probabilities determined by total amounts of their pheromone spread on routes. The main contribution of this paper is to present sophisticated and efficient implementation of one of the ACO approaches on the GPU. In our implementation, we have considered many programming issues of the GPU architecture including coalesced access of global memory, shared memory bank conflicts, etc. In particular, we present a very efficient method for random selection of next cities by a number of ants, Our new method uses iterative random trial which can find next cities in few computational costs with high probability. The experimental results on NVIDIA GeForce GTX 580 show that our implementation for 1002 cities runs in 8.71 seconds, while a conventional CPU implementation runs in 381.95 seconds. Thus, our GPU implementation attains a speed-up factor of 43.47.**

***Index Terms*—Ant Colony Optimization, Traveling Salesman Problem, GPU, CUDA, Parallel Processing**

I. Introduction

Graphics Processing Units (GPUs) are specialized microprocessors that accelerate graphics operations. Recent GPUs, which have many processing units connected with an off-chip global memory, can be used for general purpose parallel computation. CUDA (Compute Unified Device Architecture) [1] is an architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [2], [3], [4], [5], [6].

Ant colony optimization (ACO) was introduced as a nature-inspired meta-heuristic for the solution of combinatorial optimization problems [7], [8]. The idea of ACO is based on the behavior of real ants exploring a path between their colony and a source of food. More specifically, when searching for food, ants initially explore the area surrounding their nest at random. Once an ant finds a food source, it evaluates the quantity and the quality of the food and carries some of it back to the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of pheromone will guide other ants to the food source. Indirect communication between the ants via pheromone trails makes them possible to find shortest paths between their nest and food sources. In ACO, the characteristic of real ant colonies is exploited in simulated ant colonies to solve problems. The generic ACO algorithm consists of the following two steps:

Step 1: Initialization
- Initialize the pheromone trail

Step 2: Iteration
- For each ant repeat until stopping criteria
  - Construct a solution using the pheromone trail
  - Update the pheromone trail

The first step mainly consists in the initialization of the pheromone trail. In the iteration step, each ant constructs a complete solution for the problem according to a probabilistic state transition rule. The rule depends chiefly on the quantity of the pheromone. Once all ants construct solutions, the quantity of the pheromone is update in two phases: an evaporation phase in which a fraction of the pheromone evaporates, and a deposit phase in which each ant deposits an amount of pheromone that is proportional to the fitness of its solution. This process is repeated until stopping criteria.

Several variants of ACO have been proposed in the past. The typical ones of them are Ant System (AS), Max-Min Ant System (MMAS), and Ant Colony System (ACS). AS was the first ACO algorithm to be proposed [7], [8]. The characteristic is that pheromone trails is updated when all the ants have completed the tour shown in the above algorithm. MMAS is an improved algorithm over the AS [9]. The main different points are that only the best ant can update the pheromone trails and the minimum and maximum values of the pheromone are limited. Another improvement over the original AS is ACS [10]. The pheromone update, called local pheromone update, is performed during the tour construction process in addition to the end of the tour construction.

The main contribution of this paper is to implement the AS to solve the *traveling salesman problem* (TSP) [11] on the GPU. In TSP, a salesman visits n cities, and makes a tour visiting each city exactly once to try to find the shortest possible tour. We model the problem as a complete graph with n vertices that represent the cities. Let $v_0, v_1, \ldots, v_{n-1}$ be vertices that represent n cities, $e_{i,j}$ $(0 \leq i, j \leq n-1)$ denote edges between cities, and (x_i, y_i) $(0 \leq i \leq n-1)$ be the location of v_i. Let $d(i,j)$ be the distance between v_i and v_j. In this paper, we assume that the distance between two cities is their Euclidean distance. Namely, each distance between cities i and j is $d(i,j) = d(j,i) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Given a tour T, TSP is to find a tour which minimizes the objective function S:

$$S = \sum_{e_{i,j} \in T} d(i,j).$$

TSP is well known as an NP-hard problem in combinatorial optimization and utilized as a benchmark problem for various meta-heuristics such as ACO, genetic algorithm, tabu search, etc.

Many algorithms of ACO for the TSP have been proposed in the past. Mandrin *et al.* have shown a parallel algorithm of MMAS with 4 network-connected computers using MPI [12]. Delisle *et al.* have proposed an efficient and straightforward OpenMP implementation with the multi-processor system [13]. Also, GPU implementations have been proposed. In [14], a GPU implementation of MMAS is shown. Kobashi *et al.* have shown a GPU implementation of AS [15]. The implementation introduces nearest neighbor technique to reduce the computing time of tour construction. Cecilia *et al.* have proposed a GPU implementation of AS [16]. To reduce the computing time of tour construction on the GPU, instead of the ordinary roulette-wheel selection used when ants select a next city to visit, they introduced an alternative method, called *I-Roulette*. The method is similar to the roulette-wheel selection, however, it does not exactly compute the roulette-wheel selection.

In our implementation, we have considered many programming issues of the GPU architecture such as coalesced access of global memory, shared memory bank conflicts, etc. To be concrete, arranging various data in the global memory efficiently, we try to make the bandwidth of the global memory of the GPU maximized. Also, to avoid the access to the global memory as much as possible, we utilize the shared memory that is on chip memory of the GPU.

In addition, we have introduced a stochastic method, called *stochastic trial*, instead of the roulette-wheel selection that is used when ants determine the next city to visit. Using the stochastic trial, most prefix sum computation that is performed in the roulette-wheel selection can be omitted. Since the computing time of the prefix sum computation is dominated in that of the AS for TSP, we attained further speed-up of it.

Note that our goal in this paper is to accelerate the AS on the GPU, not to improve the accuracy of the solution. The solution obtained by our implementation is basically the same as that by the original AS for the TSP. We have implemented our parallel algorithm in NVIDIA GeForce GTX 580. The experimental results show that our implementation can perform the AS for 1002 cities, that repeats tour construction and pheromone update 100 times, in 8.71 seconds, while a conventional CPU implementation runs in 381.95 seconds. Thus, our GPU implementation attains a speed-up factor of 43.47 over the conventional CPU implementation.

The rest of this paper is organized as follows; Section II introduces ant colony optimization for traveling salesman problem. In Section III, we show the GPU and CUDA architectures to understand our new idea. Section IV proposes our new ideas to implement the ant colony optimization for traveling salesman problem on the GPU. The experimental results are shown in Section V. Finally, Section VI offers concluding remarks.

II. Ant Colony Optimization for the Traveling Salesman Problem

In this section, we describe a solution for TSP with ant colony optimization. Specially, we explain an algorithm solving this problem by ant system (AS). Recall that in TSP, a salesman visits n cities. and the salesman makes a tour visiting each city exactly once to try to find the shortest possible tour. In AS for TSP, ants are used as agents that perform distributed search. Each ant visits each city exactly once, ending up back at the starting city and then offers the tour as its solution. Each ant has the following characteristic:

- An ant selects which city to visit, using a transition rule that is a function of the distance to the city and the quantity of pheromone present along the connecting path.
- Transitions to already visited cities are added to a *visited list* and not allowed.
- When a tour is complete, the ant deposits a pheromone trail along paths visited in the tour.

Using the characteristic of ants, AS performs the following three steps; (i) *initialization*, (ii) *tour construction* and (iii) *pheromone update*. First of all, initialization is performed, and tour construction and pheromone update are repeated until stopping criteria. Given n cities, the distances between the cities, and m ants, the details of these three steps are spelled out as follows.

A. Initialization

In the initialization step, the initial quantities of all the pheromone trail are determined using the greedy manner [17] as follows:

$$\tau(i,j) = \frac{n}{C_g} \quad \forall (i,j) \in L, \tag{1}$$

where L denotes all edges between cities and C_g is the total length of a tour obtained by the greedy algorithm such that starting from an arbitrary city as current city, the shortest edge that connects current city and an unvisited city is selected. The quantities of pheromone assigned to each edge between two cities are initially set to a reciprocal of the average of C_g.

B. Tour construction

In tour constriction, m ants independently visit each city exactly once. Each ant starts at a city decided randomly, and selects which city to visit probabilistically. A probability

$p_k(i,j)$ to visit city j from city i for ant k is computed by Eq. (2).

$$p_k(i,j) = \begin{cases} \frac{f(i,j)}{\sum_{l \in N_k(i)} f(i,l)} & \texttt{if } j \in N_k(i) \\ 0 & \texttt{otherwise}, \end{cases} \tag{2}$$

where $N_k(i)$ is a set of unvisited adjacent cities for ant k in city i, and $f(i,j)$ is a fitness between cities i and j

$$f(i,j) = [\tau(i,j)]^\alpha [\eta(i,j)]^\beta, \tag{3}$$

where $\tau(i,j)$ denotes a quantity of pheromone between cities i and j, $\eta(i,j)$ represents heuristic information which is a reciprocal of the distance between cities i and j, and α and β control the relative influence of pheromone versus distance. These equations mean that when the quantity of pheromone between cities i and j is large and the distance between cities i and j is short, the probability to visit city j becomes large. Using this probability, each ant visits each city exactly once, ending up back at the starting city. The method such that ants select which city to visit using the above probability is well-known as *roulette-wheel selection* [18]. Visiting cities with the roulette-wheel selection, each ant constructs a tour.

C. Pheromone update

When all the ants complete tour construction, the pheromone assigned between cities is updated using information of each tour. The update consists of pheromone evaporation and pheromone deposit.

Pheromone evaporation is utilized to avoid falling into local optima. Every quantity of pheromone is reduced with the following equation;

$$\tau(i,j) \leftarrow (1-\rho)\tau(i,j) \quad \forall (i,j) \in L, \tag{4}$$

where ρ is an evaporation rate of pheromone.

After the pheromone evaporation, for every pheromone between cities, pheromone deposit is performed with the results of the tour construction as follows;

$$\tau(i,j) \leftarrow \tau(i,j) + \sum_{k=1}^{m} \Delta\tau_k(i,j) \quad \forall (i,j) \in L, \tag{5}$$

where $\Delta\tau_k(i,j)$ is a quantity of pheromone between cities i and j which is deposited by ant k. The quantity is computed by

$$\Delta\tau_k(i,j) = \begin{cases} \frac{1}{C_k} & \texttt{if } e_{i,j} \in T_k \\ 0 & \texttt{otherwise}, \end{cases} \tag{6}$$

where C_k is the tour length of ant k, and T_k is the tour of ant k. This equation means that when an edge is included in shorter tours and is selected by more ants in the tour construction, the quantity of additional pheromone is larger.

III. Compute Unified Device Architecture (CUDA)

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [19]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [20], [6]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory. Figure 1 illustrates the CUDA hardware architecture.

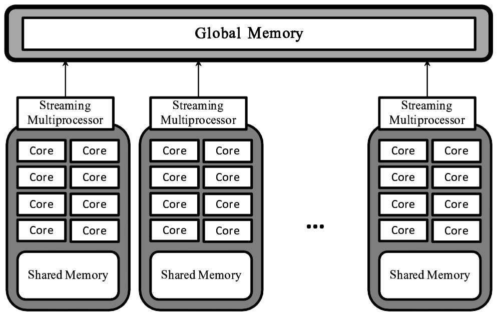


Fig. 1. CUDA hardware architecture

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, as we can see in Figure 1, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories.

CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each block are executed by processor cores in a single streaming processor. In the execution, threads in a block are split into groups of thread called *warps*. Each of these warps contains the same number of threads and is execute independently. When a warp is selected for execution, all threads execute the same instruction. When one warp is paused or stalled, other warps can be executed to hide latencies and keep the hardware busy.

As we have mentioned, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 2, when threads access to continuous locations in a row of a 2-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed in the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed in the same time (*stride access*). From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus, we should avoid the stride access (or the vertical access) and perform the coalesced access (or the horizontal access) whenever possible.

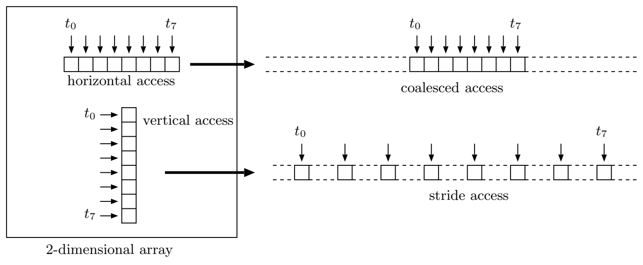


Fig. 2. Coalesced and stride access

Just as the global memory is divided into several partitions, shared memory is also divided into 16 (or 32) equally-sized modules of 32-bit width, called banks (Figure 3). In the shared memory, the successive 32-bit words are assigned to successive banks. To achieve maximum throughput, concurrent threads of a thread block should access different banks, otherwise, bank conflicts will occur. In practice, the shared memory can be used as a cache to hide the access latency of the global memory.

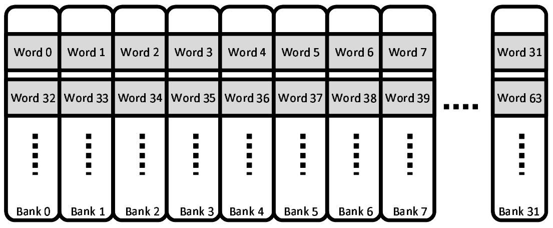


Fig. 3. The structure of the shared memory

IV. GPU Implementation

The main purpose of this section is to show a GPU implementation of AS for TSP. The Ideas of our implementation to consider programming issues of the GPU system such as coalesced access of global memory and shared memory bank conflicts in Section III. Given n coordinates (x_i, y_i) of city i $(0 \leq i \leq n-1)$, our implementation computes the shortest possible route that visits each city once and returns to the origin city. Our implementation consists of three CUDA parts, initialization, tour construction, and pheromone update. We describe the details of them as follows.

A. Initialization

This part is an initialization process for the followings. Given n coordinates (x_i, y_i) of city i, each distance $d(i,j)$ between cities i and j and initial values of pheromone $\tau_{i,j}$ in Eq. (1) are computed. Also, initializing random seeds for CURAND used in the following process is performed. CURAND is a library that provides a pseudorandom number generator on the GPU by NVIDIA [21].

B. Tour construction

Recall that in the tour constriction, m ants are initially positioned on n cities chosen randomly. Each ant makes a tour with roulette-wheel selection independently. Whenever each ant visits a city, it determines which city to visit with roulette-wheel selection. To perform the tour construction on the GPU, we consider four methods, *SelectionWithoutCompression*, *SelectionWithCompression*, *SelectionWithStochasticTrial*, and a hybrid method that is a combination of the above methods. Let us consider the case when ant k is in city i. In advance, the fitness values $f(i,j)$ $(0 \leq i,j \leq n-1)$ are computed by Eq. (3) and stored to the 2-dimensional array in the global memory. Also, the elements related to city i, i.e., $f(i,0),\ldots,f(i,n-1)$, are stores in the same row so that the access to the elements can be performed with coalesced access. In the tour construction, ant k $(0 \leq k \leq m-1)$ makes a tour index array t_k such that element $t_k(i)$ stores the index of the next city from city i shown in Figure 4.

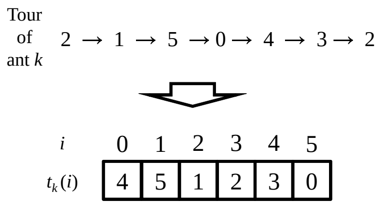


Fig. 4. Representation of tour list

1) *SelectionWithoutCompression:*

Each ant has an unvisited list $u_0, u_1, \ldots, u_{n-1}$ such that

$$u_j = \begin{cases} 0 & \text{if city } j \text{ has been visited} \\ 1 & \text{otherwise.} \end{cases} \tag{7}$$

To perform the roulette-wheel selection, when ant k is in city i. we compute as follows;

Step 1: Calculate the prefix sums $q_j (0 \leq j \leq n-1)$ of the fitness values for adjacent cities and a sentinel q_{-1} such that

$$q_j = \begin{cases} \sum_{s=0}^{j} f(i,s) \cdot u_j & 0 \leq j \leq n-1 \\ 0 & j = -1. \end{cases} \tag{8}$$

Step 2: Generate a random number r in $[0, q_{n-1}]$.

Step 3: Find j such that $q_{j-1} < r \leq q_j$ $(0 \leq j \leq n-1)$. City j is selected as the next city.

Figure 5 shows a summary of SelectionWithoutCompression. In Step 1, values $\tau(i,j)$ and $\eta(i,j)$ $(0 \leq j \leq n-1)$ to

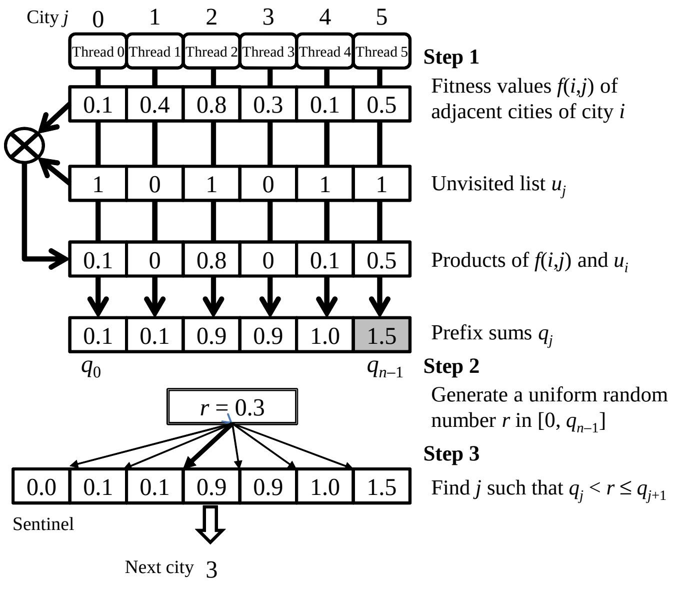

City j
0 1 2 3 4 5
Thread 0 Thread 1 Thread 2 Thread 3 Thread 4 Thread 5
Step 1
Fitness values $f(i,j)$ of adjacent cities of city i
0.1 0.4 0.8 0.3 0.1 0.5
Unvisited list u_j
1 0 1 0 1 1
Products of $f(i,j)$ and u_i
0.1 0 0.8 0 0.1 0.5
Prefix sums q_j
0.1 0.1 0.9 0.9 1.0 1.5
q_0
q_{n-1}
Step 2
Generate a uniform random number r in $[0, q_{n-1}]$
$r = 0.3$
Step 3
Find j such that $q_j < r \leq q_{j+1}$
0.0 0.1 0.1 0.9 0.9 1.0 1.5
Sentinel
Next city 3

Fig. 5. Parallel roulette-wheel selection in SelectionWithoutCompression

compute the fitness function in Eq. (3) are read from the global memory by threads with coalesced access and stored to the shared memory. After that, prefix sums in Eq. (8) are computed, where the fitness values of visited cities are 0 not to be selected. To avoid the branch instruction whether the candidate of the next cities has been visited or not, we multiply $f(i,j)$ and u_j with the unvisited list in Eq. (7). In our implementation, the prefix sum computation is performed using the parallel prefix sum algorithm proposed by Harris *et al.* [22], Chapter 39. It is an in-place parallel prefix sum algorithm with the shared memory on the GPU. Also, it can avoid most bank conflicts by adding a variable amount of padding to each shared memory array index. On the other hand, this method has a fault that the number of elements that it can perform must be power of two. Therefore, when the number of elements is a little more than power of two numbers, the efficiency is decreased. For example, if the number of elements is 4097, the method must perform for 8192 elements. This fault can be ignored for small number of elements. However, it cannot be ignored for large number. After that, a uniform random number r in $[0, q_{n-1}]$ is generated with by CURAND. Using the random number by CURAND, an index j such that $q_{j-1} < r \leq q_j$ is searched and city j is the next city to visit. In the search, we use a parallel search method based on the parallel K-ary search [23]. The idea of the parallel K-ary search is that a search space in each iteration is divided into K partitions and the search space is reduced to one of the partitions. In general, Binary search is a special case $(K = 2)$ of K-ary search. In our parallel search method, we divide the search space into 32 partitions. Sampling the first elements of each partition, a partition that includes the objective element to search is found by 32 threads, i.e., 1 warp. After that the objective element is searched from the partition by threads whose number is the number of elements in the partition.

The feature of this method is that the fitness values can be read from the global memory with coalesced access. Although the number of unvisited cities is smaller, in every selection to determine the next city to visit, the roulette-wheel selection has to be performed for both visited and unvisited cities. Namely, the data related to both of the visited and unvisited cities is necessary When the number of unvisited cities is smaller, computing time is not reduced. In other words, it does not depend on the number of visited cities.

2) SelectionWithCompression: The idea of this method is to select only from unvisited cities excluding the visited cities. Instead of the unvisited list in the above method, we use an *unvisited index array* that stores indexes of unvisited cities. When the number of unvisited cities is n', The array consists of elements $v_0, v_1, \ldots, v_{n'-1}$ and each element stores an index of one of the unvisited cities. When a city is visited, the city has to be removed from the index array. The removing operation takes $O(1)$ time by overwriting the index of the next city with that of the last element, then removing the last element (Figure 6). Using the index array of unvisited cities, it is not necessary to read the data related to the visited cities to compute the prefix sums in Eq. (8) though SelectionWithoutCompression requires data related to both visited and unvisited cities. Therefore, when the number of unvisited cities is smaller, the computing time becomes shorter. However, the global memory access necessary to compute the prefix sums may not be done with coalesced access because the contents of the index array are out of order using the above array update. Therefore, when the number of unvisited cities is large, computing time of SelectionWithCompression is perhaps slower than that of SelectionWithoutCompression.

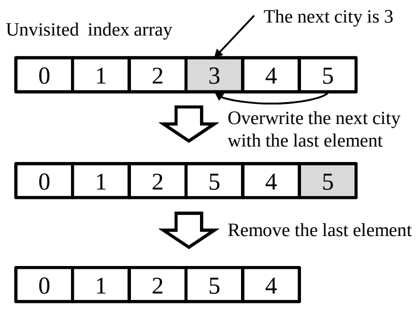


Fig. 6. Update of the unvisited index array when city 3 is selected as the next city.

3) SelectionWithStochasticTrial: In the above two methods, whenever each ant visits a city, the prefix sum calculation has to be performed. The prefix sum calculation occupies the most of the computing time of the tour construction. The idea of this method is to avoid the prefix sum calculation as much as possible using *stochastic trial*. The details of the stochastic trial are shown as follows.

Before ants start visiting cities, the prefix sums for each city are calculated such that

$$q'(i,j) = \begin{cases} \sum_{s=0}^{j} f(i,s) & 0 \leq i,j \leq n-1 \\ 0 & j=-1, \end{cases} \tag{9}$$

where all the cities have been unvisited, i.e., $u_j = 1$ $(0 \leq j \leq n-1)$ in Eq. (8). The results are stored to the 2-dimensional array in the global memory such that the prefix sums for city i to each city, $q'(i,0), \ldots, q'(i,n-1)$, are stored to the same row to be read with coalesced access. When an ant is in city i, to select the next city, the following steps are repeated until the next city is determined or the number of the iteration exceeds w.

Step 1: Generate a random number r in $[0, q'(i,n-1)]$.

Step 2: Find j such that $q'(i,j-1) < r \leq q'(i,j)$ $(0 \leq j \leq n-1)$. If city j is unvisited, it is selected as the next city. If not, these steps are performed again.

In Step 2, the unvisited list (Eq. (7)) is used to find whether the city has been visited or not by the parallel search shown in the above methods. If the next city is not determined after the w-time iteration, the next city is selected by SelectionWithoutCompression. These steps are similar to the roulette-wheel selection in the above methods. The difference point is that it is not always to determine the next city since a candidate of the next city found by the random selection may have been visited. In followings, the above operation is called *stochastic trial*. SelectionWithStochasticTrial repeats the stochastic trial at most w times. If the next city cannot be determined, it is selected by SelectionWithoutCompression. When the number of unvisited city is smaller or some of the fitness values of visited cities are larger, almost the trial cannot select the next city. However, the computing time is much shorter than that of the prefix sum calculation. Therefore, if the next city can be determined in the above steps within w times, the total computing time can be reduced by this method. It is important for this method to determine w. This is because w has to be determined considering the balance between the computing time of the iteration of the stochastic trial and that of SelectionWithoutCompression performed when the next city cannot be determined. In Section V, we will obtain the optimal times w by experiments.

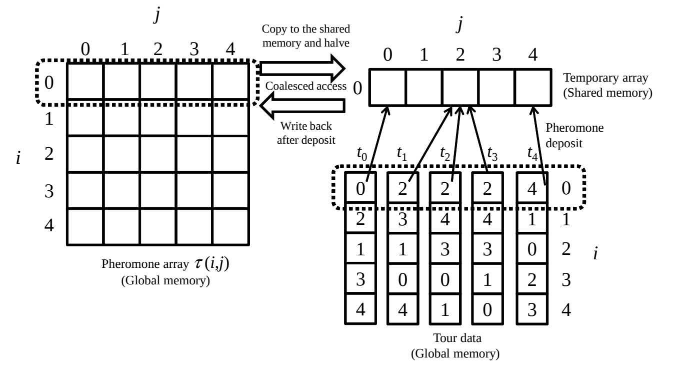
4) Hybrid Method: In SelectionWithStochasticTrial, however, when the number of visited cities is large, the next city may not be determined by the stochastic trial and has to be selected by SelectionWithoutCompression. Therefore, we introduce a hybrid method such that when the number of visited city is small, SelectionWithStochasticTrial is performed. Then, SelectionWithStochasticTrial is switched to SelectionWithoutCompression. After that the next city is determined by SelectionWithCompression until all the cities are visited. The reason that SelectionWithCompression is performed after SelectionWithStochasticTrial is that when the number of unvisited cities is small, SelectionWithCompression is performed faster than SelectionWithoutCompression. In the followings, we call such method *hybrid method*. An important point of this hybrid method is to determine the timing when SelectionWithStochasticTrial is switched such that the computing time is minimized. In Section V, we will obtain the optimal timing by experiments.

C. Pheromone update

In the followings, we show a GPU implementation of pheromone update. Recall that pheromone update consists of pheromone evaporation and pheromone deposit. In our implementation, the values of pheromone $\tau(i,j)$ $(0 \leq i \leq j \leq n-1)$ are stored in a 2-dimensional array, which is a symmetric array, that is, $\tau(i,j) = \tau(j,i)$, in the global memory and are updated by the results of the tour construction. Making the array symmetric, the elements related to city i, i.e., $\tau(i,0), \tau(i,1), \ldots, \tau(i,n-1)$, are stores in the same row so that the access to the elements can be performed with coalesced access. Our implementation consists of two kernels, *PheromoneUpdateKernel* and *SymmetrizeKernel*.

1) PheromoneUpdateKernel: This kernel assigns n blocks that consist of multiple threads to each row of the array and each block performs the followings independently. Figure 7 shows a summary of the pheromone update on the GPU for a block that perform pheromone update for city 0. Threads in block i read $\tau(i,0), \tau(i,1), \ldots, \tau(i,n-1)$ in the i-th row with coalesced access, and then store them to the shared memory. When the values are stored to the shared memory, each value is halved in advance since they are doubled in the following kernel, SymmetrizeKernel. After that, pheromone evaporation is preformed, i.e., each value is reduced by Eq. (4) by threads in parallel. To perform pheromone deposit, block i reads the values $t_0(i), t_1(i), \ldots, t_{m-1}(i)$ in the i-th row of the tour lists. The read operation is performed with coalesced access by threads. Also, each total tour length of each ant $C_0, C_1, \ldots, C_{m-1}$ stored in the global memory is read. After that threads add a quantity obtained by Eq. (6) to the corresponding values of pheromone in parallel. In the addition, some threads may add to the same pheromone simultaneously. To avoid it, we use the atomic add operation supported by CUDA [19]. After the addition, the values of pheromone are stored back to the global memory. Note that since the 2-dimensional array that stores the pheromone values are symmetry, if addition to $\tau(i,j)$ is performed, that to $\tau(j,i)$ has to be also performed. However, the above deposit operation adds to either $\tau(i,j)$ or $\tau(j,i)$. To obtain the correct results, SymmetrizeKernel is performed.

2) SymmetrizeKernel: This kernel symmetrizes the array for the results of PheromoneUpdateKernel. More specifically, summing corresponding two elements that are symmetric, each value of symmetric elements is made identical. In this kernel, to make the access to the global memory coalesced, the 2-dimensional array that stores pheromone values is divided into subarrays whose size is 32×32. We assign one block to two subarrays that are symmetric or one subarray that includes symmetric element. Blocks symmetrize the whole array subarray by subarray. To symmetrize the subarrays, one array has to be transposed. For the transposing, we utilize an efficient method proposed in [24]. The method transposes a 2-dimensional data stored in the global memory via the shared



j
Copy to the shared memory and halve
Coalesced access
Write back after deposit
Temporary array (Shared memory)
Pheromone deposit
i
Pheromone array τ(i,j) (Global memory)
Tour data (Global memory)
t_0 t_1 t_2 t_3 t_4

Fig. 7. A summary of PheromoneUpdateKernel

memory with coalesced access and avoidance of bank conflict on the GPU. Note that when the symmetrization is performed, each value is doubled since the original values are added twice. Therefore they are halved in advance in the previous kernel, PheromoneUpdateKernel.

V. Performance Evaluation

We have implemented our AS for the TSP using CUDA C. We have used NVIDIA GeForce GTX580 with 512 processing cores (16 Streaming Multiprocessors which has 32 processing cores) running in 1.544GHz and 3GB memory. For the purpose of estimating the speed up of our GPU implementation, we have also implemented a conventional software approach of AS for the TSP using GNU C. We have used Intel Core i7 860 running in 2.8GHz and 3GB memory to run the sequential algorithm for the AS. We have evaluated our implementation using a set of benchmark instances from the TSPLIB library [25]. In the following evaluation, we utilize 8 instances: *d198, a280, lin318, pcb442, rat783, pr1002,* and *pr2392* from TSBLIB. Each name consists of the name of the instance and the number of cities. For example, *pr1002* means that the name of the instance is *pr* and the number of cities is 1002. The parameters of ACO, α, β, and ρ in Eq. (3) and Eq. (4), are set to 1.0, 2.0, and 0.5, respectively. Also, the number of used ants m is set to the number of cities. Those parameters are recommended in [26]. In CUDA, it is important to determine the number of blocks and the number of threads in each block. It greatly influences the performance of the implementation on the GPU. In the followings, we select the optimal numbers obtained by experiments. We first explain the performance of tour construction and pheromone update, and then the results of overall performance are shown.

A. Evaluation of tour construction

Before performance of tour construction is evaluated, we determine the optimal parameters. One is an upper limit of times of iteration how many times the stochastic trial is repeated if the next city is not determined in SelectionWithStochasticTrial. The other is timing when SelectionWithStochasticTrial is switched to SelectionWithCompression in the hybrid method.

To obtain an optimal upper limit of iteration of the stochastic trial if the next city is not determined in SelectionWithStochasticTrial, we evaluated the number of times necessary to determine the next city in a tour construction for the pheromone values obtained after the tour construction and pheromone update were repeated 100 times for pr1002. Figure 8 is a graph that shows a histogram of the number of city and its cumulative histogram of the percentage of cities to the number of times of iteration how many times the stochastic trial is repeated. For example, when the number of times of iteration is 5, the number of cities is about 25 and the percentage of cities is about 84%. This means that in about 500 cities, the next city was determined by the stochastic trial 5 times and in 84% cities, it was determined by the stochastic trial within 5 times. From the figure, in approximately half of cities, the next city can be determined by the stochastic trial one time. Also, in about 90% cities, the next city can be selected within about 32 times. In several cities, the next city cannot be determined when the stochastic trial has to be repeated more than 2000 times. Considering the balance of computing time between the stochastic trial and SelectionWithoutCompression when the next city cannot be determined, in the following experiments, we set 8 times to the upper limit of times of iteration.

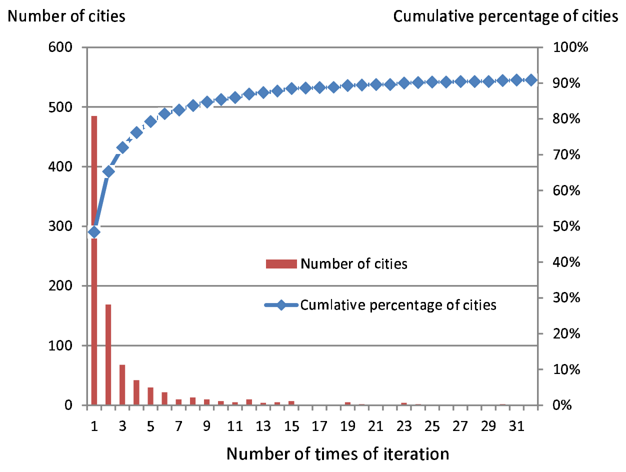


Fig. 8. A histogram of the number of city and its cumulative histogram of the percentage of cities to the number of times of iteration of the stochastic trial

To obtain the timing when SelectionWithStochasticTrial is switched to SelectionWithCompression in the hybrid method, we measured the computing time of tour construction for various percentages when SelectionWithStochasticTrial is switched to SelectionWithCompression. Figure 9 shows the computing time of tour construction for various instances. According to the figure, the percentage of the visited cities is larger, the computing time is shorter and if it is closed to 100%, it becomes larger. According to Figure 9, computing time is minimized by switching the method when about 85% cities are visited. Therefore, we switch from SelectionWithStochasticTrial to SelectionWithCompression when 85% cities are visited.

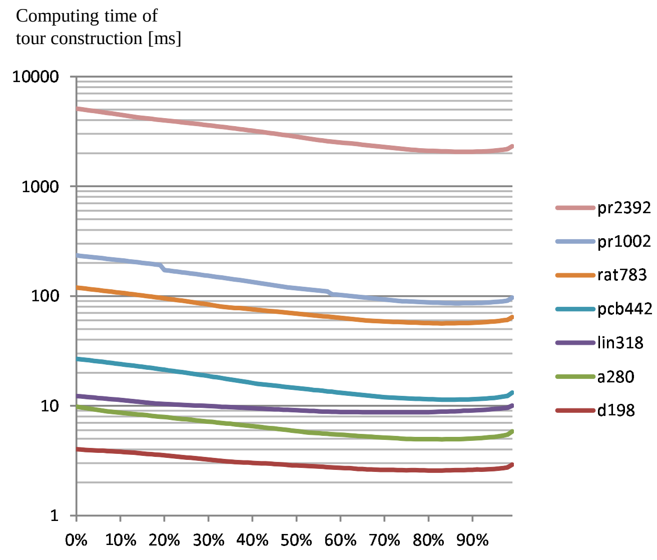


Fig. 9. Computing time of tour construction for the various percentages when SelectionWithStochasticTrial is switched to SelectionWithoutCompression for pr1002 with various w

To compare the performance among our proposed tour construction methods, we have evaluated the computing time of them. Table I shows the computing time of tour construction with various methods for pr1002. The computing time of SelectWithCompression is a little shorter than that of SelectWithoutCompression. Since the computing time of SelectWithCompression becomes shorter when the number of unvisited cities is small, the total computing time becomes shorter. Compared to the methods without the stochastic trial, the computing time of the methods with the stochastic trial is approximately halved. In addition, the computing time of the hybrid method is approximately 10% shorter than that of SelectWithStochasticTrial.

TABLE I
COMPUTING TIME OF TOUR CONSTRUCTION FOR PR1002

Tour construction method	Time[ms]
SelectWithoutCompression	235.43
SelectWithCompression	217.94
SelectWithStochasticTrial	96.37
Hybrid method	86.43

Table II shows the computing time of tour construction for various instances. From 198 to 1002 cities, when the number of cities is larger, the speed-up factor is larger. However, The speed-up factor for 2392 cities is smaller than that for 1002 cities. This is because in the parallel prefix sum computation shown in Section IV can be performed only for power of two numbers. Therefore, for the instance of which number of cities is 2392, the parallel prefix sum computation for 4096 elements must be performed. Therefore, approximate half of elements are redundant. Since in CPU implementation, the redundant elements are not necessary to compute the prefix sum, the computing time of GPU implementation becomes longer, that is, the speed-up factor becomes smaller.

TABLE II
COMPUTING TIME OF TOUR CONSTRUCTION FOR VARIOUS INSTANCES

Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	19.84	2.58	7.69
a280 (280)	47.05	4.95	9.50
lin318 (318)	95.15	8.86	10.74
pcb442 (442)	180.61	11.35	15.92
rat783 (783)	1215.31	56.38	21.56
pr1002 (1002)	3784.43	86.43	43.79
pr2392 (2392)	58452.20	2078.98	28.12

B. Evaluation of pheromone update

Table III shows the computing time of pheromone update for various instances. The computing time of both the CPU and GPU implementation is Our GPU implementation can achieve speed-up factors of 22 to 67. Compared to the computing time of tour construction, the computing time of pheromone update is much shorter.

TABLE III
COMPUTING TIME OF PHEROMONE UPDATE

Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	0.963	0.036	26.64
a280 (280)	1.384	0.060	22.92
lin318 (318)	2.797	0.070	39.88
pcb442 (442)	4.692	0.113	41.43
rat783 (783)	16.770	0.320	52.37
pr1002 (1002)	34.877	0.520	67.08
pr2392 (2392)	222.762	5.411	41.17

C. Evaluation of overall performance

Table IV shows overall performance that is the total computing time of AS for various instances. Each execution includes the initialization and 100 times iteration of tour construction and pheromone update. Since the computing time of tour construction is much larger than other process, each speed-up factor is similar to that of tour construction. Our GPU implementation can achieve speed-up factors of 7.52 to 43.47 over the CPU implementation.

In the related works of ACO for TSP shown in Section I, several GPU implementations have been proposed. Since those implemented methods, used instance, and utilized GPUs differ, we cannot directly compare our implementation with them. However, GPU implementations proposed in papers [14], [15], [16] achieved the maximum speed-up factor of 23.9, 23.5, and 20.0 over their CPU implementations, respectively. Since the speed-up factor we achieved is 43.47, our GPU implementation is more effective than them.

TABLE IV
TOTAL COMPUTING TIME OF OUR IMPLEMENTATION WHEN TOUR CONSTRICTION AND PHEROMONE UPDATE ARE REPEATED 100 TIMES

Instance (# cities)	CPU[ms]	GPU[ms]	Speed-up
d198 (198)	2080.72	263.91	7.52
a280 (280)	4844.59	505.51	9.31
lin318 (318)	9797.03	897.29	10.61
pcb442 (442)	18534.37	1153.95	15.66
rat783 (783)	123220.58	5673.15	21.43
pr1002 (1002)	381949.72	8706.32	43.47
pr2392 (2392)	5867605.87	208478.18	28.04

VI. CONCLUSIONS

In this paper, we have proposed an implementation of the ant colony optimization algorithm, especially AS, for the traveling salesman problem on the GPU. In our implementation, we have considered many programming issues of the GPU architecture such as coalesced access of global memory and shared memory bank conflicts. In addition, we have introduced a method with the stochastic trial in the roulette-wheel selection. We have implemented our parallel algorithm in NVIDIA GeForce GTX 580. The experimental results show that our implementation can perform the AS for 1002 cities, that repeats tour construction and pheromone update 100 times, in 8.71 seconds, while a conventional CPU implementation runs in 381.95 seconds. Thus, our GPU implementation attains a speed-up factor of 43.47.

Future works include GPU implementations for various algorithms of ant colony optimization such as MMAS, ACS, and AS with the idea of nearest neighbor to obtain further acceleration and accuracy. In addition to TSP, other combinatorial optimization problems such as the quadratic assignment problem, etc. are applied by our method.

REFERENCES

[1] NVIDIA Corp., "CUDA ZONE," http://developer.nvidia.com/category/zone/cuda-zone.

[2] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proceedings of International Workshop on Advances in Networking and Computing*, 2010, pp. 279–280.

[3] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," in *Proceedings of International Workshop on Challenges on Massively Parallel Processors*, 2011, pp. 313–319.

[4] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proceedings of International Conference on Networking and Computing*, 2011, pp. 153–159.

[5] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proceedings of International Workshop on Challenges on Massively Parallel Processors*, 2011, pp. 320–326.

[6] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, 2011.

[7] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, 1992.

[8] M. Dorigo, V. Maniezzo, and A. Colorni, "The ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics–Part B*, vol. 26, no. 1, pp. 29–41, 1996.

[9] T. Stützle and H. H. Hoos, "MAX–MIN ant system," *Future Generation Computer Systems*, vol. 16, no. 8, pp. 889–914, 2000.

[10] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, April 1997.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[12] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel ant colony optimization for the traveling salesman problem," in *Proc. of 5th International Workshop on Ant Colony Optimization and Swarm Intelligence*, vol. LNCS 4150. Springer-Verlag, 2006, pp. 224–234.

[13] P. Delisle, M. Krahecki, M. Gravel, and C. Gagné, "Parrallel implementation of an ant colony optimization metaheuristic with OpenMP," in *Proc. of the 3rd European Workshop on OpenMP*, 2001.

[14] A. Delévacq, P. Delisle, M. Gravel, and M. Krahecki, "Parallel ant colony optimization on graphics processing units," in *Proc. of the International Conference on Parallel Distributed Processing Techniques and Applications*, 2010, pp. 196–202.

[15] K. Kobashi, A. Fujii, T. Tanaka, and K. Miyoshi, "Acceleration of ant colony optimization for the traveling salesman problem on a GPU," in *Proc. of the IASTED International Conference Parallel and Distribited Computing and Systems*, December 2011, pp. 108–115.

[16] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón, "Enhancing data parallelism for Ant Colony Optimization on GPUs," *Journal of Parallel and Distributed Computing*, to appear, 2012.

[17] G. Gutin, A. Yeo, and A. Zverovich, "Traveling salesman shoud not be greedy: domination analysis of greedy-type heuristics for the TSP," *Discrete Applied Mathematics*, vol. 117, pp. 81–86, 2002.

[18] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Machanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, March 2011.

[19] NVIDIA Corp., *NVIDIA CUDA Programming Guide Version 4.1*, 2011.

[20] ——, *CUDA C Best Practice Guide Version 4.1*, 2012.

[21] ——, *CUDA Toolkit 4.1 CURAND Guide*, 2011.

[22] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[23] B. Schlegel, R. Gemulla, and W. Lehner, "k-ary search on modern processors," in *Proc. of the Fifth International Workshop on Data Management on New Hardware*, 2009, pp. 52–60.

[24] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of parallel computation of Euclidean distance map in multicore processors and GPUs," in *Proc. of International Conference on Networking and Computing*, 2010, pp. 120–127.

[25] G. Reinelt, "TSPLIB–a traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, pp. 376–384, 1991.

[26] M. Dorigo and T. Stützle, *Ant Colony Optimization*. A Bradford Book, 2004.

Accelerating Computation of Euclidean Distance Map using the GPU with Efficient Memory Access

Duhu Man, Kenji Uda, Yasuaki Ito and Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashi Hiroshima, 739–8527 Japan

***Abstract*—Recent Graphics Processing Units (GPUs), which have many processing units, can be used for general purpose parallel computation. To utilize the powerful computing ability, GPUs are widely used for general purpose processing. Since GPUs have very high memory bandwidth, the performance of GPUs greatly depends on memory access. The main contribution of this paper is to present a GPU implementation of computing Euclidean Distance Map (EDM) with efficient memory access. Given a 2-dimensional binary image, EDM is a 2-dimensional array of the same size such that each element is storing the Euclidean distance to the nearest black pixel. In the proposed GPU implementation, we have considered many programming issues of the GPU system such as coalesced access of global memory and shared memory bank conflicts. To be concrete, transposing 2-dimensional arrays, which are temporal data stored in the global memory, with the shared memory, the main access from/to the global memory enables to be performed by coalesced access. In practice, we have implemented our parallel algorithm in the following three modern GPU systems: Tesla C1060, GTX 480 and GTX 580, respectively. The experimental results have shown that, for an input binary image with size of 9216×9216, our implementation can achieve a speedup factor of 54 over the sequential algorithm implementation.**

***Keywords*-Euclidean distance map; proximate points; GPU; coalesced memory access; bank conflict; CUDA**

I. INTRODUCTION

Recent Graphics Processing Units (GPUs), which have a lot of processing units, can be used for general purpose parallel computation. Since GPUs have very high memory bandwidth, the performance of GPUs greatly depends on memory access. CUDA (Compute Unified Device Architecture) [1] is the architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [2], [3], [4], [5], [6], [7], [8], [9], [10].

In many applications of image processing such as blurring effects, skeletonizing and matching, it is essential to measure distances between featured pixels and nonfeatured pixels. For a 2-dimensional binary image with size of $n \times n$, treating black pixels as featured pixels, Euclidean Distance Map (EDM) assigns each pixel with the distance to the nearest black pixel using Euclidean distance as underlying distance metric. We refer readers to Figure 1 for an illustration of Euclidean Distance Map. Assuming that the points p and q of the plane are represented by their Cartesian coordinates $(x(p), y(p))$ and $(x(q), y(q))$, as usual, we denote the Euclidean distance between the points p and q by $d(p,q) = \sqrt{(x(p)-x(q))^2 + (y(p)-y(q))^2}$.

$\sqrt{10}$	3	$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	3	$\sqrt{10}$
$\sqrt{5}$	2	$\sqrt{5}$	2	1	0	1	2	2	$\sqrt{5}$
$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2}$	1	$\sqrt{2}$
1	0	1	2	2	2	2	1	0	1
$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$
$\sqrt{5}$	2	2	1	0	1	2	$\sqrt{5}$	2	$\sqrt{5}$
$\sqrt{10}$	$\sqrt{8}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	3	3	$\sqrt{10}$
$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$	$\sqrt{5}$	2	$\sqrt{5}$	$\sqrt{8}$
3	2	1	0	1	2	$\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{5}$
$\sqrt{10}$	$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$	2	1	0	1	2

Figure 1. Euclidean Distance Map

As is known to us, the computing time is an important issue in the real-time image processing, especially for images with large size. For example, the real-time image processing is the main part of many industrial applications such as the vision-guided robot bin-picking system etc. Actually the vision-guided robot bin-picking is one of the systems with highest interest of the industry. In order to positioning bins precisely, bins with markers can be used. Especially, circle markers are used for robot vision [11] since a circle must be seen as an ellipse from any angle. Thus, a fast and reliable ellipse detection algorithm is needed. The Euclidean distance transform can be used for the evaluation of the estimated ellipses in real time [12]. Therefore we also need a faster algorithm to implement the Euclidean distance transform.

Many algorithms for computing EDM have been proposed in the past, such as sequential algorithm [13], [14], [15], [16] and parallel algorithm [17], [18], [19]. Breu *et al.* [13] and Chen *et al.* [14], [15] have presented $O(n^2)$-time sequential algorithm for computing Euclidean Distance Map. Since all pixels must be read at least once, these sequential algorithms with time complexity of $O(n^2)$ is optimal. Since in any EDM algorithm, each of the n^2 pixels has to be scanned at least once. Roughly at the same time, Hirata [16] presented a simpler $O(n^2)$-time sequential algorithm to compute the distance map for various distance metrics including Euclidean, four-neighbor, eight-neighbor, chamfer, and octagonal. On the other hand, for accelerating sequential ones, numerous parallel EDM algorithms have been developed for various parallel model. Lee *et al.* [20] presented an $O(\log^2 n)$-time algorithm using n^2 processors on the EREW PRAM. Pavel and Akl [19] presented an algorithm running in $O(\log n)$ time and using n^2 processors on the EREW PRAM. Clearly, these two algorithms are not work-optimal. Fujiwara *et al.* [17] have presented a work-optimal algorithm running in $O(\log n)$ time using $\frac{n^2}{\log n}$ $EREW$ processors and in $O(\frac{\log n}{\log\log n})$ time using $\frac{n^2 \log\log n}{\log n}$ $CRCW$ processors. Later, Hayashi *et al.* [18] have exhibited a more efficient algorithm running in $O(\log n)$ time using $\frac{n^2}{\log n}$ processors on the EREW PRAM and in $O(\log\log n)$ time using $\frac{n^2}{\log\log n}$ processors on the PRAM. Since the product of the computing time and the number of processors is $O(n^2)$ these algorithms are work optimal. Also, it was proved that the computing time cannot be improved as long as work optimality is satisfied, these algorithms are also work optimal. Thus, these algorithms are work-time optimal. Recently, Chen *et al.* [21] have proposed two parallel algorithms for EDM on Linear Array with Reconfigurable Pipeline Bus System [22]. Their first algorithm can computes EDM in $O(\frac{\log n \log\log n}{\log\log\log n})$ time using n^2 processors and the second algorithm can compute EDM in $O(\log n \log\log n)$ time using $\frac{n^2}{\log\log n}$ processors.

In practice, now many applications have employed emerging GPUs (Graphics Processing Unit) as real platforms to achieve an efficient acceleration. In GPU implementation, there are some programming issues of the GPU system such as coalesced access of global memory and shared memory bank conflicts [23]. Coalesced access is necessary to hide the access latency of the global memory. When sequential threads access sequential and aligned values in the off-chip global memory, the GPU will automatically combine them into a single transaction. An on-chip shared memory is divided into 16 or 32 equally-sized modules of 32-bit width, called banks. In the on-chip shared memory, the successive 32-bit words are assigned to successive banks. To avoid bank conflicts and achieve maximum throughput, concurrent threads should access different banks.

In our previous paper [5], we have shown an optimal parallel algorithm for computing Euclidean Distance Map (EDM) of a 2-dimensional binary image. Using proximate points problem as preliminary foundation, we have proposed a simple but efficient parallel EDM algorithm which can achieve $O(\frac{n^2}{k})$ time using k processors. To evaluate the performance of the proposed algorithm, we have implemented it in a Linux server with four Intel hexad-core processors and a modern GPU system, respectively. The experimental results have shown that, for an input binary image with size of 10000×10000, the proposed parallel algorithm can achieve 18 times speedup in the multicore system, comparing with the performance of general sequential algorithm. Meanwhile, for the same input image, the proposed parallel algorithm can achieve 5 times speedup in that of GPU system. However, it is not enough to cope with the above programming issues. Especially, in our implementation, 2-dimensional arrays are mainly accessed from/to the global memory four times. However, two times of them cannot reap the benefit of the coalesced access.

The main contribution of this paper is to show an improved GPU implementation of the algorithm with more efficient memory access. In our new implementation, we have considered programming issues of the GPU system such as coalesced access for global memory and shared memory bank conflicts. The new idea of our implementation is that we have improved the access for 2-dimensional arrays that are temporal data stored in the global memory which cannot be done with coalesced access in the previous implementation. To be concrete, transposing the 2-dimensional arrays with the shared memory, the access enables to be performed by coalesced access. We have implemented and evaluated our proposed parallel EDM algorithm in the following three GPU systems, Tesla C1060 [24], GTX 480 [25] and GTX 580 [26], respectively. The experimental results have shown that for an input binary image with size of 9216×9216, our implementation can achieve a speedup factor of 54 over the sequential algorithm implementation. Also, we have presented that the density of black pixels in an input image affects the performance of the proposed GPU implementation.

The remainder of this paper is organized as follows: Section II introduces the proximate points problem for Euclidean distance metric and discusses several technicalities that will be crucial ingredients to our subsequent parallel EDM algorithm. Section III shows the proposed parallel algorithm for computing Euclidean distance map of a 2-dimensional binary image. Section IV introduces the features of the GPU system in CUDA. In Section V, we review our previous GPU implementation. Section VI exhibits a new GPU implementation considering programming issues for the GPU system. Section VII shows the performance of the new GPU implementations on different GPU systems. Finally, Section VIII offers concluding remarks.

II. Proximate Points Problem

In this section, we review the proximate problem [18] along with a number of geometric results that will lay the foundation of our subsequent algorithms. Throughout, we assume that a point p is represented by its Cartesian coordinates $(x(p), y(p))$.

Consider a collection $P = \{p_1, p_2, ..., p_n\}$ of n points sorted by x-coordinate, that is, $x(p_1) < x(p_2) < ... < x(p_n)$. We assume, without loss of generality, that all the

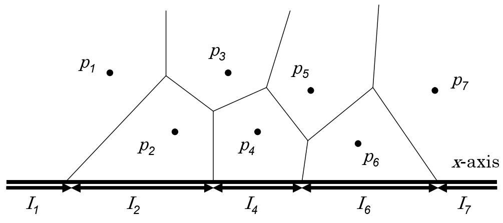


Figure 2. Proximate intervals

points in P have distinct x-coordinates and that all of them lie above the x-axis. The reader should have no difficulty to confirm that these assumptions are made for convenience only and do not impact the complexity of our algorithms.

Recall that for every point p_i of P the locus of all the points in the plane that are closer to p_i than to any other points in P is referred to as the *Voronoi polygon* associated with p_i and is denoted by $V(i)$. The collection of all the Voronoi polygons of points in P partitions the plane into the Voronoi diagram of P (see [27], p. 204). Let I_i, $(1 \le i \le n)$, be the locus of all the points q on the x-axis for which $d(q, p_i) \le d(q, p_j)$ for all p_j, $(1 \le j \le n)$. In other words, $q \in I_i$ if and only if q belongs to the intersection of the x-axis with $V(i)$, as illustrated in Figure 2. In turn, this implies that I_i must be an interval on the x-axis and that some of the intervals I_i, $(2 \le i \le n-1)$, may be empty. A point p_i of P is termed a *proximate point* whenever the interval I_i is nonempty. Thus, the Voronoi diagram of P partitions the x-axis into *proximate intervals*. Since the points of P are sorted by x-coordinate, the corresponding proximate intervals are ordered, left to right, as $I : I_1, I_2, ..., I_n$. A point q on the x-axis is said to be a *boundary point* between p_i and p_j if q is equidistance to p_i and p_j, that is, $d(p_i, q) = d(p_j, q)$. It should be clear that p is boundary point between proximate points p_i and p_j if and only if the q is the intersection of the (closed) intervals I_i and I_j. To summarize the previous discussion, we state the following result;

Proposition 2.1: The following statements are satisfied:

1) *Each I_i is an interval on the x-axis;*

2) *The intervals $I_1, I_2, ..., I_n$ lie on x-axis in this order, that is, for any nonempty I_i and I_j with $i < j$, I_i lies to the left of I_j.*

3) *If the nonempty proximate intervals I_i and I_j are adjacent, then the boundary point between p_i and p_j separates $I_i \cup I_j$ into I_i and I_j.*

Referring again to Figure 2, among the seven points, five points p_1, p_2, p_4, p_6 and p_7 are proximate points, while the others are not. Note that the leftmost point p_1 and the rightmost point p_n are always proximate points.

It is also clear that, the boundary of any two points can be computed by $O(1)$ time. For example, as shown in Figure 3, the coordinates of p_i and p_j are given. The coordinates of the midpoint of p_i and p_j can be computed in the formulas: $x_{mid} = \frac{(x_i+x_j)}{2}$ and $y_{mid} = \frac{(y_i+y_j)}{2}$. The slope of the line which crosses the points p_i and p_j can be computed by the formula: $\alpha = \frac{(y_j-y_i)}{(x_j-x_i)}$, here the α represents the slope of the line. Further, the slope of the perpendicular bisector line of p_i and p_j can be computed by the formula: $\beta = -\frac{1}{\alpha} = -\frac{(x_j-x_i)}{(y_j-y_i)}$, here the β represents the slope of the perpendicular bisector line. Finally the perpendicular bisector line of p_i and p_j can be computed by the formula: $y = \beta(x - x_{mid}) + y_{mid} = -\frac{(x_j-x_i)}{(y_j-y_i)}(x - \frac{(x_i+x_j)}{2}) + \frac{(y_i+y_j)}{2}$. The x-coordinate of the intersection point of the perpendicular bisector line and the x-axis can be obtained as follow: $x_{inter} = \frac{(y_j^2-y_i^2)+(x_j^2-x_i^2)}{2(x_j-x_i)}$. This intersection point is also the boundary point of p_i and p_j. Therefore the coordinate of the boundary point is ($\frac{(y_j^2-y_i^2)+(x_j^2-x_i^2)}{2(x_j-x_i)}$, 0). The coordinate of

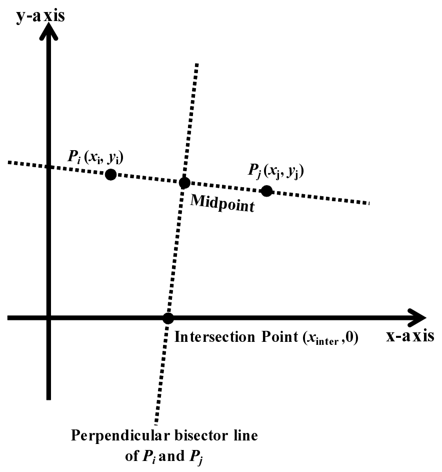


Figure 3. Perpendicular bisector line of two points

the boundary point can be computed in $O(1)$ time using a single processor.

Given three points p_i, p_j, p_k with $i < j < k$, we say that p_j is *dominated* by p_i and p_k whenever p_j fails to be a proximate point of the set consisting of these three points. Clearly, p_j is dominated by p_i and p_k if the boundary of p_i and p_j is to the right of that of p_j and p_k. Since, the boundary of any two points can be computed in $O(1)$ time, therefore the task of deciding for every triple (p_i, p_j, p_k), whether p_j is dominated by p_i and p_k takes $O(1)$ time using single processor.

Consider a collection $P = \{p_1, p_2, ..., p_n\}$ of points in the plane sorted by x-coordinate, and a point p to the right of P, that is, such that $x(p_1) < x(p_2) < ... < x(p_n) < x(p)$. We are interested in updating the proximate intervals of P to reflect the addition of p to P, as illustrated in Figure 4.

We assume, without loss of generality, that all points in P

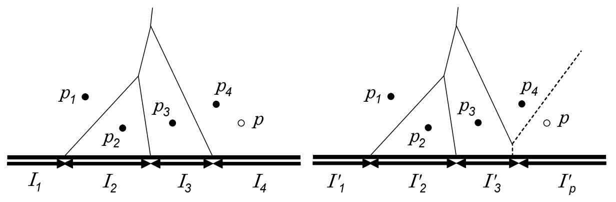


Figure 4. Illustrating the addition of p to $P = \{p_1, p_2, p_3, p_4\}$

are proximate points and let $I_1, I_2, ..., I_n$ be the corresponding proximate intervals. Further, let $I'_1, I'_2, ..., I'_n, I'_p$ be the updated proximate intervals of $P \cup \{p\}$. Let p_i be a point such that I'_i and I'_p are adjacent.

Lemma 2.2: There exists a unique point of p_i of P such that:

- *The only proximate points of $P \cup \{p\}$ are $p_1, p_2, ..., p_i, p$.*
- *For $2 \leq j \leq i$, the point p_j is not dominated by p_{j-1} and p. Moreover, for $1 \leq j \leq i-1$, $I'_j = I_j$.*
- *For $i < j \leq n$, the point p_j is dominated by p_{j-1} and p and the interval I'_j is empty.*
- *I'_i and I'_p are consecutive on the x-axis and are separated by the boundary point between p_i and p.*

We show an intuitive proof of the lemma by geometry. As shown in Figure 5(a), the line $\overline{p_n p}$ and line $\overline{p_{n-1}p_n}$ denote the perpendicular bisector lines of the point pair $\{p_n, p\}$ and the point pair $\{p_{n-1}, p_n\}$. The intersection of $\overline{p_n p}$ and the x-axis is located left to the intersection of $\overline{p_{n-1}p_n}$ and the x-axis. It implies the proximate interval of p_n is empty.

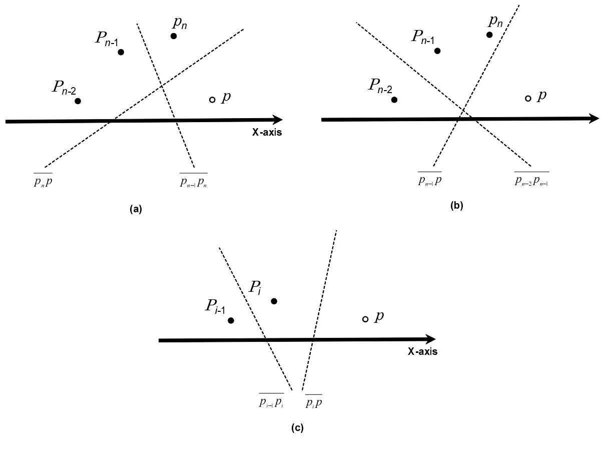


Figure 5. Perpendicular bisector lines

Now we draw the perpendicular bisector lines of the point pair of $\{p_{n-2}, p_{n-1}\}$ and the point pair of $\{p_{n-1}, p\}$, they are denoted by line $\overline{p_{n-2}p_{n-1}}$ and line $\overline{p_{n-1}p}$, see Figure 5(b). The intersection of $\overline{p_{n-1}p}$ and the x-axis is also located left to the intersection of $\overline{p_{n-2}p_{n-1}}$ and the x-axis. It means the proximate interval of p_{n-1} is also empty. We repeat the procedure until find a point, p_i, $1 < i < n-1$, its proximate interval is nonempty, see Figure 5(c). As shown in the figure, the line $\overline{p_{i-1}p_i}$ denotes the perpendicular bisector line of the point pair of $\{p_{i-1}, p_i\}$ and the line $\overline{p_i p}$ denotes the perpendicular bisector line of the point pair of $\{p_i, p\}$. It is clear that the intersection of $\overline{p_i p}$ and x-axis is located right to the intersection of $\overline{p_{i-1}p_i}$ and x-axis. It means the proximate interval of p is decided. The proximate interval of p_i is adjacent to the proximate interval of p. The intersection of $\overline{p_i p}$ and x-axis is the boundary point of p_i and p. It also imply that the point p can not affect the proximate interval of p_j, where $1 \leq j \leq i-1$.

Let $P = \{p_1, p_2, ..., p_n\}$ be a collection of proximate points sorted by x-coordinate and let p be a point to the left of P, that is $x(p) < x(p_1) < x(p_2) < ... < x(p_n)$. For further reference, we now take note of the following companion result to Lemma 2.2. The proof is identical and, thus, omitted.

Lemma 2.3: There exists a unique points of p_i of P such that:

- *The only proximate points of $P \cup \{p\}$ are $p, p_i, p_{i+1}, ..., p_n$.*
- *For $i \leq j \leq n$, the point p_j is not dominated by p and p_{j+1}. Moreover, for $i+1 \leq j \leq n$, $I'_j = I_j$.*
- *For $1 \leq j < i$, the point p_j is dominated by p and p_{j+1} and the interval I'_j is empty.*
- *I'_p and I'_i are consecutive on the x-axis and are separated by the boundary point between p and p_i.*

The unique point p_i whose existence is guaranteed by Lemma 2.2 is termed the *contact point* between P and p. The second statement of Lemma 2.2 suggests that the task of determining the unique contact point between P and a point p to the right or the left of P reduces, essentially, to binary search.

Now, suppose that the set $P = \{p_1, p_2, ..., p_{2n}\}$, with $x(p_1) < x(p_2) < ... < x(p_{2n})$ is partitioned into two subsets $P_L = \{p_1, p_2, ..., p_n\}$ and $P_R = \{p_{n+1}, p_{n+2}, ..., p_{2n}\}$. We are interested in updating the proximate intervals in the process or merging P_L and P_R. For this purpose, let $I_1, I_2, ..., I_n$ and $I_{n+1}, I_{n+2}, ..., I_{2n}$ be the proximate intervals of P_L and P_R, respectively. We assume, without loss of generality, that all these proximate intervals are nonempty. Let $I'_1, I'_2, ..., I'_{2n}$ be the proximate intervals of $P = P_L \cup P_R$. We are now in a position to state and prove the next result which turns out to be a key ingredient in our algorithms.

Lemma 2.4: There exists a unique pair of proximate points $p_i \in P_L$ and $p_j \in P_R$ such that

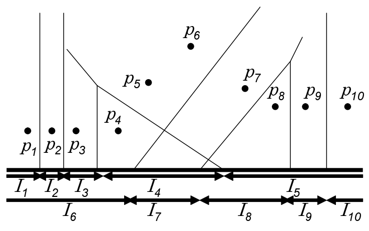
- *The only proximate points in $P_L \cup P_R$ are $p_1, p_2, ..., p_i, p_j, ..., p_{2n}$.*
- *$I'_{i+1}, ..., I'_{j-1}$ are empty, and $I'_k = I_k$ for $1 \leq k \leq i-1$ and $j+1 \leq k \leq 2n$.*
- *The proximate intervals I'_i and I'_j are consecutive and*

are separated by the boundary point between p_i and p_j.

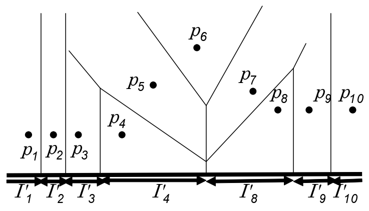
Proof: Let i be the smallest subscript for which $p_i \in P_L$ is the contact point between P_L and a point in P_R. Similarly, let j be the largest subscript for which the point $p_j \in P_R$ is the contact point between P_R and some point in P_L. Clearly, no point in P_L to the left of p_i can be proximate point of P. Likewise, no point in P_R to the left of p_j can be a proximate point of P.

Finally, by Lemma 2.2, every point in P_L to the left of p_i must be a proximate point of P. Similarly, by Lemma 2.3, every point in P_R to the right of p_i must be a proximate point of P, and proof of the lemma is complete. ■

The points p_i and p_j whose existence is guaranteed by Theorem 2.4 are termed the *contact points* between P_L and P_R. We refer the reader to Figure 6 for an illustration. Here, the contact points between $P_L = \{p_1, p_2, p_3, p_4, p_5\}$ and $P_R = \{p_6, p_7, p_8, p_9, p_{10}\}$ are p_4 and p_8.



(a) Proximate interval of each point in two sets



(b) Merge of two point sets and their contact points

Figure 6. Illustrating the contact points between two sets of points

Next, we discuss a geometric property that enables the computation of the contact points p_i and p_j between P_L and P_R. For each point p_k of P_L, let q_k denote the contact point between p_k and P_R as specified by Lemma 2.3. We have the following result.

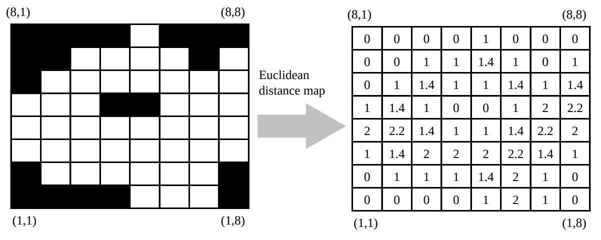
Lemma 2.5: The point p_k is not dominated by p_{k-1} and q_k if $2 \leq k \leq i$, and dominated otherwise.

Proof: If p_k, $(2 \leq k \leq i)$, is dominated by p_{k-1} and q_k, then I'_k must be empty. Thus, Lemma 2.4 guarantees that p_k, $(2 \leq k \leq i)$, is not dominated by p_{k-1} and q_k. Suppose that p_k, $(i+1 \leq k \leq n)$, is not dominated by p_{k-1} and q_k. Then, the boundary point between p_k and q_k is to the right of that between these two boundaries corresponds to I'_k, a contradiction. Therefore, p_k, $(i+1 \leq k \leq n)$, is dominated by p_{k-1} and q_k, completing the proof. ■

Lemma 2.5 suggests a simple, binary search-like, approach to finding the contact points p_i and p_j between two sets P_L and P_R. In fact, using a similar idea, Breu et al. [13] proposed a sequential algorithm that computes the proximate points of an n-point planar set in $O(n)$ time. The algorithm in [13] uses a stack to store the proximate points found.

III. Parallel Euclidean Distance Map of 2-dimensional Binary Image

A binary image I of size $n \times n$ is maintained in an array $b_{i,j}$, $(1 \leq i, j \leq n)$. It is customary to refer to pixel (i, j) as *black* if $b_{i,j} = 1$ and as *white* if $b_{i,j} = 0$. The rows of the image will be numbered bottom up starting from 1. Likewise, the columns will be numbered left to right, with column 1 being the leftmost. In this notation, pixel $b_{1,1}$ is in the south-west corner of the image, as illustrated in Figure 7(a). In Figure 7(a), each square represents a pixel. For this binary image, its final distance mapping array is shown in Figure 7(b).



