第 11 回
情報科学ワークショップ

The 11th Workshop on Theoretical Computer Science, Kitanagoya, Aichi, September 2015 (WTCS2015)

片山 喜章
泉 泰介
金 鎔煥

主担当：名古屋工業大学

序言

本論文集は 2015 年 9 月 16 日～18 日，北名古屋市タナベ名古屋研修センターにて開催された第 11 回情報科学ワークショップでの発表原稿をまとめたものである．

本ワークショップは，日本の並列／分散計算の研究者が研究室以上の議論と研究会以下のフォーマルさを目指し，合宿形式でお互いを徹底的に切りまくる「ちゃんばら大会」を目的とし，大阪大学，九州大学，九州工業大学，京都工芸繊維大学，名古屋工業大学，奈良先端科学技術大学院大学の研究者有志によって 2005 年に始まったものである．第 1 回は 2005 年 9 月に出雲で開催して以来，第 2 回瀬戸，第 3 回門司，第 4 回長浜，第 5 回広島，第 6 回桑名，第 7 回福岡，第 8 回神戸，第 9 回唐津， 第１０回福山に続き，今回で 11 回目の開催となる．

今回は大阪大学，名古屋工業大学，九州大学，九州工業大学，国立情報学研究所，豊橋技術科学大学，奈良先端科学技術大学院大学，広島大学，法政大学から 48 人の参加者があり，35 件の発表が行われた．今回の開催場所は，愛知県北名古屋市のタナベ名古屋研修センターで，愛知県の中心である名古屋市や「尾張の小京都」と称される歴史のある都市の犬山市に隣接した立地の良い場所である．このような愛知県の文化と歴史に囲まれた環境で夜遅くまで活発な討論が行われ，有意義な時間を共有することができた．今回からは，学生をはじめとする若手研究者たちのモチベーションの向上を念頭に，参加した大学教員らの審査により，優れた研究に対して優秀研究賞を，さらに素晴らしいプレゼンテーションを行った学生を対象にプレゼンテーション賞を授与した．本予稿集に受賞者一覧を掲載している．受賞者らには今後益々の活躍を，そして未受賞者諸君には次の受賞者を目指して更なる研鑽を積んでいただきたいと思う．なお 2016 年度は法政大学・工学院大学を中心とした東京チームが主担当の予定である．

最後に，今回の開催にあたりご協力をいただいた皆様，ワークショップ運営にご協力をいただいた参加者の皆様に厚く御礼申し上げる．

2015 年 12 月

片山 喜章

泉 泰介

金 鎔煥

目次

9月16日 （12：30～18：10）

12:30				開会（10min）	
分散アルゴリズム1				座長 ： 亀井 清華	
番号	時間	名前	所属	タイトル	発表時間
A-1	12:40	柴田 将拡	大阪大学	非同期リングにおけるモバイルエージェント均一配置アルゴリズム	ミドル
A-2	13:00	難波 瑛次郎	大阪大学	匿名単方向リングネットワークにおけるモバイルエージェント集合問題に対するトークンを用いた乱択アルゴリズ	ミドル
A-3	13:20	片岡 大輝	名工大	個体群プロトコルによる1-区間連結性の模倣	ミドル
Coffee Break (15min)					
分散アルゴリズム2				座長 ： 山内 由紀子	
番号	時間	名前	所属	タイトル	発表時間
B-1	13:55	小野 優也	大阪大学	無線ビープネットワークにおける極大マッチングアルゴリズム	ミドル
B-2	14:15	安達 駿	大阪大学	モバイルノードを利用したトリガ数え上げアルゴリズム	ミドル
B-3	14:35	小森 康祐	大阪大学	Time-varying Graph における1 対1 のk トークン転送アルゴリズム	ミドル
B-4	14:55	渡部 連太郎	大阪大学	部分スナップショットアルゴリズムの効率的な並行実行の実現	ミドル
Coffee Break (15min)					
システム・サービス				座長 ： 金 鎔煥	
番号	時間	名前	所属	タイトル	発表時間
C-1	15:30	中村 純哉	豊橋技科大学	コンテナ型仮想化を利用した学内向けWebホスティングサービスの更新	ミドル
C-2	15:50	土中 哲秀	九州大学	産業連関ネットワーク解析のための疎化処理と閾値の関係について	ミドル
C-3	16:10	房 冠深	広島大学	時間特徴ベクトルの生成手法と観光地推薦システムへ応用	ショート
C-4	16:25	山中 景太	広島大学	手描き情報共有システムの開発	ショート
Coffee Break (15min)					
自己安定				座長 ： 泉 泰介	
番号	時間	名前	所属	タイトル	発表時間
D-1	16:55	Anissa Lamal	九州大学	Self-stabilizing Oscillation in Population Protocols	ミドル
D-2	17:15	中川 遼	大阪大学	Proof-Labeling スキームに基づく故障封じ込め自己安定アルゴリズムの提案	ミドル
D-3	17:35	圃孝 直人	大阪大学	不安定な仮想グリッド環境下におけるzigzag プロトコル適用手法とシミュレーション結果	ミドル
D-4	17:55	清洲 星顕	大阪大学	個体群プロトコルモデルにおける緩自己安定リーダ選挙のシミュレーション評価	ショート

9月17日 （8：30～11：45）

計算理論1				座長 ： 中村 純哉	
番号	時間	名前	所属	タイトル	発表時間
E-1	8:30	小野 廣隆	九州大学	一般化ジャンケンに対するゲーム理論的解析	ミドル
E-2	8:50	馬路 裕光	名工大	Listing Center Strings under the Edit Distance Metric	ミドル
E-3	9:10	本田 巧	広島大学	GPU向け多倍長整数乗算	ミドル
E-4	9:30	木村哲也	法政大学	Dynamic Parallelismを用いたマルチスレッドアルゴリズムの効率的な実現について	ショート
E-5	9:45	茂木啓輔	法政大学	GPGPUにおけるシストリックアルゴリズムの効率的な実現について	ショート
Coffee Break (15min)					
計算理論2				座長 ： 小野 廣隆	
番号	時間	名前	所属	タイトル	発表時間
F-1	10:15	小池 敦	総研大	GPUを用いたTree-Reduction計算の高速化	ミドル
F-2	10:35	藤田 徹	広島大学	GPUを用いたライフゲームの高速化	ミドル
F-3	10:55	船坂 峻慈	広島大学	A parallel algorithm for LZW decompression, with GPU implementation	ミドル
F-4	11:15	椎葉知泰	九州工業大学	Asynchronous enzymatic numerical P systems for the compare-and-exchange and sor	ショート
F-5	11:30	西田豊	九州工業大学	Asynchronous membrane computing for the compare-and-exchange and sorting	ショート

9月18日 （8：30～12：00）

ロボット1				座長 ： 大下 福仁	
番号	時間	名前	所属	タイトル	発表時間
H-1	8:30	山内由紀子	九州大学	3次元空間中のロボットの合意形成：平面形成とパターン形成	ミドル
H-2	8:50	八神貴裕	九州大学	複数のエージェントによるオンライン木探索アルゴリズム	ミドル
H-3	9:10	寺井智史	法政大学	状態を持つ自律分散ロボット群を用いた集合問題の可解性について	ミドル
H-4	9:30	田邊太一	法政大学	自律分散ロボット群の理論モデルに対する実機シミュレーションに関する研究	ミドル
Coffee Break (15min)					
ロボット2				座長 ： 伊藤 靖朗	
番号	時間	名前	所属	タイトル	発表時間
I-1	10:05	貝野 太地	名工大	ライトを保持した自律分散ロボット群による基地局集合問題について	ミドル
I-2	10:25	白川遥平	法政大学	正六角形グリッド上でのファットロボットの集合問題について	ミドル
I-3	10:45	奥村太加志	法政大学	非同期モデルのロボットによる、半同期モデルのシミュレートについて	ショート
I-4	11:00	冨田祐作	九州大学	キラリティのない分散ロボットの平面形成問題	ショート
I-5	11:15	門出顕宏	九州大学	限られた視界を持つ自律ロボット群による線分被覆問題	ショート
11:30				表彰式・閉会（30min）	

受賞者一覧

優秀研究賞（発表順）

- 柴田 将拡 (大阪大学), 大下 福仁 (奈良先端大), 角川 裕次 (大阪大学), 増澤 利光 (大阪大学), ″非同期リングにおけるモバイルエージェント均一配置アルゴリズム″.

- 土中 哲秀 (九州大学), 小野 廣隆 (九州大学), ″産業連関ネットワーク解析のための疎化処理と閾値の関係について″.

優秀プレゼンテーション賞（発表順）

- 小森 康祐 (大阪大学), ″Time-varying Graph における 1 対 1 の k トークン転送アルゴリズム″.

- Anissa Lamani (九州大学), ″Self-stabilizing Oscillation in Population Protocols”.

- 本田 巧 (広島大学), ″GPU 向け多倍長整数乗算″.

- 小池　敦 (総研大), ″GPU を用いた Tree-Reduction 計算の高速化″.

- 藤田 徹 (広島大学), ″GPU を用いたライフゲームの高速化″.

- 八神貴裕 (九州大学), ″複数のエージェントによるオンライン木探索アルゴリズム″.

- 片岡 大輝 (名古屋工業大学), ″個体群プロトコルによる 1-区間連結性の模倣″.

セッション１

分散アルゴリズム１

非同期リングにおけるモバイルエージェント均一配置アルゴリズム

Masahiro Shibata †, Fukuhito Ooshita◇, Hirotsugu Kakugawa†, and Toshimitsu Masuzawa†

†Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
{m-sibata, kakugawa, masuzawa}@ist.osaka-u.ac.jp
◇Graduate School of Information Science, NAIST
Takayama 8916-5, Ikoma, Nara 630-0192, Japan
f-oosita@is.naist.jp

Abstract. In this paper, we consider the uniform deployment problem of mobile agents in asynchronous uniditrectional ring networks. The uniform deployment problem requires agents to uniformly spread in finite time. In this paper, we consider the uniform deployment problem for the case that agents know neither the number of nodes nor agents. At first we show that, when termination detection is required, there exists no algorithm to solve the uniform deployment problem. For this reason, we consider the relaxed uniform deployment problem that does not require termination detection, and we propose an algorithm to solve the relaxed uniform deployment problem. This algorithm requires $O(k/\ell \log(n/\ell))$ memory per agent, $O(n/\ell)$ time, and $O(kn/\ell)$ total moves, where n is the number of nodes, k is the number of agents, and ℓ is the symmetry of the initial configuration ($\ell \geq 1$). Note that both the algorithms achieve the uniform deployment from any initial configuration, which is a striking difference from the rendezvous problem because the rendezvous problem is not solvable from some initial configurations.
keyword: distributed system, mobile agent, uniform deployment, ring network, token

1 Introduction

A *distributed system* is a system that consists of a set of computers (*nodes*) and communication links. Recently, distributed systems have become large and design of distributed systems has become complicated. As a way to design distributed systems, (mobile) agents have attracted a lot of attention [1]. Agents can traverse the system and process tasks on each node, and hence they can simplify design of distributed systems [2].

Many fundamental problems for cooperation of mobile agents have been studied. For instance, Suzuki et al. [3] 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*, which requires all agents to meet at a single node. The rendezvous problem is considered in rings [4, 5], torus [6], trees [7], and arbitrary networks [8]. Some works assume that agents can use whiteboard on each node, and others assume that agents can use only tokens, which are markers that agents can release on nodes. Chalopin et al. [9] considered *black hole search*, which makes agents locate a particularly dangerous node called black hole. They investigated the minimum resources (the numbers of agents and tokens) necessary for locating all links incident to the black hole.

In this paper, we consider the *uniform deployment* (or *uniform scattering*) problem, which requires all agents to spread uniformly. From a practical point of view, the uniform deployment is useful for the network management. For instance, if agents that can repair faulty nodes are deployed uniformly, such agents can quickly reach and repair faulty nodes after the faults are detected. If agents with database replicas 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 sometimes impossible) to attain in distributed systems. Consequently, it is interesting to clarify how easily the uniform deployment can be attained compared to the rendezvous.

As related works, Flocchini et al. [10] and Elor et al. [11] considered the uniform deployment problem in ring networks, while Barriere et al. [12] considered it in grid networks. All of them propose uniform deployment algorithms under the assumption that agents are oblivious (or memoryless) but can observe multiple nodes within its visibility range. On the other hand, Mega et al. [13] considered the uniform deployment problem for agents that have memory but cannot observe nodes except for their currently visiting nodes. They assumed

Table 1. Results in each model

	First result in [13]	Second result in [13]	Model 1	Model 2
Knowledge of k	Available	Available	Not Available	Not Available
Termination detection	Required	Required	Required	Not Required
Solvable / Unsolvable	Solvable	Solvable	Unsolvable	Solvable
Agent memory	$O(k \log n)$	$O(n)$	-	$O(k/\ell \log(n/\ell))$
Time complexity	$O(n)$	$O(n \log k)$	-	$O(n/\ell)$
Total moves	$O(kn)$	$O(kn)$	-	$O(kn/\ell)$

n: the number of nodes, k: the number of agents, ℓ: the symmetry of the initial configuration

agents with knowledge of the number of agents and proposed two move-optimal algorithms to solve the uniform deployment problem in unidirectional synchronous ring networks. In addition to the two algorithms, they considered the trade-off between the time complexity and the memory requirement par agent.

In this paper, we consider the uniform deployment problem in unidirectional asynchronous ring networks. Similarly to [13], we consider agents that have memory but cannot observe nodes except for their currently visiting node. Each agent initially has a token and can release it on a visiting nodes. After a token is released at some node, agents cannot remove such a token. Different from [13], we assume that agents have no knowledge of the number of agents or nodes. At first we show that, when termination detection is required, there exists no algorithm to solve the uniform deployment problem. Intuitively, it is due to impossibility of finding k or n when the initial configuration has symmetry: when an agent misestimates these at smaller numbers than actual ones, it prematurely terminates and the uniform deployment cannot be achieved. For this reason, we consider the relaxed uniform deployment problem that does not require termination detection, and we propose an algorithm to solve the relaxed uniform deployment problem. In this algorithm, each agent estimates k and n (possibly at smaller values than actual ones) and behaves based on the estimation. Thus, the efficiency of the algorithm depends on the estimation. To evaluate the efficiency, we introduce the following parameter ℓ to denote the symmetry degree of an initial configuration: we say that an initial configuration has symmetry ℓ when the its distance sequence can be represented as ℓ-times repetition of some non-periodic sequence. For example, an asymmetric initial configuration has symmetry 1, and the symmetry becomes larger for a higher symmetric initial configuration. Note that agents cannot know ℓ but the efficiency depends on it. Using the symmetry parameter ℓ, the efficiency of the algorithm is denoted as follows: this algorithm requires $O(k/\ell \log(n/\ell))$ memory per agent, $O(n/\ell)$ time, and $O(kn/\ell)$ total moves. This result shows a natural but interesting property: the algorithm achieves the uniform deployment more efficiently when the initial configuration has higher symmetry. For an asymmetric initial configuration, this algorithm requires $O(k \log n)$ memory per agent, $O(n)$ time, and $O(kn)$ total moves. However, when ℓ is $\omega(1)$, this algorithm requires $o(k \log n)$ memory per agent, $o(n)$ time, and $o(kn)$ total moves. When ℓ is $\Omega(n)$, this algorithm requires $O(1)$ memory per agent, $O(1)$ time, and $O(k)$ total moves.

Note that all proposed algorithms achieve the uniform deployment from any initial configuration, which is a striking difference from the rendezvous problem because the rendezvous problem is not solvable from some initial configurations. Due to limitation of space, we describe several proofs of lemmas and theorems in the appendix. In Table 1, we compare our contributions with results in [13].[1]

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 anonymous nodes (i.e, nodes having no IDs) and E is a set of unidirectional links. We denote by $n\,(= |V|)$ the number of nodes. Then, we define $V = \{v_0, v_1, \ldots, v_{n-1}\}$ 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}. We define the direction from v_i to v_{i+1} as the *forward* direction. In addition, we define the j-th $(j \neq 0)$ forward agent a' of agent a as the agent that exists in the a's forward direction and there are $j-1$ agents between a and a'. Moreover, the *distance* from v_i to v_j $(0 \leq i, j \leq n-1)$ is defined to be $(j - i) \bmod n$.

An agent is a state machine having an *initial state*. Let $A = \{a_0, a_1, \ldots, a_{k-1}\}$ be a set of $k\,(\leq n)$ agents. For simplicity, operations to an index of an agent assume calculation under modulo k. Since the ring is unidirectional, agents staying at v_i can move only to v_{i+1}. We assume that agents are anonymous (i.e., agents have no IDs). In addition, each agent initially has a 1-bit memory called *token* and can release it on a node that it visits. After a token is released at some node, agents cannot remove the token. Note that since agents are anonymous, they cannot recognize the owner of each token. Moreover, we assume that agents can send a message of any size to

[1] The model in [13] can be easily to applied to our paper.

any agent at the same node. We consider two types of agents: agents with knowledge of k and agents with no knowledge of k or n.

Each agent executes the following five operations in an atomic action: 1) The agent reaches a node, say v (or it starts operations at v), 2) the agent receives messages (if any), 3) the agent executes local computation at v, 4) the agent sends a message to agents v (if any) if it decides to send a message, and 5) the agent leaves v if it decides to move. We assume that agents move through a link in a FIFO manner, that is, when agent a_p leaves v_i after agent a_q leaves v_i, a_p reaches v_{i+1} after a_q reaches v_{i+1}. We consider an *asynchronous* system, that is, the time for each agent to perform an operation is finite but unbounded.

A (global) *configuration* C is defined as a 5-tuple $C = (S, T, M, P, Q)$. The first element S is a k-tuple $S = (s_0, s_1, \ldots, s_{k-1})$ representing the agent states where s_i is the state (including the state of holding a token or not) of $a_i\,(0 \leq i \leq k-1)$. The second element T is an n-tuple $T = (t_0, t_1, \ldots, t_{n-1})$ denoting the node states where t_i is the state (i.e., the number of tokens) of $v_i\,(0 \leq i \leq n-1)$. The third element M is a k-tuple $M = (m_0, m_1 \ldots m_{k-1})$, where m_i is a sequence of messages that are sent to a_i and not received by a_i. The remaining elements P and Q represent the positions of agents. The element P is an n-tuple $P = (p_0, p_1, \ldots, p_{n-1})$, where p_i is a set of agents staying at node v_i $(0 \leq i \leq n-1)$. The element Q is an n-tuple $Q = (q_0, q_1, \ldots, q_{n-1})$, where q_i is a sequence of agents residing in the FIFO queue corresponding to link (v_{i-1}, v_i) $(0 \leq i \leq n-1)$. Hence, agents in q_i are those in transit from v_{i-1} to v_i.

We denote by $\mathcal{C}$ the set of all the possible configurations. In *initial configuration* $C_0 \in \mathcal{C}$, all agents are in the initial states and placed at distinct nodes respectively, and no node has any token. In addition, in C_0 the node where agent a stays is called the *home node* of a and denoted by $v_{HOME}(a)$. We assume that in C_0 agent a is in the head of queue q_i if v_i is the home node of a. This assures that agent a starts the algorithm at $v_{HOME}(a)$ before any other agent visits $v_{HOME}(a)$, that is, a is the first agent that takes an action at $v_{HOME}(a)$.

In addition, we define *periodic rings*. For initial configuration C_0, we define the *distance sequence* of agent a_i as $D_i(C_0) = (d^i_0(C_0), \ldots, d^i_{k-1}(C_0))$, where $d^i_j(C_0)$ is the distance from the j-th forward agent of a_i to the $(j+1)$-th forward agent of a_i in C_0. Then, we define the distance sequence of configuration C_0 as the lexicographically minimum sequence among $\{D_i(C_0) | a_i \in A\}$, and we denote it by $D(C_0)$. In addition, let $\mathit{shift}(D, x) = (d_x, d_{x+1}, \ldots, d_{k-1}, d_0, d_1, \ldots, d_{x-1})$ for sequence $D = (d_0, d_1, \ldots, d_{k-1})$. Then, when $D(C_0) = \mathit{shift}(D(C_0), x)$ holds for some x such that $0 < x < k$ holds, we say the ring is *periodic*. Otherwise, we say the ring is *not periodic*.

A *schedule* is an infinite sequence of agents. A schedule $X = \rho_1, \rho_2, \ldots$ is fair if every agent appears in X infinitely often. An infinite sequence of configurations $E = C_0, C_1, \ldots$ is called an *execution* from C_0 if there exists a fair schedule $X = \rho_1, \rho_2, \ldots$ that satisfies the following conditions for each h $(h > 0)$:

- If $\rho_h \in p_i$ holds for some i in a configuration C_h, the states of ρ_h and v_i in C_{h-1} are changed to those in C_h by a local computation of ρ_h. Let $a_j = \rho_h$. If $m_j \neq \emptyset$, all messages in m_j are delivered to a_j and consumed, that is, m_j becomes $\emptyset$. In addition if ρ_h sends a message to some agent a_l at v_i, the message is appended to the tail of sequence m_l. Moreover if ρ_h releases its token at v_i, t_i changes, that is, the number of tokens at v_i increments. After this if ρ_h decides to move to v_{i+1}, ρ_h is removed from p_i and is appended to the tail of sequence q_{i+1}. If ρ_h decides to stay, ρ_h is still in p_i. The other elements in C_{h-1} are the same as those in C_h.

- If ρ_h is at the head of q_i for some i in a configuration C_h, ρ_h moves to v_i, that is, ρ_h is removed from q_i. Then, the states of ρ_h and v_i in C_{h-1} are changed to those in C_h by a local computation of ρ_h. If ρ_h sends a message to some agent a_l at v_i, the message is appended to the tail of sequence m_l. Moreover if ρ_h releases its token at v_i, t_i changes, that is, the number of tokens at v_i increments. After this if ρ_h decides to move to v_{i+1}, ρ_h is appended to the tail of sequence q_{i+1}. If ρ_h decides to stay, ρ_h is inserted in p_i. The other elements in C_{h-1} are the same as those in C_h.

- Otherwise, C_{h-1} is the same as C_h.

2.2 The uniform deployment problem

The uniform deployment problem requires $k\,(\geq 2)$ agents to spread uniformly in the ring, that is, the distance between any two *adjacent agents* should become identical like Fig. 1. Here, we say two agents are adjacent when there exists no agent between them. However, we should consider the case that n is not a multiple of k. In this case, we aim to distribute the agents so that the distance d of two adjacent agents should satisfy $\lfloor n/k \rfloor \leq d \leq \lceil n/k \rceil$.

We consider *the uniform deployment problem with termination detection* and *the uniform deployment problem without termination detection*. At first, we define the uniform deployment problem with termination detection. In this case, *a halt state* is defined as follows: when agent a_i enters a halt state, it terminates the algorithm, that is, a_i neither changes its state nor leaves the current node even if another agent sends a message to a_i. Hence

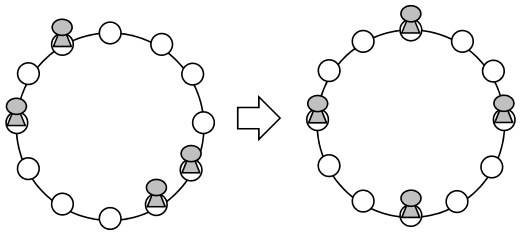


Fig. 1. An example of the uniform deployment ($n = 16, k = 4, d = 3$)

if an agent enters a halt state, it can detect its termination. Now, we define the uniform deployment problem with termination detection as follows.

Definition 1. *An algorithm solves the uniform deployment problem* with *termination detection if any execution satisfies the following conditions.*

- *All agents change their states to halt states in finite time.*
- *When all agents are in the halt states, $q_i = \emptyset$ holds for any $q_i \in Q$ and each distance d of two adjacent agents satisfies $\lfloor n/k \rfloor \leq d \leq \lceil n/k \rceil$.*

Next, we define the uniform deployment problem without termination detection. In this case, *a suspended state* is defined as follows: when agent a_i enters a suspended state, it neither changes its state nor leaves the current node unless another agent sends a message to a_i. If a_i receives a message, it can resume its behavior and leave the current node. The uniform deployment problem without termination detection allows agents to stop in suspended states.

Definition 2. *An algorithm solves the uniform deployment problem* without *termination detection if any execution satisfies the following conditions.*

- *All agents change their states to suspended states in finite time.*
- *When all agents are in the suspended states, $q_i = \emptyset$ holds for any $q_i \in Q$ and each distance d of two adjacent agents satisfies $\lfloor n/k \rfloor \leq d \leq \lceil n/k \rceil$.*

For the uniform deployment problem, we have the following lower bound of total moves.

Theorem 1. *When $k \leq cn$ holds for some constant $c\,(c < 1)$, a lower bound of the total moves to solve the uniform deployment problem (with or without termination detection) is $\Omega(kn)$ even if agents have knowledge of k.*

Proof. We consider the initial configuration such that all agents stay in a quarter part of the ring like Fig. 2. In this case, the ring is divided into four quarter parts, and in the initial configuration, all agents are in the part a (we assume $k \leq n/4$). To achieve the uniform deployment, $k/4$ agents need to move to the part c, the opposite part of a, and each of them must move at least $n/4$ times. Thus the total number of moves is at least $(k/4) \times (n/4) = kn/16$. □

Next, we evaluate the *time complexity* as the time required to achieve the uniform deployment. Since there is no assumption about the period of each action of agents in asynchronous systems, it is impossible to measure the exact time. Instead we consider the *ideal time complexity*, which is defined as the execution time under the following assumptions: 1) The time required for an agent to move from a node to its neighboring node is at most one, and 2) the time required for local computation is ignored (i.e., zero). In the following, we simply use terms "time complexity" and "time" instead of "ideal time complexity". Then, we can show the following theorem similarly to Theorem 1.

Theorem 2. *A lower bound of the time complexity to solve the uniform deployment problem (with or without termination detection) is $\Omega(n)$.*

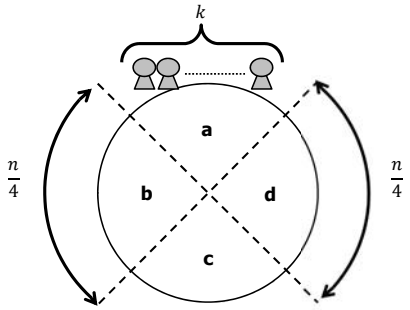


Fig. 2. The initial configuration to derive a lower bound $\Omega(kn)$ of the total moves

3 Impossibility result

In this section, we consider the uniform deployment problem with termination detection. We have the following theorem.

Theorem 3. *Even when the ring is not periodic, there exists no algorithm to solve the uniform deployment problem with termination detection.*

Proof. We prove the theorem by contradiction, that is, we assume that there exists algorithm $\mathcal{A}$ to solve the uniform deployment problem with termination detection.

At first, let us consider n-node ring R and the initial configuration C_0 such that k agents $a_0, a_1, \ldots, a_{k-1}$ exist in this order. Let $V = \{v_0, v_1, \ldots, v_{n-1}\}$ and assume that $d = n/k$ is a positive integer. From hypothesis, there is an execution E_R of $\mathcal{A}$ to solve the uniform deployment problem in R. We define $T(E_R)$ as the length of E_R and denote $E_R = C_0, C_1, \ldots, C_{T(E_R)}$. Note that in $C_{T(E_R)}$, all agents are in the halt states and every distance between two adjacent agents is d.

Next, let us consider a larger ring R' consisting of $2qn + 2n$ nodes, where q is the minimum integer such that $qn \geq T(E_R)$ holds. Let $V' = \{v'_0, v'_1, \ldots, v'_{2qn+2n-1}\}$. We consider the initial configuration C'_0 such that $kq + k$ agents $a'_0, a'_1, \ldots a'_{kq+k-1}$ exist in this order in R'. Then in R', the interval of the uniform deployment is $2d$. In addition, we define the initial position of each agent in R' as follows. Let $v_{f(i)}$ be the node where agent a_i initially stays in R. We assume that $f(i) < f(i+1)$ holds if $i \neq k-1$ and $f(i) > f(i+1)$ holds otherwise. Then, we assume that agent a'_i initially stays at node $v'_{f(i \mod k)+n\cdot\lfloor i/k \rfloor}$. That is, the initial positions for R are repeated from v'_0 to v'_{qn+n-1}, and there is no agent from v'_{qn+n} to $v'_{2qn+2n-1}$. For each node v'_j in R', we define $C_v(v'_j) = v_{j \mod n}$ as the corresponding node of v'_j in R. In the following, we show that each agent $a'_i\,(0 \leq i \leq k-1)$ behaves in the exactly same way as agent a_i in R and a'_i enters a halt state at the same time as a_i. Then, the distance between the two adjacent agents is d, which contradicts that the interval of the uniform deployment in R' is $2d$.

At first, we have the following lemma. We define the *local configuration* of node v as the 2-tuple that consists of the state of v and the states of all agents at v.

Lemma 1. *Let us consider execution $E_{R'} = C'_0, C'_1, \ldots, C'_{T(E_R)}, \ldots$ for ring R'. We define $V'_t = \{v'_t, v'_{t+1}, \ldots, v'_{qn+n-1}\}$. For any $t \leq E(T)$, configuration c'_t satisfies the following condition: for each $v'_j \in V'_t$, the local configuration of v'_j in C'_t is the same as that of $C_v(v'_j)$ in C_t.*

Proof. We prove Lemma 1 by induction on t. For $t = 0$, Lemma 1 holds from the definition of R'. Next, we show that when Lemma 1 holds for $t\,(t < T(E_R))$, Lemma 1 holds for $t + 1$.

From the hypothesis, for each $v'_j \in V'_{t+1}$ the local configurations of v'_{j-1} and v'_j in C'_t are the same as those of $C_v(v'_{j-1})$ and $C_v(v'_j)$ in C_t respectively. Hence, agents at v'_{j-1} and v'_j in C'_t behave in the exactly same way as those at $C_v(v'_{j-1})$ and $C_v(v'_j)$ in C_t. Since only agents at nodes v'_{j-1} and v'_j can change the local configuration of v'_j in unidirectional rings, the local configuration of v'_j in C'_{t+1} is the same as that of $C_v(v'_j)$ in C_{t+1}.

Therefore, we have the lemma. □

From Lemma 1, in $C'_{T(E_R)}$ local configuration of each node in $V^* = \{v'_{qn}, v'_{qn+1}, \ldots, v'_{qn+n-1}\} \subseteq V'_{T(E_R)}$ is the same as that of the corresponding node in $C_{T(E_R)}$. Note that the set of nodes corresponding to nodes in V^* is equal to V, and every agent in V^* also stops in the halt state in configuration $C'_{T(E_R)}$. Hence in $C'_{T(E_R)}$, there exist k agents in the halt states in V^*. Then, the distance between the adjacent agents in V^* is d, which is a contradiction.

Therefore, we have the theorem. □

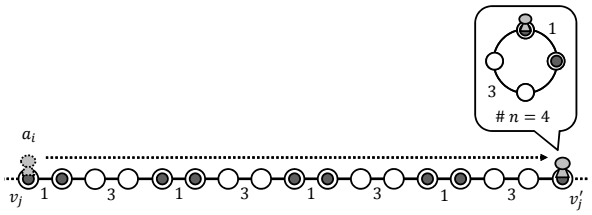


Fig. 3. An example that an agent guesses the number of nodes

4 An algorithm for uniform deployment without termination detection

4.1 Proposed algorithm

In this section, we propose an algorithm to solve the uniform deployment problem without termination detection and we show that this algorithm requires $O(k/\ell \log(n/\ell))$ memory per agent, $O(n/\ell)$ time, and $O(kn/\ell)$ total moves, where ℓ is the symmetry of the initial configuration. At first, we consider the case that the ring is not periodic. After this, we show that our proposed algorithm achieves the uniform deployment also in periodic rings.

4.1.1 Case for non-periodic rings The algorithm consists of three phases: guessing phase, patrolling phase, and deployment phase. In the guessing phase, each agent a_i moves in the ring and guesses the number of nodes. At the end of this phase, we can show that at least one agent guesses the correct number n of nodes. In the patrolling phase, a_i moves in the ring several times depending on its guessed number of nodes. During the movement, if a_i visits the node where another agent exists, a_i sends its guessed number of nodes (with some information) to the agent. By this behavior, we can show that every agent eventually gets the correct number n of nodes and its correct target node. In the deployment phase, a_i moves to its target node and enters a suspended state. After this, if a_i receives a message and recognizes that it mistakenly guesses the number of nodes, a_i decides its new target node from the message and moves there. For simplicity we assume $n = ck$ for some positive integer c in the following description, and this restriction can be removed Appendx A. In addition for sequence Y, we define $Y^1 = Y$ and $Y^{l+1} = Y^l \cdot Y$.

Guessing phase. In the guessing phase, each agent a_i firstly releases its token at its home node. After this, a_i moves in the ring and memorizes the distance *dis* between two adjacent token nodes. Agent a_i continues such a behavior until it completes guessing the number of nodes. Concretely, a_i continues to move until it observes the same distance sequence four times consecutively. After this, a_i considers it travelled four times around the ring and guesses the number of nodes: if a_i observes the same distance sequence four times consecutively when a_i visits $4n'$ nodes, a_i guesses n' as the number of nodes. For example, let us consider Fig. 3. Each number in the figure represents the distance between two adjacent token nodes. Agent a_i moves from node v_j to v_j' and gets the distance sequence $D = (1, 3, 1, 3, 1, 3, 1, 3) = (1, 3)^4$. Then, a_i guesses 4 as the number of nodes. By this behavior, we can show that 1) at least one agent guesses the correct number n of nodes, and 2) if the guessed number n' is not correct, $n' \leq n/2$ holds. The pseudocode is described in Algorithm 1. Variable k' represents the guessed number of agents (tokens) in the ring, and variable *nodes* represents the number of nodes a_i has ever visited.

Patrolling phase. In the patrolling phase, a_i moves $8n'$ times. Finishing this behavior, a_i considers it traveled twelve times around the ring from the beginning about its guessed number of nodes n'. During the movement, a_i may observe some agent a_h staying at some node. In this case, a_h may guess the incorrect number of nodes and prematurely stop moving at an incorrect target node. Hence if a_i observes such an agent, a_i sends $n', k', nodes$, and D to a_h. By this behavior, we can show that every agent eventually gets the correct number n of nodes and its correct target node. The pseudocode is described in Algorithm 2.

Deployment phase. In the deployment phase, agent a_i selects its target node and moves there as follows. Let $D = (d_0, d_1, \ldots, d_{k'-1})^4$ be the distance sequence a_i obtained in the guessing phase, where d_j is the distance from the j-th token node it found to the $(j+1)$-th token node. Note that, $a_i's$ home node $v_{HOME}(a_i)$ is considered as the 0-th token node. In addition, x be the minimum number such that $shift(D, x) = D_{min}$ holds, where D_{min} is the lexicographically minimum distance sequence among $\{shift(D, x) | 0 \leq x \leq k' - 1\}$. Then, a_i selects base node v_{base} where the agent whose distance sequence is D_{min} initially stays. For example in Fig. 4 (a), we assume that each agent finishes the patrolling phase and returns to its home node. Then agents select the node where agent a_0 initially exists as a base node because a_0's distance sequence is the lexicographically minimum. In addition, a_i considers that it is *rank*-th agent ($0 \leq rank \leq k' - 1$) from v_{base} (the agent staying

Algorithm 1 The behavior of agent a_i in the guessing phase

Behavior of Agent a_i

/* guessing phase */
release a token at its home node $v_{HOME}(a_i)$
while $n' = 0$ **do**
  move to the next token node and get the distance *dis* between two token nodes
  $D[i] = dis$, $i = i + 1$
  if $(i \mod 4 = 0) \wedge (\forall x\ (0 \leq x \leq i/4 - 1)$
  $D[x] = D[x + i/4] = D[x + 2 \times i/4] = D[x + 3 \times i/4])$ **then**
    // completing guessing the numbers of nodes and tokens
    $k' = i/4$
    $n' = D[0] + D[1] + \cdots + D[k' - 1]$
    $nodes = 4n'$
  end if
end while
change to the patrolling phase

Algorithm 2 The behavior of agent a_i in the patrolling phase

Behavior of Agent a_i

/* patrolling phase */
while $nodes \neq 12n'$ **do**
  move to the forward node
  $nodes = nodes + 1$
  if there exists another agent a_h **then** send $(n', k', nodes, D[])$ to a_h
end while
change to the deployment phase

at v_{base} is considered as 0-th agent). Let *disBase* be the distance between the current node and the v_{base} (if a_i already stays at v_{base}, we assume that $disBase = n'$). At first, a_i moves *disBase* times and reaches v_{base}. After this, a_i moves to its target node by moving $rank \times n/k$ times and enters a suspended state. In Fig. 4 (b), each agent firstly moves to v_{base}, moves to its target node, and enters a suspended state. When all agents enter suspended states, agents solve the uniform deployment problem.

However, a_i may stay at an incorrect target node with incorrect guessed number of nodes. In this case, a_i eventually receives a message from another agent a_ℓ. Let n'_ℓ, k'_ℓ, $nodes_\ell$, and D_ℓ be the guessed number of nodes, the guessed number of agents, the number of nodes ever visited, and the distance sequence included in a message from a_l respectively. If $n' \leq n'_\ell/2$ holds and there exists t such that $(\forall i\,(0 \leq i \leq 4k' - 1)$ $D[i] = D_\ell[i + t]) \wedge (D_\ell[0] + \cdots D_\ell[t - 1] = nodes_\ell - nodes)$ hold, it means that a_l guesses at least twice number of nodes than a_i and memorizes a_i's whole distance sequence D as a part of D_ℓ. Then, a_i recognizes that it mistakenly guesses the number of nodes and resumes its behavior. Concretely, a_i firstly moves $12n'_\ell - nodes$ times. Note that, we show later that $12n'_\ell - nodes$ is positive. Then, a_i considers it traveled twelve times around the ring from the beginning about new guessed number of nodes n'_ℓ. After this, it decides the new base node and its new target node from n'_ℓ, k'_ℓ, $nodes_\ell$ and D_ℓ, moves to its new target node as mentioned before, and enters a suspended state again. When all agents enter suspended states, agents solve the uniform deployment problem. The pseudocode is described in Algorithm 3.

An example As an example, let us consider the ring like Fig. 5. This ring is not periodic but has some *periodic subsequence*, that is, some agent observes 4-times repeated subsequence before it travels once around the ring. In such a ring, some agent mistakenly guesses the number of nodes and enters a suspended state at an incorrect target node. However in this case, we can show that at least one agent a_i guesses the correct number n of nodes and informs prematurely suspending agents of n during the patrolling phase. Let us consider the behavior of agents a_1 and a_2. For simplicity, we assume that they behave in a synchronous manner. In the guessing phase, agent a_2 gets the distance sequence $D = (1, 3, 1, 3, 1, 3, 1, 3) = (1, 3)^4$ and guesses 4 as the number of nodes, which is incorrect (Fig. 5 (a) to Fig. 5 (b)). After this a_2 executes the patrolling and deployment phases, and enters a suspended state at incorrect target node v'_j (Fig. 5 (b) to Fig. 5 (c)). On the other hand, agent a_1 is still in the guessing phase. When a_1 observes $D = (11, 1, 3, 1, 3, 1, 3, 1, 3)^4$, it completes the guessing phase and guesses the correct number of nodes 27. After this in the patrolling phase, a_1 observes a_2 at v'_j, sends its guessed number of nodes with other information to a_2 (Fig. 5 (c) to Fig. 5 (d)), and moves to its target node. When a_2 receives the message from a_1, it recognizes that it mistakenly guesses the number of nodes and resumes its behavior.

In the following, we show that every agents eventually gets the correct number n of nodes and its correct target node. To show this, we use the following lemmas.

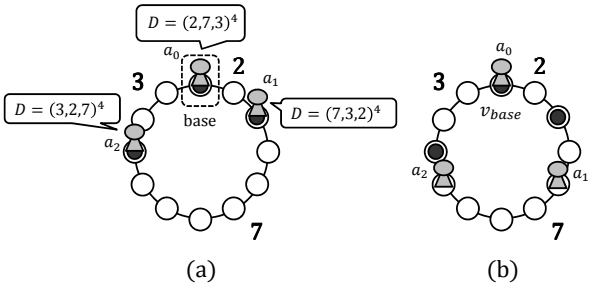


Fig. 4. An example of the deployment phase ($n = 12, k = 3, d = 4$)

Algorithm 3 The behavior of agent a_i in the deployment phase

Behavior of Agent a_i

```
 1: /* deployment phase */
 2: let D_min be the lexicographically minimum sequence among {shift(D, x)|0 ≤ x ≤ k' − 1}
 3: rank = min{x ≥ 0|shift(D, x) = D_min}
 4: disBase = D[0] + D[1] + ··· + D[k − 1 − rank]
 5: move disBase times
 6: nodes = nodes + disBase
 7: move rank × n/k times
 8: nodes = nodes + rank × n/k
 9: change its state to a suspended state
10:
11: /* behavior in the suspended state */
12: wait at the current node until a_i receives (n'_ℓ, k'_ℓ, nodes_ℓ, D_ℓ[]) from some agent a_ℓ
13: if (n' ≤ n'_ℓ/2) ∧ (there exists t such that (∀i (0 ≤ i ≤ 4k' − 1)
    D[i] = D_ℓ[i + t]) ∧ (D_ℓ[0] + ··· D_ℓ[t − 1] = nodes_ℓ − nodes) hold) then
14:     // a_i recognizes that it misunderstands the number of nodes
15:     n' = n'_ℓ, k' = k'_ℓ, D[] = shift(D_ℓ[], t)
16:     move 12n' − nodes times
17:     nodes = 12n'
18:     go to line 2
19: end if
```

Lemma 2. [14] *Consider an p-length sequence $A = a_0, \ldots, a_{p-1}$ and an p'-length sequence $B = b_0, \ldots, b_{p'-1}$ such that $p' < p$ holds. If B^3 is the prefix of A^3, either $p' \leq p/2$ holds or B is periodic.*

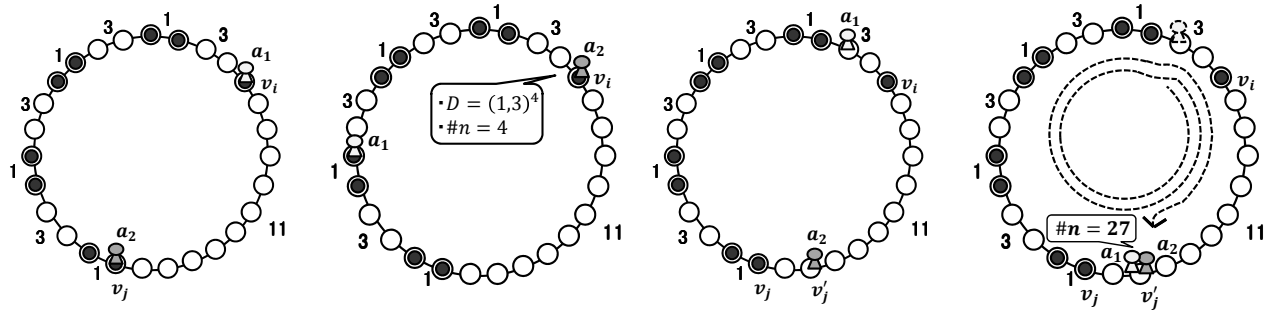
Lemma 3. *If agent a_ℓ guesses the incorrect number of nodes n_ℓ (i.e., $n_\ell \neq n$ holds), $n_\ell \leq n/2$ holds.*

Proof. Let $k_\ell\ (< k)$ be the number of agents (tokens) guessed by a_ℓ. Since a_ℓ observes $4k_\ell$ tokens in the guessing phase, it stores the same distance sequence $(D[0], \ldots, D[k_\ell - 1])$ four times, that is, $(D[0], \ldots, D[4k_\ell - 1]) = (D[0], \ldots, D[k_\ell - 1])^4$ holds. Then, $n_\ell = D[0] + \cdots + D[k_\ell - 1]$ holds. On the other hand since the number of tokens in the ring is $k > k_\ell$, sequence $(D[0], \ldots, D[k_\ell - 1])^4$ is the prefix of $(D[0], \ldots, D[k - 1])^4$. Note that, $n = D[0] + \cdots + D[k - 1]$ holds. Then from Lemma 2, $(D[0], \ldots, D[k_\ell - 1])$ is periodic or $k_\ell \leq k/2$ holds. If $D([0], \ldots, D[k_\ell - 1])$ is periodic, there exists $k'_\ell < k_\ell$ such that $(D[0], \ldots, D[4k'_\ell - 1]) = (D[0], \ldots, D[k'_\ell - 1])^4$ holds. This is a contradiction because a_ℓ should guess n_ℓ as the number of nodes. Hence, $k_\ell \leq k/2$ holds. Then since $(D[0], \ldots, D[k_\ell - 1])$ is the prefix of $(D[0], \ldots, D[k - 1])$, $(D[0], \ldots, D[k - 1]) = (D[0], \ldots, D[k_\ell - 1], D[0], \ldots, D[k_\ell - 1], D[2k_\ell], D[2k_\ell + 1], \ldots)$ holds. Thus, $(D[0] + \cdots + D[k_\ell - 1]) \leq (D[0] + \cdots + D[k - 1])/2$ holds, that is, $n_\ell \leq n/2$ holds. Therefore, we have the lemma. □

Then, we have the following lemmas.

Lemma 4. *If ring R is not periodic, at least one agent guesses the correct number n of nodes and gets distance sequence D of the initial configuration in R.*

Proof. We show that at least one agent guesses the correct number n of nodes. Then from Algorithm 1 and 3, the agent clearly gets the distance sequence D for the initial concifgration in R. We prove the lemma by contradiction, that is, we assume that the number of nodes guessed by each agent is less than n. Without loss of generality, we assume that in the initial configuration agents $a_0, a_1, \ldots, a_{k-1}$ exist in this order. We define



(a) (b) (c) (d)

Fig. 5. An example in the ring having some periodic subsequence ($n = 27, k = 9, d = 3$)

n_i as the number of nodes guessed by a_i and D_i as the distance sequence observed by a_i. In addition, let S_i be the distance sequence such that $D_i = S_i^4$ holds.

Let a_m be the agent that guesses the maximum number of nodes $n_m\,(< n)$ among all agents, and let $\ell = |S_m|\,(< k)$. We assume that the distance sequence a_m observes in Algorithm 1 is D_m=$(d_0^m, \ldots, d_{\ell-1}^m, d_\ell^m, \ldots, d_{2\ell-1}^m, d_{2\ell}^m, \ldots, l_{3\ell-1}^m, l_{3\ell}^m, \ldots d_{4\ell-1}^m) = (d_0^m, \ldots, d_{\ell-1}^m)^4 = S_m^4$. Note that, $S_m = (d_0^m, \ldots, d_{\ell-1}^m)$ is not periodic and $\forall j\ (0 \le j \le \ell - 1)\ d_j^m = d_{j+\ell}^m = d_{j+2\ell}^m = d_{j+3\ell}^m$ holds.

Next, let us consider the agent $a_{m+\ell}$. Then, either $n_{m+\ell} < n_m$ or $n_{m+\ell} = n_m$ holds because n_m is the maximum. We show that $n_{m+\ell} = n_m$ always holds by contradiction, that is, we assume that $n_{m+\ell} < n_m$ holds. Then, $|S_{m+\ell}| < |S_m|$ clearly holds. Consequently, $S_{m+\ell}^3$ is the prefix of S_m^3 because $a_{m+\ell}$ gets the distance sequence $(d_\ell^m, \ldots, d_{2\ell-1}^m) = S_m$ when it observes ℓ tokens. Then from Lemma 2, either $|S_{m+\ell}| \le |S_m|/2$ holds or $S_{m+\ell}$ is periodic. If $|S_{m+\ell}| \le |S_m|/2$ holds, agent a_m observes $S_{m+\ell}^4$ before observing S_m^4 because $(d_0^m, \ldots, d_{2\ell-1}^m) = (d_\ell^m, \ldots, d_{3\ell-1}^m)$ contains $S_{m+\ell}^4$ as its prefix. Consequently, a_m guesses $n_{m+\ell} < n_m$ as the number of nodes, which is a contradiction. If $S_{m+\ell}$ is periodic, $S_{m+\ell} = (S'_{m+\ell})^t$ holds for some distance sequence $S'_{m+\ell}$ and some positive integer t ($S'_{m+\ell}$ is not periodic and $|S'_{m+\ell}| \le |S_{m+\ell}|/2$ holds). Hence, a_m observes $(S'_{m+\ell})^4$ before observing S_m^4 and the number of nodes a_m guesses is less than n_m, which is also a contradiction. Therefore, $n_{m+\ell} = n_m$ holds.

Let $m(i) = m + i\ell$ and $A_m = \{a_{m(i)} | \, i \ge 0\}$. As mentioned above, $n_m = n_{m+\ell}$ and $S_{m(0)} = S_{m(1)} = S_m$ hold. In addition, $a_{m(1)}$ observes the same distance sequence of length $4|S_m|$ as $a_{m(0)}$. Hence recursively, $a_{m(i+1)}$ observes the same distance sequence of length $4|S_m|$ as $a_{m(i)}$ and consequently each agent in A_m observes S_m as the first ℓ consecutive distances. When k is divided by ℓ, since every agent $a_{m(i)}$ observes S_m as the first ℓ consecutive distances and $\ell < k$ holds, the ring is periodic, which is a contradiction. In the following, we consider the case that k is not divided by ℓ and show that $S_{m(0)} (= S_m)$ is periodic in this case. When k is not divided by ℓ, $k = \alpha\ell + \beta\,(0 < \beta < \ell)$ holds for some integers α and β. Let $m(\alpha) = m + \alpha\ell$. Then, the prefix of $S_{m(0)}$ is identical to the suffix of $S_{m(\alpha)}$ because the trajectories of $a_{m(0)}$ and $a_{m(\alpha)}$ include the same part of the ring. We assume that t elements are overlapped, that is, $(d_0^{m(0)}, \ldots, d_{t-1}^{m(0)}) = (d_{\ell-t}^{m(\alpha)}, \ldots, d_{\ell-1}^{m(\alpha)})$ holds. In addition since $S_{m(0)} = S_{m(\alpha)}$ holds, $(d_0^{m(0)}, \ldots, d_{\ell-1}^{m(0)}) = (d_0^{m(\alpha)}, \ldots, d_{\ell-1}^{m(\alpha)})$ holds. If $t > \ell/2$ holds, $(d_t^{m(0)}, \ldots, d_{\ell-1}^{m(0)}) = (d_0^{m(\alpha)}, \ldots, d_{\ell-t-1}^{m(\alpha)})$ holds. Hence, $shift(S_{m(0)}, t) = S_{m(\alpha)} = S_{m(0)}$ holds. If $t \le \ell/2$ holds, $(d_{\ell-t}^{m(0)}, \ldots, d_{\ell-1}^{m(0)}) = (d_{\ell-t}^{m(\alpha)}, \ldots, d_{\ell-1}^{m(\alpha)}) = (d_0^{m(0)}, \ldots, d_{t-1}^{m(0)}) = (d_0^{m(\alpha)}, \ldots, d_{t-1}^{m(0)})$ and $(d_t^{m(0)}, \ldots, d_{\ell-t-1}^{m(0)}) = (d_t^{m(\alpha)}, \ldots, d_{\ell-t-1}^{m(\alpha)})$ hold. Thus, $(d_t^{m(0)}, \ldots, d_{\ell-1}^{m(0)}) = (d_0^{m(\alpha)}, \ldots, d_{\ell-t-1}^{m(\alpha)})$ holds. Consequently, $shift(S_{m(0)}, t) = S_{m(\alpha)} = S_{m(0)}$ holds. Therefore, $S_{m(0)}$ is periodic since $0 < t < \ell$ holds. However, this contradicts the assumption that $S_{m(0)} (= S_m)$ is not periodic.

Therefore, we have the lemma. □

Lemma 5. *If ring R is not periodic, every agent eventually gets the correct number n of nodes and distance sequence D of the initial configuration in R.*

Proof. We show that all agents eventually get the correct number n of nodes. Then from Algorithms 1 to 3, all agents can clearly get distance sequence D of the initial configuration in R. We prove the lemma by contradiction, that is, we assume that when all agents are in the suspended states, there exists at least one agent a_h whose guessed number of nodes n' is less than n. Then from Lemma 3, $n' \le n/2$ holds. On the other hand from Lemma 4, at least one agent a_c guesses the correct number n of nodes. In the following we show that a_c observes a_h during the patrolling phase and sends its guessed number of nodes n to a_h, which contradicts the assumption of $n' < n$.

At first, let us consider the number of nodes a_h visits. Let n_1 be the number of nodes a_h guesses in the guessing phase. From Algorithms 1 to 3, a_h moves at most $14n_1$ times by the time a_h enters a suspended state for the first time. After this, we assume that a_h receives messages and updates its guessed number of nodes to $n_2, n_3, \ldots, n_l = n'$ in this order. When a_h updates it guessed number of node to n_2, a_h's total moves at that point (i.e, *nodes*) is at most $7n_2$ since $n_1 \leq n_2/2$ holds. Hence, $12n_2 - nodes$ is clearly positive. Then, a_h firstly moves in the ring until its total moves becomes $12n_2$ by moving $12n_2 - nodes$ times. After this, a_h moves to a new target node and enters a suspended state again. This requires at most $14n_2$ total moves. Then since $n_3 \leq n_2/2$ holds from Algorithm 3, *nodes* is at most $7n_3$ and $12n_3 - nodes$ is clearly positive. Thus recursively, we can show that $12n_i - nodes$ is always positive ($2 \leq i \leq l$) and a_h's total moves unless it does not get the correct number n of nodes is at most $14n' \leq 7n$. On the other hand, agent a_c moves $8n$ times in the patrolling phase. Thus, a_c clearly observes a_h during the patrolling phase and sends its guessed number n of nodes to a_h, which is a contradiction.

Therefore, we have the lemma. □

Finally, we have the following lemma.

Lemma 6. *When ring R is not periodic, agents solve the uniform deployment problem without termination detection.*

Proof. From Lemma 5, all agents eventually get the corect number n of nodes and distance sequence D for the initital configuration in R. Then, each agent can compute its correct target node from D and move there. Thus, we have the lemma. □

4.1.2 Case for periodic rings Next, let us consider the case that the ring is periodic. Let R' be a periodic ring and D' be the distance sequence of the initial configuration in R'. We say R' is a (N, ℓ)-node ring if there exists a non-periodic distance sequence D such that $D' = D^\ell$ holds and the total sum of elements of D is N. Then, $n = N\ell$ holds and ℓ is equivalent to the symmetry of the initial configuration in R'. We call the ring R with the distance sequence D the *fundamental ring of* R' (e.g. Fig. 6). Note that a non-periodic ring can be denoted by a $(n, 1)$-node ring. In addition for each agent a_i in R, there exist ℓ agents in R' such that the distance sequence of each agent is ℓ-times repetition of the distance sequence of a_i. We say such agents in R' are *corresponding agents* of agent a_i in R and denote by a_i^j ($0 \leq j \leq \ell - 1$). We assume that agents $a_i^0, a_i^1, \ldots, a_i^{\ell-1}$ are deployed in this order and operations to an above index of a_i^j assume calculation under modulo ℓ. Then, the distance from a_i^j to a_i^{j+1} is N. In this case, all agents eventually guess the incorrect number $N = n/\ell$ of nodes, but we can show that agents can solve the uniform deployment problem similarly to in R. Concretely from Algorithms in Section 4.1, each agent moves to its target node after considering, based on the guessed number N of nodes, it traveled twelve times around the ring. This means that each agent stays at its target node during its twelfth or thirteenth circulations in the ring of the guessed size N, which guarantees that when all agents are in the suspended states, no agents stay at the same node and they can achieve the uniform deployment. For example, let us consider rings in Fig. 6. Ring R' is the (6,2)-node periodic ring and R is the fundamental ring of R'. In R, each agent guesses the correct number 6 of nodes in the guessing phase and moves to its correct target node (Fig. 6 (a)). On the other hand in R', each agent also guesses the number 6 of nodes, which is incorrect (Fig. 6 (b)). By Algorithms 1 to 3, each agent moves to its target node after considering, based on the guessed size 6, it travelled twelve times around the ring, that is, after each agent moves 72 times (actually, each agent travelled six times around ring R'). This guarantees that when all agents are in the suspended states, no agents stay at the same node and they can achieve the uniform deployment (Fig. 6 (c)).

Now, we have the following lemmas, which can be proved similarly to the case of non-periodic rings.

Lemma 7. *Let R' be a (N, ℓ)-node periodic ring and R be the fundamental ring of R'. Let a_i in R be the agent guessing the number N of nodes in the guessing phase. Then in R', agent a_i^j ($0 \leq j \leq \ell - 1$) corresponding to a_i also guesses the number N of nodes.*

Proof. From the definition of R', a_i^j observes the same distance sequence as that of a_i. In addition since agents have no knowledge of k or n, agents determine their guessed number of nodes depending only on the distance sequence they observe. Thus, a_i^j guesses the same number of nodes as that of a_i. □

Lemma 8. *Let R' be a (N, ℓ)-node periodic ring and R be the fundamental ring of R'. Then in R', every agent eventually gets the number N of nodes and distance sequence D of the initial configuration in R.*

Proof. We show that all agents eventually get the number n of nodes. Then from Algorithms 1 to 3, all agents can clearly get distance sequence D of the initial configuration in R. We prove the lemma by contradiction, that is, we assume that when all agents are in the suspended states, there exists at least one agent a_h whose guessed number of nodes n' is less than N. On the other hand from Lemma 7, there exists agent a_c^j ($0 \leq j \leq \ell - 1$)

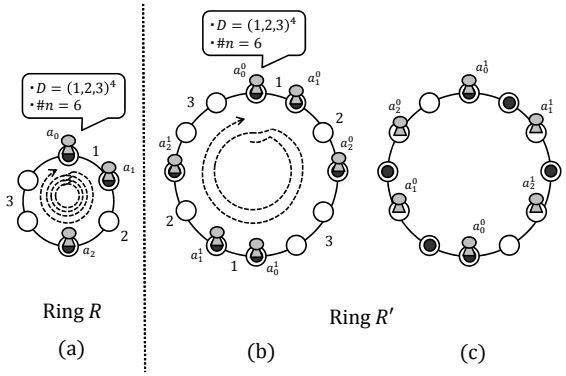


Fig. 6. An example for the periodic ring

guessing the number N of nodes in the guessing phase. Let $A_c = \{a_c^0, a_c^1 \ldots, a_c^{\ell-1}\}$. In the following, we show that some agent in A_c observes a_h during the patrolling phase and sends its guessed number N of nodes to a_h, which contradicts the assumption of $n' < N$.

At first, let us consider the number of nodes a_h visits. Similarly to the case for non-periodic rings, when a_h updates its guessed number of nodes from n'' to n', it firstly moves in the ring until its total moves becomes $12n'$ by moving $12n' - \mathit{nodes}$ times. After this, a_h moves to a new target node and enters a suspended state again. This requires at most $14n'$ total moves. Hence unless a_h does not get the number n of nodes, its total moves is at most $14n' \leq 7N$.