0	0	0	0	1	0	0	0
0	0	1	1	1.4	1	0	1
0	1	1.4	1	1	1.4	1	1.4
1	1.4	1	0	0	1	2	2.2
2	2.2	1.4	1	1	1.4	2.2	2
1	1.4	2	2	2	2.2	1.4	1
0	1	1	1	1.4	2	1	0
0	0	0	0	1	2	1	0

(a) Binary image

(b) Mapping array

Figure 7. A binary image and its mapping array

The *Voronoi map* associates with every pixel in I the closest black pixel to it (in the Euclidean metric). More formally, the Voronoi map of I is a function $v : I \rightarrow I$ such that, for every (i, j), $(1 \leq i, j \leq n)$, $v(i, j) = v(i', j')$ if and only if

$$d((i,j),(i',j')) = min\{d((i,j),(i'',j'')) \mid b_{i'',j''} = 1\},$$

where $d((i,j),(i',j')) = \sqrt{(i-i')^2 + (j-j')^2}$ is the Euclidean distance between pixels (i, j) and (i', j').

The *Euclidean Distance Map* of image I associates with every pixel in I in the Euclidean distance to the closest black pixel. Formally, the Euclidean Distance Map is a function m: $I \rightarrow R$ such that for every (i, j), $(1 \leq i, j \leq n)$, $m(i, j) = d((i, j), v(i, j))$.

We now outline the basic idea of our algorithm for computing the Euclidean Distance Map of image I. We begin by determining, for every pixel in row j, $(1 \leq j \leq n)$, the nearest black pixel, if any, in the same column of I. More precisely, with every pixel (i, j) we associate the value

$$d_{i,j} = min\{d((i,j),(i',j')) \mid b_{i',j'} = 1, 1 \leq j' \leq n\}.$$

If $b_{i',j'} = 0$ for every $1 \leq j' \leq n$, then let $d_{i,j} = +\infty$. Next, we construct an instance of the proximate points problem for every row j, $(1 \leq j \leq n)$, in the image I involving the set P_j of points in the plane defined as $P_j = \{p_{i,j} = (i, d_{i,j}) \mid 1 \leq i \leq n\}$.

Having solved, in parallel, all these instances of the proximate points problem, we determine, for every proximate point $p_{i,j}$ in P_j, its corresponding proximity interval I_i. With j fixed, we determine, for every pixel (i,j) (that we perceive as a point on the x-axis), the identity of the proximity interval to which it belongs. This allows each pixel (i,j) to determine the identity of the nearest pixel to it. The same task is executed for all rows $1, 2, ..., n$ in parallel, to determine, for every pixel (i,j) in row j, the nearest black pixel. The details are spelled out in the following algorithm:

vspace2mm **Algorithm :** *Euclidean Distance Map(I)*

Step 1 For each pixel (i,j), compute the distances

$$d_{i,j} = min\{|k-i| \mid b_{k,j} = 1, 1 \leq k \leq n\}$$

to the nearest black pixel in the same column.

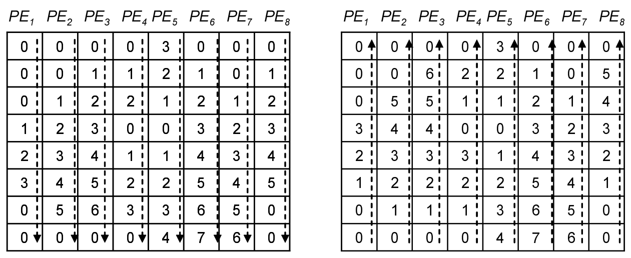
vspace2mm **Step 2** let $P_j = \{p_{i,j} = (i, d_{i,j}) \mid 1 \leq i \leq n\}$. Compute the proximate points $E(P_j)$ of P_j.

Step 3 For every point p in $E(P_j)$ determine its proximity interval of P_j.

Step 4 For every i, $(1 \leq i \leq n)$, determine the proximate interval of P_j to which the point $(i, 0)$ (corresponding to pixel (i,j)) belongs. vspace2mm

e assume that there are n processors PE(1), PE(2), ..., PE(n) available. The parallel implementation of above algorithm is shown as follows:

Step 1. We assign the i-th column ($1 \leq i \leq n$) to processor PE(i) to compute the distance to the nearest black pixel in the same column. First, each PE(i) ($1 \leq i \leq n$) reads pixel values in the i-th column from up to bottom to compute that distance, as illustrated in Figure 8(a) (its original input image is shown in Fig 7). Second, each processor PE(i) ($1 \leq i \leq n$)



PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7	PE_8
0	0	0	0	3	0	0	0
0	0	1	1	2	1	0	1
0	1	2	2	1	2	1	2
1	2	3	0	0	3	2	3
2	3	4	1	1	4	3	4
3	4	5	2	2	5	4	5
0	5	6	3	3	6	5	0
0	0	0	0	4	7	6	0

(a) process with up to bottom

PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7	PE_8
0	0	0	0	3	0	0	0
0	0	6	2	2	1	0	5
0	5	5	1	1	2	1	4
3	4	4	0	0	3	2	3
2	3	3	3	1	4	3	2
1	2	2	2	2	5	4	1
0	1	1	1	3	6	5	0
0	0	0	0	4	7	6	0

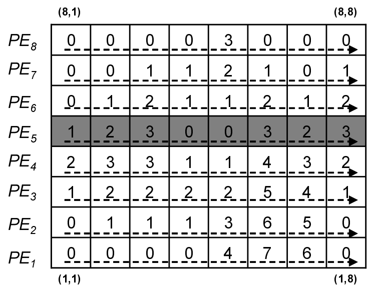
(b) process with bottom to up

Figure 8. Process each column with two directions

reads pixel values in the same column from bottom to up to compute that distance, as illustrated in Figure 8(b). Finally, each processor selects a minimum value of calculated two distances as final value of the distance. It is clear that the time complexity of this step is $O(n)$.

noindent **Step 2.** Again, we compute Euclidean Distance Map of input image I along with row wise.

Step 2.1 For every i-th row ($1 \leq i \leq n$), each processor PE(i) computes the proximate points using the theorem of proximate points problem as foundation, as illustrated in Figure 9 and Figure 10.



PE_8	0	0	0	0	3	0	0	0
PE_7	0	0	1	1	2	1	0	1
PE_6	0	1	2	1	1	2	1	2
PE_5	1	2	3	0	0	3	2	3
PE_4	2	3	3	1	1	4	3	2
PE_3	1	2	2	2	2	5	4	1
PE_2	0	1	1	1	3	6	5	0
PE_1	0	0	0	0	4	7	6	0

Figure 9. Processing with row wise

In Figure 10, the Voronoi polygons correspond to 5th row (shaded row) of the image illustrated in Figure 9. The obtained proximate points are saved in a stack.

It should be clear that each column has its own corresponding stack. Therefore, in order to add a new proximate point to the stack, we need to calculate boundary points of this new point and existed proximate points which are kept in the stack. Then according to locus of boundary points, we decide which points need to be deleted from the stack.

Step 2.2 For every i-th row ($1 \leq i \leq n$), each processor PE(i) determines proximate intervals of obtained proximate points by computing boundary point of each pair of adjacent proximate points. The boundary point of each pair of adjacent proximate points can be obtained by calculating the intersection point of two lines, one line is x-axis and another is the normal line of the line which connects two adjacent proximate points. We refer reader to Figure 11 for the illustration. Each pair of adjacent proximate points can

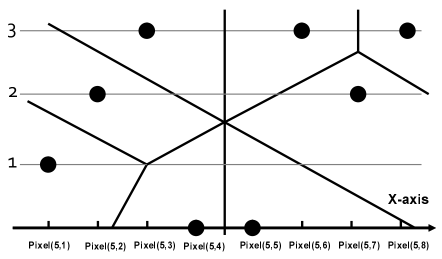


Figure 10. Voronoi polygons

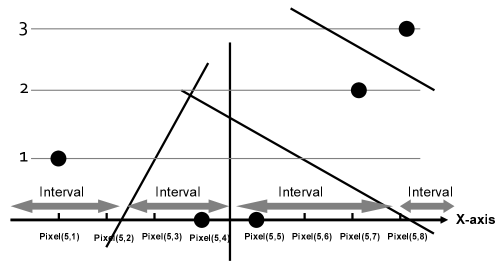


Figure 11. Proximate intervals

be obtained from the stack.

Step 2.3 According to the locus of boundary points obtained from Step 2.2, each processor determines the closest black pixel to each pixel of the input image. The distance between a given pixel and its closest black pixel is also calculated in the obvious way.

It should be clear that, the whole Step 2 can be implemented in $O(n)$ time using n processors.

Theorem 3.1: For a given binary image I with the size of $n \times n$, Euclidean Distance Map of image I can be computed in $O(n)$ time using n processors.

Suppose that we have k processors ($k < n$). If this is the case, a straightforward simulation of n processors by k processors can achieve optimal slowdown. In other words, each of the k processors performs the task of $\frac{n}{k}$ processors in our Euclidean Distance Map algorithm. For example, in Step 1, the i-th processor ($1 \leq i \leq k$) computes the nearest black pixel within the same column for rows from $(i-1) \cdot \frac{n}{k} + 1$-th to $i \cdot \frac{n}{k}$. This can be done in $O(n \cdot \frac{n}{k}) = O(\frac{n^2}{k})$ time. Thus, we have,

Corollary 3.2: For a given binary image I with the size of $n \times n$, Euclidean Distance Map of image I can be computed in $O(\frac{n^2}{k})$ time using k processors.

IV. Compute Unified Device Architecture (CUDA)

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [23]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [28], [5]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, threads should perform coalesced access when they access to the global memory. Figure 12 illustrates the CUDA hardware architecture.

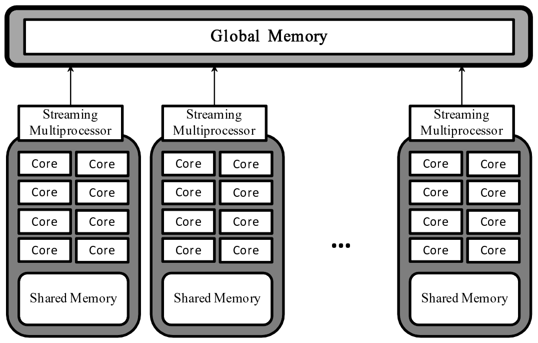


Figure 12. CUDA hardware architecture

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access to the global memory. However, as we can see in Figure 12, threads in a block can access to the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories.

CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each block are executed by processor cores in a single streaming processor. In the execution, threads in a block are split into groups of thread called *warps*. Each of these warps contains the same number of threads and is execute independently. When a warp is selected for execution, all threads execute the same instruction. Any flow control instruction (e.g. if-statements in C language) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge, that is, to follow different execution paths. If this happens, the different execution paths have to be serialized. When all the different execution paths have completed, the threads back to the same execution path. For example, for an if-else statement, if some threads in a warp take the if-clause and others take the else-clause, both clauses are executed in serial. On the other hand, when all threads in a warp branch in the same direction, all threads in a warp take the if-clause, or all take the else-clause. Therefore, to improve the performance, it is important to make branch behavior of all threads in a warp uniform.

As we have mentioned, the coalesced access to the global memory is a key issue to accelerate the computation. As illustrated in Figure 13, when threads access to continuous

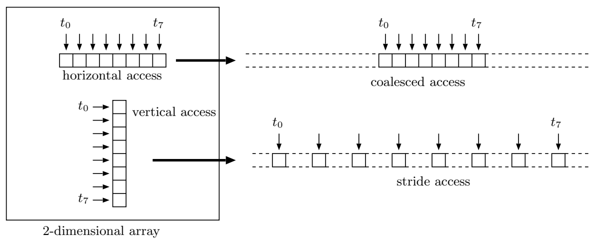


Figure 13. Coalesced and stride access

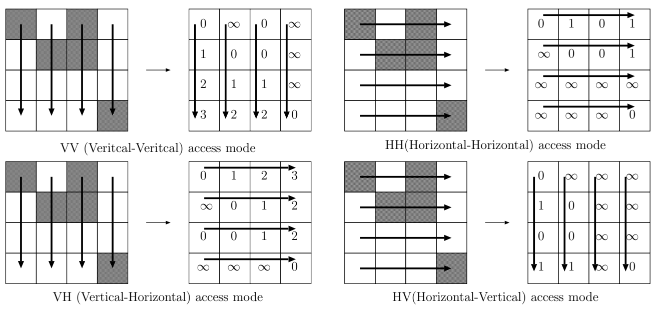

VV (Veritcal-Veritcal) access mode
HH(Horizontal-Horizontal) access mode
VH (Vertical-Horizontal) access mode
HV(Horizontal-Vertical) access mode

Figure 14. Access modes for Step 1

locations in a row of a 2-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed in the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed in the same time (*stride access*). From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus, we should avoid the stride access (or the vertical access) and perform the coalesced access (or the horizontal access) whenever possible.

V. Our Previous Implementation of EDM Algorithm on GPUs

In this section, we show our previous implementation of EDM algorithm on GPUs [5]. We have defined several memory access modes which affect the performance of our algorithm. Using the access modes, we have implemented a parallel EDM algorithm.

A. Access Modes

The key part of our Euclidean Distance Map algorithm is Step 1 and Step 2. We will define several access modes which affect the performance of our algorithm. Recall that in Step 1, pixel values are read in column wise, and the distances to the nearest black pixel are written in column wise. Instead, we can write the distances to the nearest black pixel in row wise. In other words, we can read the pixel values in column wise (i.e. *Vertical*), or in row wise (i.e. *Horizontal*) and write the distances in column wise (i.e. *Vertical*) or in row wise (i.e. *Horizontal*). The readers should refer to Figure 14 for illustrating the possible four access modes of Step 1.

Let $d_{i,j}$ denote the resulting distances of Step 1. For each access mode we can write $d_{i,j}$ as follows:

VV (Vertical-Vertical) $d_{i,j} = min\{|k-i| \mid b_{k,j} = 1, 1 \leq k \leq n\}$

VH (Vertical-Horizontal) $d_{j,i} = min\{|k-i| \mid b_{k,j} = 1, 1 \leq k \leq n\}$

HH (Horizontal-Horizontal) $d_{i,j} = min\{|k-j| \mid b_{i,k} = 1, 1 \leq k \leq n\}$

HV (Horizontal-Vertical) $d_{j,i} = min\{|k-j| \mid b_{i,k} = 1, 1 \leq k \leq n\}$

Note that, for VH and HV access modes, the resulting values stored in the two dimensional array is transposed.

In the same way, we can define four possible access modes VV, VH, HH and HV for Step 2. For example, in VV mode, the distances are read in column wise and the resulting values of Euclidean Distance Map are written in column wise.

The readers should have no difficulty to confirm that possible combinations of access modes for Steps 1 and 2 are **VV-HH**, **HH-VV**, **VH-VH**, and **HV-HV**, because the access mode satisfies the following two conditions:

Condition 1 If the resulting values in Step 1 are stored in a transposed array, those in Step 2 also must be transposed. Otherwise, the resulting Euclidean Distance Map is transposed.

Condition 2 The writing directions of Step 1 and Step 2 must be orthogonal.

Therefore, in the notation $r_1w_1r_2w_2$ of access modes, w_1 and r_2 must be distinct from Condition 1 and the number of H in r_1, w_1, r_2, and w_2 must be even from Condition 2. Therefore, the possible access modes are VV-HH, HH-VV, VH-VH, and HV-HV.

B. Implementations with Different Access Modes

In our previous work [5], we have implemented our proposed parallel EDM algorithm with the above four access modes. Also, we have evaluated our proposed parallel EDM algorithm with Tesla C1060 [24] which consists of 240 Streaming Processor Cores and 4GB global memory. The experimental result shown in [5], the performance of VH-VH access mode was better than the other access modes. This is because in VH-VH access mode, the GPU implementation can benefit from coalesced access to the global memory significantly.

For clear explanation, first we describe the details of the GPU implementation of the parallel Euclidean Distance Map algorithm. Here we just describe the GPU implementation of VH-VH access mode. For other access modes, their implementations can be understood in the same way.

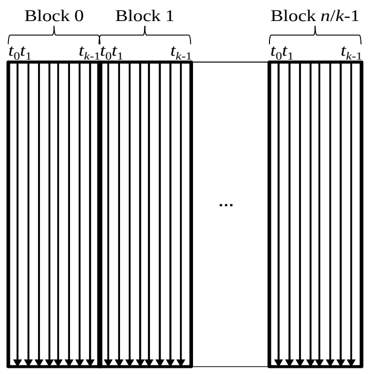


Figure 15. Mapping blocks into subimages

For implementing Step 1 of the algorithm, we partition the original input image of $n \times n$ into $\frac{n}{k}$ subimages along with column wise, where k is the number of threads in one block. We assign $\frac{n}{k}$ blocks are assigned to subimages and each block processes each corresponding subimage independently. Each thread of a block processes each corresponding column of the subimage. We refer readers to Figure 15 as an simple illustration. In Figure 15, each $t_i (0 \leq i \leq k-1)$ represents a thread of a block and each arrow represents an access of a pixel value by one thread. It is clear that, for a subimage, the access to each row can be performed in coalescing.

By following Step 1 of the parallel EDM algorithm, each thread needs to access each pixel value of the corresponding column two times. One is access for computing results of up-to-bottom process and the other is access for computing results of bottom-to-up process. After selecting the minimum value for each pixel, each thread writes the minimum one into an extra array which stores the results of Step 1 along with row wise. It is clear that, the both up-to-bottom process and bottom-to-up process can benefit from full coalescing. However, the writing of the extra array cannot benefit from the coalescing at all. On the other hand, in the implementation of VV-HH access mode, the writing of the extra array is also can benefit from the full coalescing. Therefore in VV-HH access mode, the implementation of Step 1 can achieve the most significant performance. Differently, in HH-VV access mode, the whole implementation of Step 1 cannot benefit from the coalescing at all since the read and write operation for the global memory is stride access. Therefore Step 1 of the HH-VV access mode achieved the worst performance.

In Step 2 of the algorithm, stacks are necessary for computing boundary points. Since one stack is used for the computation of each column, n stacks are necessary in total. We allocate a 2-dimensional array in the global memory to the stacks. Each stack is assigned to one column of the 2-dimensional array. Also, each thread reads elements of corresponding column of the extra array, which stores the results of Step 1, to obtain elements of corresponding stack. However the push-pop operations for the stacks are not uniform. Therefore the access of the extra array cannot be performed in full coalescing. In the same way, the access of the stacks also cannot be performed in full coalescing. This is reason that the implementation of Step 2 cannot achieve a significant performance even in HH-VV access mode. After computing boundary points, we compare the y-coordinate of each boundary point with the y-coordinate of each pixel to obtain the distance to the closest black pixel. If we assume that the mapping results will be stored in a 2-dimensional array named output array, it needs all threads accesses the output array along with row wise. In other words, each thread accesses the corresponding row of the output array, and it cannot utilize the coalescing. However, in Step 2 of VV-HH access mode, its whole implementation cannot benefit from the coalescing at all. This is the reason that Step 2 of HV-HV access mode can be little faster than Step 2 of VV-HH access mode.

VI. New Implementation of EDM Algorithm on GPUs

The main purpose of this section is to show our new implementation of EDM algorithm in the GPU. In the followings, we introduce a new access mode and a new implementation with it.

A. New Access Mode with Efficient Memory Access

As we see in the previous section, VH-VH access mode can obtain the best performance of four access modes. Therefore it is clear that coalesced access to global memory plays an important role in our GPU implementations. However, VH-VH access mode cannot fully benefit from coalesced access because its memory writing does not support coalesced access. Therefore, in this subsection, we show a new implementation of the proposed algorithm which can fully utilize the coalescing in each implementing step in memory read and write. We call the access mode of the new implementation as *VTV-VTV access mode* (VTV stands for *Vertical-Transpose-Vertical*). To keep two conditions as shown in the previous section, following operations are performed in each step;

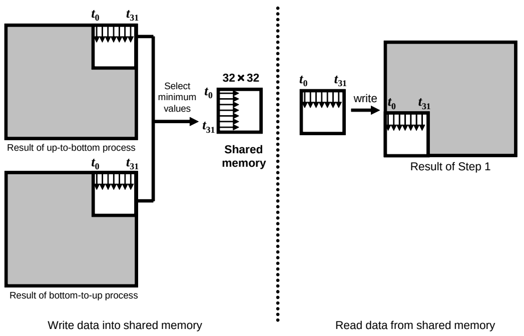
1) An input data is read from global memory with coalesced read.
2) The results are transposed with shared memory.
3) The transposed results are written into the global memory with coalesced write.

More specifically, in the new access mode of Step 1, the 2-dimensional array of the input image is read in column wise by each thread. After processing, the results are transposed using shared memory. The transposed data is written into another array in column wise by each thread as the results of Step 1 and the input data of Step 2. In the new implementation of Step 2, the 2-dimensional extra array which contains the results of Step 1 is read in column wise by each thread. After reading data from the 2-dimensional extra array, the resulting values of Step 2 are transposed using shared memory. The transposed results are written into the extra array column by column by each thread. It is clear that, in VTV-VTV access mode, each step can be implemented with full coalescing.

B. GPU Implementation with New Access Mode

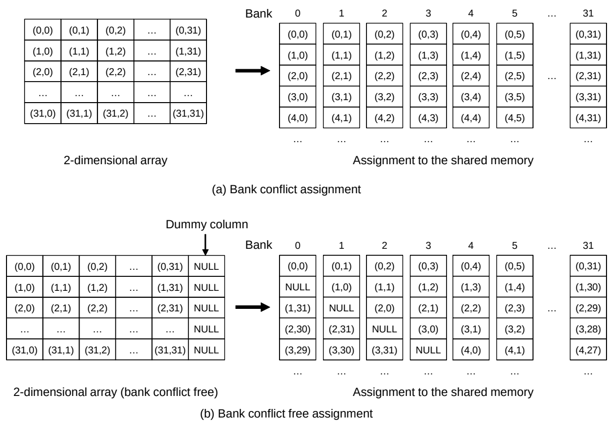
We now show the new implementation of Step 1 for VTV-VTV access mode. The results, which are stored to 2-dimensional arrays, of up-to-bottom and bottom-to-up process are obtained by the same manner of the implementation for VH-VH access mode shown in Section V-B. After that, each resulting 2-dimensional array is divided into subimages whose size is 32×32. One block is assigned to each subimage and each block runs independently.

In each block, the minimum values from corresponding elements in the two 2-dimensional arrays are selected. To obtain the results of Step 1 in VTV-VTV access mode, the minimum ones are transposed. In our proposed implementation, to transpose them, we utilize the shared memory. As shown in Figure 16, the 32 resulting values of up-to-bottom and bottom-to-up process each are read in column wise using coalesced access with 32 threads. The minimum ones are selected and written to the shared memory in column wise. The above read and write operation is executed column by column. After that, the values are written to the corresponding transposed position in the global memory in column wise with coalesced access. Using the shared memory, all the access from/to the global memory can be coalesced.



t_0 t_{31}
Select minimum values
32×32
Shared memory
write
Result of up-to-bottom process
Result of bottom-to-up process
Result of Step 1
Write data into shared memory
Read data from shared memory

Figure 16. Coalesced Transpose with Shared Memory



Bank
0 1 2 3 4 5 ... 31
2-dimensional array
Assignment to the shared memory
(a) Bank conflict assignment
Dummy column
NULL
2-dimensional array (bank conflict free)
Assignment to the shared memory
(b) Bank conflict free assignment

Figure 17. Bank conflict free map

However, in the above implementation, the use of shared memory causes another problem, shared memory bank conflicts. As given above, the size of the shared memory array is 32×32. It means that one column of this array is mapped into the same bank of shared memory, since there are 16 or 32 banks in shared memory of CUDA GPU [23]. If multiple threads in a block access to the distinct banks in the shared memory, the access can be serviced simultaneously. On the other hand, if threads access to the same bank, the access has to be serialized. In our implementation, when threads write the minimum values to the shared memory, they write the minimum ones to the same column of the 2-dimensional array in the shared memory (Figure 17(a)). Therefore, bank conflict occurs. To avoid the bank conflict, we add a dummy column to the shared memory array (Figure 17(b)). Adding the dummy column, elements of each column are mapped into different banks and all the access in the transposing is free from the bank conflict.

In Step 2 of the implementation, the resulting values are transposed with the shared memory in the same manner as the above.

VII. Performance Evaluation

In this section, we show the performance evaluation of the proposed GPU implementation through different experiments. In all the experiments, we have used a binary image of size 9216×9216. Every measurement is the average value of 20 experiments. For all measurements obtained from GPU systems, the variance corresponding to each measurement is always less than 1. For example, the experimental system is GTX 580 and the input image is the Lenna image (see Figure 18), then the variance of the 20 experiments is only 0.64.

Table I shows the performance of the new implementation on different GPU systems. For the binary image of Lenna (see Figure 18), our new implementation using VTV-VTV access mode can achieve 20, 46 and 54 times speedup on

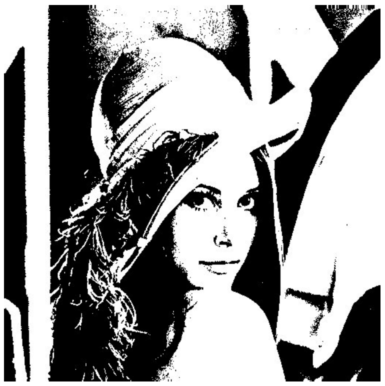


Figure 18. Binary Image of Lenna

Tesla C1060, GTX 480 and GTX 580 system respectively, over the performance of the sequential algorithm implemented on a CPU system with Intel Core i7 processor [29]. The experimental results also show that, even if the total computing time includes data transfer time between host memory and global memory, our new implementation also can achieve about 10, 30 and 34 times speedup on Tesla C1060, GTX 480 and GTX 580 system, respectively. The table also show that, the implementation with the VTV-VTV access mode can achieve 1.6x speedup, compared with the implementation with VHVH access mode, in GTX 580 system. However it just achieve 1.4x speedup in Tesla C1060 system. Actually Tesla C1060 only support previous generation CUDA architecture. However GTX 580 can support new generation CUDA architecture, *Fermi* architecture [30] . Compared with the previous generation CUDA architecture, the Fermi architecture introduces several architectural innovations. For example, in the Fermi architecture, at most 512 CUDA cores can be supported, the global memory is featured by L1/L2 caches, the dual warp scheduler is supported, etc. On the other hand, compared with the previous generation CUDA architecture, the number of memory transactions required by a fully coalesced memory access is also reduced in the Fermi architecture. In the previous generation CUDA architecture, a global memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. It means that, for a warp, it needs at least two memory transactions to access the global memory, even the global memory accesses are coalesced. However, in the Fermi architecture, a global memory request for a warp is issued into one memory transaction, if the global memory accesses are coalesced. This is reason to why the coalesed access of global memory can achieves more speedups in GTX 580.

It should be clear that the execution time depends on contents of the input images. Therefore, we evaluated the performance for the input images that have the different density of black pixels. We generated input images whose

Table I
PERFORMANCE OF IMPLEMENTATION WITH VH-VH AND VTV-VTV ACCESS MODE ON DIFFERENT GPU SYSTEMS (n=9216)

(a) Tesla C1060

	CPU	VH-VH access mode		VTV-VTV access mode	
	Time[ms]	Time[ms]	Speed-up	Time[ms]	Speed-up
Step1	3956	147	26.9	39	101.4
Step2	7205	621	11.6	508	14.7
Total	11161	768	14.5	547	20.4

(b) GTX 480

	CPU	VH-VH access mode		VTV-VTV access mode	
	Time[ms]	Time[ms]	Speed-up	Time[ms]	Speed-up
Step1	3956	90	43.9	20	197.8
Step2	7205	273	26.3	221	35.4
Total	11161	363	30.7	241	46.0

(c) GTX 580

	CPU	VH-VH access mode		VTV-VTV access mode	
	Time[ms]	Time[ms]	Speed-up	Time[ms]	Speed-up
Step1	3956	93	42.5	16	247.2
Step2	7205	238	30.2	190	39.1
Total	11161	331	33.7	206	54.1

black pixels are randomly distributed such that the density of black pixels is varied from 0% to 100%. Figure 19 shows the performance of the GPU implementation with two different access modes on the GTX 580 system. From the figure, the GPU implementation with VTV-VTV access mode can achieve a higher performance than that with VH-VH access mode for each density of black pixels. The reason is that more global memory accesses can be coalesced in VTV-VTV access mode.

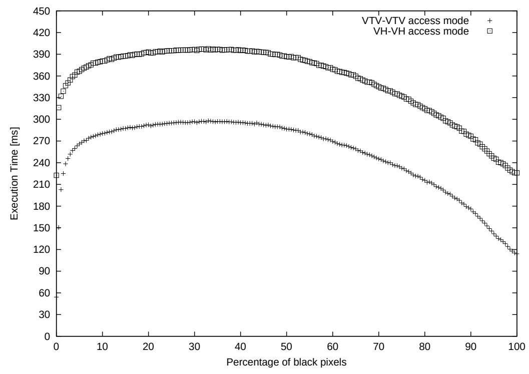


Figure 19. Performance of the GPU implementation with different access modes

Figure 19 also shows how the performance of the GPU implementation is affected by the density of black pixels in the input image. However, the computing time of Step 1 is independent from contents of the input images. The computing time of Step 2 depends only on the contents. Therefore, we focus on the behavior in Step 2. If the density of black pixels is small, pixels of input image have the common nearest black pixel. In other words, each of the black pixels dominates relative large area of the input image.

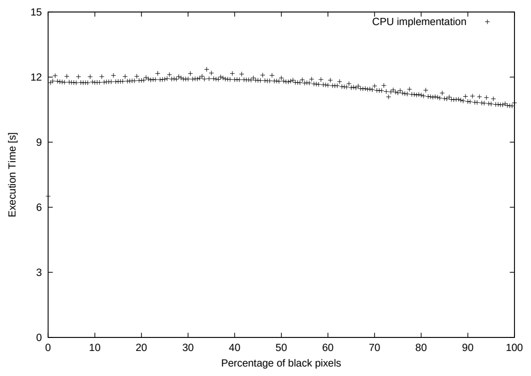


Figure 20. Performance of CPU implementation with HV-HV access mode

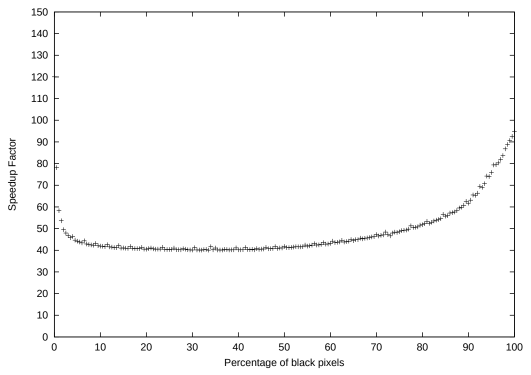


Figure 21. Speedup factor of GPU implementation compared with CPU implementation

Therefore, the behavior of the threads in each warp is almost the same and computing time becomes shorter. According to the figure, when the percentage of black pixels is close to about 40%, the proposed GPU implementation achieves the worst performance. When the density is the above, many of pixels of input image do not have the common nearest black pixel. Therefore, the behavior of the threads in each warp differs and it causes worse performance. On the other hand, when the percentage of black pixels is larger than 40%, the execution time of the GPU implementation is decreasing along the increase of the percentage of black pixels. The behavior of the threads in each warp is almost the same, which is similar to the lower density of black pixels. Therefore, better performance is achieved. Especially, if the density is close to the 100%, that is almost all the pixels are black, access of stacks assigned to threads in a warp is almost identical. Namely, all the access to the global memory over the whole process reaps the benefit of coalesced access.

Figure 20 shows the performance of the CPU implementation of the sequential algorithm on images with different percentage of randomly distributed black pixels. In our previous paper [5], we have shown that the CPU implementation can achieve the best performance in HV-HV access mode. Therefore, we only show the performance of the CPU implementation with HV-HV access mode. In the figure, it is clear that the density of black pixels has no significant effect on the performance of the CPU implementation. Figure 21 shows the speedup factor of the GPU implementation with VTV-VTV access mode, compared with the CPU implementation. From the figure, for the input images with different percentage of randomly distributed black pixels, our proposed GPU implementation can achieve at least 40 times speedup compared with the optimal CPU implementation.

On the other hand, experiments show that, the uniform distribution of black pixels (see Figure 22) will result in the worst performance. Since the uniform distribution of black pixels will bring a more complicated global memory access on GPUs. Therefore, in this paper, we just show the performance of the uniform distribution.

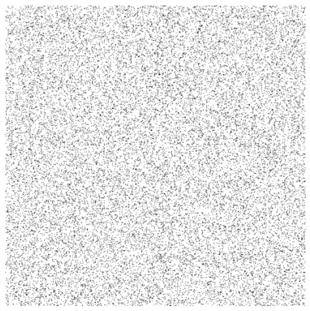


Figure 22. Uniform distribution with 10% black pixels

VIII. Conclusions

In this paper, we have proposed a simple parallel algorithm for the Euclidean distance map and shown an intuitive GPU implementation of the proposed algorithm. In the GPU implementation, we have considered many programming issues of the GPU system such as coalesced access of global memory and shared memory bank conflicts. We have implemented our parallel algorithm in the following three modern GPU systems: Tesla C1060, GTX 480 and GTX 580, respectively. The experimental results have shown that, for an input binary image with size of 9216×9216, our implementation can achieve a speedup factor of 54 over the sequential algorithm implementation. On the other hand, we have also presented that the density of black pixels in an input image affects the performance of the proposed GPU implementation.

References

[1] NVIDIA Corp., “CUDA ZONE,” http://developer.nvidia.com/category/zone/cuda-zone.

[2] R. Farivar, D. Rebolledo, E. Chan, and R.H. Campbell, "A parallel implementation of k-means clustering on GPUs," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp.340–345, July 2008.

[3] P. Harish and P.J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," Proceedings of the 14th International Conference on High Performance Computing, pp.197–208, 2007.

[4] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," Proceedings of International Workshop on Challenges on Massively Parallel Processors, pp.313–319, 2011.

[5] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," International Journal of Networking and Computing, vol.1, no.2, pp.260–276, 2011.

[6] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," Proceedings of International Workshop on Challenges on Massively Parallel Processors, pp.320–326, 2011.

[7] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," Proceedings of International Workshop on Advances in Networking and Computing, pp.279–280, 2010.

[8] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," Proceedings of International Conference on Networking and Computing, pp.153–159, 2011.

[9] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," Proceedings of Asilomar Conference on Signals, Systems, and Computers, pp.171–175, Oct. 2008.

[10] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," Proceedings of International Parallel and Distributed Processing Symposium, pp.1–8, 2010.

[11] Y. Mochizuki, A. Imiya, and A. Torii, "Circle-marker detection method for omnidirectional images and its application to robot positioning," Proc. of International Conference on Computer Vison, pp.1–8, 2007.

[12] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using hough transform on the GPU," Proc. of International Workshop on Challenges on Massively Parallel Processors (CMPP), pp.313–319, Dec. 2011.

[13] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman, "Linear time Euclidean distance transform algorithms," IEEE Trans. Pattern Analysis and Machine Intelligence, vol.17, no.5, pp.529–533, May 1995.

[14] L. Chen, "Optimal algorithm for complete Euclidean distance transform," Chinese J. Computers, vol.18, no.8, pp.611–616, 1995.

[15] L. Chen and H.Y.H. Chuang, "A fast algorithm for Euclidean distance maps of a 2-d binary image," Information Processing Letters, vol.51, pp.25–29, 1994.

[16] T. Hirata, "A unified linear-time algorithm for computing distance maps," Information Processing Letters, vol.58, pp.129–133, 1996.

[17] A. Fujiwara, T. Masuzawa, and H. Fujiwara, "An optimal parallel algorithm for the Euclidean distance maps of 2-d binary images," Information Processing Letters, vol.54, pp.295–300, 1995.

[18] T. Hayashi, K. Nakano, and S. Olariu, "Optimal parallel algorithm for finding proximate points, with applications," IEEE Transactions on Parallel and Distributed Systems, vol.9, no.12, pp.1153–1166, Dec. 1998.

[19] S. Pavel and S.G. Akl, "Efficient algorithms for the Euclidean distance transform," Parallel Processing Letters, vol.5, no.2, pp.205–212, 1995.

[20] Y.-H. Lee, S.-J. Horng, T.-W. Kao, F.-S. Jaung, Y.-J. Chen, and H.-R. Tsai, "Parallel computation of exact Euclidean distance transform," Parallel Computing, vol.22, no.2, pp.311–325, 1996.

[21] L. Chen, P. Yi, Ch. Yixin, and X. Xiaohua, "Efficient parallel algorithms for Euclidean distance transform," The Computer Journal, vol.47, no.6, pp.694–700, 2004.

[22] Li. K and Z.S. Q, Parallel computing using optical interconnections, Boston, USA, Kluwer Academic Publishers, 1998.

[23] NVIDIA Corp., "NVIDIA CUDA programming guide version 4.1," 2011.

[24] NVIDIA Corp., "Tesla C1060 Computing Processor". http://www.nvidia.com/object/product_tesla_c1060_us.html

[25] NVIDIA Corp., "GeForce gtx 480". http://www.nvidia.com/object/product_geforce_gtx_480_us .html

[26] NVIDIA Corp., "GeForce gtx 580". http://www.nvidia.com/object/product-geforce-gtx-580-us.html

[27] F.P. Preparata and M.I. Shamos, Computational geometry: An introduction, third corrected printing edition, Berlin: Springer-Verlag, 1990.

[28] NVIDIA Corp., "CUDA C best practice guide version 4.1," 2012.

[29] Intel Corp., "Intel core i7 processor". http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html

[30] NVIDIA Corp., "Fermi White Paper". http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

完全 k 部グラフにおける移動ビザンチン合意問題アルゴリズムの提案

佐々木徹 * 山内由紀子 * 来嶋秀治 * 山下雅史 *
(*九州大学 工学部 電気情報工学科)

1　はじめに

本研究では、完全 k 部グラフ ($k \geq 4$) 上の同期システムにおける移動ビザンチン故障モデルでの合意問題を解くアルゴリズムを提案する。移動ビザンチン故障モデルでは、ビザンチン故障がプロセスを移動し、プロセスは故障している間はアルゴリズムに関係なく任意の振る舞いをする。

ビザンチン故障が静止している場合の合意問題についてはすでに研究が行われており、n を全プロセス数とすると、ビザンチン合意問題を解くには完全グラフでは故障数は $\frac{n}{3}$ 未満 [1]、不完全グラフではさらに各プロセスが故障数の 2 倍個より多くのプロセスとの通信路を持つ必要がある [2]。また完全グラフにおける移動ビザンチン問題については、各ラウンド毎に故障が移動する場合、故障しないプロセスを 1 つ設けると故障数が $\frac{n}{6}$ 未満の時に解けることが分かっている [3]。

本研究では、各ラウンドの故障プロセス数がおよそ $\frac{n}{6}$ 以下の時、完全 k 部グラフにおいて高々$3n$ ラウンドで合意が得られることを示す。

2　モデル

本研究では、n 個のプロセスから成る分散システムを想定する。各プロセスは他のプロセスとメッセージの送受信により通信を行う。送信、受信、内部計算を 1 ラウンドとする同期モデルで、ネットワークは完全 k 部グラフ、プロセス数を n、各パータイトのプロセス数は一定でその数を m とする ($n = km$)。同時に故障の許されるプロセスの数を t とする。また、故障が移動できるプロセスの集合を S とする。

3　移動ビザンチン合意問題の定義

初期値の全集合を $B = \{0, 1\}$ 、プロセス ID を $\{1, \cdots, n\}$ とする。各プロセス p は初期値と合意値を格納する局所変数 d_p と v_p を持つとする。移動ビザンチン故障が存在するモデルにおいて各プロセス p が以下の条件を満足する合意値を決定するためのアルゴリズムを設計する問題を移動ビザンチン合意問題 (*Mobile- Fault Byzantine Agreement* problem, MBA) という。

(合意性) すべての正常プロセスは同じ値を合意値とする。

(決定性) すべての正常プロセスはいつかは合意値を決定する。

(妥当性) すべての正常プロセスの初期値が同じ値だった場合、正常プロセスはその値を合意値にしなければならない。

(合意維持性) 一度正常プロセス間である値で合意を達成した場合、故障プロセスが変わっても新たな正常プロセスは再びその値で合意しなければならない。

4　k-Partite_ MPK

提案アルゴリズムは 3 つのラウンドを 1 フェーズとし、このフェーズを繰り返し行うものである。第 1 ラウンドで各プロセスは自分の値をブロードキャストし、同じ値がある値未満であれば、各フェーズ毎に 1 つ定められるプロセス (フェーズキング) が第 2 ラウンドでブロードキャストした値を採用する。ただし、フェーズキングと同じパータイト内のプロセスは値を受け取ることができない。第 3 ラウンドは、フェーズキングから直接値を受け取っていないプロセスも合意するためのラウンドである。あるプロセス p のアルゴリズムの動作を以下に示す。

第 1 ラウンド：最初のフェーズのみ初期値 d_p を合意値 v_p に格納する。v_p をブロードキャストする。他のプロセスから受け取った値を一次元配列 $rec1_p$ に格納する。$rec1_p$ をもとに新たに v_p と c_p を決定する。1 が $\frac{(k-1)m+1}{2}$ 以上ならば、$v = 1$,$c =$ (受け取った 1 の個数) とし、それ以外ならば $v = 0, c =$ (1 以外の個数) とする。

第 2 ラウンド：$rec1_p$ をブロードキャストする。プロセス ID が $l \ mod \ n+1$(自分がフェーズキング) ならばこのとき v_p もブロードキャストする。他のプロセスから受け取った $rec1$ を二次元配列 $rec2_p$ に格納する。$rec2_p$ をもとに $rec1_p$ が正しいか計算する。正しくないと判断した場合は $rec2_p$ から v_p と c_p を再計算する。$c_p < (k-1)m - 2t + 1$ ならば送られた v を新たに v_p とする。

第 3 ラウンド：再び v_p をブロードキャストし、受け取った値をもとに第 1 ラウンドと同様の方法で v_p を定める。

定理 1 完全 k 部グラフ ($k \geq 4$) では、$m > \frac{6t-3}{k-3}$、$|S| = n - 1$ のとき、*MBA* を解くことができる。

5　今度の課題

本研究で与えられた条件のもとでの、故障数の制約の上界および下界の発見を今後の課題としたい。

参考文献

[1] 増澤利光, 山下雅史, 適応的分散アルゴリズム, 共立出版, 2010.
[2] D. Dolev,The Byzantine generals strike again, Stanford University, STAN-CS-81-846, 1982.
[3] J. A. Garay, Reaching (and maintaining) agreement in the presence of mobile faults, In Proc. of 8th International Workshop on Distributed Algorithms, pp.253–264, 1994.

動的ネットワークにおける関数監視問題の定式化について

神崎 裕信 [†], 泉 泰介 [†] 和田 幸一 [††]

[†] 名古屋工業大学大学院工学研究科情報工学専攻〒 466-8555 愛知県名古屋市昭和区御器所町
[††] 法政大学大学院工学研究科情報電子工学専攻〒 184-8584 東京都小金井市梶野町 3-7-2

1 はじめに

時間によって構造が変化する動的ネットワーク G をサーバ S が監視する問題を考える．S の仕事は任意の時刻 kt において $G_i(0 \leq i \leq t)$ を入力とする関数 f の値を出力することである．このような問題を動的ネットワーク監視問題と呼ぶことにする．

この問題の自明な解は，任意の時刻において各ノードが自分自身の隣接情報が変化するたびに，即座にその情報を S に送信することである．そうすれば S は常にネットワーク $G_0, G_1, \cdots, G_i$ の全ての履歴が保持することができ，任意の時刻 t において目的関数 f の厳密な計算が可能となる．このアルゴリズムの時刻 k 区間当たりの通信メッセージ数はノード数 n に変化がないと仮定すれば，$\Theta(kn)$ であり，通信ビット数は G 中の各ノードが自身の隣接ノードに関する情報を送信する必要があるため，重みなしグラフの場合で $O(kn^2)$ となる．しかしながら一般には，f の計算について，ネットワークの完全な情報が常に必要であるとは限らない．また，f の値の厳密な計算を必要としない場合も，ネットワークの情報の一部のみからその近似値を計算できる可能性がある．

この問題を緩和し，求める S の出力に一定の誤差 ε を認めた場合の最適解は自明ではない．あるノードの隣接状態が 1 箇所変化するたびに S に報告せず，隣接状態の変化量が大きくなってから，情報を圧縮して送信することなどにより，通信メッセージ数と通信ビット数をそれぞれ $o(nk), o(n^2k)$ に抑えられる可能性がある．この緩和した動的ネットワーク監視問題においてどのような目的関数ならば，通信コストを効率化にできるか，あるいはできないかを明らかにすることは，ノード数が莫大な Peer-to-Peer ネットワークや無線センサネットワークなどの動的ネットワークにおける異常検出などの分野で有用である．この論文では，動的ネットワークにおける関数監視問題を定式化し，現時点で考えられる問題の解法へのアプローチ方法を紹介する．

2 問題の定式化

2.1 ネットワークモデル

動的ネットワーク G の監視問題を，重み付き有向グラフ $G = (V, E)$ でモデル化する．V は全時刻において G に含まれる可能性のあるノードの集合とし，$E \subset V \times V$, $|V| = n$ とする．時刻 $t \in N$ における G の状態を $G_t = (V_t, E_t = V_t \times V_t)$ で表す．また，時刻 t における辺の重み付けを関数 $w_t : e \in E_t \to \mathbb{R}$ で表す．

2.2 目的関数と誤差のモデル

この問題の目的は G を監視するサーバ S が G を入力する目的関数 f について，任意の時刻 t における $P(G_t)$ の（誤差を許した）値を出力することである．この目的関数 f の推定値に許す誤差には f の種類に応じて以下のようないくつかのバリエーションが考えられる．

1. f の値域が二値，つまり $f : G \times \sigma \to 0, 1$ の場合
   - f の真の値が 0 の時に確率 $0 \leq \varepsilon_0 < 1$ で 1 を出力
   - f の真の値が 1 の時に確率 $0 \leq \varepsilon_1 < 1$ で 0 を出力
2. f が二値関数だが，中間出力 p が閾値 γ を超えた場合に 1, 超えなかった場合に 0 を出力する場合
   - $p \leq (1 - \varepsilon_0)\gamma$ ならば 0 を出力 $(0 \leq \varepsilon_0 < 1)$
   - $p \geq (1 + \varepsilon_1)\gamma$ ならば 0 を出力 $(0 \leq \varepsilon_1)$
   - $\varepsilon_0 > 0$ かつ $\varepsilon_1 > 0$ の場合，$(1 - \varepsilon_0)\gamma < p < (1 + \varepsilon_1)\gamma$ の出力は未定
3. f の値域が十分に広い場合
   (a) 真の出力 p に対し $(1 - \varepsilon_0)p \leq p \leq (1 + \varepsilon_1)p$ の範囲の値を常に出力
   (b) 真の出力 p に対し確率 $1 - \varepsilon$ で p を出力し，確率 $0 < \varepsilon < 1$ でそれ以外の値を出力

などのモデルが考えられる．

2.3 時刻・通信モデル

単純化のため同期ラウンドモデル用いる．1 ラウンドはサーバを含む任意のノード間の通信遅延よりも十分に大きく，あるラウンド i で生成されたメッセージはラウンド i の終わりにあて先に配送され，読み出し可能となる．つまり, 一般的な同期メッセージパッシングモデルを利用する．ラウンド t 中にサーバ S を含む各ノードが可能な動作はラウンド $t-1$ に自分宛てに送信されたメッセージの読み出し，ローカルでの計算, メッセージの送信である．

ノード v_i がラウンド t 中に送信可能なメッセージの宛先は，v_i から出る有効辺 $(v_i, v_j) \in E_t$ が存在する任意のノード v_j とサーバ S である．また，S は任意のノード宛にメッセージ送信が可能とする．このモデルでは，メッセージの消失・エラーは発生しないとする．

2.4 ネットワークの変化タイミング

ネットワークの変化とは，G に含まれる頂点と辺の追加・削除および辺の重みの変化を指す．ネットワークの変化は全てラウンドの開始時に発生し，ラウンド中の変化は起きないものとする．各ノード v_i は任意のノード $v_j \in V_T$ について有向辺 (v_i, v_j) が追加または削除されると即座に検知でき, 辺の重み $w_t((v_i, v_j))$ を常に把握できる．ラウンド t の開始時に辺 (v_i, v_j) が削除された場合，v_i は v_j がグラフから削除された $(v_j \notin V_t)$ のか，そうでない $(v_j \in V_t)$ のかの判別ができないとする．変化の内容については一切の仮定を置かず, 頂点が全て入れ替わる $(V_t \cap V_{t+1} = \emptyset)$，$G_t$ が非連結になる，ラウンド間でネットワークが変化しない（$V_t = V_{t+1}$ かつ $E_t = E_{t+1}$）場合などを許す．

2.5 アルゴリズムの評価基準

この問題を解くアルルゴリズムの基本的な評価基準は推定値を求めるのに必要な通信メッセージ数と, 通信ビット数である．より具体的には，以下の二種類が考えられる．

- a_1 任意のラウンド t における，総通信メッセージ数および総通信ビット数の最大値
- a_2 十分に長い k 区間における総通信メッセージ数および総通信ビット数を k で割った平均値の最大値

a_1 は各ラウンドでの動作がほぼ均一なアルゴリズムにおいては問題ないが，通信の多いラウンドと少ないラウンドに分かれるアルゴリズムの総通信量を過大に見積もる可能性が存在する．例として，任意のラウンドで常に $O(\log k)$ 個のメッセージを送信するアルゴリズム A_1 と，任意の k 区間中に $O(1)$ 個のメッセージを送信するラウンドが $k-1$ 個存在し，$O(k)$ 個のメッセージを送信するラウンドが 1 個存在するアルゴリズム A_2 を a_1 と a_2 で比較する．1 ラウンドの送信メッセージ数の最大値は A_1 は $O(\log k)$, A_2 は $O(k)$ で，A_2 の方が性能が悪いように見える．一方で，k 区間の総通信メッセージ数を k で割った値は，A_1 が $O(\log k)$，A_2 が $O(1)$ であり，A_2 の方が優れている．このように，a_1 と a_2 で優劣が逆転する例が存在してしまう．一方で，輻輳や混信の可能性を考慮しなければならないネットワークなどでは，全体の通信量を増やしてでも，1 ラウンドの通信量の最大値を低く保ちたい場合が考えられる．したがって，常に a_2 のみを評価基準とするのも適切ではない．

以上より，扱う問題の性質に応じて a_1, a_2 適切な評価基準を選ぶ必要がある．

3 問題へのアプローチ方法

動的ネットワークにおける関数監視問題を解くにあたり，現時点で考えられる 2 種類のアプローチとしてデータストリームアルゴリズムと，特性検査アルゴリズムを紹介する．

3.1 データストリームアルゴリズム

データストリームアルゴリズムは，入力のサイズに対してメモリ空間が小さく，入力を全て保持できないような環境で入力データストリームを先頭から順番に読み込んで問題を解くアルゴリズムである．この分野ではすでに多くの関数 f について，誤差 $1 \pm \varepsilon$ の推定値を大きさが $1/\epsilon$ と $\log n$ の多項式程度に限られたメモリ空間を使って求めるアルゴリズムが存在する．

ネットワーク G の変化の時系列そのものを入力データストリームとみなし，サーバを含むノード間の通信量をメモリ空間のサイズと対応させることで，関数監視問題を近似的にデータストリームアルゴリズムの問題に変換できる可能性がある．ある目的関数 f に関する関数監視問題を，データストリームアルゴリズムですでに扱われている問題に帰着できれば，通信ビット数を $1/\epsilon, \log n$ の多項式に抑えて関数監視問題を解けるかもしれない．分散環境でのストリーミングアルゴリズムの応用例に，functional monitaring[1)] が存在する．この研究では k 個のノードと 1 個のサーバが存在する分散ネットワークにおいて, 入力ストリームを入力とする関数 f の出力をが閾値 τ を越える時刻を誤差 ε で監視する問題を，$(k, f, \tau, \varepsilon)$functional monitoring と定式化し，頻度モーメント (frequency moments) 関数 $F_p\,(p = 0, 1, 2)$ について解くアルゴリズムを示した．このアルゴリズムでは，各ノードが入力を一定個数受け取ってからサーバに情報を送ることで，通信ビット数を削減している．そして通信ビット数の上界は k と $1/\epsilon$ の多項式であり，ストリームの長さに依存しない．

3.2 性質検査

性質検査（property testing）アルゴリズムは巨大なデータ集合に関する決定問題を高速 (データサイズの sublinear または定数時間) に解く（乱択）アルゴリズムである．性質検査は入力データの一部だけを読み込み，局所部分の性質から全体の性質を推定することで高速化を達成しており，通信量の削減が期待できる．Chazelle[2] らは，Goldreich[3] らによるグラフの連結性に関する性質検査アルゴリズムを応用し，sublinear 時間で重み付きグラフの最小全域木（MST）のサイズを推定するアルゴリズムを示した．このアルゴリズムの実行時間は，辺の重みの集合 $\{1, \cdots, w\}$ のサイズ w, MST の推定誤差 ε, グラフの平均次数 d に対し $O(dw\varepsilon^{-2} \log \frac{dw}{\varepsilon})$ である．

グラフ問題おけるランダムサンプリングの別の例としては，Feige[4] の研究がある．この研究ではランダムサンプリングした $O(\sqrt{n})$ 個の頂点の平均次数から，グラフ全体の平均次数の 2-近似値を求めるアルゴリズムを提案している．ランダムサンプリングが可能であることを仮定すれば，このアルゴリズムは即座に平均次数をもとめる関数の 2-近似，$O(\sqrt{n}k)$ メッセージ複雑度のアルゴリズムを与えることになる．

4 まとめ

この論文では，動的ネットワークにおける関数監視問題を定式化し，問題を解くアプローチとして，データストリーミングアルゴリズムと特性検査アルゴリズムを示した．今後の課題は，実際にこの問題を sublinear な通信コストで解くことのできる具体的な関数およびアルゴリズムを見つけることである．

参考文献

1) Graham Cormode, S. Muthukrishnan, and Ke Yi. Algorithms for distributed functional monitoring. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 1076–1085, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

2) Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6):1370–1379, June 2005.

3) Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. In *Algorithmica*, pages 406–415, 1997.

4) Uriel Feige. On sums of independent random variables with unbounded variance and estimating the average degree in a graph. *SIAM J. Comput.*, 35(4):964–984, April 2006.

弱安定アルゴリズムに対する遷移グラフに関する考察

服部 雄典 † 泉 泰介 † 和田 幸一 ††

† 名古屋工業大学大学院工学研究科情報工学専攻
〒466-8555 愛知県名古屋市昭和区御器所町
†† 法政大学大学院工学研究科情報電子工学専攻
〒184-8584 東京都 小金井市梶野町 3-7-2

1　はじめに

自己安定システムとは，任意の初期状態から正当な状況への復帰が保証されるシステムである．その特徴から，自己安定アルゴリズムは任意の種類の一時故障に対する耐故障性やネットワークの形状変化に対する適応性を実現するための有効な手法として認知されている．

一般に，自己安定性は任意の初期状態からの復帰という強い性質を保証するため，その設計は容易ではなく，またある種の問題に対しては実現不可能であることも知られている．そのため，自己安定が保証する収束性 (正当な状況に必ず到達する) と閉包性 (正当な状況から抜け出すことはない) のうち閉包性を弱めた弱安定の概念が導入されている．論文 [1] では自己安定アルゴリズムと比べた弱安定アルゴリズムの利点などが挙げられている．弱安定アルゴリズムは自己安定アルゴリズムと比較して設計が容易なため，弱安定性から強公平の仮定の下で自己安定性を実現する方法がある．この方法の一つとして，Gouda の論文 [2] が挙げられる．[2] では，Gouda の強公平という概念の導入によって状況の数が有限である弱安定アルゴリズムを自己安定なアルゴリズムに変換できると示している．

自己安定アルゴリズムでは収束に要する時間で効率を測るが，弱安定アルゴリズムでは本質的に収束時間は定義することはできない．そこで，弱安定アルゴリズムの効率を測る方法として，遷移グラフの構造に注目しその効率の評価方法を検討する．

2　諸定義

2.1　モデルの定義

分散システムとは複数のプロセスにより構成されるシステムのことである．分散システムのモデルはプロセスの集合を頂点集合 V，プロセス間の通信リンクを辺集合 E とする有向グラフ G=(V,E) で表される．

各プロセスは状態変数を持つ．プロセス上の計算は，自分自身およびその隣接プロセスの状態を参照し，自身の状態を更新することにより行われる．システム中の全プロセスの状態を状況と呼び，すべての状況の集合を記号 C で表す．また，与えられた問題の仕様 SP が満たされている状況のことを正当な状況と呼び，正当な状況の集合を記号 $\mathcal{C}_{\mathcal{L}}$ で表す．また，正当な状況以外の状況の集合を記号 $\mathcal{C}_{\bar{\mathcal{L}}}$ で表す．

本論文では匿名性のある分散システムを扱う．匿名性のある分散システムではプロセスは固有の識別子を持たず，入次数と出次数により他のプロセスを区別する．ただし，説明のためにプロセスに識別子を付ける場合がある．

2.2　動作

各プロセスは自身の状態をローカルアルゴリズムの実行によって変化させる．ローカルアルゴリズムは以下の形式で表される．

- ⟨label⟩ :: ⟨guard⟩ → ⟨statement⟩

⟨label⟩ はその動作の名前を表し，⟨guard⟩ は動作を実行するプロセス自身の状態や隣接プロセスの状態に関する論理式で表される．⟨guard⟩ が true の場合，⟨statement⟩ に記述される動作を実行し状態の変更を行う．また，各プロセスの動作速度については何も仮定をしない．

ある時点 t における状況 $\mathcal{C}_t$ において ⟨guard⟩ が true となるプロセスのことを動作可能なプロセスと呼ぶ．プロセスの動作の結果，システムの状況が変化することを遷移と呼ぶ．すべての遷移の集合を記号 δ とし，状況 $\mathcal{C}_i$ から $\mathcal{C}_j$ への遷移が存在する場合，$(\mathcal{C}_i,\mathcal{C}_j) \in \delta$ のように表す．

2.3　分散システム上での実行

分散システム上での実行 E は状況の有限又は無限列 $\mathcal{C}_1,\mathcal{C}_2,\ldots$ で表される．ただし，任意の $i \geqq 1$ に対して $(\mathcal{C}_i,\mathcal{C}_{i+1}) \in \delta$ でなければならない．状況 $\mathcal{C}_i$ からはじまり，$\mathcal{C}_j$ で終わるような実行 $\mathcal{C}_i,\ldots,\mathcal{C}_j$ が存在するとき，状況 $\mathcal{C}_i$ から $\mathcal{C}_j$ へ到達可能と呼び，$(\mathcal{C}_i,\mathcal{C}_j)$ のように表す．

2.4　スケジューラ

スケジューラとは，ある時点 t における状況 $\mathcal{C}_t$ で動作可能なプロセスの中から動作を実行させるプロセスを選択するものである．プロセスの選択の仕方をスケジューリングと呼び，公平性により以下のように分類される．

- 不公平 (unfair): プロセスの選択の仕方に一切の仮定を置かない
- 弱公平 (weakly fair): すべての継続的に動作可能であるプロセスはいずれ選択される
- 強公平 (strongly fair): すべての無限にしばしば動作可能なプロセスはいずれ選択される
- 同期 (synchronous): 常にすべての動作可能なプロセスが選択される

また，同時に動作可能なプロセスの数によって以下のように分類される．

- 集中: 同時に動作できるプロセスの数は高々一つ．
  C-デーモンとも呼ばれる．
- 分散: 同時に複数のプロセスが動作可能．
  D-デーモンとも呼ばれる．

また，分散システム S とスケジューラ σ が与えられたとき，とりうるすべての実行の集合を $e(S,\sigma)$ で表す．

2.5　自己安定・弱安定

自己安定とは，分散システム上でのいかなる一時故障にも耐えうる性質である．自己安定システムでは故障が生じている状況を初期状況とみなして，その状況から正当な状況へ復帰することを保証する．また，一度正当な状況へ復帰すると再び故障が生じない限り正当な状況から抜け出すことはない．

定義 2.1（自己安定システム） 分散システムにおいて次の二つの性質が満たされる場合，その分散システムは自己安定であるという．

- 収束性: $e(S,\sigma)$ 中の任意の実行はいずれ $\mathcal{C}_\mathcal{L}$ の一つに到達する．
- 閉包性: $\mathcal{C}_\mathcal{L}$ 中の任意の状況からはじまる任意の実行 $E \in e(S,\sigma)$ は常に仕様を満たす．

弱安定とは，自己安定が保証する二つの性質のうち収束性を弱めたものである．自己安定ではどんな実行でも必ず収束することが保証されるのに対し，弱安定ではどの状況からも収束するような実行が存在することが保証されている．

定義 2.2（弱安定システム） 分散システムにおいて次の二つの性質が満たされる場合，その分散システムは弱安定であるという．

- 弱い収束性: $\mathcal{C}_{\bar{\mathcal{L}}}$ 中の任意の状況に対して，$\mathcal{C}_\mathcal{L}$ に到達するような実行 E が少なくとも一つ存在する．
- 閉包性: $\mathcal{C}_\mathcal{L}$ 中の任意の状況からはじまる任意の実行 E は常に仕様を満たす．

3　遷移グラフ

アルゴリズム A とグラフ G，スケジューラ σ が与えられると，とりうる状況の集合 C と遷移関係 δ が一意に決まる．そのためスケジューラを σ とするグラフ G 上でアルゴリズム A を実行する分散システムを有向グラフとして定義できる．この有向グラフを自己安定アルゴリズムや弱安定アルゴリズムの性質を調べるために使用し，本論文では遷移グラフとして定義する．

定義 3.1（遷移グラフ） スケジューラを σ とするグラフ G 上でアルゴリズム A を実行する分散システムに対して，その状態集合 C および遷移 δ により決まる有向グラフ $S = (C, \delta)$ を A の遷移グラフと呼ぶ

弱安定アルゴリズムの遷移グラフには以下のような性質がある．

- 自己安定アルゴリズムの遷移グラフとは異なり，遷移グラフの正当でない状況の集合に閉路が存在する．
- 正当でない状況の集合は複数の強連結成分に分けられ，不可逆な構造となる．自己安定アルゴリズムの場合は強連結成分を作ることができない．

この構造のことを遷移グラフの階層構造と定義する．また，階層構造において正当な状況に近い階層を下位層，遠い階層を上位層とする．

命題 3.1 (弱安定アルゴリズムの遷移グラフの階層構造)
弱安定アルゴリズムの遷移グラフの階層構造は次の二つの性質を持つ．

- ある階層に属する任意の状況から，その階層より上位の層に属する状況への経路は存在しない．
- ある階層に属する任意の状況から，その階層より下位の層に属する状況への経路が存在する．

この階層構造を調べるために，弱安定アルゴリズムの遷移グラフの強連結成分を頂点とする有向グラフ $\widetilde{S}$ を定義する．

定義 3.2 有向グラフ $\widetilde{S}$ は，遷移グラフの強連結成分を頂点集合とし，強連結成分間の遷移を辺集合とする有向グラフである．

階層構造が持つ特徴はアルゴリズムや対象とするグラフごとに異なる．そのため，弱安定アルゴリズムの新たな評価手法としてこの遷移グラフの階層構造の性質を提案する．

4　アルゴリズム

アルゴリズムの紹介する．

4.1　トークン巡回アルゴリズム

匿名性のある単方向リング上かつ強公平スケジューラの下で，トークン巡回問題を解く決定性弱安定アルゴリズムを紹介する．この論文で扱うトークン巡回問題とは以下の定義である．

定義 4.1 トークン巡回問題とは，以下の二つの条件を満たすようにネットワーク上でトークンを巡回させる問題である．

- ネットワーク上には，ちょうど一つのトークンが存在する
- ネットワーク上のすべてのプロセスは，無限にしばしばトークンを保持する

この問題を解く弱安定アルゴリズムが [1] で提案されている．

```
Algorithm 1 Code for every process $i$
Variable: $v_i \in \{0, \ldots, m_N - 1\}$
Macro:
  $PassToken_i \equiv v_i := (v_{i-1} + 1) \bmod m_N$
Predicate:
  $Token_i \equiv [v_i \neq ((v_{i-1} + 1) \bmod m_N)]$
Action:
  A :: $Token_i \rightarrow PassToken_i$
```

各プロセス i は変数 v_i を持ち，$Token_i$ が true となる場合プロセス i はトークンを保持

している．$PassToken_i$ はプロセス i がプロセス $i+1$ にトークンを渡すマクロである．N はネットワーク上のプロセス数であり，m_N は N を割り切れない最小の整数とする．

定理 4.1 [1] アルゴリズム 1 は匿名性のある単方向リング上かつ強公平スケジューラの下でトークン巡回問題を解く決定性弱安定アルゴリズムである．

補題 4.1 ネットワーク上のトークンを保持するプロセスの数は 1 以上である．

m_N は N を割り切れない整数であるから，すべてのプロセスが $v_i = ((v_{i-1}+1) \bmod m_N)$ を満たすことはできない．すなわち，少なくとも 1 つのプロセスは $Token_i$ を満たす．よって，補題 4.1 が成り立つ．

補題 4.2 ネットワーク上のトークンの数が増えることはない．

トークンを保持していないプロセス i がトークンを獲得するには，プロセス $i-1$ が $PassToken_{i-1}$ を実行しなければならない．プロセス $i-1$ が $PassToken_{i-1}$ を実行するには，プロセス $i-1$ は $Token_{i-1}$ を満たしていなければならない．プロセス $i-1$ が $PassToken_{i-1}$ を実行すると，プロセス $i-1$ は $Token_{i-1}$ を満たさなくなる．よって，補題 4.2 が成り立つ．

このアルゴリズムの遷移グラフ S と $\widetilde{S}$ は以下の図 1, 図 2 のようになる．図 1 の円の内部は分散システムの状況を示している．

4.2 リーダ選挙アルゴリズム

匿名性のある木上かつ強公平スケジューラの下で，リーダ選挙問題を解く決定性弱安定アルゴリズムを紹介する．この論文で扱うリーダ選挙問題とは以下の定義である．

定義 4.2 リーダ選挙問題とは，以下の二つの条件を満たすようにネットワーク上でただ一つのリーダを選出する問題である．

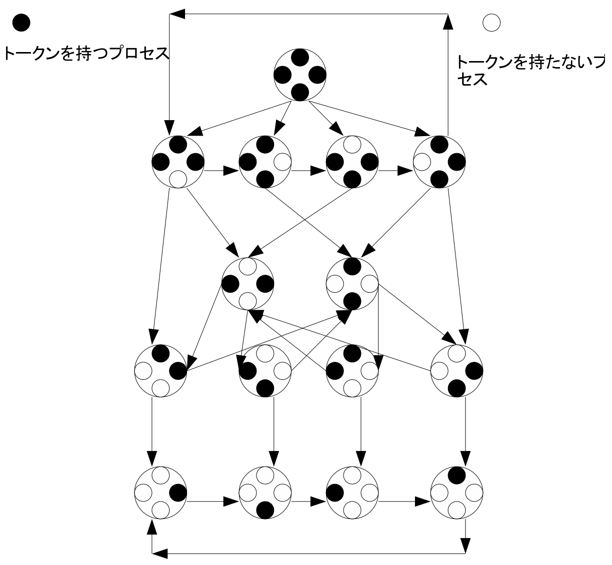


図 1 トークン巡回アルゴリズムの遷移グラフ (N=4)

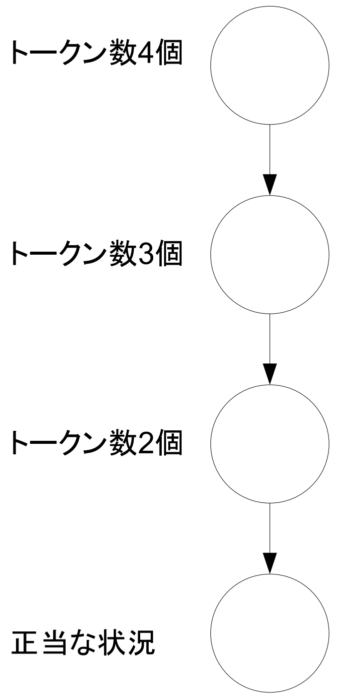


図 2 トークン巡回アルゴリズムの $\widetilde{S}$(N=4)

- ネットワーク上のリーダとして選出されたプロセスは, 自身が選出されていることを認識できる.
- ネットワーク上のリーダ以外のすべてのプロセスは, 他のプロセスが選出されていることを認識できる.

この問題を解く弱安定アルゴリズムが [1] で提案されている.

Algorithm 2 Code for every process p

```
Variable: $v_p \in L_p \cup \{\perp\}$
Macro:
  $Child_p \quad \equiv \quad \{q \in L_p : v_q = p\}$
Predicate:
  $Leader_p \quad \equiv \quad (v_p = \perp)$
Actions:
    $\mathrm{A}_1 \quad :: \quad (v_p \neq \perp) \wedge (|Child_p| = \delta_p)$
    $\rightarrow v_p := \perp$
    $\mathrm{A}_2 \quad :: \quad (v_p \neq \perp) \wedge [L_p \setminus (Child_p \cup \{v_p\})$
    $\neq \emptyset] \rightarrow v_p := (v_p + 1) \bmod \delta_p$
    $\mathrm{A}_3 \quad :: \quad (v_p = \perp) \wedge (|Child_p| < \delta_p)$
    $\rightarrow v_p := \min(L_p \setminus Child_p)$
```

δ_p はプロセス p の隣接プロセスの数である. v_p はプロセス p がリーダに選出したプロセスへのインデックスである. $v_p = q$ のとき, プロセス p はプロセス q をリーダとする. また, $v_p = \perp$ のとき, プロセス p は自身をリーダとする. L_p はプロセス p の隣接プロセスへのローカルインデックスの集合である. すなわち, $L_p = \{0, 1, \ldots \delta_p - 1\}$ である. マクロ $Child_p$ はプロセス p の隣接プロセスのうち, プロセス p をリーダとするプロセスの集合を示す.

動作 A_1 は, 隣接プロセスをリーダとしているプロセス p が, リーダを自身に変更する動作である. 動作 A_2 は, 隣接プロセスをリーダとしているプロセス p が, リーダを他の隣接プロセスに変更する動作である. 動作 A_3 は, 自身をリーダとしているプロセス p が, リーダを隣接プロセスに変更する動作である. A_2 と A_3 において, リーダはインデックスの値に基いて決定される.

定理 4.2 [1] アルゴリズム 2 は強公平スケジューラの下でリーダ選挙問題を解く決定性弱安定アルゴリズムである.

補題 4.3 プロセス p が動作可能であるとき, L_p に含まれるプロセスのうち, 少なくとも一つは動作可能である.

プロセス p が A_1 を使用可能であるとき, プロセス v_p は A_1 または A_2 を使用可能である. プロセス p が A_2 を使用可能であるとき, すべてのプロセス $q \in L_p \setminus (Child_p \cup \{v_p\})$ は A_2 または A_3 を使用可能である. プロセス p が A_3 を使用可能であるとき, すべてのプロセス $q \in L_p \setminus Child_p$ は A_2 または A_3 を使用可能である. 以上から, 補題 4.3 が成り立つ.

4.3 リーダ選挙アルゴリズム（改変）

アルゴリズム 2 の動作 A_3 は, v_p を決定するのに $Child_p$ を参照する. しかし, 動作 A_2 は $Child_p$ を参照せず, インデックス値のみを参照している. すなわち, 動作 A_3 はプロセス p をリーダとする隣接プロセスをリーダに選出しないのに対して, 動作 A_2 はプロセス p をリーダとする隣接プロセスをリーダに選出することがある. しかしながら, [1] のアルゴリズム 2 の正当性の証明によると, この動作は必ずしも必要な動作ではない. そこで, この節では動作 A_2 についても $Child_p$ を参照するように変更したアルゴリズムを導入する.

定理 4.3 アルゴリズム 3 は強公平スケジューラの下でリーダ選挙問題を解く決定性弱安定アルゴリズムである.

この定理はアルゴリズム 2 と同じ議論で証明できる.

補題 4.4 $v_p \neq \perp$ であるプロセス p が動作不可であるとき, プロセス v_p の動作によりプロセス p が動作可能に変わることはない.

Algorithm 3 Code for every process p

```
Variable: $v_p \in L_p \cup \{\perp\}$
Macro:
  $Child_p \quad \equiv \quad \{q \in L_p : v_q = p\}$
Predicate:
  $Leader_p \quad \equiv \quad (v_p = \perp)$
Actions:
    A$_1$  ::  $(v_p \neq \perp) \wedge (|Child_p| = \delta_p)$
    $\rightarrow v_p := \perp$
    A$_2$  ::  $(v_p \neq \perp) \wedge [L_p \setminus (Child_p \cup \{v_p\}) \neq \emptyset]$
           $v_p := (v_p + x) \bmod \delta_p$
    $\rightarrow$      $(x : min\{x \in \{1, 2, \ldots, \delta_p - 1\} :$
           $(v_p + x) \in (L_p \setminus Child_p)\})$
    A$_3$  ::  $(v_p = \perp) \wedge (|Child_p| < \delta_p)$
    $\rightarrow v_p := \min(L_p \setminus Child_p)$
```

アルゴリズム 3 のいずれの動作も，v_q にプロセス $p \in Child_q$ を代入することはない．よって，補題 4.4 が成り立つ．

このアルゴリズムのグラフがスター型の場合とライン型の場合の遷移グラフ S と $\widetilde{S}$ は以下の図 3,4,5,6 のようになる．図 3,5 の円の内部は分散システムの状況を示している．また，プロセス p から q への有向辺が存在するとき，v_p = q とし，p を始点とする有向辺が存在しない場合，$v_p = \perp$ とする．

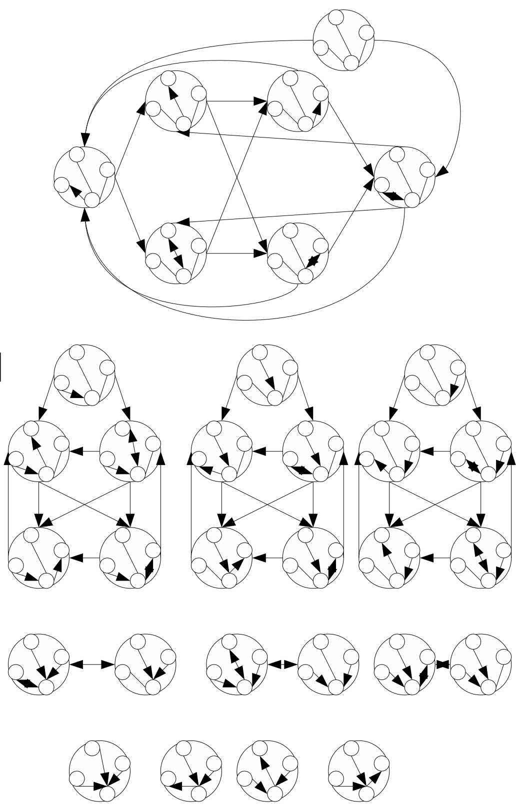


図 3　スター上でのリーダ選挙の遷移グラフ (N=4)

5　各アルゴリズムの比較と考察

5.1　トークン巡回アルゴリズム

補題 4.1 と 4.2 より，ネットワーク上のトークンの数によって正当でない状況の集合を N-1 個の階層に分けることができる．

5.2　リーダ選挙・スター

スター上でのリーダ選挙の次の述語を満たすプロセスの個数により階層が分けられる．ただし，スターの中心のプロセスを c，それ以外のプロセスを b とする．

- $v_b = v_c \wedge v_c \neq v_b$

よって，正当でない状況の集合を N-1 個の階層に分けることができる．

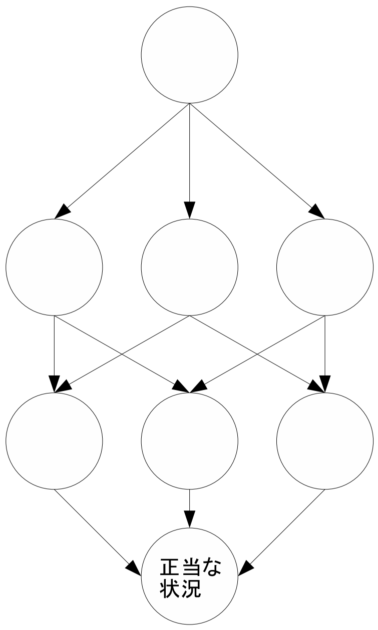


図 4　スター上でのリーダ選挙の $\widetilde{S}$(N=4)

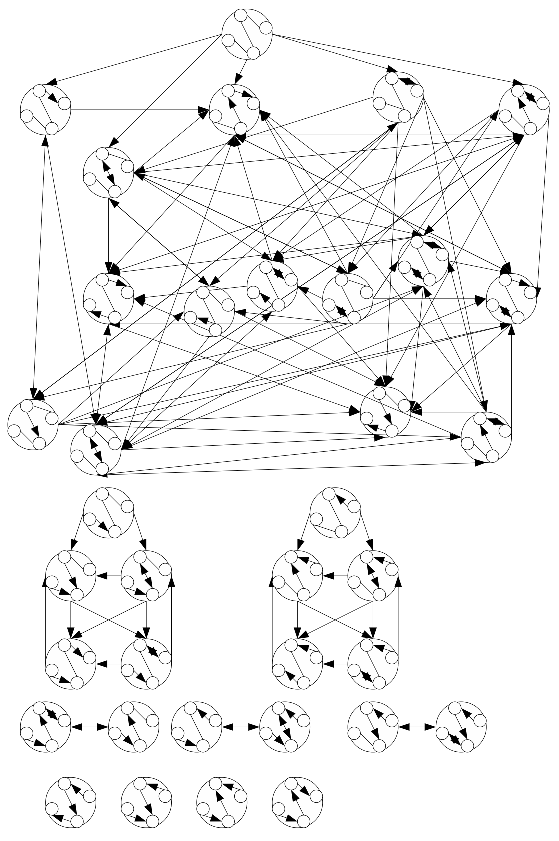


図 5　ライン上でのリーダ選挙の遷移グラフ (N=4)

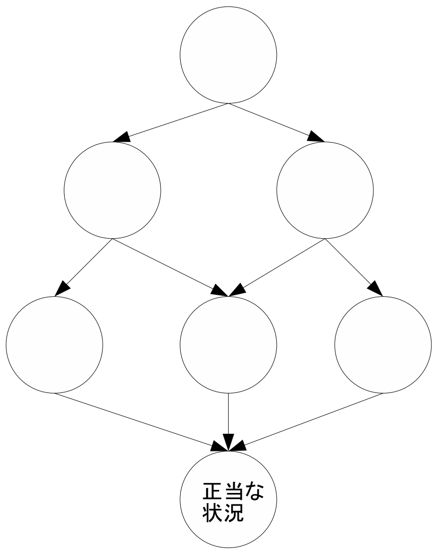


図 6　ライン上でのリーダ選挙の $\widetilde{S}$(N=4)

5.3　リーダ選挙・ライン

ライン上でのリーダ選挙の次の述語を満たすプロセスの個数により階層が分けられる．ただし，ライン上でプロセス p の左隣のプロセスを pl，右隣のプロセスを pr と表記する．

- ラインの両端のプロセスの場合
  $v_p = \mathrm{pr} \land v_{pr} \neq \mathrm{p} \lor v_p = \mathrm{pl} \land v_{pl} \neq \mathrm{p}$
- ラインの両端以外プロセスの場合
  $v_p = \mathrm{pr} \land v_{pr} \neq \mathrm{p} \land$ (pl がこの述語を満たす) $\lor\ v_p = \mathrm{pl} \land v_{pl} \neq \mathrm{p} \land$ (pr がこの述語を満たす)

補題 4.3 より，正当でない状況の集合を N-1 個の階層に分けることができる．

5.4　考察

今回調査を行ったアルゴリズムすべてについて正当でない状況の集合を N-1 個の階層に分けることができた．

6　まとめと今後の課題

今回，弱安定アルゴリズムの評価手法として遷移グラフが持つ性質を提案をした．しかし今回の調査では各強連結成分がどのような構造となっているか，その構造が弱安定アルゴリズムの効率にどのような影響を与えるか，などについては触れていない．そのため，この点に関してはさらなる調査が必要となる．

また，既存の弱安定アルゴリズムの遷移グラフが持つ性質の調査を行った．今回はトークン巡回アルゴリズムとリーダ選挙アルゴリズムを扱ったが他の弱安定アルゴリズムについても調査をする必要がある．

参考文献

[1] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. Self vs. Probabilistic Stabilization. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pp. 681–688. IEEE Computer Society, 2008.

[2] Gouda, Mohamed G. The Theory of Weak Stabilization In *Proceedings of the 5th International Workshop on Self-Stabilizing Systems*

A New Direction for Counting Perfect Matchings

Taisuke Izumi and Tadashi Wadayama
Graduate School of Engineering
Nagoya Institute of Technology
Aichi, Japan
Email: $\{$*t-izumi,wadayama*$\}$*@nitech.ac.jp*

***Abstract*—In this paper, we present a new exact algorithm for counting perfect matchings, which relies on neither inclusion-exclusion principle nor tree-decompositions. For any bipartite graph of $2n$ nodes and Δn edges such that $\Delta \geq 3$, our algorithm runs with $O^*(2^{(1-1/O(\Delta \log \Delta))n})$ time and exponential space. Compared to the previous algorithms, it achieves a better time bound in the sense that the performance degradation to the increase of Δ is quite slower. The main idea of our algorithm is a new reduction to the problem of computing the cut-weight distribution of the input graph. The primary ingredient of this reduction is MacWilliams Identity derived from elementary coding theory. The whole of our algorithm is designed by combining that reduction with a non-trivial fast algorithm computing the cut-weight distribution. To the best of our knowledge, the approach posed in this paper is new and may be of independent interest.**

***Keywords*-counting perfect matchings, exponential algorithm, coding theory, MacWilliams identity**

I. Introduction

Counting perfect matchings in given input graph G is recognized as one of hard combinatorial problems. In particular, the case that G is bipartite has attracted much attention with its long history because of the relation to the computation of permanent, which is a characteristic value of matrices with many important applications. Since counting perfect matchings for bipartite graphs belongs to #P-complete, there seems to be no algorithm which runs within polynomial time for any input. Thus all of the previous studies lies on one (or more) of the following directions: Approximation, restriction of input graphs, or exact exponential algorithms. In this paper, we focus on the third line.

A seminal exponential-time algorithm for counting perfect matchings is Ryser's one based on the inclusion-exclusion principle [1]. For any bipartite graph G of $2n$ vertices, it counts perfect matchings with $O^*(2^n)$ time[1] and polynomial memory space. There has been several improvements following that work: Bax and Franklin have shown an algorithm running with $O^*(2^{(1-1/O(n^{2/3} \ln n))n})$ expected time and exponential space [2]. Servedio and Wan have given an algorithm with a time upper bound depending on the average degree Δ [3]. It achieves $O^*(2^{(1-1/O(\exp(\Delta)))n})$ time and polynomial space. Another approach to this problem is the usage of tree decompositions [4, 5]. By combining the fact that sparse graphs have a treewidth less than $(1-\epsilon)n$ for some constant ϵ (e.g., if $\Delta \leq 3$, $\epsilon \approx 5/6$ holds [6]), we can obtain an algorithm running $O^*(2^{(1-\epsilon)n})$ time. All of these algorithms break $O^*(2^n)$-time barrier in some sense. However, during last 50 years, there has been proposed no algorithm achieving exponential-time speedup for *any* graph, which is a big open problem in this topic.

Our result presented in this paper can be put on the same line. The main contribution is to propose a new algorithm for counting perfect matchings. For any bipartite graph of $2n$ nodes and Δn edges, it runs with $O^*(2^{(1-1/O(\Delta \log \Delta))n})$ time and exponential space. While this algorithm does not settle the open problem stated above, its speed-up factor becomes substantially closer to the exponential compared to the previous algorithms.

An important remark is that the approach we adopt is quite different from any previous solutions. It relies on neither inclusion-exclusion nor tree decomposition. Actually, the main idea is an extremely-simple reduction to the problem of computing the cut-weight distribution of the input graph. The precise construction of our algorithm can be summarized as follows:

- For any *odd* input bipartite graph G of $2n$ nodes and m edges, we can show that the number of G's perfect matchings is equal to the number of elements with weight $m-n$ in its cycle space. In addition, for any bipartite graph G, it is possible to construct the odd bipartite graph $\tilde{G}$ which has the same number of perfect matchings as G, by adding a constant number of nodes.
- By utilizing the primal-dual relation between cycle space and cut space, we can reduce the problem of counting cycle-space elements with weight $m-n$ to computing the weight distribution of the cut space. The technical tool behind this reduction is the use of MacWilliams identity, which is a well-known theorem derived from elementary coding-theory. That identity provides the linear transformation (by so-called *Krawtchouk matrices*) that maps the weight-distribution vector of any cut space to the corresponding cycle space.
- Since the cardinality of the cut space is vertex-exponential, it is easy to construct a naive algorithm

[1] O^* means the Big-O notation with omitting $\mathrm{poly}(n)$ factors.

with $O^*(2^{2n})$ running time. We improve its running time by utilizing the bipartiteness property and a novel technique analogous to separator decompositions.

It should be noted that except for the last step, our approach is applicable to any graphs which may not be bipartite. Our reduction technique can be seen as an algebraic approach to the design of exact algorithms as considered in [7, 8], where several kinds of algebraic transformations are used for appropriate handling of target universes. To the best of our knowledge, this is the first attempt using the transformation by MacWilliams Identity (or equivalently Kratwtchouk matrices) for that objective.

The organization of the paper is as follows: We first presents several notions and definitions in Section II, which includes an tiny tutorial of linear codes. Section III introduces our reduction to cut space. The algorithm to compute the cut-weight distribution is shown in Section IV gives an algorithm computing the cut space. We mention the related work in Section V, and finally conclude the paper in Section VI with the open problems posed by our result.

II. Preliminaries from Coding Theory

A linear code C over $\mathbb{F}_2$ defined by $n \times m$ matrix M is the set of m-dimensional vectors as follows:

$$C = \{\boldsymbol{v}M | \boldsymbol{v} \in \mathbb{F}_2^n\}.$$

The matrix M is called the *generator matrix* of C. By the definition, code C is the linear subspace of $\mathbb{F}_2^m$ spanned by the row vectors of M. The rank of that subspace is denoted by $rank(M)$. Clearly, the number of codewords in C (denoted by $|C|$) is equal to $2^{rank(M)}$. A (m, r)-linear code is the one such that the length of codewords is m and its rank is r.

Let C be a linear code with generator matrix M. The *parity check matrix* H of C is the $m \times (m - \mathrm{rank}(M))$ matrix satisfying $H\boldsymbol{w}^T = \boldsymbol{0}$ for any codeword $\boldsymbol{w} \in C$. It is well-known that there is a duality between generator matrices and parity check matrices: For the code $C^\perp$ with generator matrix H, it is easily verified that $\boldsymbol{v}^T M = 0$ holds for any $\boldsymbol{v} \in C^\perp$. That is, M is the parity check matrix of $C^\perp$. Then the code $C^\perp$ is called the *dual* code of C. Obviously $\boldsymbol{v}^T \boldsymbol{v}^\perp = 0$ holds for any $\boldsymbol{v} \in C$ and $\boldsymbol{v}^\perp \in C^\perp$. It implies that the dual code is the orthogonal complement of the primary code.

Given a codeword $\boldsymbol{w}$, the number of appearance of value 1 in $\boldsymbol{w}$ is called the *weight* of $\boldsymbol{w}$. The *weight distribution* of a (m, r)-linear code C is the m-dimensional vector whose k-th entry $W_C[k]$ is the number of codewords with weight k in C. The weight distribution is often represented as the form of generating functions $F_C(x) = \sum_{w=0}^{m} W_C[w]x^w$. This function is called the *weight-distribution polynomial* of C. There is a well-known theorem providing a relationship between the weight-distribution polynomials of primary and dual codes:

Theorem 1 (MacWilliams Identity [9]) *Let C be a (m, r)-linear code over $\mathbb{F}_2$ and $C^\perp$ be its dual. Then, the following identity holds:*

$$F_C(x) = \frac{1}{2^r}(1 + x)^m F_{C^\perp}\left(\frac{1-x}{1+x}\right).$$

By comparing the coefficient of each monomial in both sides, we have the representation of $W_C[k]$ by a linear sum of the weight distribution of $C^\perp$:

$$W_C[i] = \frac{1}{2^r} \sum_{j=0}^{m} K_m(j, i) W_{C^\perp}[j], \tag{1}$$

where $K_m(j, i)$ is the value known as Krawtchouk polynomials, defined as follows:

$$K_m(j, i) = \sum_{k=0}^{m} (-1)^k \binom{i}{k}\binom{m-i}{j-k}.$$

III. Counting Perfect Matchings via Cycle Space

A. Cut and Cycle Spaces

In this section any arithmetic operation for elements of vectors and matrices is over field $\mathbb{F}_2$. Letting $G = (V, E)$ be an undirected graph with n vertices $v_1, v_2, \cdots, v_n$ and m edges $e_1, e_2, \cdots, e_m$, its *incidence matrix* $A^G = (A^G_{i,j}) \in \mathbb{F}_2^{n \times m}$ is the one such that $A^G_{i,j} = 1$ if and only if v_i is incident to e_j and $A^G_{i,j} = 0$ otherwise. It is easy to check that the i-th row of A^G is the 0-1 vector representation of the set of edges incident to v_i. Given a 0-1 (row) vector representation of $\boldsymbol{v}_S$ for a vertex subset $S \subseteq V$, $\boldsymbol{v}_S A^G$ is the cutset between S and $V \setminus S$. It implies that the linear code defined by the generator matrix A^G is equivalent to the family of edge subsets each of which represents a cutset, so-called the *cut space* of G.

As an well-known fact, the set of all cycles in G induces a linear subspace of $\mathbb{F}_2^m$, where each element is a 0-1 vector representation of the edge set constituting one or more cycle(s). This subspace is called the *cycle space* of G. Note that the cycle space can be recognized as the set of all spanning even subgraphs (i.e., subgraphs where every vertex has an even degree). The matrix whose row is the basis of G's cycle space is denoted by B^G. Similarly to the cut space, we regard the cycle space as a linear code defined by the generator matrix B^G. An important relationship between cut space and cycle space, stated below, is known:

Fact 1 *The cycle space of G is the orthogonal complement of the cut space of G.*

This fact implies that the linear code associated with a cycle space is the dual code of that with the corresponding cut space, and vice versa. In the following argument, given an undirected graph G, $C(G)$ and $C^\perp(G)$ denote the linear codes defined by the generator matrices B^G and A^G respectively. We often use term "cutset of G" as the meaning of the codeword of $C(G)$ associated with that cutset. The same usage is also applied for cycle spaces.

B. From Cycle Space to Number of Perfect Matchings

Given an undirected graph $G = (V, E)$, we consider counting the number of perfect matchings of G. Since there is no perfect matching if the number of vertices is odd, we define $2n = |V|$. Let $m = |E|$ for short. The degree of vertex v is denoted by $d(v)$. First we focus on the case that G is an *odd* graph, i.e., a graph such that $d(v)$ is odd for any v in V. The number of perfect matchings of odd graph G is related to G's cycle space by the following lemma.

Lemma 1 *For any odd graph G, the number of perfect matchings in G is equal to $W_{C(G)}[m-n]$.*

Proof: Let $V = \{v_0, v_1, \cdots, v_{2n-1}\}$ be the set of vertices in G. We prove the lemma by defining a bijection between the set of codewords with weight $m-n$ and perfect matchings. More precisely, we prove that the complement (in terms of the edge set of G) of any codeword $\boldsymbol{w}$ in $C(G)$ with weight $m-n$ is a 1-factor (equivalent to a perfect matching). Let G_w be a spanning even subgraph corresponding to w. The degree of $v_i \in V$ in G_w is denoted by $d'(v_i)$. To prove that the complement of G_w is a 1-factor, it suffices to show that $d'(v_i) = d(v_i) - 1$ holds for any $v_i \in V$. Suppose for contradiction that $d'(v_i) \neq d(v_i) - 1$ holds for some $v_i \in V$. Since $d'(v_i) \leq d(v_i)$, $d(v_i)$ is odd, and $d'(v_i)$ is even (recall G_w is a spanning even subgraph of G), we have $d'(v_i) < d(v_i) - 1$. To make $\sum_{i=0}^{n-1} d'(v_i) = 2(m-n)$ hold, there must exist another vertex v_j satisfying $d'(v_j) > d(v_j) - 1 \Rightarrow d(v_j) = d'(v_j)$. It contradicts the fact that $d(v_j)$ is odd. ■

Combining the lemma above and Theorem 1, we obtain the following corollary:

Corollary 1 *Let G be an arbitrary odd graph. There exists an algorithm to count the number of perfect matchings in G with $O(m\tau(5m))$ time provided that the weight distribution $W_{C^{\perp}(G)}$ is available, where m is the number of edges in G and $\tau(x)$ be the time required for arithmetic operations of two x-bit integers.*

Note that the absolute value of Krawtchouk polynomials has a trivial upper bound $|K_m(j,i)| \leq \mathrm{poly}(m)\binom{m}{m/2}^2 \leq 2^{2m+O(\log m)}$, and the number of all codewords of $C^{\perp}(G)$ is at most $2^n \leq 2^m$. Thus, the time required for each arithmetic operation in the right term of formula 1 is bounded by $\tau(5m)$.

C. Transformation to Odd Bipartite Graph

While the result in the previous subsection assumes that G is an odd graph, that assumption can be easily removed. The fundamental idea is to construct the odd graph $\tilde{G}$ that has the same number of perfect matchings as G. While we only consider the case that G is a bipartite graph in this paper, general graph can be handled similarly. Let $G = (V_1 \cup V_2, E)$, be an arbitrary bipartite graph such that $|V_1| = |V_2| = n$, and $V = V_1 \cup V_2$ for short. The set of even-degree vertices in V_i is denoted by V_i^{even} ($i \in \{1, 2\}$). We can easily show the following lemma:

Lemma 2 *The values of $|V_1^{\mathrm{even}}|$ and $|V_2^{\mathrm{even}}|$ have the same parity.*

Proof: Assume that $|E|$ is odd. Since $\sum_{v \in V_i^{\mathrm{even}}} d(v)$ is even for any $i \in \{1, 2\}$, $\sum_{v \in V \setminus V_i^{\mathrm{even}}} d(v)$ must be odd. Thus, $|V \setminus V_i^{\mathrm{even}}|$ is odd for any $i \in \{1, 2\}$ because any node in $V \setminus V_i^{\mathrm{even}}$ has an odd degree. It implies that $|V_i^{\mathrm{even}}|$ is odd for any $\{1, 2\}$. The case of even $|E|$ can be proved similarly. ■

The construction of $\tilde{G}$ is given as follows:

- Add two vertices $\tilde{v}_{i,1}$, and $\tilde{v}_{i,2}$ to V_i for each $i \in \{1, 2\}$.
- For each $i \in \{1, 2\}$, connect each node in V_i^{even} with $\tilde{v}_{3-i,1}$, and $\tilde{v}_{3-i,1}$ with $\hat{v}_{i,2}$.
- If $d(\tilde{v}_{1,1})$ and $d(\tilde{v}_{2,1})$ are even, connect them. Recall that $d(\tilde{v}_{1,1})$ and $d(\tilde{v}_{2,1})$ have the same parity from Lemma 2.

An example of the construction is shown in Figure 1. For the constructed graph $\tilde{G}$, we have the following lemma.

Lemma 3 *The graph $\tilde{G}$ is an odd bipartite graph, and has the same number of perfect matchings as G.*

Proof: Any node in $\tilde{G}$ clearly has an odd degree. Let $M \subseteq \tilde{E}$ be any perfect matching of $\tilde{G}$. Since $\tilde{v}_{1,2}$ and $\tilde{v}_{2,2}$ is degree one, edges $\{\tilde{v}_{1,1}, \tilde{v}_{2,2}\}$ and $\{\tilde{v}_{2,1}, \tilde{v}_{1,2}\}$ are necessarily included in M. Then $M \setminus \{\{\tilde{v}_{1,1}, \tilde{v}_{2,2}\}, \{\tilde{v}_{2,1}, \tilde{v}_{1,2}\}\}$ is a perfect matching of G. Conversely, given a perfect matching $M' \subseteq E$ of G, $G \cup \{\{\tilde{v}_{1,1}, \tilde{v}_{2,2}\}, \{\tilde{v}_{2,1}, \tilde{v}_{1,2}\}\}$ is a perfect matching of $\tilde{G}$. Thus, we have a one-to-one correspondence between the perfect matchings of G and those of $\tilde{G}$. The lemma is proved. ■

IV. Computing Weight Distribution

As seen in the previous section, the computation of the cut weight distribution for graph $\tilde{G}$ induces the number of perfect matchings of G. Thus, in what follows, we focus on algorithms for computing the cut weight distribution.

The set of edges constituting a cut is associated with a partition of all vertices: A partition $(S, V \setminus S)$ of all vertices V induces a cutset, which is the set of edges crossing between S and $V \setminus S$. Thus we often use the sentence "partition $(S, V \setminus S)$ of V" as the meaning of the cut associated with that partition. We define $c(S, T)$ to be the set of edges crossing two disjoint subsets S and T ($S, T \subseteq V$). In particular, if S (resp. T) is a singleton $\{v\}$, we use notation $c(v, T)$ (resp. $c(S, v)$).

While two different partitions can lead the same cutset (e.g., $(S, V \setminus S)$ and $(V \setminus S, S)$), it is well-known that exactly

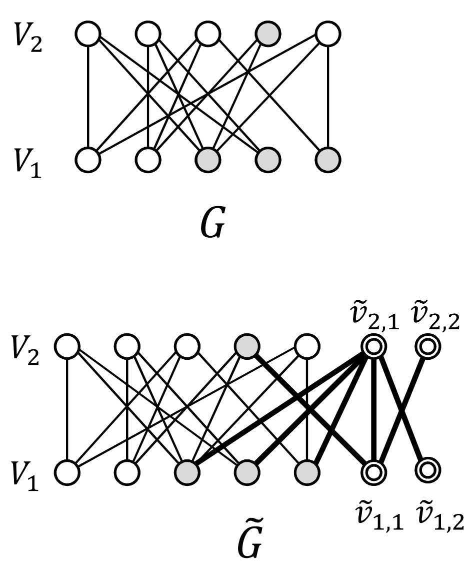


Figure 1. The construction of $\tilde{G}$

2^d subsets induce the same cutset, where d is the number of connected components of G and equal to $n - \mathrm{rank}(A^G)$. Thus, instead of computing $W_{C^\perp(G)}$, we rather consider the cut-weight distribution $W'_{C^\perp(G)}$ over all partitions, that is, $W'_{C^\perp(G)}[k] = |\{S \subseteq V | |c(S, V \setminus S)| = k\}|$. It is easy to calculate $W_{C^\perp(G)}$ from $W'_{C^\perp(G)}$ because of the relation of $W_{C^\perp(G)} = 2^{-d} \cdot W'_{C^\perp(G)}$.

A. $O^*(2^n)$-time Algorithm

A straightforward way of computing $W'_{C^\perp(G)}$ is to enumerate all partitions of V with computing their weights, which trivially takes $O^*(2^{2n})$ time. In the case of bipartite graphs, we can reduce the time required for computing the cut-weight distribution. As a first step, this subsection proposes an $O^*(2^n)$-time algorithm, which has the same performance as Ryser's one [1] (in terms of the base of the exponential part). Further improvement of the running time is considered in the following subsection.

Let $G = (V_1 \cup V_2, E)$ be the input bipartite graph such that $|V_1| = |V_2| = n$ and $|E| = m$, and $V = V_1 \cup V_2$ for short. For weight-distribution vector W and integer value $x \in [-m, m]$, we define $\sigma_x(W)$ as the vector obtained by shifting each element of W x times. That is,

$$\sigma_x(W)[i] = \begin{cases} 0 & \text{if } i < x, \\ W[i - x] & \text{if } n - 1 \geq i \geq x, \\ 0 & \text{if } i \geq n + x. \end{cases}$$

Note that the case of $i < x$ or $i \geq n + x$ applies only when x is positive or negative respectively. Let V' be a subset of V. We say that partition $(S, V \setminus S)$ is *conditioned* by a subset partition $(S', V' \setminus S')$ if $S \supseteq S'$ and $(V \setminus S) \supseteq (V' \setminus S')$ holds. Let $\mathcal{P}_{S'|V'}$ be the set of all partitions of V conditioned by $(S', V' \setminus S')$, and $W_{S'|V'}$ be the cut-weight distribution over all partitions in $\mathcal{P}_{S'|V'}$. Our algorithm relies on the fact that $W_{S|V_1}$ can be computed within polynomial time in n provided that a partition $(S, V_1 \setminus S)$ of V_1 is given. In the following argument, we introduce an arbitrary ordering $v_0, v_1, \cdots v_{n-1}$ of vertices in V_2. We define $V^i = \{v_i, v_{i+1}, \cdots v_{n-1}\} \cup V_1$. The lemma behind the correctness of our algorithm is stated below:

Algorithm 1 shift: Function for computing $W_{X|V^{n-1}}$

```
1: function shift($W, L$)     /* $W \in \mathbb{N}^m$ and , $L \in \mathbb{N}^*$ */
2:     while $L$ is not empty do
3:         $l \leftarrow$ the head of $L$
4:         Remove the head of $L$
5:         $W \leftarrow W + \sigma_l(W)$
6:     endwhile
7:     return $W$
```

Lemma 4 *For a given partition $(S, V_1 \setminus S)$, let $l = |c(v_i, V_1 \setminus S)| - |c(v_i, S)|$. Then $W_{S|V^{i+1}} = W_{S|V^i} + \sigma_l(W_{S|V^i})$ holds.*

Proof: From the definition of $W_{S|V^i}$, $W_{S|V^{i+1}} = W_{S \cup \{v_i\}|V^i} + W_{S|V^i}$ clearly holds. Thus it suffices to show $W_{S \cup \{v_i\}|V^i} = \sigma_l(W_{S|V^i})$. Let $(S', V \setminus S')$ be a partition in $\mathcal{P}_{S|V^i}$, and k be its weight. By adding v_i to S', the weight increases by l. That is, the weight of partition $(S' \cup \{v_i\}, V \setminus (S' \cup \{v_i\})$ is $k + l$. It implies a one-to-one correspondence between the partitions in $\mathcal{P}_{S|V^i}$ with weight k and those in $\mathcal{P}_{S \cup \{v_i\}|V^i}$ with weight $k + l$. Hence we have $W_{S \cup \{v_i\}|V^i}[k + l] = W_{S|V^i}[k]$ for any k. It clearly follows $W_{S \cup \{v_i\}|V^i} = \sigma_l(W_{S|V^i})$. The lemma is proved. ■

The recursive formula in Lemma 4 trivially allows us to compute $W_{S|V_1} = W_{S|V^{n-1}}$ within polynomial time in n. For the usefulness of the following argument, we encapsulate this recursion process by function shift shown in the pseudocode of Algorithm 1. Let $L : 2^{V_1} \rightarrow \mathbb{Z}^{|V_1|}$ be the function such that $L(X)[i] = |c(v_i, V_1 \setminus X)| - |c(v_i, X)|$ holds for any $v_i \in V_2$. Our $O^*(2^n)$-time algorithm computes and sums up the values of $\mathsf{shift}(W_{X|V^0}, L(X))$ over all partitions of V_1. That is, our algorithm computes the right side of the following equality:

$$W'_{C^\perp(G)} = \sum_{S \subseteq V_1} \mathsf{shift}(W_{S|V^0}, L(S)). \qquad (2)$$

The correctness of this formula is obvious from the definition of $W_{S|V_1}$.

Theorem 2 *There is an algorithm computing $W'_{C^\perp(G)}$ with $O^*(2^n)$ time.*

B. *Function* shift *as a Linear Transformation*

Before introducing the faster algorithm, we show several properties of Function shift. Let $H = \{h_{i,j}\} \in \mathbb{R}^{m\times m}$ be the matrix defined as $h_{i,j} = 1$ if $j = i+1$ and 0 otherwise. It is easy to check this matrix works as the operator σ_1, i.e., for any m-dimensional vector W, $WH^x = \sigma_x(W)$ holds. Hence we can describe function $\mathsf{shift}(W, L)$ for a given sequence $L = (l_0, l_1, \cdots l_{n-1})$ as follows:

$$\mathsf{shift}(W, L) = W \left(\prod_{i=0}^{n-1} (H^{l_i} + I) \right), \tag{3}$$

where I be the $m \times m$ identity matrix. We can obtain the following lemma:

Lemma 5 *Letting L and L' be two sequences of integers, and $W_1, W_2 \in \mathbb{N}^m$. Then the following properties hold:*

1) $\sigma_x(\mathsf{shift}(W, L)) = \mathsf{shift}(\sigma_x(W), L)$,
2) $\mathsf{shift}(\mathsf{shift}(W, L), L') = \mathsf{shift}(W, L \circ L')$,
3) $\sigma_x(W_1 + W_2) = \sigma_x(W_1) + \sigma_x(W_2)$, *and* $\mathsf{shift}(W_1 + W_2, L) = \mathsf{shift}(W_1, L) + \mathsf{shift}(W_2, L)$,

where $\circ$ is the concatenation of two sequences.

Proof: Since $\sigma_x(W) = \mathsf{shift}(W, (x))$, we can treat σ_x equivalently to shift. Clearly, Equation 3 implies that $\mathsf{shift}(*, L)$ is a commutative linear transformation. Thus all properties obviously hold. ■

C. *Improving Running Time*

In this subsection, we consider an improvement of $O^*(2^n)$-time algorithm. The running time of the improved algorithm is $O^*(2^{(1-\frac{1}{5\Delta \log \Delta})n})$ and consumes exponential space, where Δ is the average degree of the input graph.

The underlying principle of the improved algorithm is very simple: Separating two smaller subproblems. Let (T_1, U_1) be a partition of V_1 (i.e., $T_1 = V_1 \setminus U_1$) fixed by the algorithm, $N(U_1) \subseteq V_2$ be the set of vertices adjacent to U_1, and $v_0, v_1, \cdots v_{n-1}$ be an arbitrary ordering of V_2 such that the last $|N(U_1)|$ vertices correspond to $N(U_1)$. The cardinality of $N(U_1)$ is denoted by h for short. Now we consider the situation where U_1 and T_1 are partitioned into $(X, U_1 \setminus X)$ and $(Y, T_1 \setminus Y)$. If we regards X and Y as variables, the first $n-h$ entries $(l_0, l_1, \cdots l_{n-h})$ of $L(X \cup Y)$ become a function of X, which are independent of the value of Y. In contrast, the last h entries $(l_{n-h}, l_{n-h+1}, \cdots l_{n-1})$ are a function of both X and Y. Consequently, by two appropriate functions $L_T : 2^{|T_1|} \to \mathbb{Z}^{n-h}$ and $L_U : 2^{T_1} \times 2^{U_1} \to \mathbb{Z}^h$, the sequence $L(X \cup Y)$ can be described as follows:

$$L(X, Y) = L_T(X) \circ L_U(X, Y).$$

Then the following lemma holds:

Lemma 6

$$L_U(X, Y) = L_U(X, \emptyset) + L_U(\emptyset, Y) - L_U(\emptyset, \emptyset).$$

Proof: We prove $L_U(X, Y)[i] = L_U(X, \emptyset)[i] + L_U(\emptyset, Y)[i] - L(\emptyset, \emptyset)[i]$ for any i. Since $X \subseteq T_1$ and $Y \subseteq U_1$ are mutually disjoint, the sets of edges $c(v_i, X)$ and $c(v_i, Y)$ are mutually disjoint. Thus we have $|c(v_i, X \cup Y)| = |c(v_i, X)| + |c(v_i, Y)|$. Similarly, we have $|c(v_i, V_1 \setminus (X \cup Y))| = |c(v_i, (T_1 \setminus X) \cup (U_1 \setminus Y))| = |c(v_i, (T_1 \setminus X))| + |c(v_i, (U_1 \setminus Y))|$. Then we can obtain the following equality:

$$\begin{aligned}
&L_U(X, Y)[i] \\
&= |c(v_i, V_1 \setminus X \cup Y)| - |c(v_i, (X \cup Y))| \\
&= |c(v_i, (T_1 \setminus X))| - |c(v_i, X)| \\
&\quad + |c(v_i, (U_1 \setminus Y))| - |c(v_i, Y)| \\
&= |c(v_i, (V_1 \setminus X))| - |c(v_i, T_1)| - |c(v_i, X)| \\
&\quad + |c(v_i, (V_1 \setminus Y))| - |c(v_i, U_1)| - |c(v_i, Y)| \\
&= L_U(X, \emptyset)[i] + L_U(\emptyset, Y)[i] - |c(v_i, T_1)| - |c(v_i, U_1)| \\
&= L_U(X, \emptyset)[i] + L_U(\emptyset, Y)[i] - |c(v_i, T_1 \cup U_1)| \\
&= L_U(X, \emptyset)[i] + L_U(\emptyset, Y)[i] - L_U(\emptyset, \emptyset)[i].
\end{aligned}$$

The lemma is proved. ■

The improved algorithm runs as follows:

- (Step 1) We divide all partitions of T_1 into several classes $\mathcal{C}_0, \mathcal{C}_1, \cdots \mathcal{C}_x$ such that for any two partitions $(X_1, T_1 \setminus X_1)$ and $(X_2, T_1 \setminus X_2)$ in the same class, $L_U(X_1, \emptyset) = L(X_2, \emptyset)$ holds.
- (Step 2) For each $i \in [1, x]$, we compute weight distribution $W_i = \sum_{(X, T_1 \setminus X) \in \mathcal{C}_i} W_{X|V^{n-h-1}}$.
(Note that $W_i = \sum_{(X, T_1 \setminus X) \in \mathcal{C}_i} \mathsf{shift}(W_{X|V^0}, L_T(X))$ holds.)
- (Step 3) Let $L(i)$ be the value of $L_U(X, \emptyset)$ associated with class $\mathcal{C}_i$ and $c_Y = |c(Y, V_2)|$ for short. For each $i \in [0, x]$ and each partition $(Y, U_1 \setminus Y)$ of U_1, we compute $L_U(i, Y) = L(i) + L_U(\emptyset, Y) - L_U(\emptyset, \emptyset)$ and $\mathsf{shift}(\sigma_{c_Y}(W_i), L_U(i, Y))$. The sum of all the values returned by function shift is the output of the algorithm.

We can show the following lemma, which directly leads the correctness of the algorithm:

Lemma 7

$$W'_{C^\perp(G)} = \sum_{i=1}^{x} \sum_{Y \subseteq U_1} \mathsf{shift}(\sigma_{c_Y}(W_i), L_U(i, Y)).$$

Proof: Since $W_{X|V^0}$ is the distribution over singleton $\{(X, V^0 \setminus X)\}$, we have $W_{X|V^0}[i] = 1$ for $i = |c(X, V^0 \setminus X)|$ and 0 otherwise. Thus, we have $\sigma_{c_Y}(W_{X|V^0})[i] = 1$ for $i = |c(X, V^0 \setminus X)| + c_Y$ and 0 otherwise. Since $|c(X, V^0 \setminus X)| + c_Y = |c(X \cup Y, V \setminus (X \cup Y))|$ holds, we obtain

$$\sigma_{c(Y)}(W_{X|V^0}) = W_{X \cup Y|V^0}. \tag{4}$$

By using this equation, Lemma 5 and 7, we can obtain the following equality:

$$
\begin{aligned}
&\sum_{i=1}^{x}\sum_{Y\subseteq U_1}\mathsf{shift}(\sigma_{c_Y}(W_i),L_U(i,Y))\\
&=\sum_{\substack{1\le i\le x\\ Y\subseteq U_1}}\mathsf{shift}\Big(\sigma_{c_Y}\Big(\sum_{(X,T_1\setminus X)\in\mathcal{C}_i}W_{X|V^{n-h-1}}\Big),L_U(i,Y)\Big)\\
&=\sum_{\substack{1\le i\le x\\ Y\subseteq U_1}}\sum_{(X,T_1\setminus X)\in\mathcal{C}_i}\mathsf{shift}\left(\sigma_{c_Y}(W_{X|V^{n-h-1}}),L_U(X,Y)\right)\\
&=\sum_{\substack{X\subseteq T_1\\ Y\subseteq U_1}}\mathsf{shift}\left(\sigma_{c_Y}(\mathsf{shift}(W_{X|V^0},L_T(X))),L_U(X,Y)\right)\\
&=\sum_{\substack{X\subseteq T_1\\ Y\subseteq U_1}}\mathsf{shift}\left(\mathsf{shift}(\sigma_{c_Y}(W_{X|V^0}),L_T(X)),L_U(X,Y)\right)\\
&=\sum_{\substack{X\subseteq T_1\\ Y\subseteq U_1}}\mathsf{shift}\left(\sigma_{c_Y}(W_{X|V^0}),L_T(X)\circ L_U(X,Y)\right)\\
&=\sum_{\substack{X\subseteq T_1\\ Y\subseteq U_1}}\mathsf{shift}(W_{X\cup Y|V^0},L(X\cup Y))\\
&=W'_{C^{\perp}(G)}.
\end{aligned}
$$

The lemma is proved. ∎

We focus on the running time of the algorithm. Clearly the first and second steps of the algorithm take $O^*(2^{n-|U_1|})$ time respectively. The third step requires time of $O^*(x2^{|U_1|})$. Thus the total running time is $O^*(2^{n-|U_1|}+x2^{|U_1|})$.

How small can we bound x? Clearly, it is upper bounded by the size of the domain of $L_U(X)$. From the definition, the value of $L_U(X)[i-(n-h)]$ can take $d(v_i)+1$ different values for any $v_i\in N(U_1)$. It follows $x\le\prod_{v_i\in N(U_1)}(d(v_i)+1)$. By applying the arithmetic mean-geometric mean inequality, we can further bound x by $((\sum_{v_i\in N(U_1)}(d(v_i)+1))/|N(U_1)|)^{|N(U_1)|}$. Letting Δ_X be the average degree over $X\subseteq V$ in G, we have

$$x\le(\Delta_{N(U_1)}+1)^{|N(U_1)|}. \tag{5}$$

We consider how to choose U_1. Letting Δ be the average degree of G, V_1 contains a subset X of $n/5$ vertices whose degrees are at most $5\Delta/4$. We choose $n/(5\Delta\log\Delta)$ vertices from X as U_1. For that choice we have $|N(U_1)|\le n/(4\log\Delta)$. Since $|N(U_1)|\Delta_{N(U_1)}\le\Delta n$ holds, we obtain $\Delta_{N(U_1)}\le 4\Delta\log\Delta$. By assigning this bound to Inequality 5, we obtain

$$x\le(4\Delta\log\Delta+1)^{\frac{n}{4\log\Delta}}\le(4\Delta^2)^{\frac{n}{4\log\Delta}}=O(2^{\frac{5n}{6}})$$

Consequently, it follows that the running time of our algorithm is $O^*(2^{(1-\frac{1}{5\Delta\log\Delta})n})$.

Theorem 3 *There is an an algorithm for counting perfect matchings of bipartite graphs which runs with $O^*(2^{(1-\frac{1}{5\Delta\log\Delta})n})$ time and exponential space.*

V. Related Work

As seen in the introduction, we have roughly three lines about the studies on counting perfect matchings. We introduce the related work along them respectively.

There has been proposed two different approach for approximating the number of perfect matchings. The first one is the Markov-chain Monte Carlo method, which gives a fully-polynomial randomized approximation scheme (FPRAS) for counting perfect matchings [10–12]. The second one is a randomized averaging of the determinant [13–15]. The fastest approximation algorithm on this approach is one by Chien et.al. [15], which runs with $O(1.2^n)$ time. It is still an open problem whether there exists a FPRAS following this approach or not.

The second line is the algorithm design for restricted inputs. A seminal work on this line is a polynomial-time exact counting algorithm for planar graphs [16]. As other restrictions, graphs of bounded genus [17, 18] or bounded treewidth [4, 5], and chordal graphs with its subclass [19] are considered.

About the line of exact algorithms, we have already mentioned the results for bipartite graphs in the introduction. Thus we introduce only the work on counting perfect matchings for general graphs. A first result breaking the trivial $O^*(2^m)$-time bound is one by Björklund and Husfeldt [20], which has shown two algorithms: The first one runs with $O^*(2^{2n})$ time and polynomial space, and the second rounds with $O^*(1.733^{2n})$ time and exponential space. These algorithms are similar with our result in the sense that it also reduces the problem into a counting over a different universe. A number of the following studies improve this bound [21–25]. The most recent and fastest one is the algorithm by Björklund [25], which achieves the same running time as Ryser's algorithm (that is, currently we do not find the difference of inherent difficulty between bipartite and general graphs). About time complexity, Dell et.al. [26] has shown that any algorithm has an instance of m edges incurring $\Omega(exp(m/\log m))$ time if we believe that a counting version of the Exponential Time Hypothesis [27] is true.

VI. Concluding Remarks

In this paper, we presented a new algorithm for the problem of counting perfect matchings, which has an improved time bound depending on the average degree Δ of the input graph. Compared to previous results, our algorithm runs faster for many cases. In particular, the performance degradation to the increase of Δ is quite slower than the previous algorithms. The main idea of our algorithm is a new reduction to computing the cut-weight distribution of the input graph. Our algorithm is designed by combining this reduction with a novel algorithm for the computation of cut-weight distribution. The approach itself is quite new, and may be of independent interest. Finally, we conclude the paper with several open problems related to our approach.

- Can we achieve the running time exponentially faster than Ryser's one by designing a faster algorithm computing cut-weight distribution?
- The reduction part of our result is directly applicable to any graph (which may not be bipartite). Can we use the reduction to obtain a faster algorithm for general graphs? Actually, letting $I(G)$ be the independent sets of the input graph G, we can easily obtain an algorithm with $O^*(2^{2n-|I(G)|})$ running time by regarding G as a "quasi" bipartite graph of two vertex sets $I(G)$ and $V \setminus I(G)$ and applying our $O^*(2^n)$-time algorithm, which gives the same performance as the algorithm by [23].
- Is it possible to design a faster FPRAS for counting perfect matchings based on our method? Note that an $(1+\epsilon)$-approximation of the cut-weight distribution trivially induces an $(1+\epsilon)$-approximation of the number of perfect matchings because of the linearity of the transformation.
- Computing cut-weight distribution is a special case of the counting version of 2-CSP, which is addressed by Williams [28]. In this sense, our reduction gives a new linkage from counting perfect matchings to CSP. Can we use this linkage for obtaining some new complexity result around those problems?
- Can we apply the same technique to other combinatorial problems? Interestingly, there has been proposed a variety of MacWilliams-style Identities in the field of the coding theory. We may find a useful transformation from those resources. In addition, it may be an interesting approach to focus on the primal-dual relationship of two universes. Can we design a kind of primal-dual algorithms for counting problems?

References

[1] H. Ryser, *Combinatorial Mathematics*, ser. The Carus mathematical monographs. Mathematical Association of America, 1963.

[2] E. T. Bax and J. Franklin, "A permanent algorithm with $exp[\omega(n^{1/3}/2\ln(n))]$ expected speedup for 0-1 matrices." *Algorithmica*, vol. 32, no. 1, pp. 157–162, 2002.

[3] R. A. Servedio and A. Wan, "Computing sparse permanents faster," *Information Processing. Letters*, vol. 96, pp. 89–92, November 2005.

[4] S. Arnborg, J. Lagergren, and D. Seese, "Easy problems for tree-decomposable graphs," *J. Algorithms*, vol. 12, pp. 308–340, 1991.

[5] J. M. M. Rooij, H. L. Bodlaender, and P. Rossmanith, "Dynamic programming on tree decompositions using generalised fast subset convolution," in *Proc. of 17th Annual European Symposium on Algorithms (ESA)*, 2009, pp. 566–577.

[6] F. V. Fomin and K. Høie, "Pathwidth of cubic graphs and exact algorithms," *Information Processing Letters*, vol. 97, pp. 191–196, March 2006.

[7] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto, "Fourier meets möbius: fast subset convolution," in *Proc. of the 39th annual ACM symposium on Theory of computing (STOC)*, 2007, pp. 67–74.

[8] D. Lokshtanov and J. Nederlof, "Saving space by algebraization," in *Proc. of the 42th annual ACM symposium on Theory of computing (STOC)*, 2010, pp. 321–330.

[9] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Elsevier, 1977.

[10] M. Jerrum and A. Sinclair, "Approximating the permanent," *SIAM Journal on Computing*, vol. 18, pp. 1149–1178, 1989.

[11] M. Jerrum, A. Sinclair, and E. Vigoda, "A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries," *Journal of the ACM*, vol. 51, pp. 671–697, 2004.

[12] I. Bezáková, D. Štefankovič, V. V. Vazirani, and E. Vigoda, "Accelerating simulated annealing for the permanent and combinatorial counting problems," in *Proc. of the 17th annual ACM-SIAM symposium on Discrete algorithm (SODA)*, 2006, pp. 900–907.

[13] C. D. Godsil and I. Gutman, "On the matching polynomial of a graph," in *Algebraic Methods in Graph Theory*, 1981, pp. 241–249.

[14] N. Karmarkar, R. Karp, R. Lipton, L. Lovász, and M. Luby, "A monte-carlo algorithm for estimating the permanent," *SIAM Journal on Computing*, vol. 22, pp. 284–293, 1993.

[15] S. Chien, L. Rasmussen, and A. Sinclair, "Clifford algebras and approximating the permanent," in *Proc. of the 34th annual ACM symposium on Theory of computing (STOC)*, 2002, pp. 222–231.

[16] P. Kasteleyn, "Graph theory and crystal physics," in *Graph Theory and Theoretical Physics*. Academic Press, 1967, pp. 43–110.

[17] A. Galluccio and M. Loebl, "On the theory of pfaffian orientations. i. perfect matchings and permanents," *Electronic Journal of Combinatorics*, vol. 6, 1999.

[18] G. Tesler, "Matchings in graphs on non-orientable surfaces," *Journal of Combinatrial Theory Series B*, vol. 78, pp. 198–231, 2000.

[19] Y. Okamoto, R. Uehara, and T. Uno, "Counting the number of matchings in chordal and chordal bipartite graph classes," in *Proc. of 35th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, 2009, pp. 296–307.

[20] A. Björklund and T. Husfeldt, "Exact algorithms for exact satisfiability and number of perfect matchings," *Algorithmica*, vol. 52, pp. 226–249, 2008.

[21] M. Koivisto, “Partitioning into sets of bounded cardinality,” in *Proc. of 4th International Workshop on Parameterized and Exact Computation (IWPEC)*, vol. LNCS 5917, 2009, pp. 258–263.

[22] J. Nederlof, “Inclusion excluion for hard problems,” 2008.

[23] O. Amini, F. V. Fomin, and S. Saurabh, “Counting subgraphs via homomorphisms,” in *Proc. of the 36th international colloquium conference on Automata, languages and programming (ICALP)*, 2009, pp. 71–82.

[24] J. Nederlof, “Fast polynomial-space algorithms using mobius inversion: Improving on steiner tree and related problems,” 2010, available at http://www.uib.no/People/jne061/Steinerfull.pdf.

[25] A. Björklund, “Counting perfect matchings as fast as ryser,” in *Proc of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012, pp. 914–921.

[26] H. Dell, T. Husfeldt, and M. Wahlén, “Exponential time complexity of the permanent and the tutte polynomial,” in *Proc. of the 37th international colloquium conference on Automata, languages and programming (ICALP)*, vol. LNCS 6198, 2010, pp. 426–437.

[27] R. Impagliazzo, R. Paturi, and F. Zane, “Which problems have strongly exponential complexity?” *Journal of Computer and System Sciences*, vol. 63, no. 4, pp. 512–530, 2001.

[28] R. Williams, “A new algorithm for optimal 2-constraint satisfaction and its implications,” *Theoretical Computer Science*, vol. 348, no. 2, pp. 357–365, dec 2005.

2 次元軌跡データの比較における 文字列アルゴリズムの応用について

鈴木大輔 †　泉泰介 †　和田幸一 ††
† 名古屋工業大学大学院　工学研究科　情報工学専攻
〒466-8555　愛知県名古屋市昭和区御器所町
†† 法政大学大学院　工学研究科　情報電子工学専攻
〒184-8584　東京都小金井市梶野町 3-7-2

1　はじめに

近年，GPS 端末の普及により端末の移動軌跡のデータ収集が可能になってきた．収集された軌跡データを分析することによって様々な用途に応用することが期待されている [4]．また，軌跡を比較するための類似度関数が数多く研究されている．例えば EDR や LCSS,DTW などが挙げられる [3]．通常，軌跡データは (２次元の) 実数値座標の系列の集合として表現される．本研究では，この実数座標系列を量子化により有限種のアルファベットの文字列と見なすことで，文字列処理アルゴリズムの軌跡データへの適用を検討する．文字列処理のためのアルゴリズムには長い歴史があり，そのリソースを軌跡データ処理に応用できることは有用であると考えられる．本稿では，直交座標系を用いた量子化と，極座標系を用いた量子化の２種類を提案し，それらの手法による軌跡の近似が，軌跡データの評価に対してどのような影響が出るかを実験により評価する．今回評価の対象として，軌跡データの類似度尺度として良く知られた EDR と，量子化後の文字列の編集距離の比較を行う．そして，EDR と文字列の編集距離との相関を計算機実験により評価し，その存在を実データにおいて確認した．

2　軌跡データの量子化

今回二つの方法で軌跡データを量子化し，文字列に対応付けた．一つ目は直交座標系を用いた量子化で，二つ目は極座標系を用いた量子化である．また，軌跡データは実数値 (x, y) の点列であると定義する．また，軌跡の点の数をその軌跡の長さとする．

2.1　直交座標系を用いた量子化

まず，全軌跡データの x 座標の最大値を x_{max}，最小値を x_{min} とし，y 座標の最大値を y_{max}，最小値を y_{min} とする．そして，(x_{min}, y_{min}) を原点とし，(x_{max}, y_{max}) が最大となる矩形を作る．この矩形を直線で等分し量子化する．等分された各矩形は近似的な座標点に相当し，それぞれの１区画がアルファベットの１文字に相当する．図１は各区画に英文字２字のアルファベットを割り当てることで量子化した例である．

図１の例で量子化によって軌跡データを文字列に変換すると，黒い軌跡 S と赤い軌跡 T はそれぞれ，dacbccbdbebfbz, gabedbccddeeefgz となる．

2.2　極座標系を用いた量子化

まず，軌跡データ中のある軌跡 S を考える．軌跡 S の i 番目の点を (Sx_i, Sy_i) とする．S の長さは m とし，(Sx_i, Sy_i) と (Sx_{i+1}, Sy_{i+1}) を考える．ここで，極座標の概念を取り入れ，始点 (Sx_i, Sy_i) から (Sx_{i+1}, Sy_{i+1}) までの長さを r，偏角を θ と

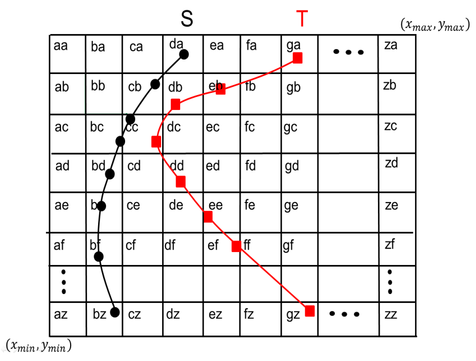


図 1　直交座標系を用いた量子化

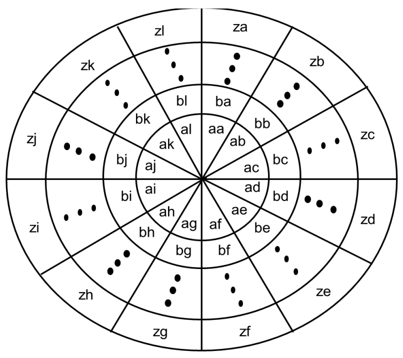

za zb zc zd ze zf zg zh zi zj zk zl
aa ab ac ad ae af ag ah ai aj ak al
ba bb bc bd be bf bg bh bi bj bk bl

図 2　極座標系を用いた量子化

する．また，全軌跡データの r の最大値を r_{max} とする．そして，半径 r_{max} の円を作り，この半径を今回は 26 等分し，中心から順に a,b,c,...,z を割り当てる．また，中心角を 30 °ずつに区切り，図 2 のように順に a,b,c,...,l を割り当てる．具体例を示す．$r_{max} = 1$ とし，$(r, \theta) = (\frac{12}{26}, 20°), (\frac{6}{26}, 45°), (\frac{15}{26}, 175°), (\frac{2}{26}, 260°)$ のように軌跡データの極座標表現が与えられているとする．この軌跡データを量子化によって文字列に変換すると，mcgbpjcg となる．

3　実験及び考察

量子化による軌跡の近似により，軌跡データの評価に対してどのような影響が出るかを実験で評価する．評価の対象として，編集距離と EDR の比較を行う．

3.1　編集距離

編集距離 (EditDistance) とは，二つの文字列がどの程度異なっているかを示す数値である．具体的には二つの文字列 x と y が与えられたとき，x を y へ変換するために必要な挿入・削除・置換命令の最小数である．以下に具体的な例を示す．
次のような文字列 x,y が与えられたとする．

$$x = \text{kitten}$$

$$y = \text{sitting}$$

編集命令 (挿入・削除・置換) を使って x を y に変換することを考える．

ステップ 1　kitten → sitten(k を s に置換)
ステップ 2　sitten → sittin(e を i に置換)
ステップ 3　sittin → sitting(g を挿入)

このように 3 回の命令で x を y に変換することができたので x と y の編集距離は 3 となる．形式的には，アルファベットの集合を Σ としたとき，2 つの文字列 $x, y \in \Sigma^*$ に対して，以下のように再帰的に定義される．

定義 3.1　それぞれ長さ m, n の文字列 x と y の編集距離を $ED(x, y)$ とする．

$$ED(x,y) = \begin{cases} m & if \;\; n = 0 \\ n & if \;\; m = 0 \\ min\{ED(Rest(x), Rest(y)) + 1, \\ \quad ED(Rest(x), y) + 1, \\ \quad ED(x, Rest(y)) + 1\} \\ \quad\quad otherwise \end{cases}$$

$(Rest(x)$ は x の先頭の文字を除いた文字$)$

3.2 EDR

EDR(Edit Distance on RealSequences) とは軌跡の比較のための類似度関数である．一般に軌跡データはユークリッド空間中の点列として表される．軌跡の類似度である EDR は，この点列に対するある種の編集距離として定義される．まず，2 点間の match というものを定義する．

定義 3.2 2 つの軌跡 R, S のそれぞれの軌跡要素ベクトル r_i と s_j が与えられ, $match(r_i, s_j) = true$

$$\iff |r_{i,x} - s_{j,x}| \le \epsilon \; and \; |r_{i,y} - s_{j,y}| \le \epsilon$$

を満たすとき，この組は match するという．ここで ϵ はマッチング閾値である．

また，それぞれ長さ n と m の軌跡 R,S が与えられたとき、R,S 間の EDR は、R を S に変換するときに必要な挿入、消去、置換の命令の数である．具体的な定義は以下のようになる．

定義 3.3

$$EDR(R,S) = \begin{cases} m & if \;\; n = 0 \\ n & if \;\; m = 0 \\ min\{EDR(Rest(R), Rest(S)) \\ \quad +cost, \; EDR(Rest(R), S) + 1, \\ \quad EDR(R, Rest(S)) + 1\} \\ \qquad otherwise \end{cases}$$

$cost = 0 \quad if \quad match(r_1, s_1) = true$

$cost = 1 \quad otherwise$

3.3 実験

次のような環境で実験を行った．

CPU AMD Athlon(tm)64 X2 Dual Core Processor 2.7GHz

メモリ 2.00GB

OS Windows7 Professional

今回，軌跡データを量子化し文字列に対応付けたが，この文字列の編集距離による比較と，軌跡の EDR による比較が相関があるかどうかを調べた．使用したデータは，University of California, Irvine Knowledge Discovery in Databases Archive の Australian Sign Language である．これは位置情報を記録できる特殊なグローブを用いて，オーストラリア人の手話の軌跡をデータにしたものである．元データは 3 次元の軌跡であるが，今回はその xy 平面への射影を軌跡データとして扱う．軌跡データのデータ数は 6274 であり，軌跡長の平均は 57.11, 最大，最小は 485,6 である．

実験は，無作為に選んだ 1 軌跡に対し，その他の軌跡との EDR および量子化後の編集距離の計算を行った．

3.4 結果・考察

実験の結果を図 3 に示す．縦軸は編集距離または EDR である．各軌跡に対して計算した評価値を EDR 値の昇順で並べて折れ線で記述している．

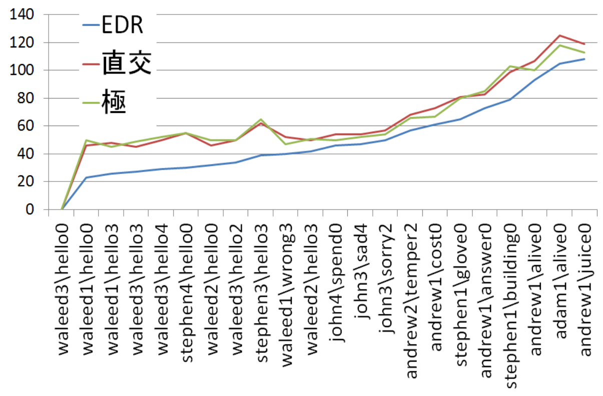


図 3 EDR と編集距離の相関

図 3 を見ると，EDR と編集距離の間に確かな相関があることが分かる．よって，量子化後に計算した編集距離は，EDR のよい近似となっていると言える

4 まとめと今後

今回，軌跡データを二つの方法 (直交座標系と極座標系) で量子化し，文字列に変換した．この文字列を編集距離を使って比較することは軌跡の類似度測定において有益であることを EDR との比較実験により示した．長さ n_1, n_2 の 2 軌跡間の EDR，量子化後の編集距離の計算時間はいずれも $O(n_1 n_2)$

であり．現時点の評価においては量子化を利用するメリットはないように思われるが，文字列間編集距離の計算は，高速な近似計算が可能であることが知られている [1, 2]．そのような手法を組み合わせることで，量子化を用いた手法の計算量的な有効性を示していくことが今度の課題である．

参考文献

[1] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. In *STOC*, pp. 199–204, 2009.

[2] Ziv Bar-yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *Proc. FOCS 2004*, pp. 550–559. The Institute of Electrical and Electronics Engineers, 2004.

[3] Lei Chen and M. Tamer Ozsu. Robust and fast similarity search for moving object trajectories. In *International Conf. on Data Engineering*, pp. 491–502, 2005.

[4] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, Yu Zheng, and Xing Xie. Searching trajectories by locations: an efficiency study. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pp. 255–266, New York, NY, USA, 2010. ACM.

[5] Vladimir.I.Levenshtein. binary codes capable of correcting deletions,insertions,and reversals. pp. 707–710. Cybernetics and Control Theory, 1966.