On the other hand from Lemma 7, there exists agent a_c^j in A_c such that it guesses N as the number of nodes and the distance from $v_{HOME}(a_c^j)$ to $v_{HOME}(a_h)$ is less than N. Remind that, $v_{HOME}(a)$ is the home node of agent a. Then, let us consider the behavior of agent a_c^{j-4}. Agent a_c^{j-4} firstly moves $4N$ times and finishes the guessing phase at node $v_{HOME}(a_c^j)$. After this, a_c^{j-4} moves $8N$ times from $v_{HOME}(a_c^j)$ in the patrolling phase. On the other hand, a_h moves at most $7N$ times from $v_{HOME}(a_h)$. Since the distance from $v_{HOME}(a_c^j)$ to $v_{HOME}(a_h)$ is less than N, a_c^{j-4} observes a_h during the patrolling phase and sends the number N of nodes to a_h, which is a contradiction.

Therefore, we have the lemma. □

Lemma 9. *Even when ring R' is periodic, agents solve the uniform deployment problem without termination detection.*

Proof. From Lemma 8, all agents eventually get the number N of nodes and distance sequence D of the initial configuration in R, where R is the fundamental ring of R'. From Algorithm 3 when agent a_i^j gets the number N of nodes it firstly moves in the ring until its total moves becomes $12N$. Then, a_i^j is at $v_{HOME}(a_i^{j+12})$. After this, a_i^j computes its target node from D and moves there, which requries at most $2N$ moves. Hence, a_i^j eventually stays between $v_{HOME}(a_i^{j+12})$ and $v_{HOME}(a_i^{j+14})$. This mean that letting v_{base} (resp., v'_{base}) be the base node exsiting between $v_{HOME}(a_i^{j+12})$ and $v_{HOME}(a_i^{j+13})$ (resp., $v_{HOME}(a_i^{j+13})$ and $v_{HOME}(a_i^{j+14})$) a_i^j eventually stays between v_{base} and v'_{base}. Moerover, it crarly holds total moves of each of a_i^j $(0 \leq j \leq \ell - 1)$ are the same. Thus when all agents are in the supended states, no agents stay at the same node and agents can achive the uniform deployment.

Therefore, we have the lemma. □

Finally, we have the following theorem for (N, ℓ)-node rings.

Theorem 4. *For agents with no knowledge of k or n, the proposed algorithm solves the uniform deployment problem without termination detection. This algorithm requires $O(k/\ell \log(n/\ell))$ memory per agent, $O(n/\ell)$ time, and $O(kn/\ell)$ total moves.*

Proof. From Lemmas 6 and 9, agents solve the uniform deployment problem. In the following, we analyze complexity measures.

At first, we evaluate the memory requirement per agent. Each agent eventually gets the distance sequence $D = (d_0, d_1, \ldots, d_{(4k/\ell)-1})$. Since each d_i is at most n/ℓ, this sequence requires $O(k/\ell \log(n/\ell))$ memory. Moreover, the other variables require $O(\log(n/\ell))$ bit memory. Therefore, the memory requirement per agent is $O(k/\ell \log(n/\ell))$.

Next, we analyze the time complexity. Let $A_{correct}$ be the set of agents that guess the number $n/\ell\,(= N)$ of nodes in the guessing phase. Each agent $a_c \in A_{correct}$ moves its correct target node without stopping on

the way, which requires at most $14n/\ell$ unit times. In addition from the proof of Lemmas 5 and 8, each agent $a_h \notin A_{correct}$ gets the number n/ℓ of nodes within $12n/\ell$ unit times. After this, a_h requires at most $14n/\ell$ unit times to moves to its correct target node. Thus, the time complexity is $O(n/\ell)$.

At last, we analyze the total number of agent moves. Each agentrequires at most $14n/\ell$ moves to move to its target node. Thus, the total number of agent moves is $O(kn/\ell)$. □

5 Conclusion

In this paper, we considered the uniform deployment problem of mobile agents in asynchronous unidirectional ring networks. At first we showed that, when termination detection is required, there exists no algorithm to solve the uniform deployment problem. Next, we proposed an algorithm to solve the uniform deployment problem without termination detection in non-periodic rings. This algorithm requires $O(k/\ell \log(n/\ell))$ memory per agent, $O(n/\ell)$ time, and $O(kn/\ell)$ total moves. As a future work, we want to consider the lower bound of memory requirement per agent. We conjecture that it is $\Omega(\log n)$, and if the conjecture is correct, we can show that the first algorithm is asymptotically optimal in terms of memory requirement.

References

1. D. Kotz S. R. Gray, G. Cybenko, A.R. Peterson, and D. Rus. D'agents: Applications and performance of a mobile-agent system. *Software: Practice and Experience*, 32(6):543–573, 2002.
2. D.B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.
3. T. Suzuki, T. Izumi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Move-optimal gossiping among mobile agents, Theoretical Computer Science. 393(1):90–101, 2008.
4. E. Kranakis, D. Krozanc, and E. Markou. The mobile agent rendezvous problem in the ring, Synthesis Lectures on Distributed Computing Theory, Vol. 1. pages 1–122, 2010.
5. P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk. Multiple mobile agent rendezvous in a ring, LATIN, LNCS, Vol. 2976. pages 599–608, 2004.
6. E. Kranakis, D. Krizanc, and E. Markou. Mobile agent rendezvous in a synchronous torus, LATIN LNCS, Vol. 3887. pages 653–664, 2006.
7. P. Fraigniaud and A. Pelc. Deterministic rendezvous in trees with little memory, disc, LNCS, Vol. 6950. pages 242–256, 2008.
8. A.Kosowski J. Czyzowicz and A. Pelc. How to meet when you forget: Log-space rendezvous in arbitrary graphs. *DISC*, 25(2):165–178, 2012.
9. A. Labourel J. Chalopin, S. Das and E. Markou. Tight bounds for scattered black hole search in a ring. In *SIROCCO*, pages 186–197. Springer, 2011.
10. P. Flocchini, G. Prencipe, and N. Santoro. Self-deployment of mobile sensors on a ring. *Theoretical Computer Science*, 402(1):67–80, 2008.
11. E. Yotam and B. M. Alfred. Uniform multi-agent deployment on a ring. *Theoretical Computer Science*, 412(8):783–795, 2011.
12. E. Mesa-Barrameda L. Barriere, P. Flocchini and N. Santoro. Uniform scattering of autonomous mobile robots in a grid. *International Journal of Foundations of Computer Science*, 22(03):679–697, 2011.
13. T. Mega, F.Ooshita, H. Kakugawa, and T. Masuzawa. Algorithms for uniform deployment of mobile agents on synchronous rings. *COMP*, 112(24):9–16, 2012.
14. S. Kawai, F. Ooshita, H. Kakugawa, and T. Masuzawa. Randomized rendezvous of mobile agents in anonymous unidirectional ring networks, SIROCCO LNCS, Vol. 7355. pages 303–314, 2012.

Appendix

A The uniform deployment for the case of $n \neq ck$

To remove the restriction of $n = ck$ imposed in Section 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 $\lceil n/k \rceil$ or $\lfloor n/k \rfloor$.

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 nodes as the base nodes, the common target nodes can be determined using the base node as a reference node: Let b be the number of the base nodes, and $r = n \bmod k$. The distance of every pair of adjacent base nodes is identical even in the case of $n \neq ck$, and is $n/b = (\lfloor n/k \rfloor \times k + r)/b = \lfloor n/k \rfloor \times k/b + r/b$ (notice that k/b and r/b are integers). This implies that we should select $k/b - 1$ 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 achieve the uniform deployment.

匿名単方向リングネットワークにおけるモバイルエージェント集合問題に対するトークンを用いた乱択アルゴリズム

難波 瑛次郎[1] 大下 福仁[2] 角川 裕次[1] 増澤 利光[1]

[1] 大阪大学 大学院情報科学研究科

[2] 奈良先端科学技術大学院大学

概要　本稿では，匿名単方向リングネットワークにおいて，トークンを用いてモバイルエージェント集合問題を解くアルゴリズムについて考察する．集合問題では，初期状況において任意のノードに分散している複数のエージェントが有限時間内に同一のノードに集合することを目的とする．本稿では，匿名単方向リングネットワークにおいて，エージェントがノード数やエージェント数を知らないモデルを扱う．本モデルに対する既存手法では，各ノードに白板と呼ばれる十分な量のメモリを仮定することで，集合問題を高確率で解いている．それに対して，本稿では，各ノードに 1 ビットのトークンを置けるという仮定のもとで，集合問題に対する乱択アルゴリズムを提案する．シミュレーションにより，提案アルゴリズムはランダムな初期配置から高確率で集合問題を解くことを示す．また，決定性 アルゴリズムで集合問題を解けない周期的な初期配置に対しても，提案アルゴリズムにより確率的に集合問題が解けることを示す．

1　はじめに

分散システムとは，多数の計算機（以下，**ノード**）とそれらを繋ぐ通信リンク（以下，**リンク**）から構成されたシステムである．近年，分散システムは大規模化が進んでおり，分散システムの設計はますます困難になっている．そのため，大規模化・複雑化する分散システムを設計するための有効な手法として，現在，モバイルエージェント (以下，**エージェント**) を利用した設計法が注目を集めている．エージェントとは，ネットワーク中のノード間を移動しながら，自律的に動作するソフトウェアである．

エージェントシステムを用いた代表的な問題に集合問題がある [1]．集合問題とは，初期状況においてネットワーク中に分散している複数のエージェントが，有限時間内にひとつのノードで集合することを目的とする．ひとつのノードに集合することで，各エージェントが保持している局所情報を共有したり，あるいは，エージェントの動作を同期させたりすることができる．ネットワーク中の各ノードに一意な ID が存在するとき，集合問題は容易に解くことができる．例えば，特定の ID を持つノードで集合するようにアルゴリズムを設計すればよい．しかし，ネットワークによってはセキュリティ上の問題から，ノードの ID を利用できない場合がある．よって，各ノードに ID がない匿名ネットワークにおいて，集合問題を解くことも重要である．しかし，匿名ネットワークにおいて決定性アルゴリズムで集合問題を解くことは不可能である [2]．そのため，ノードに情報を残せる [3, 4, 5]，乱数を利用できる [6, 7, 8]，ネットワークの辺ラベリングを制限する [5, 9]，ネットワークのトポロジを制限する [10] などの仮定を加えることで集合問題を解くアルゴリズムが提案されている．本稿では，匿名単方向リングネットワークにおいて，ノードに情報を残し，乱数を利用することができる場合の集合問題について考察する．

匿名単方向リングネットワークにおける集合問題に対する乱択アルゴリズムとして，文献 [2] のアルゴリズムが挙げられる．このアルゴリズムは，リングネットワーク上の各ノードに，**白板**と呼ばれる読み書き可能なメモリが存在することを仮定している．その白板上に乱数を用いて生成したラベルを書き込むことで，ノードに一意な ID が存在するネットワークを高確率で作成し，その ID を用いて集合を実現している．そのため，各ノードはラベルを保存するために十分な量の白板を管理する必要がある．

本稿では，匿名単方向リングネットワークにおいて，**トークン**を用いて集合問題を解く乱択アルゴリズムを提案する．トークンとは，ノードに置くことができる 1 ビットの目印である．各ノードは 1 ビットのメモリでトークンの情報を管理することができ，白板を用いる場合に比べて，各ノードが管理するメモリの量を削減することができる．本稿では，シミュレーションにより提案アルゴリズムが，ランダムな初期配置から高確率で集合問題を解くことができることを示す．また，決定性アルゴリズムでは集合不可能である周期的な初期配置からも確率的に集合問題を解くことができることを示す．

本稿の構成は以下の通りである．2 章では，本稿で用いるモデルと問題の定義を行う．3 章では，本モデルにおいて，集合問題を解くアルゴリズムを提案し 4 章でその評価を行う．5 章では本稿の結果をまとめる．

2　諸定義

2.1　ネットワークモデル

単方向リングネットワーク R は，2 項組 $R = (V, L)$ で定義される．ここで V はノードの集合であり，L は通信リンクの集合である．ネットワークのノード数を $n = |V|$ と定義する．また，本稿ではリングネットワークを次のように定義する．

- $V = \{v_0, v_1, ..., v_{n-1}\}$
- $L = \{\{v_i, v_{(i+1) \bmod n}\} | 0 \leq i \leq n-1\}$

リング R において，ノード $v_0, v_1, ..., v_{n-1}$ は時計回りにこの順で並んでいるとする．以下，簡単のため，ノードのインデックスにおける計算は n の法のもとに計算されるものとして，$v_{i \bmod n}$ を単に v_i と表す．さらに，v_i と v_{i+1} はリンクで結ばれているため，互いに**隣接ノード**であるという．また，v_i にとって，v_{i+1} は**前方ノード**であるといい，それに対して，v_{i+1} にとって，v_i は**後方ノード**であるという．また，v_i から v_{i+1} への方向を**前方向**，v_{i+1} から v_i への方向を**後方向**であるという．

ネットワークには，ノードに一意な ID が存在する**ラベル付きネットワーク**と，一意な ID が存在しない**匿名ネットワーク**が存在する．本稿では，匿名ネットワークの上で議論を進めていく．また，ネットワーク中の各ノード v_i にはトークンと呼ばれる 1 ビットの目印を置くことができる．以下，v_i 上に存在するトークンを $t_i = \{true, false\}$ で表す．$t_i = true$ のとき，ノード v_i にトークンが存在することを表し，$t_i = false$ のとき，ノード v_i にトークンが存在しないことを表す．

2.2 エージェントモデル

ネットワーク中のエージェントの集合を $A = \{a_0, a_1, ..., a_{k-1}\}$ とする．エージェントは有限ムーア型状態機械 $(S, \delta, s_{initial})$ で定義される．状態集合 S はエージェントの状態の集合であり，エージェントの状態はエージェントが持つ全変数の割り当てにより定義される．状態集合 S の中には，**初期状態** $s_{initial}$ と**終了状態** s_{final} という特別な状態が含まれる．本稿では，全てのエージェントは同じ状態機械であると仮定し，全てのエージェントは同じ初期状態 $s_{initial}$ から実行を開始する．また，エージェントはトークンを各々1 つずつ有しており，ノード v_i にトークンが置かれていなければ 1 つ置くことができる．

状態遷移関数 δ は $\delta : S \times T \times RN \to S \times T \times M$ であり，エージェントの状態，ノード上のトークンの状態，乱数値から，次のエージェントの状態，次のトークンの状態，次の移動を決める関数である．$T = \{true, false\}$ はエージェントが滞在するノード上のトークンの状態を表し，$true$ はトークンが存在すること，$false$ はトークンが存在しないことを表す．RN は乱数の取りうる値の集合を表す．また，$M = \{move, stay\}$ はエージェントの移動を表す．$move$ はエージェントが移動することを表し，$stay$ はエージェントが待機することを表す．このとき，エージェントの移動は瞬間的であるする．つまり，エージェントはリンク上に存在することはなく，常にノードに存在する．さらに，単方向リングネットワークと仮定しているため，v_i に存在するエージェントは v_{i+1} にしか移動することができない．本稿では，各エージェントが同一の状態機械の場合を想定するため，全てのエージェントは同一の状態遷移関数に従って動作する．

2.3 システムの状況

エージェントシステムの**状況** c は，システム内にいる各エージェント a_i の状態 $(s_i \in S)$ と各ノード v_j の状態 $(t_j \in T)$，各エージェントの位置 $(l_i \in L)$ で表す．つまり，各エージェントの全変数の割り当てと各ノードのトークンの状態，エージェントの位置によって状況が決定される．ここで，エージェントの位置は整数の列 $(l_0, l_1, ..., l_{k-1})$ で表す．$L = \{0, 1, ..., n-1\}$ としたとき，$l_i (\in L)$ はエージェント $a_i (\in A)$ がいるノードの位置 (インデックス) を表す．C をエージェントシステムにおいて，取り得る全状況の集合とする．**初期状況** $c_0 (\in C)$ では，各エージェントは同一の状態機械であることを想定しているため，初期状態は同じ $s_{initial}$ であり，各ノードのメモリにはトークンはない状態 $t_i = false$ である．よって初期状況はエージェントの初期位置にのみ依存する．ただし初期状況において，複数のエージェントが同一のノードに存在することはないものとする．

A_i を空ではない任意のエージェントの集合とする．状況 c_i において，A_i に属する全ての実行可能なエージェントが状態遷移関数に従い動作を行うことを $c_i \xrightarrow{A_i} c_{i+1}$ と表記し，**ステップ**と呼ぶ．このとき，A_i に同一のノードにいる複数のエージェントが選択されたとき，それらのエージェントの動作する順序は任意とする．任意の i について，$A_i = A$ のとき，全てのエージェントが同時に動作を行い，このモデルを**同期**モデルと呼ぶ．それに対して，この条件が成り立たない場合，このモデルを**非同期**モデルと呼ぶ．初期状況から i ステップ後の状況を c_i と表す．**実行** E は状況の列であり，$E = c_0, c_1, ...$ と表す．全てのエージェントが終了状態 s_{final} へ遷移した状況を**終了状況** c_{final} と呼ぶ．終了状況以降どのエージェントも動作を行わない．なお，本稿で提案するアルゴリズムは同期モデルを仮定している．

2.4 集合問題

集合問題は，初期状況において任意のノードに分散している複数のエージェントが有限時間内に同一のノードに集合することを目的とする．

休止状態 $(rest)$ をトークンの内容が変わらない限り動作しない終了状態とする．

定義 1 実行 E が以下の条件を満たすとき，E は集合問題を解くという．

- E は有限長である．
- 終了状況において，全てのエージェントが同一のノードに存在し，全てのエージェントの状態が**休止状態**である．

3　アルゴリズム

まず初めに，アルゴリズムの方針を簡潔に述べる．各エージェントは初期状況における初期ノードにそれぞれが有するトークンを置き，同時にアルゴリズムの実行を開始する．エージェントは移動しながら，トークン間の間隔，すなわち，トークンを有するノード間のリンク数を記憶していく．次に，トークン間隔の列が周期的になったとき，すなわち，同じ間隔の列が 2 回繰り返されたときリングネットワーク上を 2 周したものとみなし，リング上の 1 ノードを集合ノードとして選んで集合しようとする．この時，常にエージェントは暫定のノード数 n' を計算している．誤って休止状態となった場合，後方からきたエージェントにより n' を比較した後，n' の大きい方のエージェントと以後同様に振る舞う．

次に, 提案アルゴリズムの変数を Algorithm 1、擬似コードを Algorithm 2 に示す．エージェントはトークンが置かれているノード v_i と，その前方向にはじめて存在するトークンが置かれているノード $v_{i+\Delta_{i_0}}$ との間のリンクの数 Δ_i を記憶する．その後，同様にして Δ のリスト（以下，パターン）$P_i = (\Delta_{i_0}, \Delta_{i_1}, ..., \Delta_{i_j})$ を作成する．作成されたパターンに 2 度同じ繰り返しを発見した時，つまり $\{\Delta_{i_0}, ..., \Delta_{i_x}\} = \{\Delta_{i_{x+1}}, ..., \Delta_{i_{2x+1}}\}$ なった時，パターンを二等分する．そしてそのパターン $P = \{\Delta_{i_0} ..., \Delta_{i_x}\}$ を**辞書式最小順**に並べ替る．リスト $\{x_y, x_{y+1}, ..., x_n, x_0, x_1, ..., x_{y-1}\}$ を辞書式最小順 $\{x_0, x_1, ..., x_n\}$ に並べ替え，その差の合計 $x_y + x_{y+1} + ... + x_n$ から目的ノードまでの距離を算出する．目的ノードへ到着した時，エージェントは休止状態へと移行する．また，同じノードにエージェントがいる場合はエージェント間で通信を行い，n' が大きい方のエージェントと以降同様に振る舞う．ここで暫定ノード数 n' には常にパターン P の要素の合計を 2 で割った値が格納されている．各エージェントは同一のアルゴリズムにしたがって処理を行うため，通信した際にエージェントの変数を全て変更するだけで，それ以降変数を変更されたエージェントは変更したエージェントと同様の動作を行う．

Algorithm 1 変数

Variables in Agent a_i

int $state = 0$ //エージェントの状態変数

ArrayList<Integer> $pattern$ //パターンを記憶するリスト

int $goal = 0$ //集合地点までの距離

int $n' = 0$ // エージェントが推測するネットワークのノード数

int $round = 0$ //同期用変数

int count=0 //リンク数のカウンタ

Variable in Node v_j

boolean $node = false$ //ノード上のトークンの有無を表す

Algorithm 2 決定性アルゴリズム

Main Routine

状態 0：$state == 0$
  $node \leftarrow true$
  $count++$
  $n'++$
  $state \leftarrow 1$

状態 1：$state == 1$
  if $node$ **then**
    $pattern.add(count)$
    $count \leftarrow 1$
    if $pattern$ に繰り返しを発見 **then**
      $state \leftarrow 2$
      目的ノード$\leftarrow DictionaryMinimum(pattern)$//
      目的ノードまでの移動数を算出
    end if
  end if

状態 2：$state == 2$
  if 目的ノードへ到着 **then**
    $state \leftarrow 3$
    次のノードへ
  end if

状態 3：$state == 3$
  休止状態

default：
  $round++$
  if $state$ が 2 のエージェントが同じノードに存在する **then**
    各変数を共有
  end if

しかしこの決定性アルゴリズムでは，集合が不可能である場合が存在する．それは，リングネットワーク全体に周期性が存在する場合である．リングネットワークにおいて全体として周期性が存在することは，リング R の全てのトークンの間の間隔を記憶したリスト I が 2 回以上の同一のリスト I' の連続で構成されていることを表す．つまり $P = (I' + ... + I')$ と表される．このような場合でも確率的に集合可能にするため，下述の乱択アルゴリズムを上記の決定性アルゴリズムに追加する．

Algorithm 3 に上記の決定性アルゴリズムの状態 3 以降の部分を変更した乱択アルゴリズムを示す．上記の決定性アルゴリズムで，各エージェントが目的のノードまで集合するまでに，最初にエージェントが目的ノードへ到着し休止状態となった時点から高々n' ステップの内に目的ノードへと集合する．そのため，n' ステップの間最初に目的ノードへと到着したエージェントは休止状態のまま待機し，$n'+1$ ステップ目で休止状態を解き，同ノードへと集合しているエージェントを起動する．その後，以下の 2 処理を 50% の確率でどちらか一方のみ行う．

- 暫定のリング R' を 1 周，つまり n' 回ノードを移動し確認を行う．暫定のエージェント数は n' の場合と同様にパターンのリストから推測でき，トークンの数がエージェント数であるため，リスト $pattern$ の大きさとなる．
- その場で休止状態となる

そのため．リング R 全体に周期性が存在し，その周期が $\frac{n}{2}$ であるようなリング R の場合，1 回の確認作業で 50% の確率正しく集合することが可能となる．

Algorithm 3 乱択アルゴリズム

Main Routine
状態 3：$state == 3$
  n' ラウンド待機
  if n' の待機が終了 **then**
    if 50% の確率 **then**
      $state \leftarrow 4$
      $goal \leftarrow n'$
      同一ノード存在する休止状態のエージェント群を起動
      起動したエージェント群に全ての変数を共有
    else
      $state \leftarrow 5$
      同一ノードに存在するエージェント群に共有
    end if
  end if
状態 4：$state == 4$
  if $goal > 0$ **then**
    次のノードへ
  else
    $state \leftarrow 5$
  end if
状態 5：$state == 5$
  休止状態
default：
  $round++$
  n' を推測
  if ($state == 2$ または $state == 4$ であり)，かつ (n' が自分より小さい) エージェントが同じノードに存在 **then**
    各変数を共有
  end if

4 アルゴリズムの評価

4.1 シミュレーション環境

3 章で提案したアルゴリズムのランダムな初期配置におけるエージェント集合確率を評価するためにシミュレーションを行った．シミュレーション環境は表 1 の通りである．

開発環境	:	eclipse Luna 4.4.1
仕様言語	:	java
JDK Version	:	JDK 1.7
試行回数	:	10000 回
ノード数上限	:	500
エージェント数	:	$2 \leq k \leq n$

表 1: シミュレーション環境

4.2 シミュレーション結果と考察

提案するアルゴリズムの集合成功率は表 2 のとおりである．この結果から，乱択アルゴリズムの追加により決定性アルゴリズムに悪影響を与えずに，集合可能な初期配置を増加させることが可能であるといえる．下記のシミュレーション結果において，エージェントの初期配置はランダムとなっている．表 2 に、エージェント数ノード数の両方をランダムに設定し、初期配置もランダムに配置したシミュレーション結果を示す．

アルゴリズム	失敗回数	成功確率
決定性アルゴリズムのみ	99 回	99.01%
乱択アルゴリズム追加後	91 回	99.09%

表 2: シミュレーション結果:$n \leq 500, 2 \leq k < n$ のとき (10000 回)

また，入力としてランダムではなく，全体として周期性の出来やすいノード数 4 エージェント数 2 の場合を 5000 回実行した結果を表 3 1 に掲載する．

アルゴリズム	失敗回数	成功確率
決定性アルゴリズムのみ	1580 回	68.4%
乱択アルゴリズム追加後	795 回	84.1%

表 3: シミュレーション結果:$n = 4, k = 2$ のとき (5000 回)

これらの結果から以下のことがいえる．

- 追加した乱択アルゴリズムは，元の決定性アルゴリズムの動作に悪影響を与えない．決定性アルゴリズムのみのとき，短いリングと勘違いしたエージェントは止まっているだけであるため，正しいリング長を認識したエージェントがいれば必ず追いつく．しかし，乱択部分を追加すると，勘違いしたエージェントが再移動するため，最終的に追いつかれることなく集合に失敗する可能性がある．しかし，このようなことがあまり起こらないことを確認した．
- 全体として周期的な初期位置の場合，確率的に集合確率を向上させることができる．ノード数 4 エージェント数 2 のとき，約 $\frac{1}{3}$ の確率で全体として周期的である配置となる．乱択アルゴリズムを追加して実行した場合，その失敗回数が約半分となっている．そのため，周期的な初期配置の場合でも，確率的に集合問題を解く事ができるといえる．

またこの結果から，乱択部分を複数回実行するように設計すれば，さらに集合確率を上げられることが考えられる．具体的には，$state = 3$ となり n' 回移動して確認処理を行った後，移動した先のノードに暫定エージェント数 k' 以上のエージェントが存在した場合は，さらにアルゴリズムを再実行する．次回以降は，パターンの繰り返しを判定する部分において，前回のパターン長よりも長い場合のみ判定するようにすれば良い．また，移動した先のエージェント数が想定していたエージェント数である状態が複数回続いた場合は，休止状態となれば良い．

5 おわりに

本稿では，匿名単方向リングネットワークにおいて，エージェントがノード数やエージェント数を知らないとき，トークンを用いて集合問題を解く乱択アルゴリズムを提案した．シミュレーションにより，ランダムな初期配置に対して，集合が実現できることを示した．また決定性アルゴリズムでは集合不可能である周期的な初期配置からの集合も確率的に実現できることを示し，アルゴリズムの有効性を確認した．

今後の課題として，乱択アルゴリズムで行っているランダムな移動を繰り返し適用することにより，集合成功確率や，移動回数を理論的に評価するアルゴリズムの提案が挙げられる．また，エージェントをノードに配置する際に周期的な初期配置とならないように配置する乱択アルゴリズムの検討も今後の課題である．

参考文献

[1] E.Kranakis, D.Krizanc, and E.Markou. *The Mobile Agent Rendezvous Problem in the Ring*. Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool Publishers, 2010.

[2] T. Masuzawa, H. Kakugawa, F. Ooshita, and S. Kawai. Randomized gathering of mobile agents in anonymous unidirectional ring networks. Vol. 25, No. 5, pp. 1289–1296, May 2014.

[3] J. Chalopin, S. Das, and P. Widmayer. Rendezvous of mobile agents in directed graphs. In *DISC 2010 – 24th International Conference on Distributed Computing*, Vol. 6343 of *Lecture Notes in Computer Science*, pp. 282–296. Springer, 2010.

[4] S. Das, R. Srámek M. Mihalák, E. Vicari, and P. Widmayer. Rendezvous of mobile agents when tokens fail anytime. Vol. 5401 of *LNCS*, pp. 463–480, Berlin, 2008. Springer.

[5] E. Kranakis, D. Krizanc, and E. Markou. Mobile agent rendezvous in a synchronous torus. In J. Correa, A. Hevia, and M. Kiwi, editors, *Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN 2006)*, Vol. 3887 of *LNCS*, pp. 653–664, Berlin, 2006. Springer.

[6] S. Alpern, V.J. Baston, and S. Essegaier. Rendezvous search on a graph. *Journal of Applied Probability*, Vol. 36, No. 1, pp. 223–231, 1999.

[7] E. Kranakis and D. Krizanc. An algorithmic theory of mobile agents. In R. Bruni and U. Montanari, editors, *2nd Symposium on Trustworthy Global Computing*, No. 4661 in LNCS, pp. 89–97, Berlin, 2007. Springer.

[8] S. Ghosh. Distributed systems: an algorithmic approach. 2007.

[9] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Rendezvous and election of mobile agents: Impact of sense of direction. *Theory Comput. Syst.*, Vol. 40, No. 2, pp. 143–162, 2007.

[10] D. Baba, T. Izumi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Space-optimal rendezvous of mobile agents in asynchronous trees. *Structural Information and Communication Complexity*, pp. 86–100, 2010.

ライトを保持した自律分散ロボット群による基地局集合問題について

貝野　太地　　泉　泰介

近年，自律分散ロボット群の協調動作のためのアルゴリズム設計が注目を集めている．これらの多くの研究のロボットは平面上の点と構成されており，ロボット自身の能力は大きく制限されたものと仮定される．同アルゴリズム設計の主要な目的は，制限されたモデル上で与えられたタスクにおける可解性を明らかにすることである．本研究では，視野の制限された自律分散ロボット群による基地局集合問題について検討する．本研究の最終的な目的はn個の全てのロボットを基地局に集合させることである．ここで基地局集合問題とは，一点集合問題の変種であり，事前に定められたある一点に存在するロボット (基地局) のもとへと全てのロボットを集合させる問題として定義される．基地局集合問題はリーダの存在する場合における一点集合問題とみなすことが可能である．本研究では，同問題に対して，通信が可能な記憶有りロボットを用いた解法を提案する．

本研究において，各ロボットは以下の特徴を持つ．

- 匿名性 : 各ロボットは自身と他のロボットを区別する特定のIDを持たない．
- 一様性 : 各ロボットは共通のアルゴリズムで動作する．
- 通信有 : 各ロボットは他のロボットに自身の状態を色で伝える．
- メモリ有 : 各ロボットはラウンド数，自身の色，特定のロボットの配置を記録するためのメモリを持つ．
- センサー有 : 各ロボットは自身の周囲の他のロボットの配置，色を確認するためのセンサーを保持している．

各ラウンドにおいて，各ロボットは以下の$Look-Compute-Move(LCM)$サイクルを実行する．

1. $Look$: センサーを用いて自身の周囲のスナップショットを取る．
2. $Compute$: アルゴリズムに従って，スナップショットから目的地を決定，内部状態を更新する．
3. $Move$: 目的地に移動する．

以上のサイクルを繰り返し，本研究ではn台のロボットが基地局に集合する．

本研究では，基地局に対してn個のロボットが集合する問題を検討する．この問題は1台のリーダーが存在する集合問題と類似している．リーダーが居る集合問題の上界として知られている最も高速なものは$O(n^2)$ラウンドであるが，本研究では，時間複雑度に注目し，ロボットにメモリとライトの特徴を持たせることで，$O(D)$ラウンドで解くアルゴリズムを提案する．ここでDは初期状況における可視性グラフの直径である．また，ロボットが利用するライトの色数は2であり，記憶領域として4ビットと1台のロボットの座標を記憶しておくためのレジスタのみを必要とする．

セッション２

分散アルゴリズム２

無線ビープネットワークにおける極大マッチングアルゴリズム

小野優也† 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 コンピュータサイエンス専攻

†† 奈良先端科学技術大学院大学 情報科学研究科

概要　本稿では，ビープモデルを用いて極大マッチングを解く乱択アルゴリズムを提案する．ビープモデルとは，ノードの通信能力を搬送波の送信と検知のみに限定したブロードキャストモデルである．本アルゴリズムは，ネットワークトポロジは任意かつ匿名ネットワークを仮定している．さらに，ノードの初期知識は使用可能チャネル数 $\mathcal{F}$ のみである．また，各ノードは同期起動かつスロット同期される．同期起動とは，ある時刻で全てのノードが一斉にアルゴリズムを開始することであり，スロット同期とは一つの通信アクションの時間が全てのノードにおいて等しいことである．本稿では，はじめに諸定義とモデルを説明し，次に提案アルゴリズムの概要と今後の研究方針について述べる．

1　はじめに

本稿では，無線ネットワークにおける極めて厳しいブロードキャストネットワークモデルであるビープモデルを用いて，極大マッチングアルゴリズムを考える．ビープモデルとは，ノードの通信能力を搬送波の送信 (ビープ) と検知 (リッスン) のみに限定したモデルであり，2010 年に実装手法が提案 [6] されてから様々なアルゴリズムがビープモデルで提案されている [1–3,5,7,8]．搬送波のみを用いるため，メッセージパスモデルと比較して通信コストが低いことや小型機器への実装が容易であることが特徴である．以降では，はじめに諸定義とビープモデルを説明し，次に極大マッチングを解く乱択アルゴリズムについて説明する．最後に今後の研究方針について述べる．

2　諸定義

ノード集合を V，リンク集合を E として，ネットワークを無向グラフ $G = (V, E)$ とする．ここで，システムに参加するノード数を $n = |V|$ とし，リンク $(v, w) \in E$ をビープを相互に受信できるノードの組とする．ノード u の隣接ノードを $N(u) = \{v \in V | \{u, v\} \in E\}$ とする．また，ノード u から高々 h ホップ内のノードを $N_h(u)$ とする．

ノード $u \in V$ に関して $d(u) = |N(u)|$ をノード u の次数とし，G の最大次数を $\Delta = \max_{u \in V} d(u)$ で表す．次に，グラフ理論で用いられるマッチング集合と極大マッチング集合の定義を示す．

定義 2.1 無向グラフ G について，辺部分集合 $M \subseteq E$ の互いに異なる任意の二つの辺が異なる端点を持つとき，M を**マッチング集合**とする．

定義 2.2 無向グラフ G について，M がマッチング集合かつ任意の辺 $e \in E - M$ に対して $M \cup \{e\}$ が G のマッチングではないとき M を**極大マッチング集合**とする．

3　モデル

ノードの動作は文献 [2] の定義を用いる．時間を長さ μ で分割し，分割された時間を**スロット**とよぶ．全ノードの μ は等しく，μ 時間経過後には次のスロットを開始する．また，各ノードは 1 スロット毎に以下の動作を行う．

- ビープもしくはリッスン
- 局所計算

さらに，全ノードは同じ時刻で起動し，1 スロット目からアルゴリズムを開始する．このとき，各ノードは隣接ノードやグラフ構造などの情報は知らず，自身も識別子を持たない．ビープモデルでは，あるスロットでビープもしくはリッスンしたノードは隣接ノードの通信アクションに応じてそれぞれ以下の情報を区別する．

送信者側　どの隣接ノードもビープしなかった (サイレント) もしくは隣接ノードのうち少なくとも一つのノードがビープした (衝突)．

受信者側　どの隣接ノードもビープしなかった (サイレント) かちょうど一つの隣接ノードがビープした (ビープ) もしくは少なくとも二つのノードがビープした (衝突)．

3.1　マルチチャネルモデル

さらに本稿では，文献 [4] で用いられるマルチチャネルモデルをビープモデルでも同様に定義する．各ノードは 1 スロット毎にビープもしくはリッスンを一つのチャネル c に対して行う．あるスロットでノード u がチャネル c を選択したときに，どの隣接ノードも c と異なるチャネルを選択した場合は，隣接ノードの動作に関わらずに u はサイレントと判別する．使用できるチャネル数を $\mathcal{F}$ で表し，$\mathcal{F} > 2(\Delta - 1)$ であると仮定する．

本稿では，以上のマルチチャネルビープモデルを用いるため，マッチング集合と極大マッチング集合を以下のように定義する．

定義 3.1 各ノードが高々一つのチャネルを決定し，それらがちょうど一つの隣接ノードと等しいとき，その状況を**マルチチャネルモデルにおけるマッチング**と呼ぶ．

定義 3.2 チャネルを決定していないノードがチャネルをどのように選択してもマッチングを満たさないとき，その状況を**マルチチャネルモデルにおける極大マッチング**と呼ぶ．

4　極大マッチングアルゴリズム

本節では，マルチチャネルビープモデルを用いて定義 3.2 の極大マッチング問題を解くアルゴリズムを提案する．アルゴリズムの基本方針は，まず各ノードが 1 から $\mathcal{F}$ の中からチャネルを一つ選択し，それらがちょうど一つの隣接ノードと等しければマッチングしたと判定して終了し，

Algorithm 1 アルゴリズムのアウトライン

1: $match \leftarrow \perp, state \leftarrow \mathbb{L}, ch \leftarrow \{1, 2, \ldots, \mathcal{F}\}$
2: **loop**
3: $candidate \leftarrow \perp$
4: *Step1 //チャネル候補を選出
5: *Step2 //マッチング生成
6: *Step3 //隣接ノードとの状態・マッチング済みチャネル通知

Step 1 チャネル候補の選出

1: $C \leftarrow$ random choice in ch
2: beep on C
3: **if** (collision) **then**
4: $candidate \leftarrow C$

マッチングしていないノードはマッチング不可能である状態を検知したときに終了する．また，チャネル 0 を隣接ノード間との状態通知のための特別なチャネルとして用いる．Algorithm1 に極大マッチングアルゴリズムの概要を示す．ノードは以下の局所変数を保持する．

- *state* ノードの状態，$\mathbb{L}$:未マッチング，$\mathbb{M}$:マッチング，最初は $\mathbb{L}$
- *match* マッチングしたチャネル番号，最初は空
- *ch* マッチングに使用可能なチャネル番号のリスト，最初は $1..\mathcal{F}$

全ノードは最初休眠状態であり，ある時刻 t_s で一斉に起動する．したがって，全てのノードはアルゴリズムのどの段階においても同期されている．起動直後に局所変数を初期化し，その後 Step1-3 の処理をアルゴリズムが終了するまで反復する．以降では，Step1-3 について解説する．

4.1 Step1:チャネル候補の選出

各ノードはマッチング用に使用可能なチャネル集合から一つのチャネルを乱択して，そのチャネルに対してビープを行う (図 1(a))．このとき衝突を検知したノードは，一つ以上の隣接ノードが自身と同じチャネルを選択したことが分かるため，そのチャネルを候補チャネルとして記憶する (4 行目)．衝突を検知しなかったノードは，隣接ノード全てが自身と異なるチャネルを選択したため，候補無しとして次のステップに移行する．

4.2 Step2:マッチング生成

候補チャネルが無いノードは，このステップでは待機する (図 1(b) ノード l)．候補チャネルがあるノードは，自身の候補チャネルがちょうど一つの隣接ノードと等しいかを確認する．具体的には最初の 2 スロットを用いて，1 スロット目でビープし 2 スロット目でリッスンするか，もしくはその逆を行うかを 1/2 の確率で選択する (5,6 行目:2exchanges)．このとき，衝突を検知したらマッチングに失敗したと判定する (図 1(b) ノード j, k)．一方で，図 1(b) のノード i のように，2exchanges で衝突を検知しなかった場合でも実際はマッチングに失敗している状況がある．そこで，2exchanges でマッチング失敗と判定したノードは，3 スロット目で隣接ノードにビープして失敗通

Step 2 マッチング生成

1: **if** ($candidate = \perp$) **then**
2: **wait** for 3 slots
3: **else**
4: *** 2 exchanges ***
5: with probability 1/2 beep on *candidate* and listen
6: otherwise listen on *candidate* and beep
7: **if** (collision in 2 exchanges) **then**
8: beep on *candidate*
9: **else**
10: listen on *candidate*
11: **if** (silent) **then**
12: $match \leftarrow candidate$
13: $state \leftarrow \mathbb{M}$

知を行う (8 行目)．もし，最初の 2 スロットで衝突を検知しなかったかつ 3 スロット目で失敗通知を受けなかったノードは，マッチング生成に成功したと判定し，自身の状態を $\mathbb{M}$ にする (13 行目)．

4.3 Step3:隣接ノードとの状態・マッチング済みチャネル通知

最初の 1 スロットでは，状態 $\mathbb{L}$ のノードが隣接に状態 $\mathbb{L}$ のノードが存在するかの確認を行う．具体的には，状態 $\mathbb{L}$ のノードはチャネル 0 でビープし，衝突を検知しなかったら，全ての隣接ノードが状態 $\mathbb{M}$ でありマッチング不可能であることが分かるため，状態 $\mathbb{L}$ としてアルゴリズムを終了する (3 行目)．次の $\mathcal{F}$ スロットでは，Step2 でマッチングに成功したノードは，隣接ノードにマッチングしたチャネルを通知し (8 行目)，それを受けた状態 $\mathbb{L}$ のノードはそのチャネルを使用可能チャネルから除外する (13 行目)．これは，隣接ノードが他のノードと既にマッチングしている場合，そのチャネルではマッチング不可能となるからである．また，マッチングチャネルを通知したノードはその時点でアルゴリズムを終了する (9 行目)．

5 おわりに

本稿では，ビープモデルを用いた極大マッチングアルゴリズムを提案した．本アルゴリズムは，ネットワークトポ

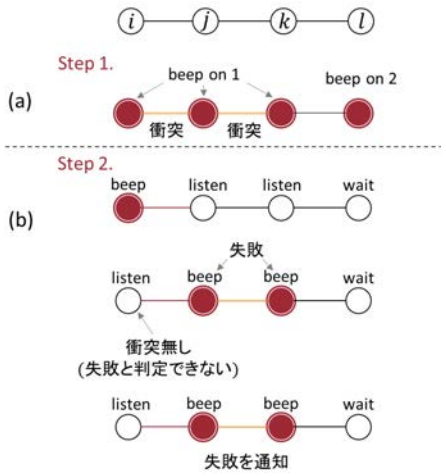


図 1: Step1 と Step2 の実行例

Step 3 隣接との状態通知・検出

1: **if** $(state = \mathbb{L})$ **then**
2: beep on 0
3: **if** (silent) **then terminate**
4: **else**
5: **wait** for 1 slot
6: **for each** $x \in \{1, 2, \ldots, \mathcal{F}\}$ **do**
7: **if** $(state = \mathbb{M} \wedge x = match)$ **then**
8: beep on x
9: **terminate**
10: **else if** $(state = \mathbb{L})$ **then**
11: listen on x
12: **if** (beep or collision) **then**
13: $ch \leftarrow ch \setminus \{x\}$
14: **end for**

ロジが任意かつ匿名であり，ノードがチャネル数 $\mathcal{F}$ しか知らない条件で動作し，解の出力は必ず正しいが実行時間は確率的であるラスベガスアルゴリズムである．また，アルゴリズムの開始時刻は全てのノードで同じであるが，文献 [5] でビープモデルにおける同期起動アルゴリズムが提案されているため，厳しい制約ではないといえる．今回はチャネル数 $\mathcal{F}$ に制限を加えているが，この制限を除去した場合においてもアルゴリズムに少し変更を加えるだけで動作する．しかし，定義 2.2 と合致した解が得られないため，今回は制限付きのアルゴリズムを提案した．

今後の課題としては，本アルゴリズムの正当性と実行時間の解析を行うことである．さらに，本アルゴリズムを衝突検知無しモデルに適用する予定である．

参考文献

[1] Y. Afek, N. Alon, Z. Bar-Joseph, A. Cornejo, B. Haeupler, and F. Kuhn, *Beeping a maximal independent set*, Distributed Computing **26** (2013), no. 4, 195–208.

[2] A. Cornejo and F. Kuhn, *Deploying wireless networks with beeps*, Proceedings of the 24th international conference on distributed computing, 2010, pp. 148–162.

[3] A. Czumaj and P. Davies, *Communicating with beeps*, CoRR **abs/1505.06107** (2015).

[4] S. Daum, M. Ghaffari, S. Gilbert, F. Kuhn, and C. Newport, *Maximal independent sets in multichannel radio networks*, Proceedings of the 2013 acm symposium on principles of distributed computing, 2013, pp. 335–344.

[5] K.-T. Frster, J. Seidel, and R. Wattenhofer, *Deterministic leader election in multi-hop beeping networks*, Distributed computing, 2014, pp. 212–226.

[6] R. Flury and R. Wattenhofer, *Slotted programming for sensor networks*, Proceedings of the 9th acm/ieee international conference on information processing in sensor networks, 2010, pp. 24–34.

[7] B. Huang and T. Moscibroda, *Conflict resolution and membership problem in beeping channels*, International symposium on distributed computing (disc) 2013.

[8] Y. Métivier, J. M. Robson, and A. Zemmari, *On distributed computing with beeps*, ArXiv e-prints (July 2015), available at 1507.02721.

モバイルノードを導入したトリガ数え上げアルゴリズム

安達 駿　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　センサネットワークにおいてモニタリングは重要な問題である．代表的なモニタリングの一つに，ネットワーク全体で一定数のトリガが発生したことを検出するトリガ数え上げ問題がある．各ノードではメッセージの送受信に電力を消費する一方で，センサネットワークなどでは消費可能電力が限られていることが多いため，できる限り少ないメッセージ数でこの問題を解くアルゴリズムが必要とされている．そこで本稿では，トリガが正規分布に従って発生するような場合を考え，効率的にトリガ数え上げ問題を解くアイデアを説明し解析を行う．そして最後に今後の課題について述べる．

1　はじめに

センサネットワークなどのネットワークにおいて，天候の状態，ログインした端末の数などを監視する**分散モニタリング**は重要な問題である．その分散モニタリングにおいて，降雨量やログインした端末の数などの発生した事象（トリガ）が一定数（以降，**検出トリガ数**）を越えたことを検出しユーザに知らせる問題は**トリガ数え上げ問題**（図 1）と呼ばれ研究が行われている．[1]

図 1: 検出トリガ数が 5 個の場合の検出例

センサネットワークの場合，各ノードは電力の限られたバッテリで動作することが多く，そのバッテリの交換にも多大なコストがかかるため，できるだけメッセージのやりとりを少なくするようなアルゴリズムが望まれている．

そこで文献 [1] では，ノードの数を n，検出トリガ数を w とした時，検出までにシステム全体で交換するメッセージの数（以降，**総メッセージ数**）の下界が $\Omega(n \log w)$ であることを示し，総メッセージ数が最適である $O(n \log w)$ となる集中型アルゴリズムを提案している．しかしこのアルゴリズムは，あるノードがシステム全体を連携させる代表としての役割を果たし，その他のノードはそのノードに従うという集中型制御に基づいているため，各ノードが受信するメッセージの最大数（以降，**最大受信数**）は総メッセージ数と同じ $O(n \log w)$ となってしまうことが分かっている．そこで文献 [2] では，乱択アルゴリズムによりメッセージを各ノードに分散して受信させ，総メッセージ数は最適のまま，最大受信数を高い確率で $O(\log w)$ に改善することに成功している．

これらのアルゴリズムでは，どのノードでトリガが発生するか，またそのタイミングは予め知ることができないと仮定している．しかし，集中的な降雨などは，過去のトリガ発生情報をもとに推測可能な場合も考えられる．そこで本稿では，トリガの発生確率が事前に分かっている場合（もしくは推測が可能な場合）について考え，より効率的にトリガ数え上げ問題を解くことを考える．具体的には，トリガが正規分布に従い発生するような場合について考え，簡単な検出のアイデアの説明とその解析を行う．そして最後に，今後の研究方針について述べる．

2　諸定義

2.1　モデルと問題

本稿で考える分散システムは $m \times m$ のグリッドを想定している．以下が分散システムの定義である．

分散システム $G = (V, E)$ は $n = m \times m$ 個のノードで構成されている。各ノードは位置する座標 $[0, m-1] \times [0, m-1]$ を表す固有の識別子（以降，ID）を持っており，ノードの集合 $V = \{v_{0,0}, v_{0,1}, \cdots, v_{i,j}, \cdots, v_{m,m}\}$ と表される．ただし，ノード $v_{i,j}$ は座標 (i, j) に位置するものとする．また，各ノード $v_{i,j}$ はリンクを通じて，他のノードと通信を行う．このノード間のリンクの集合を E とし，本稿ではグリッドを想定しているため $E = \{(v_{i,j}, v_{i-1,j}), (v_{i,j}, v_{i+1,j}), (v_{i,j}, v_{i,j+1}), (v_{i,j}, v_{i,j-1}) | v_{i,j} \in V\}$ となる．(図 2)

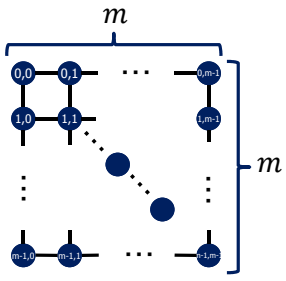


図 2: $m \times m$ グリッドネットワーク

このような分散システム G の各ノードで，トリガの発生する座標それぞれが正規分布 $f(x) = \frac{1}{\sqrt{2\pi}\rho} exp\{-\frac{(x-\mu)^2}{2\rho^2}\}$ に従い発生する状況を考える．(x はノードの ID，ρ^2 は分散，μ は平均を意味している.）各ノードは正規分布によりトリガが発生することを知っていると仮定しているため，トリガが最も集中して発生するノードの ID(μ_x, μ_y) を知っている．そしてシステム全体で発生したトリガの総数がある与えられた値 w（以降，**検出トリガ数**）に達したことを，いずれかのノードが検出する問題を**トリガ数え上げ問題**と呼んでいる．

2.2 アルゴリズムの評価尺度

メッセージの送受信には電力を消費するが，センサネットワークなどでは各ノードの消費可能な電力が限られていることが多い．あるノードの送受信するメッセージ数が多数ある場合，そのノードのライフタイムを縮める要因となる．そこでアルゴリズムの性能評価尺度として，次の 2 つの評価を行うことにする．

- **総メッセージ数**（totalMsg）
  －システム全体で交換されるメッセージの総数
- **最大受信数**（maxRcv）
  －各ノードが受信するメッセージの最大数

3 アルゴリズムと評価

3.1 アルゴリズム

本稿の検出アルゴリズムは非常にシンプルである。正規分布が平均値 μ の付近にデータが集中する分布であることを利用し，そのピークとなるノード v_{μ_x,μ_y}（以降，**代表ノード**と呼ぶ）が，メッセージによりトリガ情報を集め，システムを代表して発生トリガ数を数えるというものである．各ノード $v_{i,j}(\neq v_{\mu_x,\mu_y})$ の動作は Algorithm1 の通りである．概要としては，そのノードでトリガが発生した場合，代表ノードへトリガ発生を通知する TRIGGER メッセージを伝達するというものである．この TRIGGER メッセージは，トリガを検出したノードから代表ノードまで最短経路を用いて伝達される．

Algorithm2 は代表ノード v_{μ_x,μ_y} の動作を表している．発生したトリガと各ノードから送られてくる TRIGGER メッセージを数え，その数が検出トリガ数 w を越えた場合にユーザへ通知を行っている．

Algorithm 1 各ノード $v_{i,j}(\neq v_{\mu_x,\mu_y})$ の動作

Constants:
　w：検出トリガ数
　v_{μ_x,μ_y}：代表ノード
Function: トリガが発生した場合
　＜代表ノード v_{μ_x,μ_y} 宛にメッセージ送信＞
1:　自分自身 $v_{i1,j1}$ 宛に TRIGGER メッセージを送信
Function: TRIGGER メッセージを受信した場合
2: **if** $(i < \mu_x)$ **then**
3:　$v_{i+1,j}$ 宛に TRIGGER メッセージを転送
4: **else if** $(i > \mu_x)$ **then**
5:　$v_{i-1,j}$ 宛に TRIGGER メッセージを転送
6: **else if** $(j < \mu_y)$ **then**
7:　$v_{i,j+1}$ 宛に TRIGGER メッセージを転送
8: **else if** $(j > \mu_y)$ **then**
9:　$v_{i-1,j-1}$ 宛に TRIGGER メッセージを転送

Algorithm 2 代表ノード v_{μ_x,μ_y} の動作

Constants:
　w：検出トリガ数
Variables:
　$counter \leftarrow 0$：カウンタを初期化
Function: TRIGGER メッセージを受信した場合
1: $counter++$
2: **if** $(counter == w)$ **then**
3:　検出トリガ数 w に達したことをユーザに通知
Function: トリガが発生した場合
4: $counter++$
5: **if** $(counter == w)$ **then**
6:　検出トリガ数 w に達したことをユーザに通知

3.2 シミュレーションによる評価と考察

グリッドのサイズ $m = 10, 20, \cdots, 200$ について総メッセージ数，最大受信数について，シミュレーションにより評価を行った．シミュレーションの際のパラメータは表 1 の通りである．

検出トリガ数	100000
平均 μ	$\lfloor m/2 \rfloor$
分散 ρ	1.0
代表ノードの ID(μ_x, μ_y)	$(\lfloor m/2 \rfloor, \lfloor m/2 \rfloor)$

表 1: シミュレーションのパラメータ

総メッセージ数，最大受信数は図 3,4,5 のようになった．

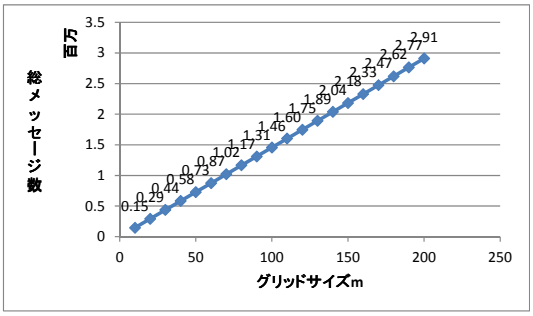


図 3: シミュレーション結果：総メッセージ数

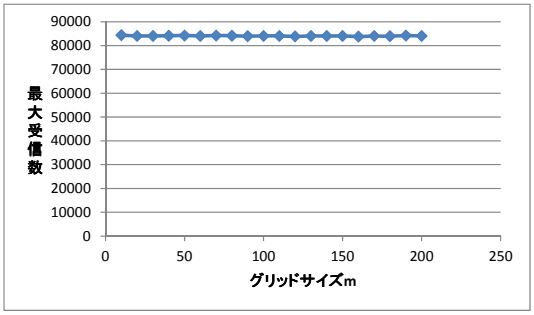


図 4: シミュレーション結果：最大受信数

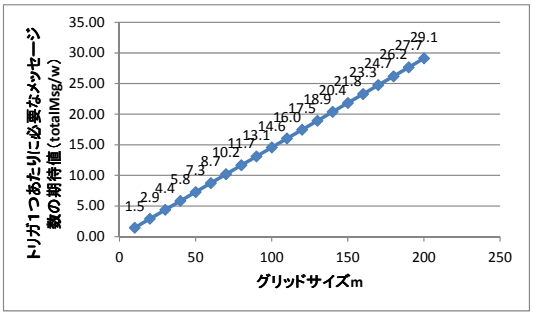


図 5: トリガを代表ノードに知らせるのに必要なメッセージ数の期待値

図 5 より，トリガ 1 つに対するメッセージ数の期待値はグリッドのサイズ $m = \sqrt{n}$ に比例していること分かる．トリガ数え上げ問題の下界（$\Omega(n \log w)$）が n に比例したものであったことを考えると，改善が期待できる結果となっている．その一方で，図 4 のように，１つのノード（代表ノード）にメッセージが集中してしまう欠点があることも分かる．

4　今後の課題

本稿では，正規分布に従いトリガが発生する場合についてシンプルなアルゴリズムのシミュレーションを行い，メッセージ数の評価を行った．

今後の課題としては，図 4 からも分かるように，本稿の手法は１つのノード（代表ノード）にメッセージが集中してしまう欠点の改善が挙げられる．そこで近年注目されているドローンのようなモバイルノードの導入を検討したいと考えている．モバイルノードは，電力の制限を考慮しなくてよいという長所がある一方で，通信範囲外のノードとの通信には大幅な遅延が発生するという欠点がある．そこでこのモバイルノードが代表ノードの役割を果たすことで，効率的なアルゴリズムを考案できるのではないかと考えている．さらに，集中的にトリガが発生する場所が変化していくような（トリガ発生の正規分布のピークが変化していくような）場合についても考え，モバイルノードの制御を含めたアルゴリズムを検討していきたいと考えている．

参考文献

[1] Rahul Garg, Vijay K.Garg, and Yogish Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *Proceedings of the 20th annual international conference on Supercomputing(ICS)*, pp. 269–277, 2006.

[2] Venkatesan T. Chakaravarthy and Yogish Sabharwal. An optimal decentralized algorithm for the distributed trigger counting problem. Technical report, IBM Research - India, 2011.

Time-varying Graph における 1 対 1 の k トークン転送アルゴリズム

小森康祐† 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

†† 奈良先端科学技術大学院大学情報科学研究科

概要　近年インフラを利用しない動的なネットワークにおける研究が盛んになっている. 動的なネットワークの例としてモバイルアドホックネットワーク, センサネットワーク, 車両ネットワークなどがある. これらのネットワークは time-varying graphs としてモデル化される. 本研究では, このような動的なネットワークにおいて, 1 対 1 の k-token 転送問題を提案し，この問題を解く逐次アルゴリズムについて検討する．まず転送を行う際に使用する経路を構築する方法が最小費用フロー問題を解くことにより求められることを示し，その後求めた経路を利用し転送を行う決定性アルゴリズムを示す.

1　はじめに

近年無線端末のみで構成されたインフラを必要としないネットワークに対する研究が盛んとなっている．その中でもトポロジが時間により変化する動的ネットワークに注目が集まっている．それらの動的ネットワークは delay-tolerant,disruptive-tolerant,opportunistic といった様に様々な呼称があるが，多くの動的ネットワークモデルでは，常時連結であることを仮定していない. つまり，ある時間においてはネットワークは非連結であるかもしれない，しかしある時間以内には複数ノードを経由して任意の 2 ノード間で通信が出来るという仮定が一般的である.

ネットワークの動的変化には，ノードやエッジが出現・消滅する，エッジの帯域が変化するなど様々な物がある．そのような動的ネットワークのモデルの一つとしては，**Time-varying Graph(TVG)** が提案されている [1]．例えば，エッジが出現・消滅する動的ネットワークは TVG 上のエッジ集合に存在関数（エッジ存在関数と呼ぶ）を設定することで，ある時間にはエッジが存在し，またある時間にはエッジが存在しないという状況を一つのグラフ上で表現することができる. しかし，システムの複雑性のために解析的結果はまだ少ない.

TVG における多くの研究は，エッジの出現関数などをマルコフ過程などを仮定し確率的に扱っている．一方で決定性の解法を考える際には，エッジ存在関数がわかっているという仮定のもとで，最適なブロードキャストやルーティングを実現するための送信スケジュールを求めることが多い．そのようなネットワーク上での問題を解く際には，完全では無いとしてもエッジスケジュールに関する前提の知識があれば，より問題が解きやすい．さらに実際のスケジュールパターンを利用することで，実用的な解が得られることも期待できる [2].

本稿ではエッジの存在関数がある制約を持つネットワーク上において，特定のノードから目的のノードへと k 個のトークンを転送する k-token 転送問題に取り組む．以降ではまず諸定義を示し，その後最小費用流を利用して k-token 転送問題を解くアルゴリズムを示す．最後に，本稿のまとめと今後の研究方針について述べる.

2　諸定義と問題の概要

2.1　再帰 Time-varying Graphs

本稿では，エッジが一定のスケジュールを持つネットワークについて，エッジが再帰性を持つ再帰 Time-varying Graph として表現する．再帰 TVG の基本グラフを $G = (V, E)$ とする．ノード数 $|V| = n$ であり，E は断続的に利用可能なエッジ集合である．各エッジ $e \in E$ は出現・消滅を繰り返すが，高々 Δ 時間待てば必ず再出現するものとする．なおこの再帰性については文献 [2] に従って以下のように定義する.

$$\forall e \in E, \forall t \in \mathcal{T}, \exists t' \in [t, t+\Delta), \rho(e, t') = 1, \quad \text{for some} \Delta \in \mathbb{T} \text{ and } G \text{ is connected} \quad (1)$$

エッジの両端にある接続ノードは，エッジが出現している間はお互い通信が可能である．また各エッジの出現と消滅は接続ノードにより瞬時に検知される.

2.2　1 対 1 の k-token 転送問題

再帰 TVG 上の任意の 2 ノードを，ソースノード S と受信ノード T と定める．ソースノードは k 個のトークンを保持しており，全てのトークンを受信ノードへ転送できればアルゴリズムは終了となる．各トークンは分割・統合は出来ない．各ノードが出来るのはトークンの保持・コピー・転送のみである．エッジが出現したら少なくとも 1 つのトークンを転送することが可能である．また，各ノードは各エッジに対して送りたいトークンを選択できる．今回は最悪時（アルゴリズムにとって最悪のエッジ関数）における実行時間のみを考える.

3　提案手法

3.1　アイデア

ソースノードからネットワーク上のノードへデータを散布する場合には，フラッディングを用いたり [3]，ネットワーク上にソースノードを根とした木を構築し，その木を利用してデータをブロードキャストすることが多い．ソースノードから特定の 1 ノードへのデータの転送という問題も木を作ることで解くことも出来るが，その場合データの転送に利用する経路は 1 つだけとなる．そこで，今回のアルゴリズムではソースノードから目的のノードへデータを転送する際に複数の**辺素**な経路を構築し，トークンを並行的に転送することを可能にする．ただし，最悪のエッジ存在関数を考えると，経路に現れる各エッジに対し，Δ 時間に 1 つのトークンしか転送することが出来ない．そこで，各辺の容量を 1，費用を Δ としたネットワークを考え，最小費用フローを解くことで，最適な経路を構築できることを示す．

3.2　ホップ数と転送可能なトークン数の関係

最悪のエッジ存在関数を考えると，各辺に対し，Δ 時間につき 1 つのトークンを転送できることがわかる．つまり S ホップの経路で 1 つのトークンを転送するのに要する時間は $S\Delta$ 時間となる．更に，S ホップの経路で I 個のトークンを転送するのに要する時間は $(S+I-1)\Delta$ 時間となる (図 1)．これは，先頭のトークンが目的のノードの 1 つ手前に届くまでに $(S-1)\Delta$ 時間，そのノードから目的のノードへの 1 ホップで I 個のトークンを転送するのに要する時間が $I\Delta$ 時間かかるためである．

図 1: S ホップの経路での I 個のトークン転送時間

3.3　最小費用フロー

ソースノードから目的のノードへの転送を考える際には，どのような経路を構築すれば最悪時に最も早く転送できるかを考える必要がある．ここで，ソースノード S から目的のノード T との間に q 個の辺素な経路 $P_1, P_2, \ldots, P_q$ が存在するとする．各経路はホップ数が異なる．最小ホップ数の経路 P_1 の長さを S ホップとする．また，各経路 $P_i (1 \leq i \leq q)$ の長さは $S+Y_i$ (ただし, $Y_1 = 0 \leq Y_2 \leq \ldots \leq Y_q$) ホップとする．各経路で同じ数だけのトークンを送った時に，それぞれの経路で転送に要する時間は図 2 のとおりとなる．ここで，図 2 の左下，経路ホップ数合計の部分に着目する．各経路で送るトークン数に関わらず，この経路のホップ数合計が小さくなるほど全体としての実行時間が早く終了することがわかる．このことから，経路ホップ数合計が最小となる，辺素な m 個の経路を求める問題は，全経路の容量が 1 でコストも 1 のネットワーク上でフローが m の最小費用フローを求める問題に帰結することが出来る．

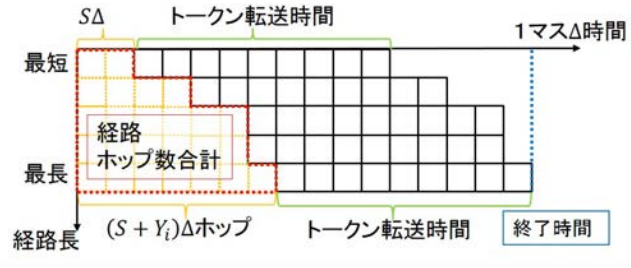


図 2: トークン転送に要する時間

基本的には，多くの経路で転送を分散するほうが早く実行が終了するが，場合によっては少ない経路数を利用するほうが早く実行が終了することがある．例えば，図 3 のように，経路を 1 つだけ構築すればホップ数は 10 で済むが，2 つ構築するとなると各経路のホップ数が膨大になってしまう場合である．

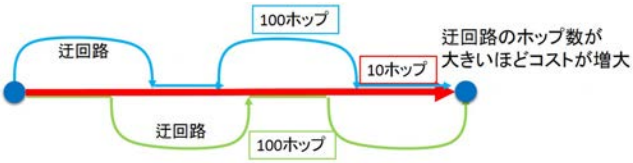


図 3: 少ない経路の方が早く実行できる例

このようなケースを考慮するため，経路数ごとに最小費用フローを求め，その中でも実行時間が最小となる経路数を採用し転送を行うことで，最悪時に最も早く実行を終了することが可能になる．構築可能な最大経路数を求めるには，最大フローを考えれば良い．つまり，経路数 1 の場合の最小費用フロー，更に 2 の場合，3 の場合と徐々に経路を増やし，最大値までを求めた後，実行時間最小となる経路を利用する．最小費用フローと最大フローに要する時間はそれぞれ，$O((m+n\log n)m)$ と $O(mn\log n)$ となる．なお，n はノード数，m はエッジ数とし，全ての流量を整数とする．従って，ソースノードから目的ノードへの辺素な経路が最大 q 個存在する場合，経路構築に要する時間は $O((m+n\log n)mq)$ になる．$q \leq n$ なので，この計算時間は $O((m+n\log n)mn)$ と表せる．

3.4 k トークン転送アルゴリズム

最適となる辺素な経路が選択できたら，各経路を用いて合計 k 個のトークンを転送する．基本的には，すべての経路での転送がほぼ同時に終了するように，ホップ数が短い経路ほど多くのトークンを転送する．例えば，以下の状況を考える．

- 使用できる辺素な経路の本数は X 本
- 最短経路のホップ数は S
- 各経路 $P_i(1 \leq i \leq X)$ と最短経路とのホップ数の差は Y_i
- 各経路 $P_i(1 \leq i \leq X)$ のホップ数は $S + Y_i$

この時，全体の実行時間が最も早く終了できる最適な配分でトークンを転送出来る状況を考えると，図 4 のように最短経路での転送完了時に全トークンの転送が完了するように配分すれば良い．

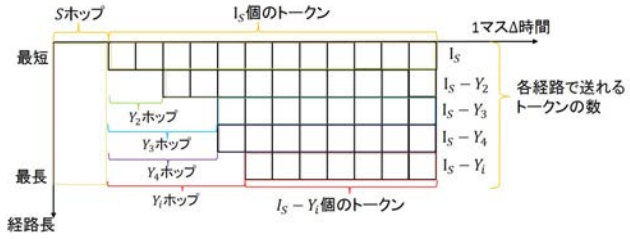


図 4: 最適な配分時の実行時間

最も短い経路で転送出来るトークン数を I_S とすると，この経路を用いた，トークン転送に要する時間は $S + I_S$ で表せる．そこで I_S を求める．図の通り，経路 1 から I_S 個届いた時点で，各経路が最悪時に転送出来るトークン数は高々 $I_S - Y_X$ となる．ここで全トークン数は k 個なので以下の式が成り立つ．

$$X \cdot I_S - \sum_{i=0}^{X} Y_i = k \tag{2}$$

これを整理すると I_S は以下のように表せる．

$$I_S = \frac{k + \sum_{i=0}^{X} Y_i}{X} \tag{3}$$

よって k 個のトークンの転送に必要な時間は $S + I_S = S + \frac{k + \sum_{i=0}^{X} Y_i}{X}$ となる．またこの時，各経路 R_i が転送するトークンの数は $\frac{k + \sum_{i=0}^{X} Y_i}{X} - Y_X$ となる．

4 まとめと今後の方針

本稿では，最小費用フローを利用することで k-token 転送問題について，転送時間 $S + \frac{k+\sum_{i=0}^{X} Y_i}{X}$ のスケジュールを求めるアルゴリズムを示した．今後の研究方針としては，各エッジに対して異なる再帰時間が設定されているようなネットワークにおいて，転送スケジュールを求めるアルゴリズムを検討していく予定である．

参考文献

[1] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 27, No. 5, pp. 387–408, 2012.

[2] Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. Deterministic computations in time-varying graphs: Broadcasting under unstructured mobility. In *Theoretical Computer Science*, Vol. 323, pp. 111–124. Springer, 2010.

[3] Chinmoy Dutta, Gopal Pandurangan, Rajmohan Rajaraman, Zhifeng Sun, and Emanuele Viola. On the complexity of information spreading in dynamic networks. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 717–736. SIAM, 2013.

部分スナップショットアルゴリズムの効率的な並行実行の実現

渡部連太郎[1] 金鎔煥[2] 大下福仁[3] 角川裕次[1] 増澤利光[1]
1 大阪大学大学院 情報科学研究科
2 名古屋工業大学 情報工学専攻
3 奈良先端科学技術大学院大学 情報科学研究科

概要 計算機（以降ノードとする）同士が協調的に動作する分散システムにおいて，一部のノードの故障や離脱がシステム全体へ影響を与える可能性がある．そのため，いかに分散システムに耐故障性を持たせるかが重要である．システムに耐故障性を持たせる手法の 1 つにチェックポイントロールバック手法がある．これは，システムに故障が発生した際，事前に記録済みの安定状態 (チェックポイント) へシステムを復元 (ロールバック) する手法である．分散システムにこの手法を用いるため，スナップショットアルゴリズムを利用してシステム全体の状態を保存する必要がある．スナップショットとは，分散システム全体の状態を局所状態の集合として記録したものであり，スナップショットアルゴリズムはスナップショットを矛盾なく作成する手法を指す．本稿では CSS アルゴリズム [Kim et al, IEICE E97, No.1, 2014] の概要を紹介し，CSS アルゴリズムのメッセージ複雑度を改善するアルゴリズムを提案する．

1 はじめに

ネットワークを介して複数の計算機が相互に通信しながら協調して動作するシステムを分散システムと言う．分散システムでは一部のノードの故障がシステム全体に悪影響を及ぼすことがある．近年の分散システムでは大規模化が進んでおり，それに伴いシステムのどこかで故障が発生する確率も高くなる．このような背景のもとで，システムの耐故障性が重要になってきている．

システムに耐故障性を持たせる手法の 1 つにチェックポイントロールバックがある．これはシステムが正常なうちにチェックポイントとして安定状態を記録しておき，故障発生時には最新のチェックポイントへシステムを復元（ロールバック）するものである．分散システムに対してこの手法を適応する際，チェックポイントとして使用する分散システム全体の状態を保存する必要がある．矛盾のない分散システム全体の状態を記録するアルゴリズムをスナップショットアルゴリズムと言う [1]．

Chandy と Lamport によるスナップショットアルゴリズム (CL アルゴリズム)[1] は，手続きの簡潔さからその代表的なものである．しかし CL アルゴリズムでは，メッセージを送信可能なノード全てに送信して伝播させるため，適応するシステムの規模が大きくなるとメッセージ複雑度が増大し，実用が難しいという問題があった．

CL アルゴリズムを基に考案された Sub-Snapshot(SSS) アルゴリズム [2] はこの問題を解決している．これはスナップショットアルゴリズムを最初に開始するノード（initiator）と一部のノード集合 (スナップショットグループ) のみでスナップショットをとるアルゴリズムである．これにより同時にスナップショットをとるノード数を削減し，ノードの参加や離脱への対応にも成功した．しかし SSS アルゴリズムは並行的に実行されることを仮定していない．この仮定は，多くのノードがスナップショットを取得する必要のある大規模なシステムにおいては現実的ではない．仮に異なるノードを initiator としてアルゴリズムを並行実行すると，スナップショットグループが重複 (衝突) した場合に生成されるスナップショットに矛盾が発生する可能性がある．

SSS アルゴリズムを基に処理の並行実行を可能にした Concurrent Sub-Snapshot(CSS) アルゴリズム [3] は，スナップショットグループの衝突が発生した場合，スナップショットグループ同士を統合し，1 つのスナップショットグループとして機能するよう改良された．しかしスナップショットグループを統合するためには多くのメッセージ通信を行う必要がある．本稿では，各アルゴリズムの概要を述べた後に CSS アルゴリズムのメッセージ複雑度を改善するアルゴリズムを提案する．

2 既存研究

2.1 CL アルゴリズム

CL アルゴリズム [1] はマーカと呼ばれる特別なメッセージを用いてシンプルな動作を行うだけで，矛盾のないスナップショットを効率よく作成するアルゴリズムであり，以降で紹介するアルゴリズムの基盤となっている．initiator は最初に局所状態を記録し，各隣接ノードへマーカを送信する．initiator 以外のノードは，このマーカを受信することでアルゴリズムが開始する．初めてマーカを受信したノードは局所状態を記録し，各隣接ノードへマーカを送信する．initiator を含めた全てのノードは，全ての隣接ノードからマーカを受信するとアルゴリズムを終了する．

CL アルゴリズムを実行するためには，各ノードは隣接ノードを予め知っていなければならないという制約がある．実際の分散システムは動的な変化 (ノードの参加/離脱などによるネットワーク構造の変化) が頻繁に起こるが，制約により CL アルゴリズムはこれに対応していない．また，各ノードは送信できる全てのノードに対してマーカを送信するため，ネットワークが大規模化するとメッセージ数が膨大になる．

[1] 分散システムにおいて，あるノードが送信していないメッセージを他のノードが受信しているとき，このメッセージを orphan message と呼ぶ．分散システム全体で orphan message が存在しない記録の状態を“矛盾がない”という．

2.2 SSS アルゴリズム

SSS アルゴリズム [2] は，CL アルゴリズムを基に動的なノードの参加/離脱への対応とマーカ送信回数の削減に成功している．これはアルゴリズム内で initiator と動的な因果関係を持つノード集合 (スナップショットグループ) を決定し，その要素ノードのみとスナップショットをとることによる．

このアルゴリズムでは依存関係の概念を導入する．各ノードは，送信/受信，生成/被生成イベントが発生した際，関与したノード ID を動的に依存関係集合 (DS) へ追加していく．アルゴリズムを開始すると，initiator はまず CL アルゴリズムと同様に局所状態を保存し，ID を添付したマーカを，保持している DS に含まれる全てのノードに送信する．初めてマーカを受信したノードは局所状態を保存し，DS の各要素にマーカを送信，さらに initiator に DS を送信する．initiator はスナップショットグループを決定するため，受信した DS の和集合とその送信ノード ID の集合を管理する．この 2 つの集合が一致したとき，それが決定したスナップショットグループである．その後 initiator は，各ノードがマーカを受け取らなければならない受信元ノード集合を計算し，それぞれのノードへ送信してアルゴリズムを終了する．initiator からその集合を受信したノードは，集合に属する全てのノードからマーカを受信するとアルゴリズムを終了する．

注意点として，このアルゴリズムでは並行的に 2 つ以上の initiator がアルゴリズムを開始させることはないと仮定している．仮に異なる複数のノードを initiator として複数のアルゴリズムが同時実行された場合，スナップショットグループの衝突が発生すると正しい動作を保証できなくなる．

2.3 CSS アルゴリズム

SSS アルゴリズムを並行実行できるよう改良したものが CSS アルゴリズム [3] である．CSS アルゴリズムでは，並行実行した二つのアルゴリズム間でスナップショットグループの衝突が発生した際，矛盾がないよう衝突したスナップショットグループを統合する．衝突したスナップショットグループを管理していた initiator のうち一方が統合後のグループを管理する．この initiator を main initiator，他方を sub initiator と呼ぶ．

以下で統合の流れを説明する．あるグループからのマーカを既に受信しているノードが新たなグループからのマーカを受信したとき，ノードは衝突を検知し，属するグループの initiator へ衝突を報告する．報告を受け取った initiator は，もう一方のグループの initiator へ統合要請メッセージを送る．そしてメッセージを受け取った相手 initiator が，どちらが main initiator になるか決定し返信する．最後に sub initiator となる initiator は，main initiator へ保持していた依存関係集合やノード ID などの情報を送信する．

このアルゴリズムの注意点として，衝突回数が増加するとメッセージの送信回数が膨大になる点が挙げられる．衝突が増加し多くのスナップショットグループが統合されていると，衝突報告/統合要請メッセージを目的ノードへ届けるために必要な中継ノードが増加する．よってこのとき新たな衝突が発生するとメッセージの複雑度が大きくなると考えられる．

3 提案アルゴリズム

本章では，CSS アルゴリズムに比べ，アルゴリズム実行中の衝突回数が大きくなった場合のメッセージ送信回数を削減するアルゴリズムを提案する．CSS アルゴリズムでは衝突発生時，スナップショットグループを矛盾なく統合することにより並行実行を可能にしているが，衝突や統合の前後において，関連ノードは最初にマーカを受信した際に保存した局所状態の削除や，新たな局所状態の保存といった変更を行わない．つまり“統合の有無に関わらず，最終的に取得するスナップショットの要素になるのは最初にマーカを受信した際に保存した局所状態である”と言える．そこで本アルゴリズムでは衝突発生時にスナップショットグループの統合処理は行わず，衝突相手のグループの記憶のみを行う．以降で，アルゴリズムを構成する 2 つのステップを説明し，その後ステップ 2 のメッセージ複雑度を改善する手法を紹介する．

3.1 ステップ 1

ステップ 1 では，各 initiator が CSS アルゴリズムと同様にマーカを伝播させることでスナップショットグループを決定する．このとき，衝突が発生した場合は CSS アルゴリズムのように統合を行わず，衝突相手の initiator の ID の記憶のみ行う．また，本来衝突が起きたノード間の依存関係によりマーカを送るべきであったノードは全て衝突相手のスナップショットグループで局所状態が保存されるので，以降そのアルゴリズム中ではスナップショットグループに含めない．

3.2 ステップ 2

ステップ 2 では，ステップ 1 で記憶しておいた衝突関係のあるグループの initiator 同士でメッセージを交換することで，互いにステップ 1 が終了していることを確認した後，アルゴリズムを終了する．ステップ 2 を開始した initiator はまず，ステップ 1 で記憶していた，自身が直接衝突したグループの各 initiator へ，自身の ID を載せた終了確認メッセージを送信する．initiator はこのメッセージの broadcast，convergecast により全連結 initiator の終了を確認する．すなわちこの時送信した

メッセージが各 initiator から返ってきた時，衝突関係により構成される initiator 間のネットワークに含まれる全ての initiator は正常にスナップショットグループを決定している．自身のものではない ID が載ったメッセージを受信した場合，その送信元以外の隣接 initiator へメッセージを伝播し，それら全てから同メッセージが返ってくると，最初の送信元へメッセージを返す．ネットワーク内のどれか 1 つでも initiator が全 initiator の終了を確認すると，その initiator は最後の終了許可メッセージを broadcast する．これを受信した initiator は，メッセージを伝播するとアルゴリズムを終了する．

3.3 ステップ 2 の改良

前述のステップ 2 では，全ての initiator が broadcast と convergecast を試みるが，実際はどれか 1 つの initiator だけが行えばよいため，無駄なメッセージが多く発生していた．この点を改良し，メッセージ複雑度を小さくしたアルゴリズムを以降で紹介する．

新たなステップ 2 では，initiator 群の中から代表 initiator（ID が最も小さい initiator）を 1 つ選出する．この代表選出プロセスのために，各 initiator は自身の知る最小 ID を保存する変数 *MinID* を持つ（初期値は自身の ID）．ステップ 2 を開始した initiator はまず，ステップ 1 で記録した衝突先 initiator へ，自身の ID id_i を載せた終了確認メッセージを送信する．このメッセージを受け取った initiator は，メッセージに添付された ID id_i と自身の *MinID* を比較し，$id_i > MinID$ ならば，メッセージの送信元へ，最小の ID を知らせるメッセージに *MinID* を載せて送信する．$id_i < MinID$ ならば，*MinID* の ID を保持する initiator へ，最小の ID が更新されたことを知らせるメッセージに id_i を載せて送信する．更にこのメッセージを受信した initiator も，メッセージに添付された ID と自身の *MinID* を比較し，代表となる initiator を決定，最小の ID が更新されれば，それをメッセージで伝える．各 initator はこのような処理を繰り返す．このプロセスにより，代表 initiator は自身が代表であることを知っている状態が保持される．このようにして代表となる initiator が決定すると，衝突関係にある全ての initiator へアルゴリズム終了を許可するメッセージを伝播させ，受信した initiator の管理するスナップショットグループはアルゴリズムを終了する．

このアルゴリズムは衝突毎の統合を行わないため，CSS アルゴリズムに比べて，衝突回数の増加によるメッセージ送信回数の増加が抑えられることが期待できる．

4 おわりに

本稿では CSS アルゴリズムをメッセージ送信回数の観点から見たその問題点を述べ，それを改善するアルゴリズムの概要を示した．アルゴリズムのステップ 2 は従来のものから改良がなされた．アルゴリズムの正当性は証明済みであるため，今後はシミュレーション実験を行い，それにより実際にメッセージ送信回数が削減されていることを示そうと考える．

参考文献

[1] K.Chandy and L.Lamport, “ Distributed snap-shots : Determining global states of distributed systems,”ACM Trans. Computer Systems, Vol.3, No.1, pp.63-75, 1985.

[2] S.Moriya and T.Araragi,“Dynamic Snapshot Algorithm and Partial Rollback Algorithm for Internet Agents, ” Proceeding of the 15th International Symposium on Distributed Computing, Brief Announcements, pp.23-28, 2001.

[3] Y.Kim, T.Araragi, J.Nakamura and T.Masuzawa, “ A Concurrent Partial Snapshot Algorithm for Large-scale and Dynamic Distributed Systems, ”IEICE Transactions on Information and Systems, Vol.E97-D, No.1, Jan.2014.

セッション３

システム・サービス

コンテナ型仮想化を利用した 学内向け Web ホスティングサービスの更新

中村 純哉[*1]

豊橋技術科学大学 情報メディア基盤センター[*2]

概要

豊橋技術科学大学 情報メディア基盤センターでは、学内の系・研究室・事務局に対してホスティングサービスの提供を行ってきた。ホスティングサービスで提供されるサービスは、Web サイト・権威 DNS サーバ・メーリングリストの 3 種である。情報メディア基盤センターがサービスを管理するため利用者はソフトウェアの脆弱性やセキュリティアップデートを意識する必要が無く、気軽に利用することができることから、現在 110 以上の学内組織が利用している。

ホスティングサービスの提供を開始してから 7 年以上が経過し、運用について幾つかの問題が見えてきた。第 1 に、サービスを提供するサーバの老朽化である。サーバ機器は既に EOL（End of Life）を迎え、保守契約も切れており、いつ壊れてもおかしくない状況である。ディスク故障や原因不明の応答遅延なども発生しており、早急な更新が必要である。第 2 に、サービスの基盤となっている OS の老朽化である。ホスティングサービスは Linux VServer と呼ばれるコンテナ技術で実現されている。この機能は標準の Linux Distribution ではサポートされていないため、利用するためには Linux kernel のソースコードにパッチをあててビルドし直す必要がある。このことが OS バージョンアップの負担を大きなものとしている。第 3 に、各組織ごとの設定情報の散逸である。ホスティングサービスで提供されているサービスの仕様は全組織で共通ではなく、各組織の要望に応じて利用可能なソフトウェアや設定が一部異なっている。しかしこれらの設定情報の違いは情報メディア基盤センター内でも厳密には管理されていないため、運用負荷を上昇させている。

前述の問題を解決するため、情報メディア基盤センターでは学内向けホスティングサービスの更新作業を進めている。新しいホスティングサービスでは、サービス基盤として Linux におけるコンテナ型仮想化の代表的な技術である Docker を用いる。Docker は多くの Linux Distribution で標準的に利用可能であり、今後の継続性も期待できることから採用した。各組織の設定情報については、Dockerfile という Docker のコンテナイメージ生成方法を記述したファイルで管理する。単一ファイルで一元的に管理することにより、設定情報の散逸を防ぐ。本発表では、新しいホスティングサービスの構築方針について述べ、テスト移行の状況について報告する。

[*1] junya@imc.tut.ac.jp

[*2] 愛知県豊橋市天伯町雲雀ヶ丘 1–1

産業連関ネットワーク解析のための疎化処理と閾値の関係について

九州大学　　土中哲秀
九州大学　　小野廣隆

1　はじめに

近年, 産業連関ネットワークに関する分析が盛んに行われている．産業連関ネットワークは, 産業間の取引関係のデータである産業連関表を隣接行列とみなした, 点を産業, 辺を産業間の取引, 辺重みを取引量としたネットワーク（グラフ）である．産業連関ネットワーク分析の中で, 産業間の重要な部分構造を見つけるクラスタ分析はグラフ最適化問題とも親和性が高い. Kagawa らは, 正規化カット問題を応用したクラスタ分析を行い, 日本の自動車サプライチェーンにおいて, 4 つの CO_2 排出クラスタを抽出した [6]. この他にも様々な産業連関ネットワークからの各種定義に基づくクラスタ抽出・それに基づく政策提言を行う研究がなされている.

このように, 産業連関ネットワーク分析において, 重要な部分構造（組合せ構造）抽出が必要とされる場面が多く存在する．しかし, そのような部分構造抽出問題は多くの場合, 計算論的な難問であることが多く (NP 困難), 産業数が数百程度のネットワークでも素朴なアルゴリズムでは解析が不可能となる. 一方, グラフアルゴリズム論の発展により, 一般には NP 困難である問題であっても, 何らかのパラメータ値が小さいグラフを対象とするならば, 高速なアルゴリズムの設計が可能となる場合がある. 例えば, 木幅と呼ばれるグラフパラメータが小さなグラフでは, 多くの組合せ最適化問題を高速に解くことができることが知られている．このようなとき, 木幅に関する固定パラメータアルゴリズムが存在するという．例えば, 後述する最小 k-分割問題は, 木幅に関する $O(2^{\omega} \cdot n^3)$-時間固定パラメータ容易アルゴリズムが存在する [8]．ただし, n は頂点数, ω はグラフの木幅である．本研究ではグラフの木幅の上限値と下限値を求めることにより, 木幅に関する高性能アルゴリズムの適用可能性について考える．一般に産業連関分析ネットワークは木幅が大きく, そのままでは木幅に関する固定パラメータアルゴリズムは適用できない．そこで, 本研究ではネットワークの辺重みに閾値を導入し, 閾値未満の重みの辺を削除することにより, 対象となるグラフの木幅を小さくすることを考える（グラフ疎化）．この際の閾値の選択によっては分析対象となる構造まで壊しかねない. このようなグラフ疎化に対する影響を調べるために, モジュラリティと呼ばれるクラスタリング指標を用いて, グラフ疎化による構造の変化についても考察する.

2　産業連関ネットワーク

産業連関ネットワークは, 産業連関表から作成されたネットワーク（グラフ）である．産業連関表とは産業間の取引関係を表した正方行列であり, 各要素は産業間の取引量を表す．これを重み付き隣接行列と見立てて構成したグラフが産業連関ネットワークである.

このような生成法から, 産業連関ネットワークはいくつかの特徴的な性質を持つ. 第一に, 地理的, あるいは経済圏の影響を反映した部分的に密なグラフ構造を持つ. 例えば, 日本国内の県別に分けた産業を

対象とした産業連関ネットワークを考えると, (1) 同県内, あるいは近県間の産業間に重みの大きい辺が張られやすい, (2) 遠い県の関連の薄い産業間にはそもそも辺が張られない (取引量 0), 等の特徴がある. 同様の特徴は, 国際産業連関ネットワークにおける, 国別の産業の関係等でも見られる. このことは, 産業連関ネットワークの対象とする産業の分類にも依存するが, 大雑把には産業連関ネットワークは, 極めて密度が大きいいくつかの部分を全体としては比較的疎な形で結んだようなグラフとなっていることを示唆するものである. 2005 年の日本産業連関表 (397 部門) からなるグラフの例では, グラフ密度が約 0.468 (潜在的な辺のうち, 46.8%が実際に辺となっている) と密であり, また大きさ 93 のクリークを持つ.

第 2 に, 辺重みの分布がロングテール的, すなわち重みの大きな辺は少数であり, 重みの小さい辺が大部分を占めている. 2005 年の日本産業連関表 (397 部門) における辺重みの分布は図 2 のようになる. すなわち, 上述のように 2005 年の日本産業連関表の表すネットワークは密ではあるが, その辺の多くは比較的小規模の取引を表しており, 重要と考えられる大きな規模の取引を表す辺はごく少数であることを示唆している. このような辺重み分布のグラフに対して, グラフ分割問題によるクラスタリングを考えるとき, 閾値によるグラフ疎化は有効な手段であると考えられる.

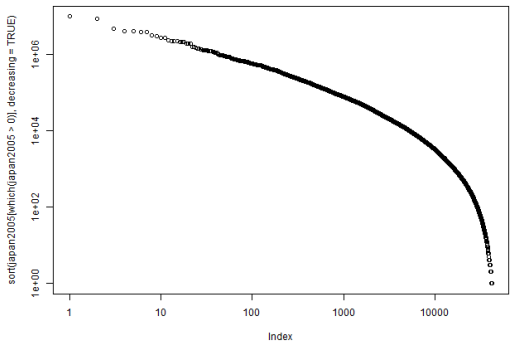


図 1: 2005 年日本産業連関表 (397 部門) の辺重み分布 (ログスケール)

3 グラフ分割問題

本節では, 産業連関ネットワークに対するクラスタ分析への応用が期待される正規化カット問題と最小 k-分割問題について述べる. はじめに, 正規化カット問題は以下のように定義される.

定義 3.1 ([9])

入力: $G = (V, E)$, 辺重み関数 $w : E \to Q^+$, 自然数 K.

出力: $\sum_{k=1}^{K} cut(C_k) / \sum_{u \in C_k, v \in V} w_{uv}$ を最小にする K 個の点部分集合 $C_1, \ldots, C_k \subseteq V$. ただし, $C_k \cap C_l = \emptyset$ $(\forall k, l \in \{1, 2, \ldots, K\}, k \neq l)$ かつ $\bigcup_{k=1}^{K} C_k = V$.

ここで w_{uv} は, $e = (u, v) \in E$ のとき, $w_{uv} = w_e$, そうでないとき, $w_{uv} = 0$ とする. また, $cut(C) = \sum_{u \in C, v \notin C} w_{uv}$ である. この問題は, 一般のカット問題と異なり, 出力されるクラスタ, すなわち K 個の点部分集合から得られる誘導部分グラフの辺の合計値でカット値を割る. これによって, クラスタの内側は密になり, 外側は疎になる. 正規化カット問題は NP-困難であるが, グラフスペクトル理論と k-平均法を用いて, 近似解を求める方法がある.

一方, 最小 k-分割問題は以下のように定義される.

定義 3.2 ([5])

入力: $G = (V, E)$, 自然数 K.

出力: $\sum_{k=1}^{K} cut(C_k)$ を最小にする K 個の点部分集合 $C_1, \ldots, C_k \subseteq V$. ただし, $||C_k| - |C_l|| \leq 1, C_k \cap C_l = \emptyset$ $(\forall k, l \in \{1, 2, \ldots, K\}, k \neq l)$ かつ $\bigcup_{k=1}^{K} C_k = V$.

すなわち, カット値が最小であり, かつそれぞれの点部分集合の大きさの差が 1 以下である K 個の点部分集合を見つける問題である. 最小 k-分割問題は NP-困難である. また $k = 2$ のとき, 特に最小 2 分割問

題と呼ばれる．最小 2 分割問題も, 同様に NP-困難である．しかし, 木幅に関する $O(2^{\omega} \cdot n^3)$-時間固定パラメータ容易アルゴリズムが存在する [5].

Kagawa らは正規化カット問題を応用することによりクラスタ分析を行っているが, 解が k - 平均法の初期値に影響を受けやすく, 計算結果が不安定であるという問題があった [6]. また, 応用先である産業クラスタの発見は, クラスタ間の産業の協力により CO_2 の削減や生産の強化などを行うときの効果の上昇を目指すものである．したがって, クラスタ内の産業の数がなるべく近くなるように産業を分割することが求められる．以上から, 産業連関ネットワークに対するクラスタ分析に対して, 最小 k-分割問題とそれに対するアルゴリズムの応用が期待される．次節では, 最小 k-分割問題に対する木幅に関するアルゴリズムの適用可能性を調べるために, 木幅の上限値と下限値を求める．

4 閾値と木幅

本節では, 木幅の上限値と下限値を求める．初めに木幅の性質を述べる．

木幅とは, 木分解することにより得られるグラフパラメータである．まず, 木分解の定義を与える．

定義 4.1 (木分解 [8]) *グラフ $G = (V, E)$, ノード集合 $I = \{1, 2, \ldots, N\}$, ノードのバッグ集合 $\mathcal{X} = \{X_1, X_2, \ldots, X_N \subseteq V\}$, , 木 T に対して, 以下の条件を満たす $\langle \{X_i | i \in I\}, T \rangle$ を木分解 (Tree Decomposition) という．*

1. $\bigcup_{i \in I} X_i = V$,

2. $\forall \{u, v\} \in E$ に対して, $\exists i \in I : \{u, v\} \subseteq X_i$,

3. $\forall i, j, k \in I$ に対して, T において i と k のパスの間に j が存在するとき, $X_i \cap X_k \subseteq X_j$.

このとき, 木幅は $\omega := \max_{i \in I} |X_i| - 1$ と定義される．木幅を求めることは, NP-困難であるが, $O(n^{\omega+2})$-時間アルゴリズムなどが知られている [1].

また, $G = (V, E)$ に対して, 木幅を ω, 最大クリークの大きさを s とすると以下の性質を満たす．

性質 4.1 ([3])

1. $\omega \geq s$,

2. G が弦グラフならば, $\omega = s$.

したがって, グラフの最大クリークの大きさとグラフを三角化して得られる弦グラフの最大クリークの大きさを求めることにより, 木幅の上限値と下限値を求めることができる．

実際に求めた最大クリークの大きさを図 2, 3 に示す．図 2 は 2005 年日本産業連関表 (397 部門), 図 3 は WIOD2011(40 カ国 35 部門) のデータを使用している．図の縦軸は最大クリークの大きさ, 横軸は辺重みに関する閾値を表わす．右にいくほど閾値は高くなり, グラフは疎になっている．また, 黒の点が元のグラフ, 赤の点が三角化して得られるグラフを表わす．したがって, 赤と黒の点の間に木幅が存在する．

本研究では, R のグラフパッケージ igraph と gRbase を用いて検証した [10, 4]. 図が示すように, はじめはグラフの木幅は大きいが, グラフを疎化することにより, 急激に減少していくことがわかる．

5 閾値とモジュラリティ

前節では, グラフを疎化していくときの木幅の大きさの上限値と下限値を変化を調べた．本節では, グラフ疎化によって, 重要な部分構造が失われていないかをモジュラリティと呼ばれる指標を用いて考察する．

モジュラリティとはグラフのクラスタリングの良さを表わす指標である．重み付きグラフ G において, K 個のクラスタが与えらたとき, モジュラリティは以下のように定義される．

定義 5.1 ([7])

$$Q(K) = \sum_{k=1}^{K} \left\{ \frac{\sum_{u,v \in C_k} w_{uv}}{\sum_{u,v \in V} w_{uv}} - \left(\frac{\sum_{u \in C_k, v \in V} w_{uv}}{\sum_{u,v \in V} w_{uv}} \right)^2 \right\}$$

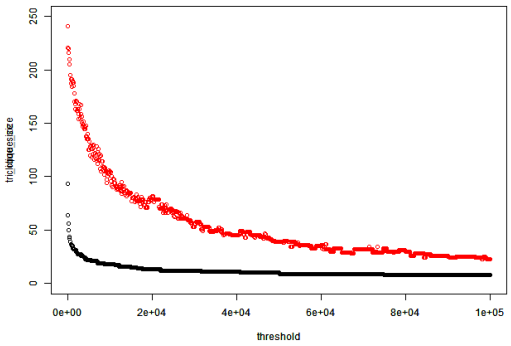


図 2: 2005 年日本産業連関表 (397 部門) の閾値と最大クリークの変化

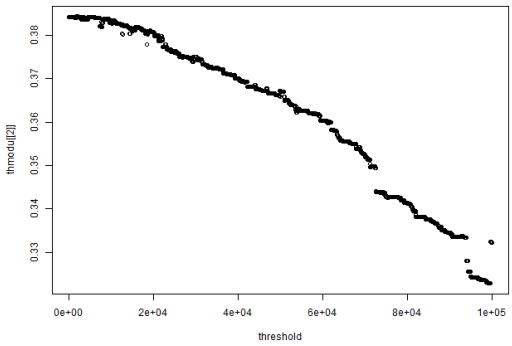


図 4: 2005 年日本産業連関表 (397 部門) の閾値とモジュラリティの変化

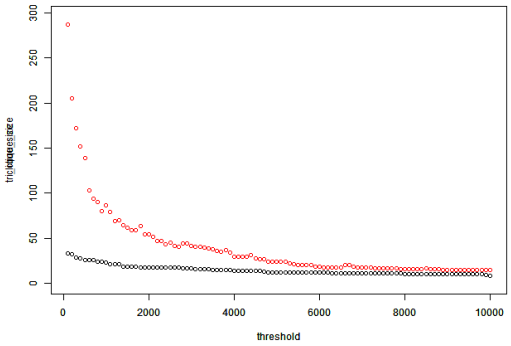


図 3: WIOD2011(40 カ国 35 部門) との閾値と最大クリークの変化

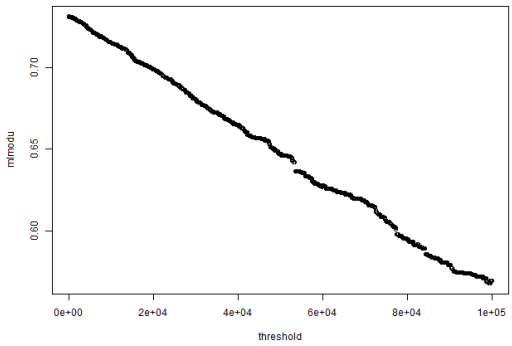


図 5: WIOD2011(40 カ国 35 部門) との閾値とモジュラリティの変化

我々は, 閾値で辺を切ったあとの疎化されたグラフにグラフクラスタリングアルゴリズムを適用し, 得られたクラスタを再び疎化前のグラフに当てはめることにより, 疎化による影響について分析した. 本研究では, グラフクラスタリングアルゴリズムとして, 計算時間が速い Blondel らのアルゴリズムを使用した [2].

2005 年日本産業連関表と WIOD2011 に対する結果をそれぞれ図 4, 5 に示す. 図の縦軸はぞれぞれの閾値で切ったときのモジュラリティ, 横軸は閾値である. 最大クリークの大きさと比較すると, モジュラリティはあまり減少しない. また, 疎化したときに生じる孤立点を適切にクラスタに割り当てれば, さらに上昇することが期待される.

6 まとめと今後の課題

本研究では, 木幅の上限値と下限値を分析することにより, グラフ分割問題に対する木幅に関する固定パラメータ容易アルゴリズムの適用可能性について考察した. そのままのグラフでは木幅が大きいが, グラフを疎化することで加速度的に減少することがわかった. 一方で, グラフの構造の変化を見るためにモジュラリティを指標とした分析も行った. その結

果, モジュラリティは木幅に比べて, 減少しにくいことがわかった. すなわち, 重要な部分構造が残ったまま疎化されているといえる. 以上のことから, 産業連関ネットワークに対するグラフの疎化処理は有効であることが予想される. 今後は, 木幅に関するヒューリスティックアルゴリズムを用いてより木幅に近い近似値を求め, さらに実際にグラフ分割問題を解くことによって, グラフ疎化の解に対する影響を分析することによって, 産業連関ネットワークに対するグラフ疎化がより有効であることを示したい.

参考文献

[1] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. “ Complexity of finding embeddings in a k-tree”, *SIAM Journal on Algebraic and Discrete Methods, Volume 8*, 1987, pp. 277-284

[2] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte and Etienne Lefebvre. “ Fast unfolding of communities in large networks”, *J. Stat. Mech. (2008) P10008* , 2008

[3] Hans L. Bodlaender and Arie M.C.A,“ Kosterm Treewidth computations I. Upper bounds”, *Information and Computation, Volume 208, Issue 3*, 2010, Pages 259-275

[4] Claus Dethlefsen and Søren Højsgaard, “ A Common Platform for Graphical Models in R: The gRbase Package”, *Journal of Statistical Software Vol. 14, No. 17*, 2005

[5] Klaus Jansen, Marek Karpinski, Andrzej Lingas, and Eike Seidel, “Polynomial Time Approximation Schemes for MAX-BISECTION on Planar and Geometric Graphs”, *STACS 2001 Volume 2010, LNCS*, 2001, pp 365-375

[6] Shigemi Kagawa, Sangwon Suh, Yasushi Kondo and Keisuke Nansai, “Identifying environmentally important supply chain clusters in the automobile industry”, *Economic Systems Research, Volume 25, Issue 3*, 2013, pp. 265-286

[7] M. E. J. Newman and M. Girvan, “Finding and evaluating community structure in networks”, *Physical Review E 69 026113*, 2004

[8] Rolf Niedermeier, “Invitation to Fixed-Parameter Algorithms”, *Oxford University Press*, (2006)

[9] Jianbo Shi and Jitendra Malik, “Normalized Cuts and Image Segmentation”, *IEEE Transactions on Pattern Analysis and Machine Intelligence archive Volume 22 Issue 8*, 2000 , pp. 888-905

[10] http://igraph.org/

時間特徴ベクトルの生成手法と観光地推薦システムへの応用

房　冠深†　　亀井　清華‡　　藤田　聡‡

†‡広島大学工学研究科 〒739-8527 東広島市鏡山１丁目４－１

E-mail:　†bou@se.hiroshima-u.ac.jp,　‡{s-kamei, fujita}@ se.hiroshima-u.ac.jp

あらまし　現在、ユーザの日常のニーズを満たし、アイテムを推薦するために、情報推薦システムが広く使われている。本文は情報推薦システムにおけるアイテムの時間的な特徴の抽出または抽象に注目し、自動的な時間特徴ベクトルの生成手法を提案する。主なアイディアは：1)　Wikipedia を用いてアイテムに関連するコーパスを定義する;　2) Twitter を用いてアイテムに関連するトレンドを定義する;3)　アイテムのトレンドに含まれた時間の特徴をハイライトする。提案手法の有効性を検証するため、旅行推薦システムを構築した。実験の結果は以下の結論に導いた：1)　収集した 6057 個の観光地の中、提案手法は約 9%に対して優れた機能を果した;2)　この 9%の観光地の時間特徴ベクトルはベクトル空間の中の分布は互いの類似度に関連している;　3)　時間特徴ベクトルのベクトル空間での変化は季節的な観光地推薦に用いられる。

キーワード　観光推薦システム，季節特徴ベクトル，Wikipedia，Twitter

1. 研究背景

現在、推薦システムは各分野で広く使用されている[1][2]。本文は推薦システムにおけるアイテムが有する時間とともに変化する時間特徴に注目する。その特徴は常に季節、天気などのコンテクスト要素と関連している。例えば季節または期間限定のメニューを提供するレストラン、天気に敏感する高速道路、季節のアトラクションを備えている観光地の特徴が分析の対象となる。複数の分析の方法が既存研究に提案されたが、人工的にコーパスの用意、ドメインに制限されることや様々の欠点が存在する。

本文は時間の特徴に対し、通用性の高い自動的な時間特徴ベクトルの生成手法を提案する。主要な考えは以下の三点にある:　1)　Wikipedia を用いてアイテムに関連するコーパスを定義する;　2) Twitter を用いてアイテムに関連するトレンドを定義する;3)　アイテムのトレンドに含まれた時間の特徴をハイライトする。Wikipedia は 480000 ページ以上の規模のソーシャルメディアとし、ステップ 1 に相応しいデータソースだと考える。また推薦システムの研究で広く使用されているデータソースとした Twitter は近年、モバイルコミュニケーションと共に迅速に発展した[3]。

提案手法を評価するため、本文では提案手法に基づいた観光地推薦システムも提案し、構築された。提案システムではユーザのプロファイルに加え、予定の旅行時期も考慮して推薦結果を決定する。収集された 6057 個の観光地情報は全日本に覆い、Wikipedia において詳しい紹介文が設けられている。観光地に関連するトレンド情報は Twitter Streaming API を用いて抽出された。

2. 提案手法

2.1. 時間特徴ベクトルの生成

先ず、時間軸を分割し、時間の特徴が変化する時間スライスを定義する。与えられたスライスにおいてアイテムの特徴は不変と仮定する。目標はスライス毎に、時間特徴ベクトル(TFV)を生成する。TFV は与えられたスライスの特徴を表し、基本特徴ベクトル(BFV)から拡張してくる。BFV はベクトルとして TF-IDF で計算され、時間の要素に関わらない視点でアイテムの特徴を表現する:

$$\vec{v}_{\mathrm{i}}^{\mathrm{b}} = \left\{ \left(w_j, TF_{i,w_j} \right) \times IDF_{w_j} \right) \middle| w_j \in W_i \} \qquad (1)$$

その中、与えられたアイテムはo_iとし、W_iはアイテムの紹介文d_iに含まれた単語の集合とする。全アイテムの紹介文の集合をD =∪ d_i と定義する。

TFV を生成するために、BFV の式(1)の TF 部分を拡張する。与えられたスライスs_kに発表された tweet の集合をt_kとし、W_iに含まれた単語w_jの Twitter における TF は以下のように:

$$TF'_{k,w_j} = \frac{n'_{k,w_j}}{\sum_{w \in W} n'_{k,w}} \qquad (2)$$

中に、n'_{k,w_j}は単語w_jの頻度とし、Wはすべてのアイテムに関連する単語の集合とする。即ち、tweet の中のアイテムの紹介文Dに含まれていない単語を除外する。変数αでTF_{i,w_j}とTF'_{k,w_j}を調整することで、BFV の$\vec{v}_{\mathrm{i}}^{\mathrm{b}}$を TFV に拡張する:

$$\vec{v}_{\mathrm{i,k}}^{\mathrm{t}} = \left\{ \left(w_j, \left((1-\alpha) TF_{i,w_j} + \alpha TF'_{k,w_j} \right) \times IDF_{w_j} \right) \middle| w_j \in W_i \right\} \quad (3)$$

2.2. 旅行推薦システム

この節は提案手法の有効性を証明するために、提案された季節のアトラクションに対応できる観光地推薦システムを説明する。提案システムは観光地の季節的な特徴(e.g.,花見、紅葉またはイベント)を考慮し、全日本の観光地を推薦対象とする。

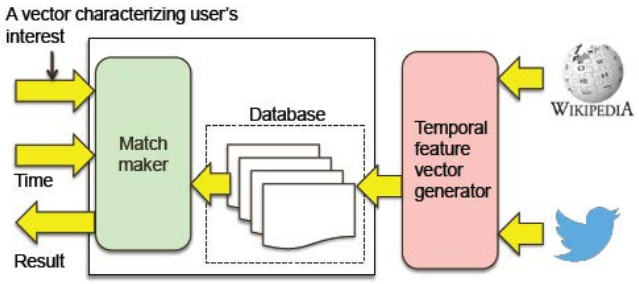


Figure 1:提案されたシステムの構造

Figure 1:提案されたシステムの構造は提案された推薦システムを構造を表す。システムは予め Wikipedia のカテゴリ:"日本の観光地"に属する観光地の紹介文を収集し、Mecab で形態素解析を行う。一方、収集された観光地の名称を用いて Twitter Streaming API で 2013.9 から 2014.3 まで約 50 万の関連 tweet を抽出した。実装の段階で、時間スライスを一ヶ月間と定義した。i.e. 一年間を季節に 12 個分割した。式(3)のTF_{i,w_j}とTF'_{k,w_j}を 1 対 1 の比率にするため、αの値を 0.995 に修正した。

3. 実験と評価

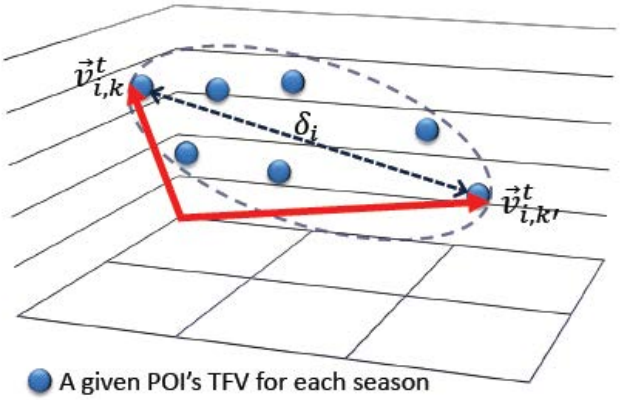
提案手法の有効性を提案された観光地推薦システムで評価する。このセクションでは先ず、各季節の TFV の多様性について評価する。次に、提案システムに対して、TFV の変化の視点からシステムの性能を測る。

3.1. TFV の多様性

全観光地の TFV が属したベクトル空間をΩと定義し、Ωの次元ごとに単語と関連する。故に観光地の季節s_kの TFV である$\vec{v}^{t}_{i,k}$を空間Ωでの点とマッピングする。従って与えられた観光地o_iの直径を定義する：

$$\delta_i = max_{k \neq k'}\{\|\vec{v}^{t}_{i,k} - \vec{v}^{t}_{i,k'}\|\}$$

$\|\cdot\|$はL^2norm とする。以下の図のように表す：



TFV の多様性の評価するために、収集された観光地

直径の大きさ	観光地の数
[0, 0.005)	4531
[0.005, 0.01)	892
[0.01, 0.015)	424
[0.015, 0.02)	86
[0.02, 0.025)	41
[0.025, 0.03)	18
[0.03, ∞)	1

Table 1 各々の観光地の直径

各々の直径を計算する。テーブル Table 1 各々の観光地の直径は観光地の直径を表す。その中、75%の観光地は小さい直径を得ており、9%の直径は 0.01 より大きい。即ち、提案手法は特にこの 9%の観光地に有効であり、生成された TFV は十分季節の特徴を表現している。故に 0.01 を閾値として収集された観光地を active と inactive に分ける。

3.2. TFV の時間的な変化

この節では提案手法がベースとして提案システムでの性能を評価する。即ち、与えられたユーザのプロファイルと旅行の予定時期に対し、ベクトル空間Ωで TFV が最もユーザプロファイルに類似度の高い観光地を観察する。現在ユーザのプロファイル構築は未実装なので、仮に或る観光地の TFV をプロファイルとしてシミュレートする。

実験の結果を表示するため、Active 観光地から仁和寺を選び、ユーザのプロファイルとし、他の観光地の TFV とのコサイン類似度を計算する。類似度が高い上位三個をテーブル Table 2 仁和寺の類似度が高い観光地に表している。

11 月の TFV	2 月の TFV
下呂温泉合掌村	姫路城
大覚寺	八戸公園
再び公園	高遠城

Table 2 仁和寺の類似度が高い観光地

4. まとめ

本文は通用性が高い自動的な時間特徴ベクトルの生成手法を提案した。その上、提案手法を評価するため、季節のアトラクションに対応する観光地推薦システムを提案した。実験の結果は、提案手法は正しく観光地の時間的な特徴を抽象したことを証明した。

文　献

[1] J. Hong, W.-S. Hwang, J.-H. Kim, and S.-W. Kim,, "Context-aware music recommendation in mobile smart devices," Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014, pp. 1463–1468..

[2] X. W. Zhao, Y. Guo, Y. He, H. Jiang, Y. Wu, and X. Li, "We know what you want to buy: A demographic-based system for product recommendation on microblogs,"Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2014,pp.1935-1944.

[3] K. Oku, K. Ueno, and F. Hattori, "Mapping geotagged tweets to tourist spots for recommender systems," Proceedings of the IIAI 3rd International Conference on Advanced Applied Informatics, 2014, pp. 178-794. .

手描き情報共有システムの開発

広島大学大学院工学研究科情報工学専攻　分散システム学研究室
山中 景太*，亀井 清華，藤田 聡

概要

近年，複数のユーザが共有ファイルを同時編集することのできる共同編集システムが広く用いられ始めている．現在，共同編集する対象として主なものはドキュメントファイルであり，手描き情報を共有するような共同編集システムはあまり開発されていない．また，主な共同編集システムはサーバ・クライアント型の通信によって行われており，アクセスの集中によるサーバの過負荷，サーバ停止に伴うシステムの停止などといった問題点が存在する．そこで本研究では，手描き情報を共有可能で，かつサーバ・クライアント型での問題点を解決するピア・ツー・ピア（P2P）型手描き情報共有システムの開発を目的とする．P2P 型ファイル共有システム上ではレプリカ（複製）をネットワーク上に分散配置することが多く，レプリカを持つ全てのピアに対し迅速に更新を反映させる方法(整合性維持手法)が重要であり，その方法を述べる．システムにおいてはキャンバスの仕組み，描いたものの操作など，現時点での実装状況を述べる．

１　はじめに

近年，Google Docs[1] や Office Web Apps[2] などの共同編集システムが広く活用されるようになってきている．インターネット環境さえあれば，いつ，どこからでも保存されているドキュメントにアクセスし，複数のユーザでの同時編集が可能である．共同編集する対象としての主なものはドキュメントファイルであり，ペイントソフトで描く様な手描き情報を共有するシステムはあまり開発されていない．また，一般的にこれらの共同編集システムはサーバ・クライアント型の通信によって行われており，大量の同時アクセスによるサーバへの負荷の集中や，サーバが何らかの原因により停止した場合におけるシステムの停止，などといった問題点が存在する．サーバ・クライアント型に対しピア・ツー・ピア（P2P）型の通信の場合，特定のノードへの負荷の集中を緩和，離脱耐性の性能向上などサーバ・クライアント型の問題点を解決することができる．以上のことから，本研究では P2P 型の通信を行う手描き情報共有システムの開発を目的とする．

２　整合性維持手法

一般に P2P 型ファイル共有システムでは，負荷分散，離脱耐性，クエリヒット率などの性能向上を図るため，各共有ファイルのレプリカ（複製）をネットワーク上に分散配置することが多い．従って，更新情報をネットワークに参加している全てのピアに対し迅速に反映させるための整合性維持手法を組み込む必要がある．

これまでに P2P 型ファイル共有システムを対象とする整合性維持手法がいくつか考えられてきたが，本研究では伝播遅延・メッセージ数・離脱耐性において高い性能を示した中島らの整合性維持手法[3]をシステムに組み込む．中島らの手法では，レプリカを保持するノードからなる静的な木構造

を共有ファイルごとに構築する手法を提案している．更新情報は木構造に沿って伝えられるため，更新情報の伝播を効率的かつ確実に行うことに成功している．

３　システム実装

現在開発中のシステムは，JAVA 言語の GUI ツールキット Swing を用いて開発を行っている．機能についてはペンの色，太さの変更，意思疎通のための簡単なチャット機能などを実装している．また，描いたものは１つのオブジェクトとして配置され，オブジェクトごとに移動，削除などの操作が行えるようになっている．

キャンバスはポップアップメニューを表示するキャンバス，リアルタイムで描かれている手描き曲線を表示するキャンバス，描かれた曲線のオブジェクトを配置するキャンバスの３層からなる．現在描かれているものに対し上書きを行った場合，下にある情報は消えることなく下に残るような仕組みになっている．

今後実装する機能としては，描いたものの保存，オブジェクトの所有権を考慮しての操作制限などを考えている．

４　おわりに

本研究では，中島らの整合性維持手法を組み込んだ P2P 型の手描き情報共有システムを開発している．中島らの整合性維持手法は他の手法と比べ，更新情報の伝播やネットワーク構造の維持に必要なメッセージ数が少なく，更新伝播遅延や，離脱耐性の面においても優れている．そのため P2P システムを開発する上で非常に適した整合性維持手法だといえる．現時点では手描き情報の共有は実現できている．

今後の課題として再編集可能なファイルとしての保存，オブジェクトの所有権を考慮しての操作制限など様々な機能の追加が挙げられる．また，開発したシステム上での遅延時間などの計測や，

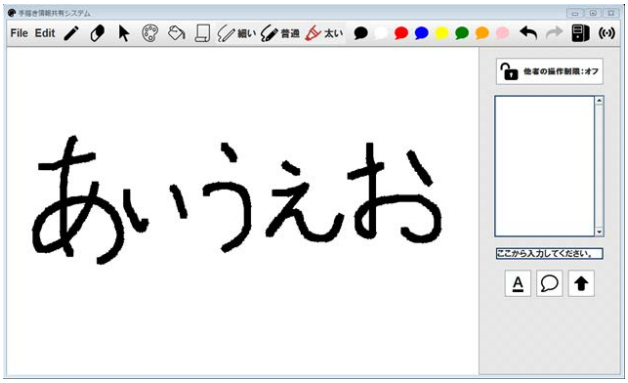


図１：開発中システムの外観

最適な木構造を構築するためのパラメータの設定なども考える必要がある．

参考文献

[1] Google Docs, https://docs.google.com/.

[2] Office Web Apps, http://www.microsoft.com/ja-jp/office/2010/webapps/default.aspx.

[3] Taishi Nakashima and Satoshi Fujita, ``Tree-Based Consistency Maintenance Scheme for Peer-To-Peer File Sharing of Editable Contents,'' IEICE Trans. on Information and Systems, vol. E97-D, no.12, pp. 3033-3040, December 2014.

セッション４

自己安定

Self-Oscillations in Population Protocols

Anissa Lamani
Department of Informatics, Kyushu University, Japan
Joint work with Masafumi Yamashita and Yukiko Yamauchi

The PP model was introduced by Angluin at al. to model passive distributed systems, in which a collection of finite-state agents interact with each other in order to accomplish a common task. The agents are assumed to be identical and uniform i.e., they are not identified and all execute the same protocol. The computations are performed through pairwise interactions i.e., when two agents interact, they exchange their local information and update their state according to a common protocol. The interaction pattern is unpredictable i.e., the agents have no control on which agent they will interact with.