軌跡データベースにおける k 点連結最良軌跡問い合わせのキャッシュを利用した高速化

栗林大輔 †　泉泰介 †　和田幸一 ††
† 名古屋工業大学大学院　工学研究科　情報工学専攻
〒466-8555　愛知県名古屋市昭和区御器所町
†† 法政大学大学院　工学研究科　情報電子工学専攻
〒184-8584　東京都小金井市梶野町 3-7-2

1　はじめに

1.1　背景

近年，携帯電話に代表されるモバイル機器の多くには GPS が搭載されるようになり，位置情報を利用したサービスの実現が容易となっている．その一方で，サービス提供者にとっても，利用者の位置情報に基づく行動履歴の収集が容易となり，そのデータをマーケティング等に利用することで業務のさらなる最適化を図ることが期待できる[*1]．各個人の移動履歴は，地理的空間上を移動する点の系列からなる軌跡データとして記録される．携帯端末の利用台数は現在日本国内に限定しても 1 億台を超えており，それら多数の利用者の移動軌跡からなるデータセット（軌跡データベース）のサイズは膨大なものとなる．そのため，軌跡データベースに対する高度な問い合わせを，高いスケーラビリティを保ちつつ処理することが実用上重要である．

本研究では，軌跡データベースへの問い合わせ処理の一種である，k 点連結最良軌跡問い合わせ (k-Best-Connected Trajectory Search: k-BCT) に注目する．この問い合わせは，問い合わせとして与えられた複数の点を繋ぐような軌跡として最も適当なものを k 本選択し，答えとして返すという処理である．k-BCT の応用として，例えば地図上の観光地を複数の指定して問い合わせることで，それに対応したモデルルートを検索するといった処理などが挙げられる．

k-BCT 問い合わせは Zhou らによる研究において初めて提唱された問い合わせのタイプである．同論文では，候補軌跡集合の枝刈りを行いながら軌跡データベースの効率的な探索を行うアルゴリズムが提案されている．具体的には，問い合わせとして与えられた各点に対してその近くを通る軌跡を近傍探索 (Nearest Neighbor search: NN) により抽出し，それらの和集合を回答の候補とすることで，軌跡データベースの探索範囲を絞り込んでいる．

本研究では，より高速な k-BCT 問い合わせ処理の実現を目的として，Zhou らのアルゴリズムの改良を目指す．特に，Zhou らのアルゴリズムの実行時間において支配的な処理である，各問い合わせ点に対する最近傍点探索 (NN 探索) の高速化を試みる．高速化の基本となるアイデアは，いくつかの点に対して，問い合わせ結果の事前キャッシュを作成しておき，キャッシュされている点，あるいはその近傍が問い合わせクエリに含まれていた場合は，キャッシュ内容を NN 探索の結果として利用するということである．一般論として，データベース問い合わせ処理においては，非常に限定されたパタンのクエリがきわめて高い頻度で実行される傾向があることが経験的に知られている．そのため，k-BCT

[*1] 実際には利用者のプライバシ侵害の問題から，完全なデータを収集することは必ずしも容易ではないが，本研究ではこの点については注目しない．

問いあわせにおいても，少数の限られた点（例えば前述の観光ルート推薦の例では，人気のある観光地に対応するような点）のみが高い頻度で問い合わせ点としてクエリに含まれることが想定される．本研究のアプローチは，そのような点の NN 探索結果を事前に記録しておくことで，平均的な問い合わせ速度の向上を，比較的少ない点をキャッシュするだけで達成することができるという期待に基づいている．

本論文では，上記の提案手法と既存の手法の実験比較により，提案手法の有効性を検証している．比較実験の結果，キャッシュとして用意した NN 探索結果のサイズが入力されたクエリに適しているようなケースにおいて大幅な速度向上を達成できていることが確認できた．

1.2 構成

本論文の構成を以下に示す．第 2 章では k-BCT 問い合わせの定義を述べる．第 3 章では既存の研究として k-NN 探索と IKNN アルゴリズムについて述べる．第 4 章では提案手法について述べる．第 5 章では実験結果や考察を述べる．第 6 章でまとめと今後の課題を述べる．

2 諸定義

n 本の 2 次元空間中の軌跡からなるデータ集合 $\mathcal{D} = R_1, R_2, \cdots R_n$ を考える．ここで，データ集合中の各軌跡 R_i は 2 次元空間中の点の列 $R_i = (p_1, p_2, \cdots)$ として，表現されるものとする．なお，R_i は順序列であるが，特に断りなく R_i を軌跡を構成する点の集合として扱う場合がある．軌跡 R_i を構成する点の個数をその軌跡の長さと呼び，l_i で表す．

本論文は，軌跡データ集合に対する k-連結最良軌跡 (k-BCT) 問い合わせを取り扱う．同問い合わせは，クエリ点として与えられる m 個の点に対して，それらを最もよく連結するような k 本の軌跡をデータ集合から選択して返す．形式的には，　以下のように点 q_i を用いてクエリ Q を定義する．

$$Q = \{q_1, q_2, \cdots, q_m\}$$

以降，本論文では特に断りがなければ m はクエリ点の数を表す．

また，ある軌跡がどの程度 ” よく ” 各クエリ点を接続しているのかを評価するために，その軌跡から各クエリ点までの距離を考慮することで類似度関数を定義する．

クエリ点 q_i と軌跡 $R = \{p_1, p_2, \cdots, p_l\}$ 間の距離 $Dist_q$ を二点間のユークリッド距離 $Dist_e(q_i, p_j)$ を用いて次のように定義する．

$$Dist_q(q_i, R) = \min_{p_j \in R} \{Dist_e(q_i, p_j)\}$$

上式で表されるとおり，R と q_i の距離とは，q_i から R 上の任意の点への最短距離である．q_i に最も近い R 上の点を p_j とする．p_j は軌跡 R において q_i とマッチするといい，対 (p_j, q_i) をマッチドペアと呼ぶ．今，クエリ $Q = q_1, q_2, \cdots q_m$ に対して，軌跡 R の類似度 $sim(Q, R)$ を，各クエリ点 q_i におけるマッチドペアの距離に基づいて次のように定義する．

$$Sim(Q, R) = \sum_{i=1}^{m} e^{-Dist_q(q_i, R)}$$

定義 1（k-BCT 問い合わせ） 軌跡集合 $T = \{R_1, R_2, \cdots, R_n\}$ $(n \geq k)$, 点集合 Q と類似度関数に相当するものが与えられたとき，次のような条件を満たす k 軌跡 T' を T から見つけることを k-BCT 問い合わせとよぶ．

$$Sim(Q, R_i)_{R_i \in T'} \geq Sim(Q, R_j)_{R_j \in T - T'}$$

3 既存研究

本章では, 既存研究 [2] にて示されている, k-BCT 問い合わせアルゴリズムについて説明する．
既存研究における k-BCT 問い合わせの処理は，大きく分けて次の 2 つの手続きから構成される．

(1) 各クエリ点に対して, その近接点を構成点として含むような軌跡の集合をリストアップする．

(2) (1) においてリストアップされた軌跡の集合から，最も類似度関数値の高い軌跡を選択する．

ステップ (1) の効率的な処理のためには，各クエリ点に対する近接点の探索を高速に行う必要がある．そのため，既存研究では，全軌跡を構成する点の集合が R 木 [4] によりインデックス付けされていることを仮定している．以下，本研究においても，同様のインデックス付けの存在を仮定して話を進める．

3.1 k-NN 探索

軌跡点を R 木でインデックス付けすることにより，クエリ点に近接する軌跡点を効率よく探索する k-近傍 (k-NN) 探索 [3][5] を利用できる．既存研究における提案アルゴリズムでは，k-BCT 向けに設計された R 木上の k-近傍探索アルゴリズムとして，探索方法の違いにより，最良優先探索と深さ優先探索の二通りの探索手法が検討されている．

本論文では，k-NN 探索自体の改良はおこなわないが，以下にその概略を紹介する．そのためにまず MBR，R 木，MINDIST について定義する．

定義 2（MBR） MBR とは，オブジェクト群を内包する最小の矩形であり，かつその各辺が座標軸に平行な領域のことをいう．
MBR を R とし，その両端点を $S = \{s_x, s_y\}$，$T = \{t_x, t_y\}$ とする．ただし，$s_x \leq t_x$，$s_y \leq t_y$ である．このとき R を以下のように定義する．

$$R = (S, T)$$

定義 3（R 木） R 木とは，ノードと MBR を対応づけた B 木である．R 木の任意のノードに対応する MBR は，親ノードに対応する MBR に内包される．また，葉ノードには実際のオブジェクトが格納され，根ノードはすべてのオブジェクト群を内包する MBR である

定義 4（MINDIST） ある点 $P = (p_x, p_y)$ と MBR R との最小距離 $MINDIST(P, R)$ は次のように定義される．

$$MINDIST(P, R) = \sqrt{(p_x - r_x)^2 + (p_y - r_y)^2}$$

$$r_x = \begin{cases} s_x & \text{if } p_x < s_x \\ t_x & \text{if } p_x > t_x \\ p_x & \text{otherwise.} \end{cases} \qquad r_y = \begin{cases} s_y & \text{if } p_y < s_y \\ t_y & \text{if } p_y > t_y \\ p_y & \text{otherwise.} \end{cases}$$

R 木を用いた k-NN 探索においては，クエリ点を包含するような MBR に対して，それを根とする部分木を重点的に探索することにより近接点の発見を行う．探索の方法にはいくつかのバリエーションがあり，Zhou らの k-BCT 問い合わせアルゴリズムにおいては以下の方法が利用されている．

最良優先探索 [3]

最良優先探索とは，クエリ点から MBR までの MINDIST をキーとしてまだ訪れていないエントリをプライオリティキューで管理することで実現する探索方式である．最初にプライオリティキューには根ノードのエントリのみが含まれる．次に訪れるエントリを決める際，キューの先頭にあるエントリを選ぶ．選ばれたエントリはキューから削除され，その子エントリがキューに追加される．

この探索方式は深さ優先探索に比べて高速に探索を行えるが，メモリ使用量の保証ができず，最悪の場合はシステムが停止するおそれがある．

深さ優先探索 [5]

深さ優先探索の基本的な考えは，深さ優先で再帰的に R 木を下り，k-NN 候補をグローバルリストで管理している．この下方への処理は根ノードから始まり，内部ノードをクエリ点からの MINDIST でソートした際に最小の値を持つエントリを訪問することを葉ノードまで再帰的に繰り返す．上のレベルに引き返す際，これまでに見つかった k 番目の候補との距離より小さな MINDIST をもつエントリのみを訪問する．
この方法を実現するために，どのエントリを訪問したのかを保持する再帰スタックが必要となる．

この探索方式は最良優先探索に比べて探索に時間がかかるが，理論的にメモリ使用量の保証ができる

[2].

3.2 IKNN **アルゴリズム** [2]

以降，k-BCT の k と k-NN 探索の k との混同を避けるために，k-NN 探索を λ-NN 探索と呼ぶ．与えられたクエリ点に対して λ 個の近接点を探索するという意味である．

まずはじめに，以下のように各クエリ点に対して λ-NN 探索を行う．

$$\lambda\text{-NN}(q_1) = \{p_1^1, p_1^2, \cdots, p_1^\lambda\}$$
$$\lambda\text{-NN}(q_2) = \{p_2^1, p_2^2, \cdots, p_2^\lambda\}$$
$$\cdots$$
$$\lambda\text{-NN}(q_m) = \{p_m^1, p_m^2, \cdots, p_m^\lambda\}$$

軌跡点 p を含むような軌跡を $R(p)$ とすると，λ-NN(q_i) によって探索された $R(q_i^j)$ が構成する集合 C_i を q_i に対する候補集合とよぶ．$|C_i| \leq \lambda$ となることに注意が必要である．各クエリ点に対する候補集合の和をとることで $f = |\cup_i C_i|$ 個の異なった軌跡を含む集合 C が得られる．この C を k-BCT に対する候補集合と呼ぶ．

$$C = C_1 \cup C_2 \cup \cdots \cup C_m = \{R_1, R_2, \cdots, R_f\}$$

C に含まれる各軌跡 $R_x (x \in [1, f])$ に対しその類似度の下界 $LB(R_x)$ を次のように定義する．

$$LB(R_x) = \sum_{i\in[1,m]\wedge R_x\in C_i} \left(\max_{j\in[1,\lambda]\wedge p_i^j\in R_x} \{e^{-Dist_e(q_i, p_i^j)}\} \right)$$

同様にして，C に含まれない軌跡の類似度の上界 UB_n を次のように定義する．添え字の n はまだ C に含まれていない *non-scanned* ということを意味し，後程定義する C に含まれる軌跡の上界を表す UB と区別するために付している．

$$UB_n = \sum_{i=1}^{m} e^{-Dist_e(q_i, p_i^\lambda)}$$

この類似度の上界と下界について次の定理が成り立つ．

定理 1 [2]

k-BCT 問い合わせ $Q = \{q_1, q_2, \cdots, q_m\}$ に対して，各クエリ点 q_i に λ-NN 探索を行った後に得られる候補集合を C とする．このとき，k 本の軌跡からなるある部分集合 $C' \subseteq C$ が得られ,$\min_{R_x\in C'}\{LB(R_x)\} \geq UB_n$ が成り立つならば，Q に対する k-BCT は必ず C の中に含まれる．

上記定理 1 が満たされるような候補集合 C を構築した後，この C から k-BCT に対する解を抽出する．k-BCT を含む候補集合 C から問い合わせの解を抽出するアルゴリズムに必要となる $UB(R_x)(R_x \in C)$ を次のように定義する．

$$UB(R_x) = \sum_{i\in[1,m]\wedge R_x\in C_i} \left(\max_{j\in[1,\lambda]\wedge p_i^j\in R_x} \left\{e^{-Dist_e(q_i, p_i^j)}\right\} \right) + \sum_{i\in[1,m]\wedge R_x\notin C_i} \left(e^{-Dist_e(q_i, p_i^\lambda)} \right)$$

候補集合 C に含まれる全軌跡 R_x について，$UB(R_x)$ を計算する．候補集合 C を計算した $UB(R_x)$ で降順にソートし，先頭要素から順に類似度 $Sim(Q, R_x,)$ を計算する．計算した $Sim(Q, R_x)$ によって降順になるように k-BCT の解のリストに挿入し，このリストには k 個の軌跡のみを保持する．C の軌跡すべてについて $Sim(Q, R_x)$ を計算し終えるか，あるいは現在の解の候補リストの k 番目の軌跡の類似度が，次に類似度を計算しようとしている軌跡 R_{x+1} の軌跡の類似度の上界 $UB(R_{x+1})$ 以上であるとき現在の解のリストを k-BCT として返す．

前述のとおり，IKNN アルゴリズムには，λ-NN 探索として最良優先探索を用いる IKNN$_{bf}$ と，深さ優先探索を用いる IKNN$_{df}$ がある．また，本稿では詳細は割愛するが，論文 [2] に IKNN の各ステップにおける λ-NN 探索を改良するいくつかのアイデアも併せて提案されている．

4　提案手法

4.1　基本的な考え

第 3 章で紹介した IKNN アルゴリズムは，定理 1 を満たす候補集合 C を構築できるまで λ の値を定数 Δ で増やしながら λ-NN 探索を繰り返す．この λ-NN 探索の実行コストを削減するため，前処理としてあらかじめ特定の点に対して λ-NN 探索を実行して保存しておき，その結果を利用するというアルゴリズムを提案する．

4.2　諸定義

あらかじめ λ-NN 探索をおこない結果を用意しておく点のことをキャッシュ点とよび，cp_k と表す．また，その集合 $CP = \{cp_1, cp_2, \cdots, cp_n\}$ をキャッシュ集合とよぶ．ここで，λ にはあらかじめ適当な値を設定する必要がある．このとき λ に設定する値をキャッシュサイズ（cachesize）とよぶ．あるクエリ点 q_i と q_i に最も近い CP 上の点 cp_k の二点間の距離が閾値 r より小さいとき，q_i はキャッシュヒットしているという．

4.3　アルゴリズム

この手法を実現するためには，キャッシュ集合 CP からクエリ点 q_i に最も近いキャッシュ点を検索する必要がある．そこで，軌跡点を管理する R 木とは別に CP に含まれるキャッシュ点を管理する R 木を用いる．また，本研究では閾値 r として q_i とマッチドペアをなす軌跡点までの距離 $Dist_e(q_i, p_i^1)$ を用いる．

Algorithm1 にキャッシュを利用するための前処理を，Algorithm2 にキャッシュを用いた IKNN アルゴリズムを示す．

Algorithm 1 preprocess()

```
input : $Cache\ Set\ CP, cachesize$
1 Create the $R$Tree of CP;{ キャッシュ点を管理する $R$ 木を構築する }
2 Integer $\lambda \leftarrow cachesize$
3 for $each\ cp_k \in CP$ do
4     $cached\lambda$-NN($cp_k$) ←KNN($cp_k, \lambda$);{$cachesize$-NN を探索する }
5     register $cached\lambda$-NN($cp_k$){ 探索結果を保存しておく }
6 end for
```

5　実験および考察

本実験は下表に示す環境で行った．

CPU	AMD Athlon(tm) Dual Core Processor 4450e 2.30GHz
OS	Windows7 Proffessional
メモリ	2GB
記述言語	Java
データセット	Microsoft GeoLife Project が提供している北京市内の軌跡データセット [1]
軌跡総数	8349
軌跡点総数	1762612

実験は既存の IKNN と提案手法の比較と，提案手法におけるパラメータを変更した際に実行時間にどのような変化が起きるのか調査をおこなった．
なお，既存の IKNN アルゴリズムも提案手法も λ-NN 探索として深さ優先探索を実装しており，既存の IKNN アルゴリズムには既存研究 [2] で提案されている改良は実装していない．すべての実験に共通のこととして，入力されたクエリ点がキャッシュヒットする場合，ちょうどその点にキャッシュ点が用意されているものとする．すなわち，本実験では，キャッシュ中の点をクエリ点の近似値として利用するようなケースは考えない．

Algorithm 2 IKNNwithCache()

```
input : $k, Q$
output: $k$-BCT
1  Candidate Set C;
2  Upperbound $UB_n$;
3  Lowerbounds $LB[]$, $k$-$LB[]$;
4  Integer $\lambda \leftarrow k$;
5  while $true$ do
6    for $each\ q_i \in Q\ from\ q_1\ to\ q_m$ do
7      find the closest cache point $cp_k \in CP$ by traversing the $R$Tree of $CP$;{ 各クエリ点に最も近いキャッシュ点を探索する }
8      compute r;{ 閾値 $r$ を計算する }
9      if $Dist_e(q_i, cp_k) \leq r${ キャッシュヒットの判定 } then
10       $\lambda$-NN($q_i$) $\leftarrow$ $cached\lambda$-NN($cp_k$);{ キャッシュヒットの場合はキャッシュを利用 }
11       $C_i \leftarrow$ trajectories scanned by $\lambda$-NN($q_i$);
12     else
13       $\lambda$-NN($q_i$) $\leftarrow$KNN($q_i, \lambda$);{ キャッシュヒットしない場合は通常の IKNN を実行 }
14       $C_i \leftarrow$ trajectories scanned by $\lambda$-NN($q_i$);
15     end if
16   end for
17   $C \leftarrow C_1 \cup C_2 \cup \cdots \cup C_m$;
18   if $|C| \geq k$ then
19     compute $LB[]$ for all trajectories in $C$;
20     compute $UB_n$;
21     $k$-$LB[] \leftarrow LB[]$.topK();
22     if $k$-$LB[].min \geq UB_n$ then
23       $k$-BCT$\leftarrow$refine($C$);
24       return $k$-BCT;
25     end if
26   end if
27   $\lambda \leftarrow \lambda + \Delta$;
28 end while
```

5.1 既存の IKNN と提案手法の比較

本節では，既存の IKNN アルゴリズムと提案手法を比較するための実験をおこなう．実験内容は以下の二つである．

- k=15 に固定して m を 2 から 10 まで変化させたときの実行時間の比較
- m=8 に固定して k を 1 から 25 まで変化させたときの実行時間の比較

なお，提案手法におけるパラメータであるキャッシュサイズは 1500 で固定している．

5.1.1 クエリ点の数による実行時間の違い

実験結果を図 1 に示す．グラフからわかるとおり，m の値が大きくなるほど，わずかではあるがより実行時間を改善していることがわかる．しかし，実行時間の改善がわずかなのは入力したクエリが m の増加に対して IKNN の実行回数をそれほど増加させないからではないかと思われる．

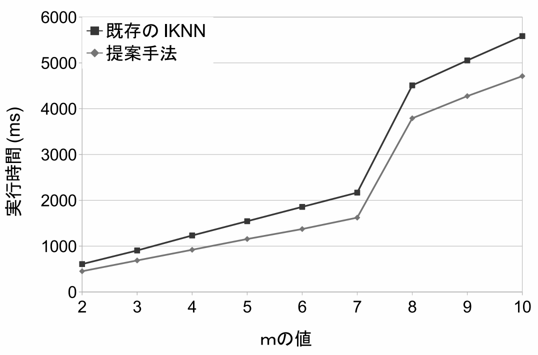


図 1

5.1.2 k の値による実行時間の違い

実験結果を図 2 に示す．このグラフは，本論文で提案している手法の特徴を表しているといえる．グラフからわかるとおり，k が小さな値である時，既存の IKNN アルゴリズムよりも実行時間が悪化している．しかし，k=7 を境に k=12 まで既存の IKNN アルゴリズムの実行時間よりはるかに改善されている．またその改善率も k=7 から k=12 まで

上昇し続け，k=12 でピークを迎えることがわかる．

これは，キャッシュサイズを 1500 に固定しているため，k が小さな値の時は候補集合 C のサイズが大きすぎて refine ステップでの計算時間が増加することで実行時間を悪化させる結果となっていると推測される．

一方，$k \geq 13$ のとき，実行時間は改善されてはいるものの，その改善率が小さいことがグラフからわかる．これは先程とは逆に，キャッシュを利用して構成された候補集合 C のサイズが十分でないためであると考えられる．

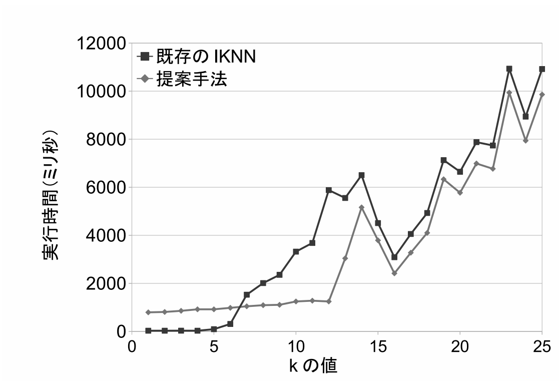


図 2

5.2 提案手法の実行時間の評価

本節では，提案手法のパラメータであるキャッシュサイズを変化させたとき実行時間にどのような影響が出るのか，入力されたクエリ点の総数に対して，キャッシュヒットしているクエリ点の数が変化するとき実行時間にどのような影響が出るのかを実験した．具体的には，以下の 2 設定のもとで実験を行った．

- キャッシュサイズを 500 から 3500 まで変化させたとき，以下の三つの条件で実行時間を計測
  - k=1,m=2 の場合
  - k=15,m=8 の場合
  - k=25,m=10 の場合
- k=15,m=10 の同一のクエリに対し，用意する利用可能なキャッシュ点の数を 0 から 10 まで変化させたときの実行時間を計測

なお，後者の実験ではキャッシュサイズを 2500 に固定している．

5.2.1 キャッシュサイズによる実行時間の変化

実験結果を図 3 に示す．グラフからわかるとおり，m や k の値によって実行時間が大きく改善されるキャッシュサイズが存在することが確認できた．一方で，キャッシュサイズが大きくなると，キャッシュ結果の探索のコストが，検索全体に占める割合が相対的に増大するため，とくに k, m の値が小さい場合は必ずしも速度向上につながらないということが見てとれる．入力される k や m の値の傾向に応じて適切なキャッシュサイズに変更するようなアルゴリズムを構成することができれば，多くの場合において実行時間が改善されることが期待される．

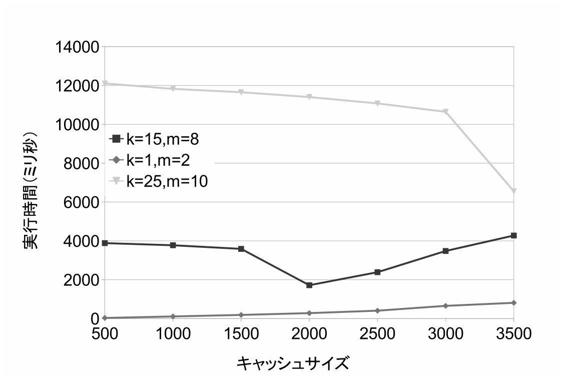


図 3

5.2.2 キャッシュを利用できるクエリ点の数による実行時間の変化

実験結果を図 4 に示す．グラフからわかるとおりすべてのクエリをキャッシュヒットさせることができれば劇的に実行時間が短縮できることがわかる．しかしながら，一つでもキャッシュヒットしないクエリ点が存在するとキャッシュヒット数 0，すなわち事実上既存の IKNN アルゴリズムより実行時間が悪化していることがわかる．

IKNN アルゴリズムは定理 1 が満たされるまで λ を定数 Δ で増加させながら探索を繰り返す．

キャッシュを利用できないクエリ点が存在する場合，その点に対しての候補集合が定理１を満たすまで定理１の計算を繰り返すことになる．また，大きなキャッシュサイズのキャッシュを利用する場合，その点についての候補集合が全体の候補集合を大きくする．そのため IKNN アルゴリズムの一回のステップで定理１を計算する際，候補軌跡の類似度の下界の計算コストが増加することが実行時間の悪化につながっているといえる．

上記の改善策として，キャッシュヒットしなかった点を含んでいるとき，一回目の探索で定理１を満たさなかったならば，λ にキャッシュサイズと同じ値を代入して次の探索を行うようにするといった方法が挙げられる．

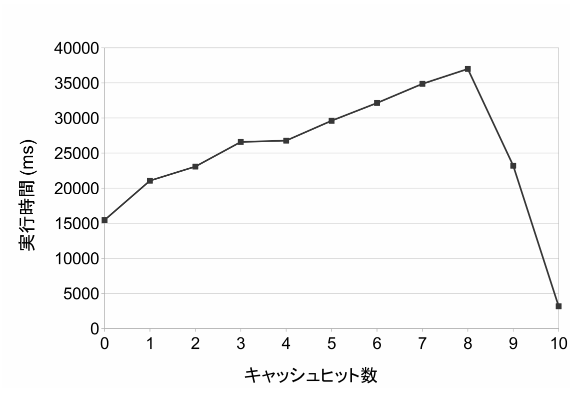


図 4

6　まとめと今後の課題

本研究ではクエリに対してあらかじめ λ-NN 探索の結果を用意しておくことにより IKNN アルゴリズム内で実行される λ-NN 探索を省略することで IKNN の実行時間を改善できることを明らかにした．しかしながら，k や m の値によって適切にキャッシュサイズを設定しなければ大幅な実行時間の改善につながらないこと，キャッシュヒットしないクエリ点が含まれると既存の手法より実行時間が悪化することも判明した．

また，本研究ではキャッシュ点がヒットする場合は意図的にキャッシュ点とクエリ点が一致するようにしているため，正しい実行結果が得られるが，実際にはキャッシュヒットしていてもキャッシュ点とクエリ点が一致していない場合，得られた実行結果がどの程度の誤差を含むのかに関しては一切評価していないためその点についても議論する必要がある．

今後の課題としては，キャッシュの効率的な実装および利用, また, NN 探索の実行時間を改善する手法の一つである LSH(Locality Sensitive Hashing) を利用した枝刈り等を応用することが挙げられる．

参考文献

[1] http://research.microsoft.com/en-us/projects/geolife/.

[2] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, Yu Zheng, and Xing Xie. Searching trajectories by locations - an efficiency study. *SIGMOD*, 2010.

[3] H.Samet G.R.Hjaltason. Distance browsing in spatial databases. *TODS*, 1999.

[4] A Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD*, pages 47–57, 1984.

[5] F.Vincent N.Roussopoulos, S.Kelley. R-trees: a dynamic index structure for spatial searching. *SIGMOD*, pages 71–79, 1995.

Memory Machine Models for GPUs

Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan
Email: nakano@cs.hiroshima-u.ac.jp

***Abstract*—The main contribution of this paper is to introduce two parallel memory machines, the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM). Unlike well studied theoretical parallel computational models such as PRAMs, these parallel memory machines are practical and capture the essential feature of GPU memory accesses. As a first step of the development of algorithmic techniques on the DMM and the UMM, we first evaluate the computing time for the contiguous access and the stride access to the memory on these models. We then go on to present parallel algorithms to transpose a 2-dimensional array on these models and evaluate their performance. We also how that, for any permutation given in off-line, data in an array can be moved efficiently along the given permutation both on the DMM and on the UMM. Finally, we show that the sum and the prefix-sums algorithms on the DMM and on the UMM. Since the computing time of our algorithms on the DMM and the UMM is equal to the sum of the lower bounds obtained from the memory bandwidth limitation and the latency limitation, they are optimal from the theoretical point of view. We believe that the DMM and the UMM can be good theoretical platforms to develop algorithmic techniques for GPUs.**

***Keywords*-memory banks, parallel computing models, parallel algorithms, stride memory access, matrix transpose, array permutation, prefix-sums, GPU, CUDA**

I. Introduction

A. Background

The research of parallel algorithms has a long history of more than 40 years. Sequential algorithms have been developed mostly on the Random Access Machine (RAM) [1]. In contrast, since there are a variety of connection methods and patterns between processors and memories, many parallel computing models have been presented and many parallel algorithmic techniques have been shown on them. The most well-studied parallel computing model is the Parallel Random Access Machine (PRAM) [2], [3], [4], which consists of processors and a shared memory. Each processor on the PRAM can access any address of the shared memory in a time unit. The PRAM is a good parallel computing model in the sense that parallelism of each problem can be revealed by the performance of parallel algorithms on the PRAM. However, since the PRAM requires a shared memory that can be accessed by all processors in the same time, it is imaginary and impractical.

The GPU (Graphical Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [5], [6], [7], [8]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [5], [9]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [10], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [11], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [10]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The global memory is implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [6], [11], [12]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access to the same memory banks in the same time, the access requests are processed sequentially. Hence, to maximize the memory access performance, threads of CUDA should access to distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, CUDA threads should perform coalesced access when they access to the global memory.

There are several previously published works that aim to present theoretical practical parallel computing models capturing the essence of parallel computers. Many researchers have been devoted to developing efficient parallel algorithms to find algorithmic techniques on such parallel computing models. For example, processors connected by interconnection networks such as hypercubes, meshes, trees, among others [13], bulk synchronous models [14], LogP models [15], reconfigurable models [16], among others. As far as we know, no sophisticated and simple parallel computing model for GPUs has been presented. Since GPUs are attractive parallel computing devices for many developers, it is challenging work to introduce a theoretical parallel computing model for GPUs.

B. Our Contribution: Introduction to the Discrete Memory Machine and the Unified Memory Machine

The first contribution of this paper is to introduce simple parallel memory machine models that capture the essential features of the bank conflict of the shared memory access and the coalescing of the global memory access. More specifically, we present two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs.

The outline of the architectures off the DMM and the UMM are illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [1], which can execute fundamental operations in a time unit. We do not discuss the architecture of the sea of threads in this paper, but we can imagine that it consists of a set of multi-core processors which can execute many threads in parallel. Threads are executed in SIMD [17] fashion, and the processors run on the same program and work on the different data. In principle, each thread is assigned a local memory (or local registers) that can access $O(1)$ words of data. However, sometimes, we assume that each thread has more than $O(1)$ local registers, if many registers are very useful to accelerate the computation. If this is the case, we assume that each thread has r local registers to store words of data. In either cases, we assume that each thread can access to a local register in 1 time unit.

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM.

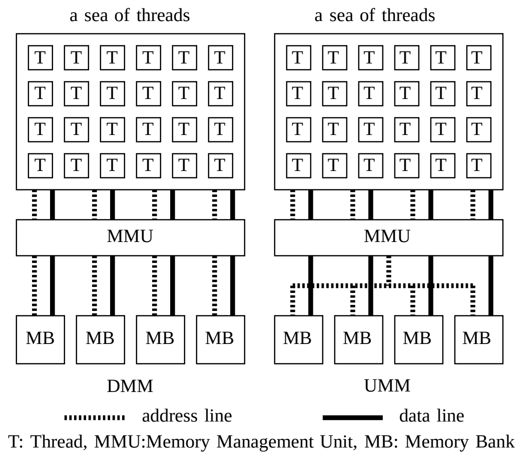


Figure 1. The architectures of the DMM and the UMM

The performance of algorithms on the PRAM is usually evaluated using two parameters: the size n of the input and the number p of processors. For example, it is well known that the sum of n numbers can be computed in $O(\frac{n}{p}+\log p)$ time on the PRAM [2]. We will use four parameters, the size n of the input, the number p of threads, the width w and the latency l of the memory when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of memory banks and the latency l is the number of time units to complete the memory access. Hence, the performance of algorithms on the DMM and the UMM is evaluated as a function of n (the size of a problem), p (the number of threads), w (the width of a memory), and l (the latency of a memory). Further, r (the number of local registers used by each thread) may be additionally used.

In NVIDIA GPUs, the width w of the shared memory and the global memory is 16 or 32. Also, the latency l of the global memory is several hundreds clock cycles. In CUDA, a grid can have at most 65535 blocks with at most 1024 threads each [10]. Thus, the number p of threads can be 65 million.

C. Position and Role of Memory Machine Models, the DMM and the UMM

The DMM and the UMM are theoretical models of parallel computation, that capture the essential feature of the shared memory and the global memory of GPUs. The architecture of the GPUs are more complicated. It is a hybrid of the DMM and the UMM. Also, when we develop efficient programs running on the GPUs, we need to consider several issues. NVIDIA GPUs have other features such as hierarchical architecture grid/block/thread. All threads are partitioned into equal sized blocks. Synchronization of all threads in each block can be done by calling barrier synchronization function `_syncthreads()`, which has fairly low overhead. On the other hand, no direct way is provided for synchronization of all threads in all blocks. There are several indirect ways of synchronization of all threads, but they have rather high overhead. It follows that, local barrier synchronization is acceptable while global barrier synchronization should be avoid. This fact is not incorporated in the DMM and the UMM. It may be possible to incorporate many features of GPUs and introduce a more exact parallel computing model for GPUs. If all features of GPUs are incorporated in our theoretical parallel models, they will be too complicated and need more parameters. The development of algorithms on such complicated models may have too much non-essential and tedious optimizations. Thus, we focus on just memory access features on the current GPUs, and introduce parallel computing models, the DMM and the UMM. Actually, efficient memory access is a key issue to develop high performance programs on the GPUs [12], [18]. Thus, we have introduced two simple parallel models, the DMM and the UMM, which focus on the memory access to the shared memory and the global memory of NVIDIA GPUs. Sometimes, direct implementation of efficient algorithms on the DMM and the UMM may not be efficient on an actual GPU. However, we believe that algorithmic techniques on the DMM and the UMM are useful for developing algorithms on GPUs.

In [19], a GPU memory model has been shown and a cache-efficient FFT has been presented. However, their model focuses on the cache mechanism and ignores the coalescing and the bank conflict. Also, in [20], acceleration techniques for GPU have been discussed. Although they are taking care of the limited bandwidth of the global memory, the details of the memory architecture are not considered. As far as we know, this paper is the first work that introduces simple theoretical parallel computing models for GPUs. We believe that the development of algorithms on these models are useful to investigate algorithmic techniques for the GPUs.

Further, the parallel architecture of our memory machines make senses not only for GPUs, but also for a class of all parallel machines that support a uniform shared address space designed using a set of off-chip memory chips or on-chip memory blocks. Usually, DRAMs [21] are used to constitute an off-chip memory. An on-chip memory block can be implemented in a rectangular block of a VLSI chip. For example, modern FPGAs has a lot of block RAMs, each of which can store 18kbit data [22], can be used as a memory bank. To increase the capacity and the bandwidth, we should use multiple on-chip memory chips or on-chip memory blocks. To connect a set of processor cores with these memory elements though the MMU, the architecture of the UMM and the DMM make a whole lot of sense.

D. Our Contribution: Fundamental Data Movement Algorithms on the DMM and the UMM

The second contribution of this paper is to evaluate the performance of two memory access methods, *the contiguous access* and *the stride access* on the DMM and the UMM. The reader should refer to Figure 2 for illustrating these two access methods by four threads $T(0), T(1), T(2)$, and $T(3)$. It is well-known that the contiguous access is much more efficient than the stride access on the GPUs [12]. We will show that, the contiguous access is also more efficient on the DMM and on the UMM. More specifically, we first show that the contiguous access of an array of size n can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time units on the DMM and the UMM. We also show two lower bounds, $\Omega(\frac{n}{w})$ time units by *the bandwidth limitation* and $\Omega(\frac{nl}{p})$ time units by *the latency limitation* to access all of data in an array of size n. Thus, the contiguous access on the DMM and the UMM is optimal. Further, we will show that the stride access on the DMM can be done in $O(\frac{n}{w}\cdot \mathrm{GCD}(\frac{n}{p}, w)+\frac{nl}{p})$ time units on the DMM, where $\mathrm{GCD}(\frac{n}{p}, w)$ is the greatest common divisor of $\frac{n}{p}$ and w. Hence, the stride access on the DMM is optimal if $\frac{n}{p}$ and w are co-prime. The stride access on the UMM can be done in $O(\min(n, \frac{n}{w}\cdot\frac{n}{p}+\frac{nl}{p}))$ time units. Hence, the stride access on the UMM needs an overhead of a factor of $\frac{n}{p}$.

From these memory access results, we have one important observation as follows. The factor $\frac{n}{w}$ in the computing time comes from *the bandwidth limitation* of the memory. It takes at least $\frac{n}{w}$ time units to access whole data in an array of size n from the memory bandwidth w. Also, the factor $\frac{nl}{p}$ comes from *the latency limitation*. From the memory access latency l, each thread cannot send a new access request in l time units. It follows that, each thread can access to the memory once in l time units and any consecutive l time units can have at most p access requests by p threads. Hence, $\frac{nl}{p}$ time units are necessary to access all of the elements in an array of size n. Further, to hide the latency overhead factor $\frac{nl}{p}$ from the bandwidth limitation factor $\frac{n}{w}$, the number p of the threads must be no less than wl. We can confirm this fact from a different aspect. We can think that *the memory access requests* are stored in a pipeline buffer of size l for each memory bank. Since we have w memory banks, we have wl pipeline registers to store memory access requests at all. Since at most one memory request per thread are stored in the wl pipeline registers, $wl \leq p$ must be satisfied to fill the pipeline registers full of memory access requests.

E. Our Contribution: Transpose and Permutation on the DMM and the UMM

The third contribution is to show optimal off-line permutation algorithms on the DMM and the UMM.

As a preliminary step, we show transposing algorithms for a 2-dimensional array of size $\sqrt{n}\times\sqrt{n}$. In [18], several techniques are presented for transposing a 2-dimensional array stored in the shared memory and the global memory

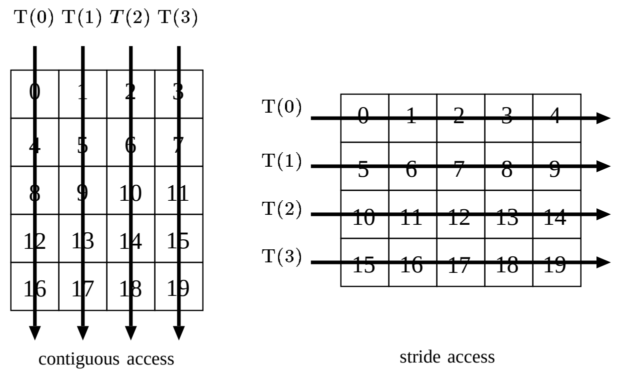


Figure 2. The contiguous access and the stride access for $p = 4$ and $n = 16$.

on GPUs. We have adapted these techniques on the DMM and the UMM. The resulting transposing algorithms run in $O(\frac{n}{w} + \frac{nl}{p})$ time units and in $O((\frac{n}{w} + \frac{nl}{p})\sqrt{\frac{w}{r}})$ time units on the DMM and the UMM, respectively.

We next show a permutation algorithm on the DMM. We use a graph theoretic result of bipartite graph edge-coloring to schedule data routing. The resulting algorithm runs in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM.

Finally, we show a permutation algorithm on the UMM. This algorithm repeatedly performs transposing and row-wise permutation. The resulting algorithm runs in $O((\frac{n}{w} + \frac{nl}{p})\sqrt{\frac{w}{r}})$ time units on the UMM, respectively.

F. Our Contribution: the sums and the prefix-sums algorithms on the DMM and the UMM

Suppose that an array a of n numbers is given. The prefix-sums of a is the array of size n such that the i-th ($0 \leq i \leq n-1$) element is $a[0] + a[1] + \cdots + a[i]$. Clearly, a sequential algorithm can compute the prefix sums by executing $a[i+1] \leftarrow a[i+1] + a[i]$ for all i ($0 \leq i \leq n-1$). The computation of the prefix-sums of an array is one of the most important algorithmic procedures. Many algorithms such as graph algorithms, geometric algorithms, image processing and matrix computation call prefix-sums algorithms as a subroutine. In particular, many parallel algorithms uses a parallel prefix-sums algorithm. For example, the prefix-sums computation is used to obtain the pre-order, the in-order, and the post-order of a rooted binary tree in parallel [2]. So, it is very important to develop efficient parallel algorithms for the prefix-sums.

This paper shows an optimal prefix-sums algorithm on the DMM and the UMM. We first show that the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units using p threads on the DMM and the UMM with width w and latency l. We then go on to discuss the lower bound of the time complexity and show three lower bounds, $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{nl}{p})$-time latency limitation, and $\Omega(l\log n)$-time reduction limitation. From this discussion, the computation of the sum and the prefix-sums takes at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units on the DMM and the UMM. Thus, the sum algorithm is optimal. For the computation of the prefix-sums, we first evaluate the computing time of a well-known naive algorithm [29], [4]. We show that a naive prefix-sums algorithm runs in $O(\frac{n\log n}{w} + \frac{nl\log n}{p} + l\log n)$ time. Hence, this fact shows this naive prefix-sums algorithm is not optimal and it has an overhead of factor $\log n$ both for the bandwidth limitation $\frac{n}{m}$ and for the latency limitation $\frac{nl}{p}$. Finally, we show an optimal parallel algorithm that computes the prefix-sums of n numbers in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units on the DMM and the UMM. However, this algorithm uses work space of size n and it may not be acceptable if the size n of the input is very large. We also show that the prefix-sums can also be computed in the same time units, even if work space can store only $\min(p\log p, wl\log(wl))$ numbers.

Several techniques for computing the prefix-sums on GPUs have been shown in [29]. They have presented a complicated data routing technique to avoid the bank conflict in the computation of the prefix-sums. However, their algorithm performs memory access to distant locations in parallel and it performs non-coalesced memory access. Hence it is not efficient for the UMM, that is, the global memory of GPUs. In [30] a work-efficient parallel algorithm for prefix-sums on the GPU has been presented. However, the algorithm uses work space of $n\log n$, and also the performance of the algorithm has not been evaluated.

This paper is organized as follows. We first define the DMM and the UMM in Section II. In Section III, we evaluate the performance of the DMM and the UMM for the contiguous access and the stride access to the memory. Section IV discusses lower bounds obtained by *the bandwidth limitation* and *the latency limitation*. Section V presents algorithms that perform the transpose of 2-dimensional array on the DMM and the UMM. In Section VI, we show that any permutation on an array can be done efficiently on the DMM. Section VII presents a permutation algorithm on the UMM. Using the contiguous access, we show that the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units in Section VIII. We then go on to discuss the lower bound of the time complexity and show three lower bounds, $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{nl}{p})$-time latency limitation, and $\Omega(l\log n)$-time reduction limitation in Section IV. Section IX shows a naive prefix-sums algorithm, which runs in $O(\frac{n\log n}{w} + \frac{nl\log n}{p} + l\log n)$ time units. Finally, we show an optimal parallel prefix-sums algorithm running in $O(\frac{n}{w} + \frac{nl}{p} + l\log n)$ time units. Section XI offers conclusion of this paper.

II. Parallel Memory Machines: DMM and UMM

We first introduce *the Discrete Memory Machine (DMM)* of width w and latency l. Let $m[i]$ ($i \geq 0$) denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j+w], m[j+2w], m[j+3w], \ldots\}$ $(0 \leq j \leq w-1)$ denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k+l-1$ time units to complete k continuous access requests to a particular bank.

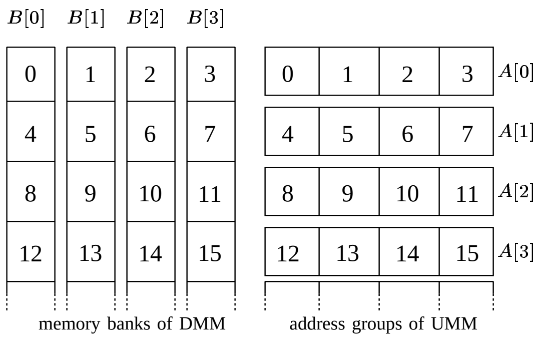


Figure 3. Banks and address groups for $w = 4$

Let $T(0), T(1), \ldots, T(p-1)$ denote p threads on the memory machine. We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{\mathrm{T}(i \cdot w), \mathrm{T}(i \cdot w+1), \ldots, \mathrm{T}((i+1) \cdot w - 1)\}$ $(0 \leq i \leq \frac{p}{w} - 1)$. Warps are activated for memory access in turn, and w threads in a warp try to access the memory in the same time. In other words, $W(0), W(1), \ldots, W(w-1)$ are activated in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not activated for memory access and is skipped. When $W(i)$ is activated, w threads in $W(i)$ send memory access requests, one request per thread, to the memory bank. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread send a memory access request, it must wait l time units to send a new memory access request.

For the reader's benefit, let us evaluate the time for memory access using Figure 4 on the DMM for $p = 8$, $w = 4$, and $l = 3$. In the figure, $p = 8$ threads are partitioned into $\frac{p}{w} = 2$ warps $W(0) = \{\mathrm{T}(0), T(1), T(2), T(3)\}$ and $W(1) = \{\mathrm{T}(4), \mathrm{T}(5), \mathrm{T}(6), \mathrm{T}(7)\}$. As illustrated in the figure, 4 threads in $W(0)$ try to access $\boldsymbol{m[0], m[1], m[10]}$, and $\boldsymbol{m[6]}$, and those in $W(1)$ try to access $m[8], m[9], m[14]$, and $m[15]$. The time for the memory access are evaluated under the assumption that memory access are processed by imaginary l pipeline stages with w registers each as illustrated in the figure. Each pipeline register in the first stage receives memory access requests from threads in an activated warp. Each i-th $(0 \leq i \leq w-1)$ pipeline register receives the request to memory bank $M(i)$. In each time unit, a memory request in a pipeline register is moved to the next one. We assume that the memory access completes when the request reaches a last pipeline register.

Note that, the architecture of pipeline registers illustrated in Figure 4 are imaginary, and it is used only for evaluating the computing time. The actual architecture should involves a multistage interconnection network [23], [24] or sorting network [25], [26], to route memory access requests.

Let us evaluate the time for memory access on the DMM. First, access requests for $m[0], m[1], m[6]$ are sent to the first stage. Since $m[6]$ and $m[10]$ are in the same bank $B[2]$, their memory requests cannot be sent to the first stage in the same time. Next, the $m[10]$ is sent to the first stage. After that, memory access requests for $m[8], m[9], m[14], m[15]$ are sent in the same time, because they are in different memory banks. Finally, after $l - 1 = 2$ time units, these memory requests are processed. Hence, the DMM takes 5 time units to complete the memory access.

We next define *the Unified Memory Machine* (*UMM* for short) of width w as follows. Let $A[j] = \{m[j \cdot w], m[j \cdot w + 1], \ldots, m[(j+1) \cdot w - 1]\}$ denote the j-th address group. We assume that memory cells in the same address group are processed in the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp access to the memory in turn.

Again, let us evaluate the time for memory access using Figure 4 on the UMM for $p = 8$, $w = 4$, and $l = 3$. The memory access requests by $W(0)$ are in three address groups. Thus, three time units are necessary to send them to the first stage. Next, two time units are necessary to send memory access requests by $W(1)$, because they are in two address groups. After that, it takes $l - 1 = 2$ time units to process the memory access requests. Hence, totally $3 + 2 + 2 = 7$ time units are necessary to complete all memory accesses.

III. Sequential memory access operations

We begin with simple operations to see the potentiality of the DMM and the UMM. Let p and w be the number of threads and the width of the memory machines. We assume that an array m of size n is arranged in the memory. Let $m[i]$ $(0 \leq i \leq n-1)$ denote the i-th word of the memory. We assume that $w \leq p$ and n is divisible by p. We consider two access operations to the memory such that each of the p threads accesses to the $\frac{n}{p}$ memory cells out of the n memory cells. Suppose that array m is arranged in a 2-dimensional

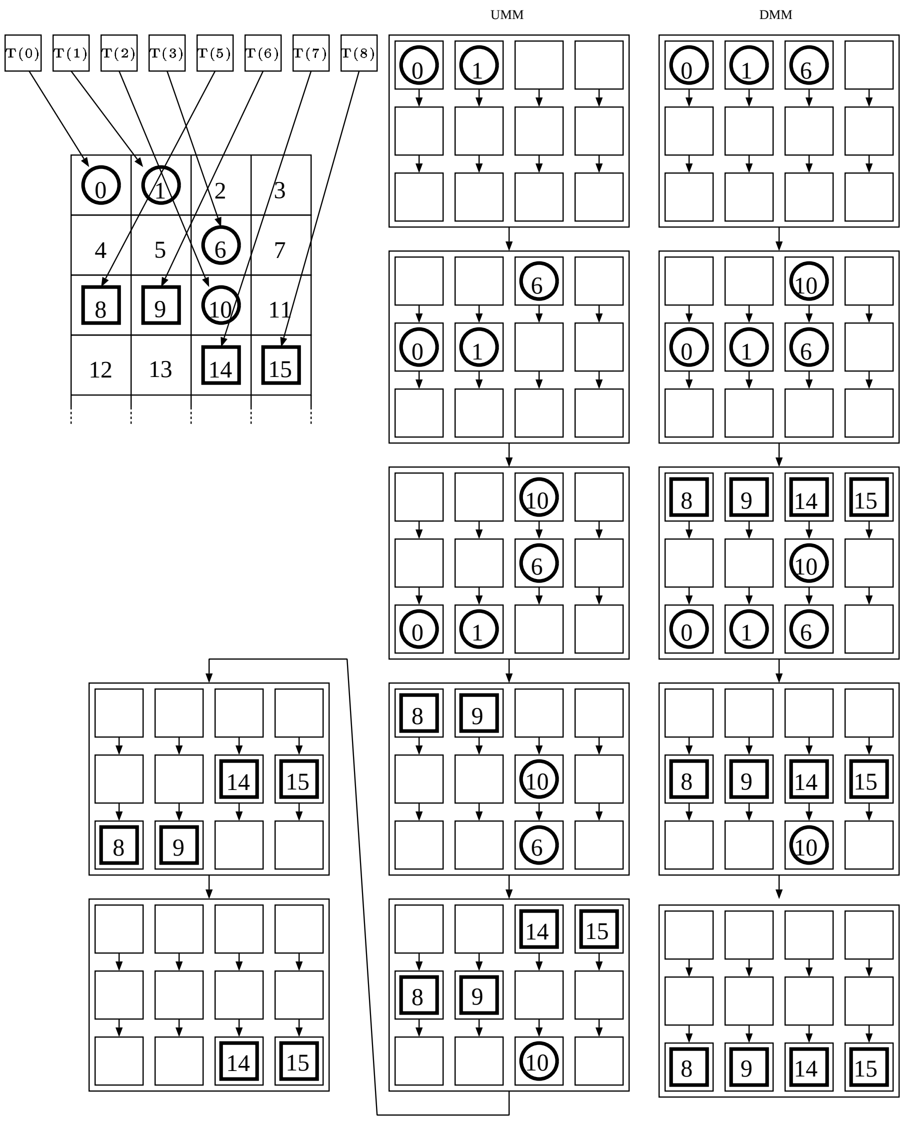


Figure 4. An example of memory access

array m_c of size $\frac{n}{p} \times p$ (i.e. $\frac{n}{p}$ rows and p columns) such that $m_c[i][j] = m[i \cdot p + j]$ for all i and j ($0 \le i \le \frac{n}{p} - 1$ and $0 \le j \le p - 1$). Similarly, let m_s be a 2-dimensional array of size $p \times \frac{n}{p}$ (i.e. p rows and $\frac{n}{p}$ columns) such that $m_s[i][j] = m[i \cdot \frac{n}{p} + j]$ for all i and j ($0 \le i \le p - 1$ and $0 \le j \le \frac{n}{p} - 1$). The contiguous access and the stride access can be written as follows:

```
[Contiguous Access]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    T$(i)$ accesses to $m_c[t][i]$   $(= m[t \cdot p + i])$
```

```
[Stride Access]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    T$(i)$ accesses to $m_s[i][t]$   $(= m[i \cdot \frac{n}{p} + t])$
```

The readers should refer to Figure 2 for illustrating the contiguous and stride accesses for $n = 20$, $p = 4$, and $\frac{n}{p} = 5$. At time $t = 0$, p threads access to contiguous locations $m[0], m[1], m[2]$, and $m[3]$ in the contiguous access, while they access to distant locations $m[0], m[5], m[10]$, and $m[15]$ in the stride access.

Let us evaluate the time necessary to complete the contiguous access and the stride access. In the contiguous access, w threads in each warp access memory cells in different memory banks. Hence, the memory access by a warp takes l time units. Also, *the memory access requests* by a warp is sent in every 1 time unit. Since we have

$\frac{p}{w}$ warps, the access to p memory cells by p threads can be completed in $\frac{p}{w}+l-1$ time units. Since this access operation is repeated $\frac{n}{p}$ times, the contiguous access takes $(\frac{p}{w}+l-1)\cdot\frac{n}{p}=O(\frac{n}{w}+\frac{nl}{p})$ time units on the DMM. In the contiguous access on the UMM, each warp access to the memory cells in the same address group. Thus, the memory access by a warp takes l time unit and the whole contiguous access is completed in $O(\frac{n}{w}+\frac{nl}{p})$ time units.

The performance analysis of the stride access on the DMM is a bit complicated. Let us start with a simple case: $\frac{n}{p}=w$. In this case, the p threads access to p memory cells $m[t], m[w+t], m[2w+t], \ldots, m[(p-1)w+t]$ for each t $(0 \leq t \leq w-1)$. Unfortunately, these memory cells are in the same memory bank $B[t]$. Hence, the memory access by a warp takes $w+l-1$ time units and the memory access to the p memory cells takes $w\cdot\frac{p}{w}+l-1=p+l-1$ time units. Thus, the stride access when $\frac{n}{p}=w$ takes at least $(p+l-1)\cdot\frac{n}{p}=O(n+\frac{nl}{p})$ time units.

Next, let us consider general case. The w threads in the first warp access to $m[t], m[\frac{n}{p}+t], m[2\frac{n}{p}+t], \ldots, m[(w-1)\frac{n}{p}+t]$ for each t $(0 \leq t \leq w-1)$. These w memory cells are allocated in the banks $B[t \bmod w], B[(\frac{n}{p}+t) \bmod w], B[(2\frac{n}{p}+t) \bmod w], \ldots, B[((w-1)\frac{n}{p}+t) \bmod w]$. Let $L=\mathrm{LCM}(\frac{n}{p},w)$ and $G=\mathrm{GCD}(\frac{n}{p},w)$ be the Least Common Multiple and the Greatest Common Divisor of $\frac{n}{p}$ and w, respectively. From the basic number theory, it should be clear that $t \bmod w=(\frac{L}{\frac{n}{p}}\cdot\frac{n}{p}+t) \bmod w$, and the values of $t \bmod w$, $(\frac{n}{p}+t) \bmod w$, …, $((\frac{L}{\frac{n}{p}}-1)\cdot\frac{n}{p}+t) \bmod w$ are distinct. Thus, the w memory cells are in the $\frac{L}{\frac{n}{p}}=\frac{w}{G}$ banks $B[t \bmod w], B[(\frac{n}{p}+t) \bmod w], B[(2\frac{n}{p}+t) \bmod w], \ldots, B[((\frac{w}{G}-1)\frac{n}{p}+t) \bmod w]$ equally, and each bank has G memory cells of the w memory cells. Hence, the w threads in a warp take $G+l-1$ time units for each t, and the p threads take $G\cdot\frac{p}{w}+l-1$ time units for each t. Therefore, the DMM takes $(G\cdot\frac{p}{w}+l-1)\cdot\frac{n}{p}=O(\frac{nG}{w}+\frac{nl}{p})$ time units to complete the stride access. If $\frac{n}{p}=w$ then $G=w$ and the time for the stride access is $O(n+\frac{nl}{p})$. If $\frac{n}{p}$ and w are co-prime, $G=1$ and the stride access takes $O(\frac{n}{w}+\frac{nl}{p})$ time units.

Finally, we will evaluate the computing time of the stride access on the UMM. If $\frac{n}{p} \geq w$ (i.e. $n \geq pw$), then the w memory cells are accessed by w threads in a warp are in the different address group. Thus, w threads access to w memory cells in $w+l-1$ time units, and the stride access takes $(w\cdot\frac{p}{w}+l-1)\cdot\frac{n}{p}=O(n+\frac{nl}{p})$ time units. When $\frac{n}{p}<w$ (i.e. $n<pw$), the w memory cells accessed by w threads in a warp are in at most $\lceil\frac{(w-1)\frac{n}{p}+1}{w}\rceil \leq \frac{n}{p}$ address groups. Hence, the stride access by p threads for each t takes at most $\frac{n}{p}\cdot\frac{p}{w}+l-1=\frac{n}{w}+l-1$ time units, and thus, the whole stride access takes $(\frac{n}{w}+l-1)\cdot\frac{n}{p}=O(\frac{n^2}{pw}+\frac{nl}{p})$ time units. Consequently, the stride access can be completed in $O(\min(n,\frac{n^2}{pw})+\frac{nl}{p}))$ time units for all values of $\frac{n}{p}$. Thus, we have,

Theorem 1: The contiguous access and the stride access on the DMM and the UMM can be completed in time units shown in Table I.

Suppose that we have two arrays a and b of size n each. The copy operation from a and b can be done by the contiguous read and the contiguous write in an obvious way. Since both the DMM and the UMM can perform the contiguous access in $O(\frac{n}{w}+\frac{nl}{p})$ time units from Theorem 1, we have,

Corollary 2: The copy between two arrays of size n each can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time units using p threads on the DMM and on the UMM with width w and latency l.

IV. The lower bounds of the computing time and the latency hiding

Let us discuss the lower bound of the computing time of the DMM and the UMM for non-trivial problems, which require to access all words in an input array of size n.

Since the bandwidth of the memory is w, at most w words in the memory can be accessed in a time unit. Thus, it takes at least $\Omega(\frac{n}{w})$ time to solve a non-trivial problem. We call the $\Omega(\frac{n}{w})$-time lower bound *the bandwidth limitation*.

Since the memory access takes latency l, a thread can send at most $\frac{t}{l}$ memory access requests in t time units. Thus, the p threads can send at most $\frac{pt}{l}$ access requests totally. Since at least n memory access requests to solve a non-trivial problem, $\frac{pt}{l} \geq n$ must be satisfied. Thus, at least $t=\Omega(\frac{nl}{p})$ time units are necessary. We call the $\Omega(\frac{nl}{p})$-time lower bound *the latency limitation*.

From the discussion above, we have,

Theorem 3: Both the DMM and the UMM with p threads, width w, and latency l takes at least $\Omega(\frac{n}{w}+\frac{nl}{p})$ time units to solve a non-trivial problem of size n.
From Theorem 3, the copy operation for Corollary 2 is optimal. In the following sections, we will show algorithms for data movement running in $O(\frac{n}{w}+\frac{nl}{p}$ time. Since data movements are non-trivial problems, they have a lower bound of $\Omega(\frac{n}{w}+\frac{nl}{p})$ time units. Hence, the algorithms for data movement are optimal.

Let us discuss two factors, $\frac{n}{w}$ for bandwidth limitation and $\frac{nl}{p}$ for latency limitation. If $\frac{n}{w} \geq \frac{nl}{p}$, that is, $wl \leq p$, then the bandwidth limitation dominates the latency limitation. As illustrated in Figure 4, both the DMM and the UMM have wl imaginary pipeline registers. Each thread can occupy one of the wl imaginary pipeline registers for memory access. Thus, we need at least wl threads to fill all the pipeline registers with memory access requests. Otherwise, that is, if $wl>p$, then a set of wl pipeline registers always has an empty one. It follows that, for the purpose of hiding the latency overhead, the number p of threads must be at least the number wl of the pipeline registers.

Table I
THE RUNNING TIME FOR THE CONTIGUOUS ACCESS AND THE STRIDE ACCESS

	DMM	UMM
Contiguous Access	$O(\frac{n}{w}+\frac{nl}{p})$	$O(\frac{n}{w}+\frac{nl}{p})$
Stride Access	$O(\frac{n}{w}\cdot\mathrm{GCD}(\frac{n}{p},w)+\frac{nl}{p})$	$O(\min(n,\frac{n^2}{pw})+\frac{nl}{p})$

n =#data, p =#threads, w =memory bandwidth, l =memory latency

V. TRANSPOSE OF A 2-DIMENSIONAL ARRAY

Suppose that a 2-dimensional array a and b of size $\sqrt{n}\times\sqrt{n}$ is arranged in the memory. The transpose of the 2-dimensional array is a task to move a word of data stored in $a[i][j]$ to $b[j][i]$ for all $(0\le i,j\le\sqrt{n}-1)$.

Let us start with a straightforward transpose algorithm using the contiguous access and the stride access. The following algorithm transposes a 2-dimensional array a of size $\sqrt{n}\times\sqrt{n}$.

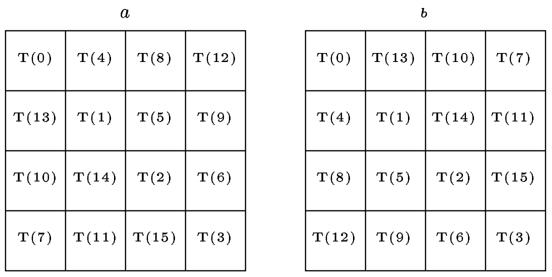
[Straightforward transposing algorithm]
```
for $t \leftarrow 0$ to $\frac{n}{p}-1$
  for $i \leftarrow 0$ to $p-1$ do in parallel
    $j \leftarrow (t\cdot p+i)/\sqrt{n}$
    $k \leftarrow (t\cdot p+i) \bmod \sqrt{n}$
    $\mathrm{T}(i)$ performs $b[j][k] \leftarrow a[k][j]$
```

On the PRAM, simultaneous reading and simultaneous writing by processors can be done in $O(1)$ time. Hence, this straightforward transposing algorithm runs in $O(\frac{n}{p})$ time on the PRAM. Also, it takes at least $\Omega(\frac{n}{p})$ time to access n words by p processors on the PRAM. Thus, this straightforward transposing algorithm is time optimal for the PRAM.

Since the straightforward algorithm involves the stride access, it is not difficult to see that the DMM and the UMM take $O(\frac{n}{w}\cdot\mathrm{GCD}(\sqrt{n},w)+\frac{nl}{p})$ time units and $O(\min(n,\frac{n^2}{pw})+\frac{nl}{p})$ time units for transposing a 2-dimensional array, respectively. On the DMM, $\mathrm{GCD}(\sqrt{n},w)=w$ if $\sqrt{n}$ is divisible by w. If this is the case, the transpose takes $O(n)$ time units the DMM. We will show that, regardless of the value of n, the transpose can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time units both on the DMM and on the UMM.

We first show an efficient transposing algorithm on the DMM. The technique used in this algorithm is essentially the same as the diagonal block reordering presented in [18]. The key idea is to access the array in diagonal fashion. The details of the algorithm are spelled out as follows:

[Transpose by the diagonal access on the DMM]
```
for $t \leftarrow 0$ to $\frac{n}{p}-1$
  for $i \leftarrow 0$ to $p-1$ do in parallel
    $j \leftarrow (t\cdot p+i)/\sqrt{n}$
    $k \leftarrow (t\cdot p+i) \bmod \sqrt{n}$
    $\mathrm{T}(i)$ performs $b[(j+k) \bmod \sqrt{n}][k] \leftarrow a[k][(j+k) \bmod \sqrt{n}]$
```


a

T(0)	T(4)	T(8)	T(12)
T(13)	T(1)	T(5)	T(9)
T(10)	T(14)	T(2)	T(6)
T(7)	T(11)	T(15)	T(3)

b

T(0)	T(13)	T(10)	T(7)
T(4)	T(1)	T(14)	T(11)
T(8)	T(5)	T(2)	T(15)
T(12)	T(9)	T(6)	T(3)

Figure 5. Transposing on the DMM

The readers should refer to Figure 5 for illustrating the indexes of threads reading from memory cells in a and writing in memory cells of b for $n=p=16$ and $w=4$. From the figure, we can confirm that threads $\mathrm{T}(j\cdot 4+0),\mathrm{T}(j\cdot 4+1),\mathrm{T}(j\cdot 4+2),\mathrm{T}(j\cdot 4+3)$ read from memory cells in diagonal location of a and write to memory cells in diagonal location of b for every j $(0\le j\le 3)$. Thus, reading and writing to memory banks by w threads in a warp are different. Hence, p threads can copy p memory cells in $\frac{p}{w}+l-1$ time units and thus the total computing time is $(\frac{p}{w}+l-1)\cdot\frac{n}{p}=O(\frac{n}{w}+\frac{nl}{p})$ time units. Therefore, we have,

Lemma 4: The transpose of a 2-dimensional array of size $\sqrt{n}\times\sqrt{n}$ can be done in $O(\frac{n}{w}+\frac{nl}{p})$ time units using p threads on the DMM with memory width w and latency l.

Next, we will show that the transpose of a 2-dimensional array can be also done in $O(\frac{n}{w}+\frac{nl}{p})$ on the UMM if every thread has w local registers. As a preliminary step, we will show that the UMM can transpose a 2-dimensional array of size $w\times w$ in wl time units using w threads with each thread having a local storage of size w. We assume that each thread has w local registers. Let $r_i[0],r_i[1],\ldots r_i[w-1]$ denote w local registers of $\mathrm{T}(i)$.

[Transpose by the rotating technique on the UMM]
```
for $t \leftarrow 0$ to $w-1$
  for $i \leftarrow 0$ to $w-1$ do in parallel
    $\mathrm{T}(i)$ performs $r_i[t] \leftarrow a[t][(t+i) \bmod w]$
for $t \leftarrow 0$ to $w-1$
  for $i \leftarrow 0$ to $w-1$ do in parallel
    $\mathrm{T}(i)$ performs $b[t][(t-i) \bmod w] \leftarrow r_i[(t-i) \bmod w]$
```

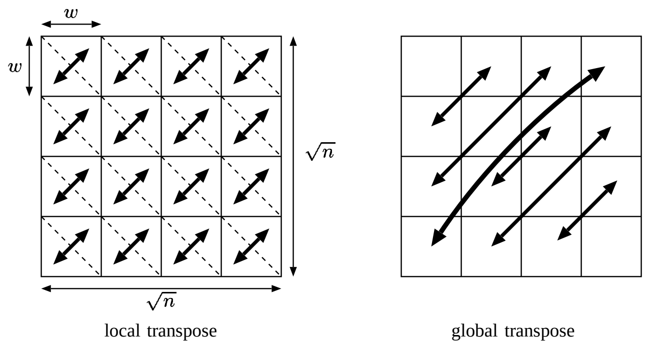


Figure 7. Transposing on the UMM

Let (i, j) denote the value stored in $a[i][j]$ initially. The readers should refer to Figure 6 for illustrating how these values are transposed.

Let us confirm that the algorithm above correctly transpose the 2-dimensional array a. In other words, we will show that, when the algorithm terminates, $b[i][j]$ stores (j, i). It should be clear that, the value stored in $r_i[t]$ is $(t, (t+i) \bmod w)$. Since $((t-i) \bmod w, t)$ is stored in $r_i[(t-i) \bmod w]$, it is also stored in $b[t][(t-i) \bmod w]$ when the algorithm terminates. Thus, every $b[i][j]$ $(0 \leq i, j \leq w-1)$ stores (j, i). This completes the proof of the correctness of our transpose algorithm on the UMM.

Let us evaluate the computing time. In the reading operation $r_i[t] \leftarrow a[t][(t+i) \bmod w]$, w memory cells $a[t][(t + 0 \bmod w)], a[t][(t+1 \bmod w)], \ldots, a[t][(t+w-1 \bmod w)]$ are in the different memory banks. Also, in the writing operation $b[t][(t-i) \bmod w] \leftarrow r_i[(t-i) \bmod w]$, w memory cells $b[t][(t-0 \bmod w)], b[t][(t-1 \bmod w)], \ldots, b[t][(t-(w-1) \bmod w)]$ are in the different memory banks. Thus, each reading and writing operation can be done in $O(l)$ time units and this algorithm runs in $O(wl)$ time units.

The transpose of a larger 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done by repeating the transpose of a 2-dimensional array of size $w \times w$. The algorithm has two steps. More specifically, the 2-dimensional array is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ subarrays of size $w \times w$. Let $A[i][j]$ $(0 \leq i, j \leq \frac{n}{w} - 1)$ denote the subarray of size $w \times w$. First, each subarray $A[i][j]$ is transposed independently using w threads (local transpose). After that, the corresponding words of $A[i][j]$ and $A[j][i]$ are swapped for all i and j in an obvious way (global transpose). Figure 7 illustrates the transposing algorithm on the UMM.