Many investigations have considered the PP model, different problems have been addressed as computing a function, electing a leader, counting, coloring and naming [1, 2, 3, 4, 6, 8]. Most of these problems are concerned with the convergence to a specific configuration that contains the answer to the problem that is considered. These problems are hence said to be static. Only few investigations have considered dynamic problems such as the self-stabilizing token circulation problem on rings [4], the self-stabilizing mutual exclusion and the group mutual exclusion problems [5] and recently the self-stabilizing oscillation problem [7].

About the self-stabilizing oscillation problem, it has been shown that under a deterministic scheduler, the self-stabilizing leader election (SS-LE) and the self-stabilizing oscillation problem (SS-OSC) are equivalent [7], in the sense that an SS-OSC protocol is constructible from a given SS-LE protocol and vice versa, which unfortunately implies that (1) resorting to a leader is inevitable (although we seek a decentralized solution) and (2) n states are necessary to create an oscillation of amplitude n, where n is the number of agents (although we seek a memory-efficient solution).

Under the probabilistic scheduler, we investigate both the self-stabilizing synchronization problem and the self-stabilizing oscillation problem. We then aim at designing a PP that represent any given periodic function f by

a non empty set of populations that exhibit an oscillatory behavior.

References

[1] Dana Angluin, James Aspnes, Melody Chan, Michael J. Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In *DCOSS*, volume 3560, pages 63–74, 2005.

[2] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC*, pages 290–299, 2004.

[3] Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

[4] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. *TAAS*, 3(4), 2008.

[5] Joffroy Beauquier and Janna Burman. Self-stabilizing mutual exclusion and group mutual exclusion for population protocols with covering. In *OPODIS*, volume 7109, pages 235–250, 2011.

[6] Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory Comput. Syst.*, 50(3):433–445, 2012.

[7] Colin Cooper, Anissa Lamani, Giovanni Vigiletta, Masafumi Yamashita, and Yukiko Yamauchi. Constructing self-stabilizing oscillators in population protocols. In *Stabilization, Safety, and Security of Distributed Systems, (SSS)*, volume 9212, pages 187–200, 2015.

[8] Keigo Kinpara, Tomoko Izumi, Taisuke Izumi, and Koichi Wada. Improving space complexity of self-stabilizing counting on mobile sensor networks. In *OPODIS*, volume 6490, pages 504–515, 2010.

Proof-Labeling スキームに基づく故障封じ込め自己安定アルゴリズムの提案

中川遼† 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

†† 奈良先端科学技術大学院大学 情報科学研究科

概要　自己安定アルゴリズムとは，任意のネットワーク状況から実行を開始しても，やがて解を求めて安定するアルゴリズムである．自己安定アルゴリズムでは，安定した状況において突発的な変化が起こっても，それを検知し自律的に安定した状態に収束することが可能である．また，故障封じ込めとは，安定した状況から少数のプロセスが故障した場合に，再び安定するまでに状態遷移するプロセスや必要とする時間を限定したものである．故障封じ込めは，自己安定アルゴリズムにおいて一般に発生しやすいと考えられている小規模な故障による故障状態からの実行において，故障の及ぶ範囲を制限しかつ素早く再安定することを目的としている．本稿では，自己安定アルゴリズムの設計において，故障の検知と安定化を分けて記述することのできる Proof-Labeling スキームを用いて，故障封じ込めの性質を付与した自己安定アルゴリズムの設計方法を提案する．

1　はじめに

分散システムは，複数のノードが相互に通信リンクで接続されたネットワークである．自己安定アルゴリズムとは，任意の初期状況から実行を始めても問題を解く（問題の要求を満たす状態に到達する）ことができる分散アルゴリズムである [1]．自己安定アルゴリズムはその性質から，ノードにおける変数の破壊などの一時的な故障（一時故障）によって問題の要求を満たす状況から任意の状況に陥っても，再び問題の要求を満たす状況に復帰して安定する．つまり任意の一時故障に対して耐故障性を持つ分散アルゴリズムであり，長期間分散システムを安定に保ち，一時故障に柔軟に対応する必要がある分散システムを動作させるのに適しており，現在盛んに研究されている分野の 1 つである．自己安定アルゴリズムは一般に，問題の要求を満たしていないことを検出する検知フェーズと，問題の要求を満たす状態に遷移する収束フェーズに分けることができる．Proof-Labeling スキームはこの考えを定式化したものであり [2]，自己安定アルゴリズムの設計を検知と収束の 2 つに分けて設計することが可能になり，自己安定アルゴリズムの設計を容易化することができる．

分散システムにおいて同時に多数のノードが故障することはまれであり，一般には少数のノードが一時故障を起こし，要求を満たす状況からわずかに変化した状況になることがほとんどである．従来の自己安定アルゴリズムではそのようなわずかな変化を考慮していないため，少数ノードの故障の影響がネットワーク全体に及ぶことがある．そこで，故障封じ込めと呼ばれる単一ノードの故障から定数時間で再安定することのできる性質を付与することは実際のシステムにおいて非常に重要である [3]．

様々な問題に対する自己安定アルゴリズムが設計されてきた．また，その自己安定アルゴリズムに故障封じ込めの性質を付与する変換手法も提案されている．それに加えて近年，自己安定アルゴリズムの設計方法として，一般に従来の設計よりも簡単と言われる Proof-Labeling スキームを用いた設計が注目を浴びている．本稿では Proof-Labeling スキームを用いて設計された自己安定アルゴリズムに，故障封じ込めの性質を付与したアルゴリズムを提案する．

2　諸定義

2.1　ネットワークモデル

本稿では，n 個のノードが通信リンクで接続されたネットワーク N を扱う．各ノードは共有レジスタを持ち，自身の共有レジスタに Read/Write を実行可能であり，隣接ノードの共有レジスタに対して Read が可能である．1 つのアトミックな動作において，各ノードは Read;Compute;Write を行うことができる．また，本稿では各ノードが少なくとも 2 つの隣接ノードを持つネットワークのみを対象とする．

2.2　分散システムの状況

故障封じ込め自己安定アルゴリズムにおいて，ネットワークは 3 つの状況に大別することが可能である．

1. Consistent
   全てのノードが目的の述語を満たしている状況．
2. 1-Faulty
   Consistent な状況から，ただ 1 つのノードの状態だけを変化させて得られる状況．状態が変化したノードを故障ノードと呼ぶ．アルゴリズムによっては，1-故障状況において，故障ノードを特定できない場合もあるが，本稿で構成する故障封じ込めアルゴリズムでは，1-故障状況で故障ノードを特定可能である．
3. Inconsistent
   Consistent でも 1-Faulty でもない状況．

2.3 自己安定性

任意の状況から実行を開始しても，いずれは正当な状況（Consistent 状況）に到達して安定する性質を自己安定性と呼び，自己安定性を持つアルゴリズムのことを自己安定アルゴリズムと呼ぶ．自己安定アルゴリズムは突発的な変化により Consistent 状況でなくなるとそれを検知し，自律的に収束する．

2.4 故障封じ込め

自己安定アルゴリズムにおいて，任意の 1-Faulty 状態から O(1) 時間で Consistent 状態に到達するならば，その自己安定アルゴリズムは故障封じ込めであると呼ぶ．

2.5 Proof-Labeling スキーム

Proof-Labeling スキーム（以下，P-L）は，自己安定アルゴリズムの設計手法の 1 つであり，ラベリングアルゴリズムとデコーダーアルゴリズムのペアに分けて自己安定アルゴリズムを構成する．以下ではオリジナルのものを紹介する．以下の P-L には故障封じ込めの性質は備わっていない．

2.5.1 ラベリングアルゴリズム

各ノードに次の条件を満たすラベルを割り当てる．

1. ネットワーク状況が与えられた述語を満たすならば，各ノードで実行するデコーダーアルゴリズムが Consistent 状況を出力するようなラベルを出力する．
2. そうでなければ，少なくとも 1 つのノードでのデコーダーアルゴリズムが Inconsistent を出力するようなラベルを出力する

2.5.2 デコーダーアルゴリズム

各ノードは自身に割り当てられたラベルと隣接ノードのラベルからネットワークの状況が Consistent か Inconsistent かを出力する．

3 提案アルゴリズムの概要

本章では P-L を拡張して，故障封じ込め自己安定アルゴリズムを構成するための枠組みを新たに提案する．提案アルゴリズムでは，従来の P-L と同様にラベリングアルゴリズムとデコーダーアルゴリズムのペアを使用して分散システムの状況が Consistent でないことを検知するが，Consistent でないときは 1-Faulty か Inconsistent かも検知可能とすることで，1 つのノードの故障からの修復を可能とする．これを実現するために，本手法ではラベリングアルゴリズムにおいて，故障封じ込めでない自己安定アルゴリズムで自身に割り当てられるラベルのコピーを隣接するノードの内の 2 つのノードにもとして割り当てたものを，新たなラベルとする．

3.1 ラベリングアルゴリズム

従来の P-L で用いられていたラベル割り当ての条件 1,2 に加えて以下の条件を追加する．

- ネットワークが 1-Faulty 状況ならば，故障しているノードで実行されるデコーダーアルゴリズムは 1-Faulty を出力

ネットワークが 1-Faulty 状況であることを判断する 1 つの方法として，本手法では隣接ノードの持つ Consistent 状況を出力するラベルのコピーと自身の現在の状態を比較して故障を検知し，かつ隣接ノードが全て Consistent 状況であるとき 1-Faulty と認識する．

3.2 デコーダーアルゴリズム

各ノードは割り当てられたラベルと隣接ノードのラベルからネットワークの状況が Consistent なのか Inconsistent なのか 1-Faulty なのかを出力する．

3.3 修復・収束アルゴリズム

各ノードはデコーダーアルゴリズムでの自身の出力により，以下の動作を行う．

1. 出力が Consistent のとき
   自身は故障していないので何もしない．
2. 出力が 1-Faulty のとき
   隣接ノードが持つ自身のラベルのコピーを元に，自身の状態を修復する．
3. 出力が Inconsistent かつ隣接ノードの出力に 1-Faulty がないとき
   元の自己安定アルゴリズムに従った動作を行う．

4 おわりに

本稿では，P-L に基づいて設計された自己安定アルゴリズムに故障封じ込めの性質を付与した自己安定アルゴリズムの P-L に基づく設計方法を提案した．提案手法により，様々な問題に対する故障封じ込め自己安定アルゴリズムを系統的に実現することが可能とする．

本提案アルゴリズムでは，各ノードが隣接 2 ノードにラベルのコピーを置く性質上，最悪時には 1 つのノードで最大次数+1 のラベルを保持しなければならない．したがって最悪時のレジスタ空間複雑度が大きくなる．

この問題を解決するために，ネットワークトポロジに何らかの制限をかけることにより最悪時のレジスタ空間複雑度を小さくすることが今後の課題である．

参考文献

[1] Dolev, S.: Self-Stabilization. MIT Press (2000)

[2] Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Distributed Computing 22(4), 215-233 (2010)

[3] Ghosh, S., Gupta, A., Pemmaraju, S.V.: Fault-containing network protocols. In: Proceedings of the 1997 ACM Symposium on Applied Computing, pp. 431-437. ACM (1997).

不安定な仮想グリッド環境下における zigzag プロトコル適用手法とシミュレーション結果

團孝直人[1], 大下福仁[2], 角川裕次[1], 増澤利光[1]

1 大阪大学大学院情報科学研究科 増澤研究室

2 奈良先端科学技術大学院大学

概要　本稿では，仮想グリッドにおいて，仮想ノードを用いた自己最適化ルーティングの提案を行う．仮想グリッドとは，無線通信媒体 (ノード) が存在する領域を仮想的にグリッド状に分割したものであり，仮想グリッドを利用した自己最適化経路構成プロトコル zigzag を我々のグループで提案している．このプロトコルでは，グリッド状に分割された各領域（セル）にノードが存在することを前提とし，各セルから 1 つずつ選ばれた中継ノードで構成されるグリッド・ネットワーク上で動作する．しかし，ノードが移動端末の場合，ノードの移動によって，ノードが存在しないセル（空白セルとよぶ）が発生することがある．この問題を解決するために，本稿では仮想ノードを導入する．仮想ノードと，空白セルに対して，他のセル内のノードが空白セルのノードとして振る舞うものである．本稿では，プロトコル zigzag に仮想ノードを導入し，その効果をシミュレーションで評価する．

1　はじめに

近年，無線デバイス（ノード）の普及により，MANET（モバイルアドホックネットワーク）や WSN（ワイヤレスセンサネットワーク）などの無線ネットワークは一般的かつ重要となっている．無線ネットワークでは，ノードは二次元平面に配置されており，通信範囲内のノードとのみ直接通信可能である．宛先ノードが通信範囲外のときは，他のノードが宛先ノードへのメッセージを中継する．効率的なマルチホップルーティングを実現するために，数々のルーティングプロトコルが提案されている [1]．既存のルーティングプロトコルとして，仮想グリッドネットワーク上で最短経路を構築するアルゴリズム“ zigzag ”[2] が提案されている．

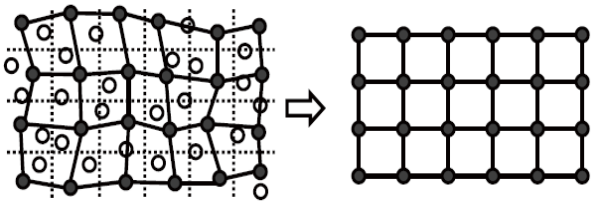


図 1: 仮想グリッドネットワーク

仮想グリッドネットワークは無線ネットワークをグリッド状に並ぶ同じ大きさの正方形の領域（セル）に仮想的に分割して構築する．各セルの大きさは，隣接するセル内の全ノードが互いに通信可能であるように決定する．各セルにおいてあるノードがルータとして選出され，グリッド･ネットワーク上の接続されたルータを使ってノード間の通信を実現する．以降，仮想グリッドネットワークにおけるルータ間通信について議論する．

仮想グリッド上で任意の経路が与えられたときに，その経路を最短経路に変更する自己最適化プロトコルとして，zigzag を我々のグループが提案している．プロトコル zigzag は各セル内に少なくとも一つのノードがあることを前提としている．しかし，ノードが移動可能な場合，ノードの移動によってノードが存在しないセル（空白セルとよぶ）が発生する可能性があるが，空白セルが発生するとプロトコル zigzag は正しく動作しない．そこで本稿では，空白セルのノードとして，仮想ノードをを導入し，その効果をシミュレーションで評価する．仮想ノードとは，空白セルの隣接セルから選ばれたノードであり，空白セル内のノードとして振る舞うものである．本稿の内容は以下の通りである．まず，zigzag プロトコルの簡単な概要を紹介し，仮想ノードの実現方法を述べ，仮想ノードの効果のシミュレーションによる評価結果を示す．そして最後に，今後の研究方針について述べる．

2　Zigzag

本節では zigzag プロトコル [2] の概要を示す．始点ノードと終点ノードをルータが知っていれば最短経路を構成することは容易である．しかし, 始点ノードや終点ノードが移動する場合，移動のたびにその情報を各ルータに伝えて最短経路を再構成するという手法は，移動のたびに大域的な通信を必要とするため効率的でない．そこで, 各ルータは経路に関する自身の局所情報のみを保持するものとする．つまり，そのルータが経路上に位置する場合，始点ノードおよび終点ノードへの経路につながるそれぞれのポートのみを知っているものとする．そして, 定数ホップ内のルータが保持する情報, つまり局所的な情報のみをもとに自律的に局所的な経路の組み替え (組替処理) を行う．この組替処理を繰り返すことでいずれ最短経路に収束することは示されている [2]．組替処理を図 2 に示す．

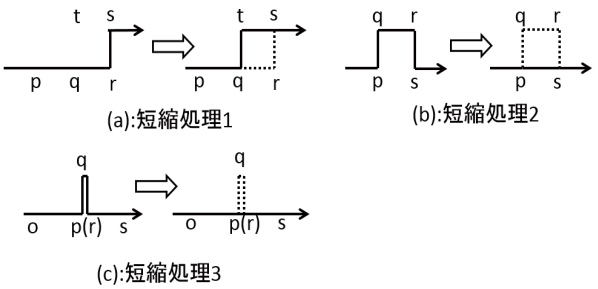


図 2: 組替処理

3　仮想ノードの実現方法

本節では仮想ノードの実現方法について考察する．ノードが移動可能な場合，経路上のセルが空白セルになる場合が考えられる．ただし本稿では，空白セルが隣接しない場合を仮定したアルゴリズムを提案する．起こりうるのは以下の 2 つのパターンである．簡単のため，a, b, c,.. を各セルの名前とする．

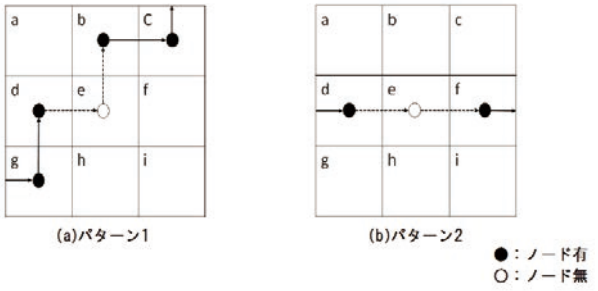


図 3: 空白セル e の発生状況

3.1　図 3(a):パターン 1

d → e → b のように，空白セル e で経路が曲がっている場合である．あるセル内のノードは上下左右のセルとは必ず通信できるので，d と b 内のノードと通信可能なセル a から仮想ノードを選択するのが望ましい．ただし,, a と e どちらのセル内にもノードが存在しない場合は，経路は構成できず不完全のままである．

3.2　図 3(b):パターン 2

d → e → f のように，空白セル e が直線部にある場合である．この場合は，d と f 内のノードと通信可能なセルは e のみである．したがって，この場合は経路を d → a → b → c → f もしくは d → g → h → i → f に変更する．なお，斜めセルとは通信ができないので，d → b → f や d → h → f のような変更が可能な場合もあるが，セル b や h 内のノードは，セル d と f のノードと通信可能とは限らないので，ここではそのような変更はしないものとする．

4　シミュレーション

zigzag プロトコルの仮想グリッド環境への適用法を評価するために，シミュレータを実装した．シミュレータの実装には Java を用いた．本節では，シミュレータについての概要の説明，実験結果とそれについての考察を行う．

4.1　概要

今回は動的に変化する仮想グリッドネットワークにおいて，ランダムに形成された経路に zigzag プロトコルを適用させるためのシミュレータを作成した．シミュレーションでは，zigzag プロトコルが最短経路を導出できるかを評価する．基盤として，縦横 1000 × 1000（ピクセル）のフィールドとランダムに配置されたノードを用意し，そこに仮想グリッドを適用してシミュレーションを行う．ノードがフィールド外に進行しようとした場合は，反射をしてフィールド外には出ないようにしている．ノードは乱数を用いて縦横 4 方向と斜め 4 方向の周囲 8 方向を移動する．乱数は Random クラスを用いて実現している．そのノードからランダムに二つのノードを選択し，それぞれをホーム，ターゲットとする．ホームからターゲットまでのランダムな経路を作成し，その経路に対して zigzag を適用する．経路の作成は，ホームから乱数を用いて 1 セルずつ進み，ターゲットまでたどり着いたら経路完成とする．仮想グリッドにおいて，ノードの存在しないセルが存在した場合に仮想ノードを用いてそのセルを補う．フィールド全体に対してのノードの存在しないセルの存在率は，仮想グリッドのセルの数とノード数に依存する．そのため今回の実験では，以下のパラメータを変更できるように実装した．

- ノード数
- グリッドサイズ

動的に変化するフィールドにおいて，ランダムに形成された経路に対して，zigzag プロトコルによって最短経路まで収束するかどうかを実験する．最短経路に収束することを成功，最短経路への収束過程で，仮想ノードで空白セルを補えなくなった場合，あるいは，空白セルが経路上で連続する場合を失敗とする．成功，失敗いずれかが判定できた時点で，次の経路を作成し実行を繰り返す．シミュレーションではさまざまなノード数とグリッドサイズに対して，以下を評価する．

- zigzag プロトコルの成功・失敗数
- 全体のセルに対する，ノードの存在しないセルの存在比率

4.2　実験結果と考察

実験は 3 種類のグリッドサイズ 10×10, 20×20, 30×30 で行う．ただし，30×30 の実験結果は，前二つと同様の結果であるため，ここでは割愛する．また仮想ノードを使用した場合と，しなかった場合での成功率および全体のセルに対するノードのないセルの存在率を評価する．またここでは，ノードが存在しないセルの割合を空きセル比と表記する．

4.2.1　グリッドサイズ 10 × 10 の場合

図 4 にグリッドサイズ 10 × 10 の場合の実験結果を示す．仮想ノードを補った効果が，3 つの規模の中で一番顕著に現れた．これはサイズが小さい分，経路長が他の 2 つと比べて全体的に短くなるからである．仮想ノードを用いなかった場合は，空きセル比が 5%をきるまでほとんど成功しなかったが仮想ノードを用いた場合は，空きセル比が 10%をきるあたりで成功率が上昇した．空きセル比が 15%以上だとほとんど成功しないことがわかる．

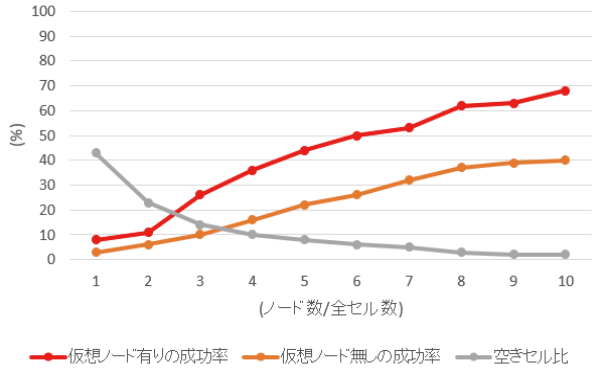


図 4: グリッドサイズ 10 × 10 の実験結果

4.2.2　グリッドサイズ 20 × 20 の場合

図 5 にグリッドサイズ 20 × 20 の場合の実験結果を示す．このサイズにでは経路長が長くなるので成功率が下がることがわかる．仮想ノードを用いなかった場合は，空きセルに関係なくほとんど成功することはなかった．仮想ノードを用いた場合は，空きセル比が 5%以下になると成功率が上昇した．

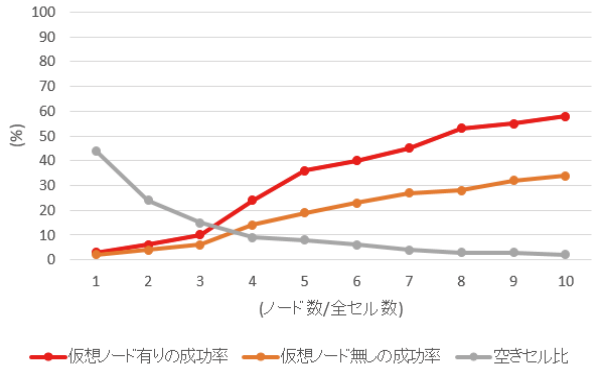


図 5: グリッドサイズ 20 × 20 の実験結果

5　おわりに

本稿では，仮想グリッドネットワーク上の自己最適化最短経路プロトコル zigzag のモバイル環境への適用法を提案した．また，提案手法を評価するために，シミュレータを実装し，パラメータを変化させ評価実験を行った．実験から，ノードの存在しないセルが 5 %を超えると最適経路へと収束することが著しく困難になることがわかった．仮想ノードを実現する方法をより細かく設定すれば zigzag プロトコルはモバイル環境へさらに適用することができる．そこで，今後の課題として，より効果的な仮想ノードの実現方法の考案を行いたい．

参考文献

[1] W. Dargie and C. Poellabauer: ”Fundamentals of wireless sensor networks:theory and practice”，John Wiley Sons,Ltd, 2010.

[2] Zigzag: ”local-information-based self-optimizing routing in virtual grid networks”，Shusuke Takatsu, Fukuhito Ooshita, Hirotsugu Kakugawa Toshimitsu Masuzawa,ICDCS, page 357-368. IEEE Computer Society, 2013

個体群プロトコルモデルにおける緩自己安定リーダ選挙のシミュレーション評価

清洲星顕[1] 首藤裕一[1] 大下福仁[2] 角川裕次[1] 増澤利光[1]
1 大阪大学大学院情報科学研究科
2 奈良先端科学技術大学院大学情報科学研究科

概要

個体群プロトコル (population protocol) は、資源の非常に制約された小型センサで構成されるモバイルセンサネットワーク上の計算を表現する抽象モデルである．センサが「個体」に、センサネットワークが「個体群」に対応する．個体は一種の有限状態機械としてモデル化される．自己安定の閉包性を緩めた緩自己安定プロトコルを提案している．このモデルにおけるリーダ選挙問題を解くプロトコルが首藤らによって提案されている．首藤らは交流する個体の組が一様ランダムに選ばれるという仮定のもとで、漸近的な評価を行った．本稿では、このプロトコルを 2 次元平面で、2 つの移動モデルを用いて実験的に評価する．

1 はじめに

分散システムは，それを構成する計算機の数の膨大さから，システムにおいて局所的な故障が頻繁に発生し得る．ゆえに，分散システムに故障耐性を持たせることは重要である．分散システムに高い故障耐性を持たせる技法のひとつに，自己安定 [3] という概念がある．自己安定の概念は次のように表現できる．(i) 任意の初期状況から実行を開始しても，システムはやがて安全状況と呼ばれる状況に達する（収束性） (ii) 一度システムが安全状況に達すると，それ以降，システムは永遠に所望の性質を満たし続ける（閉包性）．自己安定なシステムは，どのような一時故障（記憶内容の喪失など）が発生したとしても，やがてその故障から自律的に回復するという優れた故障耐性を有する．しかし，多くの場合，その実現には高いコスト（時間計算量・空間計算量など）を要する．また，いくつかの問題に対しては，自己安定性の実現自体が不可能である．
首藤らは，自己安定の閉包性を緩めた，**緩自己安定**という概念を提唱した [2]．直感的には，緩自己安定の概念は次のように表すことができる． (i) 任意の初期状況から実行を開始しても，システムは比較的短時間のうちに**安全状況**と呼ばれる状況に到達する（**収束性**） (ii) 一度システムが安全状況に到達すると，それ以降，システムは非常に長いあいだ所望の性質を満たし続ける（**緩閉包性**）．緩自己安定プロトコルは , 要件 (ii) における，システムが所望の性質を維持する時間が十分に長いのであれば（例えばネットワークのサイズに関して指数時間であるなど），応用上，自己安定プロトコルと同等の有用性を持つとみなせる．

個体群プロトコル (population protocol)[1] は、資源の非常に制約された小型センサで構成されるモバイルセンサネットワーク上の計算を表現する抽象モデルである．センサが「個体」に、センサネットワークが「個体群」に対応する．個体は一種の有限状態機械としてモデル化される．各個体は他の個体と互いに交流することによって自らの状態を変更し、それらの状態遷移が全体のネットワーク、すなわち個体群の計算を進める．

緩自己安定の有用性を示すため, 首藤らは, 自己安定プロトコルが設計不可能な問題に対して緩自己安定プロトコルが存在することを示した [2]．具体的には, 自己安定プロトコルの不在が証明されている個体群プロトコルモデルにおけるリーダ選挙問題を解く緩自己安定プロトコル P_{LE} を提案した．このプロトコルは、個体数 n の上界 N が既知であるという仮定のもと, 完全グラフ上でリーダ選挙問題を解く緩自己安定プロトコルである．任意の初期状況から実行を開始しても $O(nN \log n)$ 期待ステップのあいだに安全状況に収束し, それ以降, $\Omega(Ne^N)$ 期待ステップのあいだ唯一のリーダを保持する．

しかしながら, [2] における収束時間, 維持時間の漸近的評価は, 個体間の交流が一様ランダムに発生することを仮定している．具体的には, 各時刻で交流する個体ペアは以前の交流とは独立に全個体ペアのなかから等確率に選ばれることを仮定している．個体群プロトコルモデルの現実的な応用を想定する場合, この仮定は必ずしも成り立たない．たとえば, 実際の応用では各時刻の交流の独立性は担保されない場合がある．ある個体ペア u, v 間で交流が発生した場合, その時点で u, v は近接している (交流可能な距離に存在する) ため, しばらくのあいだ u, v 間の交流は他の個体ペアの交流よりも発生しやすいと考えるのが自然である．

そこで本稿では, より現実的な交流のモデルのもとで, 上述のリーダ選挙プロトコル P_{LE} の収束時間・維持時間をシミュレーションにより評価する．具体的には, ランダムウォーク、ランダムウェイポイントの 2 つの移動モデルに従って各個体が 2 次元平面上を移動した場合に, P_{LE} が一様ランダムな交流モデルで実行した場合と同様の収束時間・維持時間を示すかどうかをシミュレーションで評価する．

2 諸定義

2.1 個体群プロトコルモデル

個体群とは、**個体**と呼ばれる有限個のセンサノードで構成されるシステムである．各個体は常にひとつの状態を持ち、交流の発生によってその状態を変化させる．個体群は, 単純有向グラフ $G = (V, E)$ で表す．$V = \{0, 1, ..., n-1\}$ $(n \geq 2)$ は個体群を構成する個体の集合であり, $E \subseteq V \times V$ は発生し得る交流

の場合を表す. $(u,v) \in E$ のとき, 個体 u と個体 v はそれぞれ呼びかけ側, 応答側として交流し得る. なお, 本稿では G が完全グラフであると仮定する。すなわち、$E = V \times V$ である.

プロトコル $P(Q,Y,O,\delta)$ は, 状態集合 Q と出力記号集合 Y, 出力関数 $O : Q \to Y$, 状態遷移関数 $\delta : Q \times Q \to Q \times Q$ で構成される. Q と Y は有限集合である. O は各個体の出力を決定する. 個体 v が状態 $q \in Q$ を持つとき, v の出力は $O(q)$ である. δ は, ある個体ペア間で交流が発生したときの, 両個体の交流後の状態を決定する. 仮に, 状態 p,q,p',q' が $(p',q') = \delta(p,q)$ を満たすとする. このとき, 状態 p を持つ個体 u と状態 q を持つ個体 v がそれぞれ呼びかけ側, 応答側として交流を行えば, 両個体の交流後の状態はそれぞれ p',q' となる.

状況とは, 個体群中の全個体の状態を特定する写像 $C : V \to Q$ である. 任意の状況 C について, 合成関数 $O \circ C : V \to Y$ を C の出力と呼び, $O(C)$ で表す. 状況 C,D と個体 u,v が以下の条件を満たすとき, C は交流 (u,v) によって D に**遷移する**といい, $C \xrightarrow{(u,v)} D$ で表す.

$$(D(u),D(v)) = \delta(C(u),C(v))$$
$$\forall w \in V \setminus \{u,v\}, D(w) = C(w)$$

以降, 任意のプロトコル P について, 個体群の取りうるすべての状況の集合を $C_{all}(P)$ で表す.

交流発生系列 $\gamma = (u_0,v_0),(u_1,v_1),...$ は交流の無限長の系列であり, 各時点において発生する交流を決定する. 交流発生系列 $\gamma = (u_0,v_0),(u_1,v_1),...$ と整数 t $(t \leq 0)$ について,u_t,v_t をそれぞれ $\gamma_1(t),\gamma_2(t)$ で表す. また, 交流 $(\gamma_1(t),\gamma_2(t))$ を $\gamma(t)$ で表し, γ における**時点** t **での交流**という.$v \in \gamma(t)$ のとき, 個体 v は γ における時点 t での交流に**参加する**という.

初期状態 C_0 と交流発生系列 γ が与えられたとき, プロトコル P の実行 $\Xi_P(C_0,\gamma)$ は以下のように一意に決まる.

$$\Xi_P(C_0,\gamma) = C_0,C_1,\ldots$$
$$s.t. \quad \forall t\ (t \leq 0),\ C_t \xrightarrow{\gamma(t)} C_{t+1}$$

2.2 リーダ選挙問題

挙動は, 問題の仕様, すなわち, 問題を解くプロトコルが満たすべき所望の性質を定める.

Z を任意の記号集合とする. 任意の個体群 $G(V,E)$ について, V から Z への写像の系列を G 上の**トレース**といい, Z をこのトレースの**アルファベット**と呼ぶ. プロトコル $P(Q,Y,O,\delta)$ について, アルファベットが Q のトレースを P の**状況トレース**と呼ぶ. 有限長もしくは無限長の P の状況トレース $T = C_0,C_1,\ldots$ に対して, $OT_P(T) = O(C_0),O(C_1),\ldots$ を P に関する T の**出力トレース**と呼ぶ. $OT_P(T)$ のアルファベットは Y であることに注意されたい.

有限長のトレース $T = \lambda_0,\lambda_1,\ldots,\lambda_{l-1}$ に対し, その長さ l を $|T|$ で表す. トレース T が無限長のトレースであるとき, $|T| = \infty$ とする. 有限超もしくは無限長のトレース $T = \lambda_0,\lambda_1,\ldots$ に対して, その部分トレース $T_{x,y}$ $(0 \leq x \leq y \leq |T|)$ を次式で定義する.

$$T_{x,y} = \lambda_x,\lambda_{x+1},\ldots,\lambda_y$$

特に, $T_{0,t}$ を T の長さ $t+1$ の接頭辞と呼び $T_{pre}(t)$ で表す.

個体群 G 上の**挙動** $B(Z)$ とは, 同一のアルファベット Z を持つ G 上のトレースの集合であり, Z をその挙動のアルファベットと呼ぶ. 文脈から Z が明らかな場合は, $B(Z)$ を単に B と表記する. **問題**は, その問題に対する正当な出力トレースからなる挙動として定義される. $B(Y)$ を任意の挙動とし, T をプロトコル $P(Q,Y,O,\delta)$ の任意の状況トレースとする. このとき, T は, $OT_P(T) \in B$ であるとき, かつそのときのみ挙動 B で定義される問題に対して正当であるという, 挙動 B 中の任意のトレース T と任意の整数 x,y $(0 \leq x \leq y \leq |T|)$ に対して, $T_{x,y} \in B$ が成り立つとき, B は**標準的**であるという.

定義 1 （リーダ選挙問題）
ある個体 $v_l \in V$ に対して $w(v_l) = L$ を満たし, 他のすべての個体 $v \in V, v \neq v_l$ に対して $w(v) = F$ を満たすようなすべての写像 $w : V \to F,L$ の集合を le で表す. リーダ選挙問題に対応する個体群 $G(V,E)$ 上の挙動 $LE(\{F,L\})$ を次式で定義する.

$$LE = \{T = w,w,\cdots | 1 \leq |T| \leq \infty \wedge w \in le\}$$

LE がプロトコルの実行に対して求める仕様は, その実行における全状況を通して, 記号 L を出力する 1 個のリーダ個体 (全状況を通して固定) と記号 F を出力する $n-1$ 個の非リーダ個体が存在することである. LE はあきらかに標準的な挙動である.

2.3 緩自己安定プロトコル

本節では, 確率的緩自己安定の定義を与える.

プロトコル $P(Q,Y,O,\delta)$ と標準的な挙動 $B(Y)$ を考える. 有限長もしくは無限長の P の状況トレース $T = D_0,D_1,\ldots$ について, ある整数 t $(0 \leq t \leq |T|)$ が存在して $OT_P(T_{pre}(t)) \in B$ かつ $OT_P(T_{pre}(t+1)) \notin B$ であるとき, $T_{pre}(t)$ を P と B に関する T の**維持トレース**と呼び, $HT_P(T,B)$ で表す. そのような t が存在しないときは, $HT_P(T,B$ を以下のように定義する.

$$HT_P(T,B) = \begin{cases} T & (OT_P(T_{pre}(0)) \in B) \\ \varepsilon & (OT_P(T_{pre}(0)) \notin B) \end{cases}$$

ここで ε は, 空状況トレースを表す. $|\varepsilon| = 0$ とする. 状況 $C_0 \in \mathcal{C}_{\rm all}$ と B に対する P の**期待維持交流回数** $EHT_P(C_0,B)$ を以下のように定義する.

$$EHT_P(C_0,B) = \mathbf{E}[\ |HT_P(\Xi_P(C_0,\Gamma),B)|\ ]$$

$EHT_P(C_0,B)$ は, 直観的には, 初期状況 C_0 からはじまる P の実行が挙動 B で定義される問題に対して正当であり続けられる交流回数の期待値を表す.

$\mathcal{C} \subseteqq \mathcal{C}_{\rm all}$ を状況の集合とする. ある整数 t $(0 \leq t \leq |T|)$ が存在して $\forall i \in \{0,1,\ldots,t\}$ $D_i \notin \mathcal{C}$ かつ $D_{t+1} \in \mathcal{C}$ となるとき, $T_{pre}(t)$ を $\mathcal{C}$ に関する T の**収束トレース**と呼び, $CT_P(T,\mathcal{C})$ で表す. そのような t が存在しないときは, $CT_P(T,\mathcal{C})$ を以下のように定義する.

$$CT_P(T,\mathcal{C}) = \begin{cases} \varepsilon & (OT_P(T_{pre}(0)) \in \mathcal{C}) \\ T & (OT_P(T_{pre}(0)) \notin \mathcal{C}) \end{cases}$$

状況 $C_0 \in \mathcal{C}_{\rm all}$ と $\mathcal{C}$ に対するプロトコル P の**期待到達交流回数** $ECT_P(C_0,\mathcal{C})$ を次式で定義する.

$$ECT_P(C_0,\mathcal{C}) = \mathbf{E}[\ |CT_P(\Xi_P(C_0,\Gamma),\mathcal{C})|\ ]$$

期待到達交流回数 $ECT_P(C_0, \mathcal{C})$ は, 直観的には, 初期状態 C_0 からはじまる P の実行が $\mathcal{C}$ 中の任意の状況に到達するまでに要する交流回数の期待値を表す.

定義 2 （緩自己安定プロトコル）
α と β を任意の正の実数とする. 次式が成立するとき, プロトコル $P(Q, Y, O, \delta)$ は標準的な挙動 $B(Y)$ と状況の集合 $\mathcal{S} \subseteq \mathcal{C}_{\mathrm{all}}$ に対する (α, β)-緩自己安定プロトコルであるという.

$$\max_{C \in \mathcal{C}_{all}(P)} ECT_P(C, S) \leq \alpha,$$

$$\min_{C \in S} EHT_P(C, B) \geq \beta.$$

P の状況 $C \in \mathcal{C}_{\mathrm{all}}(P)$ が $EHT_P(C, B) \geq \beta$ を満たすとき, C を P と B に関する β-安全状況と呼ぶ. 明らかに, 上式の $\mathcal{S}$ は P と B に関する β-安全状況のみからなる状況の集合である.
直観的には, α が比較的小さく (n の低次の多項式など)β が非常に大きい場合 (n の指数関数など), (α, β)-緩自己安定プロトコルは有用である.

3 プロトコル P_{LE}

本節では, 本稿でシミュレーション評価を行うプロトコル P_{LE} を説明する. このプロトコルにおいて, 各個体の状態は, 1 ビットの**リーダビット**と 0 から s までの値をとる**タイマ**によって構成される. すなわち, $Q = \{l, -\} \times \{0, 1, \ldots, s\}$ である. ここで, s は自由に値を設定できる定数である. Q 中の状態 q に対して, 第一項 (リーダビット) を $q.leader$, 第二項 (タイマ) を $q.time$ で表す. 任意の状態 $q \in Q$ に対して, $O(q)$ は次式で定義される.

$$O(q) = \begin{cases} L & (q.leader = l) \\ F & (q.leader \neq l) \end{cases}$$

以降, リーダビットが l である個体をリーダ個体, リーダビットが $-$ である個体を非リーダ個体と呼ぶ.
煩雑になるのを避けるため, 状態遷移関数 δ は関数の形ではなくパターン規則として書き下す (図 1). 交流 (u, v) が発生したとき, u, v の状態の組がある規則の左辺に適応するならば, u, v の状態はその規則の右辺で記される状態に遷移する. u, v の状態の組が複数の規則の左辺に適合するときは規則番号の最も小さな規則を適合する. 適合する規則がひとつも存在しない場合は, u, v ともにその状態を変化させずに交流前の状態を保持する. 各規則において, 記号*はドントケアを表す.

以降で, P_{LE} の動作を直観的に説明する. リーダ個体どうしの交流が発生した場合, 一方の個体がリーダ個体であり続け, 他方の個体が非リーダ個体になる (規則 1). リーダ個体と非リーダ個体のあいだで交流が発生した場合, 両個体のリーダビットは変化しない (規則 2, 規則 3). リーダ個体が参加する交流が発生するとき, 両個体のタイマの値は s に**初期化**される (規則 1, 規則 2, 規則 3). 非リーダ個体がリーダ個体に変化するのは, タイマの値が 0 である非リーダ個体どうしで交流を行うとき, かつそのときのみである (規則 4). この規則 4 が適用される交流が発生することを, 以降ではタイムアウトが発生するという. いずれかのタイマの値が 0 ではないような 2 つの非リーダ個体が交流するとき, 少なくともひとつの個体がタイマの値を 1 減じられる (規則 5). また, 規則 5 には, **大きなタイマの値を伝搬させる**という別の側面がある. すなわち, タイマの値が大きい非リーダ個体とタイマの値が小さい非リーダ個体とが交流を行った場合, 後者のタイマは大きな値 (大きい方の値から 1 減じた値) に設定される.

規則 1	$((l, *), (l, *)) \to ((l, s), (-, s))$
規則 2	$((l, *), (-, *)) \to ((l, s), (-, s))$
規則 3	$((-, *), (l, *)) \to ((-, s), (l, s))$
規則 4	$((-, 0), (-, 0)) \to ((l, s), (-, s))$
規則 5	$((-, i), (-, j)) \to ((-, f), (-, f))$ $(0 \leq i, j \leq f = \max(i, j) - 1)$

図 1 遷移関数 δ

少なくともひとつのリーダ個体を有するような状況においては, 規則 5 による大きなタイマの値の伝搬と, 規則 1 と規則 2, 規則 3 によるタイマの初期化によって全個体のタイマが比較的大きな値に保たれるため, タイムアウトは滅多に発生しない. 他方, リーダ個体をひとつも持たないような状態においては, タイマの初期化が発生しないため, タイムアウトは比較的短時間で発生する. このため, リーダ個体が 2 つ以上存在する状況から実行を開始した場合は規則 1 によるリーダ個体の削減によって, リーダ個体がひとつも存在しない状況から実行を開始した場合は規則 4 によるリーダ個体の選出によって, やがて唯一のリーダ個体は個体群に誕生する. P_{LE} のつくりから, 次の 2 つの性質があきらかに成り立つ.

- ある時点において少なくともひとつのリーダ個体が個体群中に存在するならば, それ以降リーダ個体の数は 0 にならない.
- 一度ただひとつのリーダ個体が誕生すると, 次のタイムアウトが発生するまでのあいだその唯一のリーダ個体は保持される.

定義 3 （安全状況）
状況 C が以下を満たすとき, C を安全状況という.

- ただ一つのリーダが存在する.
- 任意の個体の状態 q は, $q.time \geq 2/s$ を満たす.

定理 1 [2] s が 96 の倍数かつ $s \geq 3n$ であるときプロトコル P_{LE} は $(O(ns \log s), \Omega(se^{s/96}))$-緩自己安定プロトコルである.

n の上界 N が与えられたときに, $s = 96N$ と設定することで本プロトコルは $(O(nN \log N), \Omega(Ne^N))$-緩自己安定プロトコルとなる.

4 シミュレーション評価

この節では本稿の趣旨である 2 つの移動モデルを用いたシュミレーション結果とその評価について述べる. なお, この節で用いる収束時間は初期状況から安全状況に達するまでの交流回数, 維持時間は安全状況に達した後に新たなリーダが誕生するまでの交流回数を表す. また, 初期状況における各個体の状態は, リーダか否かはランダムに設定し（確率 1/2 でリーダ, 確率 1/2 で非リーダ）, タイマの値は上限値 s に固定する.

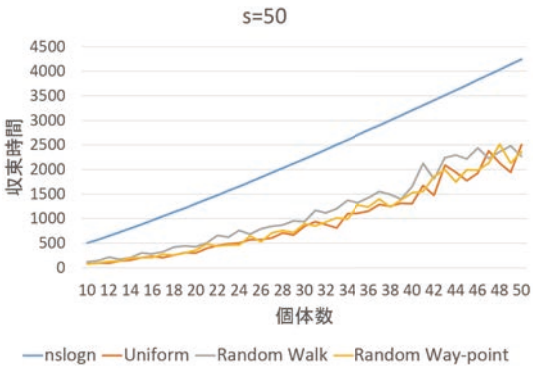


図 2

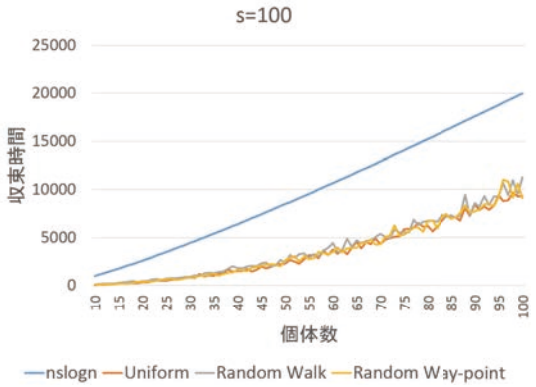


図 3

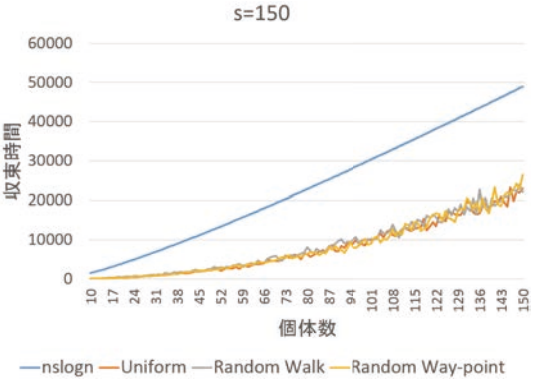


図 4

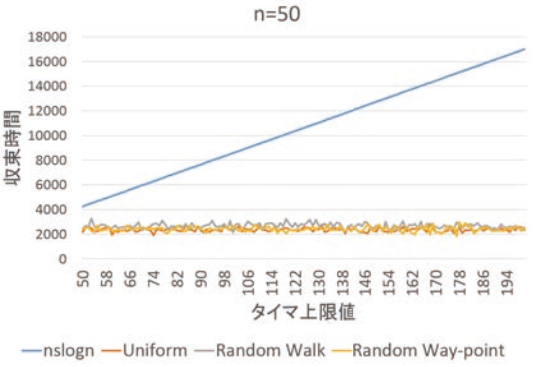


図 5

4.1 移動モデル

- ランダムウォーク
  個体は 1 秒ごとにランダムな方向を選び, 距離 1 だけ進むものを用いる.

- ランダムウェイポイント
  個体は, 目的座標をランダムで選択して目的座標に到達するまでその方向に 1 秒毎に距離 1 進むものを用いる.

4.2 設定

シミュレーションで個体が動くフィールドのサイズは 50×50 の正方形としている. ランダムウォークはフィールド外をトーラスで管理し, ランダムウェイポイントは目的座標をフィールド内に設定することで管理している. 1 秒ごとに個体 u をランダムに選択し, その個体の距離 1 以内の個体の中から個体 v を一様ランダムに選択し, u を呼びかけ側, v を応答側とする交流を発生させる.

4.3 比較

次に示す図 2, 図 3, 図 4 はそれぞれリーダが唯一に決まるタイマの上限値 s を 50, 100, 150 としたときの収束時間の 50 回の試行平均を示したシミュレーション結果である. なお, グラフの横軸は個体数 n $(10 \leq n \leq s)$ である. Uniform は, 交流の発生確率をランダム一様とし, 収束時間の測定を行った結果である. シミュレーション結果の Uniform とランダムウォーク, ランダムウェイポイントの結果を比べると, それぞれに違いが見られないといえる. またシミュレーション結果の増加傾向は, $ns \log n$ のそれと似た結果となっていることがわかる. このことから, $ns \log n$ は, 収束時間の漸近的評価としての妥当性が確認できる.

図 5, 図 6, 図 7 は個体数をそれぞれ 50, 100, 150 としたときの収束時間の 50 回の試行平均を記したシミュレーション結果である.
Uniform と $ns \log n$ を比べると常に Uniform の結果が $ns \log n$ よりも小さくなっていることがわかる. また, Uniform とランダムウォーク, ランダムウェイポイントの結果を比較すると, それぞれに違いが見られないことがわかる. しかし, シミュレーション結果の増加傾向は, $ns \log n$ と似た結果とは言いがたく, 設定したパラメータの範囲においては一定の値となっていることがわかる. なお, グラフの横軸はタイマ上限値 s $(n \leq s \leq 200)$ である.

図 8, 図 9 は個体数をそれぞれ 50, 80 としたときの維持時間のシミュレーション結果である. 横軸はタイマの上限値を示しており, 縦軸は維持時間の底を 10 とした**対数**を示す. 維持時間のシミュレーション結果は, 値が非常に大きくなり, 今回は個体数の変域は比較的狭いものとなっている. データは少ないが, 対数グラフから, Uniform, ランダムウェイポイント, ランダムウォークはタイマの上限が大きくなるにつれて, 全てほぼ指数的に増加していることがわかる. また, それぞれを比較すると, Uniform の継続時間が一番大きく, 次いでランダムウェイポイ

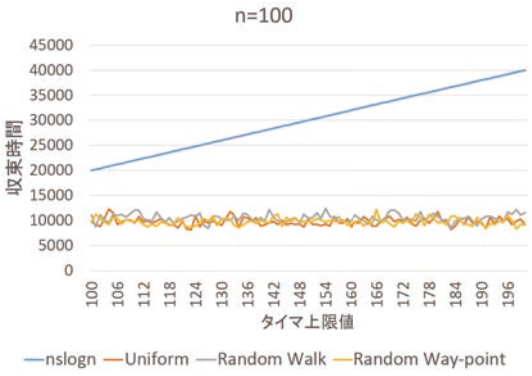


図 6

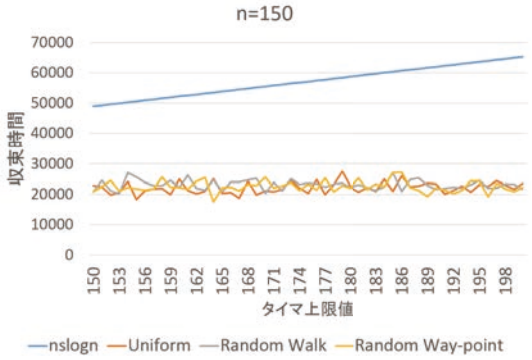


図 7

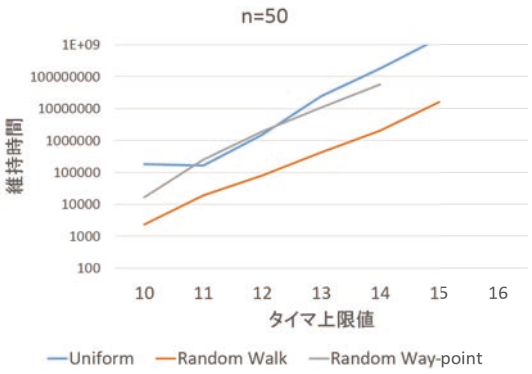


図 8

ント, ランダムウォークの順であった. その原因として, ランダムウォーク, ランダムウェイポイント, Uniform の順にリーダと長期間交流しない確率が高いということが考えられる. 具体的には, ランダムウォークは初期位置の周辺を動き回ることが多いため, リーダから比較的遠い場所に注目すると, 非リーダどうしの交流確率が高くなり, Uniform, ランダムウェイポイントと比べると新たなリーダが誕生しやすくなっていることが考えられる. ランダムウェイポイントも一様ランダムと比較すると, 同様の理由で Uniform よりも新たなリーダが誕生しやすくなっていることが考えられる.

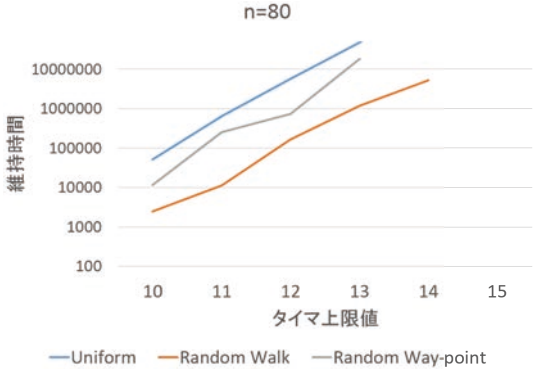


図 9

5 今後の研究

首藤らは完全グラフにおけるプロトコルの提案だけでなく, 任意のグラフにおけるプロトコルも提案している [4]. 今後は, このプロトコルの収束時間・維持時間を, 本稿同様にシミュレーションを用いて実験的に評価する予定である.

参考文献

[1] D. Angluin, J Aspnes, Z Diamadi, M.J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. Distributed Computing, 18(4):235-253, 2006.

[2] Y. Sudo, J. Nakamura, Y. Yamauchi, F. Ooshita, H. Kakugawa, T. Masuzawa. Loosely-stabilizing leader election in a population protocol model. Theoretical Computer Science 444, 100-112, 2012.

[3] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. Communications of the ACM, 17(11):643644, 1974.

[4] Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-Stabilizing Leader Election on Arbitrary Graphs in Population Protocols. OPODIS 2014, LNCS 8878, 339-354, 2014.

セッション５

計算理論１

一般化ジャンケンに対するゲーム理論的解析

九州大学経済学部経済工学科
小松秀平
shuhei@isop.co.jp
九州大学大学院経済学研究院経済工学部門
小野廣隆*
hirotaka@econ.kyushu-u.ac.jp

概要

日本のジャンケンは，グー (石)，チョキ (鋏)，パー (紙) の 3 つの手によるものが一般的であるが，世界には 4 手以上からなるジャンケンが数多く存在する．しかしその中には明らかに他の手と比べ弱く，使うべきでない手（無駄な手）が含まれるものも多々ある．伊藤らは，無駄な手のない 4 手以上のジャンケン（一般化ジャンケン）について有向グラフとしての特徴づけを行った．本研究では，一般化ジャンケンを 2 人ゼロ和ゲームとして定式化し，最適混合戦略の観点から一般化ジャンケンについて考察を行う．伊藤らの定めた無駄な手のないジャンケンであっても， 戦略的には無駄な手が存在し得ること, 戦略的に無駄な手が存在しないジャンケンは必ず奇数手からなることを示す.

1 はじめに

日本のジャンケンは，グー (石)，チョキ (鋏)，パー (紙) の 3 つの手からなり，石は鋏に勝ち，鋏は紙に勝ち，紙は石に勝つ．両者が異なる手を出せば必ず勝負がつき，同じ手を出した場合は引き分けとなる．日本人ならば誰もが知る遊びの 1 つであろう．日本だけでなく，世界中にもジャンケンやその類型は存在する．その中には，日本の 3 手からなるジャンケンと違い，4 手以上からなるものも多々ある．wikipedia によると, フランスで

* corresponding author

は 4 手 (pierre (石), papier (紙), ciseaux (鋏), puit (井戸)) からなるジャンケン, 5 手からなるジャンケン (4 手のジャンケンに Bombe (爆弾) を加えたもの) などが存在する [4]. 伝統的なジャンケン以外にも新たなジャンケンの一般化が考えられており, 例えば David C. Lovelace 氏は, 7 手, 9 手, 11 手, 15 手, 25 手, さらに 101 手からなるジャンケンを提案している [5]. 以下では，4 手以上であっても，異なる 2 手の間に必ず勝ち負けが設定されているようなジャンケンの類型のことを単にジャンケンと呼ぶ.

世界のジャンケンを調べてみると，明らかに他の手と比べて弱く，使うべきでない手が含まれるものもある．例えば上述の 4 手からなるフランスのジャンケンでは，石と井戸は共に鋏に勝ち紙に負けるが, 井戸が石に勝つため石を出すならば井戸を出すほうが良い. 伊藤らは，このように他の手の下位互換となっている手を無駄な手と定義し，手数 5 以上ならば無駄な手のないジャンケン（一般化ジャンケン）を作ることができることを示した [2].

さてこれらジャンケンを実際に遊ぶとして，どのように出す手を選ぶべきかを考えよう. 通常の 3 手からなるジャンケンの場合, 石・紙・鋏を 1/3 ずつの等確率で出すのが妥当な「戦略」であるように思われる．では伊藤らの定義した一般化ジャンケンでは出す手をどのような確率で選ぶべきだろうか. 本論文では一般化ジャンケンを 2 人ゼロ和ゲームとして定式化することにより，この問いに答えることを試みる．興味深いことに, 伊藤らが定義した無駄な手のないジャンケンであっても，最適戦略においては確率 0 で選ぶべき手, すなわち, 出すべきでない手, 戦略的に無駄な手が存在することがあることがわかる．より具体的には, 伊藤らが任意の自然数 $n \neq 2, 4$ に対して, n 手の無駄な手のないジャンケンが存在することを示したのに対し, 本論文では「戦略的に無駄」な手が存在しないジャンケンは必ず奇数手からなることを数理計画的な観点から証明する.

本論文の構成は以下の通りである．2 節で先行研究である伊藤らの研究について紹介, 記法等を導入する．3 節では，一般化ジャンケンを 2 人ゼロ和ゲームとして定式化し，3 手, 5 手, 6 手のジャンケンで取るべき最適混合戦略を求める．4 節で，単純なジャンケンを元に合成したジャンケンでの最適戦略について考察する．5 節では，戦略的に無駄な手を持たないジャンケンは必ず奇数手からなること, またその戦略の一意性などを示す．最後に 6 節で本論文のまとめを行う.

2 準備

本節では，先行研究である伊藤らの研究 [2] に基づき，グラフ理論の基本的な知識を仮定した上で, ジャンケンに関する定義，定理を紹介する.