Let us evaluate the computing time to complete the transpose of a $\sqrt{n} \times \sqrt{n}$ 2-dimensional array. Suppose that we have p $(\leq \frac{n}{w})$ threads and partition the p threads into $\frac{p}{w}$ groups with w threads each. We assign $\frac{n}{w^2} / \frac{p}{w} = \frac{n}{pw}$ subarrays to each warp of w threads. Each of the $\frac{p}{w}$ warps transposes each of the $\frac{p}{w}$ subarrays in parallel. It takes $O(w \cdot (\frac{p}{w} + l)) = O(p + wl)$ time units. The transposing of $\frac{p}{w}$ subarrays is repeated $\frac{n}{pw}$ times, the total computing time for transposing all subarrays is $\frac{n}{pw} \cdot O(p + wl) = O(\frac{n}{w} + \frac{nl}{p})$ time units. It should have no difficulty to confirm that the global transpose can be also done in $O(\frac{n}{w} + \frac{nl}{p})$ time units. Thus we have,

Lemma 5: The transpose of a 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time using p $(w \leq p \leq \frac{n}{w})$ threads on the UMM with each thread having w local registers.

Finally, we will show the case that each thread of the UMM has r $(< w)$ local registers. We first show how we transpose a 2-dimensional array a of size $\sqrt{rw} \times \sqrt{rw}$ using w threads. We first partition w threads into $\sqrt{rw}$ groups of $\sqrt{\frac{w}{r}}$ threads each. Each group has totally $\sqrt{\frac{w}{r}} \cdot r = \sqrt{rw}$ local registers and works as a single thread with $\sqrt{rw}$ local registers. Each group i $(0 \leq i \leq \sqrt{\frac{w}{r}})$ with $\sqrt{rw}$ local registers can read and store $\sqrt{rw}$ data $a[0][(i+0) \bmod \sqrt{rw}], a[1][(i+1) \bmod \sqrt{rw}], \ldots, a[\sqrt{rw}-1][(i+\sqrt{rw}-1) \bmod \sqrt{rw}]$ in the local registers. After that, they are written into $b[(i+0) \bmod \sqrt{rw}][0], a[(i+1) \bmod \sqrt{rw}][1], \ldots, a[(i+\sqrt{rw}-1) \bmod \sqrt{rw}][\sqrt{rw}-1]$. All groups read and write the arrays in turn, the transpose of a 2-dimensional array a of size $\sqrt{rw} \times \sqrt{rw}$ can be done in $O(l\sqrt{rw})$ time units.

Similarly to Lemma 5, we perform the transpose of a 2-dimensional array a of size $\sqrt{n} \times \sqrt{n}$. For this purpose, we partition a into $\sqrt{\frac{n}{rw}} \times \sqrt{\frac{n}{rw}}$ subarrays of size $\sqrt{rw} \times \sqrt{rw}$. Let us evaluate the computing time. The p threads can transpose $\frac{p}{w}$ subarrays in parallel in $O(\sqrt{rw} \cdot (\frac{p}{w} + l)) = O(p\sqrt{\frac{r}{w}} + l\sqrt{rw})$ time. Since we have $\frac{n}{rw}$ subarrays, this transpose operation is repeated $\frac{n}{rw} / \frac{p}{w} = \frac{n}{rp}$ times. Thus, the local transpose can be done in $O(p\sqrt{\frac{r}{w}} + l\sqrt{rw}) \cdot \frac{n}{rp} = O(\frac{n}{\sqrt{rw}} + \frac{nl}{p} \cdot \sqrt{\frac{w}{r}}) = O((\frac{n}{w} + \frac{nl}{p}) \cdot \sqrt{\frac{w}{r}})$ time units. The global transpose is just a copy of data, it can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units. Hence, we have,

Lemma 6: The transpose of a 2-dimensional array of size $\sqrt{n} \times \sqrt{n}$ can be done in $O((\frac{n}{w} + \frac{nl}{p}) \cdot \sqrt{\frac{w}{r}})$ time using p $(w \leq p \leq \frac{n}{r})$ threads on the UMM with each thread having r $(r \leq w)$ local registers.

Lemma 6 implies that the transpose by the UMM with r local registers has a overhead of factor $\sqrt{\frac{w}{r}}$.

VI. Permutation of an Array on the DMM

In Section V, we have presented algorithms to transpose a 2-dimensional array on the DMM and the UMM. The main purpose of this section is to show algorithms that perform any permutation of an array. Since a transpose is one of the permutations, the results of this section is a generalization of those presented in Section V.

Let a and b be one dimensional arrays of size n each, and P be a permutation of $(0, 1, \ldots, n-1)$. The goal of permutation of an array is to copy a word of data stored in $a[i]$ to $b[P(i)]$ for every i $(0 \leq i \leq n-1)$. We assume that, permutation P is given in offline. We will show that,

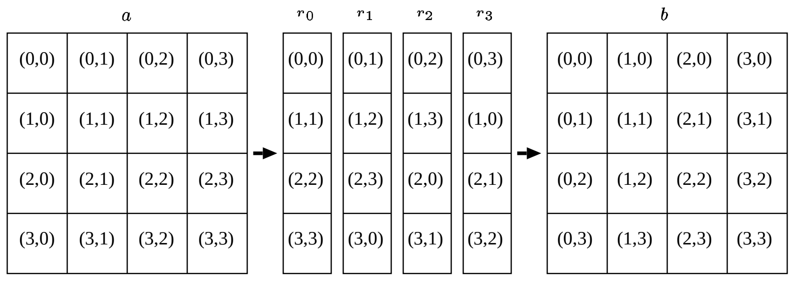


Figure 6. Transposing of a 2-dimensional array of size $w \times w$ on the UMM

for given any permutation P, permutation of an array can be done efficiently on the DMM and the UMM.

Let us start with evaluating the performance of the straightforward permutation algorithm. Suppose we need to do permutation of an array a of size n and permutation P is given.

[Straightforward permutation algorithm]

```
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  for $j \leftarrow 0$ to $p - 1$ do in parallel
    $i \leftarrow t \cdot p + j$
    T$(j)$ performs $b[P(i)] \leftarrow a[i]$
```

Clearly each t takes $O(1)$ time unit on the PRAM. Hence, the straightforward algorithm runs in $O(\frac{n}{p})$ time units on the PRAM.

This straightforward permutation algorithm also works correctly on the DMM and the UMM. However, it may take a lot of time to complete the permutation. In the worst case, this straightforward algorithm takes $O(n)$ time units on the DMM and the UMM if all writing operation to $b[P(i)]$ are in the same bank on the DMM or in the different address groups on the UMM. We will show that any permutation of an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM and the UMM.

If we can schedule reading/writing operations for permutation such that w threads in a warp read from distinct banks and write in distinct banks on the DMM, the permutation can be done efficiently. For such scheduling, we use the following important graph theoretic result [27], [28]:

Theorem 7 (König): A regular bipartite graph with degree ρ is ρ-edge-colorable.

Figure 8 illustrates an example of a regular bipartite graph with degree 4 painted by 4 colors. Each edge is painted by one of the 4 colors such that no node is connected to edges with the same color. In other words, no two edges with the same color share a node. The readers should refer to [27], [28] for the proof of Theorem 7.

We show a permutation algorithm on the DMM. Suppose that a permutation P of $(0, 1, \ldots, n-1)$ is given. We draw a bipartite graph $G = (U, V, E)$ of P as follows:

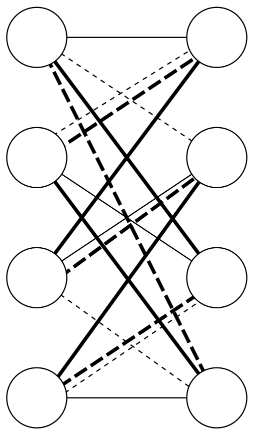


Figure 8. A regular bipartite graph with degree 4

- $U = \{B[0], B[1], B[2], \ldots, B[w-1]\}$ is a set of nodes each of which corresponds to a bank of a.
- $V = \{B[0], B[1], B[2], \ldots, B[w-1]\}$ is a set of nodes each of which corresponds to a bank of b.
- For each pair source $a[i]$ and destination $b[P(i)]$, E has a corresponding edge connecting $B[i \bmod w] (\in U)$ and $B[P(i) \bmod w] (\in V)$.

Clearly, an edge $(B[u], B[v])$ $(0 \leq u, v \leq w-1)$ corresponds to a word of data to be copied from bank $B[u]$ of a to $B[v]$ of b. Also, $G = (U, V, E)$ is a regular bipartite graph with degree $\frac{n}{w}$. Hence, G is $\frac{n}{w}$-colorable from Theorem 7. Suppose that all of the n edges in E are painted by $\frac{n}{w}$ colors 0, 1, …, $\frac{n}{w} - 1$. We determine value $s_{i,j}$ $(0 \leq i \leq \frac{n}{w} - 1, 0 \leq j \leq w-1, 0 \leq s_{i,j} \leq n-1)$ such that an edge $(B[s_{i,j} \bmod w], B[P(s_{i,j}) \bmod w])$ with color i corresponds to a pair of source $a[s_{i,j}]$ and destination $b[P(s_{i,j})]$. It should have no difficulty to confirm that, for each i,

- w banks $B[s_{i,0} \bmod w], B[s_{i,1} \bmod w], \ldots, B[s_{i,w-1} \bmod w]$ are distinct, and
- w banks values $B[P(s_{i,0}) \bmod w], B[P(s_{i,1}) \bmod w], \ldots, B[P(s_{i,w-1}) \bmod w]$ are distinct.

Thus, we have an important lemma as follows:

Lemma 8: Let $s_{i,j}$ denote a source defined above. For each i, we have, (1) $a[s_{i,0}]$, $a[s_{i,1}]$, …, $a[s_{i,w-1}]$ are in different banks, and (2) $b[P(s_{i,0})]$, $b[P(s_{i,1})]$, …, $b[P(s_{i,w-1})]$ are in different banks.

We can perform the bank conflict-free permutation using $s_{i,j}$. The details are spelled out as follows.

[Permutation algorithm on the DMM]

```
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  for $j \leftarrow 0$ to $p - 1$ do in parallel
    $i \leftarrow t \cdot p + j$
    $k \leftarrow s_{i/w, i \bmod w}$
    $\mathrm{T}(j)$ performs $b[P(k)] \leftarrow a[k]$
```

Since $b[P(k)] \leftarrow a[k]$ are performed for all k ($0 \leq k \leq n-1$), this algorithm performs data movement along permutation P correctly. We will show that this permutation algorithm terminates in $O(\frac{n}{w} + \frac{nl}{p})$ time units. For $t = 0$, warp $W(q)$ ($0 \leq q \leq \frac{p}{w} - 1$) with w threads $\mathrm{T}(wq), \mathrm{T}(wq+1), \ldots, \mathrm{T}(w(q+1)-1)$ performs $b[P(s_{q,0})] \leftarrow a[s_{q,0}]$, $b[P(s_{q,1})] \leftarrow a[s_{q,1}]$, …, $b[P(s_{q,w-1})] \leftarrow a[s_{q,w-1}]$ in parallel. From Lemma 8, these w threads read from different banks in a and write to different banks in b. Thus, p threads complete operations for $t = 0$ in $O(\frac{p}{w} + l)$ time units. Similarly, we can prove that the operation for every t can be done in $O(\frac{p}{w} + l)$ time units. Thus the total running time is $\frac{n}{p} \cdot O(\frac{p}{w} + l) = O(\frac{n}{w} + \frac{nl}{p})$ time units. Thus, we have,

Theorem 9: Any permutation on an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units using p threads on the DMM with width w and latency l.

VII. Permutation of an array on the UMM

The main purpose of this section is to show a permutation algorithm on the UMM. Our permutation algorithm uses the transpose algorithm on the UMM presented in Section V.

We start with a small array. Suppose that we have an array a of size w and permutation P on it. Since all elements in a are in the same address group, they can be read/written in a time unit. Thus, any permutation of an array a of size w can be done in $O(l)$ time units.

Next, we show a permutation algorithm for an array a of size w^2. We can consider that a permutation is defined on a 2-dimensional array a. In other words, the goal of permutation is to move a word of data stored in $a[i][j]$ to $a[P(i \cdot w + j)/w][P(i \cdot w + j) \bmod w]$ for every i and j ($0 \leq i, j \leq w-1$). We first show an algorithm for the row-wise permutation which is a permutation satisfying $P(i \cdot w + j)/w = i$ for all i and j. Figure 9 shows an example of row-wise permutation. In this figure, we assume that each $a[i][j]$ is initially storing $(P(i \cdot w + j)/w, P(i \cdot w + j) \bmod w]) = (i, P(i \cdot w + j) \bmod w])$. After the permutation, it is copied to $a[i][P(i \cdot w + j) \bmod w]$ and thus, each $a[i][j]$ stores (i, j).

We use p threads ($w \leq p \leq w^2$) partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ with w threads each.

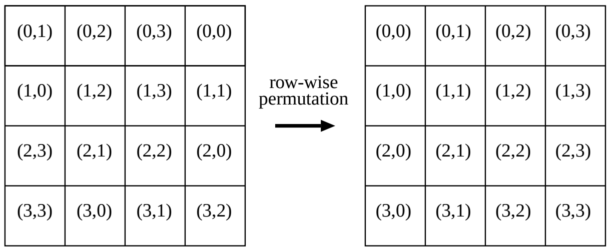


Figure 9. Row-wise permutation

The details of the row-wise permutation algorithm are as follows.

[Row-wise permutation algorithm]

```
for $t \leftarrow 0$ to $\frac{w^2}{p} - 1$
  for $i \leftarrow 0$ to $\frac{p}{w}$ do in parallel
    $W(i)$ performs permutation of the $(t \cdot \frac{p}{w} + i)$-th row.
```

Clearly, each row of an array a of size w^2 corresponds to an address group. For each t and i, $W(i)$ can perform a permutation of a row in $O(l)$ time units. Hence, for each t, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ can perform the row-wise permutation of $\frac{p}{w}$ rows in $O(\frac{p}{w} + l)$ time units. Thus, the row-wise permutation algorithm terminates in $\frac{w^2}{p} \cdot (\frac{p}{w} + l) = O(w + \frac{w^2 l}{p})$ time units. Hence we have,

Lemma 10: Any row-wise permutation of a two-dimensional array of size $w \times w$ can be done in $O(w + \frac{w^2 l}{p})$ time units using p threads ($w \leq p \leq w^2$) on the UMM with width w and latency l.

We next show an algorithm for the column-wise permutation , which is a permutation satisfying $P(i \cdot w + j) \bmod w = j$ for all i and j. This can be done by three steps as follows:

[Column-wise permutation on the UMM]
Step 1: Transpose the two-dimensional array
Step 2: Row-wise permute the two-dimensional array
Step 3: Transpose the two-dimensional array

Figure 10 illustrates the data movement of the three steps. Again, in this figure, we assume that each $a[i][j]$ is initially storing $(P(i \cdot w + j)/w, P(i \cdot w + j) \bmod w) = (P(i \cdot w + j) \bmod w, j)$. After the transpose in Step 1, $a[j][i]$ stores $(P(i \cdot w + j) \bmod w, j)$. The row-wise permutation is performed such that $a[j][i]$ stores (i, j). Finally, by transposing in Step 3, $a[i][j]$ stores (i, j).

Since column-wise permutation can be done by transposing and row-wise permutation, from Lemma 5 and Lemma 10, we have,

Lemma 11: Any column-wise permutation of a two-dimensional array of size $w \times w$ can be done in $O(wl)$ time units using w threads on the UMM with each thread having w local registers.

We next show any permutation of a 2-dimensional array

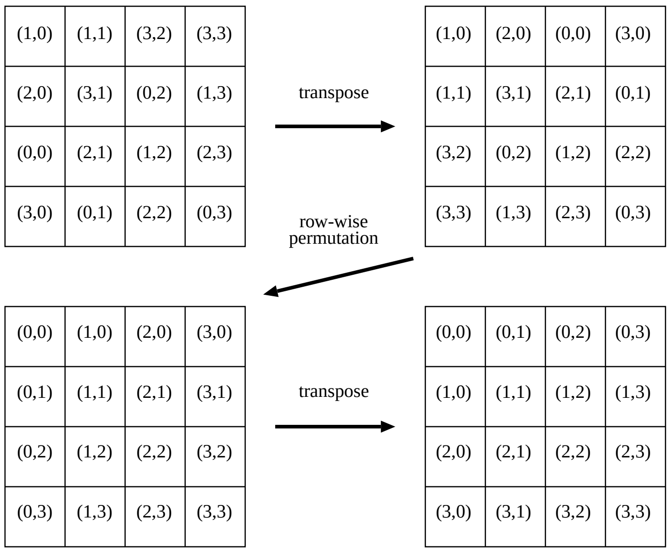


Figure 10. Column-wise permutation

of size $w \times w$ can be done in $O(wl)$ time units using w threads on the UMM by the row-wise permutation and the column-wise permutation. For a given permutation P on a 2-dimensional array a, we draw a bipartite graph $G = (U, V, E)$ as follows:

- $U = \{A[0], A[1], A[2], \ldots, A[w-1]\}$ is a set of nodes each of which corresponds to an address group of source.
- $V = \{A[0], A[1], A[2], \ldots, A[w-1]\}$ is a set of nodes each of which corresponds to an address group of destination.
- For each pair source $a[i][j]$ and destination $a[P(i \cdot w + j)/w][P(i \cdot w + j) \bmod w]$, E has a corresponding edge connecting $A[i] (\in U)$ and $A[P(i \cdot w + j)/w] (\in V)$.

For example if a word of data in $a[1][3]$ is copied to $a[2][4]$ by permutation P, an edge is drawn from node $A[1]$ in U and node $A[2]$ in V. Clearly, G is a regular bipartite graph with degree w. From Theorem 7, this bipartite graph can be painted using w colors such that w edges painted by the same color never share a node.

Suppose that, for a given permutation P on a 2-dimensional array a of size $w \times w$, we have painted edges in w colors $0, 1, \ldots, w-1$. Since each edge corresponds to a data stored in a, we can think that data is painted by the same color as the corresponding edge. Permutation can be done by three steps as follows:

[Permutation on the UMM]
Step 1: Row-wise permute the 2-dimensional array.
Step 2: Column-wise permute the 2-dimensional array.
Step 3: Row-wise permute the 2-dimensional array.

Let us see how permutation of each step is determined by edge coloring. As before, we assume that $a[i][j]$ is storing $(P(i \cdot w + j)/w, P(i \cdot w + j) \bmod w)$ and show that after the permutation algorithm is executed $a[i][j]$ stores (i, j). The readers should refer to Figure 11 for illustrating the data movement of the permutation algorithm for $w = 4$. From the figure we can confirm the following lemma:

Lemma 12: Suppose that data stored in a 2-dimensional array of $w \times w$ are painted by w colors using edge coloring of the corresponding bipartite graph above. We have: (1) data in the same row are painted by different colors, and (2) data painted by the same color has different row destination.
Since nodes in U are connected to w edges painted by different colors, we have (1) above. Also, since w edges painted by the same color connected to different nodes in V, we have (2) above.

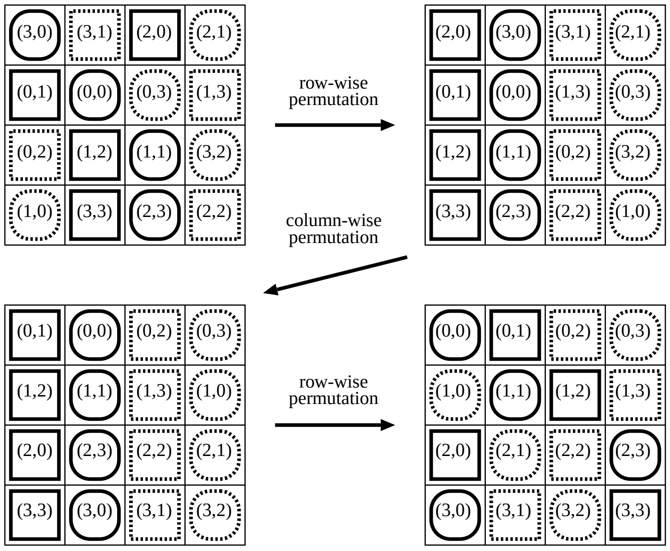


Figure 11. Illustrating a data movement of the permutation algorithm on the UMM

In Step 1, row-wise permutation is performed such that data with color i $(0 \leq i \leq w-1)$ are stored in the i-th column. From Lemma 12 (1), w data in each row are painted by w colors, Step 1 is possible. Step 2 uses column-wise permutation to move data to the final row destination. From Lemma 12 (2), w data in each column has different w row destination, Step 2 is possible. Finally, in Step 3, row-wise permutation is performed to move data to the final column destination.

Since the permutation algorithm on the UMM performs the row-wise permutation and the column-wise permutation, from Lemma 10 and Lemma 11, we have,

Lemma 13: Any permutation of an array of size w^2 can be done in $O(wl)$ time units using w threads on the UMM with each thread having local memory of w words.

We go on to show a permutation algorithm on a larger array a. Suppose we need to perform permutation of array

a of size w^4. We can consider that an array a is a 2-dimensional array of size $w^2 \times w^2$. We use the permutation algorithm for Lemma 13 to perform the row-wise permutation of the 2-dimensional array of size $w^2 \times w^2$. Similarly to the permutation algorithm for Lemma 13, we generate a bipartite graph with $G = (U, V, E)$ such that

- $U = \{0, 1, 2, \ldots, w^2 - 1\}$ is a set of nodes each of which corresponds to a row of source.
- $V = \{0, 1, 2, \ldots, w^2 - 1\}$ is a set of nodes each of which corresponds to a row of destination.
- For each pair source $a[i][j]$ and destination $a[P(i \cdot w + j)/w^2][P(i \cdot w + j) \bmod w^2]$, E has a corresponding edge connecting $i (\in U)$ and $P(i \cdot w + j)/w (\in V)$.

Similarly to the permutation algorithm for Lemma 13, any permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in three steps, row-wise permutation, column-wise permutation, and then row-wise permutation. The key idea is to use the permutation algorithm for Lemma 13 to perform the row-wise permutation and the column-wise permutation. We will discuss the details of the row-wise permutation and the column-wise permutation of a 2-dimensional array of size $w^2 \times w^2$

We show that the row-wise permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in $O(w^3 + \frac{w^4 l}{p})$ time units using p threads on the UMM. The p threads are partitioned into $\frac{p}{w}$ warps. First, each of the $\frac{p}{w}$ warps assigned a row of the first $\frac{p}{w}$ rows performs the row-wise permutation of the first $\frac{p}{w}$ row in parallel. This can be done by the permutation algorithm for Lemma 13, which runs $O(wl)$ time units. Note that, each of the w threads of a warp requests at most $O(w)$ memory access in the permutation algorithm for Lemma 13. The first memory access requests by the p threads in $\frac{p}{w}$ warps are completed $\frac{p}{w} + l$ time units. Since the memory access requests by p threads are repeated $O(w)$ times, the row-wise permutation of the first $\frac{p}{w}$ rows is completed in $O((\frac{p}{w} + l) \cdot w) = O(p + wl)$ time units. Since we have w^2 rows, this operation is repeated $w^2 / \frac{p}{w} = \frac{w^3}{p}$ times. Thus, the row-wise permutation can be done in $O((p + wl) \cdot \frac{w^3}{p}) = O(w^3 + \frac{w^4 l}{p})$ time units on the UMM.

Similarly to the row-wise permutation of a 2-dimensional array of size $w \times w$ shown in Figure 10, the column-wise permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done by transpose, row-wise permutation, and transpose. The transpose of a 2-dimensional array of size $w^2 \times w^2$ can be done in $O(w^3 + \frac{w^4 l}{p})$ time units on the UMM from Lemma 5. Also, the row-wise permutation can be done in $O(w^3 + \frac{w^4 l}{p})$ time units. Thus, the column-wise permutation can be done in $O(w^3 + \frac{w^4 l}{p})$ time units.

We are now in a position to show our permutation algorithm for a 2-dimensional array of size $w^2 \times w^2$. Similarly to permutation of a 2-dimensional array of size $w \times w$, permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in three steps, row-wise permutation, column-wise permutation and row-wise permutation. Since each step can be done in $O(w^3 + \frac{w^4 l}{p})$ time on the UMM, any permutation of a 2-dimensional array of size $w^2 \times w^2$ can be done in $O(w^3 + \frac{w^4 l}{p})$ time units on the UMM.

We can use the same technique for a permutation of an array of size $w^4 \times w^4$. The readers should have no difficulty to confirm that any permutation can be done in $O(w^7 + \frac{w^8 l}{p})$ time units on the UMM using p threads.

Repeating the same technique, we can obtain a permutation algorithm for an array of size $n = w^c \times w^c$. Permutation of a 2-dimensional array of size $w^c \times w^c$ can be done by executing the row-wise permutation recursively three times and the transpose for an array of size $w^{c/2} \times w^{c/2}$ twice. If the size n of an array satisfies $n \leq w^{O(1)}$, that is, $c = O(1)$, then the depth of the recursion is constant. If this is the case, the computing time is $O(w^{2c-1} + \frac{m^{2c} l}{p}) = O(\frac{n}{w} + \frac{nl}{p})$. Thus, we have,

Lemma 14: Any permutation of an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units ($w \leq p \leq \frac{n}{w}$) on the UMM with each thread having w local registers provided that $n \leq w^{O(1)}$.

Finally, if each register has only r ($\leq w$) local registers, we can use the transpose algorithm for Lemma 6. If this is the case, we have,

Theorem 15: Any permutation of an array of size n can be done in $O((\frac{n}{w} + \frac{nl}{p}) \cdot \sqrt{\frac{w}{r}})$ time units ($w \leq p \leq \frac{n}{r}$) on the UMM with each thread having r ($r \leq w$) local registers provided that $n \leq w^{O(1)}$.

VIII. An optimal parallel algorithm for computing the sum

The main purpose of this section is to show an optimal parallel algorithm for computing the sum on the memory machine models.

Let a be an array of $n = 2^m$ numbers. Let us show an algorithm to compute the sum $a[0] + a[1] + \cdots + a[n-1]$. The algorithm uses a well-known parallel computing technique which repeatedly computes the sums of pairs. We implement this technique to perform contiguous memory access. The details are spelled out as follows:

```
[**Optimal algorithm for computing the sum**]
for $t \leftarrow m-1$ downto 0 do
 for $i \leftarrow 0$ to $2^t - 1$ do in parallel
   $a[i] \leftarrow a[i] + a[i + 2^t]$
```

Figure 12 illustrates how the sums of pairs are computed. From the figure, the reader should have no difficulty to confirm that this algorithm compute the sum correctly.

We assume that p threads to compute the sum. For each t ($0 \leq t \leq m-1$), 2^t operations "$a[i] \leftarrow a[i] + a[i + 2^t]$" are performed. These operation involve the following memory access operations:

- reading from $a[0], a[1], \ldots, a[2^t - 1]$,

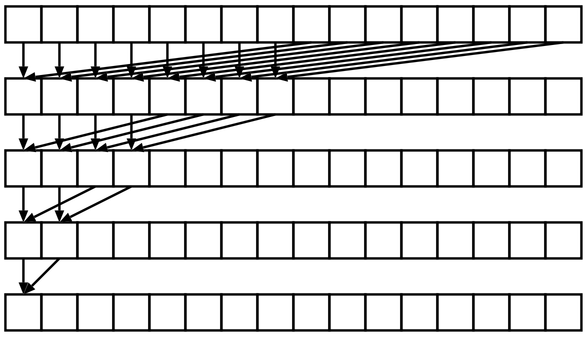


Figure 12. Illustrating the summing algorithm for n numbers

- reading from $a[2^t], a[2^t+1], \dots, a[2 \cdot 2^t - 1]$, and
- writing in $a[0], a[1], \dots, a[2^t - 1]$,

Since these memory access operations are contiguous, they can be done in $O(\frac{2^t}{w} + \frac{2^t l}{p} + l)$ time using p threads both on the DMM and on the UMM with width w and latency l from Theorem 1. Thus, the total computing time is

$$\begin{aligned}\sum_{t=0}^{m-1} O(\frac{2^t}{w} + \frac{2^t l}{p} + l) &= O(\frac{2^m}{w} + \frac{2^m l}{p} + lm) \\ &= O(\frac{n}{w} + \frac{nl}{p} + l \log n)\end{aligned}$$

and we have,

Lemma 16: The sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the DMM and on the UMM with width w and latency l.

IX. A NAIVE PREFIX-SUMS ALGORITHM

We assume that an array a with $n = 2^m$ numbers is given. Let us start with a well-known naive prefix-sums algorithm for array a [29], [30], and show it is not optimal. The naive prefix-sums algorithm is written as follows:

[A naive prefi x-sums algorithm]

```
for $t \leftarrow 0$ to $p-1$ do
  for $i \leftarrow 2^t$ to $n-1$ do in parallel
    $a[i] \leftarrow a[i] + a[i-2^t]$
```

Figure 13 illustrates how the prefix-sums are computed.

We assume that p threads are available and evaluate the computing time of the naive prefix-sums algorithm. The following three memory access operations are performed for each t $(0 \leq t \leq p-1)$: can be done by

- reading from $a[2^t], a[2^t+1], \dots, a[n-2]$,
- reading from $a[2^t+1], a[2^t+2], \dots, a[n-1]$, and
- writing in $a[2^t+1], a[2^t+2], \dots, a[n-1]$.

Each of the three operations can be done by contiguous memory access for $n - 2^t$ memory cells. Hence, the computing time of each t is $O(\frac{n-2^t}{w} + \frac{(n-2^t)l}{p} + l)$ from Theorem 1. The total computing time is:

$$\sum_{t=0}^{p-1} O(\frac{n-2^t}{w} + \frac{(n-2^t)l}{p} + l) \quad = \quad O(\frac{n \log n}{w} + \frac{nl \log n}{p}),$$

Thus, we have,

Lemma 17: The naive prefix-sums algorithm runs in $O(\frac{n \log n}{w} + \frac{nl \log n}{p})$ time units using p threads on the DMM and on the UMM with width w and latency l.

Clearly, from Theorem 3, the naive algorithm is not optimal.

X. OUR OPTIMAL PREFIX-SUMS ALGORITHM

This section shows an optimal prefix-sums algorithm running in $O(\frac{n \log n}{w} + \frac{nl}{p} + l \log n)$ time units. We use $m-1$ arrays $a_1, a_2, \dots a_{m-1}$ as work space. Each a_t $(1 \leq t \leq m-1)$ can store $2^t - 1$ numbers. Thus, the total size of the $m-1$ arrays is no more than $(2^1 - 1) + (2^2 - 1) + \cdots + (2^{m-1} - 1) = 2^m - m < n$. We assume that the input of n numbers are stored in array a_m of size n.

The algorithm has two stages. In the first stage, interval sums are stored in the $m-1$ arrays. The second stage uses interval sums in the $m-1$ arrays to compute the resulting prefix-sums. The details of the first stage is spelled out as follows.

[Compute the interval sums]

```
for $t \leftarrow m-1$ downto 1 do
  for $i \leftarrow 0$ to $2^t - 1$ do in parallel
    $a_t[i] \leftarrow a_{t+1}[2 \cdot i] + a_{t+1}[2 \cdot i + 1]$
```

Figure 14 illustrated how the interval sums are computed. When this program terminates, each $a_t[i]$ $(1 \leq t \leq m-1, 0 \leq i \leq 2^t - 2)$ stores $a_t[i \cdot \frac{n}{2^t}] + a_t[i \cdot \frac{n}{2^t} + 1] + \cdots + a_t[(i+1) \cdot \frac{n}{2^t} - 1]$.

In the second stage, the prefix-sums are computed by computing the sums of the interval sums as follows:

[Compute the sums of the interval sums]

```
for $t \leftarrow 1$ to $m-1$ do
  for $i \leftarrow 0$ to $2^t - 2$ do in parallel
    begin
      $a_{t+1}[2 \cdot i + 1] \leftarrow a_t[i]$
      $a_{t+1}[2 \cdot i + 2] \leftarrow a_{t+1}[2 \cdot i + 2] + a_t[i]$
    end
$a_m[n-1] \leftarrow a_m[n-2] + a_p[n-1]$
```

Figure 15 shows how the prefix-sums are computed. In the figure, "$a_{t+1}[2 \cdot i + 1] \leftarrow a_t[i]$" and "$a_{t+1}[2 \cdot i + 2] \leftarrow a_{t+1}[2 \cdot i + 2] + a_t[i]$" correspond to "copy" and "add", respectively.

When this algorithm terminates, each $a_p[i]$ $(0 \leq i \leq 2^t -)$ stores the prefix sum $a_p[0] + a_p[1] + \cdots + a_p[i]$. We assume that p threads are available and evaluate the computing time. The first stage involves the following memory access operations for each t $(1 \leq t \leq m-1)$:

- reading from $a_{t+1}[0], a_{t+1}[2], \dots, a_{t+1}[2^t - 2]$,

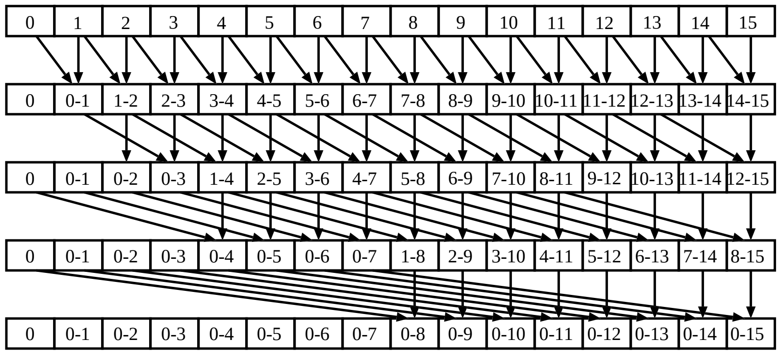


Figure 13. Illustrating the naive prefi x-sums algorithm for n numbers

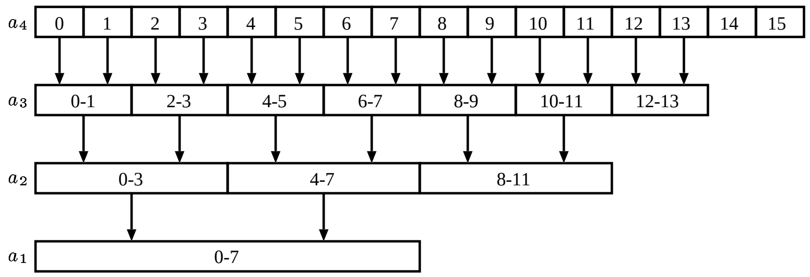


Figure 14. Illustrating the computation of interval sums in $m-1$ arrays.

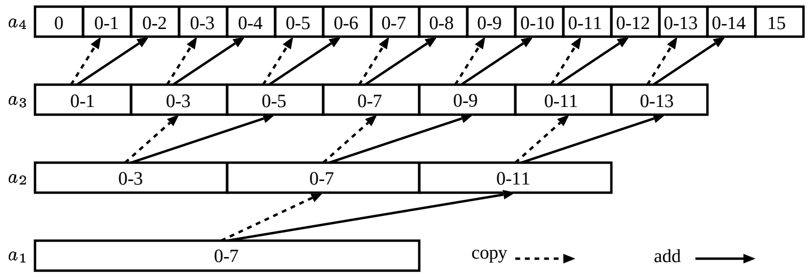


Figure 15. Illustrating the computation of interval sums in $m-1$ arrays.

- reading from $a_{t+1}[1], a_{t+1}[3], \ldots, a_{t+1}[2^t-1]$, and
- writing in $a_t[0], a_t[1], \ldots, a_t[2^t-1]$.

Since every two addresses is accessed, these four memory access operations are essentially contiguous access and they can be done in $O(\frac{2^t}{w}+\frac{2^t l}{p}+l)$ time units. Therefore, the total computing time of the firs stage is

$$\sum_{t=1}^{p-1} O(\frac{2^t}{w}+\frac{2^t l}{p}+l) \quad = \quad O(\frac{n}{w}+\frac{nl}{p}+l\log n).$$

The second stage consists of the following memory access operations for each t $(1 \le t \le m-1)$:

- reading from $a_t[0], a_t[1], \ldots, a_t[2^t-2]$,
- reading from $a_{t+1}[2], a_{t+1}[4], \ldots, a_{t+1}[2^{t+1}-2]$,
- writing in $a_{t+1}[1], a_{t+1}[3], \ldots, a_{t+1}[2^{t+1}-3]$, and
- writing in $a_{t+1}[2], a_{t+1}[4], \ldots, a_{t+1}[2^{t+1}-2]$.

Similarly, these operations can be done in $O(\frac{2^t}{w}+\frac{2^t l}{p}+l)$ time units. Hence, the total computing time of the second stage is also $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$. Thus, we have,

Theorem 18: The prefix-sums of n numbers can be computed in $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$ time units using p threads on the DMM and on the UMM with width w and latency l if work space of size n is available.

From Theorem 3, the lower bound of the computing time of the prefix-sums is $\Omega(\frac{n}{w}+\frac{nl}{p}+l\log n)$.

Suppose that n is very large and work space of size n is not available. We will show that, if work space no smaller than $\min(p\log p, wl\log(wl))$ is available, the prefix-sums can also be computed in $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$. Let k be an arbitrary number such that $p \leq k \leq n$. We partition the input a with n numbers into $\frac{n}{k}$ groups with k ($\geq p$) numbers each. Each t-th group ($0 \leq t \leq \frac{n}{k} - 1$) has k numbers $a[tk], a[tk+1], \ldots, a[(t+1)k-1]$. The prefix-sums of every group is computed using p threads in turn as follows.

[Sequential-parallel prefix-sums algorithm]

```
for $t \leftarrow 0$ to $\frac{n}{k} - 1$ do
  begin
    if($t \neq 0$) $a[tk] \leftarrow a[tk] + a[tk-1]$
    compute the prefix-sums of $k$ numbers $a[tk]$, $a[tk+1]$,
      $\ldots, a[(t+1)k-1]$
  end
```

It should be clear that this algorithm computes the prefix-sums correctly. The prefix-sums of k numbers can be computed in $O(\frac{k}{w} + \frac{kl}{p} + l \log k)$. The computation of the prefix-sums is repeated $\frac{n}{k}$ times, the total computing time is $O(\frac{k}{w} + \frac{kl}{p} + l \log k) \cdot \frac{n}{k} = O(\frac{n}{w} + \frac{nl}{p} + \frac{nl \log k}{k})$. Thus, we have,

Corollary 19: The prefix-sums of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + \frac{nl \log k}{k})$ time units using p threads on the DMM and on the UMM with width w and latency l if work space of size k is available.

If $k \geq p \log p$ then, $\frac{nl \log k}{k} \leq \frac{nl \log(p \log p)}{p \log p} < \frac{nl}{p}$. If $k \geq wl \log(wl)$ then $\frac{nl \log k}{k} \leq \frac{nl \log(wl \log(wl))}{wl \log(wl)} < \frac{n}{w}$. Thus, if $k \geq \min(p \log p, wl \log(wl))$ then the computing time is $O(\frac{n}{w} + \frac{nl}{p})$.

XI. Conclusion

In this paper, we have introduced two parallel memory machines, the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM). We first evaluated the computing time of the contiguous access and the stride access of the memory on the DMM and the UMM. We then presented an algorithm to transpose a 2-dimensional array on the DMM and the UMM. Finally, we have shown that any permutation of an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units on the DMM and the UMM with width w and latency l. Since the computing time just involves the bandwidth limitation $\frac{n}{w}$ and the latency limitation $\frac{nl}{p}$, the permutation algorithms are optimal. This paper also shows an optimal parallel prefix-sums algorithm that runs in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units.

Although the DMM and the UMM are simple, they capture the characteristic of the shared memory and the global memory of NVIDIA GPUs, Thus, these two parallel computing models are promising for developing algorithmic techniques for NVIDIA GPUs. As a future work, we plan to implement various parallel algorithms developed for the PRAM so far on the DMM and on the UMM. Also, NVIDIA GPUs have small shared memory and large global memory. Thus, it is also interesting to consider a hybrid memory machine such that threads are connected to a small memory of DMM and a large memory of UMM.

References

[1] A.V. Aho, J.D. Ullman, and J.E. Hopcroft, *Data Structures and Algorithms*, Addison Wesley, 1983.

[2] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, 1988.

[3] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*, Addison Wesley, 2003.

[4] M.J. Quinn, *Parallel Computing: Theory and Practice*, McGraw-Hill, 1994.

[5] W.W. Hwu, *GPU Computing Gems Emerald Edition*, Morgan Kaufmann, 2011.

[6] D. Man, K. Uda, Y. Ito, and K. Nakano, *A GPU Implementation of Computing Euclidean Distance Map with Efficient Memory Access*, in *Proc. of International Conference on Networking and Computing*, Dec., 2011, pp. 68–76.

[7] A. Uchida, Y. Ito, and K. Nakano, *Fast and Accurate Template Matching using Pixel Rearrangement on the GPU*, in *Proc. of International Conference on Networking and Computing*, Dec., 2011, pp. 153–159.

[8] Y. Ito, K. Ogawa, and K. Nakano, *Fast Ellipse Detection Algorithm using Hough Transform on the GPU*, in *Proc. of International Conference on Networking and Computing*, Dec., 2011, pp. 313–319.

[9] K. Nishida, Y. Ito, and K. Nakano, *Accelerating the Dynamic Programming for the Matrix Chain Product on the GPU*, in *Proc. of International Conference on Networking and Computing*, Dec., 2011, pp. 320–326.

[10] NVIDIA Corporation, *NVIDIA CUDA C programming guide version 4.0* (2011).

[11] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, *Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs*, International Journal of Networking and Computing 1 (2011), pp. 260–276.

[12] NVIDIA Corporation, *NVIDIA CUDA C best practice guide version 3.1* (2010).

[13] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, 1991.

[14] R.H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004.

[15] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. Eickenvon , *LogP: towards a realistic model of parallel computation*, in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993, pp. 1–12.

[16] R. Vaidyanathan and J.L. Trahan, *Dynamic Reconfi guration: Architectures and Algorithms*, Kluwer Academic/Plenum Publishers, 2004.

[17] M.J. Flynn, *Some computer organizations and their effectiveness*, IEEE Transactions on Computers C-21 (1972), pp. 948–960.

[18] G. Ruetsch and P. Micikevicius, *Optimizing matrix transpose in CUDA* (2009).

[19] N.K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, *A memory model for scientifi c algorithms on graphics processors*, in *Proc. of the ACM/IEEE Conference on Supercomputing*, 2006, pp. 6–6.

[20] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W. W. Hwumei , *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA*, in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.

[21] D.T. Wang, *Modern dram memory systems:performance analysis and a high performance, power-constrained dram scheduling algorithm*, Ph.D. thesis, University of Maryland, USA, 2005.

[22] Xilinx Inc., *Virtex-5 FPGA users guide* (2009).

[23] D.H. Lawrie, *Access and alignment of data in an array processor*, IEEE Trans. on Computers C-24 (1975), pp. 1145–1155.

[24] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, *The nyu ultracomputer – designing an MIMD shared memory parallel computer*, IEEE Trans. on Computers C-32 (1983), pp. 175 – 189.

[25] S.G. Akl, *Parallel Sorting Algorithms*, Academic Press, 1985.

[26] K.E. Batcher, *Sorting networks and their applications*, in *Proc. AFIPS Spring Joint Comput. Conf.*, Vol. 32, 1968, pp. 307–314.

[27] K. Nakano, *Optimal sorting algorithms on bus-connected processor arrays*, IEICE Trans. Fundamentals E76-A (1993), pp. 2008–2015.

[28] R.J. Wilson, *Introduction to Graph Theory, 3rd edition*, Longman, 1985.

[29] M. Harris, S. Sengupta, and J.D. Owens, *Chapter 39. parallel prefi x sum (scan) with CUDA*, in *GPU Gems 3*, Addison-Wesley, 2007.

[30] W.D. Hillis and G.L. Steele Jr., *Data parallel algorithms*, Commun. ACM 29 (1986), pp. 1170–1183, URL `http://doi.acm.org/10.1145/7902.7903`.

An Implementation of Conflict-Free Offline Permutation on the GPU

Akihiko Kasagi, Koji Nakano, and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The Discrete Memory Machine (DMM) is a theoretical parallel computing model that capture the essence of the shared memory access of GPUs. We need to avoid the bank conflicts for maximizing the bandwidth of the shared memory access. Offline permutation of an array is a task to copy of all elements in a into b along a given permutation. The main goal of this paper is to implement a conflict-free permutation algorithm on the DMM in a GPU. We have also implemented straightforward permutation algorithms on the GPU. The experimental results for 1024 float numbers on NVIDIA GeForce GTX-680 show that a straightforward permutation algorithm takes 246ns and 877ns for random permutation and bit-reversal permutation, respectively. Quite surpassingly, our conflict-free permutation algorithm runs in 165ns for random permutation and bit-reversal permutation each although it performs more memory access operations. It follows that our conflict-free permutation is 1.5 times faster for random permutation and 5.3 times faster for bit-reversal permutation.**

***Keywords*-memory machine models, data movement, bank conflict, shared memory, GPU, CUDA**

I. Introduction

The GPU (Graphical Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1], [2]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [3], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [4], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [3]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The global memory is implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [4], [5], [6]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access to the same memory banks in the same time, the access requests are processed sequentially. Hence, to maximize the memory access performance, threads of CUDA should access to distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus, CUDA threads should perform coalesced access when they access to the global memory.

In our previous paper [7], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs. The outline of the architectures of the DMM and the UMM are illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [8], which can execute fundamental operations in a time unit. We do not discuss the architecture of the sea of threads in this paper, but we can imagine that it consists of a set of multi-core processors which can execute many threads in parallel. Threads are executed in SIMD [9] fashion, and the processors run on the same program and work on the different data.

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. The DMM and the UMM capture the essence of the shared memory access and the global memory access of current GPUs. In our previous papers [7], [10], we have presented efficient algorithms including matrix transpose and computing the sum and the prefix-sums on the DMM and the UMM.

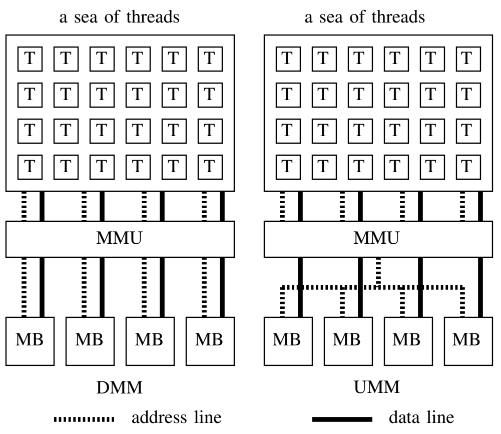


Figure 1. The architectures of the DMM and the UMM

Offline permutation is a task to move data along a permutation given beforehand. Since it has many applications, offline permutation is very important. For example, matrix transpose, which is one of the important permutations, is frequently used in matrix computation. It is known that the computation of FFT can be done by multistage network in which each stage involves permutation [11]. Sorting network such as bitonic sorting [12], [13] also involves permutation in each stage. Further, communication on processor networks such as hypercubes, meshes, and so on can be simulated by permutation on the shared memory. Thus, parallel algorithms on processor networks can be simulated on the shared memory machine by data permutations.

The main contribution of this paper is to present conflict-free offline permutation algorithm on the DMM and implement it to run on the shared memory in the GPU. Suppose that we have two arrays a and b of size n each. Let P be a permutation of $(0, 1, \ldots, n-1)$. In other words, $P(0), P(1), \ldots, P(n-1)$ take distinct integer values in the range $[0, n-1]$. Offline permutation along P is a task to copy $a[i]$ to $b[P(i)]$ for all i $(0 \leq i \leq n-1)$. The destination-designated (D-designated) algorithm just performs $b[P(i)] \leftarrow a[i]$ for all i. However, writing operation to array b may involve bank conflicts. Our idea is to use two permutations S and D which can be obtained from P. Using these two permutations our conflict-free permutation algorithm performs $b[D(i)] \leftarrow a[S(i)]$ for all i. Two permutations S and D are determined so that memory access operations to arrays b and a have no bank conflict. Two permutations S and D can be determined using a graph theoretic result about bipartite graph coloring. This idea is originally shown in our previous paper [7]. Our main contribution is to actually implement permutation algorithms including the destination-designated and our conflict-free permutation algorithms on the shared memory of the latest GPU, NVIDIA GeForce GTX-680.

The experimental results for 1024 float numbers on NVIDIA GeForce GTX-680 show that a straightforward permutation algorithm takes 246ns and 877ns for random permutation and bit-reversal permutation, respectively. Quite surpassingly, our conflict-free permutation algorithm runs in 165ns for random permutation and bit-reversal permutation each although it performs more memory access operations. It follows that our conflict-free permutation is 1.5 times faster for random permutation and 5.3 times faster for bit-reversal permutation.

This paper is organized as follows. First, we define the DMM formally in Section II. In Section III, we define off-line permutation and show straightforward algorithms. Section IV shows our conflict-free permutation algorithm and Section V describes the details of this implementation. In Section VI, experimental results using GeForce GTX-680 are shown. Section VII offers conclusions.

II. Discrete Memory Machine (DMM)

The main purpose of this section is to define the Discrete Memory Machine (DMM) introduced in our previous paper [7]. The reader should refer [7] for the details of the DMM.

Let $m[i]$ $(i \geq 0)$ denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j+w], m[j+2w], m[j+3w], \ldots\}$ $(0 \leq j \leq w-1)$ denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. Figure 2 illustrates memory banks of DMM for $w = 4$. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k+l-1$ time units to complete k access requests to a particular bank.

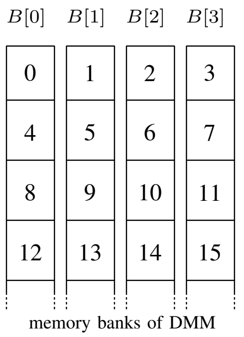


Figure 2. Memory banks for $w = 4$

Let $T(0), T(1), \ldots, T(p-1)$ be p threads. We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i+1) \cdot w - 1)\}$ $(0 \leq i \leq \frac{p}{w}-1)$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory in the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access and is skipped. When $W(i)$ is dispatched, w thread in $W(i)$ sends memory access requests, one request per thread, to the memory. We say that *the bank conflict* occurs if two or more threads in a warp access to the same bank. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread send a memory access request, it must wait l time units to send a new memory access request.

III. Offline Permutation and Conventional Algorithms

The main purpose of this section is to define offline permutation and show conventional algorithm for this task.

Suppose that we have two arrays a and b of size n each. Let P be a permutation of $(0, 1, \ldots, n-1)$. In other words, $P(0), P(1), \ldots, P(n-1)$ take distinct integer values in the range $[0, n-1]$. Offline permutation along P is a task to copy $a[i]$ to $b[P(i)]$ for all i $(0 \leq i \leq n-1)$.

Suppose that we have n threads for the task of offline permutation. We assume that $P(0), P(1), \ldots, P(n-1)$ are stored in an array p of size n, such that $p[i] = P(i)$ for all i $(0 \leq i \leq n-1)$. Let $T(i)$ $(0 \leq i \leq n-1)$ denote a thread. The following algorithm, destination designated permutation algorithm, performs the offline permutation along P.

[Destination-designated permutation algorithm]

```
for $i \leftarrow 0$ to $n-1$ do
    $T(i)$ performs $b[p[i]] \leftarrow a[i]$
```

Clearly, reading operations from arrays a and p have no bank conflict. However, writing operation in array b may have bank conflict.

For example, if $P = (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15)$ and $w = 4$, then the first warp $W(0)$ performs writing operation to $b[0], b[4], b[8]$, and $b[12]$ and they are in the same bank $B[0]$ (Figure 2). Hence, writing operations by $W(0)$ have bank conflict.

We can avoid writing bank conflict if we use the source-designated permutation Q. Let P^{-1} be the inverse of P, that is, $P^{-1}(P(i)) = i$ for all i $(0 \leq i \leq n-1)$. We assume that $P^{-1}(0), P^{-1}(1), \ldots, P^{-1}(n-1)$ are stored in an array q of size n, such that each $q[i]$ stores $P^{-1}(i)$. The following algorithm performs the offline permutation along P.

[Source-designated permutation algorithm]

```
for $i \leftarrow 0$ to $n-1$ do
    $T(i)$ performs $b[i] \leftarrow a[q[i]]$
```

Let us show that this algorithm performs the offline permutation along P correctly. The goal of the permutation along P is to satisfy $b[P(i)] = a[i]$ for all i $(0 \leq i \leq n-1)$. Hence, it is sufficient to satisfy $b[P^{-1}(P(i))] = a[Q(i)]$ for all i $(0 \leq i \leq n-1)$. From $P^{-1}(P(i)) = i$, it is also sufficient to satisfy $b[i] = a[P^{-1}(i)]$. Thus, the source-designated permutation algorithm performs the offline permutation along P correctly.

It should be clear that writing operations in b and reading operations from q have no bank conflict. However, reading operations from a may have bank conflict. For example, for P defined above, we have $P = P^{-1}$. Hence, reading operations has always bank conflicts.

We will show that, bank conflict-free permutation is possible if we use two arrays s and d determined from P appropriately. Let S and D be permutations over $(0, 1, \ldots, n-1)$. Suppose that $S^{-1}(D(i)) = P(i)$ for all i $(0 \leq i \leq n-1)$, where S^{-1} denotes the inverse of S. Let s and d be arrays of size n storing the values of S and D respectively. The following algorithm performs permutation along P:

[Conflict-free permutation algorithm]

```
for $i \leftarrow 0$ to $n-1$ do
    $T(i)$ performs $b[d[i]] \leftarrow a[s[i]]$
```

Let us see the correctness of the algorithm. When the algorithm terminates, $b[D(i)]$ is storing $a[S(i)]$ for all i $(0 \leq i \leq n-1)$. In other words, $b[S^{-1}(D(i))]$ is storing $a[S^{-1}(S(i))]$ for all i. Thus, $b[P(i)] = a[i]$ is satisfied and permutation along P is performed correctly.

Clearly, reading operations for array s and d are conflict-free. However, access to arrays a and b may have bank conflicts. If we define S and D appropriately, access to arrays s and d can be conflict-free. Let P be a permutation defined above. We define S and D as follows: $S = (0, 5, 10, 15, 1, 6, 11, 12, 2, 7, 8, 13, 3, 4, 9, 14)$ and $D = (0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11)$. For such S, we have $S^{-1} = (0, 4, 8, 12, 13, 1, 5, 9, 10, 14, 2, 6, 7, 11, 15, 3)$. Hence, $S^{-1} \cdot D = (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15) = P$. Thus, our conflict-free permutation algorithm using S and D are executed, permutation along P can be completed. Also, reading operations from a and writing operations from b are conflict-free. For example, warp $W(1)$ reads from $a[1], a[6], a[11], a[12]$ which are in banks $B[1], B[2], B[3], B[0]$, respectively. It also writes in $b[4], b[9], b[14], b[3]$ which are in banks $B[0], B[1], B[2], B[3]$, respectively.

Let us evaluate the computing time of our conflict-free permutation algorithm. We assume that n threads are used to permute an array of size n. Since we have $\frac{n}{w}$ warps of w threads each and reading from array s involve no bank conflict, reading from array s takes $O(\frac{n}{w} + l)$ time units.

Similarly, reading from array a and d, and writing in array b also take $O(\frac{n}{w}+l)$ time units. On the other hand, in the worst case, the destination-designated and source-designated permutation algorithms take $O(n+l)$ time units if memory access by a warp is performed to the same bank.

IV. Graph coloring based conflict-free permutation

This section is devoted to show how S and D are determined from P to guarantee that the conflict-free permutation using S and D involves no bank conflict. The same idea is used in our previous paper [7].

We use an important graph theoretic result [14], [15] as follows:

Theorem 1 (König): A regular bipartite graph with degree ρ is ρ-edge-colorable.

Figure 3 illustrates an example of a regular bipartite graph with degree 4 painted by 4 colors. Each edge is painted by 4 colors such that no node is connected to edges with the same color. In other words, no two edges with the same color share a node. The readers should refer to [14], [15] for the proof of Theorem 1.

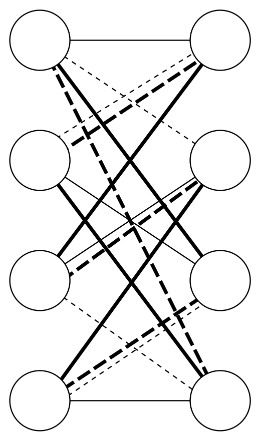


Figure 3. A regular bipartite graph with degree 4

Suppose that a permutation P of $(0,1,\ldots,n-1)$ is given. We draw a bipartite graph $G=(U,V,E)$ of P as follows:

- $U=\{B[0],B[1],B[2],\ldots,B[w-1]\}$ is a set of nodes each of which corresponds to a bank of a.
- $V=\{B[0],B[1],B[2],\ldots,B[w-1]\}$ is a set of nodes each of which corresponds to a bank of b.
- For each pair source $a[i]$ and destination $b[P(i)]$, E has a corresponding edge connecting $B[i \bmod w](\in U)$ and $B[P(i) \bmod w](\in V)$.

Clearly, an edge $(B[u],B[v])$ $(0\le u,v\le w-1)$ corresponds to a word of data to be copied from bank $B[u]$ of a to $B[v]$ of b. Also, $G=(U,V,E)$ is a regular bipartite graph with degree $\frac{n}{w}$. Hence, G is $\frac{n}{w}$-colorable from Theorem 1. Suppose that all of the n edges in E are painted by $\frac{n}{w}$ colors $0, 1, \ldots, \frac{n}{w}-1$. We determine value $c_{i,j}$ $(0\le i\le \frac{n}{w}-1, 0\le j\le w-1, 0\le c_{i,j}\le n-1)$ such that an edge $(B[c_{i,j} \bmod w], B[P(c_{i,j}) \bmod w])$ with color i corresponds to a pair of source $a[c_{i,j}]$ and destination $b[P(c_{i,j})]$. It should have no difficulty to confirm that, for each i,

- w banks $B[c_{i,0} \bmod w]$, $B[c_{i,1} \bmod w]$, $\ldots$, $B[c_{i,w-1} \bmod w]$ are distinct, and
- w banks $B[P(c_{i,0}) \bmod w]$, $B[P(c_{i,1}) \bmod w]$, $\ldots$, $B[P(c_{i,w-1}) \bmod w]$ are distinct.

Thus, we have the following important lemma:

Lemma 2: Let $c_{i,j}$ $(0\le i\le \frac{n}{w}-1, 0\le j\le w-1, 0\le c_{i,j}\le n-1)$ denote a source defined above. For each i, we have, (1) $a[c_{i,0}]$, $a[c_{i,1}]$, $\ldots$, $a[c_{i,w-1}]$ are in different banks, and (2) $b[P(c_{i,0})]$, $b[P(c_{i,1})]$, $\ldots$, $b[P(c_{i,w-1})]$ are in different banks.

We define permutation S and D using $c_{i,j}$ as follows:

$$\begin{aligned} S(i\cdot w+j) &= c_{i,j} \\ D(i\cdot w+j) &= P(c_{i,j}) \end{aligned}$$

Suppose that the conflict-free permutation algorithm using S and D above is executed. Since the copy operation is performed from $a[c_{i,j}]$ to $b[P(c_{i,j})]$, the permutation along P is completed correctly. Also, each warp $W(i)$ $(0\le i\le \frac{n}{w}-1)$ performs copy operation from $a[c_{i,0}], a[c_{i,1}], \ldots, a[c_{i,w-1}]$ to $b[P(c_{i,0})], b[P(c_{i,1})], \ldots, b[P(c_{i,w-1})]$. From Lemma 2, reading from a and writing in b by warp $W(i)$ are conflict-free.

V. Implementation of conflict-free permutation algorithm

The main purpose of this section is to show our implementation of conflict-free permutation algorithm to the GPU using CUDA.

A permutation P of $(0,1,\ldots,n-1)$ is given as an input. We first draw a bipartite graph $G=(U,V,E)$ of P shown in previous section and find an edge coloring. Recall that edges are painted by $\frac{n}{w}$ colors so that no two edge with the same color shares a node. Clearly, the edge coloring can be done by repeating a bipartite graph matching $\frac{n}{w}$ times. Also, it is known that a maximal bipartite graph matching, which is a subset of edges sharing no node, can be found in polynomial time.

For the reader's benefits, we briefly explain how a bipartite graph matching can be found. Let $G=(U,V,E)$ be a bipartite graph and M $(\subseteq E)$ is a matching. Note that M may not be a maximal. A path A of G is called an *augmenting path* if

- two terminals of A are not connected to M, and
- edges of M and $E-M$ appear alternatively in A.

Figure 4 shows examples of augmenting paths.

Clearly, the first and the last edges are in $E-M$. Also, in an augmenting path A, the number of edges of $E-M$

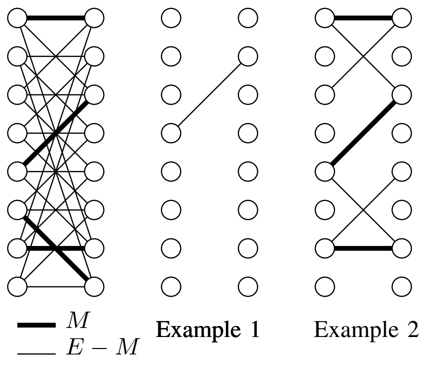


Figure 4. Examples of augmenting paths

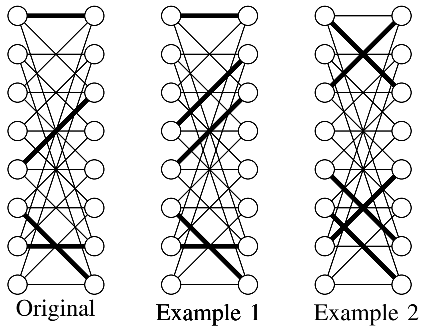


Figure 5. The resulting bipartite matchings after flipping operation

is exactly one larger than that of M. In other words, $|A \cap (E - M)| = |A \cap M| + 1$ holds.

Let us consider *the flipping operation* for an augmenting path as follows:

- $M \leftarrow M - (A \cap M)$, that is, remove edges in $A \cap M$ from M.
- $M \leftarrow M \cup (A \cap (E - M))$, that is add edges in $A \cap (E - M)$ to A.

The reader should refer to Figure 5 for illustrating the resulting bipartite matchings after the flipping operation. Clearly, the resulting M is a matching and the number of edges in M increases by one.

An augmenting path can be found in polynomial time if it exists. Pick a node connected to no edge in M. Construct a shortest path tree from the picked node such that, in all paths from the root (or the picked node) to the leaves, edges $E - M$ and M appears alternatively. If we can find a non-root node connected to no edge in M, then the path from the root to the non-root node is an augmenting path.

From these observation, we can find a maximal matching of a bipartite graph G as follows. Initially, let $M = \emptyset$. Find an augmenting path with respect to G and M and performs flipping operation. This task is repeated until we can find no augmenting path with respect to G and M. The resulting matching M is a maximal matching. If the graph is a regular bipartite graph, M is also a maximum matching. For graph coloring, we repeat finding the maximum matching. First, find the maximum matching M, paint edges in M with color 0, and remove edges in M from G. In this way, we can find a bipartite graph coloring in polynomial time.

Note that, we perform a bipartite graph coloring in offline. So, it is not necessary to find a bipartite graph coloring using a GPU. Actually, we have implemented a bipartite graph coloring to run on a convectional Linux PC.

We have implemented permutation algorithms using CUDA. Arrays a and b are defined as arrays of n float numbers in the shared memory of the GPU and arrays p, q, s, and d are defined arrays of n int numbers in the shared memory as follows:

```
__shared__ float a[n], b[n];
__shared__ int p[n], q[n], s[n], d[n];
```

Also, three permutation algorithms are implemented by CUDA device functions as follows:

[Destination-designated permutation algorithm]
```
__device__ d-designated(float *a, float *b, int *p){
  b[p[threadIdx.x]]=a[threadIdx.x];
}
```

[Source-designated permutation algorithm]
```
__device__ s-designated(float *a, float *b, int *q){
  b[threadIdx.x]=a[q[threadIdx.x]];
}
```

[Conflict-free permutation algorithm]
```
__device__ conflict-free(float *a, float *b, int *s, int *d){
  b[d[threadIdx.x]]=a[s[threadIdx.x]];
}
```

The above codes are executed by every thread with a unique ID represented by threadIdx.x such that threadIdx.x $= i$ for $T(i)$.

To reveal the overhead of permutation, we also use a simple copy CUDA device function as follows:

[Copy algorithm]
```
__device__ copy(float *a, float *b){
  b[threadIdx.x]=a[threadIdx.x];
}
```

In other words, the copy algorithm performs identical permutation such that $P(i) = i$ for all i.

Table I summarizes memory access operations performed by the algorithms. For example, the destination-designated permutation algorithm performs read operations for arrays a and p, and write operations for array b. Hence, it performs $2n + n = 3n$ memory access operations. Our conflict-free permutation algorithm performs $4n$ memory access operations. Thus, if each memory access operation have the

Table I
MEMORY ACCESS BY EACH ALGORITHM

Algorithms	a	b	p	q	s	d	read	write
Copy	r	w					n	n
D-designated	r	w	r				$2n$	n
S-designated	r	w		r			$2n$	n
Our conflict-free	r	w			r	r	$3n$	n

same access time, the conflict-free permutation algorithm is $\frac{4n}{3n} = \frac{4}{3}$ times slower than the D-designated and S-designated permutation algorithms. However, as we are going to show in the next section, our conflict-free permutation algorithm can be much faster than the destination-designated and source-designated permutation algorithms.

VI. EXPERIMENTAL RESULTS

This section is devoted to show the experimental results using GeForce GTX-680. To evaluate the performance of permutation algorithms We use several widely-used important permutations as follows:

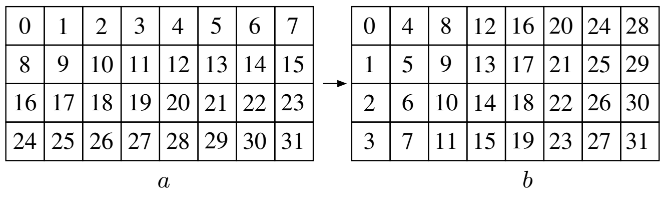
Identical: Permutation such that $P(i) = i$ for every i.

Random: One of all the possible $n!$ permutations is selected uniformly at random.

Transpose: Suppose that a and b are matrix with dimension $\frac{n}{w} \times w$. Transpose corresponds to the data movement such that a is read in row-major order and b is written in column-major order as illustrated in Figure 6. That is, $P(i \cdot w + j) = j \cdot \frac{n}{w} + i$ for every i and j $(0 \leq i \leq \frac{n}{w} - 1, 0 \leq j \leq w - 1)$.

Shuffle: Let $i_m i_{m-1} \cdots i_1$ be the binary representation of i. Shuffle permutation is defined by $P(i_m i_{m-1} \cdots i_1) = i_{m-1} \cdots i_1 i_m$. Shuffle permutation is widely used for shuffle exchanging in sorting networks [12], [13].

Bit-reversal: Shuffle permutation is defined by $P(i_m i_{m-1} \cdots i_1) = i_1 \cdots i_{m-1} i_m$. Bit-reversal is used for data reordering in the FFT algorithms [11].



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

a

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

b

Figure 6. Transpose permutation

We have evaluated the performance three permutation algorithms, the destination-designated permutation algorithm (D-designated), the source-designated permutation algorithm (S-designated), and our conflict-free permutation algorithm (conflict-free). Also, to estimate the overhead of these three permutation algorithms, we have evaluated the performance of the copy algorithm. Since any permutation algorithm cannot be faster than the copy algorithm, its computing time is the lower bound of that for any permutation algorithm.

The performance has been evaluated for $n = 1024$ using NVIDIA GeForce GTX-680. A CUDA kernel with a single block of 1024 threads is called from the host. The 1024 threads executes one of the four device functions, D-designated, S-designated, conflict-free, and copy. Note that the number w of memory banks is 32. For Transpose and Bit-reversal permutations, wiring operation by the D-designated and S-designated algorithms involves many bank conflicts in the sense that most of threads in a warp writing in the same bank. Table II shows the executing time for an array of size $n = 1024$. Since the executing time of each algorithm is too short to measure, each algorithm has been executed for each permutation 1 million times and took its average. Table II also shows the ratio of the execution time with respect to that of the simple copy. Note that, any permutation algorithm cannot be faster than the simple copy. Thus, the ratio in the table clarifies the overhead of each permutation algorithm.

According to the table, for Identical and Shuffle permutations that rarely involve bank conflicts, the D-designated and S-designated algorithms run faster than our conflict-free algorithm because extra memory access operations are necessary for the conflict-free algorithm as shown in Table I. The executing time of the S-designated algorithm for Shuffle permutation is longer than that of the D-designated algorithm since the number of bank conflict increases. On the other hand, for Random, Transpose, and Bit-reversal permutations, whose number of bank conflicts is not small, our conflict-free permutation algorithm runs faster than the others though extra memory access is necessary since all the memory access can avoid bank conflict. For Transpose and Bit-reversal permutations, our conflict-free permutation algorithm attains a speedup factor of more than 5 over the others. That is, our conflict-free permutation algorithm is efficient for permutations that frequently involve bank conflict. Also, the execution time of our conflict-free is almost constant for all permutations.

In applications using the GPU, the permutation algorithm is often executed for multiple arrays in parallel. Therefore, we have also evaluated the performance of the permutation algorithms if they executed for multiple arrays of size 1024. More specifically, a kernel call of CUDA generates multiple blocks, each of which executes a permutation algorithm for an array of size 1024 in parallel. Figure 7 shows the executing time for multiple arrays of size 1024. From the figure, when the number of arrays is less than or equal to 8, the executing time is almost the same. In other words, at most 8 CUDA blocks executing a permutation algorithm run in the same time. Further, if a kernel call generates $8k$ $(k \geq 1)$ CUDA blocks, the execution time is almost proportional to k. These facts make sense because GTX-680

Table II
THE EXECUTING TIME AND THE RATIO WITH RESPECT TO THE COPY FOR AN ARRAY OF SIZE $n = 1024$.

Permutations	Algorithms			Copy
	D-designated	S-designated	Conflict-free	
Identical	135.366ns/1.315	123.637ns/1.201	165.180ns/1.605	102.937/1.000
Random	246.918ns/2.390	265.786ns/2.582	164.544ns/1.598	
Transpose	876.329ns/8.513	891.006ns/8.656	164.851ns/1.601	
Shuffle	136.073ns/1.322	183.192ns/1.780	164.773ns/1.601	
Bit-reversal	876.891ns/8.519	891.390ns/8.660	164.764ns/1.601	

has a GPU with 8 multicore processors work in parallel.

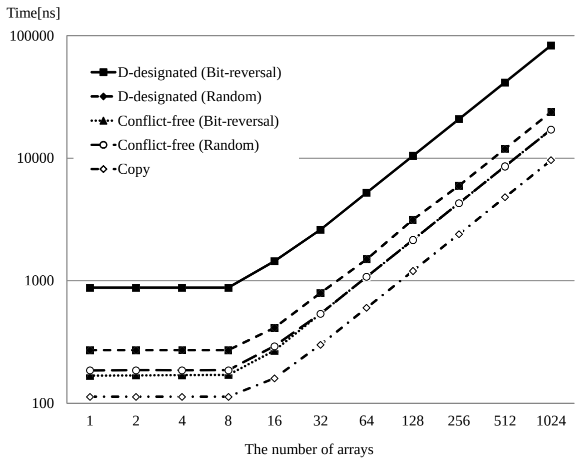


Figure 7. The executing time to permute arrays of 1024 elements.

VII. CONCLUSION

The main contribution of this paper is to implement several permutation algorithm including the straightforward and our conflict-free permutation algorithm on the shared memory of NVIDIA GeForce GTX-680 The experimental results for 1024 float numbers on NVIDIA GeForce GTX-680 show that a straightforward permutation algorithm takes 246ns for the random permutation and 877ns for the worst permutation that involves many bank conflicts. Our conflict-free permutation algorithm runs in approximately 165ns for any permutation including the random permutation and the worst permutation, although it performs more memory accesses.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 320–326.

[3] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 4.1," 2012.

[4] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, pp. 260–276, July 2011.

[5] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[6] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 4.1," 2012.

[7] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 788–797.

[8] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[9] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, 1972.

[10] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," in *to appear in Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, 2012.

[11] J. D. Scott Parker, "Notes on shuffle/exchange-type switching networks," *IEEE Trans. on Computers*, vol. C-29, no. 3, pp. 213 – 222, March 1980.

[12] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[13] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.

[14] K. Nakano, "Optimal sorting algorithms on bus-connected processor arrays," *IEICE Trans. Fundamentals*, vol. E76-A, no. 11, pp. 2008–2015, Nov. 1993.

[15] R. J. Wilson, *Introduction to Graph Theory, 3rd edition*. Longman, 1985.

マルチキャストツリーを構成する自己安定アルゴリズムに関する研究

八木渉 †　片山喜章 †　和田幸一 ‡
† 名古屋工業大学 大学院工学研究科 ‡ 法政大学 応用情報工学科

平成 24 年 9 月 3 日

1　はじめに

マルチキャストとはネットワーク上のあらかじめ決められた端末の集合に同一メッセージを送信する手法で，ネットワーク上の端末間の通信方式である．アドホックネットワークとはインフラストラクチャ(通信基盤設備) を必要とせず，無線移動端末のみで構築されるネットワークである．インフラストラクチャの構築の困難な場所，例えば災害などによってインフラが破壊された場所でもネットワークサービスを受けられるといった利点がある．特徴として，端末の出現と消滅，さらに端末が移動することが頻繁に起こる変化の頻度が高い動的ネットワークであることが挙げられる．

アドホックネットワークにおけるマルチキャストのプロトコルが提案されており，一般にマルチキャストプロトコルにはマルチキャストツリーを利用したものがある [1]．メッセージを受信する宛先の集合をマルチキャストグループといい，マルチキャストツリーとは，マルチキャストグループのプロセスを全て含み，葉がグループのメンバである木構造のネットワークである．しかし，アドホックネットワークはトポロジの変化が頻繁に起こる動的ネットワークであるため，構成したマルチキャストツリーが壊れやすく，維持が困難である．本研究では，この問題を解決するため，自己安定マルチキャストツリー生成プロトコルを提案する．

自己安定プロトコルとは分散システムが任意の状況から実行を開始 (このときの状況を初期状況という) しても，有限時間内に目的とする状況 (正当な状況) に到達でき，プロセスやリンクの生成や消滅などネットワークのトポロジが変化した状況にも対応でき，また，リンクやプロセスの一時故障にも耐性のある分散プロトコルである [1][2][3]．一時故障に耐性のあるプロトコルということは，長期に渡ってネットワークの状況を安定に保ち，一時故障が起きても自律的に復旧させることが求められる分散システムを動作させる場合に適している．また，ネットワークのトポロジが変化しても，変化後の状況を初期状況と見なせば，再び正当な状況に到達する．

マルチキャストに関する既存の研究の紹介と本稿の提案プロトコルの概要を説明する．

Sandeep K.S. Gupta ら [1] は，アドホックネットワーク上におけるマルチキャストグループに対する 2 つの分散アルゴリズムを提案した．マルチキャストを実現するために与えられたグラフの根から全てのノードへのパスの長さが最小である木 (最短経路木) を構成する戦略と，木を構成する辺の重みの総和が，そのグラフで考えうるいかなる全域木の辺の重みの和以下であるような全域木 (最小重み全域木) を構成する戦略をとっている．ネットワーク中のすべてのマルチキャストグループメンバのみからなるコスト最小の木を生成することがマルチキャストの効率の良い実装といえるが，これらの木は現実的な時間で生成することは不可能であることが知られている．そのため，ヒューリスティックアルゴリズムを用いて，効率を解析しながら提案している．

Sayaka KAMEI ら [3] は，マルチキャストグループのメンバすべてを網羅し，かつ，辺の重みの最小の木 (スタイナー木問題) を用いることが分散システム上で最適なマルチキャストの経路探索問題であることに注目して，それを自己安定アルゴリズムとして提案している．ところが，スタイナー木問題は今のところ効率の良いアルゴリズムが知られておらず，最小重み生成木を構成し，必要でない辺や節点を刈り取るという戦略で近似アルゴリズムを提案している．

本稿の提案プロトコルでは，ネットワークでマルチキャストグループに含まれるノードら全員に対して，送信者からのメッセージを受信できるようなルーティングをする自己安定プロトコルを提案する．提案するプロトコルを プロトコル $\mathcal{MRT}$ と呼び，プロトコル $\mathcal{MRT}$ は，マルチキャストのためのルーティング経路であるマルチキャストツリーを生成する．マルチキャストツリーを生成する手段として，根プロセスが所有する情報をすべてのプロセスに伝え (ブロードキャスト)，それに対する応答を根プロセスが収集する (フィードバック) という方法 [2] を用い，ブロード

キャストしながらネットワーク上に全域木を生成し，フィードバックしながら生成した全域木から不要な木の枝を刈り取り，マルチキャストツリーを生成するという戦略をとっている．全域木が生成される際に，根を除く親が決まっていない各プロセスは一番最初に情報を伝えたプロセスを親にするため，通信遅延が小さい通信経路が生成されることが期待される．生成されたマルチキャストツリーはネットワークトポロジの変化あるいはプロセスの故障が起こらない限り固定され，維持をするプロトコルである．

本稿の構成は以下の通りである．第 2 章では，ネットワークやプロトコルなどのモデルおよび MRT 生成問題について述べる．第 3 章では，提案プロトコル $\mathcal{MRT}$ について説明する．第 4 章ではその正当性の証明をおこなう．第 5 章では，まとめと今後の課題について述べる．

2　システムモデルと問題の定義

2.1　ネットワークとプロセス

本論文では，有限個のプロセスが通信リンクで接続された任意の形状のネットワーク N を扱う．ネットワークは非同期であり，根のプロセスを 1 つ持つ．

N のプロセスの集合 $\mathcal{P}$ をネットワークのプロセスの総数を n として，$\mathcal{P} = \{p_0, p_1, ..., p_{n-1}\}$ と表す．ネットワーク N 中の通信リンクの集合を $\mathcal{L}$ とする．このとき，ネットワーク N は 2 項組 $N = (\mathcal{P}, \mathcal{L})$ で定義される．各プロセスはすべて相異なる識別子を持つ．簡単のためプロセス p_i と p_i の識別子を区別せず，p_i の識別子を p_i と表す．$(p_i, p_j) \in \mathcal{L}(0 \leq i, j \leq n-1)$ のとき，p_i, p_j 間に全二重リンクが存在する．各プロセスは自分の識別子と自分に接続するリンクを知っているとする．プロセス間の通信は，状態通信モデルを仮定する．つまり，隣接するプロセスは互いの内部状態（識別子，変数の値）を直接知ることができる．書き込みは各プロセスが自分の持つ変数に対してのみ行うことができる．

ネットワークは，プロセスを頂点，リンクを辺とする無向グラフとみなせるので，グラフに対する用語や記法をネットワークに対しても用いる．

2.2　プロトコルの記述形式

本論文では，プロトコルを以下の形式で記述する．

各プロセスを状態遷移機械としてモデル化する．状態を変数で表し，プロトコルの動作にあたる状態遷移をガード付きアクション (以下，アクションという) で表す．各プロセスの各アクションはラベル付けされて

入力
…
諸定義
…
動作
$< label >::< guard > \rightarrow < statement >$
$< label >::< guard > \rightarrow < statement >$
…

図 1: プロトコル名

いる．プロセス p の各アクションのガード $< guard >$ は p と p の隣接プロセスの変数からなる論理式で表される．プロセス p はガードが真の場合のみ命令文 $< statement >$ を実行する．

2.3　実行

ガードが評価され，命令文が実行されるまでを 1 原子動作 (1 ステップ) とする．ガードが真である命令文の実行をアクションの実行とよぶ．

各プロセス p_i の状態集合を S_i で表す．分散システムのとり得る状況の集合は $\mathcal{C} = S_0 \times S_1 \times S_2 \times \cdots \times S_{n-1}$ である．分散システムの各状況 γ はプロセス状態の n 項組 $(s_0, s_1, s_2, \cdots, s_{n-1})$ である．ここで，$s_i \in S_i (0 \leq i \leq n-1)$ である．

定義 2.1. *(スケジュールと実行)* 分散プロトコル $\mathcal{D}$ を全ての状況の集合 $\mathcal{C}$ 上の 2 項関係 $\mapsto$ とみなせる．$\gamma \in \mathcal{C}$ を任意の状況とし，Q をプロセスの集合 $\mathcal{P}$ の任意の部分集合とする．Q に属するすべてのプロセスが同時に 1 ステップを行うことにより状況が γ から γ' になるとき $\gamma \mapsto \gamma'$ と表す．プロセスの空でない部分集合の無限系列をスケジュールとよび，$\mathcal{Q} = Q_1, Q_2, \cdots$ と表す．このとき，状況の無限系列が $e = (\gamma_0, \gamma_1, \cdots, \gamma_i, \gamma_{i+1} \cdots)$ が $\gamma_i \mapsto \gamma_{i+1} (i \geq 0)$ を満たすとき，e を初期状況 γ_0，スケジュール $\mathcal{Q}$ に対する実行とよぶ．　□

定義 2.2. *(動作可能性)* 状況 γ におけるプロセス p の少なくとも 1 つのガードが真であるなら,「プロセス p が動作可能 (*enabled*)」という.また，プロセス p の動作 A のガードが真となるとき「動作 A はプロセス p において動作可能 (*enabled*)」という．　□

分散システム $\mathcal{S}$ の状況 $\alpha \in \mathcal{C}$ から始まるプロトコル $\mathcal{D}$ ではスケジュールの違いにより様々な実行が生じるが，起こりうる全ての実行を ε_α と表す.$\mathcal{S}$ での $\mathcal{P}$ の起こりうる実行のすべての集合を $\varepsilon (\varepsilon = \bigcup_{\alpha \in \mathcal{C}} \varepsilon_\alpha)$ と表す.$\beta = \gamma_i (i \geq 0)$ であるような実行 $e = (\gamma_0, \gamma_1, \cdots$

$, \gamma_i, \gamma_{i+1} \cdots) \in \varepsilon_\alpha(\alpha = \gamma_0)$ が存在するなら，状況αからβへ到達可能であるという．

定義 2.3. *(弱公平な実行)* プロセス p に対して，ある状況 γ_i 以降常にガードが真でありながら，γ_i 以降命令文が実行されないようなアクションは存在しない実行を弱公平な実行という． □

スケジュールによって選ばれたプロセスは，1 原子動作のみを行うことができる．スケジュールが選び出すプロセス数と原子動作の違いにより，いくつかのモデルが考えられる．D デーモンは任意のスケジュールを考えるモデルである．つまり，同時に複数のプロセッサが動作することを許すネットワークモデルである．本稿では，弱公平な実行のみを対象とし，モデルは D デーモンを扱う．

2.4 自己安定プロトコル

ここでは，attractor を用いて，自己安定プロトコルを定義する．ただし，以下では，任意の集合 χに対し, $x \vdash X$ は要素 $x \in \chi$が集合χ上で定義された述語 X を満たすことを意味する．特別な述語 *true* を以下のように定義する．

- $\forall x \in \chi, x \vdash true$

定義 2.4. *(Attractor)* X と Y を分散システムのとり得る状況の集合 $\mathcal{C}$ に対して定義された述語とする.以下の条件を満たすとき Y は X に対する*attractor*であるといい，$X \rhd Y$ と表す．

- $\forall \alpha \vdash X : \forall e = (\gamma_0, \gamma_1, \cdots) \in \varepsilon_\alpha :: \exists i \geqq 0, \gamma_i \vdash Y$

$X \rhd Y$ は述語 X を満たす任意の状況から始まる全ての実行に対して，分散システムが述語 Y を満たすような状況に到達することを意味する． □

定義 2.5. *(自己安定プロトコル)* 分散プロトコルを$\mathcal{D}$とし, $\mathcal{D}$ の全ての実行の集合をεとする.また, $\mathcal{SP}_\mathcal{D}$を実行の集合εに対して定義される述語とする.全ての状況の集合 $\mathcal{C}$ に対して定義される述語 $\mathcal{L}_\mathcal{D}$が存在し，以下の条件を満たすとき，分散プロトコル $\mathcal{D}$ は $\mathcal{SP}_\mathcal{D}$に対して，自己安定であるという．

1. $\forall \alpha \vdash \mathcal{L}_\mathcal{D} : \forall e \in \varepsilon_\alpha :: e \vdash \mathcal{SP}_\mathcal{D}$(正当性).
2. $true \rhd \mathcal{L}_\mathcal{D}$(収束性).

述語 $\mathcal{SP}_\mathcal{D}$は，分散システムに要求される動作を規定するものである.$\mathcal{SP}_\mathcal{D}$をプロトコル要求とよび, $\mathcal{L}_\mathcal{D}$を満たす状況を安全な状況とよぶ． □

すなわち，任意の状況から開始 (このときの状況を初期状況という) しても，プロトコル $\mathcal{D}$ は有限時間以内に安全な状況 $\mathcal{L}_\mathcal{D}$ に到達し，$\mathcal{L}_\mathcal{D}$ を満たす状況に至ると，その状況から始まる任意の実行が $\mathcal{SP}_\mathcal{D}$ を満たすようになる．

2.5 時間複雑度

各プロセスの計算速度や通信速度が異なるようなモデルでの時間複雑度の尺度としてラウンドがある．直感的には，すべてのプロセスが少なくとも 1 回，一原子動作を行ったときを 1 ラウンドと呼ぶ．1 ラウンドの間に原子動作を何回も行うプロセスがあってもよい．正確には,

- 与えられた実行 $e(\in \varepsilon)$ に対して,e の最初のラウンド e' とは，初期状況から継続して *enabled* であるすべてのプロセスの原子動作がぞれぞれ 1 つ含まれている e のプレフィックスである．
- e'' を e のサフィックス $(e = e'e'')$ とすると e の 2 ラウンドとは e'' の最初のラウンドである．
- 以降のラウンドも同様.

と定義する．

2.6 マルチキャストツリー生成問題

根ノード r を 1 つ持つグラフ $G = (V, E)$ を考える．V はノードの集合，E はノード間の無向辺の集合とする．各ノードに，自分が根であるか，隣接ノードの集合，自分がマルチキャストグループのメンバかの入力が与えられたときに，以下を満たすような木を G 上に生成する問題をマルチキャストツリー生成問題といい，その木をマルチキャストツリーという．

定義 2.6. *(マルチキャストツリー生成問題)* G とマルチキャストグループと呼ぶノードの集合 $Z(\subseteq V)$ と根ノード $r(\in V)$ が与えられるとき，以下の条件を満たす木をマルチキャストツリーといい，この木を生成する問題をマルチキャストツリー生成問題という．

- rを根とする G 上の木 $M = (V', E')(V' \subseteq V, E' \subseteq E)$ であり，かつ $Z \subseteq V'$および M の葉を l とすると $\forall l \in Z$ を満たす．

□

3 提案アルゴリズム

本論文で提案するプロトコル (図 2) はフィードバック付ブロードキャスト (PIF)[2] を利用して実現する．PIF とは，根プロセスが所有する情報をすべてのプロ

セスに伝え (ブロードキャスト)，それに対する応答を根プロセスが収集する (フィードバック) ことである．提案プロトコルでは，ブロードキャストしながらネットワーク上に全域木を生成し，フィードバックしながら，生成した全域木からマルチキャストツリーを生成するという戦略をとった．

3.1　概略

プロトコル $\mathcal{MRT}$ の概略を説明する．まずは，ネットワーク全体から見たときのプロセスの動作を説明する．

1. 根プロセス r がブロードキャストのメッセージをフラッディング.
2. ブロードキャストのメッセージを受信したプロセスは送信プロセスを親にする．
3. ブロードキャストメッセージがネットワーク全体に伝わると r を根とする全域木が生成される．
4. 葉のプロセスからフィードバックメッセージが返される．その際，マルチキャストツリーに必要でない辺が刈られる．
5. 隣接プロセス全てから根にフィードバックメッセージが返されたときネットワーク全体には r を根とするマルチキャストツリーが生成.

次に，各プロセスに注目した場合の動作の様子を説明する．

1. 根プロセス r が自分の隣接している全プロセスにメッセージ m を送信する．
2. m を受け取ったプロセスは r を親にして，隣接している他の全プロセスにメッセージ m を送信する．
3. m を受け取ったプロセスは最初に受信した m の送信元プロセスを親にして，隣接している全プロセスにメッセージ m を送信する．
4. 全プロセスが m を受け取るまで 3．を繰り返す (4．が終了すると r を根とする全域木が生成されている).
5. 葉プロセスは親のプロセス p に自分がマルチキャストグループに含まれるならば，"マルチキャストツリーのメンバである"を含まれないならば，"マルチキャストツリーのメンバでない"を付与したメッセージ f を返信する．
6. p が全ての子プロセスから f を受け取ったとき，"マルチキャストツリーのメンバである"の付与されたメッセージを受け取る，あるいは p がマルチキャストグループのメンバに含まれるならば，"マルチキャストツリーのメンバである"を付与したメッセージ f を親に送信する．
7. r の隣接している全プロセスが r へ送信を完了するまで 6．を繰り返す (7．が終了すると r を根とするマルチキャストツリーが生成されている).

3.2　プロトコル $\mathcal{MRT}$

プロトコル $\mathcal{MRT}$ の各プロセスが扱う定数，変数を示す．

- $N_p \subseteq V$: p の隣接ノードの集合を表す定数．ノードは任意の順序 $\succ_p$ で配置されるとする．
- $R_p \in \{0, 1\}$: p が根ノードであるかを表す定数 . $p = r$ のとき 1, $p \neq r$ のとき 0 である．
- $M_p \in \{0, 1\}$: p がマルチキャストグループのメンバかを表す定数．マルチキャストグループのメンバのとき 1 であり，マルチキャストグループのメンバでないとき 0 である．
- $Par_p \in (\{0\} \cup V)$: 親ポインタ. p が親としているノードを指す. 0 のときは何も指していない状態 .
- $Pif_p \in \{C, B, F\}$: ノードの状態を表し，次の 3 つの値をとる．
  - C : p は隣接ノードからメッセージを受け取る準備ができている状態.
  - B : p は隣接ノードからメッセージを受け取り, $Par(p)$ を除く隣接へメッセージをブロードキャストした状態.
  - F : p は全ての隣接ノードからフィードバックを受け取り, p もまたフィードバックメッセージを隣接の親に送信した状態.
- $T_p \in \{0, 1\}$: 1 のとき p はマルチキャストツリーのメンバである.0 はマルチキャストツリーのメンバでない．
- $L_p \in \mathbf{N}$(正整数) : 根から p へのブロードキャストメッセージが送られてきたパスの距離.

4　正当性の証明

プロトコル $\mathcal{MRT}$ の正当性の証明を行う.

定義 4.1. *(ノーマルノード，アブノーマルノード，C0 ノード，IAb ノード)*
$p = r$ かつ $rNormal(p)$ を満たすとき，あるいは $p \neq r$ かつ $Normal(p)$ を満たすとき, p をノーマルノードと呼ぶ.
また，ノーマルノード以外のノードをアブノーマルノードと呼ぶ.
アブノーマルノードのなかで, $Pif_p = C$ かつ $Par_p =$

入力

N_p, R_p, M_p

- $R_p = 1$ のとき

諸定義

$rNormal(p) \equiv L_p = 0 \wedge T_p = 1 \wedge Par_p = 0;$

$Broadcast(p) \equiv (Pif_p = C) \wedge (\forall q \in N_p :: Pif_q = C);$

$Feedback(p) \equiv (Pif_p = B) \wedge (\forall q \in N_p :: Pif_q = F);$

$Cleaning(p) \equiv (Pif_p = F) \wedge (\forall q \in N_p :: Pif_q = C);$

$rAbnl(p) \equiv \neg rNormal(p)$

動作

$rB - action :: Broadcast(p) \rightarrow Pif_p := B;$

$rF - action :: Feedback(p) \rightarrow Pif_p := F;$

$rC - action :: Cleaning(p) \rightarrow Pif_p := C;$

$rReset :: rAbnl(p) \rightarrow L_p := 0; T_p := 1; Par_p := 0;$

- $R_p = 0$ のとき

諸定義

$Potential_p = \{q | q \in N_p :: (Pif_q = B) \wedge (Par_q \neq p)\};$

$Rim(p) \equiv (\forall q \in N_p :: (Pif_q \neq C) \Rightarrow (Par_q \neq p));$

$Normal(p) \equiv$

$(Par_p \in N_p) \wedge ((L_p = L_{Par_p} + 1) \wedge ((Pif_p = Pif_{Par_p}) \vee ((Pif_p = C \wedge Pif_{Par_p} = F) \vee (Pif_p = F \wedge Pif_{Par_p} = B)))) \vee ((Pif_p = C \wedge Pif_{Par_p} = B)))$

$Broadcast1(p) \equiv (Pif_p = C) \wedge (Potential_p \neq \phi) \wedge Rim(p) \wedge (Par_p = 0);$

$Broadcast2(p) \equiv (Pif_p = C) \wedge (Pif_{Par_p} = B) \wedge Rim(p);$

$Feedback(p) \equiv (Pif_p = B) \wedge Normal(p) \wedge (\forall q \in N_p :: (Pif_q \neq C \vee Par_q \neq 0) \wedge (Par_q = p \Rightarrow Pif_q = F));$

$Cleaning(p) \equiv (Pif_p = F) \wedge Normal(p) \wedge Rim(p) \wedge (\forall q \in N_p :: (Pif_q \neq B));$

$Abnl(p) \equiv \neg Normal(p);$

$Tree(p) \equiv M_p = 1 \vee (\exists q \in N_p :: Par_q = p \wedge T_q = 1);$

動作

$B - action1 :: Broadcast1(p) \rightarrow Pif_p := B; Par_p := min_{\prec_p}(Potential_p); L_p := L_{Par_p} + 1;$

$B - action2 :: Broadcast2(p) \rightarrow Pif_p := B; L_p = L_{Par_p} + 1;$

$F - action1 :: Feedback(p) \wedge Tree(p) \rightarrow Pif_p := F; T_p := 1;$

$F - action2 :: Feedback(p) \wedge \neg Tree(p) \rightarrow Pif_p := F; T_p := 0;$

$C - action :: Cleaning(p) \rightarrow Pif_p := C;$

$Reset :: Abnl(p) \wedge (Pif_p \neq C \vee Par_p \neq 0) \rightarrow Pif_p := C; Par_p := 0;$

図 2: プロセス p 上のプロトコル $\mathcal{MRT}$

0 を満たすものを $C0$ ノードと呼ぶ．アブノーマルノードの中で $Pif_p \neq C$ または $Par_p \neq 0$ を満たすものを $IAb(Inapposite\,Abnormal)$ ノードと呼ぶ． □

定義 4.2. *(ParentPath)*
任意のノード p に対して, $ParentPath(p)$ と呼ばれる一意のパス $p = p_0, p_1, p_2 \cdots, p_k$が存在するための必要十分条件は以下の条件を満たすときである．

条件 *1.* $\forall i, 0 \leq i \leq k-1, Par_{p_i} = p_{i+1}$
条件 *2.* $\forall i, 0 \leq i \leq k-1, Normal(p_i)$
条件 *3.* $p_k = r$ または p_kがアブノーマルノード

□

定義 4.3. *(Subtree)*
$p = r \vee \neg Normal(p)$ である任意のノード p に対して，ノードの集合 $Subtree(p)$ を次のように定義する．
任意のノード q に対して $q \in Subtree(p)$ であるための必要十分条件は p が $ParentPath(p)$ の端点であること． □

定義 4.4. *(BSubtree)*
$(Pif_p = B) \wedge (p = r \vee \neg Normal(p))$ である任意のノード p に対して，ノードの集合 $BSubtree(p)$ を次のように定義する．
任意のノード q に対して $q \in BSubtree(p)$ であるための必要十分条件は $q \in Subtree(p)$ かつ $Pif_q = B$ であること． □

定義 4.5. *(ソース)*
ノード p が $Subtree(q)(resp.BSubtree(q))$ のソースと呼ばれるための必要十分条件は以下の条件を満たすときである．

条件 *1.* $p \in Subtree(q)(resp.BSubtree(q))$
条件 *2.* $\forall q' \in Subtree(q)(resp.BSubtree(q)), p \neq Par_{q'}$

□

定義 4.6. *(表記について)*
X_p^iはグローバル状況γ_iにおけるノード p の X の値を表す.X はある変数や集合を表す． □

定義 4.7. *(UnVisited)*
$UnVisited$ と呼ばれる集合を定義する．
ノード p に対して $p \in UnVisited$ であるための必要十分条件は次の以下の条件を満たすときである．

条件 *1.* $Pif_p = C$
条件 *2.* ($N_p = q$ として, ($Par_p = q$ または $Par_p = 0$) かつ ($q \in UnVisited$ または $Pif_q = B$) をみたす q が存在) または $Pif_r = C$

□

定義 4.8. *(Visited)*
$Visited$ と呼ばれる集合を定義する．
ノード p に対して $p \in Visited$ であるための必要十分条件は次の以下の条件を満たすときである．

条件 *1.* $Pif_p = C$
条件 *2.* ($N_p = q$ として, (Par_qかつ ($Pif_q = F$ または $q \in Visited$)) または ($Par_p = 0$ かつ $q \notin UnVisited$ かつ $Pif_q \neq B$) をみたす q が存在) または $Pif_r = C$

□

補題 4.1. ネットワークのノード数を n とする．任意のネットワークから多くても n ラウンドにノーマルノードと $C0$ ノードしか存在しなくなる．

証明．
- $C0$ ノード, ノーマルノード (p とする) が $enable$ で動作を行っても, p 以外のノードがこの動作の影響で IAb ノードになることはない．
- $C0$ ノードが動作するならば $B-action1$ を行い，ノーマルノードになる．
- ノーマルノード p において p が動作してもノーマルノードである．
- IAb ノードは任意の状況で $enable$ で 1 ラウンド後 $C0$ ノードになる．
- IAb ノード p_0が $Reset$(または $rReset$) を行うと，$Par_{p_1} = p_0$とするノード p_1が IAb ノードになる可能性がある．
- IAb ノード p_1が $Reset$(または $rReset$) を行うと，$Par_{p_2} = p_1$とするノード p_2が IAb ノードになる可能性がある．
- IAb ノード p_k(k は $n-1$ 以下) が $Reset$(または $rReset$) を行うと, $Par_{p_{k+1}} = p_k$とするノード p_{k+1}が IAb ノードになる可能性がある．

故に, 任意のネットワークから多くても n ラウンドにノーマルノードと *C0* ノードしか存在しなくなる． □

以降ネットワーク上には IAb ノードが存在しない前提で議論することにする．γ_{normal}をアブノーマルノード及びアブノーマルパスが存在しない状況とする．

定義 4.9. 状況γ_iにおいて"$Starting\,Configuration$"と呼ばれるための必要十分条件は以下の条件を満たすときである．
$\forall p \in V :: Pif_p^i = C$

γ_{normal}から開始して"$Starting\,Configuration$"と呼ばれる状況にに有限時間以内に到達することを示す．

ここで, $Visited$ 及び $UnVisited$ の定義から以下の補題を証明する．

補題 4.2. 以下が成り立つ

1. 任意のノード p に対して, $Pif_p = C \Rightarrow p \in Visited$ または $p \in UnVisited$
2. $\forall p \in V, Pif_p = C$ のとき $Visited \cap UnVisited = V$
3. $\exists p \in V, Pif_p \neq C$ のとき $Visited \cap UnVisited = \phi$

証明． $Pif_r = C$ ならばネットワーク上の任意のノード p において $Pif_p = C$ となる．そのため 2 が成立するのは自明. それ以外のときを考える. $UnVisited$ と $Visited$ の定義からそれに含まれる $Pif_p = C$ である p は, $q \in N_p$ で, $(Par_p = q$ かつ $(Pif_q = B$ または $q \in UnVisited))$ または $(Par_p = 0$ かつ $(q \in UnVisited$ または $Pif_q = B))$ かつ $(Par_p = q$ かつ $(Pif_q = F$ または $q \in Visited))$ または $(Par_p = 0$ かつ $\neg(Pif_q = B$ または $q \in UnVisited))$ となる q が存在する．しかし，これを満たす q は存在しない．ゆえに 3 は成立する．

最後に 1 を証明する. $UnVisited$ と $Visited$ の定義から $Pif_p = C$ である p がいずれかを満たすなら $Par_p = 0$ または $q \in N_p Par_q$ で, $Par_p = q$ かつ $Pif_q = B$ または F または $q \in UnVisited \cup Visited$ となる q が存在する．そのような p は必ず存在するため 1 は成立する． □

補題 4.3. γ_i と γ_j を 2 つの状況とする.但し, $i < j$ かつ $\forall k, i \leq k \leq j, Pif_r^k \neq C$ とする.このとき, $Visited^i \subseteq Visited^j$

証明． $p \in Visited^u, p \notin Visited^{u+1}$ となるような $u(i \leq u \leq j-1)$ が存在すると仮定する．
まず, $Pif_p^u = C$ のノードは γ_{u+1} でも $Pif_p^{u+1} = C$ であることをいう. $p \in Visited^u$ より $Pif_p^u = C$，なので, γ_u のときに p が動作をするならば, $B - action$ の動作の可能性があるので $Pif_p^{u+1} \in \{C, B\}$．ところが, $p \in Visited^u$ であり $\forall q \in N_p$ に対して $Pif_q^u \neq B$ であるため γ_u では p は $B - action$ の動作を行えず, $Pif_p^{u+1} = C$ しかあり得ない．また，親ポインタが不変なのも自明．ここで, $p \notin Visited^{u+1}$ ということは，次の条件を満たす $q \in N_p^{u+1}$ が存在する. $(Par_p^{u+1}$ または $Par_p^{u+1} = 0)$ かつ $(q \in UnVisited^{u+1}$ または $Pif_q^{u+1} = B)$ ここで次の場合を考える．

1. $Pif_{p_t}^u = F$ のとき
q が $C - action$ の動作が可能だとしても $p \in Visited^u$ よりそれはあり得ない．
2. $Pif_{p_t}^u = B$ のとき
$p \in UnVisited^u$ のため矛盾．
3. $Pif_{p_t}^u = C$ のとき
q は $B - action$ の動作が不可能．

よって $p \in Visited^u, p \notin Visited^{u+1}$ となるような $u(i \leq u \leq j-1)$ は存在しない．故に補題は成り立つ． □

補題 4.4. γ_i と γ_j を 2 つの状況とする．但し, γ_j は γ_i の 1 ラウンド後の状況とし, $UnVisited^i \neq \phi$ とする．このとき, $|UnVisited^j| < |UnVisited^i|$ となる．

証明． 根ノードの Pif の状態による場合分けをする．

1. $Pif_r^i = F$ のとき
$\forall p, Pif_p^i \in \{C, F\}$ のため, $UnVisited^i = \phi$ なのでこのときは考えない．
2. $Pif_r^i = C$ のとき
$\forall q \in V, Pif_q^i = C$ であり，根ノード r の $rB - action$ によって 1 ラウンド後, $r \notin UnVisited^j$ になる．
3. $Pif_r^i = B$ のとき
ノード p が $p \in UnVisited^i$ を満たすとすると, $UnVisited$ の定義から $Pif_p^i = C, Pif_{Par_p}^i = B$ となるようなノード p が存在する．

ノード p は $B-action$ をおこなうことができるため γ_j で $p \notin UnVisited$ となる．また補題 4.3 から $UnVisited$ は γ_i から γ_j の間に増えないため，補題は成り立つ． □

補題 4.5. $UnVisited$ が空のとき，やがて $Visited = V \backslash \{r\}$ となり，その 1 ラウンド後に $Visited = V$ となる．

証明． $UnVisited$ が空のとき，ネットワーク上の全ノードは以下の 5 つの集合に分けられる．
$S_1 = \{p \in BSubtree(r)|$ 但し, p は $BSubtree(r)$ のソース $\}$
$S_2 = \{p \in BSubtree(r)|$ 但し, p は S_1 を満たすものは除く $\}$
$S_3 = \{p \in Subtree(r) | Pif_p = F \wedge Cleaning(p)\}$
$S_4 = \{p \in Subtree(r) | Pif_p = F \wedge \neg Cleaning(p)\}$
$S_5 = \{p | Pif_p = C\}$
$UnVisited$ が空なので $Pif_r \neq C$ のときを考える. S_1 に含まれる任意のノードは $F-action$ 可能であり，動作後 S_4 に含まれる．このとき, S_2 のノードから S_1 に含まれるものが存在する．
S_3 に含まれる任意のノードは $C - action$ 可能であり，動作後 S_5 に含まれる．このとき, S_4 のノードから S_3 に含まれるものが存在する．
よって最終的に根ノード r 以外の任意のノード p は S_5 に含まれ，その 1 ラウンド後に r は $rC-action$ をするので補題は成り立つ． □

補題 4.6. 任意の状況から開始して，有限時間以内にシステムは $Starting Configration$ に至る．

証明． $C0$ ノード以外のアブノーマルノードは多くても n ラウンドでなくなる (補題 4.1).
$UnVisited$ が多くても n ラウンドでなくなる (補題 4.4)

.
$|Visited| = n-1$ となる状況に多くても $2n-1$ ラウンドで至る (補題 4.5).
1 ラウンド後, $rC-action$ によりシステムは $Starting\,Configration$ に至る. □

補題 4.7. 任意のノード $p(p \neq r)$ の親ポインタにおいて, p が q を指すとき，それを辺 $(p,q) \in T$ とする. $Starting\,Configuration$ に至ってから (このときの状況をγ_i) 全ノードが $B-action1$ または $B-action2$ の動作を行ったとき (このときの状況をγ_j)，ネットワーク $G=(V,E)$ 上に根ノード r を頂点とする全域木 $S=(V,T)$ が生成される.

証明 . γ_iからγ_jの間ネットワークの全ノードは $B-action1$ または $B-action2$ の動作を 1 度だけ行うことができる.
全域木の定義から以下の 2 つを満たせばよい

1. 閉路を持たない
2. 連結である

まず, 1.を証明する.
S に閉路が存在すると仮定して矛盾を導く.
S に閉路が存在するつまり p_0, p_1, $\cdots$, p_m, $p_0(m \geq 2)$ となるパスが存在すると，アルゴリズムから各$i(0 \leq i < m)$ に対して辺 (p_i, p_{i+1}) においてノード p_i, p_{i+1} 間のレベルの関係は常に $L_{p_i} = L_{p_{i+1}} + 1$ が成立する．すると, $L_{p_0} < L_{p_m}$かつ $L_{p_0} > L_{p_m}$が成り立ってしまい矛盾．よってグラフ S は閉路を持たない.
次に 2.を証明する.
任意のノード p は根ノード r までのパス $p=p_0, p_1$，$\cdots$, p_n, r が存在する.
同様にして，任意のノード q は根ノード r までのパス $q=q_0, q_1$，$\cdots$, q_m, r が存在する．つまり，ウォーク $p=p_0, p_1$，$\cdots$, p_n, r, q_m，$\cdots$, $q_0=q$ が存在し，それは p から q までのパスが存在することなので故に, S は連結である. □

補題 4.8. 補題 4.7 のγ_jの状況から全ノードが $F-action1$ または$F-action2$の動作を行ったとき (このときの状況をγ_k) ネットワーク上にはrを根, $TEdge(p,q)$ を満たす (p,q) を枝, $T_p=1$ を満たすノード p を頂点とするマルチキャストツリーが生成される.

証明 . γ_jのとき補題 4.7 からネットワーク上には全域木 $S=(V,T)$ が生成されている. $TEdge$ の定義から $TEdge$ を満たす辺の集合は T の部分集合といえる.
γ_jからγ_kの間ネットワークの全ノードは$F-action1$ または $F-action2$ の動作を 1 度だけ行うことができる．また，任意のノード p において $F-action1$ または $F-action2$ は $q=Par_p$とするノード q より先に行われる．ここで，マルチキャストグループのメンバが$T_p=1$であることは明らか. $T_p=1 \Rightarrow T_{Par_p}=1$であることも述語$Tree(p)$からいえる．よって$\gamma_k$でネットワーク上には全域木 S 上にマルチキャストツリーが生成されている. □

補題 4.6 と補題 4.7 と補題 4.8 から以下の定理が成り立つ

定理 4.1. プロトコル $\mathcal{MRT}$ はマルチキャストツリーを生成する自己安定プロトコルである.

5 おわりに

本論文では，動的ネットワーク上でも安定してマルチキャストルーティングができる自己安定マルチキャストツリー生成プロトコルを提案し，そのプロトコルの正当性の証明をした．安定するまでの時間はネットワークのノード数を n とすると多くても $6n$ ラウンド (任意の状況から $Starting\,Configuration$ に至るまで多くても $4n$ ラウンド，$Starting\,Configuration$ からマルチキャストツリーが生成されるまで多くても $2n$ ラウンド) である．第 1 章で プロトコル $\mathcal{MRT}$ は実際に通信遅延が小さい通信経路をもつマルチキャストツリーが生成されることが期待されると述べたがシミュレーションによる評価等でそれの検証を行うことが今後の課題である.

参考文献

[1] Sandeep K.S. Gupta, Pradip K Srimani ”Self-stabilizing multicast protocols for ad hoc networks”, 2003

[2] Alain Cournier, Ajoy.K.Datta, Franck Petit, Vincent Villain ”Self-stabilizing PIF Algorithm in Arbitrary Rooted Networks”, 2001

[3] Sayaka KAMEI,Hirotsugu KAKUGAWA ”A Self-Stabilizing Distributed Algorithm for the Steiner Tree Problem”, 2004

無線ネットワークにおける
エネルギー効率に優れた自己安定プロトコル

瀧元友也　大下福仁　角川裕次　増澤利光
大阪大学大学院情報科学研究科

概要

自己安定プロトコルは，いかなる状況から実行を開始しても有限時間で自律的に所望の状況（安全状況）へ到達するプロトコルである．自己安定プロトコルでは安全状況であることを確認するために，プロセスは永続的に通信を行い続ける必要がある．ほとんどの自己安定プロトコルでは安全状況に達した後も，各プロセスはすべての隣接プロセスとの通信を継続するため，安全状況到達後のエネルギー消費が大きくなってしまう．そこで本稿では，自己安定プロトコルが安全状況到達後，プロセスの通信デバイスをスリープさせることで，通信デバイスの駆動回数を削減し，消費エネルギーの効率化を試みる．

1　はじめに

自己安定プロトコル [2] とは，いかなる状況から実行を開始しても有限時間で自律的に所望の状況（安全状況）へ到達するプロトコルである．自己安定性により，分散システムにおける一時故障やトポロジの動的な変化に対して自律的に適応することが可能である．

自己安定プロトコルの効率は，安全状況に到達するまでの時間（安定時間）で評価することがほとんどである．しかし，自己安定プロトコルでは安全状況に到達しているかどうかを確認するために，プロセスは永続的に通信を行い続ける必要がある．これまでに提案されているほとんどの自己安定プロトコルでは安全状況に到達後も，各プロセスはすべての隣接プロセスとの通信を継続するため，エネルギー消費が大きくなってしまう．また，実際の分散システムで自己安定プロトコルを運用する場合，故障やトポロジの変化の影響を受けるのは一時的であり，ほとんどの時間は安全状況で動作していると考えられる．このため，自己安定プロトコルのエネルギー効率に関して，安全状況におけるエネルギー消費量は支配的であり，安全状況でのエネルギー消費量を削減することは重要である．

そこで，文献 [1,3–5] では，安全状況での通信量を削減することにより，**通信効率**に優れた自己安定プロトコルの提案を行っている．プロセスが一対一通信を行う分散システムにおいては，各プロセスの通信相手を制限する方法 ([1,3])，プロセスが**放送型通信**を行う分散システム（**無線ネットワーク**）においては，各プロセスの放送回数を削減する方法 ([4,5]) により，通信効率の改善を行っている．しかし，これらの手法では，常に全てのプロセスが読込（受信）を行っている．そのため，通信デバイスは，通常の自己安定プロトコルと同様に常に動作している．通信デバイスによっては，通信デバイスが駆動していると通信相手を制限しても（放送を行わなくても），消費エネルギー量はほとんど変わらないことがある．そのため，エネルギー効率を改善するには，**通信デバイスの駆動数**を削減することは重要であると考えられる．

そこで本稿では，自己安定プロトコルにおいて，プロセスの通信デバイスを**スリープ**させせることにより，通信デバイス駆動数を削減することを考える．通信デバイスのスリープさせる，すなわちプロセスが放送も受信も行わないようにすることで，スリープを行っている間は通信デバイスを休止させることができる．そのため，通信デバイスの駆動数を削減することができ，エネルギー効率を改善することが可能である．しかし，通信デバイスの駆動数が少ない動作を自己安定プロトコルとして構成することは，文献 [1,3–5] の通信効率の改善と比較すると，より困難である．文献 [1,3–5] の手法では，全てのプロセスが常に読込（受信）を行うことを前提として利用としていた．しかし，通信デバイスの駆動数を削減するためには，プロセスの通信デバイスをスリープさせなければならず，スリープしている間，そのプロセスは送信メッセージの内容だけでなく，送信メッセージの有無すら知ることができない．よって，送信されたメッセージが必ず読込（受信）されるとは限らないため，任意の状況から安全状況へ到達できることを保証することが，より困難となっている．

本稿では，プロセスが放送型通信を行う無線ネットワークを対象とし，プロセスが分散システム G のプロセス数 n の知識を持つ場合，収束時間を大きく犠牲にすることなく，通信デバイス平均駆動数 $O(1)$ が得られることを示す．具体的には，収束時間の上界 T が既知である任意のサイレントな自己安定プロトコルを，通信デバイス平均駆動数 $\frac{5}{2}$，収束時間 $O(n+T)$ である自己安定プロトコルに変換する手法を示す．

2　関連研究

自己安定プロトコルの安全状況到達後の効率性に関する研究として，通信効率という概念が提案されている [1,3–5]．

文献 [1,3] では，プロセスが一対一通信を行う分散システムを対象として，通信効率性について考察を行っている．これらの文献では，通信効率の良さを安全状況到達後も通信を継続するプロセス対の数によって評価しており，その数を減らすことにより，システム全体の通信量を削減することを狙っている．

文献 [4,5] では，プロセスが放送型通信を行う分散システム（無線ネットワーク）を対象として，通信効率性について考察を行っている．放送型通信が行われる無線ネットワークでは，プロセス v が一度メッセージを放送すると，v の全ての隣接プロセスがそのメッセージを受信する．そのため，文献 [1,3] のように通信を継続するプロセス対の数を削減することは，無線ネットワークには適さない．そこで文献 [4,5] では，通信効率の良さを安全状況到達後の放送回数によって評価しており，その数を減らすことにより，システム全体の通信量を削減することを狙っている．

これら通信効率という概念では，どちらも通信に関する命令（一対一通信では読込と書込，放送型通信では放送と受信）の実行回数に着目しており，全てのプロセスが常に読込（受信）を行えることを前提としている．一方，本稿では，安全状況到達後に各プロセスが通信デバイスをスリープさせる回数に着目しており，全てのプロセスが常に読込（受信）を行っているとは限らず，実現がより困難である．しかし，プロセスの通信デバイス自体を休止させることができ，電力消費量を削減することによってエネルギー効率化を改善することが可能である．

3　諸定義

3.1　システムモデル

分散システムは，n 個のプロセスの集合 $V(G) = \{v_0, v_1, \quad , v_{n \quad 1}\}$ と，2 つの異なるプロセスを接続する

双方向通信リンクの集合 $E(G)$ から成るグラフ $G = (V(G), E(G))$ としてモデル化する．プロセス v, w がリンクで接続されているとき，このリンクを (v, w) と表記する．$(v, w) \in E(G)$ のとき，w を v の**隣接**プロセスと呼び，v の隣接プロセスの集合を $N_G(v)$ と表記する．ただし，想定する分散システム G が明らかである場合は，$N(v)$ と表記することにする．本稿では，プロセス数が 2 以上 $(n = |V(G)| \quad 2)$，かつ連結な分散システムのみを考え，そのような分散システムの集合を $\mathcal{G}$ と表記する．

本稿では，分散システムのプロセス間通信として，**放送型通信**を仮定する．放送型通信では，各プロセス v がメッセージの送信を行うことにより，v の各隣接プロセス $w \in N(v)$ がそのメッセージを受信する．本稿では無線ネットワークを想定しており，リンク v, w はプロセス v と w が無線通信可能であることを表している．このとき，プロセス v はリンク (v, w) を識別しているわけではなく，特に断らない限り，各プロセス v は，自身の隣接プロセスの集合 $N(v)$ や，隣接プロセス数 $|N(v)|$ を知っていることを前提とはしない．また，プロセス v がメッセージを受信したときも，メッセージが送信プロセスの識別子などの情報を含まない限り，メッセージの送信元を特定することはできない．本稿では，プロトコルの実行中に無線の干渉等によるメッセージの破損や損失は生じないものとする．

プロセスは状態機械としてモデル化する．また，分散システム $G = (V(G), E(G))$ における，プロセス $v_0, v_1, \quad, v_{n \ 1} \in V(G)$ の状態の列を分散システムの**状況** $c = (s_0, s_1, \quad, s_{n \ 1})$ と定義する．ここで，s_i はプロセス v_i の状態を表す．

本稿では，全てのプロセスが同時に動作する**同期分散システム**を仮定する．その動作により状況 $c_j = (s_0, s_1, \quad, s_{n \ 1})$ から状況 $c_{j+1} = (s'_0, s'_1, \quad, s'_{n \ 1})$ へ遷移することを，$c_j \mapsto c_{j+1}$ と表記し，**ステップ**と呼ぶ．ステップ $c_j \mapsto c_{j+1}$ における各プロセス v_i の動作は，以下から成る．

1. 状態 s_i に基づくメッセージの送信．
2. v_i の各隣接プロセス $w \in N(v_i)$ が同じステップで送信したメッセージの受信．
3. 受信したメッセージと内部計算に基づく状態 s_i から状態 s'_i への遷移．

ステップ $c_j \mapsto c_{j+1}$ $(j \quad 0)$ による状況の無限列 $E = c_0, c_1, c_2, \quad$ を**初期状況** c_0 から開始する**実行**と呼ぶ．また，実行 E の c_0 から始まる状況の有限部分列のことを E の**接頭辞**，実行 E から接頭辞 $c_0, c_1, \quad, c_k$ を取り除いて得られる状況の無限列 $c_{k+1}, c_{k+2}, \quad$ のことを E の**接尾辞**と呼ぶ．このとき，接尾辞 $c_{k+1}, c_{k+2}, \quad$ は初期状況 c_{k+1} から開始する実行になっている．

3.2 自己安定プロトコル

問題の要件が，静的な解を求めて，その解を保持することである場合，その問題を**静的な問題**と呼ぶ．また，各プロセスが問題に対する解を表現するための変数を**出力変数**と呼び，プロセスの出力変数を用いて解の条件を命題として記述したものを**解条件**と呼ぶ．状況 c_0 が解条件を満たし，かつ実行 $c_0, c_1, \quad$ の各状況 $c_1, c_2, \quad$ における，全プロセスの出力変数の値が状況 c_0 と同じであるとき，状況 c_0 を（静的な問題における）**安全状況**と呼ぶ．また，プロトコルの任意の初期状況から開始する実行が有限時間内に安全状況へ到達するとき，そのプロトコルを**自己安定プロトコル**と呼ぶ．自己安定プロトコルは，どのような初期状況から開始してもいつか安全状況へ到達し，一度到達すれば出力変数の値を保持する性質をもつプロトコルである．

3.3 エネルギー効率の評価尺度

無線ネットワークにおける，自己安定プロトコルの通信効率の概念では，プロセスの放送回数を減らすことを目的としていた．しかし，1 節で議論したように，エネルギー効率の観点から考えれば，通信デバイスの駆動数を削減する方が効果的であると考えられる．本節では，駆動数を削減するために用いる通信デバイスのスリープを定義した後，エネルギー効率の評価尺度として，通信デバイスの平均駆動数を定義する．

定義 1 (通信デバイスのスリープ)**.** あるプロセス v がステップ $c_i \mapsto c_{i+1}$ において通信デバイスの動作を休止するとき，プロセス v はステップ $c_i \mapsto c_{i+1}$ において**スリープ**していると呼ぶ．このときプロセス v は，ステップ $c_i \mapsto c_{i+1}$ において隣接プロセスが放送したメッセージの内容，放送メッセージの有無を知ることができない． □

通信デバイスの平均駆動数の定義のために，次の表記を導入する．分散システム $G \in \mathcal{G}$ における，自己安定プロトコル A の実行 $c_0, c_1, \ldots$ を考える．このとき，ステップ $c_i \mapsto c_{i+1}$ においてスリープしているプロセスの集合を，$S_{A,G}(c_i)$ と表記する．

定義 2 (通信デバイス平均駆動数)**.** 任意の分散システム $G \in \mathcal{G}$ における，自己安定プロトコル A の任意の実行 $c_0, c_1, \ldots$ において，ある実数 k とある非負整数 i が存在して，

$$\lim_{j \to \infty} \frac{1}{j \quad i+1} \sum_{l=i}^{j} |P(G) \setminus S_{A,G}(c_i)| \quad k$$

が成立するとき，自己安定プロトコル A の**通信デバイス平均駆動数**が k であると呼ぶ． □

4 エネルギー効率に優れた自己安定プロトコル

本章では，プロセスが分散システム G のプロセス数 n の知識を持つ場合，収束時間を大きく犠牲にすることなく，通信デバイス平均駆動数 $O(1)$ が得られることを示す．具体的には，収束時間の上界 T が既知である任意のサイレントな自己安定プロトコルを，通信デバイス平均駆動数 $\frac{5}{2}$，収束時間 $O(n+T)$ である自己安定プロトコルに変換する手法を示す．この変換手法では，分散システム G のプロセス数 n が既知であることを仮定する．また，変換の入力とする自己安定プロトコル A は以下の**入力条件**を満たすものとする．

1. A はサイレントである．すなわち，安全状況到達後は，どのプロセスも自身の状態を変更しない．
2. 任意の実行において，全てのプロセスは毎ステップ放送を行う．
3. 任意の非安全状況 c から遷移するステップ $c \mapsto c'$ において，少なくとも 1 つのプロセスは自身の状態を変更する．
4. 収束時間の上界 T が既知である．

提案する変換手法を Protocol 1 に示す．ただし，Protocol 1 の疑似コードは，入力プロトコル A を変換して得られる出力プロトコル *EngyEff(A)* として記述したものである．通信デバイス平均駆動数 $O(1)$ を得るための基本的なアイデアは文献 [4] の手法と似ている．入力プロトコル A の安全状況に到達後，各プロセスがほとんどの時間でスリープを行い，$\Theta(n)$ ステップに一度だけ通信を行うようにする．しかし，通信を行う際，プロセス v がメッセージ m の放送を行っても，その隣接プロセス $w \in N(v)$ がスリープしているならば，w

Protocol 1 *EngyEff(A)*

```
Constants:
 1: $T$ : convergence time of $A$
 2: $T' = 3n$ : convergence time of $constructST$
 3: $T^*$    $\max\{T, T'\}$;
 4: $t$    $n + T^*$;
Internal Variables:
 5: $pif\_error$ : Boolean
Output Variables:
 6: $clock_v : \{0, \dots, t + 2n \quad 3\}$
 7: $state_v^A$
 8: $state_v^{ST} = (root_v, prnt_v, dist_v, size_v, mofp_v, Child_v)$
Actions:
 9: if $clock_v < t$ then  // ordinary mode
10:   Broadcast($id_v, clock_v state_v^A, state_v^{ST}$);
11:   $M_v$    received messages;
12:   $clock_v$    $\min(\{m.clock | m \in M_v\} \cup \{clock_v\}) + 1$;
13:   $state_v^A$    $executeA(state_v^P, M_v)$;
14:   $state_v^{ST}$    $constructST(state_v^{ST}, M_v)$;
15: else  // sleep mode
16:   if $clock_v = t$ then  // check the consistensy for the protocol $A$
17:     Broadcast($id_v, clock_v, state_v^A, state_v^{ST}$);
18:     $M_v$    received messages;
19:     $state_v^A$    $executeA(state_v^A, M_v)$;
20:     $state_v^{ST}$    $constructST(state_v^{ST}, M_v)$;
21:   end if
      // check the consistency for the clock synchronization by PIF
22:   if $clock_v = [t + 1, \dots, t + n \quad 2]$ then
23:     $pif\_error$    $propagation(clock_v, state_v^{ST}, M_v)$;
24:   else if $clock_v = [t + n \quad 1, \dots, t + 2n \quad 3]$ then
25:     $pif\_error$    $feedback(clock_v, state_v^{ST}, M_v)$;
26:   end if
      // update the value of clock
27:   if $pif\_error$ = false or $state_v^A$ changed or $state_v^{ST}$ changed or $\exists m \in M_v[m.clock \neq clock_v]$ then
28:     $clock_v$    0
29:   else
30:     $clock_v$    $(clock_v \quad t + 1) \bmod (2n \quad 2) + t$;
31:   end if
32: end if
```

Function 2 *propagation*$(clock_v, state_v^{ST}, M_v)$

```
Actions:
 1: if $clock_v \quad t = dist_v$ then
 2:   Broadcast($id_v, clock_v state_v^A, state_v^{ST}$);
 3: else if $clock_v \quad t + 1 = dist_v$ then
 4:   $M_v$    received mesasges;
 5:   if $\forall m \in M_v \ [m.id \neq state_v^{ST}.prnt]$ then
 6:     return false;
 7:   end if
 8:   $state_v^{ST}$    $constructST(state_v^{ST}, M_v)$;
 9: end if
10: return true;
```

Function 3 *feedback*$(clock_v, state_v^{ST}, M_v)$

```
Internal Variables: mofp: message sent from parent;
Actions:
if $2n \quad 2 + t \quad clock_v = dist_v$ then
    Broadcast$(id_v, clock_v state_v^A, state_v^{ST})$;
else if $2n \quad 3 + t \quad clock_v = dist_v$ then
    $M_v$    received mesasges;
    if $\{m \in M_v | m.state^{ST}.prnt = state_v^{ST}.id\} \neq state_v^{ST}.Child$ then
        return false;
    end if
    $state_v^{ST} \quad constructST(state_v^{ST}, M_v)$;
    if $dist_v = 0$ and $size_v \neq n$ then
        $pif\_error$    true;
    end if
end if
return true;
```

Function 4 *constructST*$(state_v^{ST},\ M_v)$

```
Actions:
$root_v \quad \max(\{m.root | m \in M_v,\ m.dist \quad n \quad 2\} \cup \{id_v\})$;
if $root_v = id_v$ then
    $prnt_v \quad \perp$;
    $dist_v \quad 0$;
else
    $mofp \quad m$ s.t. $m.dist = \min\{m'.dist | m' \in M_v, m'.root = root_v\}$;
    $prnt_v \quad mofp.id$;
    $dist_v \quad mofp.dist + 1$;
end if
$Child_v \quad \{m.id | m \in M_v, m.prnt = id_v\}$;
$size_v \quad \sum_{u \in Child_v} size_u + 1$;
return$(root_v, prnt_v, dist_v, size_v, mofp_v, Child_v)$;
```

は m を受信しないことに注意しなければならない．そのため，スリープを行いながら確実に通信を行うためには，通信を行うタイミングをプロセス同士で同期させなければならない．また，安全状況に到達する前は，高頻度で通信を行うことを保証しなければならない．そうでなければ，放送メッセージが他のプロセスの伝搬せず，収束時間が増大する，または収束性を保証できない可能性がある．

これらの問題を克服するために，文献 [4] の手法と同様，**同期時計**メカニズムを用いる．各プロセス v は変数 $clock_v$ を持ち，次のような使用法をとる．

1. 時計値のある値 t を閾値として，安全状況に到達したか否かを判断し，動作を管理する（t の値については後述）．
2. $clock_v \quad t$ のとき，プロセス v は安全状況到達後であると判断し，ほとんどの時間でスリープを行う．このとき，確実に放送メッセージを受信させるために，通信を行うタイミングを $clock_v$ の値を用いて同期させる．また，通信を行った際に，状況が入力プロトコル A に対して安全かどうか，時計が同期されているかどうかの確認を行う．状況が非安全 (時計値が同期されていないことも含む) であることを発見した場合は，安全状況に未到達であることを他のプロセスに知らせるために $clock_v \quad 0$ を実行し，時計値をリセットする．

3. $clock_v < t$ のとき，プロセス v は安全状況に未到達であると判断し，毎ステップで通信を行い，入力プロトコル A を実行する．また，時計値を同期させるために，$clock_v \quad \min\{clock_w | w \in N(v) \cup \{v\}\} + 1$ を実行する．毎ステップで通信を行う $clock_v < t$ の間に，確実に安全状況に到達させるために，$t \quad T$ となるようにする（T の定義より，全てのプロセスが毎ステップで通信を行いながら T ステップ実行すれば，確実に A の安全状況へ到達する）．

しかし，このように同期時計メカニズムを実装しても，さらに克服すべき問題が存在する．1 つ目は，時計値のリセット信号の伝搬にかかる時間が大きくなることである．例として，全てのプロセスが $clock \quad t$ であるときに，1 つのプロセス v が時計値をリセット $clock_v \quad 0$ を実行する場合を考える．プロセス v の隣接プロセスは $\Theta(n)$ ステップに一度しか通信を行わないため，時計値 $clock_v$ が隣接プロセスに伝搬するためには，最悪 $\Theta(n)$ ステップかかる．よって，最悪時ではリセット信号が全てのプロセスに伝搬するまでに，$\Theta(n^2)$ ステップかかってしまうことになる．

2 つ目は，通信リンクで接続されたプロセスの対が，永遠に通信を行うことができない場合が存在することである．例としては図 1 のような場合が考えられる．$clock \quad t$ のとき，図 1 に示した状態遷移を行うものとする．システムの状況が図 1 のように，左の 3 つのプロセスがスリープ状態，右の 3 つのプロセスが通信状態である場合を考える．このとき，右のプロセスが通信を行おうとするが，左のプロセスはスリープしているため，左右のプロセス間でメッセージは伝わらない．同様に，左のプロセスが通信状態，右のプロセスがスリープ状態である場合も，左右のプロセス間でメッセージが伝わらない．このように左右のプロセス間で通信状態が同期しなければ，永遠に通信を行うことができない．

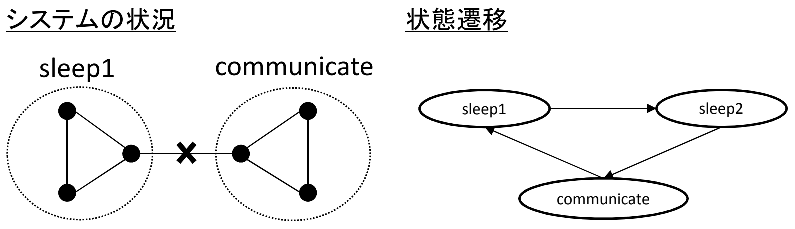


図 1　通信リンクで接続されたプロセスの対が，永遠に通信を行うことができない例

これらの問題を克服するために，入力プロトコル A の実行に並行して，自己安定的に全域木を構成することを行う．各プロセス v が全域木と時計値 $clock_v$ を用いて，以下のように通信を管理することで，問題を克服する．

1. 時計値のある値 t を閾値として，安全状況に到達したか否かを判断し，動作を管理する．
2. $clock_v \quad t$ のとき，プロセス v は安全状況到達後であると判断し，ほとんどの時間でスリープを行う．このとき，確実に放送メッセージを受信させるために，通信を行うタイミングを $clock_v$ の値を用いて同期させる．同期する方法は以下の通りである．
   (a) $clock_v = t$ のとき，放送・受信を共に行い，入力プロトコル A と全域木構成プロトコルに対して安全かどうか，時計が同期されているかどうかの確認を行う．状況が非安全 (時計値が同期されていないことも含む) であることを発見した場合は，安全状況に未到達であることを，他のプロセスに知らせるために $clock_v \quad 0$ を実行し，時計値をリセットする．
   (b) $t + 1 \quad clock_v \quad t + n \quad 2$ のとき，構成された全域木の根から葉へ，波状に通信を行う．プロセス v の根からの距離が i，プロセス v を親に持つプロセスを w とすると，$clock_v = t + i$ であると

きにプロセス v が放送，同時に（$clock_w = t + i$ であるときに）プロセス w が受信を行う（それ以外のプロセスはスリープする）．このようにすることで，全域木の根から葉へ波状に通信を行うことができる．このとき，受信を行ったプロセス w は全域木構成プロトコルに対して安全かどうか，時計が同期されているかどうかの確認を行う．状況が非安全 (時計値が同期されていないことも含む) であることを発見した場合は，$clock_w \quad 0$ を実行し，時計値をリセットする．このときに，入力プロトコル A に対して安全かどうかは判断できない．根からの距離でスリープのタイミングを同期しているため，必ずしも全ての隣接プロセスからメッセージを受信するとは限らないためである．

(c) $t + n \quad 1 \quad clock_v \quad t + 2n \quad 3$ のとき，構成された全域木の葉から根へ，波状に通信を行う．具体的な方法は，根から葉への場合と同様である．異なる部分は，根から距離 1 のプロセス v が放送を行い，根のプロセス w が受信を行う所である．このとき，プロセス w は木の大きさが n であるかどうかの確認を追加で行う．

(d) 全ての場合において, 時計のリセットを行わない限りは, $clock_v \quad (clock_v \quad t{+}1) \bmod (2n \quad 2){+}t$ を実行することによって時計の更新を行う．

3. $clock_v < t$ のとき，プロセス v は安全状況に未到達であると判断し，毎ステップで通信を行い，入力プロトコル A と全域木構成プロトコルを実行する．また，時計値を同期させるために，$clock_v \quad \min\{clock_w | w \in N(v) \cup \{v\}\} + 1$ を実行する．
ここで，毎ステップで通信を行う $clock_v < t$ の間に，確実に安全状況に到達させることを保証しなければならない．$clock_v \quad t$ の動作より，非安全であることを検知した場合は，いずれの場合もリセット $clock \quad 0$ を実行する．よって，時計値がリセットされた後ならば，時計は $clock \quad n$ である間に同期される．全域木構成プロトコルの収束時間を T' とすると，その時点から，$\max\{T, T'\}$ ステップの間，入力プロトコル A と全域木構成プロトコルを実行することによって，確実に A と全域木構成に対する安全状況へ到達することを保証できるため，$t = n + \max\{T, T'\}$ と設定すればよい．

このように，全域木構成と同期時計を連動させることによって，前に議論した 2 つの問題を克服することができる．1 つ目の問題と同じ状況を想定する（全域木が正しく構成されているとは限らない）．隣接するプロセス v, w が，親子関係を正しく築けており，かつ時計が同期している場合は（例えば，$prnt_v = id_w, dist_v = dist_w + 1, id_v \in Child_w, clock_v = clock_w$ が成立する場合），$clock \quad t$ の場合の波状の通信が正常に動作する．逆に，親子関係を正しく築けていない，または時計が同期していない場合は，波状の通信が正常に動作しない．波状の通信が正常に動作していないことをプロセスが検知した場合，そのプロセスは自身でリセットを実行する．よって，たとえ全てのプロセスが $clock \quad t$ を満たす場合でも，入力プロトコル A に対して非安全または波状の通信が正常に動作しない部分は，$\Theta(n)$ ステップ以内にリセットを実行する．入力プロトコル A に対して安全でかつ波状の通信が正常に動作する部分は，リセット信号を受信した場合，波状の通信によって，$\Theta(n)$ ステップで正常に動作する部分全体にリセット信号を伝搬する．したがって，全てのプロセスが $clock \quad t$ を満たす場合でも，$\Theta(n)$ ステップでリセット信号を全体へ伝搬させることが可能となる．

次に 2 つ目の問題と同じ状況を想定し，図 1 のシステム状況を考える．全域木が正しく構成されている場合，左右のプロセス間で時計値が同期されていないため，波状の通信が正常に動作していないことをプロセスが認識できる．よってリセットを実行できるため，永遠に通信を行わないということは起こらない．全域木が正しく構成されていない場合，例えば，左と右で独立して大きさ 3 の木が構成されている場合を考える．この

場合，2.(c) の葉から根へ波状に通信を行う際に，根のプロセスが木の大きさが本当に $n = 6$ であるかどうかを確認するため，誤りを検知することができる．よってこの場合もリセットを実行できるため，永遠に通信が行われないということは起こらない．このように，2 つ目の問題も克服することが可能である．

補題 1 (安全状況)**.** r を $id_r = \max\{id_v | v \in P(G)\}$ を満たすプロセスとする．*EngyEff(A)* の実行において，状況 c が以下の条件を満たすとき，c は**安全状況**である．

1. $\forall v, w \in P(G), clock_v = clock_w \quad t$.
2. c' を n 項組 $(state^P_{v_0}, \ldots, state^P_{v_{n-1}})$（ただし，$\{v_0, \ldots, v_n \quad 1\} = P(G)$）とすると，$c'$ はプロトコル P に対して安全状況である．
3. $root_r = id_r, prnt_r = \perp, dist_r = 0$.
4. $\forall v \in P(G) \quad \{r\}, root_v = id_r, prnt_v \in N_v, dist_v = dist_{prnt_v} + 1$.
5. $\forall v \in P(G), Child_v = \{id_w | w \in N(v), prnt_w = id_v\}, size_v = \sum_{w \in Child_v} size_w + 1$.