2.1　ジャンケンとトーナメント

n 手からなるジャンケンをグラフにより表現することを考える．各手を頂点, 勝敗関係を有向辺を用いて表すと, ジャンケンは**トーナメントグラフ**, すなわち n-完全グラフの各辺を適当に有向辺に置き換えたものとして表すことができる（図 1）．頂点数 n のトーナメントグラフのことを特に n-トーナメントグラフと呼ぶ．各ジャンケンに対し，それに対応するトーナメントグラフが存在することになり，逆に任意のトーナメントグラフに対して，それに対応するジャンケンを考えることができる．

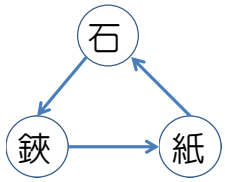


図 1　3 手のジャンケンのトーナメントグラフ表現

2.2　無駄な手

定義 1. ([2]) トーナメントグラフ $G = (V, E)$ において，ある 2 頂点 $u, v \in V$ が存在し，$(u, v) \in E$ であり，かつ，任意の $w \in V \setminus \{u, v\}$ に対して，$(v, w) \in E$ ならば $(u, w) \in E$ であるとき，u は v に**優越する**といい，v は**無駄な手**であるという．また無駄な手の存在しないトーナメントグラフとこれに対応するジャンケンを**効率的**であるという．

ある手 v に優越している手 u は手 v に勝ち，その v が勝てるすべての手にも勝つ．つまり，優越している手 u は，手 v の完全な上位互換となっており，わざわざ下位互換となっている手 v を使うのは無意味となる．これが v を無駄な手と呼ぶ理由である．なおこの定義では手数 1 のジャンケン, すなわち $|V| = 1$, $E = \emptyset$ のトーナメントグラフも効率的となる．実際に 1 手しかないジャンケンを遊ぶことはありえない（「あいこ」が続くだけである）が, 整合性からここではこのようなトーナメントグラフ (ジャンケン) を**自明なジャンケン**と呼び, 定義通り効率的であると認めることにする．

トーナメントグラフが効率的であることはグラフの直径の性質により言い換えることができる．グラフの頂点対 $u, v \in V$ に対し，u から v への距離 $\mathrm{dist}(u, v)$ を u を始点，v を終点とする最短路長で定義し，グラフ G の直径を $\mathrm{diam}(G) = \max_{u,v \in V} \mathrm{dist}(u, v)$ とする．このとき，以下の補題が成立する．

定理 1. ([2]) n 頂点からなるトーナメントグラフ $G = (V, E)$ において，以下の 3 条件は等価である．

1. G は効率的である．
2. G の直径が 2 以下である．
3. G の任意の有向辺 $(u, v) \in E$ に対して, それを含む長さ 3 の有向閉路が存在する．

2.3 効率的なトーナメントの存在性

以下の手続きにより，任意の効率的なジャンケンから，手数を 2 つ増やした新たな効率的なジャンケンを作ることが可能である（図 2)．

手続き 1. ([2]) ある n-トーナメントグラフ $G = (V, E)$ に対し，$(n + 2)$-トーナメントグラフ $G' = (V', E')$ を作成. ただし $V' = V \cup \{n + 1, n + 2\}, E' = E \cup \{(n + 1, n + 2)\} \cup \{(i, n + 1), (n + 2, i) \mid i \in V\}$ を満たす.

この手続きにより，任意のトーナメントグラフ G から手数を 2 つ増やしたトーナメントグラフ G' を作成できる．定理 1 から G が効率的であったとき，G' も効率的であることがすぐ確認できる．

2.4 効率的なジャンケン

手数 6 程度までなら定理 1 を考慮することで，しらみつぶしに効率的なジャンケンの形を求めることができる．手数 3 のときは，日本の通常のジャンケンの形が効率的であり，この 1 パターンしかない．定理 1 を考慮すると，手数 4 の効率的なジャンケンを作ることはできないことは容易に確かめられる．手数 5 の効率的なジャンケンは 2 パターン，手数 6 の効率的なジャンケンは 3 パターンある．これらから，手数 3, 6 のジャンケンを手続き 1 で拡張していくことにより, 5 以上の任意の手数において効率的なジャンケンが存在することがわかる．

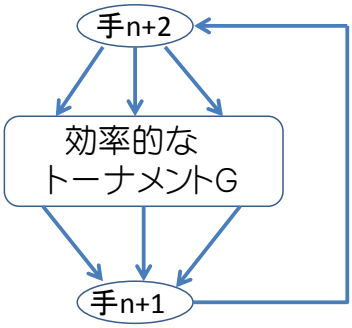


図 2　手続き 1

定理 2. ([2]) 自然数 n に対し、効率的な n-トーナメントグラフが存在する必要十分条件は $n \neq 2, 4$ である.

3　単純なジャンケンの最適戦略

3.1　2 人ゼロ和ゲームとしての定式化

一般化ジャンケンをプレイヤー 2 人（行プレイヤーと列プレイヤー）で遊ぶことを想定し, 2 人ゼロ和ゲームとして定式化する. 各プレイヤーはジャンケンのルールに従い, 勝てば $+1$, 負ければ -1 の利得を得るとし, 行プレイヤーから見た利得行列 $A = [a_{ij}]$ を用意する. 行プレイヤーの取る混合戦略を $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^\top$, 列プレイヤーの取る混合戦略を $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)^\top$ とすると, 行プレイヤーの期待利得を最大にすることを目的とした数理計画問題は, 以下のように定式化できる ($^\top$ は転置を表す). また, これを満たす $\boldsymbol{x}^*, \boldsymbol{y}^*$ がそれぞれ行プレイヤー, 列プレイヤーの最適混合戦略である.

$$
\begin{aligned}
&\max_{x_i} \min_{y_j} \sum_i \sum_j a_{ij} x_i y_j \\
\text{subject to} \quad &\sum_i \quad x_i = 1, \\
&\sum_j \quad y_j = 1, \\
&\qquad x_i \geq 0, \quad i = 1, 2, \ldots, n, \\
&\qquad y_i \geq 0, \quad i = 1, 2, \ldots, n.
\end{aligned}
$$

この定式化は $x_i y_j$ といった二次項を含むが，最適解 $\boldsymbol{x}^*$ は, 以下の線形計画問題を解くことにより得られることが知られている [1].

$$\begin{array}{lll} \text{(P)} \quad \max & z & \\ \text{subject to} & \sum_i & a_{ij}x_i - z \geq 0, \quad j = 1, 2, \ldots, n, \\ & \sum_i & x_i = 1, \\ & & x_i \geq 0, \qquad\qquad i = 1, 2, \ldots, n. \end{array}$$

以下では本定式化に基づき，一般化ジャンケンの最適戦略について考察する．まず本節にて自明でない最も単純な 3 手の最小のジャンケン，偶数手最小の効率的なジャンケンについて調べ，さらにその手数固有のジャンケンの最適戦略について考察する．その後，4 節ではジャンケンの拡張と最適戦略の関係について考察する.

3.2 手数 3 のジャンケンの最適戦略

まず，自明でない奇数手最小である手数 3 のジャンケンについて考察する．手数 3 のジャンケンは本質的には 1 種であり, その利得行列は以下の行列 A であらわされる:

$$A = \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix},$$

節点数 3 のトーナメントグラフを用いた 2 人ジャンケンについて，以下の定理が成り立つ.

定理 3. 節点数 3 のトーナメントグラフを用いた 2 人ジャンケンにおいて，戦略 $\boldsymbol{x} = (1/3, 1/3, 1/3)^\top$ が，唯一の最適戦略である.

証明. ジャンケンの 2 人ゼロ和ゲーム (P) は，自己双対問題（線形計画問題の双対問題が自身と本質的に同一であるような問題）であるから，(P) の最適値は必ず 0 となる [1]. よって，最適戦略を $\boldsymbol{x}^* = (x_1, x_2, x_3)^\top$ とおくと，制約式は，以下をみたす.

$$x_2 - x_3 \geq 0, \tag{1}$$
$$-x_1 + x_3 \geq 0, \tag{2}$$
$$x_1 - x_2 \geq 0, \tag{3}$$
$$x_1 + x_2 + x_3 = 1. \tag{4}$$

(1),(2),(3) より, $x_2 \geq x_3 \geq x_1 \geq x_2$ が成立, つまり，$x_1 = x_2 = x_3$ となる．よって，(4)

より，$x_1 = x_2 = x_3 = 1/3$ となる．また，この戦略以外に制約をみたす戦略はありえないため，これが唯一の最適戦略となる． □

3.3 手数 5，6 のジャンケンの最適戦略

次に，手数 5 のジャンケン, 偶数手最小のジャンケンである手数 6 のジャンケンについて考える．後でみるように, 手数 5 のジャンケンの最適戦略は手数 6 のジャンケンでの戦略と強い関係がある．前述の通り，節点数 5 の効率的なトーナメントグラフは 2 パターン存在するため，それぞれをパターン 5-1，パターン 5-2 と呼ぶことにする（図 3，4）．パ

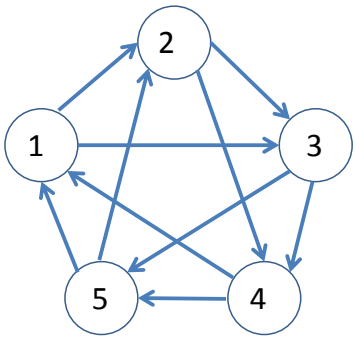


図 3　パターン 5-1 のトーナメントグラフ

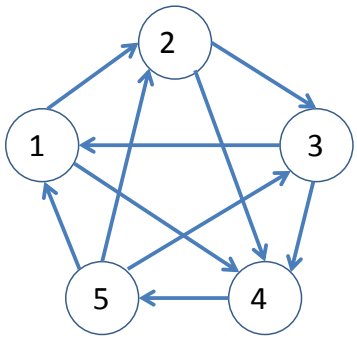


図 4　パターン 5-2 のトーナメントグラフ

ターン 5-1，パターン 5-2 における利得行列はそれぞれ以下の行列 $A_5^{(1)}$，行列 $A_5^{(2)}$ で表される：

$$A_5^{(1)} = \begin{bmatrix} 0 & 1 & 1 & -1 & -1 \\ -1 & 0 & 1 & 1 & -1 \\ -1 & -1 & 0 & 1 & 1 \\ 1 & -1 & -1 & 0 & 1 \\ 1 & 1 & -1 & -1 & 0 \end{bmatrix},$$

$$A_5^{(2)} = \begin{bmatrix} 0 & 1 & -1 & 1 & -1 \\ -1 & 0 & 1 & 1 & -1 \\ 1 & -1 & 0 & 1 & -1 \\ -1 & -1 & -1 & 0 & 1 \\ 1 & 1 & 1 & -1 & 0 \end{bmatrix}.$$

節点数 5 のトーナメントグラフ 5-1, 5-2 を用いた 2 人ジャンケンについて，以下の定理が成り立つ.

定理 4. 節点数 5 のトーナメントグラフ, パターン 5-1, パターン 5-2 における唯一の最適戦略はそれぞれ $\boldsymbol{x}^{(1)} = (1/5, 1/5, 1/5, 1/5, 1/5)^\top$, $\boldsymbol{x}^{(2)} = (1/9, 1/9, 1/9, 1/3, 1/3)^\top$ である.

証明. まずパターン 5-1 について考える. 先の証明と同様, (P) の最適値は 0 になる．よって最適戦略 $\boldsymbol{x}^{(1)} = (x_1, x_2, x_3, x_4, x_5)^\top$ は以下の制約をみたす.

$$-x_2 - x_3 + x_4 + x_5 \leq 0, \tag{5}$$
$$x_1 - x_3 - x_4 + x_5 \leq 0, \tag{6}$$
$$x_1 + x_2 - x_4 - x_5 \leq 0, \tag{7}$$
$$-x_1 + x_2 + x_3 - x_5 \leq 0, \tag{8}$$
$$-x_1 - x_2 + x_3 + x_4 \leq 0, \tag{9}$$
$$x_1 + x_2 + x_3 + x_4 + x_5 = 1. \tag{10}$$

まず (6)+(8) より，$x_2 \leq x_4$ が成立する．一方, (5)+(7)+(9) より，$x_4 \leq x_2$ が成立するため $x_2 = x_4$ となる．これを (5) に代入すると $x_5 \leq x_3$ が成立する．(7)+(9) より，$x_3 \leq x_5$ が成立するため, $x_3 = x_5$ となる． これを (6) に代入して， $x_1 \leq x_4$ が成立する．一方, (5) + (8) より，$x_4 \leq x_1$ がするため, $x_1 = x_4$ となる．(9) にこれを代入すると，$x_3 \leq x_2$ が成立する．一方, (6) + (7) より，$x_2 \leq x_3$ が成立するため, $x_2 = x_3$ となる．以上より，$x_1 = x_2 = x_3 = x_4 = x_5$ となることがわかる．よって，(10) より，$\boldsymbol{x}^{(1)} = (1/5, 1/5, 1/5, 1/5, 1/5)^\top$ となる.

次にパターン 5-2 について考える． パターン 5-1 と同様に，制約式のみから最適戦略を求めることが可能である．計算は省略するが, 最適戦略 $\boldsymbol{x}^{(2)}$ は $\boldsymbol{x}^{(2)} = (1/9, 1/9, 1/9, 1/3, 1/3)^\top$ となる．いずれも, これらは最適解が満たすべき制約式を満たす唯一の解であるため, 唯一の最適戦略を表す. □

なお, パターン 5-1 であらわされるジャンケンの唯一の最適戦略が $(1/5, 1/5, 1/5, 1/5, 1/5)^\top$ であることは, パターン 5-1 が 3.4 節で導入する平衡ジャンケンに該当するため, 定理 6 と定理 9 からも容易に確認できる． また，パターン 5-2 の唯一の最適戦略が $(1/9, 1/9, 1/9, 1/3, 1/3)^\top$ であることは, パターン 5-2 が, 手数 3 のジャンケンを手続 1 により合成したものとみなすことができるため, 後述の系 1 からも確認できる.

次に，手数 6 のジャンケンについて考える．前述の通り，節点数 6 の効率的なトーナメントは 3 パターン存在する．これらをパターン 6-1, 6-2, 6-3 と呼ぶ (図 5, 6, 7).

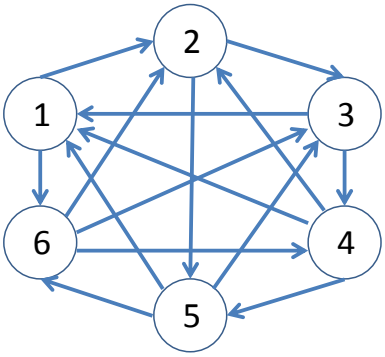


図 5　パターン 6-1

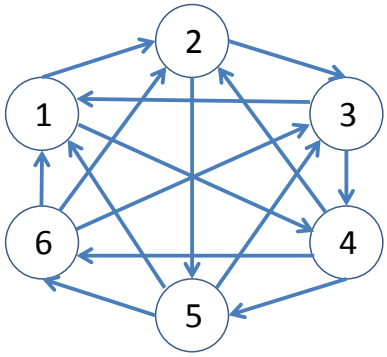


図 6　パターン 6-2

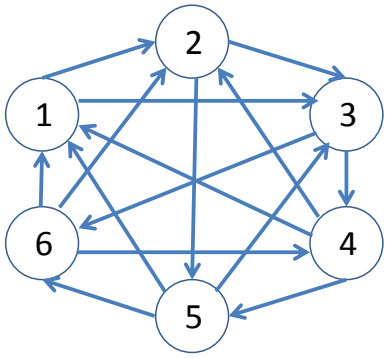


図 7　パターン 6-3

パターン 6-1, 6-2, 6-3 に対応する利得行列はそれぞれ以下の行列 $A_6^{(1)}$，行列 $A_6^{(2)}, A_6^{(3)}$ で表される：

$$A_6^{(1)} = \begin{bmatrix} 0 & 1 & -1 & -1 & -1 & 1 \\ -1 & 0 & 1 & -1 & 1 & -1 \\ 1 & -1 & 0 & 1 & -1 & -1 \\ 1 & 1 & -1 & 0 & 1 & -1 \\ 1 & -1 & 1 & -1 & 0 & 1 \\ -1 & 1 & 1 & 1 & -1 & 0 \end{bmatrix},$$

$$A_6^{(2)} = \begin{bmatrix} 0 & 1 & -1 & 1 & -1 & -1 \\ -1 & 0 & 1 & -1 & 1 & -1 \\ 1 & -1 & 0 & 1 & -1 & -1 \\ -1 & 1 & -1 & 0 & 1 & 1 \\ 1 & -1 & 1 & -1 & 0 & 1 \\ 1 & 1 & 1 & -1 & -1 & 0 \end{bmatrix},$$

$$A_6^{(3)} = \begin{bmatrix} 0 & 1 & 1 & -1 & -1 & -1 \\ -1 & 0 & 1 & -1 & 1 & -1 \\ -1 & -1 & 0 & 1 & -1 & 1 \\ 1 & 1 & -1 & 0 & 1 & -1 \\ 1 & -1 & 1 & -1 & 0 & 1 \\ 1 & 1 & -1 & 1 & -1 & 0 \end{bmatrix}.$$

節点数 6 のトーナメントグラフを用いた 2 人ジャンケンについて，以下の定理が成り立つ．

定理 5. 節点数 6 のトーナメントグラフ，パターン 6-1, 6-2, 6-3 において，$\boldsymbol{x}^{(1)} = (0, 0, 0, 1/3, 1/3, 1/3)^\top$, $\boldsymbol{x}^{(2)} = (0, 1/9, 1/3, 1/3, 1/9, 1/9)^\top$, $\boldsymbol{x}^{(3)} = (0, 1/5, 1/5, 1/5, 1/5, 1/5)^\top$ が，それぞれの唯一の最適戦略である．

証明. 定理 4 の証明と同様, $z = 0$ としたときの制約条件を満たす $\boldsymbol{x}$ がそれぞれ上述のベクトルのみであることを示すことができる．計算略． □

これらの最適戦略はいずれも出す確率を 0 とすべき手を含んでおり, 例えばパターン 6-1 は手 1, 2, 3 を選ぶ確率が 0 となっている．ここで行プレイヤー列プレイヤーが共に手 1, 2, 3 を確率 0 で出すということは, 利得行列 $A_6^{(1)}$ の第 4, 5, 6 行, 4, 5, 6 列のみを選んだ行列 $A_6^{(1)}[4, 5, 6; 4, 5, 6]$ 上でジャンケンを行っていることに相当するが，このときの利得行列は 3 手のジャンケンを表す A そのものとなっている． 同様に, パターン 6-2 は本質的にパターン 5-2, パターン 6-3 はパターン 5-1 と同じであることがわかる．

以上で，奇数手，偶数手最小のジャンケンで取るべき戦略がわかった．次に，奇数手のジャンケンのみに考えられる「平衡ジャンケン」の最適戦略について考察する．

3.4　平衡ジャンケン

一般に，各頂点の出次数, 入次数が等しい有向グラフを**平衡有向グラフ** (*balanced directed graph*) と呼ぶ．これに倣い, 各頂点の出次数, 入次数が等しいトーナメントグラフグラフを**平衡トーナメントグラフ** (balanced tournament graph), そのようなトーナメントグラフによりあらわされるジャンケンを**平衡ジャンケン**と呼ぶことにする．定義より, 平衡トーナメントグラフは常に奇数頂点からなる．平衡ジャンケンでは, そのトポロジーにかかわらず, 等確率で手を出す戦略が最適であることを示すことができる．

定理 6. ℓ を自然数とする． 節点数 $2\ell+1$ の平衡ジャンケンにおいて，戦略 $\boldsymbol{x} = (1/(2\ell+1), 1/(2\ell+1), \ldots, 1/(2\ell+1))^\top$ は，最適戦略である．

証明. 平衡ジャンケンでは, 各頂点 j において,

$$\sum_i a_{ij} = 0 \tag{11}$$

が成立する．問題 (P) の制約条件にて, $\boldsymbol{x} = (1/(2\ell+1), 1/(2\ell+1), \ldots, 1/(2\ell+1))^\top$ を代入すると, (11) より,

$$\sum_{i=1}^{2\ell+1} a_{ij}x_i = \frac{1}{2\ell+1}\sum_{i=1}^{2\ell+1} a_{ij} = 0 \geq z, j = 1, 2, \ldots, n,$$

$$\sum_{i=1}^{2\ell+1} x_i = \sum_{i=1}^{2\ell+1} \frac{1}{2\ell+1} = 1.$$

となり, 実行可能解かつ, 目的関数値が 0 となることがわかるが, 自己双対性よりこれは最適解である． □

4　ジャンケンの合成とその最適戦略

本節では, 2.3 節で紹介した手続き 1 を一般化した概念として, ジャンケンの合成について考えた後, 合成と最適戦略の関係について考察する．議論に進む前に, 以下を定義 (確認) しておく．

定義 2. 手 1 のみからなるジャンケン (手数 1 のジャンケン) における (最適) 戦略は, 確率 1 で手 1 を出すことである．

4.1 ジャンケンの合成

まずジャンケンの合成を定義する. この定義においては, 便宜上, 1 点のみからなるグラフもトーナメントグラフと見なすことにする.

定義 3. n-トーナメントグラフ $G = (\{1, 2, \ldots, n\}, E)$, n 個のトーナメントグラフの列 $(H_1, H_2, \ldots, H_n)$ (ただし, $H_i = (V_i, E_i)$, $i = 1, \ldots, n$) に対し,

$$V' = \bigcup_{i=1,\ldots,n} V_i,$$
$$E' = \bigcup_{i=1,\ldots,n} E_i \cup \bigcup_{(i,j)\in E} \{(u, v) \mid u \in V_i, v \in V_j\},$$

を考える. このとき, これらを頂点集合, 辺集合とするグラフ $G(H_1, \ldots, H_n) = (V', E')$ を $(H_1, H_2, \ldots, H_n)$ **の** G **上での合成**と定義する.

この定義について補足する. $G(H_1, \ldots, H_n)$ の頂点集合は, $H_1, H_2, \ldots, H_n$ のすべての頂点を併せたものであり, 各 V_i の中での辺は H_i のまま張られている. その他の頂点間, 例えば V_i に属する頂点 u と V_j に属する頂点 v の間に張られる辺は, G の辺の向きに従う. すなわち, $(i, j) \in E$ ならば (u, v) である, $(j, i) \in E$ ならば (v, u) である. この定義より, 各頂点間には有向辺が張られるため, 得られる $G(H_1, \ldots, H_n)$ もトーナメントである. このとき定理 1 から次の定理が成立する.

定理 7. 効率的な n-トーナメントグラフ $G = (\{1, 2, \ldots, n\}, E)$ と効率的な n 個のトーナメントグラフの列 $(H_1, H_2, \ldots, H_n)$ が与えられたとき, $(H_1, H_2, \ldots, H_n)$ を G 上で合成して得られる $G(H_1, \ldots, H_n)$ も効率的なトーナメントである.

ここで改めて手続き 1 について考える. 3 手からなる効率的なトーナメント $G_3 = (\{1, 2, 3\}, \{(1, 2), (2, 3), (3, 1)\})$ と, 与えられた n-トーナメント $G = (V, E)$ を H_1, 1 頂点のみからなる $H_2 = (\{n + 1\}, \emptyset)$, $H_3 = (\{n + 2\}, \emptyset)$ とし, (H_1, H_2, H_3) を G_3 上で合成して得られるトーナメント $G_3(H_1, H_2, H_3)$ は手続き 1 によって得られる G' そのものである. すなわち, 合成を手続き 1 の一般化とみなすことができる.

4.2 最適戦略の合成

前節で定義したトーナメントの合成とその上でのジャンケンの最適戦略の関係について, 以下を示すことができる: $G(H_1,\dots,H_n)$ における最適戦略は, G と各 H_i における最適戦略から「合成」することができる. 以下これを示すが, その前にまず記法を導入する. ベクトル $\boldsymbol{x}=(x_1,x_2,\cdots,x_n)^\top$ に対して $\boldsymbol{y}=(cx_1,cx_2,\cdots,cx_n,y_{n+1},y_{n+2},\cdots)^\top$ であるとき, これを簡潔に $\boldsymbol{y}=(c\boldsymbol{x},y_{n+1},y_{n+2},\cdots)^\top$ などと表すことにする.

定理 8. n-トーナメントグラフ $G=(\{1,2,\dots,n\},E)$ におけるジャンケンのある最適戦略 $\boldsymbol{x}^*=(x_1^*,\dots,x_n^*)^\top$ と, トーナメントグラフ $H_i=(V_i,E_i)$ $(i=1,2,\dots,n)$ 上の最適戦略 $\boldsymbol{y}^{(i)*}$ $(i=1,2,\dots,n)$ が与えられたとする. このとき, $(x_1^*\boldsymbol{y}^{(1)*},x_2^*\boldsymbol{y}^{(2)*},\dots,x_n^*\boldsymbol{y}^{(n)*})^\top$ は $G(H_1,\dots,H_n)$ におけるジャンケンの最適戦略である.

証明. G の利得行列を $A=[a_{ij}]$, H_i の利得行列を $B=[b_{uv}^{(i)}]$, $G(H_1,\dots,H_n)$ の利得行列を $C=[c_{uv}]$ とする. 頂点 u が元々 H_i の頂点であった (すなわち, $u\in V_i$) とすると,

$$c_{uv}=\begin{cases} b_{uv}^{(i)} & v\in V_i,\\ 0 & v=u,\\ a_{ij} & v\in V_j (j\neq i)\end{cases}$$

である. このとき, $\boldsymbol{\alpha}=(\alpha_u)=(x_1^*\boldsymbol{y}^{(1)*},x_2^*\boldsymbol{y}^{(2)*},\dots,x_n^*\boldsymbol{y}^{(n)*})^\top$ が問題 (P) において $z=0$ としたときの実行可能解となっていれば, 自己双対性より最適解であると言える. 以下これを示すが, 簡単のため $\boldsymbol{y}^{(i)*}$ の u に対応する頂点の成分を単に y_u^* と書くことにする. まず $\boldsymbol{x}^*,\boldsymbol{y}^{(1)*},\dots,\boldsymbol{y}^{(n)*}$ のそれぞれの実行可能性から,

$$\begin{aligned}\sum_u \alpha_u &= \sum_{i=1}^n \sum_{u\in V_i} \alpha_u = \sum_{j=1}^n \sum_{u\in V_i} x_i^* y_u^*\\ &= \sum_{i=1}^n x_i^* \sum_{u\in V_i} y_u^* = \sum_{i=1}^n x_i^* = 1.\end{aligned}$$

また (α_u) を (P) の v 行左辺に代入すると ($v \in V_k$ とする),

$$\begin{aligned}\sum_u c_{uv}x_u &= \sum_{i=1}^{n}\sum_{u\in V_i} c_{uv}\alpha_u \\ &= \sum_{i\neq k}\sum_{u\in V_i} c_{uv}\alpha_u + \sum_{u\in V_k} c_{uv}\alpha_u \\ &= \sum_{i\neq k} a_{ik}\sum_{u\in V_i}\alpha_u + \sum_{u\in V_k} b_{uv}^{(k)}\alpha_u \\ &= \sum_{i\neq k} a_{ik}x_i^*\sum_{u\in V_i} y_u^* + x_j^*\sum_{u\in V_k} b_{uv}^{(k)}y_u^* \qquad (12)\end{aligned}$$

となる. $\boldsymbol{y}^{(i)*}$ の H_i における実行可能性から, 各 i において, $\sum_{u\in V_i} y_u^* = 1$ が, $\boldsymbol{y}^{(k)*}$ の H_k における最適性から $\sum_{u\in V_k} b_{uv}^{(k)}y_u^* \geq 0$ が, $\boldsymbol{x}^*$ の G における最適性から $\sum_i a_{ik}x_i^* = \sum_{i\neq k} a_{ik}x_i^* \geq 0$ がそれぞれ成立するため, (12) は

$$\sum_u c_{uv}x_u \geq \sum_{i\neq k} a_{ik}x_i^* \geq 0$$

となるため, (P) の u 辺に対応する制約条件も $z = 0$ の下で満たすこととなる. 以上から (α_v) が最適解となることが示された. □

定理 3 とこの定理から, 手続き 1 に関する最適戦略を以下のように導出することができる.

系 1. トーナメントグラフ G で定義されるジャンケンの最適戦略が $\boldsymbol{x}^*$ であるとき, G に手続き 1 を適用して得られる G' で定義されるジャンケンにおいて,

$$\left(\frac{1}{3}\boldsymbol{x}^*, \frac{1}{3}, \frac{1}{3}\right)^\top$$

は最適戦略である.

なお, 伊藤らは手続き 1 を続けることにより作成されたジャンケンが興奮度という尺度の最大となるジャンケンであることを示している [2]. この系は, 興奮度最大のジャンケンの最適戦略を, 1 敗しかしない手と 1 勝しかできない手を選ぶ確率をそれぞれ 1/3 にする形で再帰的に導くことができることを示している.

この節の最後に, 合成とその最適戦略の例を一つ挙げておく. 手数 5 のジャンケン (パターン 5-1) における手 5 を, 手数 3 のジャンケンに置き換える形で合成する (図 8). 定

理 4, 定理 3 より, パターン 5-1 の最適戦略は，$\boldsymbol{x} = (1/5, 1/5, 1/5, 1/5, 1/5)^\top$, 手数 3 のジャンケンの最適戦略は，$\boldsymbol{y} = (1/3, 1/3, 1/3)^\top$ であるため, この 7 手のジャンケンの最適戦略は，$(1/5, 1/5, 1/5, 1/5, 1/15, 1/15, 1/15)^\top$ となる.

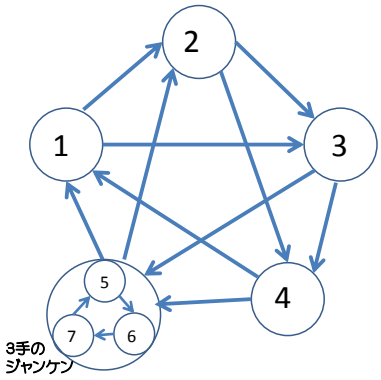


図 8　合成による 7 手のジャンケン

5　戦略的に無駄な手

3 節, 4 節と特定の形をした一般化ジャンケンにおける最適戦略, またジャンケンの合成と最適戦略の関係について考察を行った．ここで手数 6 のジャンケンの最適戦略に注目する．手数 6 の効率的なジャンケンは 3 パターンあるが，定理 5 で見たように, いずれのパターンにも最適戦略において, 出す確率を 0 とすべき手，すなわち「出すべきではない手」が存在している．しかし，この手は他の手の直接的な下位互換となっているわけではない．つまり，伊藤らの定義に基づけば意味のある手であっても，戦略的な観点からは無意味となる手が存在しうることを意味している．以上を踏まえ, 次のように定義する.

定義 4. 一般化ジャンケンが与えられたとき, ある最適戦略において確率 0 となる手を**戦略的に無駄な手**と呼ぶ．またすべての最適戦略において確率 0 となる手を**全戦略的に無駄な手**と呼ぶ．戦略的に無駄な手が存在しないトーナメントグラフを**狭義戦略効率的なトーナメントグラフ**, これに対応するジャンケンを**狭義戦略効率的なジャンケン**, 全戦略的に無駄な手が存在しないトーナメントグラフを単に**戦略効率的なトーナメントグラフ**, これに対応するジャンケンを**戦略効率的なジャンケン**と呼ぶ.

ここで「戦略的に無駄な手」は「無駄な手」の一般化に，「狭義戦略効率的なトーナメ

ントグラフ」は「効率的なジャンケン」の一般化になっている．実際, 手 i が手 j に優越するようなジャンケンを考えたとき, ある最適戦略において手 j を選ぶ確率が $x_j^* > 0$ であるようなベクトル $\boldsymbol{x}^* = (x_1^*, \ldots, x_n^*)^\top$ があったとすると, $\boldsymbol{x}^*$ の第 i 成分に第 j 成分を足し, 第 j 成分を 0 としたベクトル $\boldsymbol{x'} = (x_1^*, \ldots, (x_i^* + x_j^*), \ldots, 0, \ldots, x_n^*)^\top$ も最適戦略であり, 定義より手 j は戦略的に無駄な手となっている (後述するが，これらは共に「全戦略的に無駄な手」「戦略効率的なトーナメントグラフ」に置き換えることができる).

さて, 定理 3 より節点数 3 のトーナメントグラフは狭義戦略効率的であることから, これに手続 1 を繰り返し適用することにより得られるトーナメントは，すべて戦略効率的である (系 1). つまり, 任意の奇数 $n \geq 3$ に対して, 手数 n の戦略効率的なジャンケンが存在する．また後述の議論より, これらは狭義戦略効率的である．

では偶数手の戦略効率的なジャンケンは存在するのだろうか．ここまで見てきたように, 手数 4 の (戦略) 効率的なジャンケンは存在しない．また定理 5 より, 手数 6 の戦略効率的なジャンケンは存在しない．実は, 次の定理よりどのような偶数 n に対しても, 手数 n の戦略効率的なジャンケンは存在しない.

定理 9. あるジャンケンが戦略効率的であるなら，そのジャンケンは奇数手からなる．またその最適戦略は一意である.

以下，これを証明するための準備を行う．まず次の補題を示す.

補題 1. ジャンケンが戦略効率的であることの必要十分条件は, すべての手に対してその手を出す確率を正とする最適戦略が存在することである.

証明. この条件は 3.1 節の (P) において $x_i > 0, i = 1, \ldots, n$ となる最適解が存在することを意味する．(←) 定義より明らか．(→) 戦略効率的ならば，各 i に対して, $x_i > 0$ を満たす最適解が存在する．これを $\boldsymbol{x}^{(i)}$ とおくと, $\sum_{i=1}^n \boldsymbol{x}^{(i)}/n$ によって定義される x_i は (P) の最適解かつ, $x_i > 0, i = 1, \ldots, n$ を満たす. □

さて, (P) を詳しく見るため, $z = x_{n+1}$ とし, 以下を定義する.

$$A' = \left[\begin{array}{c|c} & -1 \\ A^\top & \vdots \\ & -1 \\ \hline 1 \cdots 1 & 0 \end{array}\right], \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ x_{n+1} \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix},$$

これらを用いると, (P) の目的関数は $\boldsymbol{b}^\top \boldsymbol{x}$ と表され, また制約条件を満たす $\boldsymbol{x}$ は $A'\boldsymbol{x} \geq \boldsymbol{b}$ を満たす. また双対問題 (D) の変数を $\boldsymbol{y} = (y_1, \ldots, y_n, y_{n+1})$ とすると, 目的関数は $\boldsymbol{b}^\top \boldsymbol{y}$, 制約条件を満たす $\boldsymbol{y}$ は $A'^\top \boldsymbol{y} \leq \boldsymbol{b}$ を満たす. これらを元に相補スラック条件を考える. 最適解の組 $\boldsymbol{x}^*, \boldsymbol{y}^*$ に対して, $\boldsymbol{b}^\top \boldsymbol{x}^* = \boldsymbol{b}^\top \boldsymbol{y}^*$ が成立することから,

$$\begin{aligned} 0 &= \boldsymbol{b}^\top \boldsymbol{x}^* - \boldsymbol{b}^\top \boldsymbol{y}^* \geq (A'^\top \boldsymbol{y}^*)^\top \boldsymbol{x}^* - \boldsymbol{y}^{*\top} \boldsymbol{b} \\ &= \boldsymbol{y}^{*\top} (A'\boldsymbol{x}^*) - \boldsymbol{y}^{*\top} \boldsymbol{b} = \boldsymbol{y}^{*\top} (A'\boldsymbol{x}^* - \boldsymbol{b}) \geq 0 \end{aligned}$$

すなわち, $j = 1, \ldots, n$ に対して, $y_j^* > 0$ ならば, $\sum_{i=1}^n a_{ij} x_i - x_{n+1} = 0$ が成立する. また (P) の $n+1$ 番目の制約は等式制約である. (P) が自己双対であることを考慮すると, 以下が成立する:

補題 2. (P) の最適解 $\boldsymbol{x}^* = (x_1, \ldots, x_n, x_{n+1})^\top$ で, $x_i > 0$ $(i = 1, \ldots, n)$ を満たすものが存在するならば, 連立方程式 $A'\boldsymbol{x} = \boldsymbol{b}$ は解を持つ. また $\boldsymbol{x}^*$ はその解の一つである.

以上で, 定理 9 の証明の準備が整った. 以下, 背理法により証明を行う.

証明. (定理 9) 偶数 n 手の戦略的に効率的なジャンケンが存在したとする. 補題 1, 2 より, $A'\boldsymbol{x} = \boldsymbol{b}$ は解を持つ. ジャンケンの定義より, 利得行列 A は歪対称行列, すなわち $A^\top = -A$ であるから, A' は $(n+1) \times (n+1)$ の歪対称行列である. 一般に $n \times n$ の歪対称行列 B は, n が奇数のとき, 非正則であることが知られている. これは $\det(B) = \det(B^\top) = \det(-B) = (-1)^n \det(B)$ から容易に確かめることができる. このことから, A' は非正則, すなわち, $\mathrm{rank}(A') \leq n$ である (rank は行列の階数を表す). このことは, A' を簡約化したときの主成分数が n 以下であることを意味する. 一方, トーナメントを表す行列の階数は n が奇数のときは $n-1$, n が偶数のときは n であることが知られている [3]. つまり, A を簡約化すると単位行列になる. 以上から, A' の簡約化は (1) まず A を単位行列に簡約化するのと同じ行基本変形をに施し, (2) 第 1 行から n 行までを順に第 $(n+1)$ 行から引くことにより完成し, 第 $(n+1)$ 行の要素はすべて 0 となる. すなわち, $[A'|\boldsymbol{b}]$ の簡約化は A' を簡約化する行基本変形 (上述) を施すことにより得られ, $\boldsymbol{b}$ の $(n+1)$ 要素の 1 はそのまま残ることになる.

以上は, A' の階数が n であるのに対し, $[A'|\boldsymbol{b}]$ の階数が $n+1$ であることを意味する. これは $A'\boldsymbol{x} = \boldsymbol{b}$ は解を持つことに反する. これより, 偶数 n 手の戦略的に効率的なジャンケンが存在しないことがわかる.

次に, 戦略的に効率的なジャンケンが存在した場合, その最適戦略は一意であることを示す. 上述の議論より, 戦略的に効率的なジャンケンが存在した場合, それは奇数手 n か

らなる. 補題 1, 2 から $A'\boldsymbol{x} = \boldsymbol{b}$ は解を持つ. さて, A' は $n+1$ 点からなるトーナメントを表す行列とみなすことができる. 上と同様 ($n+1$ が偶数であることから), [3] から A' の階数は $n+1$, すなわち正則である. よって A' には逆行列 A'^{-1} が存在するため, 最適解は $A'^{-1}\boldsymbol{b}$ と一意に定まる. □

定理 9 と補題 1 から, 任意の戦略効率的なジャンケンではすべての手に対してその手を出す確率を正とする最適戦略が存在し, それが唯一の最適戦略であることがわかる. このためいずれの手も戦略的に無駄な手ではないことから, これは狭義戦略効率的であることがわかる. つまり, 先述のように「戦略的に無駄な手」「全戦略的に無駄な手」は「無駄な手」の一般化になっているし, 「狭義戦略効率的なトーナメント (ジャンケン)」「戦略効率的なトーナメント (ジャンケン)」は「効率的なトーナメント (ジャンケン)」になっている. また前節で示した定理からいくつかの系が導かれる. 以下, これらをまとめておく.

定理 10. トーナメント G が定義するジャンケンにおいて, 以下の 4 条件は等価である.

1. G において, すべての手に対してその手を出す確率を正とする最適戦略が存在する.
2. G における最適戦略が一意であり, その最適戦略において各手を出す確率は正である.
3. G は戦略効率的である.
4. G は狭義戦略効率的である.

系 2. 戦略効率的なトーナメントグラフは効率的である.

系 3. 節点数 $2\ell+1$ の平衡ジャンケンにおける唯一の最適戦略は $\boldsymbol{x} = (1/(2\ell+1), 1/(2\ell+1), \ldots, 1/(2\ell+1))^\top$ である.

系 4. 任意の平衡トーナメントグラフ (ジャンケン) は効率的である.

系 5. 戦略効率的な n-トーナメントグラフ $G = (\{1,2,\ldots,n\}, E)$ におけるジャンケンの最適戦略 $\boldsymbol{x}^* = (x_1^*, \ldots, x_n^*)$ と, 戦略効率的なトーナメントグラフ $H_i = (V_i, E_i)(i = 1,2,\ldots,n)$ 上の最適戦略 $\boldsymbol{y}^{(i)*}(i = 1,2,\ldots,n)$ が与えられたとする. このとき $G(H_1, \ldots, H_n)$ は戦略効率的であり, その唯一の最適戦略は $(x_1^*\boldsymbol{y}^{(1)*}, x_2^*\boldsymbol{y}^{(2)*}, \ldots, x_n^*\boldsymbol{y}^{(n)*})^\top$ である.

6　まとめと今後の課題

本論文では, 伊藤らにより定義された無駄な手のない n 手ジャンケン — 効率的なジャンケン — に対して最適戦略の観点から考察を行った. 興味深いことに, 効率的なジャンケンであっても戦略的には出すべきでない手が存在する場合がある. 本論文ではこのような観点から「無駄な手」を最適戦略の観点から一般化した「戦略的に無駄な手」を定義しその性質を調べた. 伊藤らが効率的なジャンケンに対する有向グラフとしての特徴づけを行い, 構成的な証明により自然数 $n \neq 2, 4$ に対して, 手数 n の効率的なジャンケンの存在性を示したのに対し, 本論文では偶数手の戦略効率的なジャンケンは存在しないことを数理計画的に, 任意の奇数 n に対して, n 手の戦略効率的なジャンケンが存在することを構成的に示した.

定理 9 から, 偶数 n に対しては, n-トーナメントグラフは効率的であっても戦略効率的ではないことがわかる. 一方, 奇数 n に関しては, 例えば $n = 3, 5$ の効率的なトーナメントグラフは 3 節で見たようにいずれも戦略効率的である. また, 4 節の最後で見た $n = 7$ の効率的なトーナメントグラフも戦略効率的である. さらに奇数手数の平衡トーナメントも系 3 より戦略効率的であるし, 系 1 より, 戦略効率的なトーナメントグラフに手続 1 を繰り返し適用して得られるトーナメントグラフは戦略効率的である. このように眺めると,「奇数 n の効率的なトーナメントグラフはいずれも戦略効率的なのではないか？」と思われてくるが, 実際のところどうなのだろうか.

残念ながらこの「予想」は成立しない. 例えば, 3.3 節でとりあげた 6 頂点のパターン 6-1 トーナメントグラフ $G_6^{(1)}$, 1 頂点のみからなる H_1 を合成した $G_6(G_6^{(1)}, H_1, H_1, H_1, H_1, H_1)$ を考える. これは 11 頂点からなる効率的なトーナメントグラフであり, 定理 8 から $(0, 0, 0, 0, 0, 0, 0, 0, 0, 1/3, 1/3, 1/3)^\top$ はその最適戦略である. よって定理 10 より, この 11 点のトーナメントグラフが与えるジャンケンは, 効率的であるが戦略効率的ではない奇数頂点数のトーナメントグラフの例となっていることがわかる.

この例からもわかるように, 合成によって得られるジャンケンが戦略効率的か否かは合成元となるジャンケンの性質に左右される. となると, 今度は次のような疑問が浮かぶ : 合成によって得られないような奇数手のジャンケン (**固有型**と呼ぶ) で戦略効率的でないものが存在するのだろうか. 固有型のトーナメントグラフは効率的なトーナメントグラフの合成において, 合成数における素数のような役割を果たすものであるが, 例えばその典型例として以下に定義する「すくみ型」がある.

定義 5. 次の $(2\ell+1)$-トーナメントグラフ $G=(V,E)$ を以下のように定義する：頂点 $V=\{1,2,\cdots,2n+1\}$ とし，$f(x)\equiv x \mod 2\ell+1$ (ただし $1\leq f(x)\leq 2\ell+1$) としたとき，$E_i=\{(i,f(i+1)),(i,f(i+2)),\cdots,(i,f(i+\ell)),(f(i+\ell+1),i),(f(i+\ell+2),i),\ldots,(f(i+2\ell),i)\}, E=\bigcup_{i=1}^{2\ell+1}E_i$ とする．このトーナメントグラフと，これに同型のものを**すくみ型**と呼ぶ．

このすくみ型が固有型であることは以下のように背理法により確かめられる：固有型でないとすると，3 点以上からなる集合 S が存在し，S で誘導される部分グラフも効率的となっている．この S に対し，$\forall u\in S:(i,u)\in E$, $\forall u\in S:(u,j)\in E$, となるような点 $i,j\notin S$ が存在する．このことは，$S\subseteq\{f(i+1),\ldots,f(i+\ell+1)\}$, $S\subseteq\{f(j+\ell+2),\ldots,f(j+2\ell)\}$ を意味する．しかし，このことは S で誘導される部分グラフが非巡回となることを意味し，効率的となることに反する．以上から，すくみ型は固有型である．しかし，すくみ型は同時に平衡トーナメントグラフでもあるため，定理 6 から戦略効率的であり，上述の疑問に答えるものではない．

以上の議論から，例えば，奇数頂点数の固有型トーナメントグラフで戦略効率的でないものが存在するかどうか，など色々な興味深い未解決の問題が残っていることがわかる．

謝辞

定理 9 の証明に関するコメントを下さった，慶応義塾大学の田村明久先生，電気通信大学の岡本吉央先生に感謝いたします．本研究は JSPS 科研費 24220003, 26540005 の助成を受けたものです．

参考文献

[1] Vasek Chvatal: *Linear Programming*, W. H. Freeman and Company, 1983.

[2] 伊藤 大雄, 永持 仁: “ジャンケンのトーナメント表現と意味ある拡張”, 数理解析研究所講究録, 906, pp.14-23, 1995.

[3] Clifford A. McCarthy and Arthur T. Benjamin:“Determinants of the Tournaments”, *Mathematics Magazine*, pp. 133–135. 1996.

[4] `http://fr.wikipedia.org/wiki/Pierre-feuille-ciseaux` (Retrieved 2015-04-01)

[5] `http://www.umop.com/rps.htm` (Retrieved 2015-04-01)

Listing Center Strings under the Edit Distance Metric*

Hiromitsu Maji[1] and Taisuke IZUMI[1]

Graduate School of Engineering, Nagoya Institute of Technology,
Nagoya, 466-8555, Japan.
cke17607@stn.nitech.ac.jp, t-izumi@nitech.ac.jp

Abstract. Given a set W of k strings of length n over an alphabet Σ, the center string of W is defined as the string w such that the maximum distance to w of all strings in W is minimized under some specified metric. We present a new algorithm for the decision version of this problem under the edit distance metric. Given a threshold parameter d, the algorithm lists all the strings such that the distance from any input string is bounded by d in $O((3^d(d+2))^k dk|\Sigma|n + Mn)$ time, where M is the number of the output strings. To the best of our knowledge, this is the first FPT algorithm for the center string under the edit distance metric (even as a finding algorithm). By a slight modification, we also obtain an algorithm listing length-l common subsequences of W, which runs in $O((n-l)^{k+1}k|\Sigma|l + Ml)$ time.

1 Introduction

Finding a common structure from a given set of strings is recognized as one of the important problems in computational biology. A *center string* (or equivalently, *closest string*) is the one that minimizes the maximum distance of all strings in a input set W under some specific metric. The decision version of the center string problem under metric δ, which is the primary problem considered in this paper, is formalized as follows:

> **Input**: A set W of k strings of length n over an alphabet Σ, and a threshold value $d \in \mathbb{N}$.
> **Output**: A string w such that $\delta(w, w') \leq d$ holds for any $w' \in W$ if it exists. Otherwise the value of "FALSE".

In the definition above, the distance metric is not concretely defined. Usually it is chosen according to applications. Popular metrics useful in many applications are *Hamming distance* and *edit distance*. Unfortunately, for both metrics, the center string problem is NP-complete [4], and thus we need some sort of relaxation for attacking this problem. In this paper, we consider fixed-parameter algorithms for the center string problem under the edit distance metric. However, unfortunately again, that problem is W[1]-hard with respect to the number of

* This work is supported in part by KAKENHI No. 15H00852 and 25289227.

input strings k [16], which implies that the problem is unlikely to have an algorithm with a running time such as $O(f(k)\cdot poly(n))$. The currently best algorithm is the one by Nicolas et al. [16], which achieves $O(|\Sigma|n^k)$ time bound.

To circumvent the hardness results above, we focus on the fixed-parameter tractability for parameters both d and k. That is, we explore the algorithms having a running time with the form of $O(f(d,k)\cdot poly(n))$. The primary contribution of this paper is that such an algorithm actually exists. Our new algorithm finds a center string in $O((3^d(d+2))^k dk|\Sigma|n)$ time . To the best of our knowledge, this is the first FPT algorithm for finding center string problem under the edit distance metric. By a simple extension of this finding algorithm , we propose an algorithm to list all the solutions in the output-sensitive manner: The algorithm lists all the center strings for the edit distance metric in $O((3^d(d+2))^k dk|\Sigma|n + Mn)$ time, where M is the number of the output strings. Since in typical scenarios the center string problem is considered with a small d, our algorithms are practically more useful than the previous one.

The algorithms are constructed with a new dynamic-programming strategy, where the DP table records the distance information around diagonal vertices in alignment graphs. Interestingly, we can utilize the same strategy to solve another problem. Our second result is an algorithm listing length-l common subsequences of all input strings. The time complexity of this algorithm is $O((n-l)^{k+1}|\Sigma|l + Ml)$. Note that the longest common subsequence (LCS) problem, which is the optimization version of the length-l common subsequence problem, is known to be NP-complete [14], and W[1]-hard for parameter k [17]. On the other hand, finding length-l common subsequences trivially allows an $O(|\Sigma|^l poly(n,k))$-time algorithm by checking all strings of length l. That is, it is fixed-parameter tractable for parameter l in the case of constant-size alphabets. Furthermore, Irving and Fraser shows two algorithms of finding a LCS of length at least l in $O((n-l)^{k-1}kn)$ and $O((n-l)^{k-1}kl + k|\Sigma|n)$ times respectively [9]. That is, finding a common subsequence with length near to n is also fixed-parameter tractable (for $n-l$). Our algorithm can be seen as a listing version of the two algorithms by Irving and Fraser.

The paper is organized as follows: In Section 2, we present the prior work related to the topics of this paper. Section 4 provides our algorithm for the center string problem. Its extension to the LCS problem is considered in Section 5 Finally, we conclude this paper with a future direction in Section 6.

2 Related Work

The center string problem for the Hamming distance metric (often called *closest string problem*) is extensively studied. In general, that problem is NP-complete [6, 11], but allows a fixed-parameter algorithm with respect to d. Following the first FPT-algorithm by Gramm et al. [7], a number of papers improved the time complexity [3, 13, 18]. The closest substring problem, which is a generalized version of the closest string problem, is also well studied. Interestingly the closest substring problem is W[1]-hard with respect to both d and k even if the alpha-

bet is binary [15]. Marx shows an efficient algorithm for computing the closest substring for small d and/or k (but it is not an FPT algorithm) [15].

Compared to the Hamming distance metric, the center string problem under the edit distance metric is less studied. As we stated in the introduction, the paper by Nicolas and Rivals is only the one explicitly considering that setting [16]. The case of other metrics is considered in [5].

The longest common subsequence (LCS) problem for multiple strings is regarded as a special case of the center string for the edit distance metrics. It is equivalent to the center string problem for the edit distance metric with substitution cost two. About exact solutions for the LCS problem, a few papers propose several algorithms with different characteristics [8, 9]. The LCS problem for some restricted instances is considered in [1, 2].

Another variant of the center string problem is the median string problem, which requires to find the string minimizing the sum of the distance to each input string. While the median string under the Hamming distance metric is easily solvable in polynomial time, the case for edit distance is known to be NP-complete [4], and W[1]-hard for parameter k [16].

Approximated solutions for the problems introduced above are also investigated [11, 12]. PTASs are allowed for the closest (sub)string problem [12], but the longest common subsequence problem has no polynomial-time algorithm with any approximation ratio better than n^c for some constant $c > 0$ unless P = NP [10]. No polynomial-time approximated solution for the center string problem under the edit distance metric is known so far.

3 Preliminaries

3.1 Edit Distance

We denote the alphabet by Σ. An element in Σ^* is called *string*. The length of a string w is denoted by $|w|$, and the i-th character of w is denoted by $w[i]$ ($1 \leq i \leq |w|$). The operator $\circ$ means the concatenation of two strings (or characters). For $w \in \Sigma^*$, let $ta(w)$ be the string obtained by removing the first character of w. That is, $w = w[1] \circ ta(w)$. Letting W be a set of strings, we define $W \circ x = \{w \circ x | w \in W\}$.

The *edit distance* $ED(w_1, w_2)$ between two strings w_1 and w_2 is defined as follows:

$$ED(w_1, w_2) = \begin{cases} \max\{|w_1|, |w_2|\} \\ \qquad\qquad\qquad\qquad (\text{if } |w_1| = 0 \vee |w_2| = 0) \\ \min\{ED(ta(w_2), ta(w_1)) + c(w_1[1], w_2[1]), \\ \ ED(ta(w_1), w_2) + 1, ED(w_1, w_2) + 1\} \\ \qquad\qquad\qquad\qquad (\text{otherwise}), \end{cases}$$

where $c(a, b)$ is the function returning zero if $a = b$ or one otherwise. Note that while we assume that $c(a, b)$ is uniform (i.e.,the substitution cost does not depend on target characters), our algorithm can be applied to the case of non-uniform cost functions (as long as it returns an integer value).

It is well-known that the computation of the edit distance between two strings w_1 and w_2 can be reduced to the shortest path problem for some directed acyclic graph $G(w_1, w_2) = (V, E, f)$, called *alignment graph*, which defined as follows (Fig. 1):

- $V = \{v_{i,j} | i, j \in [0, n]\}$.
- For any $i, j \in [0, n]$, $v_{i,j}$ has a directed edge to each vertex $v_{i+1,j}$, $v_{i,j+1}$, and $v_{i+1,j+1}$ if it exists.
- $f(e) = \begin{cases} 0 \text{ if } e = (v_{i,j}, v_{i+1,j+1}) \text{ for some } i, j \in [0, n] \text{ and } w_1[i] = w_2[j] \\ 1 \text{ otherwise.} \end{cases}$

An edge $e = (v_{i,j}, v_{i',j'}) \in E$ is called a *horizontal*, *vertical*, or *diagonal* edge if $i = i'$, $j = j'$, or $(i \neq i' \wedge j \neq j')$ holds respectively. The set of vertices $\{v_{i,j} | j \in [0, n]\}$ and $\{v_{i,j} | i \in [0, n]\}$ are called *i-th row* and *j-th column* respectively. The distance between two vertices u and v is denoted by $dist(u, v)$. In particular, if $u = v_{0,0}$, we omit the first argument and describe $dist(u)$ for short. The following theorem is a classical fact.

Theorem 1. $dist(v_{n,n}) = ED(w_1, w_2)$.

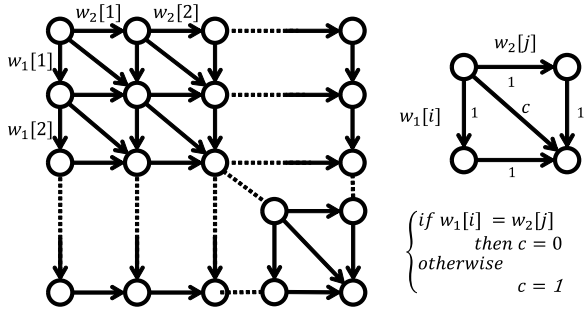


Fig. 1. Alignment graph $G(w_1, w_2)$

The *band* of alignment graphs is defined as the set of vertices $\{v_{i,j} | |i - j| \leq d/2\}$[1]. We also define $h(j)$ as the intersection size of the j-th column and the band. That is, let $h(j) = \min\{j + d/2, d, (n - j) + d/2\}$. For ease of explanation, we give aliases to each vertex in the band: The vertices of the j-th column in the band are called $u_{0,j}, u_{1,j}, \ldots, u_{h(j),j}$ from the upper side. The notations above are illustrated in Fig. 2.

We have the following lemma:

[1] The definition of the band depends on the value of d. Hence it may be more precise to include that dependency in the notation (e.g., calling d-band). However, to avoid the complication of notations, we treat the value of d as a certain kind of "global constant." Actually, in the following argument, we introduce several definitions dependent on d with no explicit description of the dependency.

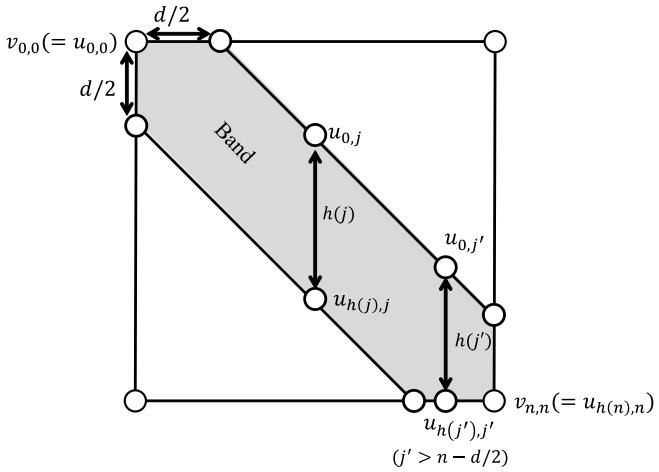


Fig. 2. Band and vertex aliasing

Lemma 1. *Given $w_1, w_2 \in \Sigma^n$ satisfying $ED(w_1, w_2) \leq d$, all the vertices constituting the shortest path from $v_{0,0}$ to $v_{n,n}$ in $G(w_1, w_2)$ are contained in the band.*

Proof. The shortest path from $v_{i,j}$ to $v_{n,n}$ must contain at least $|i - j|$ non-diagonal edges, all of which have weight one. Thus its length is more than or equal to $|i - j|$. Similarly, the length of the shortest path from $v_{0,0}$ to $v_{i,j}$ is also more than or equal to $|i - j|$. If $|i - j| > d/2$ (i.e., $v_{i,j}$ is out of the band), the length of any path from $v_{0,0}$ to $v_{n,n}$ via $v_{i,j}$ is more than d. It follows that $v_{i,j}$ cannot be contained in the shortest path because $dist(v_{n,n}) = ED(w_1, w_2) \leq d$. □

The lemma above implies that it suffices to consider the subgraph of $G(w_1, w_2)$ induced by the band because we only care about the paths from $v_{0,0}$ to $v_{n,n}$ of length at most d. We denote that induced subgraph by $B(w_1, w_2)$. The terminologies and notations introduced for $G(w_1, w_2)$ are also used for $B(w_1, w_2)$. In the following argument, we sometimes treat $B(w_1, w_2)$ for some string w_2 of length less than n. So we extend the definition of $B(w_1, w_2)$: For strings $w_1 \in \Sigma^n$ and $w_2 \in \Sigma^m$ such that $m < n$, we define $B(w_1, w_2)$ as the one in which edge $e = (v_{i,j}, v_{i',j'})$ for $j < m$ has the weight according to the original function f, and all other edges have weight one.

4 Listing Center Strings

In this section, we propose an algorithm for the center string problem, called ListCenter(W). The core idea of ListCenter(W) is to compute the intersection of

the k balls of radius d centered at each input string in $W = \{w_1, w_2, \cdots, w_k\}$. Thus, before explaining the main algorithm, we first introduce a preliminary algorithm called ListBall(w), which lists all the strings whose edit distance from w is at most d. The main algorithm is obtained by a simple extension of ListBall(w).

4.1 Algorithm ListBall: Listing Strings within Distance d

Letting $\lceil x \rceil_{d+1} = \min\{d+1, x\}$ for short, we define the (w_1, j)-*profile* of string w_2 as the vector $(\lceil dist(u_{0,j}) \rceil_{d+1}, \lceil dist(u_{1,j}) \rceil_{d+1}, \ldots, \lceil dist(u_{h(j),j}) \rceil_{d+1})$, where $dist(u_{i,j})$ is the distance in $B(w_1, w_2)$. Intuitively, the (w_1, j)-profile of w_2 is the distance vector to the vertices of the j-th column in the band, but the information about distances exceeding d are omitted. Without ambiguity, we often omit w_1 and simply call j-profile.