□

補題 2 (収束性)**.** 任意の初期状況から開始しても，プロトコル *EngyEff(A)* は $O(n+T)$ ステップで安全状況へ到達する．

証明. *EngyEff(A)* の任意の実行 $c_0, c_1, \ldots$ を考え，概要のみを示す．また $T^* = \max\{T, T'\}$ とする．

- まず $c_0, \ldots, c_{n+T^*}$ の間に $clock_v \quad t$ なるプロセス v が存在しない場合を考える．この場合，$c_0, \ldots, c_{n+T^*}$ の間はどのプロセスもスリープを行わない．よって開始から高々 $n \quad 1$ ステップで，全てのプロセスの $clock$ の値が同期される．その後，全てのプロセスが通信を行いながら T^* ステップの間，入力プロトコル P と全域木構成プロトコルを実行するので，c_{n+T^*} までに安全状況に到達することが保証される．
- 次に，$c_0, \ldots, c_{n+T^*}$ の間に $clock_v \quad t$ なるプロセス v が存在する場合を考える．詳細は省略するが，$c_{4n \quad 4}$ 以降は，全てのプロセスの $clock$ の値が同期しない限りは，新たに $clock_w$ なるプロセス w は発生しないことが示せる．よって，$T' = 3n$，即ち $T^* \quad 3n$ より，c_{n+T^*} 以降は，時計値が同期しない限りは，新たにスリープを行うプロセスは発生しない．
  - c_{n+T^*} までに $clock \quad t$ で同期する場合を考える．このとき，既に安全状況である場合は証明完了であるため，以降では，安全でない場合を考える．詳細は省略するが，このとき，$clock \quad t$ で同期後から $O(n)$ ステップで全てのプロセスがリセット（$clock \quad 0$）を実行し，その $n \quad 1$ ステップ後に全てのプロセスが $clock = n \quad 1$ で再び同期することを示せる．この議論を行うためには，全てのプロセスがリセットした後，時計値の伝搬が完了するまでに新たにスリープを始めるプロセスが存在しないことを保証する必要がある．しかし，上記（$c_{4n \quad 4}$ 以降は，全てのプロセスの時計値が同期しない限りは新たにスリープを行うプロセスが発生しない）と同様の議論を行うことで，そのようなプロセスが存在しないことは保証可能である．
  - c_{n+T^*} までに $clock \quad t$ で同期しない場合を考える．同様に，c_{n+T^*} 以降は，全てのプロセスの時計値が同期しない限りは新たにスリープを行うプロセスが発生しないことがいえる．また，このとき，c_{n+T^*} までに $clock \quad t$ であるプロセスは全て $O(n)$ ステップでリセット（$clock \quad 0$）を実行することが示せる．よって上の場合と同様の議論により，c_{n+T^*} から $O(n)$ ステップで，全てのプロセスの時計値が $clock = n \quad 1$ で同期することが示せる．

全てのプロセスの時計値が $n \quad 1$ で同期した後は，全てのプロセスが通信を行いながら T^* ステップの間，入力プロトコル A と全域木構成プロトコルを実行することにより，安全状況に到達することが保証できる．

よって，$T' = 3n$ より，任意の初期状況から実行を開始しても $O(n+T)$ ステップで安全状況へ到達する． □

プロトコル *EngyEff(A)* の任意の実行において，安全状況到達後，$2n \quad 2$ ステップ周期で波状通信を繰り返す．$2n \quad 2$ ステップの間で，$clock = t$ のとき，1 ステップで全てのプロセスが放送・受信を行うため，通信回数は n 回である．$t+1 \quad clock \quad t+2n \quad 3$ のときは放送と受信を別に行う．まず放送に関して考える．根からの距離が 1 から $n \quad 2$ のプロセスが 2 回放送を行い，根からの距離が $n \quad 1$ のプロセスが 1 回放送を行うため，通信回数は高々 $2(n \quad 1) \quad 1$ 回である．次に受信に関して考える．根からの距離が 2 から $n \quad 2$ のプロセスが 2 回受信を行い，根と根からの距離が 1，根からの距離が $n \quad 1$ のプロセスが 1 回受信を行うため，通信回数は高々 $2(n \quad 1) \quad 1$ 回である．よって，安全状況到達後，初めて $clock = k$ になる状況を c_i とすると，以下が成立する．

$$\begin{aligned}
\lim_{j\to\infty} \frac{1}{j \quad i+1} \sum_{l=i}^{j} |P(G) \setminus S_{A,G}(c_l)| &= \lim_{j'\to\infty} \frac{1}{(2n \quad 2)j'} \sum_{l=i}^{i+(2n \quad 2)j' \quad 1} |P(G) \setminus S_{A,G}(c_l)| \\
&= \lim_{j'\to\infty} \frac{1}{(2n \quad 2)j'} \{n + (2n \quad 2 \quad 1) + (2n \quad 2 \quad 1)\} j' \\
&\quad \lim_{j'\to\infty} \frac{1}{(2n \quad 2)j'} (5n \quad 5) j' = \frac{5}{2}
\end{aligned}$$

定理 1. 全体のプロセス数 n の知識を各プロセスが持ち，入力条件を満たす収束時間が T の自己安定プロトコル A が存在するとき，*EngyEff(A)* は A が対象とする問題を解く，通信デバイス平均駆動数 $\frac{5}{2}$，収束時間 $O(n+T)$ の自己安定プロトコルである． □

5 まとめ

本稿では，放送型通信を用いる無線ネットワークにおける自己安定プロトコルのエネルギー効率化を試みた．具体的には，スリープを行うことによって，プロセスが分散システム G のプロセス数 n の知識を持つ場合，通信デバイス平均駆動数 $O(1)$ が得られることを示した．

また，変換手法の条件の 1 つである，プロセスが持つ知識に関して次のことが示せる．まず，文献 [4] の結果を用いれば，通信デバイス平均駆動数の削減に関する不可能性の結果を示すことができる．具体的には，プロセスが G のプロセス数 n の上界に関する知識を持たない場合，通信デバイス平均駆動数 $o(n)$ を得ることは実質的に不可能であることが証明できる．また，本稿で提案した変換手法を拡張することにより，プロセスが持つ知識が，n でなくとも n の上界 $n \quad N < 2n$ であれば，通信デバイス平均駆動数 $O(1)$ が得られることを示せる．一方，プロセスが持つ知識が，n の上界 $N \quad 2n$ である場合，同様の手法では通信デバイス平均駆動数 $O(1)$ が得ることができない．しかし，一般的に，プロセスが持つ知識が n の上界 $N \quad 2n$ であるとき，通信デバイス平均駆動数 $O(1)$ を得ることが可能かどうかは未解決であり，今後の課題の 1 つとして挙げられる．

参考文献

[1] S. Devismes, T. Masuzawa, and S. Tixeuil, *Communication efficiency in self-stabilizing silent protocols*, Proc. of IEEE ICDCS, pp.474–481 (2009).
[2] S. Dolev, *Self-stabilization*, The MIT press, 2000.
[3] T. Masuzawa, T. Izumi, Y. Katayama, and K. Wada, *Brief announcement: Communication-efficient self-stabilizing protocols for spanning-tree construction*, Proc. of OPODIS, pp.219–224 (2009).
[4] T. Takimoto, F. Ooshita, H. Kakugawa, and T. Masuzawa, *Communication-Efficient Self-stabilization in Wireless Networks*, Proc. of SSS (2012).
[5] 瀧元友也，大下福仁，角川裕次，増澤利光, 無線ネットワークにおける自己安定プロトコルの通信効率に関する考察, 第 7 回情報科学ワークショップ (2011).

Randomized Weak Stabilizing Algorithms under Probabilistic Schedulers

Yukiko Yamauchi[1] ⋆, Sébastien Tixeuil[2], Shuji Kijima[1], Masafumi Yamashit[1]

[1] Kyushu University, Japan.
[2] UPMC Sorbonne Universites, France

Abstract. Probabilistic self-stabilizing systems guarantees that any execution eventually reaches a legitimate execution with probability 1. Unlike self-stabilizing systems, probabilistic self-stabilizing systems are easy to design, and indeed any weak stabilizing system can be automatically transformed into a probabilistically stabilizing system by randomizing the algorithm and/or by modeling a scheduler as a stochastic process, provided that the number of configurations is finite. Since the scheduler is an abstraction of the environment that we cannot control, we cannot choose a favorite probability distribution the scheduler obeys. But we can choose a one for the algorithm. In this paper, we address the problem of designing a good probability distribution for a given algorithm so that the randomized weak stabilizing system under a given probabilistic scheduler can exhibit a good convergence time. Specifically, for some wide and natural classes of probabilistic schedulers, we characterize the class of algorithms for which we can choose a probability distribution such that the corresponding convergence time becomes finite. We also extend this result to the case in which the number of configurations is infinite.

1 Introduction

Modern distributed systems and networks require that resilience to faults and attacks is considered at a very early stage of algorithm design. The paradigm of self-stabilization [9, 10, 30] yields a unified approach for recovering any kind of *transient* fault or attack, and is oblivious of their cause or extent. Intuitively, a self-stabilizing system recovers correct behavior in finite (and bounded) time after being started from an arbitrary global state. Quantifying the time needed to recover (that is, the convergence time) is the main complexity measure of self-stabilizing systems.

Amongst system hypotheses that are made by self-stabilizing distributed systems, the notion of a *scheduler* (or daemon [14]) is one of the most complex, as it captures the various options for selecting processes for execution. A scheduler is essentially a predicate on possible schedules (in each configuration, it schedules a set of processes for executing their algorithm code) and is often seen as an *adversary* by the protocol: the less restrictive the scheduler is, the more possibilities are offered to lengthen the convergence time, and sometimes preventing stabilization altogether by making the convergence time being infinite.

As various impossibility and complexity issues occur when deterministic self-stabilization is considered. They are also difficult to design and prove their correctness. Weaker notions of self-stabilization were proposed [30]. *Pseudo stabilization* [5] guarantees that every execution has a suffix that satisfies the problem specification, yet the time needed before reaching this suffix is unbounded. *Practical Stabilization* [13] loosens the requirement that after recovery, the system is always correct; instead, it remains correct for a practically infinite time (*i.e.* the time to increase a counter from 0 to 2^{128}). *Probabilistic Stabilization* [22] only guarantees that correct behavior is recovered in bounded time with probability 1 (the expected convergence time is finite). *Loose stabilization* [29] has a short (polynomial) expected convergence time and a long (exponential) stabilized phase afterwards. *Weak Stabilization* [19] does not guarantee that every execution proposed by the scheduler recovers a correct behavior. Instead, starting from any arbitrary initial state, at least one execution in the scheduler's set recovers a correct behavior.

⋆ Corresponding author. Address: 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan. Fax: +81-92-802-3637. Email: `yamauchi@inf.kyushu-u.ac.jp`

The fundamental significance of weak stabilization was recently outlined [8]. First, there exists a transformation from a weak-stabilizing system under the fair deterministic scheduler to a probabilistic stabilizing system under the uniform probabilistic scheduler (that selects each process with probability $1/2$ in each configuration). Also, uniform randomization of a deterministic algorithm translates a weak-stabilizing system under the fair deterministic scheduler to a probabilistically stabilizing system under the synchronous scheduler (that selects all processes in each configuration). This second transformation uses a simple probability distribution to randomize an algorithm: when scheduled, a process simply tosses a coin to choose whether it should execute its (deterministic) algorithm. Of course, these transformations can also be used for more elaborate cases, where the scheduler is not uniform probability distribution, the algorithm is not uniformly randomized, or the combination of them. A natural question raised by this work is the interdependency between the (possibly) biased choices of the scheduler and the (possibly) biased choices of the algorithm with respect to the expected convergence time. For a given probabilistic bias of the scheduler, can we derive a counter-measure bias for the algorithm ? What is the worst possible scheduler bias for an algorithm to cope with ?

Our Contribution. We investigate the transformation of weak stabilizing system into probabilistically stabilizing ones, with an emphasis on the expected performance (that is, convergence time) of the transformation result. We assume that the randomization of a weak-stabilizing algorithm is modeled by a probability distribution over the original transitions, and we consider probabilistic scheduler that are defined by a set of finite state Markov chains. We also assume that a scheduler is an abstraction of the environment outside the system and once an execution starts, a scheduler does not know the configuration of the system. Our performance criteria is the expected convergence time, that is, the expected number of steps from the worst possible initial configuration to a legitimate configuration (*i.e.* a configuration from which every further execution is correct).

In more details, let $\tau_{\mathcal{D},\mathcal{M}}$ be the expected convergence time of a probabilistically stabilizing system with probability distribution $\mathcal{D}$ of the algorithm and probabilistic scheduler instance $\mathcal{M}$. We show a necessary and sufficient condition for a *finite* system to have $\tau_{\mathcal{D},\mathcal{M}} < \infty$. A system is finite if its set of configurations is finite. Our condition is that the transition diagram of a system satisfies a *regularity* property, which is newly introduced in this paper. Then, we give a necessary and sufficient condition for an *infinite* system to have a finite expected convergence time. This second condition has impact on translation techniques that also promise finite expected convergence time for infinite systems, as previous work [8] only investigated the case of finite systems.

Related Works. Randomized self-stabilizing algorithms are often used for symmetry breaking [22, 24, 20, 21, 26, 4, 25, 6] or reducing space complexity of particular problems [23, 1, 2].

Most of the theoretical papers that investigate probabilistically stabilizing algorithms consider scheduler that are *adaptive*, that is, they may choose processes that are scheduled for execution based on the current global state of the system (and possibly its history in the current execution). This scheme was popularized by the scheduler-luck game paradigm [12]: the luck is the randomization of the algorithm, and the game consists in alternating (possibly non-deterministic yet not necessarily probabilistically) choices of the scheduler and random tosses by the algorithm. In this context, expected convergence time was investigated using hitting time of Markov chains [16], coupling technique of Markov chains [18], or Markov decision processes [3]. The adaptivity of the scheduler makes it extremely powerful, and permit to derive worst case theoretical upper bounds on the convergence time.

However, such strong schedulers are arguably relevant from a practical standpoint. Schedulers are supposed to model the amount of "asynchrony" in the system, and in real networks with actual hardware, the clock rates of machines and the properties of the communication media yield a probability distribution that can be obtained by statistical observation of various parameters in the network. To our knowledge, every previous implementation of self-stabilizing protocols in a simulator software [17, 28, 31, 27] (that require to implement the scheduler) assumed the scheduler was following a probability distribution that did *not* depend on the current state or history of the execution. Of course, all general network simulators do not implement the notion of an "adaptive" scheduler either. Although the non-adaptive scheduler is simpler to cope with (and thus less powerful), the obtained convergence time complexity is more likely to match the actual performance of real systems. Uniform probabilistic schedulers such as those appearing in [11, 15, 8] permit to use a Markov chain to represent the probabilistic behavior of schedulers and randomized algorithms altogether: an execution of such a probabilistic system corresponds to a random walk on the transition diagram of the system. Our notion of (possibly non-uniform) probabilistic scheduler naturally extends the previous uniform notion, as such a distribution can be obtained in practise by observing an actual network a sufficiently long time so that the obtained data is statistically meaningful.

Outline. In Section 2, we present the system model and give the formal definitions of our refinements related to probabilistic stabilization. We then show a necessary and sufficient condition for obtaining finite convergence time under probabilistic schedulers. The case of finite systems is presented in Section 3 and that of infinite systems in Section 4. Section 5 outlines concluding remarks and open questions.

2 Preliminary

A *distributed system* is defined by a pair $(N, \mathcal{A})$ of a communication graph N and a distributed algorithm $\mathcal{A}$. A communication graph $N = (P, L)$ is a directed graph where P ($|P| = n$) is the set of processes and L ($|L| = m$) is the set of communication links. An algorithm $\mathcal{A} = \{A_p : p \in P\}$ is a set of local algorithms A_p at each process $p \in P$. Each process $p \in P$ is a state machine that maintains local variables. When an edge (p, q) is in L, process p can read the local variables of q. We call q a predecessor of p and we denote by N_p the set of all predecessors of p. Note that each process p can read and write to its local variables.

A *state* of process $p \in P$ is an assignment of a value to each of its local variable drawn from its specified domain. Let Γ_p be the set of all states of p. A *configuration* is a set of states of all processes. The set of all configurations is $\Gamma = \prod_{p \in P} \Gamma_p$. We say that a distributed system is *finite* if Γ is finite, and *infinite* if Γ is countably infinite.

A *deterministic algorithm* A_p is described by a sequence of guarded commands $\langle \text{guard} \rangle \rightarrow \langle \text{command} \rangle$. A $\langle \textit{guard} \rangle$ is a predicate over the state of p and its neighboring processes, and $\langle \textit{command} \rangle$ is a statement that changes the values of local variables of p. In a configuration $\gamma \in \Gamma$, a guarded command is *enabled* if its guard is satisfied, and p is *enabled* when at least one of the guards is satisfied. If p is enabled in γ, and a scheduler, which we will define later, *activates* p, one of the commands corresponding to enabled guards is executed. When more than one guard is enabled in γ, the command corresponding to the first (in the order of $\mathcal{A}$) guard command enabled is executed when the process is activated.

A *scheduler* is an abstraction of the environment and specifies which process it allows to execute at a given time. The *deterministic scheduler* is a set of infinite sequences of a subset of P.[3] We denote by σ_F the (strongly) *fair* scheduler, which is the set of all (strongly) fair sequences, i.e., every process appears infinitely many times in every sequence in σ_F. The *synchronous* scheduler σ_S always activates P. The *central* scheduler σ_C contains all fair sequences consisting of process sets of size 1.

An *execution* of a distributed system under scheduler σ is a sequence of configurations $\mathcal{E} = \gamma_0, \gamma_1, \ldots$ that is defined as follows: Scheduler σ first non-deterministically selects a sequence $Z = Z_0, Z_1, Z_2, \ldots$ in σ. For any $t \geq 0$, let $X_t \subseteq P$ be the set of enabled processes in γ_t. Then the processes in $X_t \cap Z_t$ are activated in configuration γ_t. If the algorithm is deterministic, each of the activated processes executes its first guarded command enabled, and their executions yield the next configuration γ_{t+1}.

The *transition diagram* of a distributed system $(N, \mathcal{A})$, is a labeled directed graph $\mathcal{S} = (\Gamma, T, \lambda)$ where the set of directed edges T is the set of transitions defined over Γ by $\mathcal{A}$. Each directed edge $(\gamma, \gamma') \in T$ is labeled with $\lambda(\gamma, \gamma') \subseteq P$ that represents the set of processes which update their states in the transition from γ to γ'. From the definition of the order on guarded commands, for any configuration γ, no two transitions have the same label and if $\lambda(\gamma, \gamma') \neq \lambda(\gamma, \gamma'')$, then $\gamma' \neq \gamma''$. We use $\mathcal{S}$ to denote the distributed system and its transition diagram.

[3] In the literature, the output of a scheduler is assumed to be a subset of the enabled processes. We simply assume that the output is a subset of P, to make the behavior of the scheduler (the environment) more independent of a particular distributed system under consideration. However, whether it is a correct formulation or not is controversial. Another formulation defines a scheduler as a predicate. We do not take this formulation, since we need to specify detailed properties of instances of a scheduler

Since we cannot control the environment, we consider a scheduler as an adversary and conduct a worst case analysis, assuming that the adversary does not know the results of probabilistic choices at processes a-priori.

Randomized Algorithm. A *randomized algorithm* is a pair $\langle \mathcal{A}, \mathcal{D} \rangle$ where $\mathcal{A}$ is a deterministic algorithm and $\mathcal{D} = \{D_p : p \in P\}$ is a set of probability for the execution of $\mathcal{A}$. Like a deterministic algorithm, each algorithm A_p at $p \in P$ is a sequence of guarded commands, but the command executed when it is activated is determined probabilistically in the following way: When Z is the set of guards enabled at p in γ, then A_p is associated with a probability D_p^Z. When p is activated by the scheduler, $z \in Z$ is chosen for execution with probability $D_p^Z(z)$, where a special symbol $\perp$ means no guarded command, i.e., $D_p^Z(\perp)$ is the probability that no guarded command is executed even if p is enabled in γ. D_p^Z may depend on local information available for p, i.e., the current states of p and its predecessors N_p. For simplicity, we omit Z from D_p^Z and denote it by D_p, whenever it is obvious from the context. However, in this paper, we restrict ourselves to consider a rather restrictive class of pure probability distribution, because of the reason we will state later.

A probability distribution $\mathcal{D}$ is *pure* if for any p and Z, $D_p^Z(z) > 0$ only if z is either the first guarded command enabled in A_p or $\perp$. We denote by $\mathcal{D}_\delta$ a pure probability distribution that assigns probability $1-\delta$ $(0 < \delta \leq 1)$ to $D_p^Z(\perp)$. For example, $\mathcal{D}_{1/2}$ allows each process p execute A_p with probability $1/2$ and ignore the activation with probability $1/2$.

For any execution $\mathcal{E} = \gamma_0, \gamma_1, \ldots$, each process activated in γ_t first chooses a guarded command at random with the probability $\mathcal{D}$ from the enabled ones, and it activates it. Then like the case of deterministic algorithm, their executions yield a system transition from γ_t to γ_{t+1}.

The transition diagram of $\langle A, \mathcal{D} \rangle$ is defined by a transition diagram and transition probability of each edge. We denoted this diagram by $\mathcal{S}_\mathcal{D}$. When $\mathcal{D}$ is pure probability distribution, the transition graph of $\mathcal{S}_\mathcal{D}$ is identical to $\mathcal{S}$.

Probabilistic Scheduler. Let us fix a process set P. A Markov chain $\mathcal{M}$ over a labeled directed graph $H = (\Omega, A, \mu)$ with an edge labeling function μ from A to 2^P and a transition probability $P = (P_{(i,j),\mu(i,j)})$ is called a probabilistic scheduler instance, if it satisfies that, for any $i \in \Omega$, (1) there is a $j \in \Omega$ such that $P_{(i,j),\mu(i.j)} > 0$ and $\mu(i,j) \neq \emptyset$, and (2) $\mu(i,j) \neq \mu(i,j')$ for any edges (i,j) and (i,j'). Intuitively, an edge (i,j) has a label X means that if the scheduler is at i, then with probability $P_{(i,j),\mu(i,j)}$, it allows to activate each of the processes in $\mu(i,j) \subseteq P$ (if it is enabled). The first requirement thus states that every probabilistic scheduler instance always allows to activate some process with a positive probability. A *probabilistic scheduler* for P is a set of probabilistic scheduler instances for P. In the following, whenever it is obvious from the context, we use $\mathcal{M}$ instead of $\mathcal{M} = (H, P)$.

A probabilistic scheduler instance is *finite*, if its state space is finite. We denote the set of all finite probabilistic scheduler instances by ρ_F, and call it simply the *finite* probabilistic scheduler. In this paper, we are particularly interested in the following three subclasses of ρ_F. In a *central* probabilistic scheduler instance, an edge $(i,j) \in E$ with $P_{i,j} > 0$ is associated only with a singleton X. A probabilistic scheduler instance is said to be *oblivious* (i.e., memory-less) if Ω is a singleton. The *central* probabilistic scheduler denoted by ρ_C is the set of all central probabilistic scheduler instances, the *oblivious* probabilistic scheduler denoted by ρ_O is the set of all oblivious probabilistic scheduler instances, and by ρ_{OC} we denote the set of all oblivious and central probabilistic scheduler instances.[4] By definition, $\rho_{OC} \subset \rho_O \subset \rho_F$, and $\rho_{OC} \subset \rho_C \subset \rho_F$.

Like a deterministic scheduler, we also regard a probabilistic scheduler as an adversary and conduct a worst case analysis, i.e., it selects the worst probabilistic scheduler instance and the worst initial state for the given algorithm, but when the algorithm is randomized, we assume that the adversary does not know a-priori the results of the probabilistic choices in the processes.

Self-stabilization. A *specification* for a distributed system is a predicate defined over its executions. Consider a distributed system $\mathcal{S}$ executing an algorithm $\mathcal{A}$ on a communication graph $N = (P, L)$ under a scheduler σ, and let $\mathcal{SP}$ be the specification for $\mathcal{S}$. We say that $\mathcal{S}$ is *self-stabilizing* for $\mathcal{SP}$ under σ, if any execution under σ contains a legitimate configuration, where a configuration is *legitimate* under σ, if any execution under σ starting from the configuration satisfies $\mathcal{SP}$. We denote the set of legitimate

[4] An important subclass of ρ_{OC} is the *uniform central scheduler* ρ_{UC}, which assigns the same probability $1/|P|$ to each $\{p\}$ $(p \in P)$.

configurations by $\mathit{\Gamma}_L$. We say that $\mathcal{S}$ is *weak stabilizing* for $\mathcal{SP}$ under σ, if any configuration γ has at least one execution starting from γ under σ that reaches a legitimate configuration. We say that a (possibly randomized) distributed system is *probabilistically stabilizing* for $\mathcal{SP}$ under (possibly probabilistic) scheduler σ, if any execution under σ reaches a legitimate configuration with probability 1.[5]

The performance of a self-stabilizing system is measured by the *convergence time*, which is the maximum number of time steps of any execution to a legitimate configuration. In the case of the probabilistic self-stabilization, we take the expectation of the convergence time.

In [8], it is shown that a distributed system $\mathcal{S} = (N, \mathcal{A})$ for a specification $\mathcal{SP}$ under σ_F is weak stabilizing, if and only if the corresponding randomized system $\mathcal{S} = (N, \mathcal{A}\ \ )$ is probabilistic self-stabilizing for $\mathcal{SP}$ under σ_S, where $\mathcal{A}\ \ = \langle \mathcal{A}, \mathcal{D}_{1/2} \rangle$. We can indeed choose any $0 < \delta < 1$ for $\mathcal{D}_\delta$. But this property does not hold for an "impure" probability distribution (not only under synchronous but also under a probabilistic scheduler), which is the reason we concentrate on pure probability distributions. Our motivation is to design a good probabilistic self-stabilizing algorithm from a weak stabilizing algorithm.

3 Finite Expected Convergence Times for Finite Systems

A probabilistic self-stabilizing system is useless if the expected convergence time is not bounded by a constant. This section investigates a necessary and sufficient condition for a weak stabilizing algorithm $\mathcal{A}$ to have a finite expected convergence time when $\mathcal{A}$ is suitably randomized by associating a pure probability distribution $\mathcal{D}$, for each classes of probabilistic schedulers $\rho_{OC}, \rho_O, \rho_C$ and ρ_F.

Let $\mathcal{A}$ be a weak stabilizing algorithm under σ_F for a specification $\mathcal{SP}$. Let $\mathcal{M}$ be a probabilistic scheduler instance in ρ. For any distributed system $\mathcal{S} = (N, \langle \mathcal{A}, \mathcal{D} \rangle)$ under $\mathcal{M}$, let $\tau_{\mathcal{D},\mathcal{M}}(\gamma_0, \omega_0)$ be the expected convergence time to a legitimate configuration when the initial configuration of $\mathcal{S}$ is $\gamma_0 \in \mathit{\Gamma}$ and the initial state of $\mathcal{M}$ is $\omega_0 \in \mathit{\Omega}$. Let

$$\tau_{\mathcal{D},\mathcal{M}}(\gamma_0) = \max_{\omega_0 \in \mathit{\Omega}} \tau_{\mathcal{D},\mathcal{M}}(\gamma_0, \omega_0),$$

and

$$\tau_{\mathcal{D},\mathcal{M}} = \max_{\gamma_0 \in \mathit{\Gamma}} \tau_{\mathcal{D},\mathcal{M}}(\gamma_0).$$

Recall that ρ is an adversary and must select the worst probabilistic scheduler instance $\mathcal{M}$ and the worst initial state $\omega_0 \in \mathit{\Omega}$. Then we want to know

$$\tau\ \ = \min_{\mathcal{D}} \max_{\mathcal{M} \in \rho} \tau_{\mathcal{D},\mathcal{M}}.$$

We denote by $\mathcal{D}\ \ = \mathrm{argmin}_{\mathcal{D}} \max_{\mathcal{M} \in \rho} \tau_{\mathcal{D}}$.

When $\mathcal{D}$ and $\mathcal{M}$ are given, we can compute $\tau_{\mathcal{D},\mathcal{M}}$ as follows: Since $\mathit{\Gamma}$ and $\mathit{\Omega}$ are finite, we compute $\tau_{\mathcal{D},\mathcal{M}}(\gamma_0, \omega_0)$ for each $\gamma_0 \in \mathit{\Gamma}$ and $\omega_0 \in \mathit{\Omega}$.

To this end, consider the direct product of two Markov chains $\mathcal{S}_{\mathcal{D}}$ and $\mathcal{M}$, i.e., Markov chain $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$, whose state spaces is $\mathit{\Gamma} \times \mathit{\Omega}$. Its transition probability from (γ, ω) to (γ', ω') is obviously computable. Then we contract all states (γ, ω) such that $\gamma \in \mathit{\Gamma}_L$ into a newly introduced state γ_L (which represents all legitimate configurations), and calculate the hitting time from (γ_0, ω_0) to γ_L. This Markov chain (with labeling λ) is denoted by $\mathcal{G}$, and $G = (V, E)$ denotes its transition graph. We make use of $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$ as well as $\mathcal{G}$ in the following.

The *hitting time* $HT_{i,j}$ of a Markov chain is the number of steps that a stochastic process starting from state i takes until it reaches state j for the first time; $HT_{i,j} \equiv \min\{t > 0 : X_t = j | X_0 = i\}$. The mean hitting time $ht_{i,j}$ is $\mathrm{E}[HT_{i,j}]$.

[5] Let $\mathcal{E} = \gamma_0, \gamma_1, \ldots, \gamma_k$ be a finite execution. Intuitively, the probability of a finite execution is the product of the probability $p_{i,i+1}$ for $i = 0, 1, \ldots, k-1$, where $p_{i,i+1}$ is the probability that γ_{i+1} is reached provided γ_i, which also depends on the probabilistic scheduler instance and its current state (see Section 3 for detailed argument). Now using the set of all the finite executions as the sample space, we can define a probability model for this purpose. See e.g., [3] for the formal definition of the probability model.

3.1 τ^* under ρ_{OC}

Let $\mathcal{S} = (N, \mathcal{A})$ be a weak stabilizing system for a specification $\mathcal{SP}$ under σ_F. We show a necessary and sufficient condition for a randomized system $S_{\mathcal{D}} = (N, \langle \mathcal{A}, \mathcal{D} \rangle)$ to have a bounded worst case expected convergence time τ $(< \infty)$ for some pure probability distribution $\mathcal{D}$.

For the transition diagram $\mathcal{S} = (\Gamma, T, \lambda)$, we contract all legitimate configurations to a newly introduced configuration γ_L. Let $\widehat{S}$ be the obtained transition diagram. For process $p \in P$, let $\widehat{T}_p$ be the set of edges labeled with $\{p\}$, and $\widehat{S}_p = (\Gamma, \widehat{T}_p)$.

Definition 1. *A transition diagram $\mathcal{S} = (\Gamma, T, \lambda)$ satisfies the regularity condition if $\widehat{S}_p$ is a rooted in-tree rooted at γ_L for any $p \in P$.*

In this section, we show that the regularity is a necessary and sufficient condition for τ $< \infty$ under ρ_{OC}.

Theorem 1. *$\mathcal{S}$ satisfies the regularity condition if and only if τ $< \infty$ under ρ_{OC}.*

Since $\mathcal{D}$ is pure, the transition diagram of a distributed system $(N, \langle \mathcal{A}, \mathcal{D} \rangle)$ is the same as the transition diagram of $\mathcal{S}$, except that each directed edge is associated with a probability. Let us construct a Markov chain $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$ for any probabilistic scheduler instance $\mathcal{M} \in \rho_{OC}$. Since Ω is a singleton, the transition graph of $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$ is isomorphic to that of $\mathcal{S}$, except the difference of the transition probability matrix. Recall that each of its edges (i, j) of $\mathcal{S}$ has a label $\lambda(i, j) \subseteq P$, as well as a probability $P_{i,j}$.

Let $\mathcal{G} = (V, E)$ be the Markov chain obtained from $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$. Let (V_1, V_2) be any vertex cut of $\mathcal{G}$ such that $\gamma_L \in V_2$. We denote the directed edges that cross from V_1 to V_2 by $E(V_1, V_2) = \{(v, v') \in E | v \in V_1, v' \in V_2\}$, and let $P(V_1, V_2)$ be the set of all labels $X_{i,j} \subset P$ associated with the edges (i, j) in $E(V_1, V_2)$ such that the associated probability $Q_{i,j}$ is positive. Since $\mathcal{M} \in \rho_{OC} \subseteq \rho_C$, $Q_{i,j} > 0$ only if $X_{i,j}$ is a singleton.

Lemma 1. *For any distributed system $\mathcal{S} = (N, \mathcal{A})$, if there is a cut (V_1, V_2) such that $P(V_1, V_2) \neq \{\{p\} : p \in P\}$ in $\mathcal{G}$, then τ $= \infty$.*

Proof. Suppose that there exists a cut (V_1, V_2) such that $P(V_1, V_2) \neq \{\{p\} : p \in P\}$ and let $\{p\} \notin P(V_1, V_2)$. Consider a probabilistic scheduler instance $\mathcal{M}$ in ρ_{OC} that assigns probability $(1-\epsilon)$ to $\{p\}$ for arbitrary small ϵ. For any execution of $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$ starting from a configuration in V_1, the expected number of steps necessary to cross this cut is ϵ^{-1}. Hence the maximum convergence time is at least ϵ^{-1}. For any $\mathcal{D}$, $\mathcal{M}$ makes τ arbitrarily large. $\square$

In order for τ to have a finite value, for any cut (V_1, V_2), $P(V_1, V_2) = \{\{p\} : p \in P\}$ must hold, which implies that in any configuration γ, all processes in P are enabled unless $\gamma \in \Gamma_L$.

Let E_p be the set of edges in G labeled with $\{p\}$. From Lemma 1, for any $p \in P$, the subgraph $G_p = (V, E_p)$ is 1-regular in the sense that for any state except γ_L, the out degree is exactly one. Therefore G_p forms a rooted in-tree rooted at γ_L if and only if it is weakly connected. Otherwise, G_p consists of multiple connected components, and we have τ $= \infty$ by taking a cut that separates these connected components. We thus obtain the next lemma.

Observation 1 *If G_p is not a rooted in-tree for some $p \in P$, then τ $= \infty$.*

Because the assumption that $\mathcal{D}$ is pure and the procedure to obtain $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$ from $\mathcal{M}$, the regularity condition of $\mathcal{S}$ is equivalent to the condition that G_p is a rooted in-tree rooted at γ_L. Hence, we proved that regularity is a necessary condition for τ $< \infty$.

In the following, we show that regularity condition is sufficient for τ to be finite under ρ_{OC}. For any pure probability distribution $\mathcal{D}$ and probabilistic scheduler instance $\mathcal{M} \in \rho_{OC}$, consider $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$, and then construct $\mathcal{G}$. An execution in $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$, is a Markov chain in $\mathcal{G}$. Let X_t $(t = 0, 1, \ldots)$ be a random variable that represents the configuration of this execution at time t with $X_0 = \gamma_0$.[6] The convergence time t of this execution is then the hitting time from γ_0 to γ_L.

[6] Although the state space of $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$ is $\Gamma \times \Omega$, we identify (γ, ω) with γ, since Ω is a singleton.

Lemma 2. *Let* $\gamma \ = \mathrm{argmax}_{\gamma \in V} \Pr\left[HT_{\gamma,\gamma_L} > t\right]$. *For any* $\gamma \in V$, $\Pr\left[HT_{\gamma,\gamma_L} > t\right]$ *has the sub-multiplicativity:*

$$\Pr\left[HT_{\gamma,\gamma_L} > 2t\right] \leq \left(\Pr\left[HT_{\gamma \ ,\gamma_L} > t\right]\right)^2.$$

Proof. Let $X_0 = \gamma_0, X_1, \ldots$ be a Markov chain on $\mathcal{G}$. From the conditional probability, we have

$$\begin{aligned}
&\Pr\left[HT_{\gamma,\gamma_L} > 2t\right] \\
&= \sum_{\gamma' \in V} \Pr\left[X_t = \gamma' \wedge HT_{\gamma,\gamma_L} > t\right] \cdot \Pr\left[HT_{\gamma',\gamma_L} > t \mid X_t = \gamma' \wedge H_{\gamma,\gamma_L} > t\right] \\
&= \sum_{\gamma' \in V} \Pr\left[(HT_{\gamma',\gamma_L} > t) \wedge (X_t = \gamma' \wedge H_{\gamma,\gamma_L} > t)\right] \\
&\leq \sum_{\gamma' \in V} \Pr\left[(HT_{\gamma \ ,\gamma_L} > t) \wedge (X_t = \gamma' \wedge H_{\gamma,\gamma_L} > t)\right] \\
&= \Pr\left[HT_{\gamma \ ,\gamma_L} > t\right] \cdot \sum_{\gamma' \in V} \Pr\left[X_t = \gamma' \wedge H_{\gamma,\gamma_L} > t\right] \\
&\leq \Pr\left[HT_{\gamma \ ,\gamma_L} > t\right] \cdot \Pr\left[H_{\gamma,\gamma_L} > t\right] \leq \left(\Pr\left[HT_{\gamma \ ,\gamma_L} > t\right]\right)^2.
\end{aligned}$$

The fourth and the last lines are obtained from the definition of γ . □

Lemma 3. *$\mathcal{S}$ satisfies the regularity condition only if $\tau \ < \infty$ under ρ_{OC}.*

Proof. We show

$$\infty > \tau_{\mathcal{D}_1} = \max_{\mathcal{M} \in \rho_{OC}} \tau_{\mathcal{D}_1,\mathcal{M}} \geq \tau \ .$$

For any probabilistic scheduler instance $\mathcal{M} \in \rho_{OC}$, consider $\mathcal{S}_{\mathcal{D}}^{\mathcal{M}}$, and then construct $\mathcal{G}$. Let ϵ be the maximum probability that is assigned to a transition of $\mathcal{M}$ and $\{p\}$ be the label of the transition. Because Γ is finite, the probability that $\mathcal{G}$ reaches γ_L by tracing G_p is no less than ϵ^h, where h is the height of G_p. We have $\Pr\left[HT_{\gamma,\gamma_L} \leq h\right] \geq \epsilon^h$. Let $\delta = 1 - \epsilon^h$, then

$$\Pr[HT_{\gamma,\gamma_L} > h] < 1 - \epsilon^h = \delta.$$

Note that $\mathcal{M} \in \rho_{OC}$ cannot make ϵ arbitrarily small, and indeed $\epsilon \geq 1/|P|$.

For any $\mathcal{M} \in \rho_{OC}$, let τ be the time that an execution starting from $\gamma \ = \mathrm{argmax}_{\gamma \in \Gamma} \tau_{\mathcal{D}_1,\mathcal{M}}(\gamma)$, takes until it reaches γ_L. We have

$$\begin{aligned}
\tau_{\mathcal{D}_1,\mathcal{M}} &= \sum_{i=0}^{\infty} i \cdot \Pr[\tau = i] = \sum_{i=1}^{\infty} \Pr[\tau \geq i] \\
&= \sum_{i=1}^{h} \Pr[\tau \geq i] + \sum_{i=h+1}^{2h} \Pr[\tau \geq i] + \sum_{i=2h+1}^{3h} \Pr[\tau \geq i] + \sum_{i=3h+1}^{\infty} \Pr[\tau \geq i] \\
&\leq \sum_{i=1}^{h} \Pr[\tau \geq i] + \sum_{i=h+1}^{2h} \Pr[\tau > h] + \sum_{i=2h+1}^{3h} \Pr[\tau > 2h] + \sum_{i=3h+1}^{\infty} \Pr[\tau \geq i] \\
&\leq h + h\delta + h\delta^2 + \sum_{i=3h+1}^{\infty} \Pr[\tau \geq i] \\
&\leq h + h\delta + h\delta^2 + \cdots = h\frac{1}{1-\delta} < \frac{h}{\epsilon^h} < \infty.
\end{aligned}$$

□

The, we have the proof for Theorem 1 with Lemma 1 for the if part, and Lemma 3 for the only-if part.

3.2 τ^* under ρ_O, ρ_C or ρ_F

We next consider ρ_O whose instances $\mathcal{M}$ may activate any subset $X \subseteq P$, i.e., $\mathcal{M}$ may activate more than one processes at a time. In Theorem 1, we use $\mathcal{D}_1$ to show $\tau \ < \infty$. Here we use $\mathcal{D}_{1/|P|}$ to this end.

Theorem 2. *$\mathcal{S}$ satisfies the regularity condition if and only if $\tau \ < \infty$ under ρ_O.*

Proof. The If-part follows from $\rho_{OC} \subset \rho_O$. If $\mathcal{S}$ does not satisfy the regularity condition, then there is a cut for which there exists an $\mathcal{M} \in \rho_{OC} \subset \rho_O$ that makes $\tau \ = \infty$ by Lemma 1.

We prove the only-if part by showing

$$\infty > \tau_{\mathcal{D}_{1/|P|}} = \max_{\mathcal{M} \in \rho_O} \tau_{\mathcal{D}_{1/|P|}, \mathcal{M}} \geq \tau \ .$$

We show a positive lower bound on the probability that any execution reaches γ_L in some fixed steps. Then by using the same argument in the proof of Theorem 1, we can conclude $\tau \ < \infty$. Note that $\mathcal{D}_{1/|P|}$ adds self-loops to each configuration, which cause some slowdown, but its amount is finite.

Consider any probabilistic scheduler instance $\mathcal{M} \in \rho_O$. Let ϵ be the maximum probability that $\mathcal{M}$ assigns to a transition, and let $X \subseteq P$ be the label associated with this transition. Hence $\epsilon \geq 1/2^{|P|}$. By using $\mathcal{D}_{1/|P|}$, we can produce a singleton sequence for some $p \in X$.

The probability that only p is executed (as the result of the random choice by $\mathcal{D}_{1/|P|}$), when X is activated by $\mathcal{M}$ is $(1/|P|)(1-1/|P|)^{|X|-1}$. Let h be the height of G_p. Then the probability that h consecutive executions of p occurs is $(\epsilon(1/|P|)(1-1/|P|)^{|X|-1})^h$. Hence the probability that an execution reaches γ_L in h steps is no less than a constant (in $\mathcal{M}$) $(1/2^n(1/n)(1-1/n)^{n-1})^h = ((1-1/n)^{n-1}/(n2^n))^h$. □

We next consider ρ_C. There is a deterministic distributed system such that $\tau \ = \infty$ holds if the scheduler is not oblivious, even if it is central, i.e., under ρ_C.

Consider a distributed system $\mathcal{S}$ shown in Figure 1. It executes an algorithm $\mathcal{A}$ on $N = (P, L)$, where $P = \{p, q\}$ and $L = \{(p, q), (q, p)\}$. Process p (resp. q) has a variable v_p (reps. v_q) that takes an integer in $\{0, 1, 2, 3\}$. The legitimate configurations are those satisfying $v_p = v_q \in \{1, 2, 3\}$. The transitions of $\mathcal{S}$ is represented by the state machine shown in Figure 2, where s_4 corresponds to the legitimate configurations.

Let $\mathcal{M} \in \rho_C$ be a probabilistic scheduler instance with $\Omega = \{1, 2\}$. Its transition probabilities are defined by $P_{1,2} = P_{2,1} = 1 - \epsilon$ and $P_{1,} = P_{2,2} = \epsilon$. The transitions $(1, 2)$ and $(2, 2)$ are labeled with $\{p\}$, and $(2, 1)$ and $(1, 1)$ are labeled with $\{q\}$. If the initial configuration is $v_p = v_q = 0$, the expected convergence time can be arbitrarily large by taking arbitrarily small ϵ, that makes $\mathcal{M}$ outputs $\{q\}\{p\}\{q\}\{p\}\{q\}\ldots$ with arbitrarily high probability. To overcome this problem, we use $\mathcal{D}_{1/2}$ to ignore some activations.

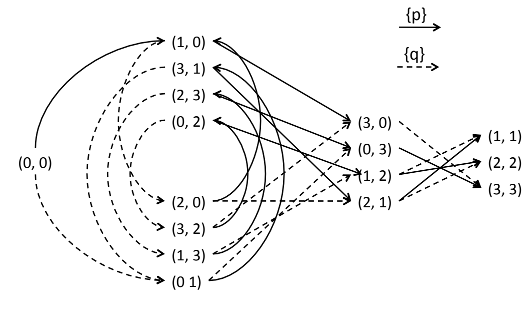


Fig. 1. The transition diagram of $\mathcal{S}$. (Each pair represents (v_p, v_q).)

Theorem 3. *$\mathcal{S}$ satisfies the regularity condition if and only if $\tau \ < \infty$ under ρ_C.*

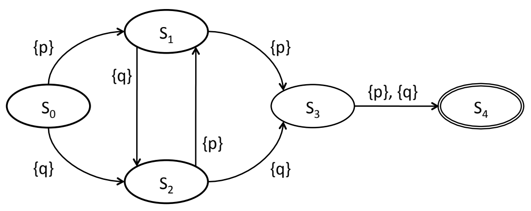


Fig. 2. The state machine corresponding to $\mathcal{S}$

Proof. Since $\rho_O C \subseteq \rho_O$, the If-part holds by the same argument as in the proof of Theorem 2.

We prove the only-if part by showing

$$\infty > \tau_{\mathcal{D}_{1/2}} = \max_{\mathcal{M} \in \rho_C} \tau_{\mathcal{D}_{1/2}, \mathcal{M}} \geq \tau \ .$$

As in the proof of Theorem 2, we show a positive lower bound on the probability that any execution reaches γ_L in some fixed steps. Then by using the same argument in the proof of Theorem 1, we can conclude $\tau \ < \infty$.

Let $\mathcal{Q} = (Q_1, Q_2, \ldots, Q_k)$ be the strongly connected components of the transition graph of $\mathcal{M}$, and let $\mathcal{Q}' \subseteq \mathcal{Q}$ be the set of sink components in $\mathcal{Q}$. Then any Markov chain reaches some sink component in a finite time. Suppose that $\mathcal{M}$ reaches $Q_i \in \mathcal{Q}$.

Let ϵ be the maximum probability assigned to an transition in Q_i and let $X \subseteq P$ be the label of this transition. Since the scheduler is central, X is a singleton and $\epsilon \geq 1/|P|$. By using $\mathcal{D}_{1/2}$, we can probabilistically produce a sufficiently long sequence of a singleton $\{p\}$ for some process $p \in P$, by making processes $q \neq p$ to refuse activations, and discarding all their activations between two activations of $\{p\}$. Since Q_i is a sink component of finite size, all states are positive recurrent, and this transition is repeated, in average, every a constant number of steps, say t_p. Without loss of generality, we assume that p is such that t_p is maximized, thus $t_p \geq 1/n$.

Let h be the height of G_p. The probability that the execution follows G_p to reach γ_L in $t \cdot h$ steps is greater than or equal to $\left(\epsilon((1/2)^{2^{|P|}})^{t_p}\right)^h$, which is a constant in $\mathcal{M}$ □

As for ρ_F, by following the proofs of Theorems 2 and 3 we obtain the following theorem.

Theorem 4. *$\mathcal{S}$ satisfies the regularity condition if and only if $\tau \ < \infty$ under ρ_F.*

4 Finite Expected Convergence Times of Infinite Systems

In this section, we investigate a necessary and sufficient condition for a infinite system under a finite probabilistic scheduler to have a finite τ . When there is a local variable with an infinite domain, then Γ is infinite.

The sufficient condition for $\tau \ < \infty$ given in Section 3 depends on the fact that the height of rooted in-tree G_p is finite for each $p \in P$. However, in an infinite system, the height of G_p may be infinite.

Let $\mathcal{S} = (N, \mathcal{A})$ be a weak stabilizing infinite system under σ_F. The following lemma promises that the height of G_p is finite even in a infinite system if $\tau \ < \infty$. Let h_p be the height of G_p for $p \in P$.

Lemma 4. *$\mathcal{S}$ satisfies the regularity condition and h_p is finite for each $p \in P$, if $\tau \ < \infty$ under ρ_{OC}.*

Proof. If there exists a cut (V_1, V_2) in G such that V_2 contains γ_L, and $\{p\} \notin P(V_1, V_2)$ for some $p \in P$, then there exists a probabilistic scheduler instance in ρ_{OC} that assigns arbitrarily small probability ϵ to each process $q \in P \setminus \{p\}$. Then the expected time to cross this cut becomes arbitrarily large when ϵ approaches to 0. Hence, if $\tau \ = \infty$, then $\mathcal{S}$ has the regularity.

Suppose, for some $q \in P$, $h_q = \infty$. Let $Y = (\dots, \gamma', \gamma'', \dots, \gamma_L)$ be an infinite directed path to γ_L in G_q. The transitions from $\gamma \in Y$ are labeled with processes in $P \setminus \{q\}$. Hence, there exists a probabilistic scheduler instance $\mathcal{M} \in \rho_{OC}$ that outputs $\{q\}$ with probability $1-\epsilon$. When ϵ approaches to 0, the executions starting from a configuration $\gamma \in Y$ traces the postfix of Y with arbitrarily high probability, and τ becomes arbitrarily large. It is a contradiction and G_p of each $p \in P$ is of finite height. □

Thus all discussions in Section 3 hold in the infinite systems.

Theorem 5. *$\mathcal{S}$ satisfies the regularity condition and h_p is finite for each $p \in P$, if and only if $\tau < \infty$ under ρ_{OC}.*

Proof. If part is by Lemma 4, and the only-if part follows the proof of Theorem 1.

From the same discussion as in Section 3, we have the following theorem.

Theorem 6. *$\mathcal{S}$ satisfies the regularity condition and for each h_p is finite for each $p \in P$, if and only if $\tau < \infty$ under ρ_O, ρ_C, and ρ_F.*

5 Conclusion

We investigated the power of algorithm randomization against adversarial yet probabilistic schedulers. We presented necessary and sufficient conditions for finite and infinite probabilistically stabilizing systems to exhibit finite expected stabilization time. Except for oblivious central schedulers, algorithm randomization is necessary to guarantee finite expected stabilization time. Two important open questions are raised by our work:

1. On the theoretical side, it is worth investigating the question of optimal randomization of an algorithm, in order to obtain the minimum expected stabilization time, for a given probabilistic scheduler distribution.
2. On the practical side, it would be interesting to collect execution metrics for actual networks and derive realistic probabilistic scheduler distributions.

References

1. Joffroy Beauquier, Stéphane Cordier, and Sylvie Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proceedings on the Workshop on Self-stabilizing Systems*, pages 15.1–15.15, 1995.
2. Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1):75–93, January 2007.
3. Joffroy Beauquier, Colette Johnen, and Stéphane Messika. All k-bounded policies are equivalent for self-stabilization. In Ajoy Kumar Datta and Maria Gradinariu, editors, *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 82–94. Springer, 2006.
4. Samuel Bernard, Stéphane Devismes, Katy Paroux, Maria Potop-Butucaru, and Sébastien Tixeuil. Probabilistic self-stabilizing vertex coloring in unidirectional anonymous networks. In *Proceedings of ICDCN 2010*, volume 5935 of *Lecture Notes in Computer Science*, pages 167–177, Kolkata, India, January 2010. Springer Berlin / Heidelberg.
5. James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.
6. Ajoy K. Datta, Maria Gradinariu, and Sébastien Tixeuil. Self-stabilizing mutual exclusion with arbitrary scheduler. *The Computer Journal*, 47(3):289–298, October 2004.
7. Ajoy Kumar Datta and Ted Herman, editors. *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, volume 2194 of *Lecture Notes in Computer Science*. Springer, 2001.
8. Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, Beijin, China, June 2008.

9. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
10. Shlomi. Dolev. *Self-stabilization.* MIT Press, March 2000.
11. Shlomi Dolev and Ted Herman. Dijkstra's self-stabilizing algorithm in unsupportive environments. In Datta and Herman [7], pages 67–81.
12. Shlomi Dolev, Amos Israeli, and Shlomo Moran. Analyzing expected time by scheduler-luck games. *IEEE Trans. Software Eng.*, 21(5):429–439, 1995.
13. Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76(8):884–900, 2010.
14. Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.
15. Philippe Duchon, Nicolas Hanusse, and Sébastien Tixeuil. Optimal randomized self-stabilizing mutual exclusion in synchronous rings. In *Proceedings of the 18th Symposium on Distributed Computing (DISC 2004)*, number 3274 in Lecture Notes in Computer Science, pages 216–229, Amsterdam, The Nederlands, October 2004. Springer Verlag.
16. Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized finite-state distributed algorithms as markov chains. In Jennifer L. Welch, editor, *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2001.
17. Mitchell Flatebo and Ajoy Kumar Datta. Simulation of self-stabilizing algorithms in distributed systems. In *Annual Simulation Symposium*, pages 32–41. IEEE Computer Society, 1992.
18. Laurent Fribourg, Stéphane Messika, and Claudine Picaronny. Coupling and self-stabilization. *Distributed Computing*, 18(3):221–232, 2006.
19. Mohamed G. Gouda. The theory of weak stabilization. In Datta and Herman [7], pages 114–123.
20. Maria Gradinariu and Sébastien Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *International Conference on Principles of Distributed Systems (OPODIS'2000)*, pages 55–70, Paris, France, December 2000.
21. Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2007)*, page 46. IEEE, June 2007.
22. Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.
23. Ted Herman. Self-stabilization: ramdomness to reduce space. *Information Processing Letters*, 6:95–98, 1992.
24. Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.
25. Toshimitsu Masuzawa and Sébastien Tixeuil. On bootstrapping topology knowledge in anonymous networks. *ACM Transactions on Adaptive and Autonomous Systems (TAAS)*, 4(1), January 2009.
26. Toshimitsu Masuzawa and Sébastien Tixeuil. Stabilizing locally maximizable tasks in unidirectional networks is hard. In *Procedings of IEEE ICDCS 2010.* IEEE Computer Society, June 2010.
27. Nathalie Mitton, Bruno Séricola, Sébastien Tixeuil, Eric Fleury, and Isabelle Guérin-Lassous. Self-stabilization in self-organized multihop wireless networks. *Ad Hoc and Sensor Wireless Networks*, 11(1-2):1–34, January 2011.
28. N. Mullner, A. Dhama, and O. Theel. Derivation of fault tolerance measures of self-stabilizing algorithms by simulation. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 183 –192, april 2008.
29. Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in population protocol model. In Shay Kutten and Janez Zerovnik, editors, *SIROCCO*, volume 5869 of *Lecture Notes in Computer Science*, pages 295–308. Springer, 2009.
30. Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.
31. Sally K. Wahba, Jason O. Hallstrom, Pradip K. Srimani, and Nigamanth Sridhar. *SFS*^{3}: a simulation *f*ramework for *s*elf-*s*tabilizing *s*ystems. In Robert M. McGraw, Eric S. Imsand, and Michael J. Chinni, editors, *SpringSim*, page 172. SCS/ACM, 2010.

非同期リング上における
モバイルエージェントの部分集合アルゴリズム

柴田 将拡 [†], 川合 慎治 [†], 大下 福仁 [†], 角川 裕次 [†], 増澤 利光 [†]

[†] 大阪大学 大学院情報科学研究科,565-0871, 大阪府吹田市山田丘 1-5
{m-sibata, s-kawai, f-oosita, kakugawa, masuzawa}@ist.osaka-u.ac.jp

Abstract

In this paper, we consider the partial gathering problem of mobile agents in asynchronous rings, which requires, for a given input g, that each agent should move to a node and terminates so that at least g agents should meet at the same node. The requirement for the partial gathering is weaker than that for the ordinary (total) gathering, and thus, we have interests in clarifying the di erence on the move complexity between them. We propose two algorithms to solve the partial gathering problem. One algorithm is deterministic and assumes unique ID of each agent. The other is randomized and assumes anonymous agents. The deterministic (resp., randomized) algorithm achieves the partial gathering in $O(gn)$ (resp., expected $O(gn + n \log k)$) total number of moves where n is the ring size and k is the number of agents, while the total gathering requires $\Omega(kn)$ moves. We show that the move complexity of the deterministic algorithm is asymptotically optimal.
keyword: distributed system, mobile agent, gathering problem, partial gathering

1 Introduction

1.1 Background and our contribution

A *distributed system* is a system that consists of a set of computers (*nodes*) and communication links In recent years, distributed systems have become large and design of distributed systems has become complicated. As a way to design ef- cient distributed systems, (mobile) agents have attracted a lot of attention [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Agents simplify design of distributed systems because they can traverse the system and process tasks on each node.

The gathering problem is a fundamental problem for cooperation of agents [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. The gathering problem requires all agents to meet at a single node in nite time. The gathering problem is useful because, by meeting at a single node, all agents can share information or synchronize behaviors among them.

In this paper, we consider a new variant of the gathering problem, called the *partial gathering problem.* The partial gathering problem does not always require all agents to gather at a single node, but requires agents to gather partially at several nodes. More precisely, we consider the problem which requires, for given input g, that each agent should move to a node and terminate so that at least g agents should meet at the same node. We de ne this problem as the *g-partial gathering problem.* Clearly, if $\frac{1}{2}k < g \quad k$ holds, the g-partial gathering problem is equal to the ordinary gathering problem. If $g \quad \frac{1}{2}k$ holds, the requirement for the g-partial gathering problem is weaker than that for the ordinary gathering problem, and thus it seems possible to solve the g-partial gathering problem with a smaller total number of moves. In addition, the g-partial gathering problem is still useful because agents can share information and process tasks

Table 1: Proposed algorithms for the g-partial gathering problem in asynchronous unidirectional rings.

Model	Algorithm 1	Algorithm 2
Unique ID	Available	Not available
Deterministic/Randomized	Deterministic	Randomized
Knowledge of k	Not available	Available
The total number of moves	$O(gn)$	$O(n \log k + gn)$

cooperatively among at least g agents.

The contribution of this paper is to clarify the di erence on the move complexity between the gathering problem and the g-partial gathering problem. We consider the g-partial gathering problem in asynchronous unidirectional rings. The contribution of this paper is summarized in Table 1.1. First, we propose a deterministic algorithm to solve the g-partial gathering problem for the case that agents have distinct IDs. This algorithm requires $O(gn)$ total number of moves. Second, we propose a randomized algorithm to solve the g-partial gathering problem for the case that agents have no IDs and agents know the number of agents. This algorithm requires $O(n \log k + gn)$ total number of moves, while the total gathering requires $\Omega(kn)$ moves. The two algorithms imply that the g-partial gathering problem can be solved in a smaller total number of moves compared to the ordinary (total) gathering problem for both cases. In addition, we show that the total number of moves is $\Omega(gn)$ for the g-partial gathering problem. This means the rst algorithm is asymptotically optimal in terms of the total number of moves.

1.2 Related works

Many fundamental problems for cooperation of mobile agents have been studied in literature. For example, the searching problem [7, 8], the gossip problem [9], the election problem [10], and the gathering problem [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] have been studied.

In particular, the gathering problem has received a lot of attention. The gathering problem has been extensively studied in many topologies, which include trees [1, 9], tori [1, 5], and rings [1, 2, 3, 4, 6, 7, 8, 9, 10, 11]. The gathering problem for rings has been extensively studied because algorithms for such highly symmetric topologies give techniques to treat the essential di culty of the gathering problem such as breaking symmetry. Actually, to solve the gathering problem, it is necessary to select exactly one gathering node, i.e., a node where all agents meet. There are many ways to select the gathering node. For example, in [1, 2, 3, 4, 5, 6], agents leave marks (tokens) on their initial nodes and select the gathering node based on every distance of neighboring tokens. In [7, 8], agents have distinct IDs and select the gathering node based on the IDs. In [11], agents can use random numbers and select the gathering node based on IDs generated randomly. In [1, 9, 10], agents execute the leader agent election and the elected leader decides the gathering node.

2 Preliminaries

2.1 Network model

A *unidirectional ring network* R is a tuple $R = (V, L)$, where V is a set of nodes and L is a set of communication links. We denote by n $(= |V|)$ the number of nodes. Then, ring R is de ned as follows.

- $V = \{v_0, v_1, \dots, v_{n\ \ 1}\}$
- $L = \{v_i, v_{(i+1) \bmod n} \mid 0 \quad i \quad n \quad 1\}$

We de ne the direction from v_i to v_{i+1} as a *forward* direction, and the direction from v_{i+1} to v_i as a *backward* direction.

In this paper, we assume nodes are anonymous, i.e., each node has no ID. Every node

$v_i \in V$ has a whiteboard and agents on node v_i can read from and write to the whiteboard of v_i. We de ne W as a set of all states of a whiteboard.

2.2 Agent model

Let $A = \{a_1, a_2, \dots, a_k\}$ be a set of agents. We consider two model variants.

In the rst model, we consider agents that are distinct (i.e., agents have distinct IDs) and execute a deterministic algorithm. We model an agent as an identical nite automaton $(S, \ , s_{initial}, s_{final})$. The rst element S is the set of all states of agents, which includes initial state $s_{initial}$ and nal state s_{final}. The second element is the state transition function. Since we treat deterministic algorithms, is described as $: S \quad W \rightarrow S \quad W \quad M$,

where $M = \{1, 0\}$ represents whether the agent makes a movement or not in the step. The value 1 represents movement to the next node and 0 represents stay at the current node. Since rings are unidirectional, each agent only moves to its forward node. We assume that agents move instantaneously, that is, agents always exist at nodes (do not exist at links). Moreover, we assume that each agent cannot detect the number of agents on its current node.

In the second model, we consider agents that are anonymous (i.e., agents have no IDs) and execute a randomized algorithm. We model an agent similarly to the rst model except for state transition function . Since we treat randomized algorithms, is described as $: S \quad W \quad R \rightarrow S \quad W \quad M$, where R represents a set of random values. In addition, we assume that each agent knows the number of agents.

2.3 System con guration

In an agent system, (global) *con guration* c is de ned as a product of states of agents, states of nodes (whiteboards), and locations of agents. We de ne C as a set of all con gurations. In initial con guration $c_0 \in C$, we assume that no pair of agents stay at the same node. We assume that each node v_j has variable $v_j.initial$ that indicates existence of agents in the initial con guration. If there exists an agent on node v_j in the initial con guration, the value of $v_j.initial$ is one. Otherwise, the value of $v_j.initial$ is zero.

Let A_i be an arbitrary non-empty set of agents. When con guration c_i changes to c_{i+1} by the action of every agent in A_i, we denote the transition by $c_i \xrightarrow{A_i} c_{i+1}$. When $a_j \in A_i$ moves to the next node or changes some states (of its own or the whiteboard), we say that agent a_j takes one step. If multiple agents at the same node are included in A_i, the agents take steps simultaneously. When $A_i = A$ holds for any i, all agents perform simultaneously. This model is called the *synchronous model*. Otherwise, the model is called the *ashynchronous model*.

If sequence of con gurations $E = c_0, c_1, \dots$ satis es $c_i \xrightarrow{A_i} c_{i+1}$ $(i \quad 0)$, E is called an *execution* starting from c_0. Execution E is in nite, or ends in nal con guration c_{final} where no agent can take a step.

2.4 Partial gathering problem

The requirement of the partial gathering problem is that, for a given input g, each agent should move to a node and terminate so that at least g agents should meet at the node. Formally, we de ne the g-partial gathering problem as follows.

De nition 2.1. *Execution E solves the g-partial gathering problem when the following conditions hold:*

- *Execution E is nite.*
- *In the nal con guration, for any node v_j such that there exist some agents on v_j, there exist at least g agents on v_j.*

For the g-partial gathering problem, we have the following lower bound.

Theorem 2.1. *The total number of moves required to solve the g-partial gathering problem is $\Omega(gn)$.*

Proof. We consider an initial con guration such that all agents are scattered evenly. We assume $n = ck$ holds for some positive integer c. Let V' be the set of nodes where agents exist in the nal con guration, and let $x = |V'|$. Since at least g agents meet at v_j for any $v_j \in V'$, we have $k \quad gx$.

For each $v_j \in V'$, we de ne A_j as the set of agents that meet at v_j. Then, among agents in A_j, the i-th smallest number of moves to get to v_j is at least $(i \quad 1)n/k$. We de ne A_j^S as the set of g agents such that the number of moves is the smallest, and $A_j^L = A_j \quad A_j^S$. Let T_j^S and T_j^L be the total number of moves of agents in A_j^S and A_j^L respectively. Then, we have

$$T_j^S \quad \sum_{i=1}^{g}(i \quad 1) \quad \frac{n}{k} = \frac{n}{k} \quad \frac{g(g \quad 1)}{2}$$

and

$$T_j^L \quad |A_j^L| \quad \frac{gn}{k}.$$

Therefore, the total number of moves is at least

$$\begin{aligned} T &= \sum_{v_j \in V'} (T_j^S + T_j^L) \\ & \quad x \quad \frac{n}{k} \quad \frac{g(g \quad 1)}{2} + | \bigcup_{v_j \in V'} A_j^L | \quad \frac{gn}{k} \\ & \quad x \quad \frac{n}{k} \quad \frac{g(g \quad 1)}{2} + (k \quad gx) \quad \frac{gn}{k} \\ &= gn \quad \frac{xng}{2k}(g+1). \end{aligned}$$

Since $k \quad gx$ holds, we have

$$T \quad \frac{n}{2}(g \quad 1).$$

Thus, the total number of moves is at least $\Omega(gn)$. □

3 A Deterministic Algorithm for Distinct Agents

In this section, we propose a deterministic algorithm to solve the g-partial gathering problem for distinct agents (i.e., agents have distinct IDs). The basic idea to solve the g-partial gathering is that agents select a leader and then the leader instructs other agents which node they meet at. However, since $\Omega(n \log k)$ total number of moves is required to elect one leader [9], it is impossible to solve the g-partial gathering in asymptotically optimal total number of moves (i.e., $O(gn)$). To overcome this lower bound, we select multiple agents as leaders by executing leader agent election partially. By this behavior, our algorithm solves the g-partial gathering problem in $O(gn)$ total number of moves.

The algorithm consists of two parts. In the rst part, agents execute leader agent election partially and elect some leader agents. In the second part, leader agents instruct the other agents which node they meet at, and the other agents move to the node by the instruction.

3.1 The rst part

The aim of the rst part is to elect leaders that satisfy the following properties: 1) At least one agent is elected as a leader, 2) At most $\lfloor k/g \rfloor$ agents are elected as leaders, and 3) There exist at least $g \quad 1$ non-leader agents between two leader agents. To attain this goal, we use a traditional leader election algorithm [12]. However the algorithm in [12] is executed by nodes and the goal is to elect exactly one leader. So we modify the algorithm to be executed by agents, and then agents elect at most $\lfloor k/g \rfloor$ leader agents by executing the algorithm partially.

During the execution of leader election, the states of agents are divided into the following three types:

- *active*: The agent is performing the leader agent election as a candidate of leaders.
- *inactive*: The agent has dropped out from the candidate of leaders.
- *leader*: The agent has been elected as a leader.

First, we explain the idea of leader election by assuming that the ring is bidirectional. The algorithm consists of several phases. In each phase, each active agent compares its own ID with IDs of its left and right neighbor active agents. More concretely, each active agent writes its ID on the whiteboard of its current node, and then moves forward and backward to observe IDs of the forward and backward active agents. If its own ID is the smallest among the three agents, the agent remains active as a candidate of leaders. Otherwise, the agent drops out from candidates of leaders and becomes inactive. By doing this, at least half active agents become inactive in each phase. Consequently, after executing $\lceil \log g \rceil$ phases, the number of active agents becomes at most $\lfloor k/g \rfloor$. Then, from [12], the number of inactive agents between two active agents is at least g 1. Therefore, all remaining active agents become leaders. Note that, during the execution of the algorithm, the number of active agents may become one. In this case, the active agent immediately becomes a leader.

In the following, we implement the above algorithm in asynchronous unidirectional rings.
First, we apply a traditional approach [12] to implement the above algorithm in a unidirectional ring. Let us consider the behavior of active agent a_h. In unidirectional rings, a_h cannot move backward and so cannot observe the ID of its backward active agent. Instead, a_h moves forward until it observes IDs of two active agents. Then, a_h observes IDs of three successive active agents. We assume a_h observes id_0, id_1, id_2 in this order. Note that id_0 is the ID of a_h. Here this situation is similar to that the active agent with ID id_1 observes id_0 as its backward active agent and id_2 as its forward active agent in bidirectional rings. For this reason, a_h behaves as if it would be an active agent with ID id_1 in bidirectional rings. That is, if id_1 is the smallest among the three IDs, a_h remains active as a candidate of leaders. Otherwise, a_h drops out from the candidate of leaders and becomes inactive. After the phase, a_h assigns id_1 to its ID if it remains active as a candidate.