It should be noted that any $(h(j)+1)$-dimensional vector cannot become a j-profile. we say that $P \in [0, d+1]^{h(j)+1}$ is *possible* if there exists $w_1, w_2 \in \Sigma^n$ such that P becomes the (w_1, j)-profile of w_2. We can show a necessary condition for the possibility of P:

Lemma 2. *If $P = (p_0, p_1, \ldots, p_{h(j)}) \in [0, d+1]^{h(j)+1}$ is a possible j-profile, $|p_i - p_{i+1}| \leq 1$ holds for any $i \in [0, h(j)-1]$.*

Proof. Since $dist(u_{i+1,j}) - dist(u_{i,j}) \leq 1$ obviously holds because edge $(u_{i,j}, u_{i+1,j})$ has weight one, it suffices to show $dist(u_{i,j}) - dist(u_{i+1,j}) \leq 1$. Let $u_{0,0} = x_0, \cdots, x_r = u_{i+1,j}$ be the shortest path from $u_{0,0}$ to $u_{i+1,j}$ in $B(w_1, w_2)$, x_c be the last vertex contained in the i-th row, and $dist(x_c) = l$ (see Fig. 3). Since x_c is reachable to $u_{i,j}$ only traversing horizontal edges, $dist(u_{i,j}) \leq l + (r - c)$ holds. The shortest path from x_c to $u_{i+1,j}$ can contain at most one diagonal edge, its length is at least $r - c - 1$, and thus $dist(u_{i+1,j}) \geq l + (r - c - 1)$ holds. Consequently, we have $dist(u_{i,j}) - dist(u_{i+1,j}) \leq 1$. □

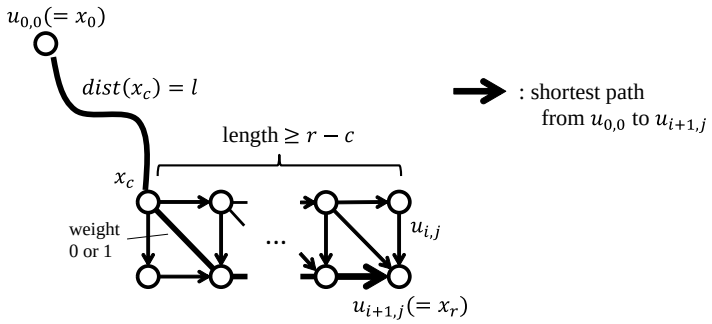


Fig. 3. Proof of Lemma 2

Let $\mathcal{P}_j$ be the set of the sequences in $[0, d+1]^{h(j)+1}$ satisfying the condition of Lemma 2. Clearly $\mathcal{P}_j$ contains all the possible profiles. From Lemma 2, we have the following corollary:

Corollary 1. *For any $j \in [0, n]$, $|\mathcal{P}_j| \leq 3^d(d+2)$.*

We also have the following corollary from the definition of profiles.

Corollary 2. *Any string w has 0-profile $P_{init} = (0, 1, 2, \ldots, h(0))$.*

For $P = (p_0, p_1, \ldots, p_{h(j)}) \in \mathcal{P}_j$ and $w \in \Sigma^n$, we define $S^w_{P,j}$ as the set of length-j strings having P as its (w, j)-profile. Let $\mathcal{P}_{term} = \{(p_0, p_1, \ldots, p_{h(n)}) \in [0, d+1]^{h(n)+1} | p_{h(n)} \leq d\}$. Then the union $\cup_{P \in \mathcal{P}_{term}} S^w_{P,n}$ is the set of the strings to be listed as the computation result. The core idea of $\mathsf{ListBall}(w)$ is to compute $S^w_{P,j}$ for any P and j via dynamic programming. To lead the DP recurrence formula, we introduce one more notation defined as follows: Let $j \in [0, n-1]$ and $w \in \Sigma^n$. For two vectors $P = (p_0, p_1, \ldots, p_{h(j)}) \in [0, d+1]^{h(j)+1}$ and $Q = (q_0, q_1, \ldots, q_{h(j+1)}) \in [0, d+1]^{h(j+1)+1}$, we say that P *is connected to* Q *with* $(x, w) \in \Sigma \times \Sigma^*$, denoted by $P \overset{w,x,j}{\longmapsto} Q$, if there exists a string w' such that $w'[j+1] = x$ and P and Q are respectively the (w, j)-profile and $(w, j+1)$-profile of w'. The key fact to obtain the recurrence formula is the next lemma:

Lemma 3. *Fixing $w \in \Sigma^n$, the $(j+1)$-profile Q satisfying $P \overset{w,x,j}{\longmapsto} Q$ is uniquely determined from P and x in $O(d)$ time.*

Proof. The uniqueness of Q is obvious because any shortest path to a vertex in the $(j+1)$-th column must pass a vertex in the j-th column in $B(w, *)$. Thus we prove that Q can be computed in $O(d)$ time. In graph $B(w, *)$, we can determine the weights of all the edges between jth and $(j+1)$-th columns by x. Thus we can compute Q by calculating the distances up to d from $v_{0,0}$ to the vertices in the $(j+1)$-th column, provided distances up to d to the vertices in the j-th column. For any $i' \in [0, n]$, the predecessor of $v_{i',j+1}$ in the shortest path from $v_{0,0}$ to $v_{i',j+1}$ is either $v_{i'-1,j}$, $v_{i',j}$, or $v_{i'-1,j+1}$. So if the distances (up to d) to those vertices are already known, the shortest path to $v_{i',j+1}$ (with length up to d) can be computed in a constant time. This implies that the values of the $(j+1)$-profile can be fixed from the upper side sequentially (i.e., in the order of $u_{j+1,0}, u_{j+1,1}, \ldots, u_{j+1,h(j+1)}$). Since $h(j+1) = O(d)$ holds, we can compute Q in $O(d)$ time. The lemma is proved. □

This lemma implies that if we know the j-profile of a string w, we can know the $(j+1)$-profile of $w \circ x$ for any $x \in \Sigma$. Conversely, if we want to know some (unknown) string $w = w' \circ x$ having some $(j+1)$-profile, it suffices to idenfity w' and its j-profile. This fact induces the following recurrence formula.

$$S^w_{Q,j+1} = \bigcup_{\substack{x \in \Sigma \\ P : P \overset{w,x,j}{\longmapsto} Q}} S^w_{P,j} \circ x. \tag{1}$$

Now we are ready to explain the algorithm, which consists of the following two steps:

– The first step of the algorithm is to construct the edge-labeled DAG $\Gamma = (V_\Gamma, E_\Gamma, f_\Gamma)$ defined as follows:
  - $V_\Gamma = (\cup_{j=0}^{n}\{(P,j)|P \in \mathcal{P}_j\} \cup \{t\}$, where t is the special sink vertex. We also give alias s to the vertex $(P_{init}, 0)$.
  - A vertex (P,j) is connected to $(Q, j+1)$ by an edge with label x if $P \overset{w,x,j}{\longmapsto} Q$. Note that if two or more characters x satisfy $P \overset{w,x,j}{\longmapsto} Q$, (P,j) and $(Q,j+1)$ are connected by multiedges. Finally, we add edges from all the vertices (P,n) satisfying $P \in \mathcal{P}_{term}$ to t with the null-character label.
– For any s-t path $X = e_0, e_1, \ldots e_n$ in Γ, we define $\gamma(X)$ as the string formed by traversing X (i.e., $\gamma(X) = f_\Gamma(e_0) \circ f_\Gamma(e_1) \circ \cdots \circ f_\Gamma(e_n)$). The second step of the algorithm is to output $\gamma(X)$ for each s-t path X in Γ.

The correctness of this algorithm relies on the following lemma:

Lemma 4. *For any $(P,j) \in V_\Gamma$ and a string $w_2 \in \Sigma^j$, $w_2 \in S^{w_1}_{P,j}$ holds if and only if there exists a path X from s to (P,j) such that $w_2 = \gamma(X)$ holds.*

Proof. The proof is done by induction on j. (**Basis**): It is obvious from Corollary 2. (**Induction step**): Suppose as the induction hypothesis that the lemma holds for $j = j' - 1$ and consider the case of $j = j'$. We prove only the direction of $\Rightarrow$ because the opposite direction ($\Leftarrow$) is easily proved by following backward the argument. Let $w_2 = w_2' \circ x$. The recurrence formula (Eq. 1) implies that if $w_2 \in S^{w_1}_{P,j'}$, there exists a $(j'-1)$-profile P' such that $w_2' \in S^{w_1}_{P',j'-1}$ and $P' \overset{w,x,j'-1}{\longmapsto} P$ hold. Then, by the induction hypothesis, there exists a path X' from s to (P', j') and $w_2' = \gamma(X')$. So there exists a path X to (P,j') from s and $\gamma(X) = w_2' \circ x$. The lemma is proved. □

We consider the running time of the algorithm. In the first step, for each vertex (P,i), the algorithm needs to compute all the pairs (x,Q) such that $P \overset{w,x,i}{\longmapsto} Q$ holds. From Lemma 3, it can be computed in $O(|\Sigma|d)$ time. From Corollary 1, we also have $|V_\Gamma| = O(3^d dn)$. Thus the total running time of the first step is $O(3^d(d^2|\Sigma|n)$. For the second step, all s-t paths can be enumerated by the naive recursion after pruning the vertices from which t is unreachable. Its running time is $O(Mn + |E_\Gamma|)$, where M is the number of paths enumerated. Finally, we have the following theorem.

Theorem 2. *Given $w \in \Sigma^n$, algorithm ListBall(w) lists all the strings whose distance from w is less than or equal to d in $O(3^d d^2|\Sigma|n + Mn)$ time.*

4.2 Listing Center Strings

By extending algorithm ListBall(w), we construct the main algorithm ListCenter(W). The primary idea is that given $W = \{w_1, w_2, \ldots, w^k\}$, ListCenter($W$) concurrently runs ListBall(w_i) for each input $w_i \in W$. We give the detailed explanation below:

A k-tuple of profiles $\mathbf{P} = (P_1, P_2, \ldots, P_k) \in \mathcal{P}_j^k$ is called the (W, j)-*profile* of a string w if P_i is w's (w_i, j)-profile for any $i \in [1, k]$. The notation $\mathbf{P} \overset{W,x,j}{\longmapsto} \mathbf{Q}$ for $\mathbf{P} = (P_1, P_2, \ldots, P_k) \in \mathcal{P}_j^k$ and $\mathbf{Q} = (Q_1, Q_2, \ldots, Q_k) \in \mathcal{P}_j^k$ means that there exists $w \in \Sigma^n$ and $j \in [0, n]$ such that $\mathbf{P}$ and $\mathbf{Q}$ are w's (W, j)-profile and $(W, j+1)$-profile respectively and $w[j] = x$ holds.

The remaining structure of $\mathsf{ListCenter}(W)$ is almost the same as $\mathsf{ListBall}(w)$, but the definition of profiles and its connectivity relationship are replaced by the ones above. More precisely, the algorithm utilizes the graph Γ^k defined as follows, instead of Γ:

- $V_\Gamma = (\cup_{i=0}^{n} \{(\mathbf{P}, i) | \mathbf{P} \in \mathcal{P}_i^k\} \cup \{t\}$, where t is the special sink vertex. We also give alias s to $((P_{init}, P_{init}, \ldots, P_{init}), 0)$.
- A vertex $(\mathbf{P}, i)$ is connected to $(\mathbf{Q}, i+1)$ by an edge with label x if $\mathbf{P} \overset{W,x,i}{\longmapsto} \mathbf{Q}$. Note that if two or more characters x satisfy $\mathbf{P} \overset{W,x,i}{\longmapsto} \mathbf{Q}$, $(\mathbf{P}, i)$ and $(\mathbf{Q}, i+1)$ are connected by multiedges. Finally, we add the edges from all the vertices $(\mathbf{P}, n)$ satisfying $\mathbf{P} \in \mathcal{P}_{term}^k$ to t with the null-character label.

It is not difficult to prove that the string $\gamma(X)$ corresponding to a s-t path X in Γ^k has a distance at most d to each string $w_i \in W$. Thus by enumerating all s-t paths we can list all center strings. We bound the running time of this algorithm. The analysis of the second step completely follows that for $\mathsf{ListBall}(w)$. For the first step, the size of Γ^k is larger than Γ. The number of vertices in Γ^k is $O((3^d(d+2))^k n)$. In addition, the computation of outgoing edges for each vertex takes obviously k times of the case for $\mathsf{ListBall}(w)$, i.e., $O(k|\Sigma|d)$ time. Hence the total running time of the first step is $O((3^d(d+2))^k dk|\Sigma|n)$. Consequently we have the following main theorem.

Theorem 3. *Algorithm $\mathsf{ListCenter}(W)$ lists all center strings for W under the edit distance metric in $O((3^d(d+2))^k dk|\Sigma|n + Mn)$ time, where M is the number of output strings.*

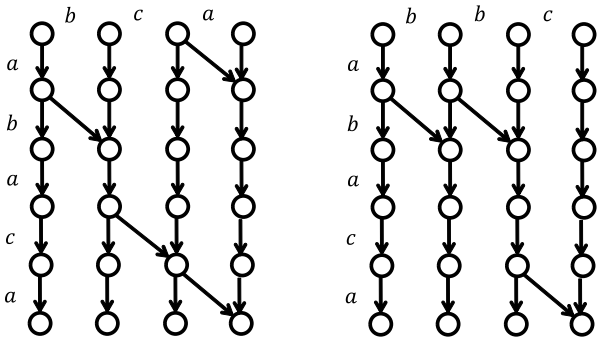
5 Listing Common Subsequences

A subsequence of a string $w \in \Sigma^n$ is any string obtained from w by deleting several characters. We denote by $Sub(w)$ the set of all subsequences of w. The decision version of the *longest common subsequence problem* (LCS) is defined as follows:

> **Input**: A set $W = \{w_1, w_2, \cdots, w_k\}$ of k strings over Σ of length n, and a threshold value $l \in \mathbb{N}$.
> **Output**: A string $w \in \cap_{i=1}^{k} Sub(w_i)$ such that $|w| \geq l$ if it exists. Otherwise the value of "FALSE".

Let $\bar{l} = n - l$ for short. In this section we show an algorithm called $\mathsf{ListLCS}_l(W)$, which is an algorithm listing common subsequences of length l. This algorithm is obtained by a refinement of $\mathsf{ListCenter}(W)$. For $w_1 \in \Sigma^n$ and $w_2 \in \Sigma^l$, we construct the *LCS alignment graph* $G_{LCS}(w_1, w_2) = (V_{LCS}, E_{LCS})$ as follows:

– $V_{LCS} = \{v_{i,j} | i \in [0, n], j \in [0, l]\}$.
– For any $i \in [0, n-1]$ and $j \in [0, l]$, add $e = (v_{i,j}, v_{i,j+1})$. In addition, add $e = (v_{i,j}, v_{i+1,j+1})$ if $w_1[i] = w_2[j]$.



(a) reachable (b) unreachable

Fig. 4. Two examples of LCS alignment graphs.

Note that LCS alignment graphs are unweighted. It is not difficult to prove the following lemma:

Lemma 5. *A string w_2 is a subsequence of a string w_1 if and only if $v_{0,0}$ is reachable to $v_{n,l}$ in $G_{LCS}(w_1, w_2)$.*

Two examples of LCS alignment graphs, which correspond to reachable and unreachable cases respectively, are shown in Fig. 4. We define the band of LCS alignment graphs as the set of vertices $\{v_{i,j} | j \leq i \leq j + \bar{l}, j \in [0, n]\}$ (see Fig. 5). The following lemma is analogous to Lemma 1 in the center-string case.

Lemma 6. *Any vertex $v_{i,j}$ out of the band is either unreachable to $v_{0,0}$ or $v_{n,l}$.*

Similarly as the center string case, let $B_{LCS}(w_1, w_2)$ be the subgraph of $G_{LCS}(w_1, w_2)$ induced by the band. Now we introduce the refined definition of profiles: The (w_1, j)-profile of w_2 is a binary $(\bar{l}+1)$-dimensional vector representing the reachability from $v_{0,0}$ to each vertex in the j-th column That is, a vertex $v_{i+j,j}$ is reachable from $v_{0,0}$ in $B_{LCS}(w_1, w_2)$ if and only if the (w_1, j)-profile $P \in [0, 1]^{\bar{l}+1}$ of w_2 satisfies $P[i] = 1$. Since $v_{i,j'}$ for $j' > j$ is reachable from $v_{0,0}$ when $v_{i,j}$ is reachable from $v_{0,0}$, any possible j-profile can be represented as the concatenation of an all-zero sequence followed by an all-one sequence. Furthermore, we do not have to consider the all-zero vector as a profile. Therefore, the total number of possible j-profiles is at most $\bar{l}$. We set $\mathcal{P}_j$ to all possible j-profiles.

The remaining part of algorithm $\mathsf{ListLCS}_l(W)$ is almost the same as $\mathsf{ListCenter}(W)$. Following the definition of profiles above, we construct the graph Γ^k and enumerate all s-t paths in Γ^k. Only the difference is the design of the source and the edges incoming to the sink in Γ^k. In the context of listing common subsequences, the $(\bar{l}+1)$-dimensional all-one vector is the unique possible 0-profile, and thus s is set to it. The vertices in $\{(P,l)|P[\bar{l}]=1\}$ is adjacent to t.

By the analysis similar with Section 4, we can bound the running time of this algorithm as follows:

Theorem 4. *For any set of k strings $W=\{w^1,w^2,\dots w^k\}$, algorithm $\mathsf{ListLCS}_l(W)$ enumerates all length-l common sequences in $O(\bar{l}^{k+1}k|\Sigma|l+Ml)$ time, where M is the number of output strings.*

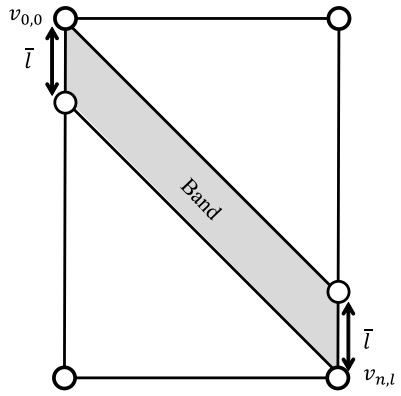


Fig. 5. Band of LCS alignment graphs.

6 Concluding Remarks

In this paper, we presented two algorithms called $\mathsf{ListCenter}(W)$ and $\mathsf{ListLCS}_l(W)$. Algorithm $\mathsf{ListCenter}$ enumerates all the center strings for given k strings and a threshold distance d in $O((3^d(d+2))^k dk|\Sigma|n+Mn)$ time. In addition, this algorithm finds one solution in $O((3^d(d+2))^k dk|\Sigma|n)$ time, which is the first FPT algorithm for the center string problem under the edit distance metric. Algorithm $\mathsf{ListLCS}_l$ is designed with the same framework as $\mathsf{ListCenter}$, which enumerates length-l common subsequences for k strings in $O(\bar{l}^{k+1}k|\Sigma|l+Ml)$ time.

On the parameterized complexity of the center string problem under the edit distance metric is surprisingly less studied. An important open problem is to show the fixed-parameter (in)tractability with respect to d only. While the

authors conjecture W[1]-hardness of that setting, the proof is still missing. Even if it is actually W[1]-hard, the exploration of faster algorithms (for example, running in $O(d^k \cdot poly(n)$ time) is also an interesting open problem.

References

1. G. Blin, P. Bonizzoni, R. Dondi, and F. Sikora. On the parameterized complexity of the repetition free longest common subsequence problem. *Inf. Process. Lett.*, 112(7):272–276, 2012.
2. G. Blin, L. Bulteau, M. Jiang, P. Tejada, and S. Vialette. Hardness of longest common subsequence for sequences with bounded run-lengths. In *Proc. of 23rd Annual Symposium on Combinatorial Pattern Matching*, pages 138–148. 2012.
3. Z.-Z. Chen and L. Wang. Fast exact algorithms for the closest string and substring problems with application to the planted (l,d)-motif model. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(5):1400–1410, 2011.
4. C. de la Higuera and F. Casacuberta. Topology of strings: Median string is np-complete. *Theoretical Computer Science*, 230(1-2):39 – 48, 2000.
5. L. P. Dinu and A. Popa. On the closest string via rank distance. In *In Proc. of 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 7354 of *Lecture Notes in Computer Science*, pages 413–426. 2012.
6. M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997.
7. J. Gramm, R. Niedermeier, P. Rossmanith, et al. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37(1):25–42, 2003.
8. K. Hakata and H. Imai. The longest common subsequence problem for small alphabet size between many strings. In *Proc. of the 3rd International Symposium on Algorithms and Computation (ISAAC)*, pages 469–478, 1992.
9. R. W. Irving and C. B. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In *In Proc. of 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644, pages 214–229. 1992.
10. T. Jiang and M. Li. On the approximation of shortest common supersequencesand longest common subsequences. *SIAM J. Comput.*, 24(5):1122–1139, oct 1995.
11. J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. *Information and Computation*, 185(1):41 – 55, 2003.
12. M. Li, B. Ma, and L. Wang. On the closest string and substring problems. *J. ACM*, 49(2):157–171, 2002.
13. B. Ma and X. Sun. More efficient algorithms for closest string and substring problems. *SIAM J. Comput.*, 39(4):1432–1443, 2009.
14. D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, 1978.
15. D. Marx. Closest substring problems with small distances. *SIAM J. Comput.*, 38(4):1382–1410, 2008.
16. F. Nicolas and E. Rivals. Hardness results for the center and median string problems under the weighted and unweighted edit distances. *Journal of Discrete Algorithms*, 3(2-4):390 – 415, 2005.
17. K. Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757–771, 2003.
18. L. Wang and B. Zhu. Efficient algorithms for the closest string and distinguishing string selection problems. In *Frontiers in Algorithmics*, pages 261–270. 2009.

A Warp-synchronous Implementation for Multiple-length Multiplication on the GPU

Takumi Honda, Yasuaki Ito, Koji Nakano
Department of Information Engineering,
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan
Email: {honda, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—If we process large-integers on the computer, they are represented by multiple-length integer. Multiple-length multiplication is widely used in areas such as scientific computation and cryptography processing. However, the computation cost is very high since CPU does not support a multiple-length integer. In this paper, we present a GPU implementation of bulk multiple-length multiplications. The idea of our GPU implementation is to adopt warp-synchronous programming. We assign each multiple-length multiplication to one warp that consists of 32 threads. In parallel processing using multiple threads, usually, it is costly to synchronize execution of threads and communicate within threads. In warp-synchronous programming, however, execution of threads in a warp can be synchronized instruction by instruction without any barrier synchronous operations. Also, inter-thread communication can be performed by warp shuffle functions without accessing shared memory. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 62 for 1024-bit multiple-length multiplication over the single CPU implementation.**

***Keywords*-Multiple-length multiplication; GPU; GPGPU; Parallel processing; warp-synchronous programming**

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].

Applications require arithmetic operations on integer numbers which exceed the range of processing by a CPU directly is called *multiple-length numbers* or *multiple-length-precision numbers* and hence, computation of these numbers is called *multiple-length arithmetic*. More specifically, application involving integer arithmetic operations for multiple-length numbers with size longer than 64 bits cannot be performed directly by conventional 64-bit CPUs, because their instruction supports integers with fixed 64 bits. To execute such application, CPUs need to repeat arithmetic operations for those numbers with fixed 64 bits which increase the execution overhead. Suppose that a multiple-length number is represented by w words, that is, a multiple-length number is $64w$ bits on conventional 64-bit CPUs. The addition of such two number can be computed in $O(w)$ time. However, the multiplication generally takes $O(w^2)$ time. Multiple-length multiplication is widely used in various applications such as cryptographic computation [6], and computational science [7]. Since multiple-length numbers of size thousands to several tens of thousands bits are used in such applications, the acceleration of the computation of their multiplications is in great demand. Also, considering practical cases, a large number of multiplications are usually computed. Therefore, in this work, we target at the computation for many multiple-length multiplications of such size.

Main contribution of this paper is to present an implementation of multiple-length multiplication optimized for CUDA-enabled GPUs. The idea of our GPU implementation is to adopt warp-synchronous programming. We assign each multiple-length multiplication to one warp that consists of 32 threads. In parallel processing using multiple threads, usually, it is costly to synchronize execution of threads and communicate within threads. In warp-synchronous programming, however, execution of threads in a warp can be synchronized instruction by instruction without any barrier synchronous operations. Also, inter-thread communication can be performed by warp shuffle functions without accessing shared memory. Using these ideas, we propose a warp synchronous implementation of 1024-bit multiplication on the GPU. In addition, we show multiple-length multiplication methods for more than 1024 bits using the 1024-bit multiplication method as a sub-routine. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 62 for 1024-bit multiple-length multiplication over the single CPU implementation.

In sequential implementation, we can utilize software libraries that support multiple-length arithmetic operations such as GMP (GNU Multiple Precision Arithmetic Library) [8]. A sequential CPU implementation with this library is used to compare the performance of our proposed GPU implementation. On the other hand, there are GPU implementations to accelerate multiple-length multiplications. In paper [9], [10], GPU implementations of very large

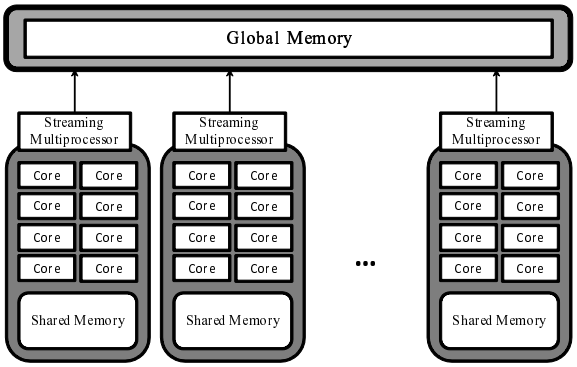


Figure 1: CUDA hardware architecture

integer multiplications using FFT are shown. Zhao *et al.* proposed multiple-length multiplication on the GPU as one of library functions [11]. This implementation is based on School method that is naive multiplication. Kitano *et al.* proposed a GPU implementation of parallel multiple-length multiplication also based on School method. In the implementation, load of each thread is equalized by reordering the computation of partial products.

The rest of this paper is organized as follows. Section II provides an overview of the GPU architecture. Section III describes multiple-length multiplication methods. This section also by reviewing the warp shuffle functions considered in this work. In Section IV, our GPU implementation of multiple-length multiplication using warp synchronize programming is proposed. Experimental results are shown in Section V. Finally, Section VI concludes the paper.

II. GPU Implementation

We briefly explain CUDA architecture that we will use. Figure 1 illustrates the CUDA hardware architecture. CUDA uses three types of memories in the NVIDIA GPUs: *the global memory*, *the shared memory*, and *the registers* [12]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-12 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The registers in CUDA are placed on each core in the multiprocessor and the fastest memory, that is, no latency is necessary. However, the size of the registers is the smallest during them. 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 [13], [14]. 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 coalescing access when they access to the global memory.

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. Also, the registers are only accessible by a thread, that is, the registers cannot be shared by multiple threads.

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*. A wrap is an implicitly synchronized group of threads. Each of these warps contains the same number of threads and is executed 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. When one warp is paused or stalled, other warps can be executed to hide latencies and keep the hardware busy.

There is a metric, called *occupancy*, related to the number of active warps on a streaming processor. The occupancy is the ratio of the number of active warps per streaming processor to the maximum number of possible active warps. It is important in determining how effectively the hardware is kept busy. The occupancy depends on the number of registers, the numbers of threads and blocks, and the size of shard memory used in a block. Namely, utilizing too many resources per thread or block may limit the occupancy. To obtain good performance with the GPUs, the occupancy should be considered.

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. On the other hand, all threads of a warp perform the same instruction at the same time. More specifically, any synchronizing operations are not necessary to synchronize threads within a warp.

In CUDA, *warp shuffle functions* allow the exchange of 32-bit data between threads within a warp, which become available on relatively recent GPUs with compute capability 3.0 and above [12]. Threads in the warp can read other threads' registers without accessing the shared memory. The exchange is performed simultaneously for all threads within the warp. Of particular interest is the `shfl()` function, that is one of the warp shuffle functions. This function takes as parameters a local register variable x and a thread index *id*. As an example, consider the following function call `shfl(`x`,4)`. The `shfl(`x`,4)` allows to transfer the data stored in the local register variable x from a thread whose *id* is 4 (Figure 2(a)). This function call corresponds to broadcasting a register variable in a thread to the other threads in a warp. We note that each thread has its own local register x, that is, each x cannot be accessed from other threads. As another example, consider the function call `shfl(`$x, (id+1)\%w$`)`. The function call performs data transfer like right circular shift between threads as illustrated in Figure 2(b). In the similar way, the `shfl(`$x, (id+w-1)\%w$`)` allows to transfer data like left right circular shift (Figure 2(c)). The above data exchange can be performed via shared memory. However, the latency of shared memory access is longer than that of the warp shuffle functions. Since the use of shared memory may cause for decreasing occupancy, if the warp shuffle functions can be used, they should be used.

Warp synchronous programming [15] is a parallel programming model such that one warp is used as an execution unit. The characteristic of this model is that any synchronous operations are not necessary. Usually, it is costly to synchronize execution of threads and communicate within threads. In our GPU implementation shown in the following sections, we adopt warp synchronous programming. Also, inter-thread communication is performed by warp shuffle functions without accessing shared memory.

III. Multiple-length multiplication

In the following, we will represent multiple-length numbers as arrays of r-bit words. In general, $r = 32$ or 64 for conventional CPUs. Let R denote the bit-length of numbers and d be the number of b-bit words. Therefore, $d = \lceil \frac{R}{r} \rceil$. For example, a 1024-bit integer consists of 32 words. Next, we will introduce several multiplication methods for such multiple-length numbers.

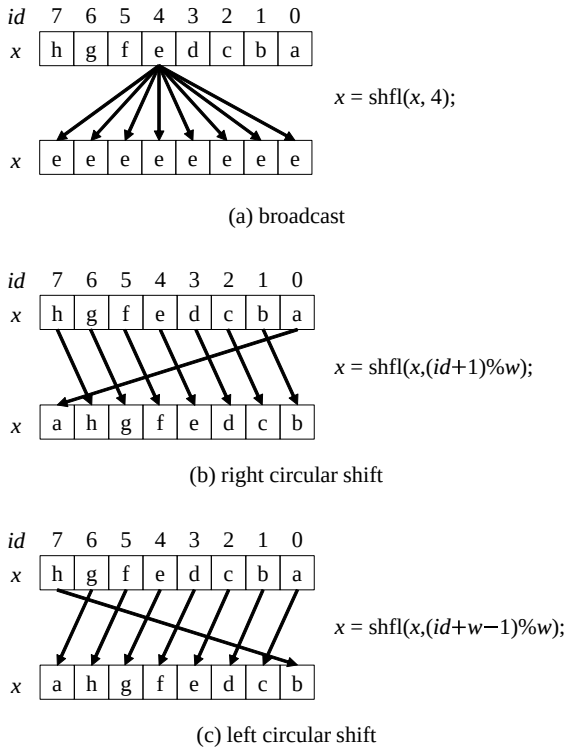
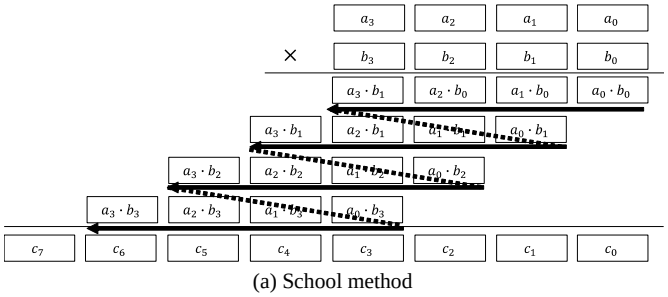


Figure 2: Example of intra-warp data exchange using warp shuffle functions

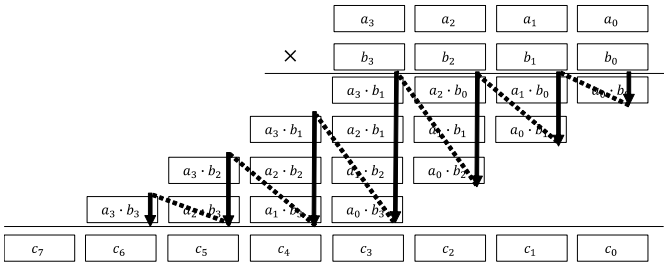
A. Multiple-Length Multiplication

Suppose A and B represent two multi-length numbers. We are multiplying A by B and the result is stored in C, that is $C = AB$. To compute this multiplication, *School method* is often used. The algorithm of School method is shown in Algorithm 1(a). For simplicity, in the algorithm, the sizes of the multiplicand and the multiplier are the same and $\{x, y\}$ denotes a concatenation of x and y. School method multiplies the multiplicand by each word of the multiplier and then adds up all the properly shifted results illustrated in Figure 3(a). As illustrated in the figure, calculation of School method is performed in the row order and some storage needs to be allocated to store intermediate results that are partial products. In School method, intermediate data that are partial products need to be stored to the memory as described at line 6 in Algorithm 1 is necessary.

To avoid storing the partial products, *Comba method* [16] is used. The algorithm of Comba method is shown in Algorithm 2. According to the algorithm, the readers may think that it is more complicated than School method. However, the difference is only the order of multiplications of words and the number of multiplications of words is the



(a) School method



(b) Comba method

Figure 3: The order of word-wise multiplication for multiple-length numbers $C = A \cdot B$

same as illustrated in Figure 3. More specifically, calculation of Comba method is performed in the column order. In Comba method, intermediate data also has to be stored. However, the data corresponds to carry data for the next column. Since the size of the carry data does not depend on the size of numbers and it is only one or two words, its storage can be placed to the register. Table I shows the number of word-wise multiplications and memory access of School and Comba methods. From the table, the number of memory access, especially memory write, of Comba method is greatly reduced.

Algorithm 1 School method

Input: $A = (a_{w-1}, \ldots, a_1, a_0), B = (b_{w-1}, \ldots, b_1, b_0)$
Output: $C = AB$

```
1: C ← 0
2: for j = 0 to w − 1 do
3:     {u, v} ← 0
4:     for i = 0 to w − 1 do
5:         {u, v} ← a_i b_j + c_{i+j} + u
6:         c_{i+j} ← v
7:     end for
8:     c_{2w+i} ← u
9: end for
```

Karatsuba method [17] is an algorithm for multiplying two numbers that reduce the number of multiplications compared with School method and Comba method. Let us consider multiple-length multiplication for $C = AB$, where A and B are multiple-length numbers of size R bits each. The two

Algorithm 2 Comba method

Input: $A = (a_{w-1}, \ldots, a_1, a_0), B = (b_{w-1}, \ldots, b_1, b_0)$
Output: $C = AB$

```
1: {t, u, v} ← 0
2: for i = 0 to w − 1 do
3:     for j = 0 to i do
4:         {t, u, v} ← a_j b_{i−j} + {t, u, v}
5:     end for
6:     c_i ← v
7:     v ← u, u ← t, t ← 0
8: end for
9: for i = w to 2w − 2 do
10:    for j = i − w + 1 to w − 1 do
11:        {t, u, v} ← a_j b_{i−j} + {t, u, v}
12:    end for
13:    c_i ← v
14:    v ← u, u ← t, t ← 0
15: end for
16: c_{2w−1} ← v
```

numbers A and B are divided into two parts of size $\frac{R}{2}$ bits each such that $A = A_1 \cdot 2^{\frac{R}{2}} + A_0$ and $B = B_1 \cdot 2^{\frac{R}{2}} + B_0$. The product C is computed as follows:

$$\begin{aligned} C &= AB \\ &= (A_1 \cdot 2^{\frac{R}{2}} + A_0)(B_1 \cdot 2^{\frac{R}{2}} + B_0) \\ &= A_1B_1 \cdot 2^R + (A_1B_0 + A_0B_1) \cdot 2^{\frac{R}{2}} + A_0A_0 \end{aligned}$$

In School method and Comba method, there are 4 multiplications $A_0 \times B_0$, $A_1 \times B_0$, $A_0 \times B_1$, and $A_1 \times B_1$. On the other hand, in Karatsuba method, the sum of two multiplications in the second term $A_1B_0 + A_0B_1$ is modified as follows:

$$A_1B_0 + A_0B_1 = (A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0$$

In this deformation, computing two products A_1B_1 and A_0B_0 beforehand, there are three multiplications $A_1 \times B_1$, $A_0 \times B_0$ and $(A_1 + A_0) \times (B_1 + B_0)$, though the number of addition/subtraction is increased. Thus, Karatsuba method can reduce the number of multiplications from four to three. This idea to the partial products can be applied recursively. Therefore, in Table— I, the number of multiplications and memory access is shown when Karatsuba method is applied once and then Comba method is used for smaller size of multiplications. Also, "Karatsuba^2" in the table represents a method such that Karatsuba method is applied twice and then Comba method is used. According to the table, when Karatsuba method is used, the number of multiplications is reduced. On the other hand, the number of memory access is increased. In the GPU, the latency of memory access is much longer than that of 32/64 bits multiplication. Therefore, in our implementation, Karatsuba method is not used recursively.

Table I: The number of multiplications and memory read/write for multiplying two w-word numbers

method	multiplication	memory read	memory write
School	w^2	$2w^2$	$w^2 + w$
Comba	w^2	$2w^2 - 2$	$2w$
Karatsuba	$\frac{3}{4}w^2$	$\frac{3}{2}w^2 + \frac{17}{2}w - 2$	$\frac{15}{2}w + 4$
Karatsuba^2	$\frac{9}{16}w^2$	$\frac{9}{4}w^2 + 20w - 4$	$\frac{31}{2}w + 14$

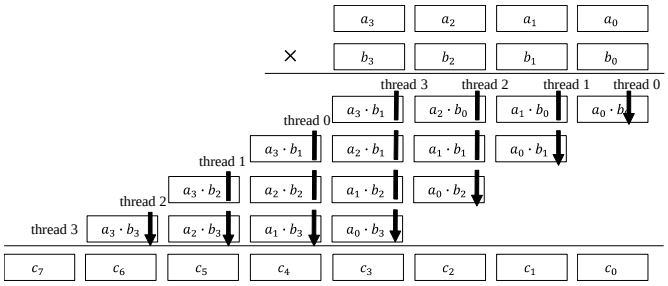


Figure 4: Parallel column-based multiplication

IV. Parallel multiple-length multiplication for the GPU

This section presents the main contribution of this work. We adopt warp-synchronous programming to the proposed parallel multiple-length multiplication. In the following, w threads, that correspond to one warp, are used and work in parallel without any barrier synchronize operations since threads within a warp execute the same instruction and synchronize for each instruction. Also, the proposed parallel multiple-length multiplication does not any use shared memory. It is a parallel algorithm that parallelizes School method basically, called *Sum-rotate multiplication*. To achieve this, we employ warp shuffle functions as described in Section II. More specifically, data exchange methods, broadcast and right/left circular shift, as shown in Figure 2 using warp shuffle function `shfl()` are utilized. The details of the parallel algorithm are presented next.

In the proposed approach, a product $C = (c_{2w-1}, \ldots, c_1, c_0)$ of two w-word numbers $A = (a_{w-1}, \ldots . a_1, a_0)$ and $B = (b_{w-1}, \ldots, b_1, b_0)$ is computed, where the size of each word is 32 bits. Since $w = 32$ unless the value of w is not changed for changing the GPU architecture in the future, this algorithm supports a multiplication of two 1024-bit numbers.

Let us consider how to perform the computation using multiple threads. A simple idea is to assign threads column by column as illustrated in Figure 4. In the figure, threads are assigned to two columns to balance the computation load of each threads. However, since threads have to switch columns in distinct timings during the computation, warp divergence, described in Section II, occurs. This parallel approach is not suitable for GPUs.

On the other hand, in the proposed approach, w threads,

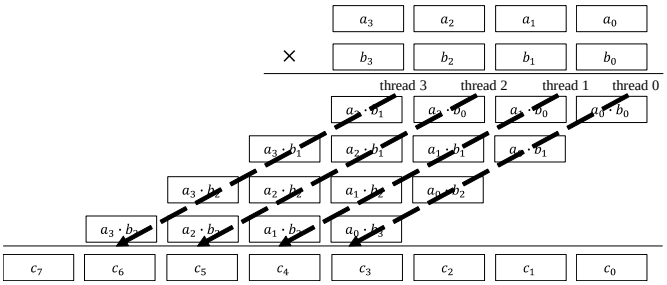


Figure 5: Sum-rotate multiplication

that correspond to one warp, are used. Each thread is assigned to one of the partial products in each column. More specifically, when w threads (thread 0, thread 1, ..., thread $w-1$) are launched, thread i computes partial products $a_i b_0, a_i b_1, \ldots, a_i b_{w-1}$ for each column as illustrated in Figure 5. Using this assignment of threads, almost all operations are the same between threads, that is, warp divergence can be avoided mostly.

In the proposed approach, since each thread takes partial products shifting to the upper digits row by row, it is necessary to obtain the partial products, except the carry, from a thread assigned to the upper digits. To achieve this, we use the inter-thread right circular shift described in Section II. in each row, thread 0 obtains the final product of c_j. According to Figure 5, a thread assigned to the lowest digits can obtain the lower words of the final product $c_0, \ldots, c_{w-1}$ for each row. On the other hand, the upper words of $c_w, \ldots, c_{2w-1}$ are finally computed by thread 0, ..., thread $w-1$, respectively. After completing the multiplication, $2w$ words of the final results are placed such that thread i has two words c_i and c_{i+w} to store the results to consecutive address of the global memory using coalescing access in parallel.

The details of Sum-rotate multiplication are shown in Algorithm 3. Each step of the algorithm is executed by w threads in parallel. First of all, in lines 3 and 4, each thread loads one word of each from A and B stored in the global memory and stores them to its own registers a and b. After that, the multiplication is performed row by row as illustrated in Figure 5. In line 6, thread j broadcasts b_j to local register b' using the warp shuffle function to compute the product $a \cdot b'$ in the next step. In line 7, partial products are computed including the addition of the carry from the upper digits. Each thread obtains the partial products except the carry from a thread assigned to the upper digits as the carry for the next digits by right circular shift of register v in line 8. In line 9, product c_i of the final product computed by thread 0 is transferred to the right thread using right circular shift of register c'. Next, thread $w-1$, that is assigned to the leftmost thread in Figure 5, set registers c' and v' to v and 0, respectively. This is for the right circular shift operations in lines 8 and 9. Since this operation is performed only by

thread $w-1$, warp divergence occurs, but the effect to the performance seems to be very small. After that, each thread obtains the value of the next digits in line 14. After for loop, each thread has the lower digits of the final products $c_0, \ldots, c_{w-1}$, respectively. At that time, the upper digits $c_w, \ldots, c_{2w-1}$ has not been computed yet since each thread still has the carry. Therefore, the while loop in lines 16 to 19, carry propagation is performed using left circular shift until any threads have no carry. In order to check whether any threads have no carry, we use warp vote function `any()` that evaluates truth values given from all threads of the warp and return non-zero if any of the truth values is non-zero [12]. This while loop is iterated at most $w-1$ times. After the loop, since thread i has two words c_i and c_{i+w}, they are stored to the global memory with coalescing access in lines 20 and 21.

Algorithm 3 Sum-rotate multiplication using a warp

Input: $A = (a_{w-1}, \ldots, a_1, a_0), B = (b_{w-1}, \ldots, b_1, b_0)$
Output: $C = AB$

1: $i \leftarrow id\ (= 0, 1, \ldots, w-1)$
2: $u \leftarrow 0, v \leftarrow 0, c' \leftarrow 0$
3: $a \leftarrow a_i$
4: $b \leftarrow b_i$
5: **for** $j \leftarrow 0$ **to** $w-1$ **do**
6: $\quad b' \leftarrow \texttt{shfl}(b_j, j)$ ▷ Broadcast b_j from thread j
7: $\quad \{t, u, v\} \leftarrow a \cdot b' + \{u, v\}$
8: $\quad v \leftarrow \texttt{shfl}(v, (i+1)\%w)$ ▷ Right circular shift v
9: $\quad c' \leftarrow \texttt{shfl}(c', (i+1)\%w)$ ▷ Right circular shift c'
10: $\quad$ **if** $id = w-1$ **then**
11: $\quad\quad c' \leftarrow v$
12: $\quad\quad v \leftarrow 0$
13: $\quad$ **end if**
14: $\quad \{u, v\} \leftarrow \{t, u\} + v$
15: **end for**
16: **while** $\texttt{any}(u) \neq 0$ **do** ▷ Loop for carry propagation
17: $\quad u \leftarrow \texttt{shfl}(u, (i+w-1)\%w)$ ▷ Left circular shift u
18: $\quad \{u, v\} \leftarrow u + v$
19: **end while**
20: $c_i \leftarrow c'$
21: $c_{i+w} \leftarrow v$

V. Experimental results

The main purpose of this section is to show the experimental results. In order to evaluate the computing time for multiple-length multiplication, we have used NVIDIA GeForce GTX 980, which has 2048 cores running on 1.216MHz [18]. In the following, the computing time is average of 10 times execution and the computing time of the GPU does not include data transfer time between the main memory in the CPU and the device memory in the GPU.

First, we evaluate the performance of the multiplication methods on the GPU. We have also implemented the single thread implementation such that each thread computes one multiplication. This implementation is based on the idea proposed in [19]. In the implementation, there is no warp divergence since all threads execute the same instructions, that is, this implementation is also based on warp-synchronous programming. In addition, to evaluate the effect of the use of warp shuffle function, we have implemented a multiplication method with the shared memory instead of the warp shuffle function. Table II shows the computing time when 100000 multiple-length multiplications are computed. Note that "Karatsuba^2" in the table denotes a multiplication method such that Karatsuba method is recursively applied twice. In the above implementations, every block has 32 threads, that is, one warp.

According to the table, we can find that one warp implementation is faster than the single thread implementation. For data communication within threads, use of warp shuffle functions is more effective than that of shared memory. Regarding multiplication methods in the one warp implementation with warp shuffle functions, for no more than 8192 bits, Comba method is faster and for more, Karatsuba method is faster than the other methods. Karatsuba^2 method is slower since the overhead such as the number of additional additions cannot be ignored. According to the results, the best configuration for the size of operands is selected such that Comba method is used for 1024 to 8192 bits and Karatsuba method is used for 16384 to 32768 bits.

We have also used Intel PC using Xeon X7460 running on 2.6GHz to evaluate the implementation by sequential algorithms. In the CPU implementation, we have utilized GMP version 4.1.4. Table III shows the comparison between CPU and GPU implementations for the computing time in milliseconds when 100000 multiple-length multiplications are computed. The best configuration in the above has been used in the GPU implementation. Using the proposed GPU implementation, the computing time can be reduced by a factor of 18.71 to 62.88.

Table III: The comparison between CPU and GPU implementations for the computing time in milliseconds when 100000 multiple-length multiplications are computed

# of bits	1024	2048	4096	8192	16384	32768
CPU	95.34	315.65	1031.40	3254.28	10505.87	30928.49
GPU	1.52	7.50	26.16	103.00	435.23	1653.45
Speed-up	62.88	42.10	39.43	31.59	24.14	18.71

VI. Conclusion

In this paper, we have presented a GPU implementation of bulk multiple-length multiplications. The idea of our GPU implementation is to adopt warp-synchronous programming. Using this idea, we have proposed Sum-rotate multiplication

Table II: The computing time of GPU implementations in milliseconds for 100000 multiple-length multiplications

execution unit	multiplication method	# of bits 1024	2048	4096	8192	16384	32768
single thread	Comba	5.06	38.96	165.56	684.61	2806.77	11411.26
	Karatsuba	6.59	21.62	80.22	320.46	1288.00	5333.75
	Karatsuba^2	5.62	16.54	58.02	219.59	867.14	3896.67
one warp (32 threads) with shared memory	Comba	1.99	8.28	32.16	125.84	501.04	2078.12
	Karatsuba	—	15.64	45.56	150.34	560.37	2057.02
	Karatsuba^2	—	—	60.24	164.61	589.34	2331.08
one warp (32 threads) with warp shuffle functions	Comba	**1.52**	**7.50**	**26.16**	**103.00**	440.38	1729.08
	Karatsuba	—	12.08	35.48	118.73	**435.23**	**1653.45**
	Karatsuba^2	—	—	53.38	144.79	490.27	1819.33

of two 1024-bit numbers. We assign each multiple-length multiplication to one warp that consists of 32 threads. The experimental results show that our GPU implementation on NVIDIA GeForce GTX 980 attains a speed-up factor of 62 for 1024-bit multiple-length multiplication over the single CPU implementation using GNU multiple precision arithmetic library.

REFERENCES

[1] NVIDIA Corporation, "CUDA ZONE." http://www.nvidia.com/page/home.html.

[2] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1369–1386, August 2012.

[3] 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 *Proceedings of International Conference on Networking and Computing*, pp. 120–127, 2010.

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

[5] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2010.

[6] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series, CRC Press, 2nd ed., 2014.

[7] L. M. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, vol. 414, no. 6566, pp. 883–887, 2001.

[8] T. Granlund, "GNU MP: The GNU multiple precision arithmetic library." http://gmplib.org/.

[9] N. Emmart and C. C. Weems, "High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes," *Parallel Processing Letters*, vol. 21, no. 3, pp. 359–375, 2011.

[10] H. Bantikyan, "Big integer multiplication with CUDA FFT (cuFFT) library," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 2, no. 11, pp. 6317–6325, 2014.

[11] K. Zhao and X. Chu, "GPUMP: A multiple-precision integer library for GPUs," in *Proc. of 2010 IEEE 10th International Conference on Computer and Information Technology*, pp. 1164–1168, 2010.

[12] NVIDIA Corporation, *CUDA C Programming Guide Version 7.0*, 2015.

[13] 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.

[14] NVIDIA Corporation, *CUDA C Best Practice Guide Version 7.0*, 2015.

[15] NVIDIA Corporation, *Tuning CUDA Applications for Kepler Version 7.0*, 2015.

[16] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[17] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Doklady Akademii Nauk SSSR*, vol. 145, no. 2, pp. 293–294, 1962.

[18] NVIDIA Corporation, "Whitepaper NVIDIA GeForce GTX 980 v1.1," 2014.

[19] D. Takafuji, K. Nakano, and Y. Ito, "C2CU: CUDA C program generator for bulk execution of a sequential algorithm," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, pp. 178–191, 2014.

Dynamic Parallelismを用いたマルチスレッドアルゴリズムの効率的な実現について

法政大学大学院理工学研究科　応用情報工学専攻　木村　哲也

法政大学理工学部　応用情報工学科　和田　幸一

大阪府立大学大学院　理学系研究科　藤本　典幸

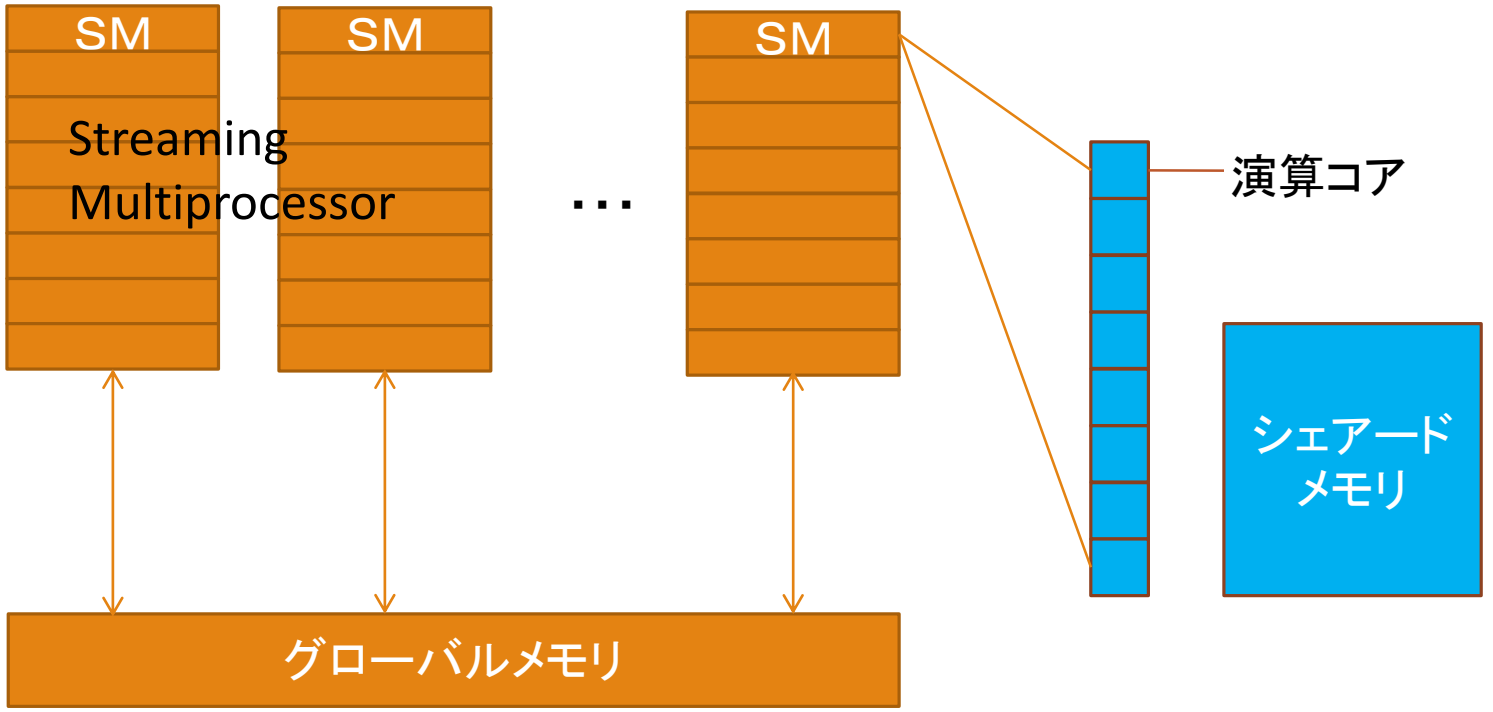
はじめに

GPUは画像処理を担当する主要な部品であり,CPUに比べて高い並列処理能力を持っているため,GPUを用いて処理の高速化が検討されている.

マルチスレッドアルゴリズムをDynamic Parallelismを用いて,GPGPU上で効率よく実現できるかを考察する.

GPUについて

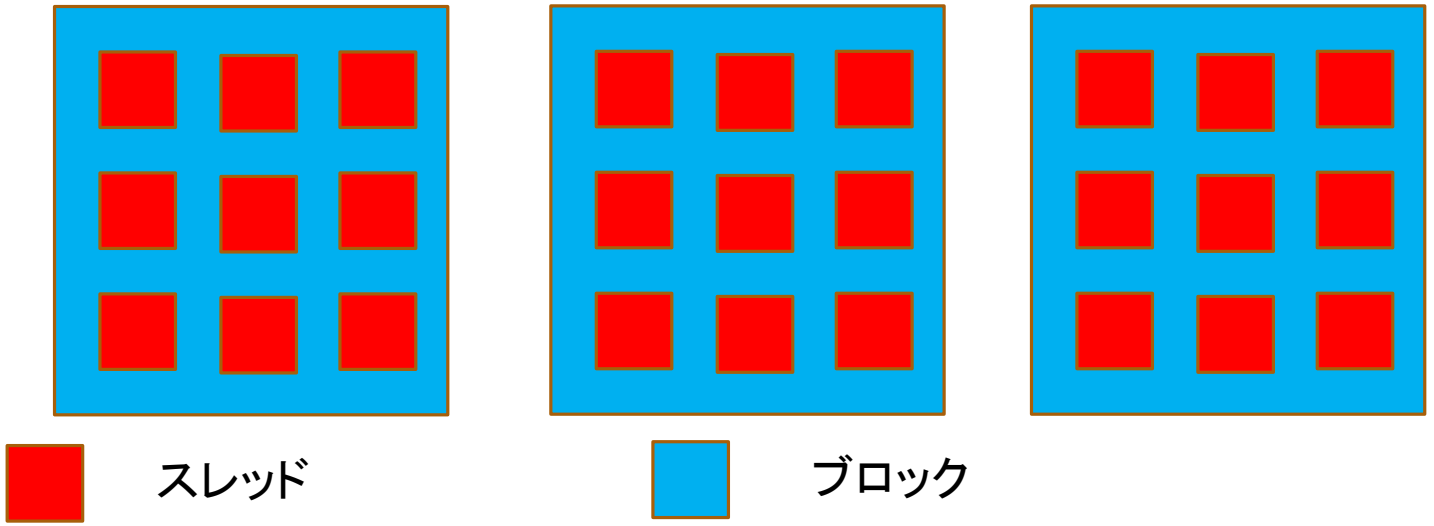
GPUのアーキテクチャを以下に示す.



ブロックとスレッド

GPUではカーネルを動作させた際の最小単位をスレッド,スレッドの集まりをブロックと呼ぶ.

1つのプロセッサに複数のスレッドを割り当て,スレッドはすべて同じ処理を行う.



マルチスレッドアルゴリズム

並列ループによって通常のfor文のループを並行して実行することができる.

動的マルチスレッドアルゴリズムでは,親が子を作り子が結果を出している間に親は計算を進めることができる.

スケジューラがスレッドを動的に確保するためにスケジュール管理する必要がある.

parallel,spawn,syncの擬似コードを使い記述できる.