For example, consider the initial con guration in Fig 1 (a). In gures, the number near each agent is the ID of the agent and the box of each node represents the whiteboard. First, each agent writes its own ID to the whiteboard on its initial node. Next, each agent moves forward until it observes two IDs, and then the con guration is changed to the one in Fig 1 (b). In this con guration, each agent compares three IDs. The agent with ID 1 observes IDs (1, 8, 3), and so it drops out from the candidate because the middle ID 8 is not the smallest. The

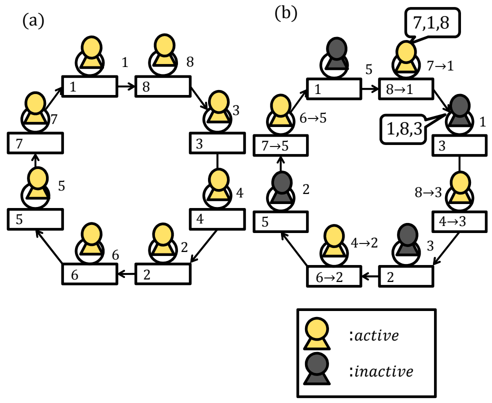


Fig 1: An example for a g-partial gathering problem($k = 9, g = 3$)

agents with IDs 3, 2, and 5 also drop out from the candidate. The agent with ID 7 observes IDs (7, 1, 8), and so it remains active as a candidate because the middle ID 1 is the smallest. Then, it updates its ID to 1. The agents with IDs 8, 4, and 6 also remain active as candidates and similarly update their IDs.

Next, we explain the way to treat asynchronous agents. To recognize the current phase, each agent manages *phase number*. Initially, the phase number is one, and it is incremented when each phase is completed. Each agent compares IDs with agents that have the same phase number. To realize this, when each agent writes its ID to the whiteboard, it also writes its phase number. That is, at the beginning of each phase, active agent a_h writes a tuple $(phase, id_h)$ to the whiteboard on its current node, where $phase$ is the current phase number and id_h is the ID of a_h. After that, agent a_h moves until it sees two IDs with the same phase number as that of a_h. Then, a_h decides whether it remains active as a candidate or becomes inactive. If a_h remains active, it updates its own ID. Agents repeat these behaviors until they complete the $\lceil \log g \rceil$-th phase.

For example, consider the initial con guration in Fig 1 (a). In gures, the number near each agent is the ID of the agent and the box of each node represents the whiteboard. First, each agent writes its own ID to the whiteboard on its initial node. Next, each agent moves forward until it observes two IDs, and then the con guration is changed to the one in Fig 1 (b). In this con guration, each agent compares three IDs. The agent with ID 1 observes IDs (1, 8, 3), and so it drops out from the candidate because the middle ID 8 is not the smallest. The agents with IDs 3, 2, and 5 also drop out from the candidate. The agent with ID 7 observes IDs (7, 1, 8), and so it remains active as a candidate because the middle ID 1 is the smallest. Then, it updates its ID to 1. The agents with IDs 8, 4, and 6 also remain active as candidates and similarly update their IDs.

Next, we explain the way to treat asynchronous agents. To recognize the current phase, each agent manages *phase number*. Initially, the phase number is one, and it is incremented when each phase is completed. Each agent compares IDs with agents that have the same phase number. To realize this, when each agent writes its ID to the whiteboard, it also writes its phase number. That is, at the beginning of each phase, active agent a_h writes a tuple $(phase, id_h)$ to the whiteboard on its current node, where *phase* is the current phase number and id_h is the ID of a_h. After that, agent a_h moves until it sees two IDs with the same phase number as that of a_h. Then, a_h decides whether it remains active as a candidate or becomes inactive. If a_h remains active, it updates its own ID. Agents repeat these behaviors until they complete the $\lceil \log g \rceil$-th phase.

Pseudocode. The pseudocode to elect leader agents is given in Algorithm 1. All agents start the algorithm with active states. The pseudocode describes the behavior of active agent a_h, and v_j represents the node where agent a_h currently stays. If agent a_h becomes an inactive state or a leader state, a_h immediately moves to the next part and executes the algorithm for an inactive state or a leader state in section 3.2.

Agent a_h uses variables $a_h.id_1$, $a_h.id_2$, and $a_h.id_3$ to store three IDs of three successive active agents. Note that a_h stores its ID on $a_h.id_1$ and initially assigns its initial ID $(a_h.id)$ to $a_h.id_1$. Variable $a_h.phase$ stores the phase number of

Algorithm 1 The behavior of active agent a_h (Node v_j is the current node of a_h.)

```
1: set $a_h.phase = 1$ and $a_h.id_1 = a_h.id$
2: if $v_j.inactive = 1$ then
3:     // Some agents have passed $a_h$ before $a_h$ starts the algorithm.
4:     become inactive
5: end if
6: set $(v_j.phase, v_j.id) = (a_h.phase, a_h.id_1)$
7: $BasicAction()$
8: set $a_h.id_2 = v_j.id$
9: $BasicAction()$
10: set $a_h.id_3 = v_j.id$
11: if $a_h.id_2 > min(a_h.id_1, a_h.id_3)$ then
12:     set $v_j.inactive = 1$
13:     become inactive
14: else
15:     if $a_h.phase = \lceil \log g \rceil$ then
16:         become a leader
17:     else
18:         set $a_h.phase = a_h.phase + 1$
19:         set $a_h.id_1 = a_h.id_2$
20:     end if
21:     return to step 6
22: end if
```

a_h. Each node v_j has variable $(v_j.phase, v_j.id)$, where an active agent writes its phase number and its ID. For any v_j, variable $(v_j.phase, v_j.id)$ is $(0, 0)$ initially. In addition, each node v_j has variable $v_j.inactive$. This variable represents whether there exists an inactive agent on v_j. That is, agents update the variable to keep the following invariant: If there exists an inactive agent on v_j, $v_j.inactive = 1$ holds, and otherwise $v_j.inactive = 0$ holds. Initially $v_j.inactive = 0$ holds for any v_j. In Algorithm 1, a_h uses procedure $BasicAction()$, by which agent a_h moves to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$.

We give the pseudocode of $BasicAction()$ in Algorithm 2. In $BasicAction()$, the main behavior of a_h is to move to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$. To realize this, a_h skip nodes such that no agent initially exists (i.e., $v_j.initial = 0$) or an inactive agent currently exists (i.e., $v_j.inactive = 1$), and continue to move until it reaches a node where some active

Algorithm 2 Procedure *BasicAction*() for a_h

```
 1: procedure $BasicAction()$
 2: move to the forward node
 3: while $(v_j.initial = 0) \vee (v_j.inactive = 1)$ do
 4:     move to the forward node
 5: end while
 6: if $v_j.phase = 0$ then
 7:     set $v_j.inactive = 1$
 8:     return to step 2
 9: end if
10: if $a_h.phase \neq v_j.phase$ then
11:     wait until $v_j.phase = a_h.phase$ or $v_j.inactive = 1$
12:     if $v_j.inactve = 1$ then
13:         return to step 2
14:     end if
15: end if
16: // $a_h$ reaches $v_j$ s.t. $v_j.phase = a_h.phase$.
17: if $(v_j.phase, v_j.id) = (a_h.phase, a_h.id_1)$ then
18:     become a leader
19: end if
20: end procedure
```

agents start the same phase (lines 3 to 5). In addition to this behavior, a_h makes some behaviors to treat asynchrony. If a_h nds agent a_x that has not yet started the algorithm on v_j, a_h makes a_x drop out from candidates by setting $v_j.inactive = 1$ (lines 6 to 9). When a_h notices that it has passed some active agents, a_h waits until the agents catch up with a_h (lines 10 to 15). If the agent becomes inactive, a_h continues to move (lines 12 to 14). During the algorithm, it is possible that a_h becomes the only one candidate of leaders. In this case, a_h immediately becomes a leader (lines 17 to 19).

Analysis. We have the following lemma about Algorithm 1 [12].

lemma 3.1. *After executing Algorithm 1, the con guration satis es the following.*

- *There exists at least one leader agent.*
- *There exist at most* $\lfloor \frac{k}{g} \rfloor$ *leader agents.*
- *There exist at least* g 1 *inactive agents between two leader agents.*

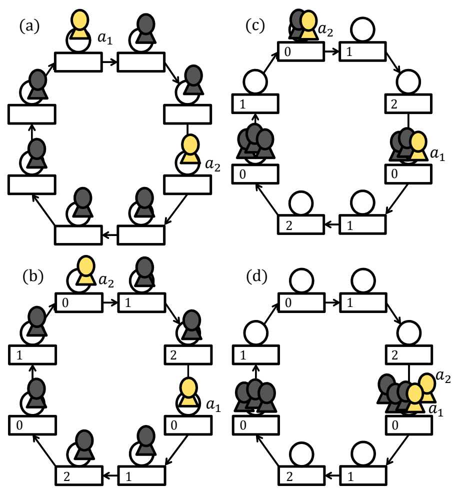


Fig 2: The realization of partial gathering($g = 3$)

In addition, we have the following lemma [12].

lemma 3.2. *The total number of moves to execute Algoritm 1 is* $O(n \log g)$.

3.2 The second part

In this section, we explain the second part, i.e., an algorithm to achieve g-partial gathering by using leaders elected by the algorithm in section 3.1. Let leader nodes (resp., inactive nodes) be the nodes where an agent becomes a leader (resp., an inactive agent) in the rst part. The idea of the algorithm is as follows: First each leader agent a_h writes 0 to the whiteboard on the current node. Then, a_h repeatedly moves and, whenever a_h visits an inactive node, a_h writes $y \bmod g$ to the whiteboard, where y is the number of inactive nodes a_h has ever visited. That is, a_h writes $0, 1, \ldots, g \quad 1, 0, 1, \ldots$ to the whiteboard on inactive nodes. This number is used to instruct inactive agents where they should move to achieve g-partial gathering. Agent a_h continues this operation until it visits the node where 0 is already written to the whiteboard. Note that this node is a leader node. For example, consider the con guration in Fig 2 (a). In this con guration, agents a_1 and a_2 are leader agents.

First, a_1 and a_2 write 0 to their current whiteboards, and then they move and write numbers to whiteboards until they visit the node where 0 is written on the whiteboard. Then, the system reaches the con guration in Fig 2 (b).

Then, each non-leader agent (i.e., inactive agent) moves based on the leader's instruction, i.e., the number written to the whiteboard. More concretely, each inactive agent moves to the node where 0 is written to the whiteboard. For example, after the con guration in Fig 2 (b), the system reaches the con guration in Fig 2 (c). Note that a node where 0 is written is a leader node or an inactive node. If the node is an inactive node, g agents meet at the node. If the node is a leader node, it is possible that only less than g agents meet at the node. In this case, the agents continue to move until they visit the next node where 0 is written. By executing such operations, agents can solve the g-partial gathering problem. For example, there exist only two agents on the node where a_2 exists in Fig 2 (c). So the two agents continue to move until they visit the next node where 0 is written (Fig 2 (d)).

Pseudocode. In the following, we show the pseudocode of the algorithm. In this part, states of agents are divided into the following three states

- *leader*: The agent instructs inactive agents where they should move.
- *inactive*: The agent waits for the leader's instruction.
- *moving*: The agent moves to its gathering node.

In this part agents continue to use $v_j.initial$ and $v_j.inactive$. Remind that $v_j.initial = 1$ if and only if there exists an agent at v_j initially. Algorithm 1 assures $v_j.inactive = 1$ if and only if there exists an inactive agent at v_j. Note that, since each agent becomes inactive or a leader at a node such that there exists an agent initially, agents can ignore and skip every node $v_{j'}$ such that $v_{j'}.initial = 0$.

The pseudocode of leader agents is described in Algorithm 3. Variable $a_h.count$ is used to count the number of inactive nodes a_h visits (The counting is done modulo g). The initial value of $a_h.count$ is 0. Variable $v_j.count$ is used for leader agents to instruct inactive agents. That is, leader agent a_h writes $a_h.count$ to $v_j.count$ when it visits inactive node v_j. For any v_j, the initial value of $v_j.count$ is $\perp$. In asynchronous rings, leader agent a_h may pass agents that still execute Algorithm 1. To avoid this, a_h waits until the agents catch up with a_h. More precisely, when leader agent a_h visits the node v_j such that $v_j.initial = 1$, it passed such agents if $v_j.inactive = 0$ and $v_j.count = \perp$ hold. This is because $v_j.inactive = 1$ should hold if some agent becomes inactive at v_j, and $v_j.count = 0$ holds if some agent becomes leader at v_j. In this case, a_h waits there until either $v_j.inactive = 1$ or $v_j.count = 0$ holds (lines 8 to 10). When leader agent updates $v_j.count$, an inactive agent on node v_j becomes a moving state (line 12). This behavior of inactive agents is given in the pseudocode of inactive agents (See Algorithm 4). After a leader agent reaches the next leader node, it becomes a moving agent to move to the node where at least g agents meet (line 17). Note that variable $a_h.Bcount$ is used in the pseudocode for moving agents, and so we explain the variable later.

The pseudocode of moving agents is described in Algorithm 5. Moving agent a_h continues to move until it visits node v_j such that $v_j.count = 0$. When a_h visits such a node, it is possible that only less than g agents come to the node like Fig 2 (c). To solve this case, a_h keeps the value of $v_l.count$ for the previous inactive node v_l as variable $a_h.Bcount$. When a_h visits node v_j such that $v_j.count = 0$, if $a_h.Bcount = g \quad 1$ holds, at least g agents come to v_j. Otherwise, less than g agents come to v_j, and so a_h moves to the next node $v_{j'}$ such that $v_{j'}.count = 0$. Note that, since there exist at least $g \quad 1$ inactive nodes between two leader nodes, at least g agents meet at $v_{j'}$.

In asynchronous rings, a moving agent may pass leader agents. To avoid this, the moving agent waits until the leader agent catches up with it. More precisely, if moving agent a_h visits node v_j such that $v_j.initial = 1$ and $v_j.count = \perp$, a_h

Algorithm 3 The behavior of leader agent a_h (Node v_j is the current node of a_h.)

```
1: set $a_h.count = 0$
2: set $v_j.count = a_h.count$ and $a_h.count = a_h.count + 1$
3: move to the forward node
4: while $v_j.count \neq 0$ do
5:    while $v_j.initial = 0$ do
6:       move to the forward node
7:    end while
8:    if $(v_j.inactive = 0) \wedge (v_j.count = \perp)$ then
9:       wait until $v_j.inactive = 1$ or $v_j.count = 0$
10:   end if
11:   if $v_j.inactive = 1$ then
12:      set $v_j.count = a_h.count$
13:      // an inactive agent at $v_j$ becomes a moving state
14:      set $a_h.count = (a_h.count + 1) \bmod g$
15:      set $a_h.Bcount = v_j.count$
16:   end if
17:   move to the forward node
18: end while
19: become a moving state
```

Algorithm 4 The behavior of inactive agent a_h (Node v_j is the current node of a_h.)

```
1: wait until $v_j.count \neq \perp$
2: become a moving state
```

passed a leader agent. To wait for the leader agent, a_h waits there until the value of $v_j.count$ is updated.

Analysis. We have the following lemma about the algorithm in section 3.2.

lemma 3.3. *After the leader agent election, agents solve the g-partial gathering problem in $O(gn)$ total number of moves.*

Proof. From the algorithm, clearly agents solve the g-partial gathering problem. In the following, we consider the total number of moves required to execute the algorithm.

First let us conider the total number of moves required for each leader agent to move to its next leader node, and required for each inactive

Algorithm 5 The behavior of moving agent a_h (Node v_j is the current node of a_h.)

```
1: while $v_j.count \neq 0$ do
2:    move to the forward node
3:    if $(v_j.initial = 1) \wedge (v_j.count = \perp)$ then
4:       wait until $v_j.count \neq \perp$
5:    end if
6:    if $v_j.count \neq \perp$ then
7:       set $a_h.Bcount = v_j.count$
8:    end if
9: end while
10: if $a_h.Bcount \neq g \quad 1$ then
11:    set $a_h.Bcount = 0$
12:    move to the forward node
13:    return to step 1
14: end if
15: terminate
```

(or moving) agent to move to node v_j such that $v_j.count = 0$ (For example, the total number of moves from Fig 2 (a) to Fig 2 (c)). The total number of these moves is at most $O(gn)$ because each link is passed by agents at most g times.

Second let us consider the total number of moves required for each agent a_h to move to its next node $v_{j'}$ such that $v_{j'}.count = 0$ in the case of $a_h.Bcount \neq g \quad 1$ (For example, the total number of moves from Fig 2 (c) to Fig 2 (d)). From the algorithm, only agents that arrived at leader nodes can make such moves. Then, the agents nd node $v_{j'}$ such that $v_{j'}.count = 0$ before it visits the next leader node. This is because there exist at least $g \quad 1$ inactive nodes between two leader nodes from Lemma 3.1. Consequently, since at most $g \quad 1$ agents start these moves at a leader node, each link is passed by agents at most $g \quad 1$ times, and thus the total number of these moves is at most $O(gn)$.

Therefore, we have the lemma. □

From Lemmas 3.2 and 3.3, we have the following theorem.

Theorem 3.1. *When agents have distinct IDs, our deterministic algorithm solves the g-partial gathering problem in $O(gn)$ total number of moves.*

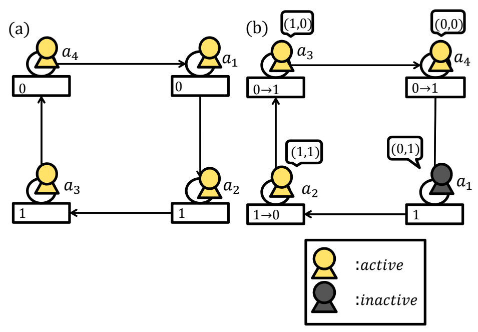


Fig 3: A randomized leader election for anonymous agents

4 A Randomized Algorithm for Anonymous Agents

In this section, we propose a randomized algorithm to solve the g-partial gathering problem for the case of anonymous agents. The idea of the algorithm is the same as that in section 3. Agents elect leader agents in the rst part, and the leader agents instruct the other agents where they move in the second part. The di erence from the algorithm in section 3 is that agents elect exactly one leader by randomization in the rst part. In the second part, we use the same algorithm as that in section 3.

4.1 The rst part

In this subsection, we explain a randomized algorithm to elect one leader agent from anonymous agents. Similarly to section 3, the state of each agent is either active, inactive, or leader. Initially all agents are active. If an agent becomes inactive or a leader, it immediately moves to the second part of the algorithm.

The algorithm consists of several phases. In each phase, each active agent a_h writes a random bit to the whiteboard and moves to the next node where its forward agent a_f writes a random bit. Then, a_h compares the random bit of a_h with that of a_f. If the random bit of a_h is zero and the random bit of a_f is one, a_h drops out from candidates of the leader. Otherwise, a_h remains active as a candidate, and moves to the next phase. Since a_h drops out with probability 1/4 and at least one of a_h and a_f remains active as a candidate, eventually exactly one active agent remains active as a candidate by repeating the phase. For example, consider the initial con guration in Fig 3 (a). Numbers on whiteboards are random bits written by the resident agents. Each agent moves to the next node and then compares random bits. Because the random bit of a_1 is zero and the random bit of its forward agent a_2 is one, a_1 drops out from the candidate of the leader. The other agents remain active as candidates and update random numbers on the whiteboards (Fig 3).

Algorithm 6 The behavior of active agent a_h (Node v_j is the current node of a_h.)

```
1: set $a_h.phase = 1$ and $a_h.num = 0$
2: if $v_j.inactive = 1$ then
3:     // Some agents pass $a_h$ before $a_h$ starts the algorithm.
4:     become inactive
5: end if
6: set $a_h.r = 0$ with probability 1/2
   and $a_h.r = 1$ with probability 1/2
7: set $(v_j.phase, v_j.r) = (a_h.phase, a_h.r)$
8: $BasicAction2()$
9: if $a_h.r = 0$ and $v_j.r = 1$ then
10:     set $v_j.inactive = 1$
11:     become inactive
12: end if
13: set $a_h.phase = a_h.phase + 1$
14: set $a_h.num = 0$
15: return to step 6
```

To execute the above algorithm, we treat asynchronous agents in the same way as the algorithm in section 3.1. Each agent manages *phase number* to recognize the current phase. Each agent writes its random bit together with its phase number, and compares its random bit with an agent that has the same phase number. In addition, since active agent a_h may pass some other active agents, a_h waits in the same way until the agents catch up with a_h.

Pseudocode. The pseudocode of active agents is described in Algorithm 6. Agent a_h stores its phase number in variable $a_h.phase$ and its random bit in variable $a_h.r$. Each node v_j has variable $(v_j.phase, v_j.r)$, where an active agent writes its phase number and its random bit. For any v_j, variable $(v_j.phase, v_j.r)$ is $(0,0)$ initially. In addition to these variables, a_h manages $a_h.num$ to count the number of inactive agents in each phase. If $a_h.num = k \quad 1$ holds, a_h is a unique active agent and thus becomes a leader (This behavior is implemented in $BasicAction2()$).

At the beginning of each phase, each a_h generates a random bit and stores it in $a_h.r$. Then, it writes $(a_h.phase, a_h.r)$ to variable $(v_j.phase, v_j.r)$ at the whiteboard of its current node v_j. Since the forward active agent of a_h also writes a random bit to the whiteboard of its current node v_f, agent a_h compares the two random bits when a_h visits v_f. In Algorithm 6, a_h uses procedure $BasicAction2()$, by which a_h moves to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$.

We give the pseudocode of $BasicAction2()$ in Algorithm 7. The implementation is almost the same as that of $BasicAction()$ in section 3.2. The di erence is that a_h increments $a_h.num$ whenever a_h sees inactive agents. If a_h observes $k \quad 1$ inactive agents, a_h is a unique active agent and thus becomes a leader.

Analysis. We have the following lemma about Algorithm 6.

lemma 4.1. *Algorithm 6 solves the leader agent election with $O(n \log k)$ expected total moves.*

Proof. Consider the s-th phase ($s = 1, 2, \ldots$). In the s-th phase, each active agent moves to the node where another active agent starts s-th phase. Consequently, the total number of moves in s-th phase is n.

In each phase, only when an active agent observes two random bits $(0,1)$, it drops out from the candidate and becomes inactive. This means that each active agent becomes inactive with probability $1/4$ in the s-th phase. Thus, the expected number of phases is $\log_{\frac{4}{3}} k$. This implies the lemma. □

Algorithm 7 procedure $BasicAction2()$

```
 1: procedure $BasicAction2()$
 2: move to the forward node
 3: while $(v_j.initial = 0) \vee (v_j.inactive = 1)$ do
 4:    if $v_j.inactive = 1$ then
 5:       set $a_h.num = a_h.num + 1$
 6:    end if
 7:    move to the forward node
 8: end while
 9: if $v_j.phase = 0$ then
10:    set $v_j.inactive = 1$
11:    set $a_h.num = a_h.num + 1$
12:    return to step 2
13: end if
14: if $v_j.phase \neq a_h.phase$ then
15:    wait until $v_j.phase = a_h.phase$ or $v_j.inactive = 1$
16:    if $v_j.inactive = 1$ then
17:       set $a_h.num = a_h.num + 1$
18:       return to step 2
19:    end if
20: end if
21: // $a_h$ reaches $v_j$ s.t. $v_j.phase = a_h.phase$.
22: if $a_h.num = k \quad 1$ then
23:    become a leader
24: end if
25: end procedure
```

4.2 The second part

In the second part of this algorithm, we use the same algorithm in section 3.2. Since Algorithm 6 selects exactly one leader agent, the conditions in Lemma 3.1 hold for Algorithm 6. In addition, Algorithm 6 satis es the following: 1) Each agent becomes inactive or a leader at node v_j such that $v_j.initial = 1$, and 2) If there exists an inactive agent on v_j, $v_j.inactive = 1$ holds, and otherwise $v_j.inactive = 0$ holds. Since these are su cient conditions to apply the algorithm in section 3.2, we can execute the algorithm in section 3.2 after the algorithm in section 4.1.

Analysis. From Lemmas 4.1 and 3.3, we have the following theorem.

Theorem 4.1. *When agents have no IDs, our randomized algorithm solves the g-partial gathering problem in $O(n \log k + gn)$ expected total moves.*

5 Conclusion

In this paper, we have proposed two algorithms to solve the g-partial gathering problem in asynchronous unidirectional rings. The rst algorithm is deterministic and assumes distinct agents, and the second algorithm is randomized and assumes anonymous agents. In the rst algorithm, several agents are elected as leaders by executing the leader agent election partially. On the other hand, in the second algorithm, the unique leader is elected. After the leader election, leader agents instruct the other agents where they meet. We have showed that the rst algorithm requires $O(gn)$ total moves, which is asymptotically optimal. One of future works is to propose a randomized algorithm for anonymous agents to solve the g-partial gathering problem in $O(gn)$ expected total moves. Another future work is to consider the solvability of deterministic g-partial gathering, that is, we will clarify what initial con gurations are solvable and what complexity is required.

Reference

[1] Kranakis, E., Krozanc, D., Markou, E.:The mobile agent rendezvous problem in the ring. Synthesis Lectures on Distributed Computing Theory (2010)

[2] Gasieniec, L., Kranakis, E., Krizanc, D., Zhang, X.: Optimal memory rendezvous of anonymous mobile agents in a unidirectional ring. SOFSEM (2006)

[3] Kranakis, E., Santoro, N., Sawchuk, S.:Mobile agent rendezvous in a ring. Distributed Computing Systems (2003)

[4] Flocchini, P., Kranakis, E., Krizanc, D,. Santoro, N., Sawchuk, C.: Multiple mobile agent rendezvous in a ring. LATIN, vol 2976, 599-608 (2004)

[5] Kranakis, E., Krizanc, D., Markou, E.: Mobile agent rendezvous in a synchronous torus. LATIN, vol 3887, 653-664 (2006)

[6] Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F., Santoro, N., Sawchuk, C.: Mobile agents rendezvous when tokens fail. SIROCCO, vol 3104, 161-172 (2004)

[7] Dobrev, S., Flocchini, P., Prencipe, G., Nicola, N.: Mobile search for a black hole in an anonymous ring. Algorithmica, vol 48, 67-90 (2007)

[8] Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Multiple agents rendezvous in a ring in spite of a black hole. OPODIS, vol 3114, 298-304 (2003)

[9] Suzuki,T., Izumi, T., Ooshita, F., Kakugawa, H., Msuzawa, T.: Move-optimal gossiping among mobile agents. Theoretical Computer Science, vol 393, 90-101 (2008)

[10] Barriere, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Rendezvous and election of mobile agents: impact of sense of direction. Theory of Computing System, vol 40, 143-162 (2007)

[11] Kawai, S., Ooshita, F., Kakugawa, H., Masuzawa, T.: Randomized rendezvous of mobile agents in anonymous unidirectional ring networks. SIROCCO, vol 7355, 303-314 (2012)

[12] Peterson G.L.: An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. TOPLAS, vol 4, 758-762 (1982)

同期リング上における
モバイルエージェント均一配置アルゴリズム

妻鹿 敏也 [†], 大下 福仁 [†], 角川 裕次 [†], 増澤 利光 [†]

[†] 大阪大学 大学院情報科学研究科,565-0871, 大阪府吹田市山田丘 1-5
{ t-mega, f-oosita, kakugawa, masuzawa}@ist.osaka-u.ac.jp

Abstract

In this paper, we consider uniform deployment of mobile agents in a synchronous unidirectional network, which requires the agents to uniformly spread on the network. First, we show a lower bound $\Omega(kn)$ of the number of agent moves, where k and n are the numbers of agents and nodes respectively. We also present three $O(kn)$-moves (i.e., asymptotically optimal) algorithms (one for the whiteboard model and two for the token model), and analyze their memory requirement and time complexity.
keyword: distributed system, mobile agent, uniform deployment, whiteboard, token

1 Introduction

In recent years, distributed systems have become important to satisfy demands for cost-effective large-scale systems. A distributed system consists of a large number of computers (nodes) and communication links. In distributed systems, each node needs to operate autonomously but cooperatively with others to achieve a common goal. In addition, faults on nodes and communication links are likely to happen, and thus it is important to design distributed systems that can work in spite of some faults. To design such distributed systems, *mobile agents* have received much attention as a promising design paradigm[1]-[11]. A (mobile) agent is an autonomous software that can move among nodes in the network with keeping some information.

A distributed system with mobile agents is called a *mobile agent system*. Agents can simplify the network management [3, 4] because multiple agents can traverse the network and monitor the network configuration cooperatively. For instance, agents can realize quick recovery from faults by detecting faulty nodes and notifying other nodes of the information.

To realize mobile agent systems, there are many studies about agent algorithms that take advantage of autonomy and cooperativeness of agents. For instance, Suzuki et al.[5] considered a *gossip problem*, which requires all agents to share their information. They proposed gossip algorithms on the assumption that agents can communicate with others staying at the same node and can use whiteboard on each node. Another fundamental and the most investigated problem is the *rendezvous problem*[6]-[10], which requires all agents to meet at a single node. The rendezvous problem is considered in rings[6]-[7], torus[8], trees[9], and arbitrary networks[10]. Some works assume that agents can use whiteboard on each node, and others assume that agents can use only tokens, which are markers agents can leave on nodes. Elor and Bruckstein[11] considered *uniform deployment* of multiple agents, which requires all agents to spread uniformly in the network. They propose uniform deployment algorithms under the assumption that agents are oblivious (or memoryless) but can observe multiple nodes within its visibility range.

In this paper, we focus on the uniform deployment on synchronous unidirectional rings. From a practical view point, the uniform deployment is useful for the network management. For instance, if agents that can repair nodes are deployed uniformly in the network, such agents can quickly reach and repair faulty nodes after the faults are detected. If agents with database are deployed uniformly, each node can quickly access the database. The uniform deployment is interesting to investigate also from a theoretical point of view. The problem exhibits a striking contrast to the rendezvous: the uniform deployment aims to attain the symmetry of agent locations while the rendezvous aims to break the symmetry. It is well known that the symmetry breaking is difficult (and

Table 1: Uniform deployment algorithms on synchronous unidirectional rings

	Algorithm 1	Algorithm 2	Algorithm 3
ID (agent)	available	not available	not available
communication model	whiteboard	token	token
memory requirement (node)	$O(\log k)$	$O(1)$	$O(1)$
memory requirement (agent)	$O(\log n)$	$O(k \log n)$	$O(\log n)$
time complexity	$O(n)$	$O(n)$	$O(n \log k)$
total agent moves	$O(kn)$	$O(kn)$	$O(kn)$

n : the number of nodes, k : the number of agents

sometimes impossible) to attain in distributed systems, and so is the rendezvous. Consequently, it is interesting to clarify, as a direct contrast of the rendezvous, how easily the uniform deployment can be attained.

We consider the uniform deployment for agents that have memory but cannot observe nodes except for its currently visiting node (this is different from [11]). Furthermore, we consider two types of communication models, the *whiteboard model* and the *token model*. In the whiteboard model, each agent can write to or read from a whiteboard on each node. In the token model, each agent initially has a single token and can leave the token on a visiting node.

Contributions of this paper are summarized in Table 1. We propose three algorithms for the uniform deployment on synchronous unidirectional rings. For all algorithms, the total number of agent moves is $O(kn)$, where k is the number of agents and n is the number of nodes. We show that $\Omega(kn)$ moves are necessary to solve the uniform deployment problem, and thus these algorithms are asymptotically optimal in terms of the total number of moves. The first algorithm achieves the uniform deployment on the whiteboard model, and it requires $O(\log k)$ memory per node, $O(\log n)$ memory per agent, and $O(n)$ time. The second and the third algorithms achieve the uniform deployment on the token model. The second algorithm realizes the uniform deployment in asymptotically optimal time (i.e., $O(n)$) but requires $O(k \log n)$ memory per agent. The third algorithm reduces the memory per agent to $O(\log n)$ by allowing $O(n \log k)$ time.

2 Preliminaries

2.1 System model

A *unidirectional ring network* R is defined as 2-tuple $R = (V,E)$, where V is a set of nodes and E is a set of unidirectional links. We denote by $n(= |V|)$ the number of nodes and let $V = \{v_0, v_1, \ldots, v_{n\ \ 1}\}$ and $E = \{e_0, e_1, \ldots, e_{n\ \ 1}\}(e_i = (v_i, v_{(i+1) \bmod n}))$. For simplicity, operations to an index of a node assume calculation under modulo n, that is, $v_{(i+1) \bmod n}$ is simply represented by v_{i+1}. The distance from $v_i(0 \quad i \quad n \quad 1)$ to $v_j(0 \quad j \quad n \quad 1)$ is defined to be $(j \quad i) \bmod n$.

We consider a *mobile agent system*, in which agents move in the network and perform some jobs at visiting nodes. An agent is a state machine having an *initial state* and a *terminal state*. Let $k\ (\quad n)$ be the number of agents and $A = \{a_0, a_1, \ldots, a_{k\ \ 1}\}$ be a set of agents. For simplicity, operations to an index of an agent assume calculation under modulo k. Since the network is a unidirectional ring, agents staying at v_i can move only to v_{i+1}. Each agent can recognize whether another agent is staying at the same node or not. The *home node* of agent a is the node where a stays initially, and is denoted by $v_h(a)$. We consider a *synchronous* system, that is, all agents simultaneously start its actions and execute their actions in a lockstep fashion.

In this paper, we consider two model variations, the *whiteboard model* and the *token model*:

Whiteboard model: Each node $v(\in V)$ has a whiteboard, and each agent staying at the node can write to or read from the whiteboard. In this model, we assume each agent has a distinct ID of $O(\log k)$ bits.

(Unremovable) token model: Each agent initially has a single token and can leave it on a visiting node. In this model, we assume that each agent has no ID, and thus, agents cannot recognize the owner of each token.

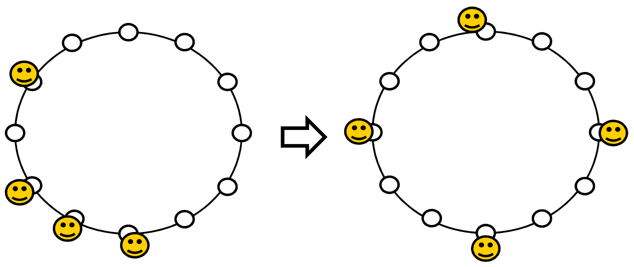


Figure 1: An initial and a terminal configurations of the uniform deployment

A *configuration* (or global state) C of the agent system is defined as a triplet $C = (S, T, P)$. The first element S is a k-tuple $S = (s_0, s_1, \ldots, s_{k\ \ 1})$ representing the agent states at C where s_i is the state of $a_i(0 \quad i \quad k \quad 1)$. The second element T is an n-tuple $T = (t_0, t_1, \ldots, t_{n\ \ 1})$ denoting the node states where t_j is the state (i.e., the whiteboard contents or the number of tokens) of $v_j(0 \quad j \quad n \quad 1)$. The last element P is a k-tuple $P = (p_0, p_1, \ldots, p_{k\ \ 1})$ denoting the agent locations where $p_i = j$ implies that agent a_i is staying at node $v_j(0 \quad i \quad k \quad 1, 0 \quad j \quad n \quad 1)$. We denote by $\mathscr{C}$ the set of all the possible configurations of the agent system.

In *initial configuration* $C_0 \in \mathscr{C}$, each agent is in its initial state, and the whiteboard of each node is empty or each node has no token. When an agent completes execution of an algorithm, it changes its own state to the terminal state. A *terminal configuration* is the one in which all agents are in the terminal states.

The configuration C_i changes to C_{i+1} by actions of agents. Each action of an agent consists of the followings.

1. Local computation: Agent a_j on a node v_i updates the states of a_j and v_i depending on its current state, the current state of v_i and the number of agents staying at v_i. When two or more agents are staying at v_i, they execute their actions in an arbitrary order but the action of each agent is atomic (or non-interrupted).

2. Movement: Agent a_j on a node v_i moves to node v_{i+1} or stays at v_i. The decision whether it moves or not depends on its (updated) state.

Let ϕ_i^j be an action of a_j and Φ_i be a set of actions of all agents, i.e., $\Phi_i = (\phi_i^0, \phi_i^1, \ldots, \phi_i^{k\ \ 1})$. An *execution* ε is a maximal alternating sequence of configurations and sets of actions $C_0\Phi_1C_1\Phi_2C_2\Phi_3\ldots$, where Φ_i changes $C_{i\ \ 1}$ to C_i. The maximality implies that the execution is infinite or ends with a terminal configuration.

2.2 The uniform deployment problem

We define the *uniform deployment problem* as the problem that requires $k(\quad 2)$ agents to spread uniformly in a ring network (Figure 1). In the initial configuration, all agents are on arbitrary nodes, and we assume that no two agents stay at the same node, that is, the home nodes of all agents are different from each other. Each node initially has the empty whiteboard in the whiteboard model, and has no token in the token model.

In the terminal configuration, the distance of any two *adjacent agents* is identical. Here, we say two agents are adjacent when there exists no agent between them in the ring. However, we should consider the case that n is not a multiple of k. So we aim to distribute the agents so that the distance d of two adjacent agents should satisfy $\lfloor n/k \rfloor \quad d \quad \lceil n/k \rceil$.

Definition 1 An algorithm solves the uniform deployment problem if any execution ε satisfies the followings.

- Execution ε is finite.
- In the terminal configuration of ε, each distance d of two adjacent agents satisfies $\lfloor n/k \rfloor \quad d \quad \lceil n/k \rceil$. $\Box$

We evaluate the *time complexity* as the time required to reach the terminal configuration from any initial configuration. In the synchronous system, we assume that it takes a time unit for agents to move to the adjacent node, while we ignore the time for local computation.

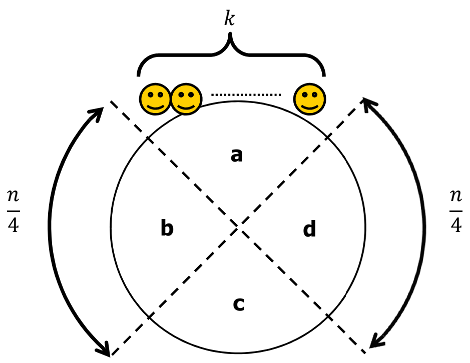


Figure 2: The initial configuration to derive a lower bound $\Omega(kn)$ of the total number of moves

3 Lower Bound of the Total Number of Agent Moves

In this section, we show a lower bound of the total number of agent moves required to achieve the uniform deployment. We consider the initial configuration such that all agents stay in a quarter part of the ring (Figure 2). In Figure 2, the ring is divided into four quarter parts, and in the initial configuration, all agents are in the part a (we assume $k \quad n/4$). To achieve the uniform deployment, $k/4$ agents need to move to the part c, and each of them must move at least $n/4$ times. Thus the total number of moves is at least $(k/4) \quad (n/4) = kn/16$.

Theorem 1 A lower bound of the total number of moves to solve the uniform deployment problem in a unidirectional ring network is $\Omega(kn)$, where n is the number of nodes and k is the number of agents. □

We can also obtain the same lower bound for *bidirectional* rings from the above argument.

Corollary 1 A lower bound of the total number of moves to solve the uniform deployment problem in a bidirectional ring network is $\Omega(kn)$, where n is the number of nodes and k is the number of agents. □

4 An Algorithm in the Whiteboard Model

In this section, we present a deterministic algorithm for the uniform deployment in the whiteboard model. We assume that each agent has a distinct ID of $O(\log k)$ bits, and knows neither n nor k. In the initial configuration, there exists at most one agent on each node.

In section 4.1, we present an algorithm under the assumption that $n = ck$ holds for some positive integer c. In section 4.2, we will remove the assumption.

4.1 An algorithm for the case of $n = ck$

Algorithm 1 consists of two phases: selection phase and deployment phase. In the selection phase, a unique *base node* is selected as a reference node of the uniform deployment. In the deployment phase, each agent determines, based on the base node, the *target nodes* where agents should stay to attain the uniform deployment, and moves to a target node.

1. selection phase: In this phase, the home node of the agent, say a_{min}, with the minimum ID is selected as the base node. Each agent finds the distance from its home node to the base node, and in addition, it finds the number n of nodes and the number k of agents.

When initialized, each agent writes its ID to whiteboard of its home node, and starts traversing the ring. During the traversal, each agent keeps, in its variables, the smallest ID it ever found (variable *minid*), the distance from its own home node to the home node of the agent with ID *minid* (*dis*), the number of nodes it visited so far (*nodenum*), and the number of agent IDs it found so far (*agentnum*). Each agent can detect completion of the traversal when it finds its own ID in the whiteboard of the visiting node. At this time,

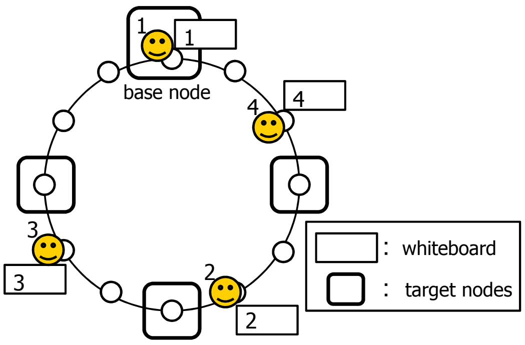


Figure 3: Algorithm 1: The base node and the target nodes (Each number denotes an agent ID.)

these variables correctly store the minimum agent ID, the distance from its home node to the base node, the number n of nodes and the number k of agents.

2. deployment phase: In this phase, by local computation using the results of the selection phase, all agents determine a common set of k target nodes. Then each agent moves to a target node to realize the uniform deployment. Notice that all the agents start the deployment phase at the same time, since they completed the selection phase at the same. Also notice that, at the beginning of the deployment phase, each agent stays at its own home node that is distinct from others.

All the agents select the set of k target nodes as follows: the base node is first selected, and other $k \quad 1$ nodes are selected so that the k nodes are uniformly distributed in the ring, that is, the distance between two adjacent base nodes should be n/k (Figure 3). An agent remains staying at its home node when it is a target node. Otherwise, the agent starts moving to a target node. It traverses the ring until it finds a vacant target node: every time agent a reaches a target node, it stays at the node if the node is vacant (i.e., no agent is staying), otherwise (or the target node is already occupied by another agent) it keeps moving to the next target node. The movement to the target nodes can be realized using the distance to the base node dis and the interval length n/k between the adjacent target nodes. Remark that no two agents reach the same target node at the same time since all the agents start moving at the same time and from distinct nodes.

We give the pseudocode of this algorithm in Algorithm 1. The following theorem clearly holds.

Theorem 2 Algorithm 1 solves the uniform deployment problem in the whiteboard model. The algorithm requires $O(\log k)$ memory for each node, $O(\log n)$ memory for each agent, $O(n)$ time, and $O(kn)$ total number of agent moves. □

4.2 The uniform deployment for the case of $n \neq ck$

To remove the restriction of $n = ck$ imposed in Subsection 4.1, only the parts for determining the target nodes and for moving to a target node should be modified. In the case that n is not a multiple of k, the distance between some adjacent target nodes should be $\lceil n/k \rceil$ while that between some other adjacent target nodes should be $\lfloor n/k \rfloor$. The target nodes should be determined by each agent so that the decisions of different agents should be identical. Since all the agents recognize the same node as the base node, the common target nodes can be determined using the base node as a reference node: Let v_C be the base node (and thus a target node), and $r = n \bmod k$. The target nodes other than v_C is determined as $v_{a_1}, v_{a_2}, \ldots, v_{a_{k \quad 1}}$, where a_j ($j = 1, 2, \ldots, k \quad 1$) is defined as follows:

$$a_j = \begin{cases} a_{j \quad 1} + \lceil \frac{n}{k} \rceil & (1 \quad j \quad r) \\ a_{j \quad 1} + \lfloor \frac{n}{k} \rfloor & (r < j \quad k \quad 1) \end{cases} \tag{1}$$

Algorithm 1 A uniform deployment algorithm on the whiteboard model

Behavior of an agent. Let v_i be its home node.

```
 1: /* selection phase */
 2: $dis, nodenum, agentnum := 0$ ;   $minid$ :=its own ID ;
 3: Write its own ID to $WB_i$ ;   // $WB_i$ is the whiteboard of $v_i$.
 4: repeat
 5:   if $WB_j \neq \perp$ for the current node $v_j$ then
 6:     Increment $agentnum$ by 1 ;
 7:     if $WB_j < minid$ then
 8:       $minid := WB_j$ ;
 9:       $dis := nodenum$ ;
10:   Move to the next node and increment $nodenum$ by 1 ;
11: until $WB_j$ =its own ID ;   // Completion of the ring traversal after $n$ time units from the beginning.
12: /* deployment phase */
13: $d_u := dis$ mod $(nodenum/agentnum)$ ;
14: Move $d_u$ times ;   // Move to the nearest target node.
15: repeat
16:   if no other agent is on the current node then terminate ;
17:   else move $nodenum/agentnum$ times ;   // Move to the next target node.
18: until 0 ;
```

When an agent moves to the next target node (lines 14 and 17 of Algorithm 1), it has to determine the number of moves required to reach the next target node. It can be calculated using the current distance to the base node, the number n of nodes and the number k of agents.

5 Algorithms in the Token Model

In this section, we present two algorithms for the uniform deployment in the token model. Remind that each agent is anonymous in this model. In the initial configuration, each agent stays at its home node distinct from others and holds one token. First, we assume $n = ck$ for some positive integer c, and this assumption is removed in the end of each subsection. In Algorithms 2 and 3, we assume that each agent knows k. Both the algorithms require $O(kn)$ of moves in total, hence are asymptotically optimal in terms of the total number of agent moves. And also, Algorithm 2 realizes the uniform deployment in asymptotically optimal time (i.e., $O(n)$), while Algorithm 3 reduces the memory per agent to $O(\log n)$ by allowing $O(n \log n)$ time.

5.1 An algorithm with the optimal time complexity

Similarly to Algorithm 1, Algorithm 2 consists of the following two phases.

1. selection phase: In this phase, base nodes are selected as reference nodes for the uniform deployment. The difference from Algorithm 1 is as follows: Algorithm 1 selects a *unique* base node using agent IDs. However, in the token model, agents are anonymous and thus it is impossible to select a unique base node (because of the impossibility of symmetry breaking). Instead, Algorithm 2 is allowed to select multiple base nodes when the initial locations of agents are periodic (or symmetric). In addition, the number n of nodes are found in this phase.

When initialized, each agent releases its own token on its home node and starts traversing the ring. The token remains at the node during execution of the algorithm and is used to notify agents that the node is the home node of an agent. During the traversal, each agent memorizes the distance between every pair of two adjacent tokens. Each agent can detect completion of the traversal by counting the number of tokens it found since the agent knows the number k of agents (or tokens). At this time, the agent knows a sequence of distances $D = (d_0, d_1, \ldots, d_{k\ \ 1})$ where d_i $(0 \quad i \quad k \quad 1)$ is the distance from the i-th token it

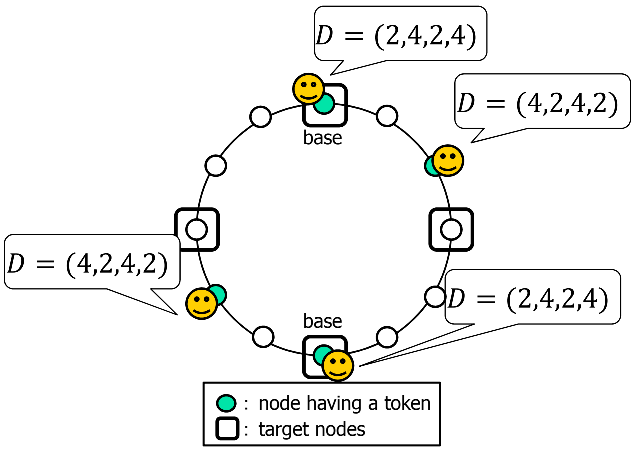


Figure 4: Algorithm 2: The base nodes and the target nodes

found to the $(i+1)$-th token. For completeness, the agent considers its own token as the 0-th and the k-th tokens.

The agent determines the base nodes from $D = (d_0, d_1, \ldots, d_{k\quad 1})$ as follows. Let $\mathscr{D}$ be the set of k sequences obtained by all the possible cyclic shifts of D, i.e., $\mathscr{D} = \{(d_i, d_{i+1}, \ldots, d_{i+k\quad 1}) \mid 0 \quad i \quad k \quad 1\}$ where operations to an index of the distance assume calculation under modulo k. Then, the agent selects, as a base node, the node holding the h-th token when $(d_h, d_{h+1}, \ldots, d_{h+k\quad 1})$ is lexicographically minimum among $\mathscr{D}$. When the distance sequence D is periodic, multiple nodes are selected as base nodes (Fig. 4).

It is quite important to confirm that the base node sets selected by different agents are identical. Let $D = (d_0, d_1, \ldots, d_{k\quad 1})$ and $D' = (d'_0, d'_1, \ldots, d'_{k\quad 1})$ be the distance sequences that two distinct agents a and a' obtain respectively. The sequences D and D' may be different, but the sequence sets $\mathscr{D}$ and $\mathscr{D}'$ obtained by the cyclic shifts of D and D' are identical since D' is a cyclic shift of D. Thus, agents a and a' select the same set of the base nodes.

2. deployment phase: In this phase, by local computation using the base node set, n and k, each agent determines the target node where it should stay to realize the uniform deployment, and moves to the node.

The base nodes selected in the selection phase are uniformly distributed on the ring network, that is, the distance between every pair of two adjacent base nodes is the same. Moreover, the number of agents and their locations between every pair of two adjacent base nodes are also the same. Therefore, the base nodes can be the target nodes of the uniform deployment. Thus, all the agents select the set of k target nodes as follows: the base nodes are first selected, and other $k \quad b$ nodes are selected so that the k nodes are uniformly distributed in the ring, where b is the number of the base nodes (Figure 4),

Since the distance sequence $D = (d_0, d_1, \ldots, d_{k\quad 1})$ is the full information of the agent locations and all the agents select the same set of the k target nodes, each agent can determine the target node it should stay at as follows. Let v be the nearest base node from the agent and dis be the distance from its home node to v. The agent can find from D that it is the j-th agent $(0 \quad j \quad k \quad 1)$ from the base node v (where the agent staying at v is considered as the 0-th agent). Then, the agent can reach its own target node by moving $((dis + j \quad n/k) \bmod n)$ times.

Algorithm 2 describes the pseudocode. The following theorem holds .

Theorem 3 Algorithm 2 solves the uniform deployment problem in the token model. The algorithm requires $O(1)$ memory for each node, $O(k \log n)$ memory for each agent, $O(n)$ time, and $O(kn)$ total number of agent moves. □

The restriction of $n = ck$ imposed in the above can be removed by the similar modification to that in Subsection 4.2. Let b be the number of the base nodes, and $r = n \bmod k$. The distance of every pair of adjacent base

Algorithm 2 A uniform deployment algorithm with the optimal time complexity on the token model

Behavior of an agent. Let v_i be its home node.

```
1:  /* selection phase */
2:  Release a token at its home node $v_i$ ;
3:  $x := 0$ ;   // $x$ denotes the number of tokens the agent found so far.
4:  repeat
5:      Move to the next token with measuring the distance $d_x$ from the previous token ;
6:      $x := x + 1$ ;
7:  until $x = k$ ;   // Continue until the agent returns to its home node $v_i$.
8:  $D := (d_0, d_1, \ldots, d_{k\ \ 1})$ ;   // $D$ is the distance sequence between tokens starting from its own token.
9:  $n := d_0 + d_1 + \quad + d_{k\ \ 1}$ ;
10: $h :=$ the minimum nonnegative integer $i$ such that $(d_i, d_{i+1}, \ldots, d_{i+k\ \ 1})$ is lexicographically minimum among
    $\mathscr{D} = \{(d_j, d_{j+1}, \ldots, d_{j+k\ \ 1}) \mid 0 \quad j \quad k \quad 1\}$ ;
11: $dis := (d_0 + d_1 + \quad + d_{h\ \ 1}) \bmod n$ ;   // $dis$ is the distance to the nearest base node.
12: /* deployment phase */
13: $j := k \quad h$ ;   // The agent is the $j$-th agent from the base node $v_{i+dis}$.
14: Move $((dis + j \quad n/k) \bmod n)$ times and terminate the algorithm ;
```

nodes is identical even in the case of $n \neq ck$, and is $n/b = (\lfloor n/k \rfloor \quad k + r)/b = \lfloor n/k \rfloor \quad k/b + r/b$ (notice that k/b and r/b are integers). This implies that we should select $k/b \quad 1$ target nodes between two adjacent base nodes so that the first r/b intervals between adjacent target nodes should be $\lceil n/k \rceil$ and others should be $\lfloor n/k \rfloor$. With considering the above, each agent can determine its own target node by local computation so that all the agents can spread over the ring to attain the uniform deployment.

5.2 An algorithm with $O(\log n)$ agent memory

Algorithm 2 in the previous subsection uses $O(k \log n)$ memory per agent to store the full information of the initial locations of all agents. The full information allows each agent to select the common set of base nodes whose size *exactly* depends on the symmetry degree of the initial locations: b base nodes are selected when the distance sequence is periodic and consists of b-times repeated subsequences. However, whether the initial locations of agents are periodic or not, multiple base nodes are helpful to realize the uniform deployment when (a) they are uniformly distributed in the ring and (b) the number of the base nodes is a divisor of k. Such base nodes can be selected without drastic increase in the number of agent moves even if the full information of the initial agent locations is not available at each agent. This is the key idea for reducing the agent memory to $O(\log n)$. Similarly to the previous two algorithms, the algorithm consists of the following two phases. Notice that we assume $n = ck$ for some positive integer c in the following description.

1. selection phase: In this phase, some of the home nodes are selected as the base nodes, and they are used as reference nodes for the uniform deployment. The selected base nodes should satisfy the following condition called the *base node condition*: 1) there exists at least one base node, 2) the distance between every pair of two adjacent base nodes is identical, and 3) the number of the home nodes between every pair of two adjacent base nodes is identical. The last condition is introduced to guarantee that the number of the selected base nodes is a divisor of k. In addition, the number n of nodes are found in this phase.

   All the agents complete the selection phase at the same time. When an agent terminates the selection phase, it stays at its home node and knows whether its home node is selected as a base node or not. We call an agent a *leader* when its home node is selected as a base node, and call it a *follower* otherwise.

   We describe the details of this phase later.

2. deployment phase: In this phase, each agent determines the set of the target nodes and moves to a target node. From the base node conditions, the base nodes are first selected as the target nodes. Letting b be the number of the base nodes, other $k \quad b$ target nodes are selected so that the k target nodes are uniformly distributd in the ring, that is, the distance between two adjacent target nodes should be n/k.

All the agents start the deployment phase at the same time. Each leader stays at its own home node since the home node is a target node. A follower knows that it is not a leader but does not know the locations of the leaders. Thus, each follower moves to search the nearest base node. The follower detects its arrival at the base node when it first reaches a node where an agent (or a leader) is staying. After reaching the base node, the agent moves to a vacant target node in the same way as Algorithm 1 and stays at the node: it moves n/k times to the next target node. Every time the agent reaches a target node, it stays at the node if the node is vacant, otherwise it moves to the next target node. It is easy to move to the next target node since the distance between the adjacent target nodes is n/k.

The selection phase

In the followings, we explain how the selection phase selects the base nodes satisfying the base node condition. To select the base nodes, some agents are slected as leaders and the home nodes of the leaders are selected as the base nodes. The state of an agent is *active*, *leader* or *follower*. Active agents are candidates of leaders, and initially all agents are active. As the selection phase progresses, the number of active agents decreases since some agents become followers. And finally the remaining active agents become leaders at the same time. Once an agent becomes a follower or a leader, it never changes its state.

When initialized, each agent releases its own token on its home node to notify agents that the node is the home node of an agent. The selection phase consists of $\lceil \log k \rceil$ *sub-phases*. Each sub-phase at least halves the number of active agents or makes all active agents leaders. By repeating such sub-phase $\lceil \log k \rceil$ times, some agents are selected as leaders so that their home nodes should satisfy the base node condition. Notice that all active agents may become leaders before the $\lceil \log k \rceil$-th sub-phase starts. Even in this case, all the agents spend $\lceil \log k \rceil$ sub-phases in the selection phase[1], while all the agents (leaders or followers) only keep staying at their home nodes after the leaders are selected.

We explain the details of the sub-phase. At the beginning of each sub-phase, each agent stays at its own home node. During the sub-phase, each agent traverses the ring once if it is active, or keeps staying at its home node if it is a leader or a follower. Thus, in the following, we identify the agent state with the state of its home node, that is, we say that a node v is active when the agent with home node v is active (the same for followers and leaders). To reduce the number of active agents, an ID (not necessarily unique) is assigned to each active agent. The ID of an active agent a is given as $(d, fnum)$, where d is the distance from its home node $v_h(a)$ to the next active node, say v_{next}, and $fnum$ is the number of follower nodes between $v_h(a)$ and v_{next} (Figure 5). Notice that the IDs of the same agent differ in different sub-phases since the set of active agents is reduced in every sub-phase. We compare two IDs by the lexicographical order: for $ID_1 = (d_1, fnum_1)$ and $ID_2 = (d_2, fnum_2)$, $ID_1 < ID_2$ if $(d_1 < d_2) \vee ((d_1 = d_2) \wedge (fnum_1 < fnum_2))$ holds.

Based on IDs, we reduce the number of active agents. Let a_i be an active agent and a_j be the next active agent of a_i. Let ID_i and ID_j be the IDs of a_i and a_j respectively. In each sub-phase, a_i decides whether it remains active or not in the following way:

- Case that all active agents have the same ID: All the active agents (including a_i) become leaders. Notice that the home nodes of the active agents satisfy the base node condition.
- Case that active agents have two distinct IDs or more: Agent a_i remains active if ID_i is the minimum among IDs of all active agents and $ID_i \neq ID_j$ holds. The second condition guarantees that, when active agents with the minimum ID appear consecutively, only one of them remains active. This guarantees that the number of active agents is at least halved in each sub-phase.

We explain the implementation of the sub-phase. In the sub-phase, each active agent a_i traverses the ring once and determines the state transition as above. This takes n unit times, and follower agents stay at their home nodes during the n unit times. While a_i traverses the ring, it executes the following operations.

1. Get its own ID $ID_i = (d_i, fnum_i)$: Agent a_i finds its own ID ID_i by moving from its home node $v_h(a_i)$ to the next active node v_{next} with counting the numbers of nodes and follower agents (Figure 5). Note that,

[1]This is for simplicity. A follower agent can detect that leaders have been selected if its home node is not visited by other agents during n time units. Thus, it is possible to complete the selection phase by spending one additional sub-phase after leaders are selected.

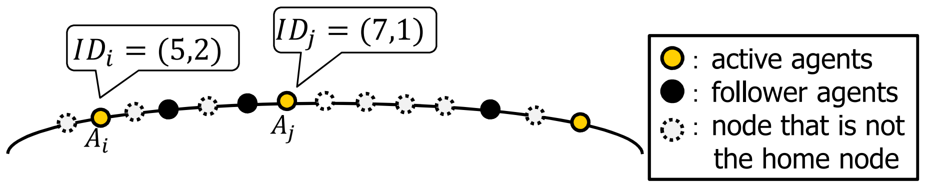


Figure 5: Algorithm 3：IDs of active agents

since all active agents are traversing the ring and all follower agents are staying at their home nodes, a_i can detect its arrival at the next active node when it visits an empty home node (having a token).

2. Get the ID of its next active agent: Similarly, a_i finds the ID of the next active agent of a_i, say a_{next}, by moving to the next active node of v_{next}. Agent a_i stores the ID of a_{next} in ID_{next}.

3. Compare its own ID with those of all active agents: During the traversal of the ring, agent a_i finds IDs of all active agents one by one, and checks 1) whether its own ID is the minimum and 2) whether the IDs of all active agents are identical. To check these, agent a_i keeps boolean variables *min* ($min = true$ implies a_i has the minimum ID) and *identical* ($identical = true$ implies that all the IDs a_i ever found are the same), and it updates the variables (if necessary) every time it finds an ID of another agent.

4. Determine its state for the next sub-phase: When a_i completes the traversal, it determines its state for the next sub-phase. If $identical = true$ holds, a_i changes its state to the leader (in this case, all active agents become leaders). In the case of $identical = false$, a_i remains active if $min = true$ and $ID_i < ID_{next}$ hold. Otherwise, a_i becomes a follower.

Algorithm 3 presents the pseudocode. In the first sub-phase, each agent finds the number n of nodes, but the code for findning n is omitted in the pseudocode. The following theorem holds.

Theorem 4 Algorithm 3 solves the uniform deployment problem in the token model. The algorithm requires $O(1)$ memory for each node, $O(\log n)$ memory for each agent, $O(n \log k)$ time, and $O(kn)$ total number of agent moves.

Proof. It is obvious that Algorithm 3 solves the uniform deployment problem.

Each agent has three variables, ID_i, ID_{next}, and ID_{other}, to store IDs, each of which requires $O(\log n)$ memory. Since other variables require $O(\log n)$ memory or less, each agent requires $O(\log n)$ memory.

The time complexity is $O(n \log k)$ because the selection phase requires $n\lfloor \log k \rfloor$ unit times and the deployment phase requires at most $2n$ unit times.

Lastly, we consider the total number of moves. First, consider the selection phase. In each sub-phase, each active agent traverses a ring once, and then at least half active agents become followers or all active agents become leaders. Hence, in the beginning of the x-th sub-phase, the number of active agents is at most $k/2^{x\ \ 1}$. Since follower agents and leader agents never move in the selection phase, the total number of moves in the selection phase is at most $\sum_{1\ \ x\ \ \log k}(k/2^{x\ \ 1})n \quad 2kn$. In the deployment phase, each follower moves to a target node to achieve the uniform deployment. Each follower moves at most $2n$ times since it first moves to the nearest base node, which requires at most n moves, and then moves to a vacant target node, which requires at most n moves. Thus, the total number of moves in the deployment phase is $O(kn)$. Therefore, the total number of agent moves is $O(kn)$. □

The restriction of $n = ck$ imposed in the above can be removed by the similar modification to that in Algorithm 2.

6 Conclusion

In this paper, we considered the uniform deployment of mobile agents in synchronous ring networks. The uniform deployment problem, which is a striking contrast to the rendezvous problem, is interesting to investigate.

Algorithm 3 A uniform deployment algorithm with $O(\log n)$ agent memory on the token model

Behavior of an agent. Let v_i be its home node.

```
1: /* selection phase */
2: $state$ := active ;    // The agent is active.
3: Release a token at the home node $v_i$ ;
4: for $s = 1$ to $\lceil \log k \rceil$
5:   $state$ := subPhase($state$) ;    // Traverse the ring and change its state.
6: /* deployment phase */
7: if $state$ = follower then
8:   Move to the first node where another agent is staying ;    // Move to a base node.
9:   while $true$ do
10:     Move $n/k$ times ;    // Move to the next target node.
11:     if no agent is staying at the currently visiting node then break;    // A vacant target node.
12: Terminate the algorithm.
function : subPhase($state$)    // $state \in \{$active, follower, leader$\}$
13: if $state$ = follower or leader then
14:   Wait for $n$ unit times ;    // Wait for termination of the sub-phase.
15:   return $state$;
16: Move to the next active node and get its own ID $ID_i$ ;
17: if $a_i$ stays at $v_i$ then  // Only the agent is active.
18:   return leader;
19: Move to the next active node and get the ID $ID_{next}$ of the next active agent.
20: if $ID_i > ID_{next}$ then $min := false$; $identical := false$;
21: else if $ID_i = ID_{next}$ then $min := true$; $identical := true$;
22: else $min := true$; $identical := false$;
23: while the currently visiting node is not $v_i$ do
24:   Move to the next active node and get $ID_{other}$ ;
25:   if $ID_i > ID_{other}$ then $min := false$; $identical := false$; // There exists an agent with a smaller ID.
26:   else if $ID_i < ID_{other}$ then $identical := false$;
27: if $identical = true$ then return leader ;    // All active agents have the identical ID.
28: else if $min = true$ and  $ID_i < ID_{next}$ then return active ;  // The agent remains active.
29: else return follower ;    // Otherwise, the agent become a fllower.
```

We proposed three algorithms that are asymptotically optimal in terms of the total number of agent moves. Especially the latter two algorithms utilize the essential charactersitic of the uniform deployment problem: the problem aims to attain the symmetry, and these algorithms solve the problem without breaking symmetry that the initial agent locations have. Such an approach in designing mobile agent algorithms seems to be applicable to other problems that aim to attain the symmetry.

Reference

[1] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson and D. Rus. D'Agents: Applications and performance of a mobile-agent system *Software: Practice and Experience*, Vol. 32, Issue 6, pages 543-573, May 2002.

[2] J. Baumann, F. Hohl, K. Rothermel and M. Straber. Mole - Concepts of a mobile agent system. *WORLD WIDE WEB*, Vol.1, Number 3, pages 123-127, 1998.

[3] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, Vol. 42, Issue. 3, pages 88-89, March 1999.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination for distributed network management. *Journal of Network and Systems Management*, Vol. 9, No. 4, pages 435-456, 2001.

[5] T. Suzuki, T. Izumi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Move-optimal gossiping among mobile agents. *Theoretical Computer Science*, Vol. 393, Issues 1-3, pages 90-101, March 2008.

[6] E. Kranakis, D. Krizanc and E. Marcou. *The Mobile Agent Rendezvous Problem in the Ring*. MORGAN & CLAYPOOL PUBLISHERS, 2010, 122pages.

[7] P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro and C. Sawchuk. Multiple mobile agent rendezvous in a ring. LATIN 2004, LNCS 2976, pp.599-608, 2004.

[8] E. Kranakis, D. Krizanc and E. Marcou. *Mobile agent rendezvous in a synchronous torus*. LATIN 2006, LNCS 3887, pp. 653-664, 2006.

[9] P. Fraigniaud and A. Pelc. *Deterministic rendezvous in trees with little memory*. Proc. of DISC, pp. 242-256, 2008.

[10] J. Czyzowicz, A. Kosowski and A. Pelc. *How to Meet You Forget: Log-speace Rendezvous in Arbitrary Graphs*. Proc. of PODC, pp. 450-459, 2010.

[11] Y. Elor and A. M.Bruckstein. Uniform multi-agent deployment on a ring. *Theoretical Computer Science*, Vol. 412, Issues 8-10, pages 783-795, March 2011.

A Appendix

A.1 Proof of Theorem 2

Theorem 2 Algorithm 1 solves the uniform deployment problem in the whiteboard model. The algorithm requires $O(\log k)$ memory for each node, $O(\log n)$ memory for each agent, $O(n)$ time, and $O(kn)$ total number of moves.

Proof. It is obvious that Algorithm 1 solves the uniform deployment problem.

In the selection phase, a traverses the ring once, which takes n unit times and n moves. Each agent moves to a target node in the deployment phase, which takes at most n unit times and n moves. Thus the time complexity is $O(n)$ and the total number of moves is $O(kn)$.

Each agent needs to keep an agent ID (or a candidate of the minimum ID) in variable *minid*, the distance to a candidate of the base node in *dis*, the number of nodes it ever visited in *nodenum*, and the number of agent IDs it ever found in *agentnum*. Thus, each agent requires $O(\log n)$ memory.

The whiteboard of a node has to store an agent ID when it is the home node of the agent, and thus it requires $O(\log k)$ memory. □

A.2 Proof of Theorem 3

Theorem 3 Algorithm 2 solves the uniform deployment problem in the token model. The algorithm requires $O(1)$ memory for each node, $O(k \log n)$ memory for each agent, $O(n)$ time, and $O(kn)$ total number of moves.

Proof. It is obvious that Algorithm 2 solves the uniform deployment problem.

In the selection, each agent traverses the ring once to get D, which takes n unit times and n moves. In the deployment phase, each agent moves to its own target node, which takes at most n unit times and n moves. Thus the time complexity is $O(n)$ and the total number of moves is $O(kn)$.

Each agent stores the sequence of distances $D = (d_0, d_1, \ldots, d_{k\ 1})$. Since d_i is at most n, this requires $O(k \log n)$ memory. Since other variables require at most $O(\log n)$ memory, each agent requires $O(k \log n)$ memory. It is clear that each node requires $O(1)$ memory since it is the token model. □