マルチスレッドアルゴリズム

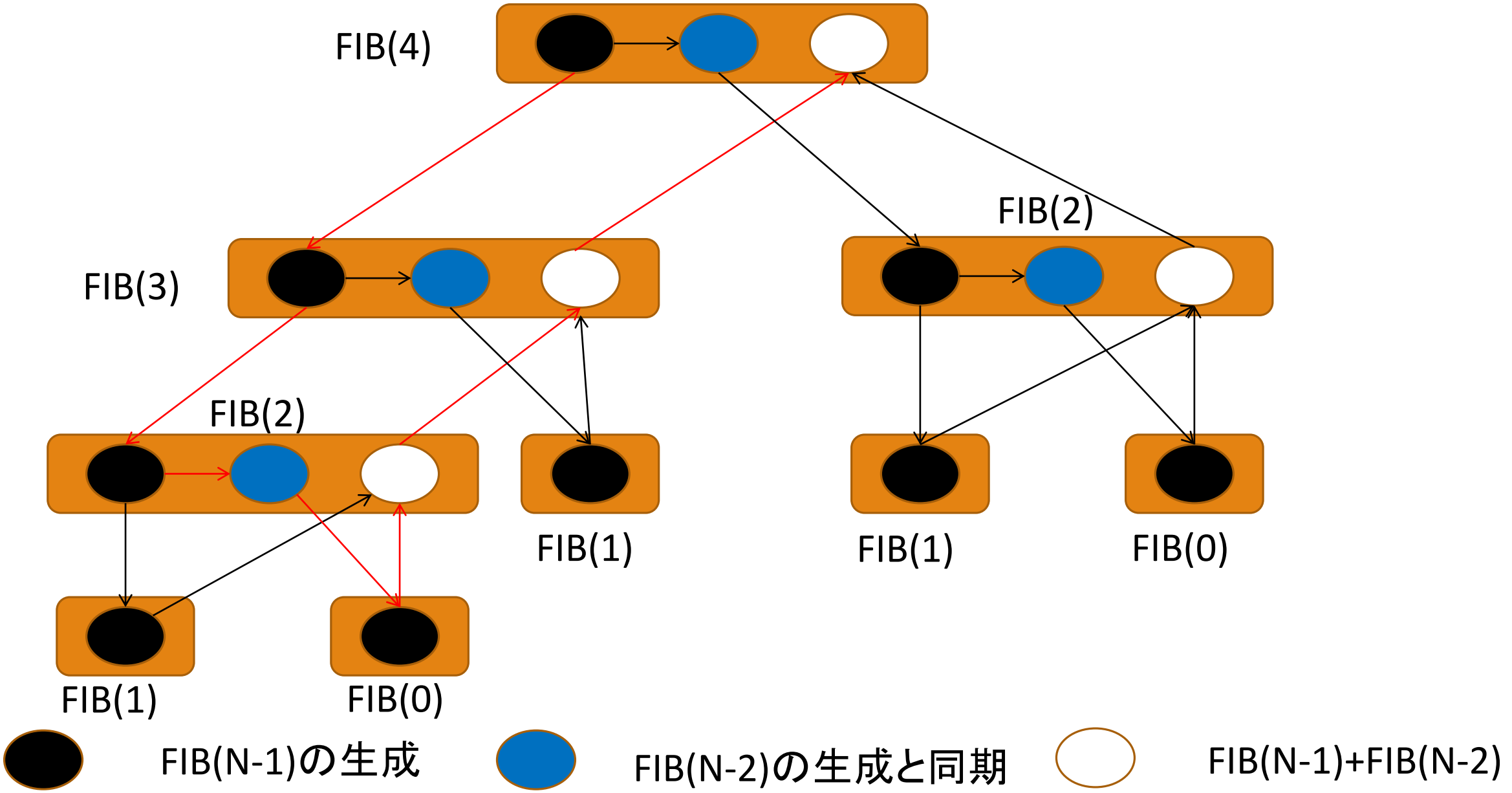
例)フィボナッチ数列を計算するアルゴリズム

逐次アルゴリズム

```
FIB(n)
  if n ≦ 1
    return n
else x=FIB(n − 1)
       y=FIB(n-2)
       return x + y
```

マルチスレッドアルゴリズム

```
P-FIB(n)
  if n ≦ 1
    return n
else x = spawn P-FIB(n − 1)
       y = P-FIB(n − 2)
       sync
       return x + y
```


FIB(4)
FIB(3)
FIB(2)
FIB(2)
FIB(1)
FIB(1)
FIB(0)
FIB(1)
FIB(0)
FIB(N-1)の生成
FIB(N-2)の生成と同期
FIB(N-1)+FIB(N-2)

マルチスレッドアルゴリズム

実行時間はプロセッサの台数や割り当て方でも影響する.

P台のプロセッサ上でのアルゴリズムの実行時間をT_Pとすると

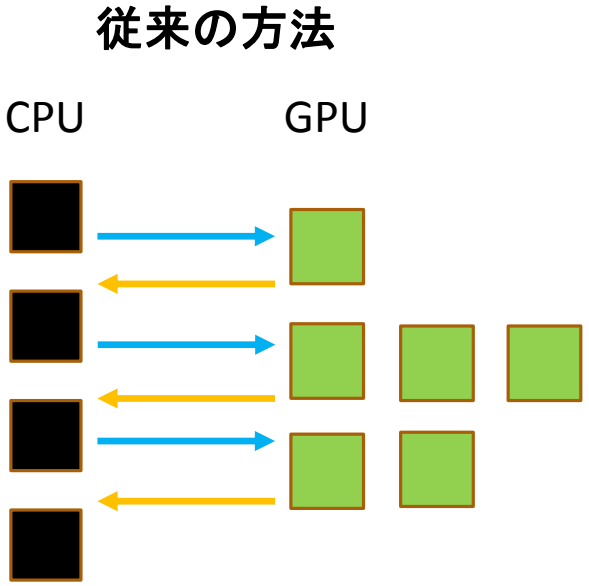
T_1＝1台のプロセッサ

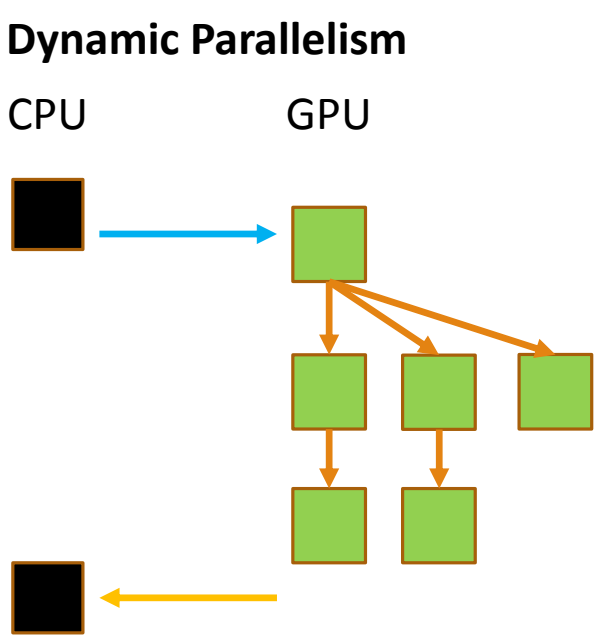
T_∞＝無限個のプロセッサ

T_1/T_∞を並列度といい,並列に実行できる平均の仕事量を表し,この値が任意のプロセッサ上で達成できる最大の高速化率になる.

Dynamic Parallelism

GPUがCUDAカーネルを実行しているときに,さらにその内部からカーネルを新しく生成する方法がDynamic Parallelismである.





Dynamic Parallelism

カーネル実行時に新たなカーネルを生成することでCPUとGPU間の通信を減らすことができる.

カーネルを再帰的に呼び出した回数のことをDepthといい,これは最大値が24という制約がある.

⇒データのサイズやデータ内容,プログラムによっては動作しない場合がある.

Dynamic Parallelism

Dynamic Parallelismを用いて以下のような研究を行っている.

・ソーティング問題
・行列積
・シストリックアルゴリズム
・漸化式で表される問題

クイックソート

通常のクイックソートとマルチスレッドアルゴリズムは以下のようになる.

逐次アルゴリズム

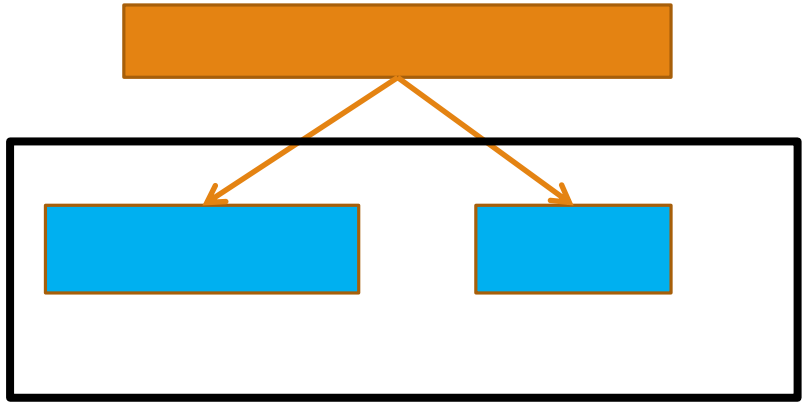
Quick(0…$n-1$)
  pivot = a[n / 2]
  p = partition(0, $n-1$ pivot)
  Quick(0…p)
  Quick($p+1$ … $n-1$)

マルチスレッド

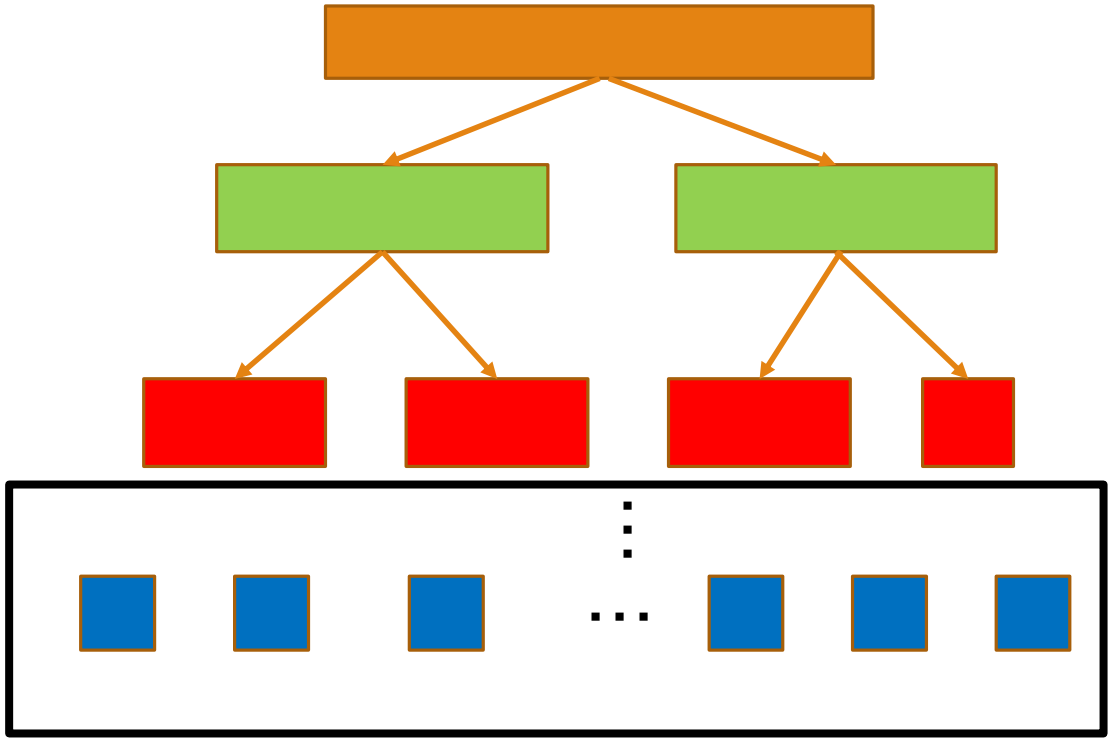
P-Quick(0…$n-1$)
  pivot = a[n / 2]
  p = partition(0, $n-1$, pivot)
  sync
  spawn P-Quick(0…p)
         P-Quick($p+1$ … $n-1$)

クイックソート

2分割



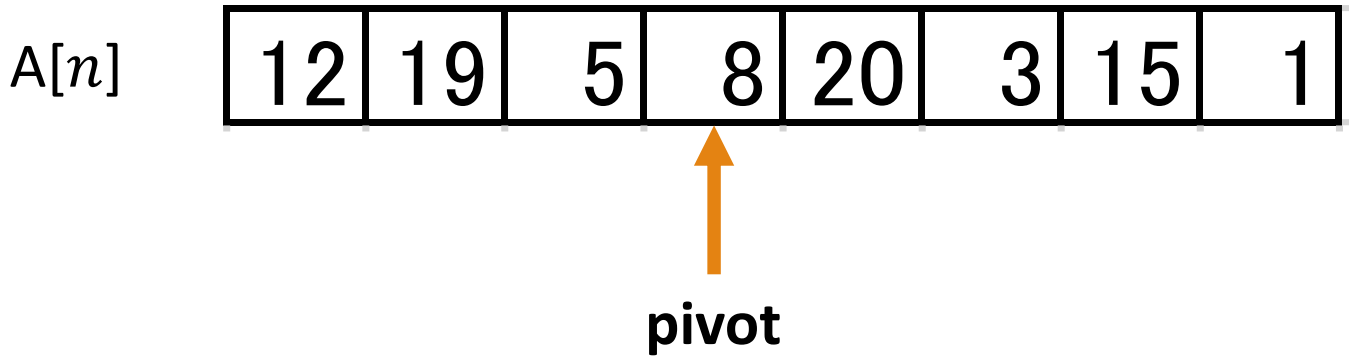
2^k分割



クイックソート

2分割のパーティションのマルチスレッド化について考える.

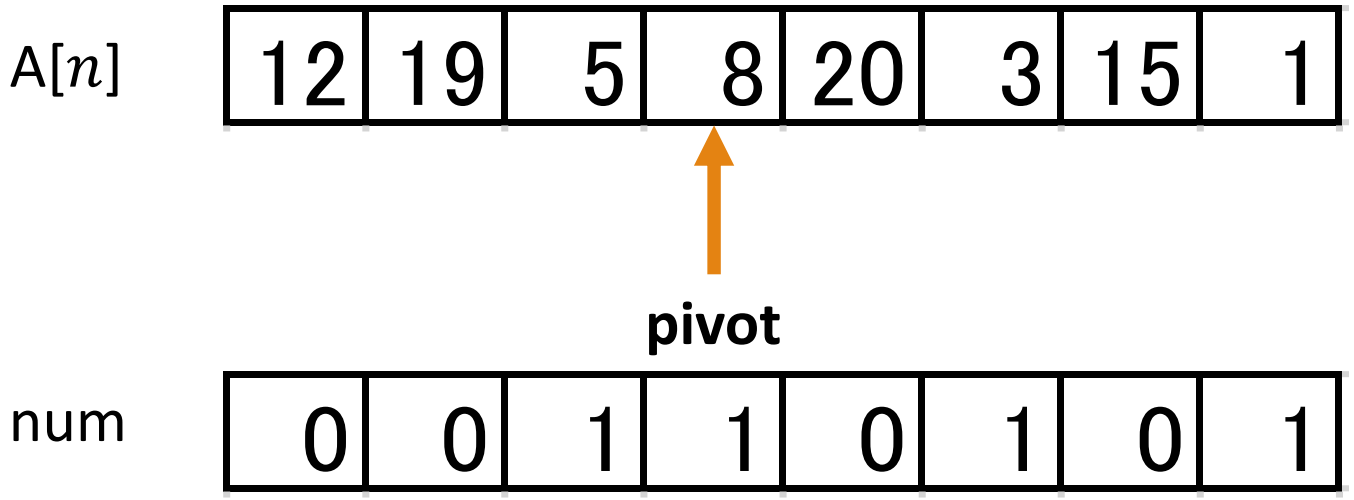
①pivotを決める.



クイックソート

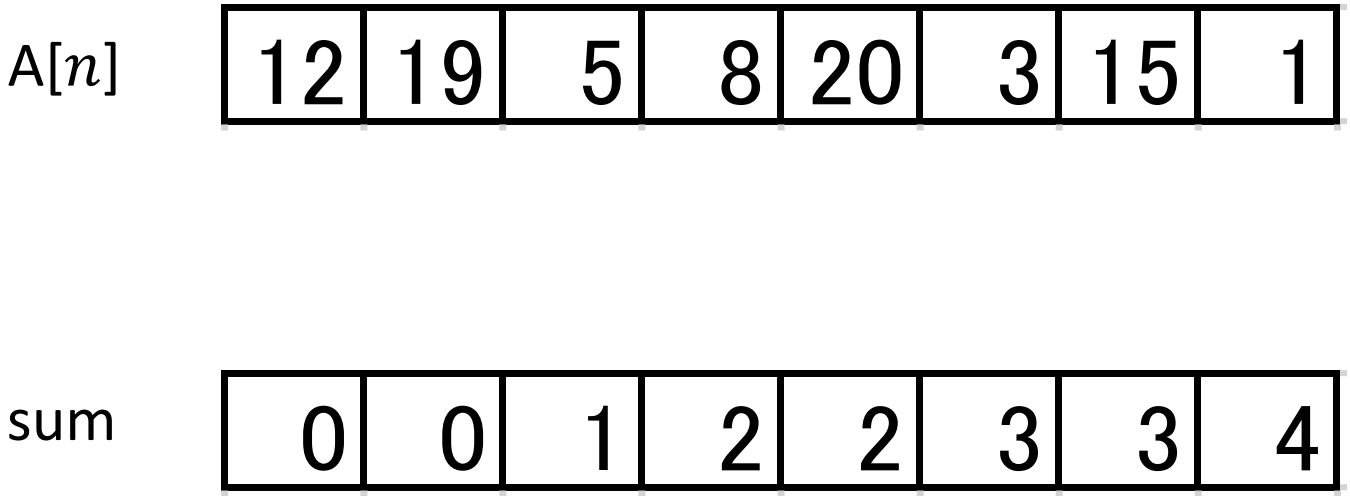
②各要素に対してpivot以下の場合は1,それ以外は0を立てる.

スレッドをn個用いることで並列に処理できる.



クイックソート

③numの値のプレフィックスサムをsumに代入.



クイックソート

④自分のsumと1つ前のsumを比較して下記の2つに場合分けする.

同一⇒A[n – 1 - idx + sum]にA[idx]を挿入.　異なる⇒A[sum - 1]にA[idx]を挿入.

A[n]	5	8	3	1	15	20	19	12

sum	0	0	1	2	2	3	3	4

クイックソート

マルチスレッド化した計算時間は次のようになる.

	仕事量	最悪スパン	最善スパン
クイックソート	$O(n\log n)$(最悪は$O(n^2)$)	$O(n\log n)$	$O((\log n)^2)$ $O(k\log n * \log_k n)$
マージソート	$O(n\log n)$	$O((\log n)^3)$	$O((\log n)^3)$

実験の評価

クイックソートをあるプログラムの一部のサブルーチンとして用いることを想定しているためCPUにCPU⇔GPUのデータ渡しの時間を含んでいる.

CPU
GPUからCPUへのデータ渡し
QuickSort
CPUからGPUへのデータ渡し

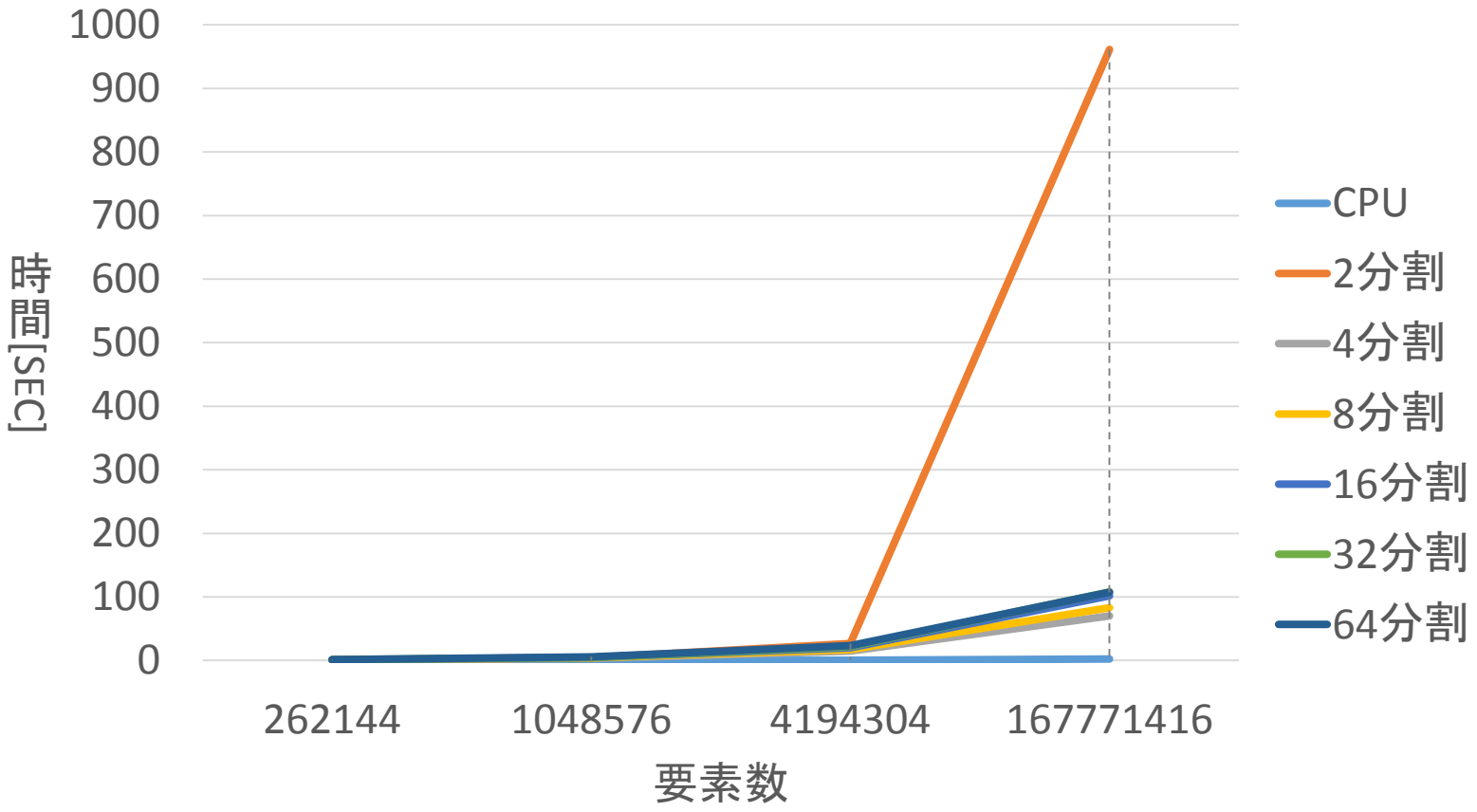
GPU
QuickSort

実験環境

GeForce GTX TITANを使い今回の実験環境を以下に示す.

演算コア数	2688
Streaming Multiprocessor数	14基
グローバルメモリ	6GB
要素の値	$0 \sim n-1$
選択ソートに分岐するサイズ	32要素

実験結果


1000
900
800
700
600
500
400
300
200
100
0
時間[SEC]
262144
1048576
4194304
167771416
要素数
CPU
2分割
4分割
8分割
16分割
32分割
64分割

まとめ・今後の方針

・マージソートについてクイックソートと同様に任意の分割数で実現していく.

・クイックソートの並列パーティションを実現する.

・選択ソートに分岐するデータのサイズの調整,pivotの位置の決め方やメモリアクセスの方法について効率のよい方法を追及していく.

GPGPUにおけるシストリックアルゴリズムの効率的な実現について

*法政大学　理工学研究科　応用情報工学専攻
†法政大学　理工学部　応用情報工学科
‡大阪府立大学　理学系研究科　情報数理科学専攻

茂木啓輔*　和田幸一†　藤本典幸‡

目次

- はじめに
- 研究の目的
- GPUのアーキテクチャ
- 理論モデルについて
- シストリックアルゴリズムとは
- GPU上でのシストリックアルゴリズムの実装方法
- 今後の展望

はじめに

- GPUとは,3Dグラフィックスの表示に必要な計算処理を行う半導体チップ.
- GPGPUとは,本来画像処理を専門とする演算装置であるGPUを画像処理以外の汎用的な目的に応用する技術である.

研究の目的

シストリックアルゴリズムをGPUを用いて実装する手法を考案

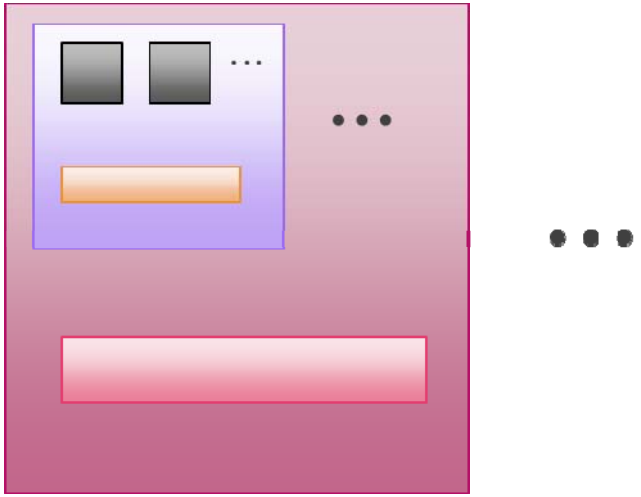
↓

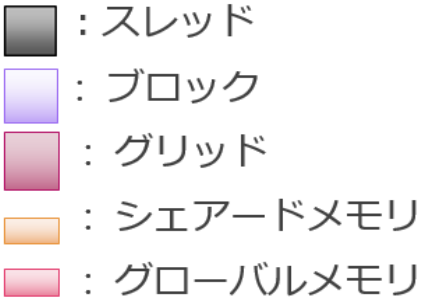
理論的な評価と実際の結果とを比較

↓

より効率的なシストリックアルゴリズムの実装方法の提案

GPUのアーキテクチャ





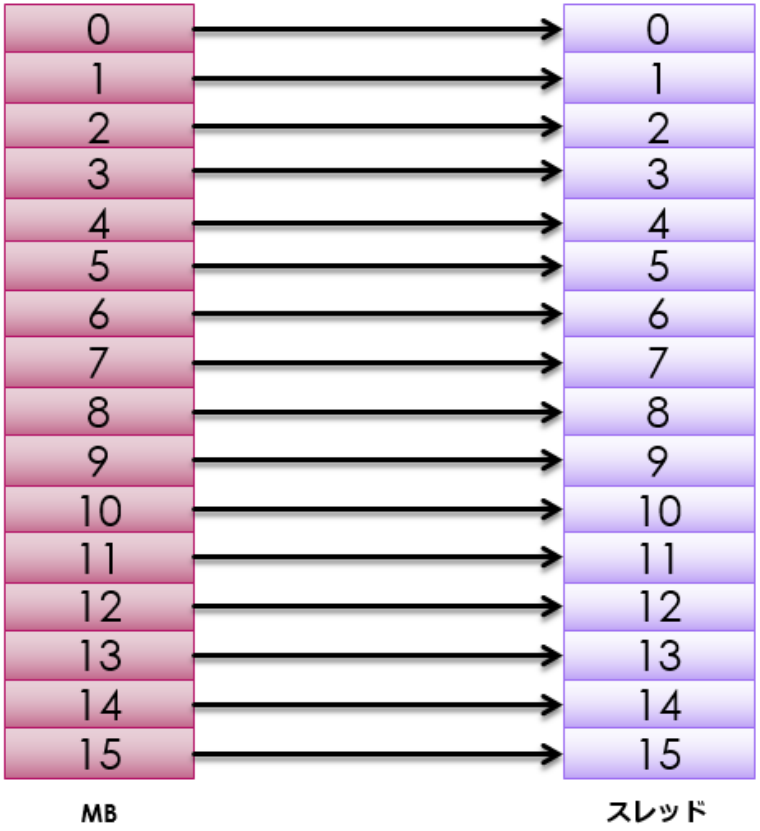
- GPUは,グリッド,ブロック(MP:マルチプロセッサ),スレッドの三階層の構成になっている.各命令はスレッド単位で実行される.
- これらはカーネル関数を実行する際に指定する.

グローバルメモリとシェアードメモリ

- GPUのメモリには2種類のメモリがある.

	グローバルメモリ	シェアードメモリ
メモリの場所	オフチップ	オンチップ
容量	1.5Gbyte	16-64Kbyte
アクセス時間	400-600clockcycle	4-6clockcycle

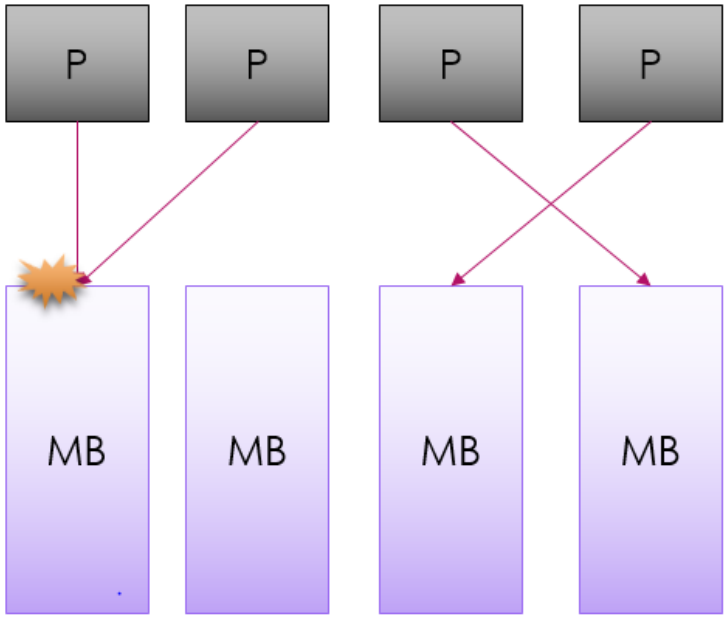
コアレッシング



- グローバルメモリの特性で,連続するデータにプロセッサがアクセスするとき,32バイト,64バイト,128バイトのまとまったデータ量を転送する事である.またハーフワープ(16スレッド)で転送される.

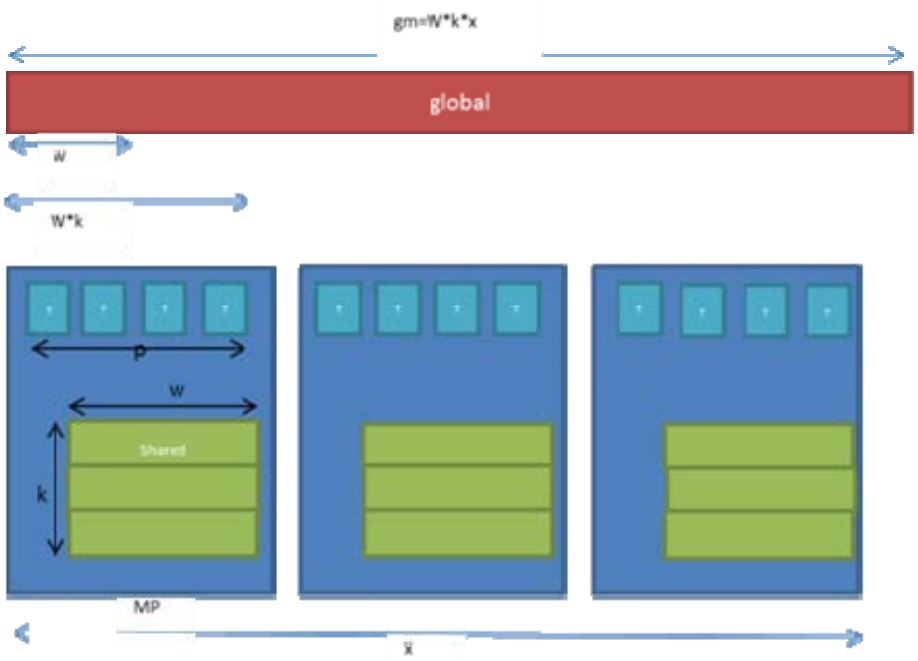
- 1ワード4バイトの時,64バイト単位
  1ワード8バイトの時,128バイト単位
  1ワード16バイトの時,128バイト単位で2回

バンクコンフリクト



- シェアードメモリの特性で,スレッドがシェアードメモリの同じメモリバンクへアクセスを行った際に,シーケンシャルに処理されてしまう.
- これを防ぐプログラミングが必要.

評価に使用する理論モデル



	各パラメータの説明
k	1バンク内のメモリ数
x	MP数
Lgm	グローバルメモリのレイテンシ
Lsm	シェアードメモリのレイテンシ
w	シェアードメモリのバンク数
wg=2*p	グローバルメモリのバンク数
gm=N=w*k*x	グローバルメモリ要素数
sm=w*k	シェアードメモリの要素数
N	入力要素数

評価に使用する理論モデルのポイント

- 本研究で使用する理論モデルは,グローバルメモリのコアレッシングとシェアードメモリのバンクコンフリクトの特徴を踏まえて二段階の構成にしている.

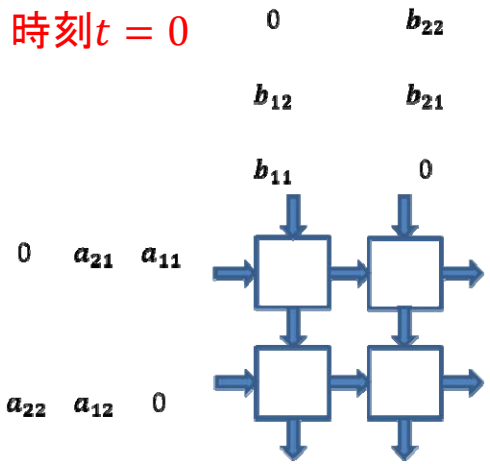
- グローバルメモリ,シェアードメモリへのアクセス時間を区別するために,Lgm,Lsmの2種類の独自のパラメータを設定した.

シストリックアルゴリズムとは

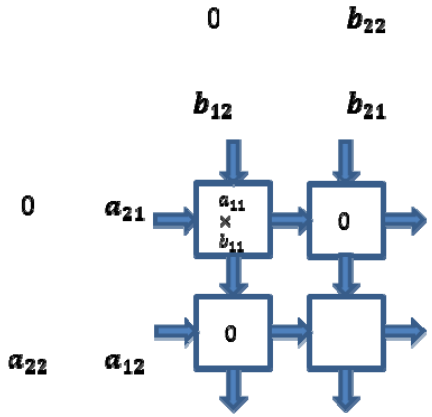
- シストリックアルゴリズムとは一般的に,”多数の同一構造の基本演算器(セル)が1次元または2次元アレイ状に規則的に配置されたシステム”のことを示す.
- このアルゴリズムでは,データの流れが規則正しく波となって一列に進んでいく.

ネットワークの動作例(2*2の行列積)

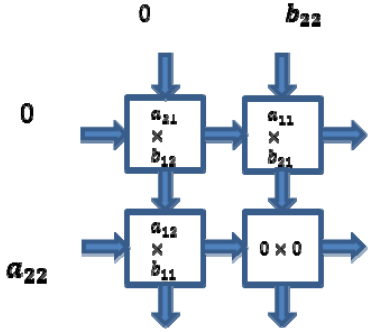
時刻$t=0$



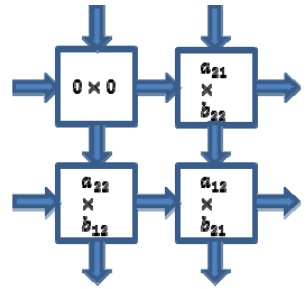
時刻$t=1$



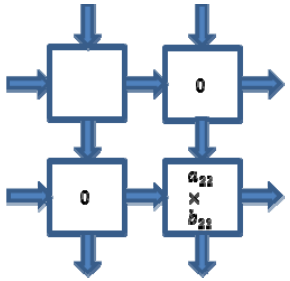
時刻$t=2$



時刻$t=3$



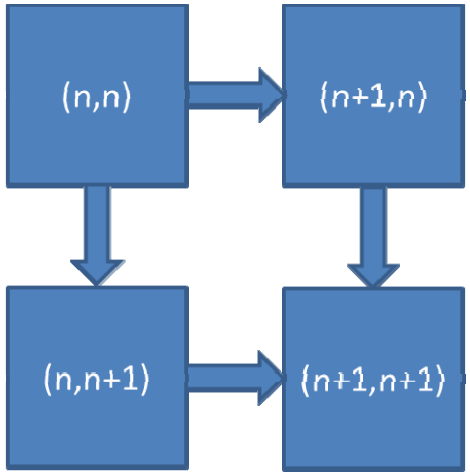
時刻$t=4$



シストリックアルゴリズムの特徴

特徴①

- 近接したモジュール(セル)間の相互結合を有効に使うという点で,非常に良い並列アルゴリズムである.

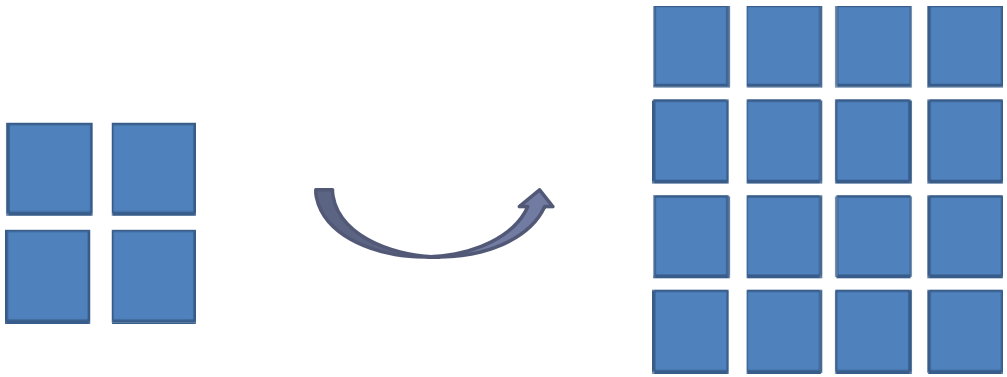


規則的な入出力により,効率的なメモリアクセスが可能になるのではないか?

シストリックアルゴリズムの特徴

特徴②

- 問題のサイズに応じて,規則的に拡張することが出来る.

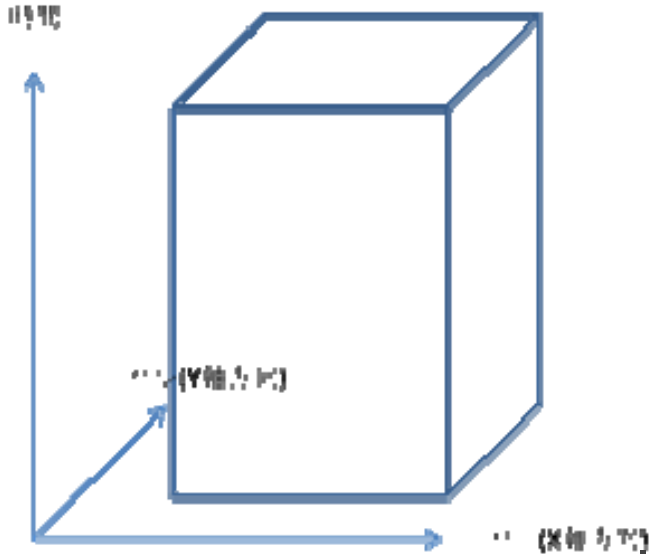


統一的にネットワークを記述することが出来る.

シストリックアルゴリズムの特徴

特徴③

- 時刻とセルを指定することで,処理を行う部分を決定することが出来る.

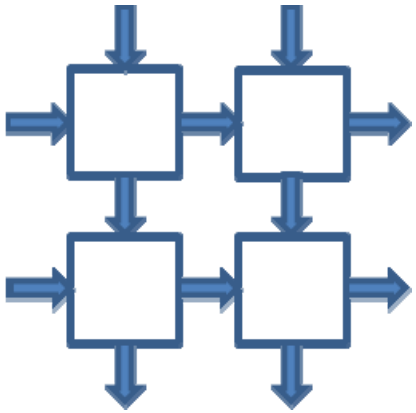


GPU上で実装する場合,
①セル単位で演算
②時刻単位で演算
2つの方法が考えられる.

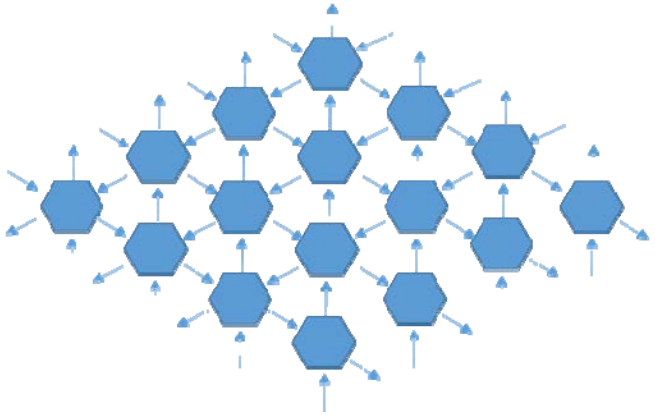
ネットワークの種類

- 本研究ではGPU上で以下の二種類のネットワークの実装を検討している.

①二次元のネットワーク

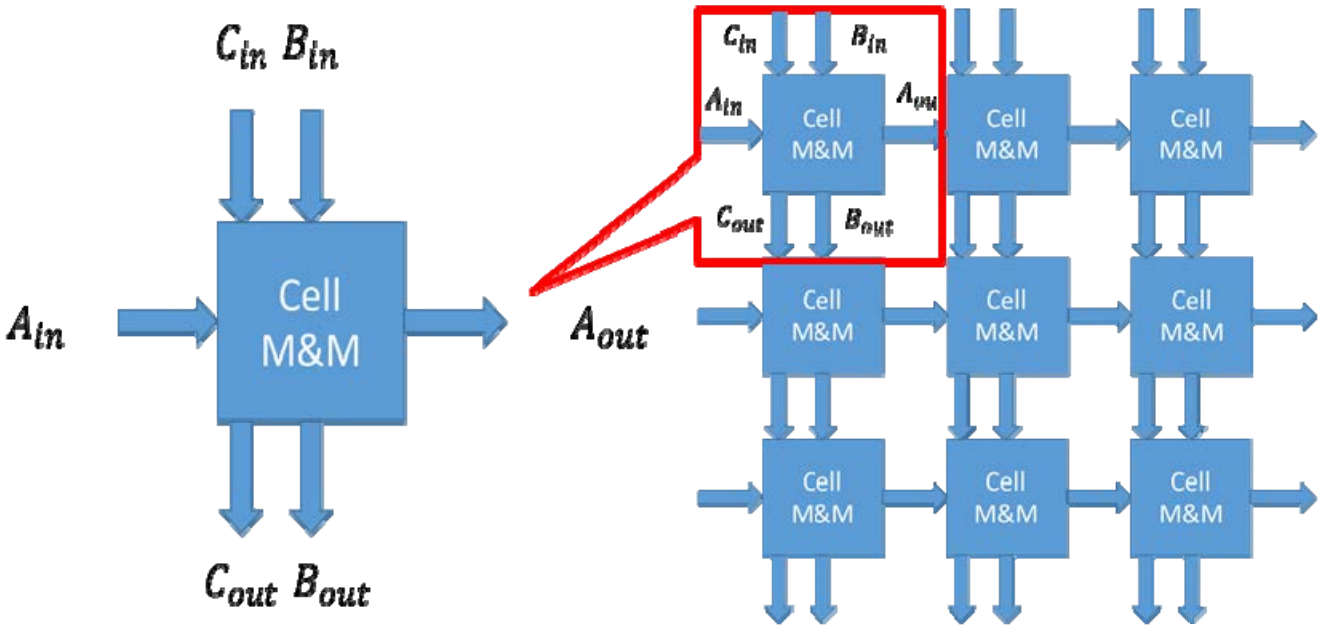


②六角接続のネットワーク



ネットワークの記述方法

- 以下のように記述することで,GPU上でシストリックアルゴリズムを実現する.



ここでは引き続き,二次元のネットワークで行列積を求めるものを例として示す

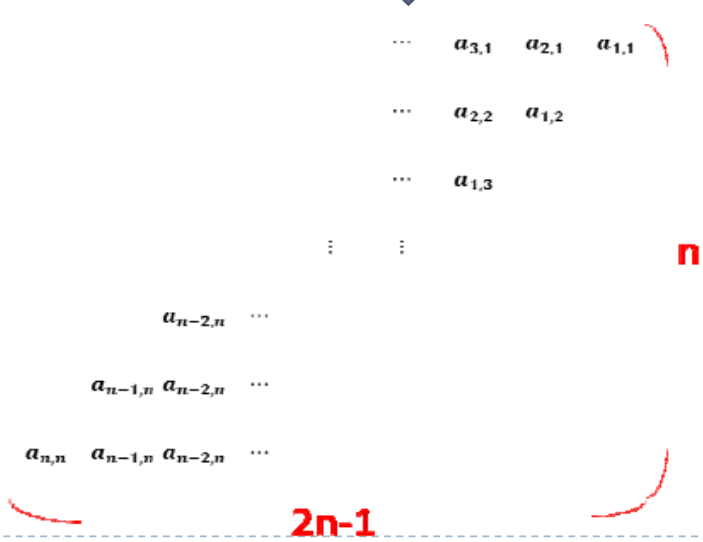
ネットワークの記述方法

- 時間と問題の記述

時刻	**t** $t \geq 0$
問題(行列)のサイズ	n n>0
横からの入力行列	a
上からの入力行列	b
結果として出力される行列	c
行列の行番号	i $1 \leq i \leq n$
行列の列番号	j $1 \leq j \leq n$
横からの入力のタイミング	$a_{i,j}$ $M_{\&}M(1,j)$ for t = 1,2n-1 $a_{1,t+j-1}$
上からの入力のタイミング	$b_{i,j}$ $M_{\&}M(i,1)$ for t = 1,2n-1 $b_{t+1-1,1}$

例:横方向からの入力行列

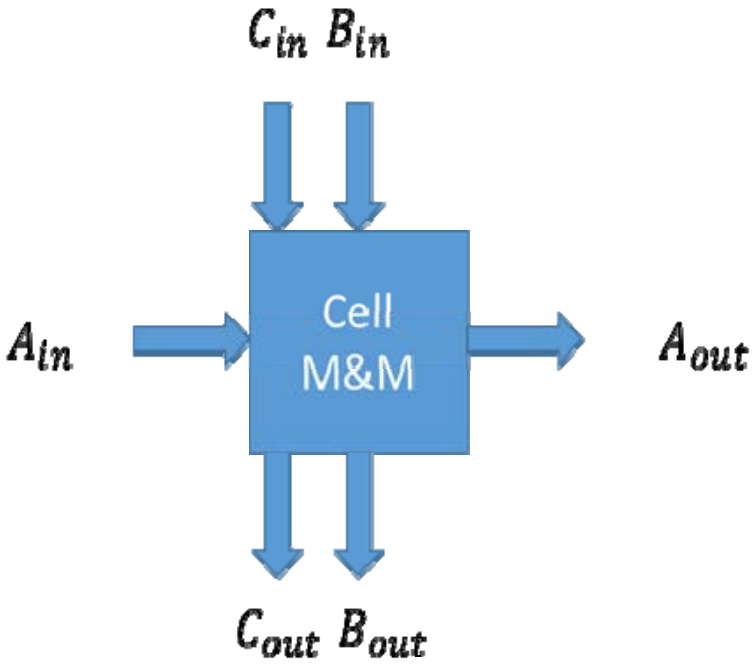
$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nn} \end{pmatrix}$$



ネットワークの記述方法

- セルの記述

セルの名前	$M_{\&}M(int\ x, y)$
上からの入力	A_{in}
左からの入力	B_{in}
下への出力	A_{out}
右への出力	B_{out}
セルでの演算結果の入力	C_{in}
セルでの演算結果の出力	C_{out}
セルのx座標	x $0 \leq x \leq n-1$
セルのy座標	y $0 \leq y \leq n-1$
セル内での演算	$A_{out}(t+1) = A_{in}(t)$
セル内での演算	$B_{out}(t+1) = B_{in}(t)$
セル内での演算	$C_{out}(t+1) = A_{in}(t) * B_{in}(t) + C_{in}(t)$

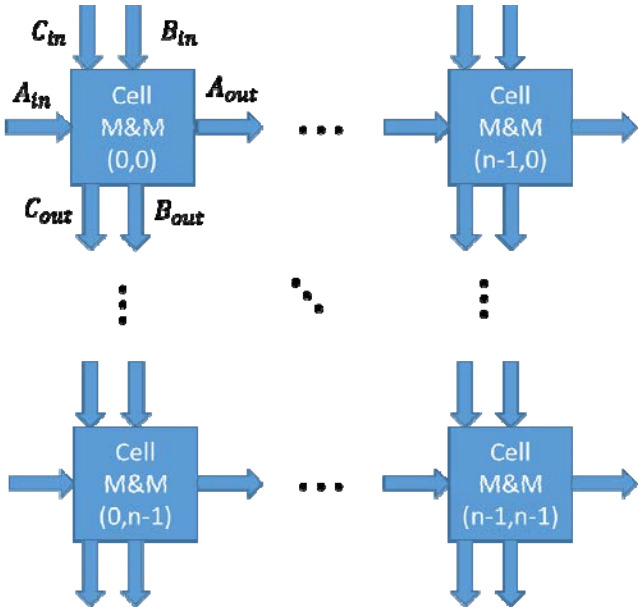


ネットワークの記述方法

- ネットワークの接続関係の記述

for x = 0,n-1 do
　　for y = 0,n-2 do
　　$M_{\&}M(x, y+1)B_{in}(t) \Leftrightarrow M_{\&}M(x, y)B_{out}(t)$

for y = 0,n-1 do
　　for x = 0,n-2 do
　　$M_{\&}M(x+1, y)A_{in}(t) \Leftrightarrow M_{\&}M(x, y)A_{out}(t)$



GPU上での実装方法

- シストリックアルゴリズムをGPU上で実装する際には,以下の3つの方法が考えられる.

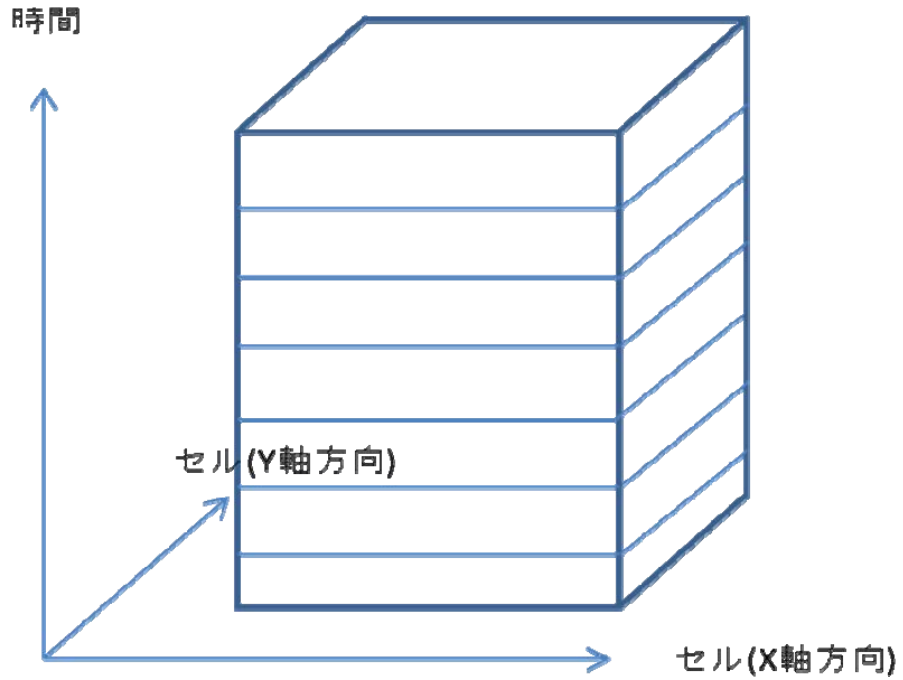
方法①:先ほどの記述を元に,1単位時刻分の演算を行い,入力の大きさに応じて繰り返し演算を行なう

方法②:①の拡張版で複数単位時刻をまとめて演算する

方法③:1単位時刻分の演算をDPを用いて分割して演算を行い,入力の大きさに応じて繰り返し演算を行なう

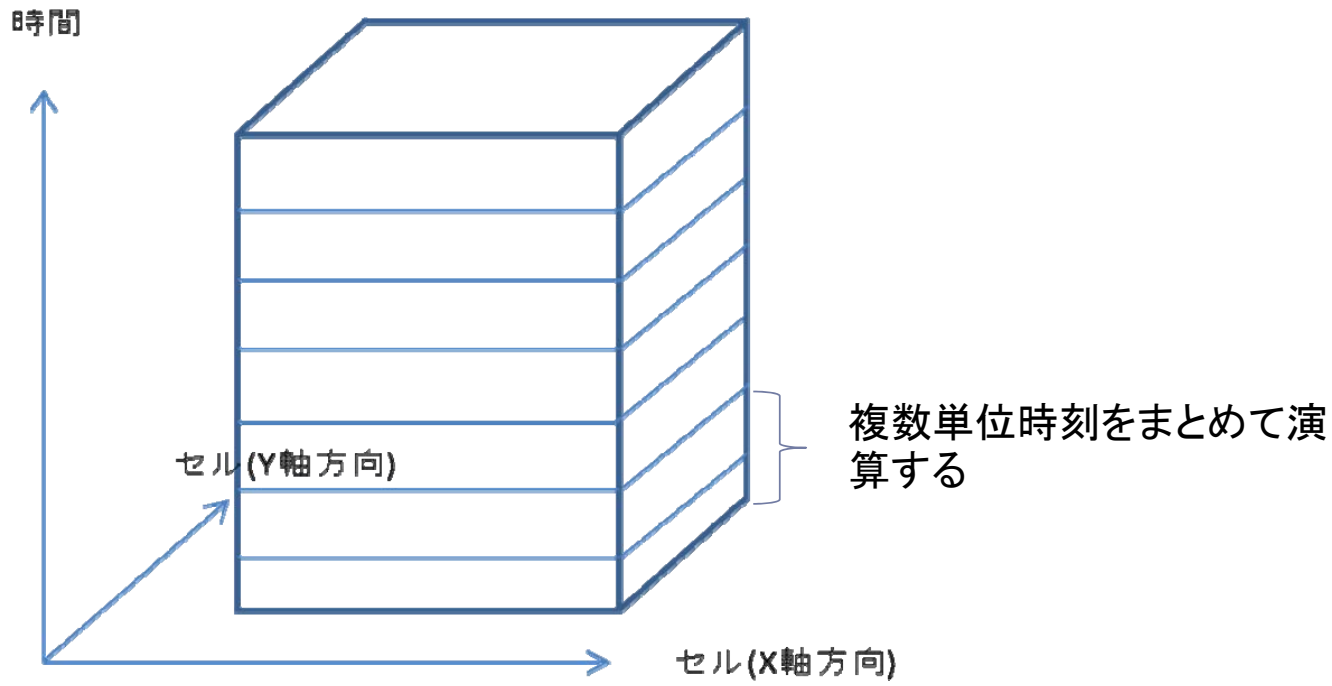
GPU上での実装方法

方法①：先ほどの記述を元に，1単位時刻分の演算を行い，入力の大きさに応じて繰り返し演算を行なう



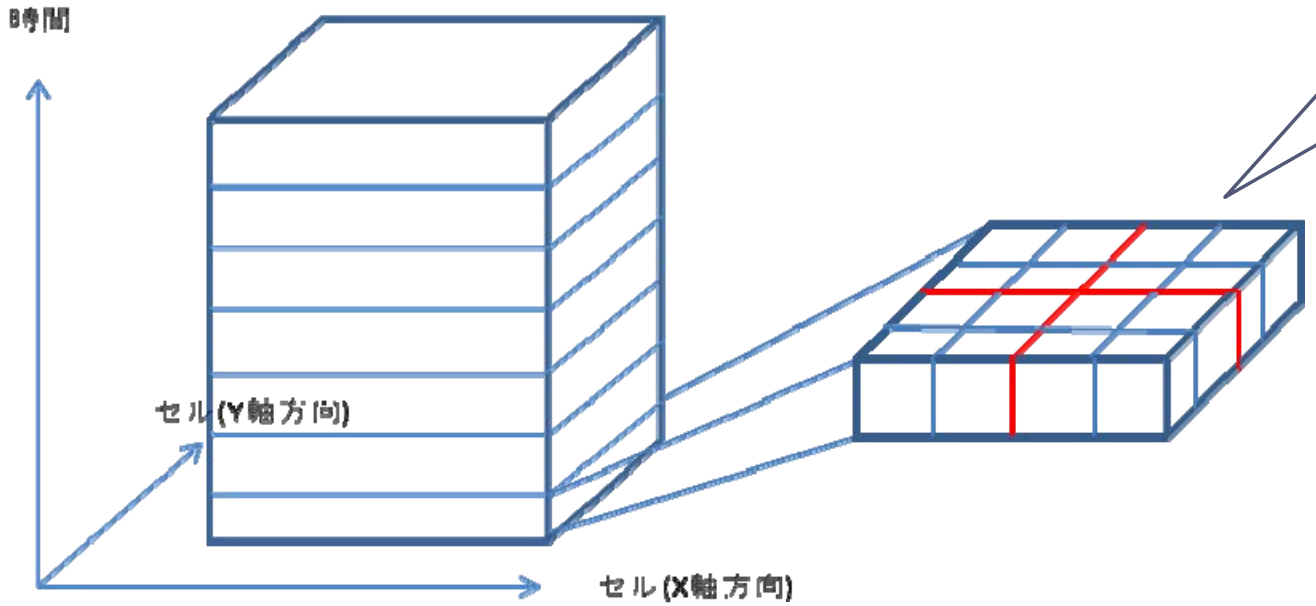
GPU上での実装方法

方法②:①の拡張版で複数単位時刻をまとめて演算する



GPU上での実装方法

方法③:DP(dynamic parallelism)を用いて行う



分割した際の境界となる部分の要素を重複して演算してしまう

様々な分割の方法を検討中

今後の展望

任意のシストリックアルゴリズムをGPU上で実現できるような手法の提案を行う

＊問題のサイズとネットワークの種類を入力として与えるだけで,自動的にネットワークを生成し,実行を行えるようなシステムの実装
＊シストリックアルゴリズムを理論モデル上でシミュレートし,実際の実装結果との比較を行う

GPU上での効率的なシストリックアルゴリズムの実装方法の提案を行う

最後に

- ご清聴ありがとうございました.

セッション６

計算理論２

GPU向け高速Tree Reductionアルゴリズム

小池 敦[1,a)]　定兼 邦彦[2,b)]

概要：GPU を用いた Tree reduction アルゴリズムを扱う．Tree reduction は並列計算における最も基本的な問題の一つであり，四則演算を用いて書かれた数式の並列評価や木に対する各種クエリ等の多くの問題の一般化となっている．GPU 向けアルゴリズムを設計する際はマルチプロセッサの SIMD アーキテクチャを考慮する必要があるが，木構造に対する処理では，一般に参照データが入力に大きく依存するため，GPU アーキテクチャを有効に活用することは難しい．しかし，本論文では，GPU アーキテクチャを適切に考慮し，高速に Tree reduction を計算するアルゴリズムを提案する．提案アルゴリズムは木の形に依存せず I/O 計算量が最適となる．

1. はじめに

プロセッサの動作クロック周波数の向上は限界を迎えており，周波数向上に代わるパフォーマンス向上の手段として並列アーキテクチャが注目されている．GPU (Graphics Processing Unit) は元々はグラフィック処理のための専用プロセッサとして開発された．しかし，非常に高い並列性を持っていることから，グラフィック処理以外にも GPU が使われ始めている．汎用の処理に GPU を使用することは GPGPU (general-purpose GPU) と呼ばれており，安価に超並列環境が構築できることから注目されている．

GPU は多数のコアを用いて効率よく処理を行うため，特殊なアーキテクチャとなっている [1]．GPU プログラミングにおいては，このアーキテクチャを適切に考慮する必要がある．NVIDIA 社は GPGPU のための開発環境として，CUDA [2] を提供しており，CUDA 上で開発することにより，様々な GPU モデル上で動作するプログラムを実装することができる．しかし，最適なパフォーマンスを得るためには，GPU アーキテクチャを適切に考慮してアルゴリズムを設計する必要がある．GPU マルチプロセッサは SIMD アーキテクチャとなっており，マルチプロセッサ内のすべてのコアが常に同一の命令を実行している．SIMD コアによるメモリアクセスは，特定の規則でアクセスする場合に効率的になるため，入力に依らずに規則的にメモリアクセスできるようにアルゴリズムを設計する必要がある．筆者らは GPU アルゴリズムに対し計算量の漸近解析を行うためのモデルとして，AGPU モデルを提案している [3]．

Tree reduction は並列計算における最も基本的な問題の一つである．木に対する各種クエリ，数字と算術演算子を用いた数式の計算などを含む多くの問題が Tree reduction に一般化され [4], [5], [6]，また，コンパイラ最適化 [7] やメッセージの復号 [8] 等多くの問題で活用されている [9], [10]．Miller と Reif [5] は Tree reduction を計算するための Parallel tree contraction アルゴリズムを提案した．その後も様々な研究が行われている [10], [11], [12], [13], [14]．また，古典的な並列計算の教科書 [15] にも多くの研究成果が紹介されている．

入力が行きがけ順 (preorder) に直列化（シリアライズ）されている場合の Tree reduction も重要であり，この場合のアルゴリズムは Gibbons と Rytter [16] により提案されている．XML データを始めとする多くのデータがこの方式でシリアライズされている．本論文ではこのデータ構造を XML 表現と呼ぶ．近年，非構造データを柔軟に扱うことができるデータベースとして XML データベースが注目されている．XML データベースに対するクエリ言語として XQuery [17] があり，クエリ処理の高速化が求められている．なお，シーケンシャルアルゴリズムについては，スタックを用いてシリアライズデータを 1 回スキャンすることで算出できることが知られている [16]．また，木に対する基本的なクエリの幾つか（木の高さ等）は，このデータ構造に対して並列 Prefix sum を行う時間で解けることが知られている [13]．

Tree reduction を効率的に並列計算するために制約をつけることを考える．Matsuzaki ら [19], [20] は拡張分配則（Extended distributivity）という制約を追加した場合の

[1] 総合研究大学院大学 複合科学研究科 情報学専攻
[2] 東京大学 大学院 情報理工学系研究科
[a)] koike@nii.ac.jp
[b)] sada@mist.i.u-tokyo.ac.jp

Tree reduction について提案しており，Kakehi ら [21] は BSP モデル [22] 上での XML 表現に対する Tree reduction に関して，拡張分配則を用いた高速アルゴリズムを提案した．計算量は $\mathcal{O}(n/p+p)$ である．また，Emoto, Imachi [23] は上記アルゴリズムを Map-Reduce モデル [24] 向けに改良した．一方，Morihata, Matsuzaki [14] は Tree reduction 計算のための別の制約として，部分縮約則を提案している．この制約を追加しても，なお多くのクエリがカバーされ（詳細は後述する），効率の良い並列計算が可能となる．

本論文では，Morihata, Matsuzaki[14] による部分縮約則を活用し，XML 表現に対する Tree reduction を GPU 上で高速計算するアルゴリズムを提案する．上記の BSP モデルや Map Reduce モデル上でのアルゴリズムでは，各コアは与えられた入力に対し，個別に計算を行うが，GPU マルチプロセッサは SIMD アーキテクチャのため，内部のコアは個別に計算を行うことはできない．すなわち，マルチプロセッサ内のコアは常に同一の命令を実行する必要がある．特にメモリアクセスについては，アクセスするデータアドレスがコアごとに入力に依存し大きく変わるため，GPU の効率的なメモリアクセスの仕組み（コアレッシング等）を活用することは難しい．本論文では上記の問題を解決し，高速なアルゴリズムを設計する．提案アルゴリズムは木の形に依存せずに AGPU 上で I/O 計算量が最適となる．AGPU(p, b, M) モデル上で n 要素に対して Tree reduction を行う時，時間計算量は $\mathcal{O}\left(\frac{n \log b}{p}\right)$，I/O 計算量は $\mathcal{O}(n/b)$ である．

2. 準備

2.1 AGPU モデル

AGPU モデルは，GPU 向けアルゴリズムの計算量の漸近解析を行うための並列計算モデルである．AGPU モデルを用いることで，GPU デバイスの詳細仕様に依らない汎用的なアルゴリズム設計と評価を行うことができる．まず，AGPU モデルのアーキテクチャを説明した後，GPU 向けアルゴリズムの評価基準について説明する．

2.1.1 アーキテクチャ

AGPU モデルのアーキテクチャを図 1 に示す．AGPU モデルのアーキテクチャは並列計算を行うためのデバイス (GPU) とデバイスを制御するためのホスト (CPU) の異種混載システムとなっている．デバイスは p 個のコアを備えている．コアのワード長は w ビットであり，コアはワード単位でデータにアクセスする．また，デバイスは k 個のマルチプロセッサで構成されており，各マルチプロセッサは b 個のコアを備えている．すなわち $p = kb$ である．マルチプロセッサはホストから起動されたプログラムを個別に実行する．すなわち，マルチプロセッサは他のマルチプロセッサとの通信手段および同期手段を持たない．ホストはすべてのマルチプロセッサの処理完了を待つことにより，

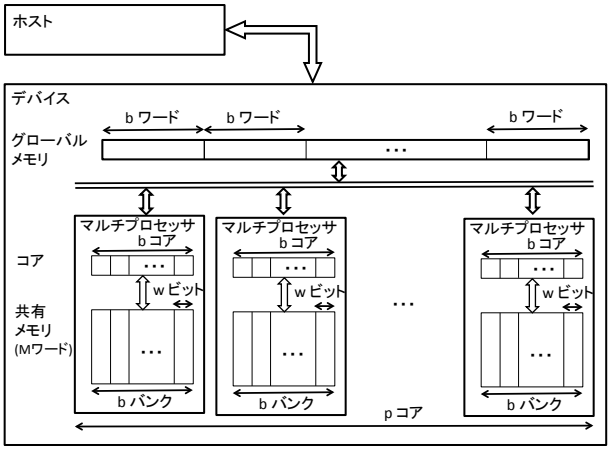


図 1　AGPU モデルのアーキテクチャ

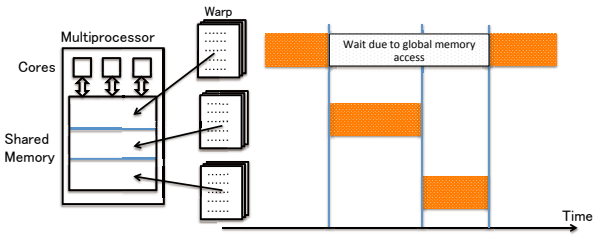


図 2　マルチスレッディングによるレイテンシ隠ぺいの例

マルチプロセッサ間の同期を行うことができる．しかし，マルチプロセッサの処理完了時，共有メモリのデータはすべて削除される．後で参照する必要があるデータはマルチプロセッサの処理終了時にすべてグローバルメモリに書き込む必要がある．

マルチプロセッサ内の b 個のコアは b 個のスレッドに対し，常に同一の命令を実行する．この時，各コアは同一命令を並列に実行するという．1 つのマルチプロセッサ内で並列に処理されるスレッドの集合をワープと呼ぶ．ただし，オペランドに指定されるデータアドレスについてはコアごとに指定することができる．また，命令には実行条件を含めることができ，条件を満たすコアのみ命令を実行させることができる．一方，各コアは複数のスレッドを時分割で切り替えながら同時実行することができる．この時，各コアは複数スレッドを並行に実行するという．言い換えれば，マルチプロセッサは複数ワープを並行に実行することができる．GPU のこの機能をマルチスレッディングと呼ぶ．マルチスレッディングにはグローバルメモリアクセスのレイテンシ（待ち時間）を隠ぺいする効果がある．すなわち，あるワープがグローバルメモリアクセスにより，待ち状態になっている場合に，マルチプロセッサは他のワープを実行することによりコアの使用率を高めることができる．図 2 に具体例を示す．マルチスレッディングは GPU における効率的なメモリアクセスのキーとなる技術である．

デバイスは 2 種類のメモリを備えている．1 つ目はグローバルメモリである．これは低速であるが大容量であ

り，すべてのマルチプロセッサおよびホストからアクセス可能である．グローバルメモリは b ワードごとのブロックに分割されている．同一命令を実行するマルチプロセッサ内の全コアが同一ブロックにアクセスする時，1 回のメモリアクセスで全コア分のデータにアクセスすることができる．これはコアレッシングと呼ばれており，処理時間に大きな影響を与える．一方，コアが複数の異なるブロックにアクセスする時は，各ブロックのに対して 1 回のアクセスが必要となる．2 つ目は共有メモリである．各マルチプロセッサは内部に容量 M ワード ($b \leq M$) の共有メモリを備えている．これは高速であるが低容量である．また，マルチプロセッサ内部のコアからのみアクセス可能である．共有メモリは b 個のバンクから構成される，同一命令を実行する b 個のコアのそれぞれが異なるバンクにアクセスする時，単位時間でデータにアクセスできる．複数のコアが同一のバンクにアクセスする時は，処理がシリアライズされる．これはバンクコンフリクトと呼ばれており，これも処理時間に大きな影響を与える．以上で定義される計算モデルを AGPU(p, b, M, w) と記載する．ただし M, w については，省略される場合がある．

2.1.2 アルゴリズムの評価基準

まず，アルゴリズムの実行時間を評価する基準として，時間計算量と I/O 計算量を使用する．時間計算量は，各マルチプロセッサで実行されるプログラムの命令発行数である．共有メモリへのアクセスでバンクコンフリクトが発生する場合，コンフリクト数に応じた時間が時間計算量に加算される．また，グローバルメモリへのアクセスについては，b ワードのブロックに対する書き込みまたは読み込みの時間計算量を 1 とする．マルチプロセッサごとに命令発行数が異なる場合には，最も多い発行数を時間計算量とする．I/O 計算量については，上記で説明したグローバルメモリアクセス回数のすべてのマルチプロセッサでの合計値とする．I/O 計算量を時間計算量とは別に評価する理由は，グローバルメモリアクセス処理に要する時間が他の処理に比べて大きくなるためである．また，グローバルメモリに対して同時にアクセス可能なマルチプロセッサ数が限られているため，アクセス回数については，すべてのマルチプロセッサでの合計値とする．

次に，メモリ使用量を評価する基準として，グローバルメモリ使用量と共有メモリ使用量を使用する．使用される単位はビットである．また，共有メモリ使用量は各マルチプロセッサで使用されるメモリ使用量の最大値とする．大規模データを扱う場合，グローバルメモリ使用量を少なくすることは特に重要である．また，共有メモリ使用量は M ワード以下にする必要がある．また，共有メモリ使用量は以下で説明する多重度にも影響する．

2.1.1 節で述べた通り，マルチスレッディングは GPU におけるメモリアクセスのキーとなる技術である．しかし，I/O 計算量の値はマルチスレッディングの効果とは無関係であるため，マルチスレッディングの効果を I/O 計算量を用いて評価することはできない．そこでマルチスレッディングの効果を評価する値として多重度を導入する．

マルチスレッディングの効率を上げるためには，マルチプロセッサに割り当てるワープ数を増やせば良い．しかし，割当ワープ数は共有メモリ使用量に依存する．マルチプロセッサ内のすべてのワープは同一の共有メモリを使用するため，全ワープでの共有メモリ使用量の合計値が共有メモリサイズを超えることはできない．AGPU(p, b, M) 上で設計されたアルゴリズムについて，各ワープの共有メモリ使用量を m とすると，多重度 $\mathcal{M}$ は $\mathcal{M} := M/m$ と定義される．これは CUDA のオキュパンシに対応する値であるが，多重度は AGPU モデルのパラメータを使用して計算することができる．共有メモリ使用量が大きい場合，多重度の値は小さくなり，マルチスレッディングによるレイテンシ隠蔽の効果が小さくなる．

2.2 Tree reduction

根付き順序木上の頂点 v に対して，以下の関数を定義する．

$$f(v, \otimes, \oplus, h) = \begin{cases} h(w_v) & (v \text{ が葉}) \\ w_v \otimes \left(\bigoplus_{u \in S(v)} f(u, \otimes, \oplus, h) \right) & (\text{その他}) \end{cases}$$

ここで，$S(v)$ は v の子の集合を表し，w_v は頂点 v の重みを表す．$\otimes, \oplus$ は 2 つの引数を取る関数であり，$x \otimes y$ は $\otimes(x, y)$ を意味し，$x \oplus y$ は $\oplus(x, y)$ を意味する．h は 1 つの引数を取る関数である．これらは入力として与えられる．また，$\bigoplus$ は以下のような計算を行う．

$$\bigoplus_{i=0}^{n-1} T[i] = T[0] \oplus T[1] \oplus \cdots \oplus T[n-1].$$

この時，木の根 r に対して，$f(r, \otimes, \oplus, h)$ を算出する計算を Tree reduction と呼ぶ．本論文では，$\oplus$ は結合則を満たすものとする．$\otimes$ は指定する場合以外は結合則を満たしている必要はなく，また，$\otimes, \oplus$ は可換である必要もない．また，$\otimes, \oplus$ は単位元を持っていると仮定する．$\otimes$ または $\oplus$ が単位元を持たない場合は単位元を添加する．また，関数 f は基本型もしくは基本型を要素とする tuple を返すものとする．すなわち，関数 f はリストを含むデータを返すことはできない．また，関数 $\otimes, \oplus, h$ は入力値によらずに定数時間で計算できるものとする．

図 3 の木に対して，Tree reduction 計算する例をいくつか挙げる．

例.1 (sum)

木の全頂点の重みの合計を求める．この時，関数 $\otimes, \oplus, h$ を以下のように指定して Tree reduction を計算すれば，所

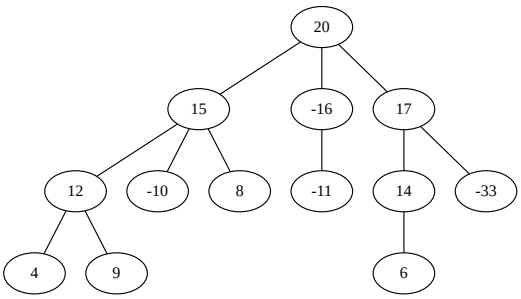


図 **3**　木の具体例

望の値 37 が得られる．

$$\otimes(x,y) = x+y$$
$$\oplus(x,y) = x+y$$
$$h(x) = x$$

例.1’ (count)

頂点の重みが 10 以上の頂点数を求める．この場合，頂点重みをあらかじめ以下の関数を用いて変換しておけば，例.1 と同様に扱うことができ，所望の値 5 が得られる．当然，事前に上記の変換を行う代わりに，頂点重みを取得する際に，常に以下の関数の戻り値を取得するようにしてもよい．

$$g(w_v) = \begin{cases} 0 & \text{if } w_v < 10, \\ 1 & \text{otherwise} \end{cases}$$

本論文では，以下，このような場合は例.1 と同様に扱う．

例.2 (max path weight)

根から葉までのパスのうち，パスコスト（パス上の頂点の重みの合計）が最大のパスのパスコストを求める．この時，関数 $\otimes, \oplus, h$ を以下のように指定して Tree reduction を計算すれば，所望の値 57 が得られる．対応するパスは根から重み 6 の葉までのパスである．

$$\otimes(x,y) = x+y$$
$$\oplus(x,y) = \max(x,y)$$
$$h(x) = x$$

例.3 (max subtree)

木に対するすべての部分木（ある頂点とその頂点のすべての子孫からなる木）のうち，頂点コストの和が最大になるもののコストを求める．この時，関数 $\otimes, \oplus, h$ を以下のように指定して Tree reduction を計算すれば，出力されるペアの第一項が所望の値 38 となる．対応する subtree は重み 15 の頂点を根とする subtree である．

$$\otimes(x,t) = (\max(x+s_t, m_t), x+s_t)$$
$$\text{(ただし } t=(m_t,s_t)\text{)}$$

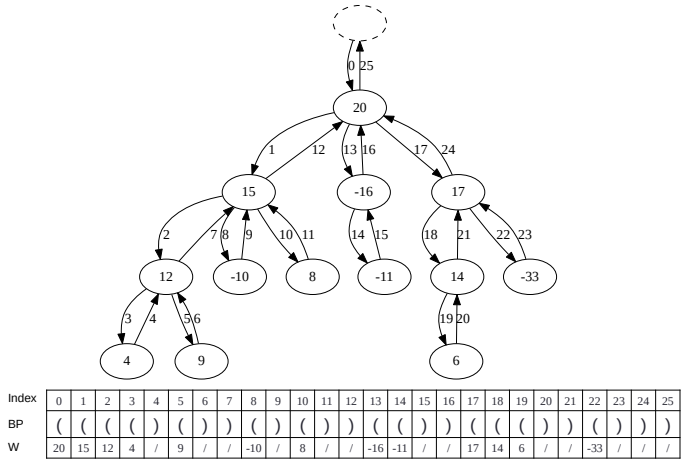


図 **4**　図 3 の BP 表現と XML 表現 (配列 W)

$$\oplus(t,u) = (\max(m_t,m_u), s_t+s_u)$$
$$\text{(ただし } t=(m_t,s_t), u=(m_u,s_u)\text{)}$$
$$h(x) = (x,x)$$

これらの例以外でも木に対する様々なクエリが Tree reduction で表せる [4], [5], [6], [15], [20].

2.3　XML 表現

木の直列化表現として，BP 表現と XML 表現を説明する．木を preorder で探索 (traverse) する時，下向き探索時には開きカッコ (を出力し，上向き探索時には閉じカッコ) を出力したものが BP 表現である．図 3 の木に対する BP 表現を図 4 に示す．元の木に対して，根の親にダミーの頂点を追加し，各枝を両向きの 2 つの有向枝に入れ替えると，BP 表現の各 index と有向枝は 1 対 1 対応する．図 4 の木の有向枝には，対応する index を記載した．ある index に対応する頂点とは，下向き探索時は探索の終点頂点，上向き探索時は探索の始点頂点のことを指す（このようにすることで開きカッコと対応する閉じカッコが同じ頂点に対応するようになる）．BP 表現は根付きの ordered tree の構造を保持できるが，重みについては保持できない．本論文で使用する XML 表現は以下のように定義される．BP 表現で開きカッコの index には対応する頂点の重みを格納し，閉じカッコに対応する index には / を格納する．図 4 の配列 W が図 3 の XML 表現である．これは木を preorder でシリアライズする際に一般的に用いられる定義である [23]. XML 表現は頂点数の 2 倍の数の配列を用いて木の構造と頂点の重みの両方を保持できる．

XML 表現に対する Tree reduction に関して，シーケンシャルアルゴリズムについてはスタックを用いて入力を 1 回スキャンすることで求められることが知られている [16]. また，Gibbons, Rytter により PRAM 上の並列アルゴリズムが提案されている [16].

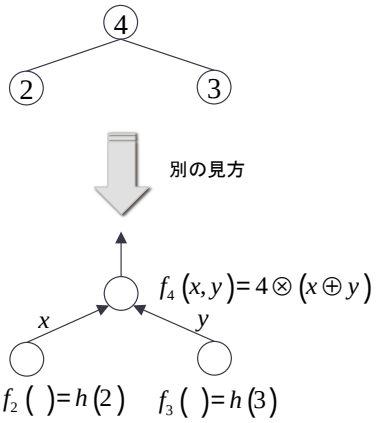


図 5　Tree reduction の回路図表現の例

2.4　Tree reduction の回路図表現

以下の説明のために，まず，Tree reduction の回路図表現について説明する．

Tree reduction の式を見ると，ある頂点の値を計算するために参照するのは，その子の値のみであり，関数の返り値は親頂点にのみ使用される．これを踏まえると，Tree reduction 計算を別の見方でみることができる．簡単な例を図 5 に示す．葉を除く各頂点は任意の数の子を入力とし，出力（Tree reduction 定義の 2 行目の計算結果）を親に送る回路であると考える．葉は常に定数値を返す回路である（Tree reduction 定義の 1 行目に相当する）．このような表現を本論文では回路図表現と呼ぶことにする．本論文では今後常に Tree reduction をこのように解釈する．

次に回路の変換について説明する．例えば，$\otimes, \oplus$ が結合的であり，$\otimes$ が $\oplus$ に対して分配則を満たす時，図 6 のような変換が可能である．Tree reduction を定義に従って計算する場合，葉から根の方向に順に計算していく必要があるが，上記の場合は，内部頂点に対して並列に回路変換を行うことで木の頂点数を減らすことができるため，効率的な並列計算が可能となる．

四則演算による算術計算も回路図表現が可能である．$(4\times5)+(6\times7)$ に対応する回路図表現を図 7 に示す．

2.5　XML 表現に対する Tree reduction の先行研究：拡張分配則（Extended distributivity）

Tree reduction を効率的に解くための制約として，拡張分配則を説明する．

定義 2.1 (拡張分配則 [19], [20])　$\oplus$ は結合的とする．また，$f_0(x)=a_0\otimes(b_0\oplus x\oplus c_0)$ とし，$f_1(x)=a_1\otimes(b_1\oplus x\oplus c_1)$ とする．この時，ある関数 p_1,p_2,p_3 が存在し，任意の a_0,b_0,c_0,a_1,b_1,c_1 に対し，以下が成り立つ時．$\otimes$ は $\oplus$ に対して，拡張分配則を満たすという．

$$f_0\circ f_1(x) = f(x)$$
$$f(x) = a\otimes(b\oplus x\oplus c)$$

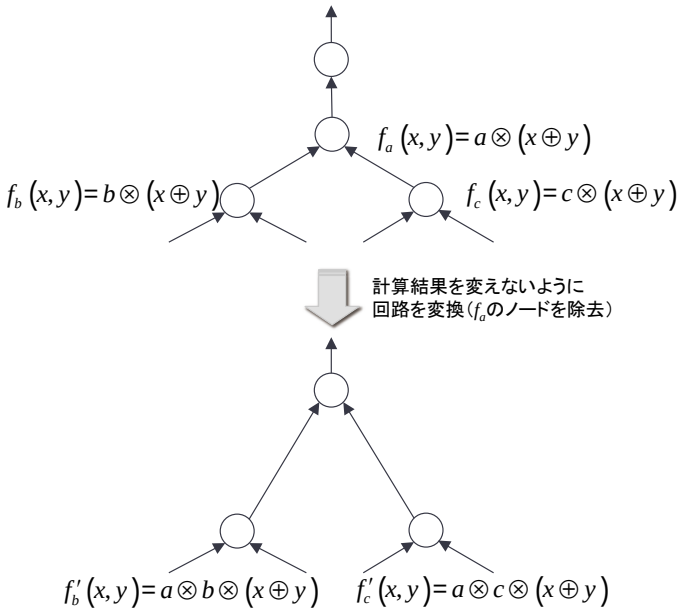


図 6　分配則を利用した回路変換の例

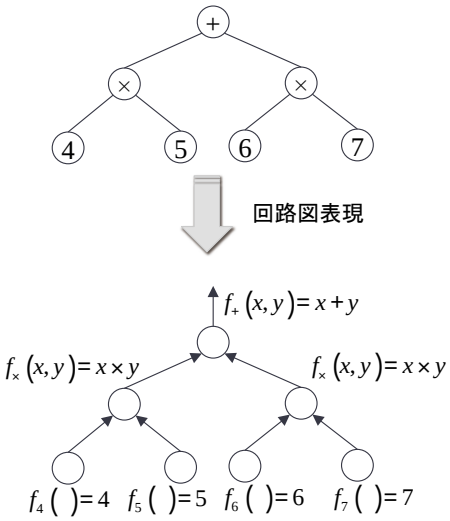


図 7　算術計算 $(4\times5)+(6\times7)$ に対応する回路図表現

$$a = p_1(a_0,b_0,c_0,a_1,b_1,c_1)$$
$$b = p_2(a_0,b_0,c_0,a_1,b_1,c_1)$$
$$c = p_3(a_0,b_0,c_0,a_1,b_1,c_1)$$

□

上記の状況および h が全単射の場合の Tree reduction との対応を図 8 に示す．h が全単射でない時は，変換後の関数に対応する入力の木は存在しないが，特に問題はない．内部頂点に対し，拡張分配則を用いて変換を行うことで頂点数を減らすことができる．これにより，分配則の場合と同様に効率的な並列計算ができるようになる．

Kakehi ら [21] は BSP モデル [22] 上での XML 表現に対する Tree reduction に関して，拡張分配則を満たす場合の高速アルゴリズムを提案した．n を要素数，p をコア数とすると，計算量は $\mathcal{O}(n/p+p)$ である．本論文で提案する

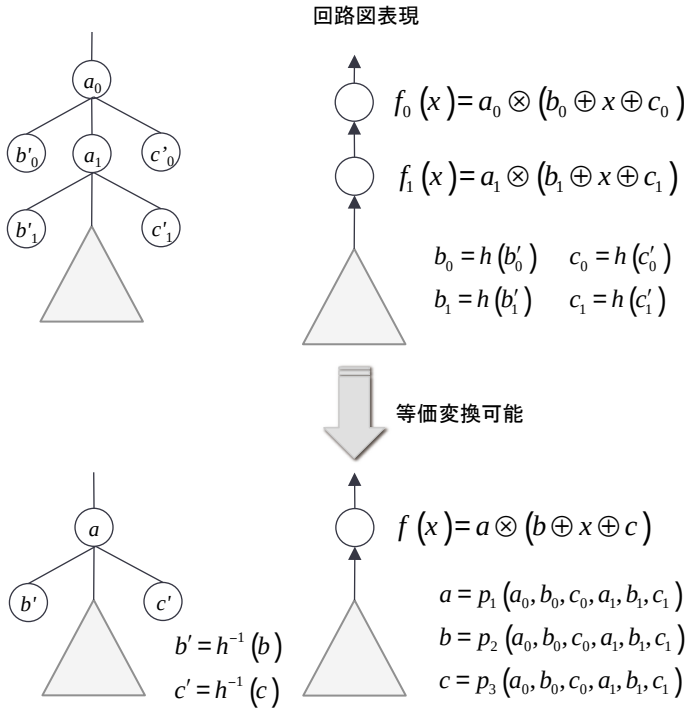


図 8　拡張分配則と Tree reduction との対応関係（h が全単射の時）

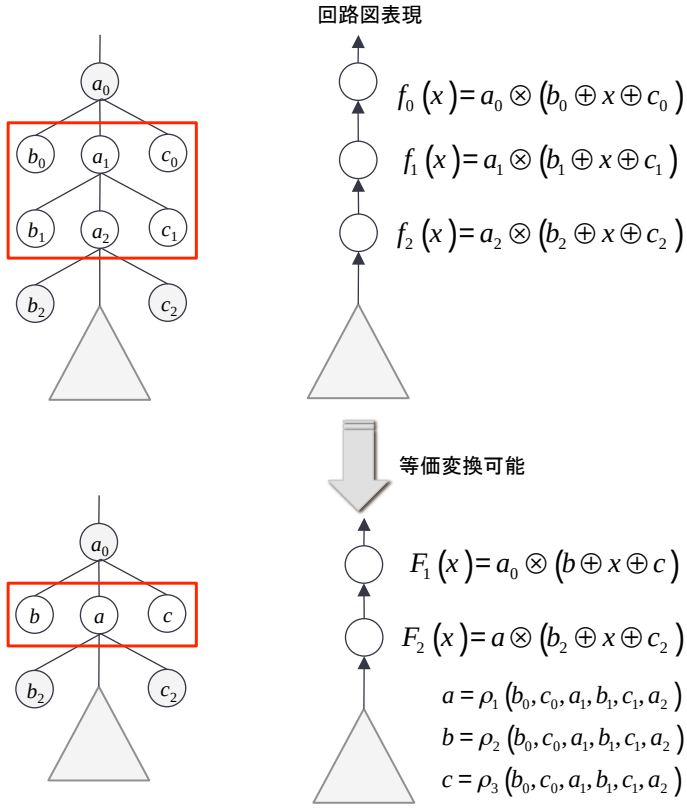


図 9　部分縮約則と Tree reduction との対応関係（$h(x) = x$ の場合）

アルゴリズムもこのアルゴリズムと同様のアイデアを使用している．また，Emoto, Imachi [23] は上記アルゴリズムを Map-Reduce モデル [24] 向けに改良した．

2.6　部分縮約則

本論文では，$\otimes, \oplus$ が部分縮約則を満たす場合の高速 GPU アルゴリズムを提案する．そこで，まず，Morihata, Matsuzaki [14] によって提案されている部分縮約則について説明する．ただし，本論文のこれまでの説明に合わせて，彼らの説明とは異なる方法で説明する．また，簡単のため，特に断らない限り $h(x) = x$ と仮定して説明する．

定義 2.2 (部分縮約則 [14])　$\oplus$ は結合的とする．また，$f_0(x) = a_0 \otimes (b_0 \oplus x \oplus c_0)$，$f_1(x) = a_1 \otimes (b_1 \oplus x \oplus c_1)$，$f_2(x) = a_2 \otimes (b_2 \oplus x \oplus c_2)$ とする．この時，入力値によらずに定数時間で計算可能な関数 ρ_1, ρ_2, ρ_3 が存在し，任意の $a_0, b_0, c_0, a_1, b_1, c_1, a_2, b_2, c_2$ に対し以下が成り立つ時，$\otimes, \oplus$ は部分縮約則を満たすという．

$$
\begin{aligned}
f_0 \circ f_1 \circ f_2(x) &= F_1 \circ F_2(x) \\
F_1(x) &= a_0 \otimes (b \oplus x \oplus c) \\
F_2(x) &= a \otimes (b_2 \oplus x \oplus c_2) \\
a &= \rho_1(b_0, c_0, a_1, b_1, c_1, a_2) \\
b &= \rho_2(b_0, c_0, a_1, b_1, c_1, a_2) \\
c &= \rho_3(b_0, c_0, a_1, b_1, c_1, a_2)
\end{aligned}
$$

□

部分縮約則と Tree reduction との対応を図 9 に示す．これは，3 つ以上の関数の合成を 2 つの関数の合成で表せる

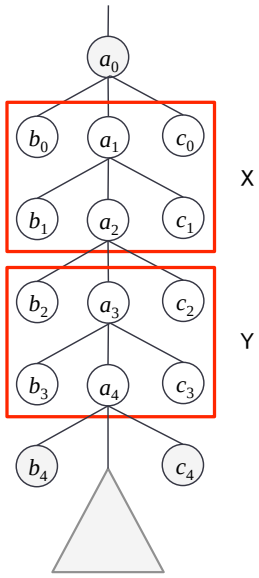


図 10　並列に部分縮約則を適用する例

ということを意味する．ただし，縮約前後で a_0, b_2, c_2 の値が変わっていないこと，および，関数 ρ_1, ρ_2, ρ_3 の引数に a_0, b_2, c_2 が含まれていないことに注意が必要である．これにより，並列化が容易になる．例えば，図 10 において部分縮約則を満たす場合，X と Y の縮約を並列に実行することができる．

$\otimes$ が結合的で，$\otimes$ が $\oplus$ に対して分配則を満たす時，$\otimes, \oplus$ は部分縮約則を満たす．この時，a, b, c は以下のように計

算できる．

$$a = a_1 \otimes a_2$$
$$b = b_0 \oplus (a_1 \otimes b_1)$$
$$c = (a_1 \otimes c_1) \oplus c_0$$

これは，図 9 上図において，a_1 を展開することで得られる．$X = b_2 \oplus x \oplus c_2$ とおくと，

$$\begin{aligned} f_0 \circ f_1 \circ f_2(x) &= a_0 \otimes (b_0 \oplus (a_1 \otimes (b_1 \oplus (a_2 \otimes X) \oplus c_1)) \\ &\quad \oplus c_0) \\ &= a_0 \otimes (b_0 \oplus (a_1 \otimes b_1) \oplus (a_1 \otimes a_2 \otimes X) \\ &\quad \oplus (a_1 \otimes c_1) \oplus c_0) \\ &= a_0 \otimes (b \oplus (a \otimes X) \oplus c) \\ &= a_0 \otimes (b \oplus (a \otimes (b_2 \oplus x \oplus c_2)) \oplus c) \\ &= F_1 \circ F_2(x) \end{aligned}$$

2.2 節の例.1，例.2，例.3 は部分縮約則を満たしている．

例.1 (sum)

a, b, c を以下のように定めることができる．（実際には $\oplus$ は可換なので c は不要）

$$a = a_1 + a_2$$
$$b = b_0 + b_1$$
$$c = c_0 + c_1$$

例.2 (max path weight)

$\otimes$ が $\oplus$ に対して分配則を満たしているので，上記の説明をそのまま適用して a, b, c を定めることができる．（実際には $\oplus$ は可換なので c は不要）

$$a = a_1 + a_2$$
$$b = \max(b_0, a_1 \otimes b_1)$$
$$c = \max(a_1 \otimes c_1, c_0)$$

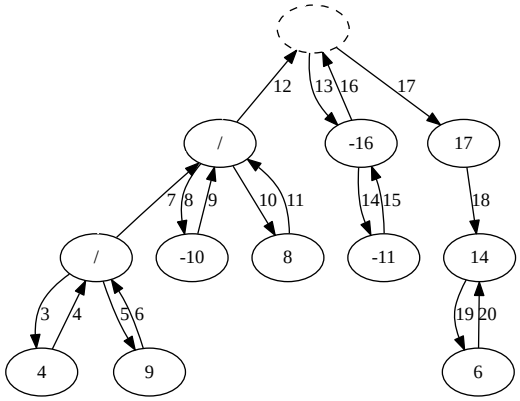
例.3 (max subtree)

やや複雑なので，まず，$\oplus$ の可換性を利用して b_i と c_i を縮約することにする．頂点 b_i と頂点 c_i を重み $b_i \oplus c_i$ の頂点に置き替え，これを頂点 b_i とする．頂点 c_i は削除する．また，このケースでは $h(x) = x$ は成り立たない．回路図表現におけるは b_0, b_1 は以下のような値を持っているとする．

$$b_0 = (m_{b_0}, s_{b_0})$$
$$b_1 = (m_{b_1}, s_{b_1})$$

この時，a, b を以下のように定めることができる．

$$a = \max(a_1 + s_{b_1} + a_2, a_2)$$
$$b = (\max(m_{b_0}, m_{b_1}), s_{b_0} + a_1 + s_{b_1} + a_2 - a)$$



Index	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
W[3..20]	4	/	9	/	/	-10	/	8	/	/	-16	-11	/	/	17	14	6	/

図 **11**　図 4 の $W[3..20]$ に対応する擬似木

3. 定式化および提案アルゴリズム

3.1 定式化

AGPU(p, b, M, w) を用いて，以下の入力から以下の出力を計算する．

input　$W[n]$: $n/2$ 頂点からなる木の XML 表現

output　$f(r, \otimes, \oplus, h)$: Tree reduction 関数の木の根 r に対する出力値

入力はグローバルメモリ上に置かれており，出力もグローバルメモリに置く．W の各要素のサイズは AGPU のワード長 w と一致しているものとする．

ただし，関数 $\otimes, \oplus$ は以下の条件を満たしているものとする．

- 関数 $\otimes, \oplus, h$ は入力値によらずに定数時間で計算可能である
- $\oplus$ は結合則を満たす
- $\otimes, \oplus$ は部分縮約則を満たす

$\otimes$ は結合的でなくてもよく，また，$\otimes, \oplus$ は可換でなくてもよい．

なお，$\otimes, \oplus$ は単位元を持っていると仮定することができる．もし，持っていない場合は添加することにする．以後，$\otimes$ と $\oplus$ の単位元を区別せずに I と書く．2 項演算において引数の一つが I の時は演算を行わなわず，もう片方の値を返す．両方 I の時は I を返す．

3.2 擬似木と縮約木

まず，提案アルゴリズムにおいてキーとなるデータ構造である擬似木と縮約木について説明する．入力配列に対する（連続する index からなる）部分配列を擬似木と呼ぶ．例として，図 4 の XML 表現に対する部分配列 $W[3..20]$ を考える．この配列の走査の様子を図示すると図 11 の上図のようになる．根に関しては対応する index が存在しないので，点線で記載してある（上向き探索では枝に対応する

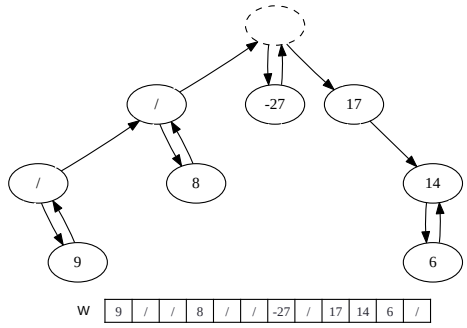


図 **12**　図 11 に対して max path weight 算出のための contraction を行った結果（縮約木と呼ぶ）

頂点は始点であることに注意）．このような頂点を擬似根と呼ぶことにする（すなわち，最も高い位置にある対応する index がない頂点が擬似根である）．すべての木，擬似木に対して，擬似根を持っていると仮定することができる．また，一部の頂点については重みの値を持っていないため，その場合の重みは / と書いた．図 11 上図において，2 頂点間が両向きの 2 本の枝で接続されている場合，その枝のペアを paired-edge と呼び，片方向の枝でしか接続されていないものを solo-edge と呼ぶことにする．擬似木の各枝は preorder の走査に対応しているので，solo-edge は擬似木の左縁か右縁にしか存在しない．言い換えれば，solo-edge は left-visible もしくは right-visible である．

Tree reduction を計算する場合，擬似木に対しても，一部の頂点について計算を行うことができる．図 11 に対して部分的に max path weight 問題のための Tree reduction を行った結果を図 12 に示す．この木を縮約木と呼ぶことにする．縮約木を算出するアルゴリズムについては後述する．縮約木の性質として，まず，擬似木のすべての solo-edge は縮約木にも含まれる．また，paired-edge には必ず葉が接続されており（そうでなければもっと contraction できる），各頂点は高々 1 つの paired-edge としか接続されない（$\oplus$ が結合的であると仮定したことにより，隣り合う sibling に対して，常に $\oplus$ による演算を行うことができる）．

以上の考察により，縮約木を保持するためには，以下の値を保持すれば十分であることがわかる．

- 配列 A: right-visible な solo-edge に対応する頂点の重みを格納する配列
- 配列 B: 配列 A の各頂点に paired-edge で接続された葉の重みを格納する配列
- 配列 C: left-visible な solo-edge に対応する頂点および擬似根に paired-edge で接続された葉の重みを格納する配列
- N_A: 配列 A の要素数
- N_C: 配列 C の要素数

これらの値を以後，5 つ組と呼ぶことにする．図 13 に配列 A, B, C を図示する．配列 B の要素数は N_A である．

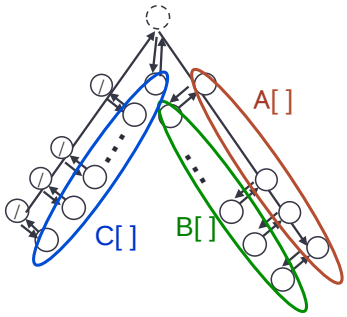


図 **13**　縮約木を保持するためのデータ構造

また，left-visible な edge に対応する頂点は重みの値を持たないため，これらのための配列は不要である．なお，right-visible または left-visible な頂点が paired-edge に接続された葉を持たない場合（例えば図 12 の重み 17 の頂点），単位元の重みを持つ葉を接続することにする．このようにすることで right-visible または left-visible な頂点は常に paired-edge に接続された葉を持つことになる．追加される頂点は高々 n 個であり，このようにしても Tree reduction の結果を変えない．以上より，right-visible な edge の数は N_A であり，left-visible な edge の数は $N_C - 1$ である．

また，配列 A, B に関しては上の要素から順に要素を格納し，配列 C に関しては下の要素から順に要素を格納する．

また，上記のデータ構造と XML 表現は相互に容易に変換できる．

3.3　提案アルゴリズムの概要

AGPU(p, b, M, w) のマルチプロセッサの数を k とおく．すなわち，$k = p/b$ である．提案アルゴリズムは以下の 4 つの手順で Tree reduction を計算する．図 14 に図示する．

(1) 入力配列を k 個の擬似木に分割し，k マルチプロセッサは個別に各擬似木に対して縮約を行い，縮約木を生成する (Local Tree Contraction for Pseudo Trees)．各マルチプロセッサは縮約木の情報を表す 5 つ組を出力する．全体として，このステップでは k 個の 5 つ組が得られる．

(2) k 個の 5 つ組を用いて，k 個の縮約木に対する parentheses matching を行う (solo-edge に対して対応する edge を探す)．この際，$\mathcal{O}(k)$ 個の区間を用いてカッコの対応関係を表現できる (Global Parentheses Matching)．これは，シーケンシャルアルゴリズムを用いて $\mathcal{O}(k)time$ で計算できる．

(3) 各マルチプロセッサは，自身に割り当てられた縮約木中の開きかっこに対して，上記の parentheses matching の情報を用いて閉じかっこの情報（具体的には，配列 C の値）を得る．そして，部分縮約則を用いて，縮約を行う．この結果，縮約されたの木の頂点数は高々 $\mathcal{O}(k)$ になる．(Local Tree Contraction for Contracted Trees)

(4) シーケンシャルアルゴリズムを用いて，残った頂点の

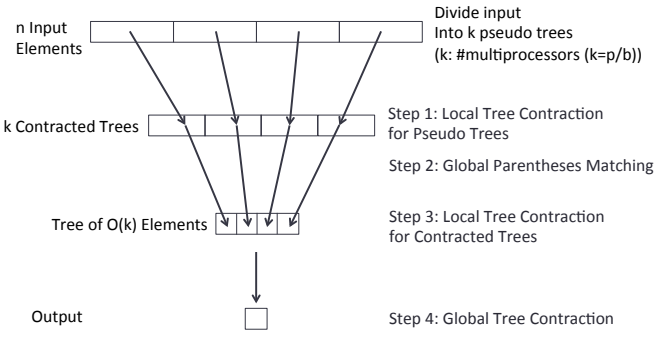


図 14　提案アルゴリズムの概要

縮約を行う．これは $\mathcal{O}(k)time$ で計算できる．(Global Tree Contraction)

以下，それぞれのステップについて，詳細を説明する．

3.4 Local Tree Contraction for Pseudo Trees

AGPU の各マルチプロセッサは高々 $\lceil n/k \rceil$ 個の要素から構成される擬似木に対し，縮約木を生成する．

まず，擬似木から縮約木を生成するためのシーケンシャルアルゴリズムを説明し，その後，そのアルゴリズムを用いて GPU 向けアルゴリズムを設計することにする．そのシーケンシャルアルゴリズムを Algorithm 3.1 に示す．これは，XML 表現に対するシーケンシャルな Tree reduction 計算アルゴリズム [16] を擬似木が扱えるように改良したものである．入力に XML 表現（部分配列でないもの）を入れた場合は，$N_C = 1, N_A = 0$ となり，Tree reduction 結果は $C[0]$ に格納される．

次に，これを用いて，GPU マルチプロセッサ向けアルゴリズムを設計する．まず，処理の概要を図 15 に示す．入力の擬似木は b^2 要素のサブブロックに分割され，サブブロックごとに処理を行う．各サブブロックに対する処理をまとめると以下のようになる．

(1) b^2 個の要素をグローバルメモリから共有メモリに読み込む

(2) 読み込んだデータは $b \times b$ の行列として表現され，マルチプロセッサ内の b コアは 1 行の b 要素を担当する．各コアは 1 行のデータに対し，Algorithm 3.1 を用いて縮約木を生成する．すると b 個の縮約木が生成される．

(3) 上記の b 個の縮約木に対し，左の縮約木から順に処理済みの縮約木とのマージを行う．この際，部分縮約則を用いることで b コアで並列に縮約処理を行う．

以下，各ステップの詳細について説明する．

3.4.1 グローバルメモリからのデータ読み込み

b^2 個の要素をグローバルメモリから共有メモリに読み込む処理について，詳細を説明する．b^2 個の要素を b 行 b 列の 2 次元配列 $w[b][b]$ とみなすことにする．先頭の b 要素は 1 行目，次の b 要素は 2 行目に配置されている．そして，アルゴリズムは 1 行ずつグローバルメモリから共有メモリに

Algorithm 3.1 縮約木算出のためのシーケンシャルアルゴリズム

```
1:  procedure GenerateContractedTree(W, n)
                                          ▷ W[n]: pseudo tree
2:      N_A ← 0                           ▷ Size of arrays A and B
3:      C[0] ← I
4:      N_C ← 1                           ▷ Size of array C
5:      for i ← 0 to n do
6:          if W[i] ≠ '/' then
7:              A[N_A] ← W[i]
8:              B[N_A] ← I
9:              N_A ← N_A + 1
10:         else
11:             if N_A == 0 then
12:                 C[N_C] ← I
13:                 N_C ← N_C + 1
14:             else if N_A == 1 then
15:                 N_A ← N_A − 1
16:                 C[N_C] ← C[N_C] ⊕ (A[N_A] ⊗ B[N_A])
17:             else
18:                 N_A ← N_A − 1
19:                 B[N_A − 1] ← B[N_A − 1] ⊕ (A[N_A] ⊗ B[N_A])
20:             end if
21:         end if
22:     end for
23:     return A, B, C, N_A, N_C
24: end procedure
```

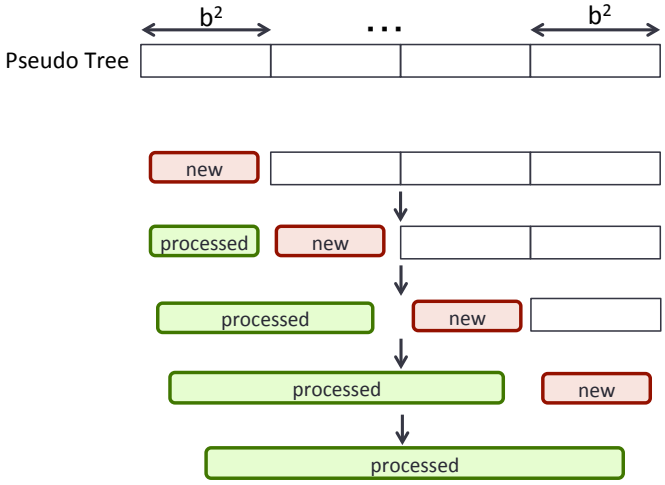
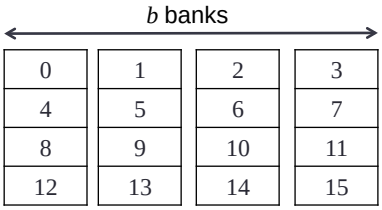


図 15　GPU マルチプロセッサ向けアルゴリズムの概要

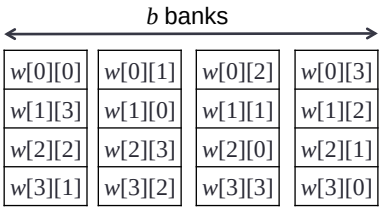
コピーする．ただし，コピー先アドレスに関して，後の処理を効率化するために工夫する．共有メモリのメモリアドレス ($w[0][0]$ のアドレスからの相対値) については図 16(a) のようになっている．この時，図 16(b) に示すように，要素 $w[i][j]$ はアドレス $bi + ((i+j)\%b)$ に配置される．この処理はグローバルメモリに対して常にコアレスアクセスを行っており，また，共有メモリに対するバンクコンフリクトは発生しない．

3.4.2 縮約木生成のための並列処理

図 16(b) に示すように共有メモリに読み込まれた b^2 個の要素に対し，b コアを用いて，並列に縮約木生成処理を



(a) 共有メモリアドレス



(b) 配列$w[b][b]$の共有メモリへの配置方法

図 **16** 配列 $w[b][b]$ の共有メモリへの配置方法 ($b = 4$ の場合)

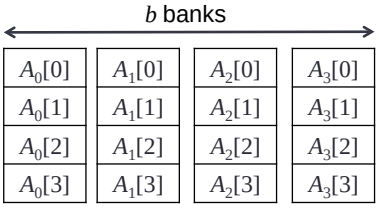


図 **17** 配列 $A_0[b], A_1[b], \cdots, A_{b-1}[b]$ の共有メモリへの配置方法 ($b = 4$ の場合)

行う．マルチプロセッサ内のコア i は i 行目を担当し，マルチプロセッサ全体で b 個の縮約木を生成する．各コアは Algorithm 3.1 を並列に実行する．ただし，メモリアクセスについては，若干修正が必要である．まず，シーケンシャルアルゴリズムが $W[j]$ にアクセスする時，コア i は $w[i][j]$ にアクセスする．よって，$w[0][j], w[1][j], \cdots, w[b-1][j]$ が並列にアクセスされることになるが，図 16(b) のメモリ配置により，バンクコンフリクトが発生しない．次に，その他のメモリについては，コア i のためのメモリはバンク i に確保する．具体例を図 17 にあげる．シーケンシャルアルゴリズムが配列 $A[b]$ を使用する時，コア i は配列 $A_i[b]$ を使用するものとする．そして，$A_i[b]$ のすべての要素はバンク i に配置される．これらの配列はコアごとに異なる index にアクセスすることがあるが，このようなメモリ配置にすることで，それらのメモリアクセスによるバンクコンフリクトを回避することができる．

3.4.3 処理済みの縮約木とのマージ処理

まず，メモリ配置について確認する．図 15 において，処理済みの縮約木 T（図 15 の緑色のブロック）はグローバルメモリに保持することとし，処理中の b 個の縮約木 t_i $(0 \le i < b)$ は，前述のとおり共有メモリに保持する．$t_0, \cdots, t_{b-1}$ を順に T にマージするが，t_i を T にマージする

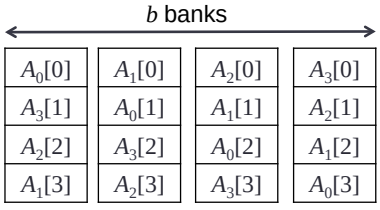


図 **18** 配列 $A_0[b], A_1[b], \cdots, A_{b-1}[b]$ の共有メモリ配置の変換 (縮約木生成後；$b = 4$ の場合)

際は，T に関する必要なデータをグローバルメモリから取得し，マージ処理したのち，再び処理結果をグローバルメモリに書き込む．マージ処理は具体的には，t_i の left-visible edge と対応する T の right-visible edge を探し，見つかった場合は，配列 A, B の値を取得し，部分縮約則を用いて並列に縮約を行う（配列 A, B, C に対し，関数 ρ_1, ρ_2, ρ_3 を使用して要素数を減らしていく）．

上記の並列縮約処理において，バンクコンフリクトを回避するため，あらかじめ共有メモリ上のメモリ配置を変換する．前節の処理が完了したとき，図 17 に示すように t_i のデータはすべて bank i に格納されている．マージ処理を行う前に，これを図 18 のように変換する．正確には $A_i[j]$ がアドレス $bj + ((i+j)\%b)$ に配置されるようにする．この処理は 1 行ずつデータを上書きしていくことにより行う．よって，この処理は $2b$ 回の共有メモリアクセスにより行うことができる．配列 A_i, B_i, C_i のすべてに対して，この処理を行う．この変換により，配列 A_i の異なる index に対し，コアが並列でアクセスできるようになる．

ここで，連続する 2 つの縮約木に関する性質について確認する．1 つめの擬似木に対して走査を行った際の最終到達頂点と 2 つめの擬似木における走査開始頂点は一致する．また，縮約木においてもこれらの頂点が縮約されることはない．よって，連続する 2 つの擬似木においてそれぞれ縮約木を生成すると，1 つめの縮約木の最右頂点と 2 つめの縮約木の最左頂点が一致する．

処理済みの縮約木と処理中の縮約木のマージ処理の例を図 19 に示す．1 縮約木のマージ処理において，最大でも b 頂点しかマージされないため，この処理を行うためには，T の最右頂点から上方向に最大 b 個の right-visible edge の情報を取得すれば十分である．よって，1 回のマージおけるグローバルメモリアクセスの回数は A, B それぞれに対して高々 2 回ずつで十分である．

次に，マージ結果の書き込みについて説明する．t_i の right-visible edge を T に追加するとき，T の縮約されずに残った最右頂点の先に接続される．right-visible edge については，上（根）から下（葉）方向にデータが格納されているため，T のデータを修正することなく，高々 2 回のグローバルメモリアクセスで，書き込みが完了する．t_i の left-visible edge を T に追加するとき，T の擬似根の上に

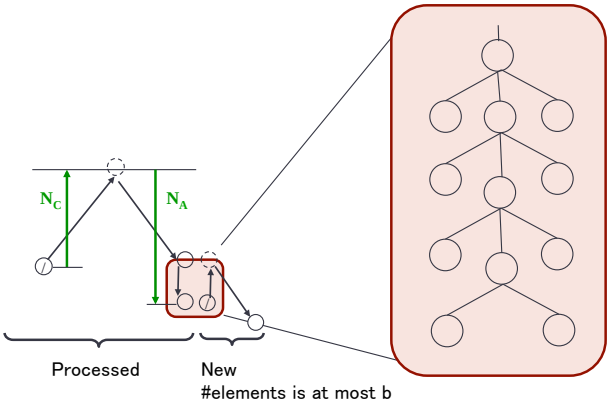


図 19　処理済みの縮約木と処理中の縮約木のマージ処理の例

接続される．left-visible edge については，下（葉）から上（根）方向にデータが格納されているため，T のデータを修正することなく，高々 2 回のグローバルメモリアクセスで，書き込みが完了する．

3.5　Global Parentheses Matching

前節までの処理で，k 個の縮約木が得られる．k 個の縮約木は連続する 2 つの縮約木の終点と始点を接続することにより，1 つの大きな擬似木であると考えることができる．そこで，次にこの擬似木に対してさらに縮約を行うことを考える．なお，これ以降のアルゴリズムは Kakehi ら [21] のアルゴリズムと同様のアイデアを使用している．各縮約木に対する N_A, N_C をスキャンすることで，right-visible edge と left-visible edge の対応を計算することができる（各縮約木に対する N_A, N_C を格納する配列を $N_A[k], N_C[k]$ とする）．この処理を本論文では Parentheses matching と呼ぶことにする（本処理は対応する BP 表現上での Parentheses matching になっているため）．$k = 4$ の場合の具体例を図 20(a) に示す．right-visible edge と left-visible edge の対応が見つかった場合，right-visible edge が left-visible edge の情報を取得することにより，図 20(b) のような形状の木となる．この部分に対して，部分縮約則を用いて縮約を行うことができる．もし木の内部にこのような構造があったとしても，縮約が可能であることに注意する．なお，図 20(a) の左 3 つの縮約木の結合部は図 20(c) のようになっている．頂点 a_2 は 2 番目の擬似木の擬似根に対応している．また，1 つめの縮約木と 2 つめの縮約木の縮約前は b_2 は子孫を持っている．また，3 つめの縮約木の c_2 よりも深い頂点は，2 つめの縮約木により読み込まれる．

Parentheses matching の詳細を述べる前に，まず，使用する用語を整理する．木の深さを 1 つめの縮約木の始点頂点からの相対的な深さで表現する．1 つめの縮約木の始点頂点の深さを 0 とし，その頂点より上にある頂点の深さは負の値をとる．深さ h から ℓ $(h \le \ell)$ において，縮約木 i の right-visible edge（に対応する頂点）と縮約木 j の

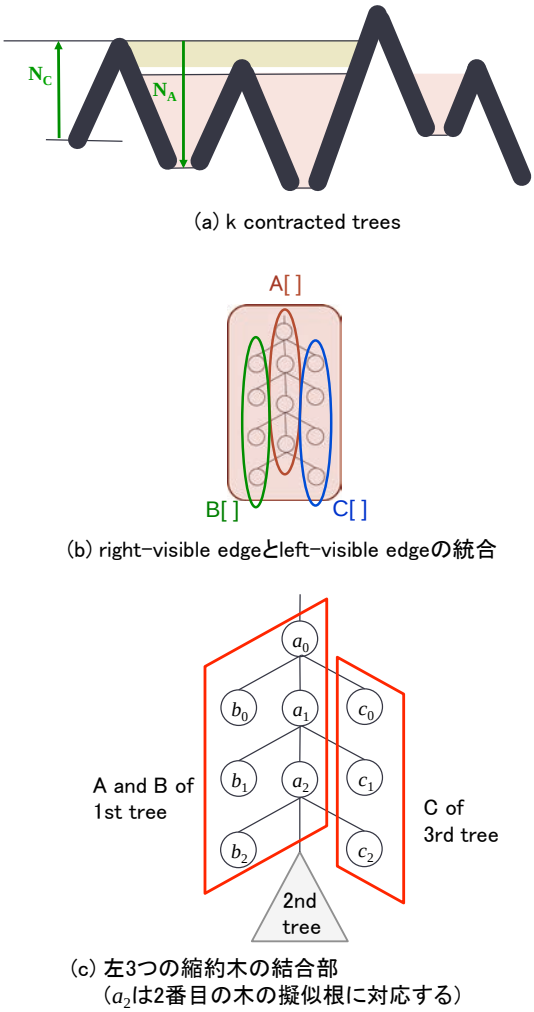


図 20　k 個の縮約木に対する Parentheses matching $(k = 4)$

left-visible edge（に対応する頂点）に対応があるとき，その対応関係を 4 つ組を用いて (h, ℓ, i, j) と表す．もし対応関係が見つかっていない場合は，(h, ℓ, i, ϕ) や (h, ℓ, ϕ, j) と書くことにする．また，各縮約木 i は上記 right-visible edge の対応関係を保持するための配列 $R_i[b]$，および対応関係のない left-visible edge の区間を保持するための変数 L_i を持つ．Parentheses matching のためのシーケンシャルアルゴリズムを Algorithm 3.2 に示す．本アルゴリズムにおいて，$R_i[b]$ は深い区間ほど前に格納されることに注意する．また，対応関係を表す 4 つ組の総数は $\mathcal{O}(k)$ である．

3.6　Local Tree Contraction for Contracted Trees

前節で求めた $R_0, \cdots, R_{k-1}$ の情報に基づき，k 個のマルチプロセッサは並列に right-visible edge に対応する left-visible edge の取得と部分縮約則による縮約処理を行う．$R_i[0]$（先端が葉である区間）では定数値が求まり，それ以外（内部区間）では，3 つ組 (a, b, c) が求まる（図 9 参照）．すなわち，$\mathcal{O}(k)$ 個の区間のそれぞれは，定数個の要素数で表現される縮約木に縮約される．詳細は省略する．

Algorithm 3.2 Parentheses matching のためのシーケンシャルアルゴリズム

```
 1: procedure MatchParentheses(N_A, N_C, k)
                                                  ▷ N_A[k], N_C[k]
 2:     N_{R_0}, N_{R_1}, ..., N_{R_{k-1}} ← 0  ▷ Size of arrays R_0, ..., R_{k-1}
 3:     N_R ← 0                                  ▷ Size of arrays R
 4:     d ← 0                                    ▷ Current depth
 5:     L_0, L_1, ..., L_{k-1} ← NULL
 6:     for i ← 0 to k − 1 do
 7:         d ← d − (N_C[i] − 1)
 8:         while N_R ≠ 0 do
 9:             N_R ← N_R − 1
10:             (h, ℓ, p, q) ← R[N_R]
                        ▷ R stores unmatched right-visible edges
11:             if h > d then                    ▷ h is deeper than d
12:                 R_p[N_{R_p}] ← (h, ℓ, p, i)
13:                 N_{R_j} ← N_{R_j} + 1
14:                 if N_R == 0 and h > d + 1 then
15:                     L_i ← (d + 1, h − 1, φ, i)
16:                 end if
17:             else
18:                 R_p[N_{R_p}] ← (d + 1, ℓ, p, i)
19:                 N_{R_p} ← N_{R_p} + 1
20:                 R[N_R] ← (h, d, p, φ)
21:                 N_R ← N_R + 1
22:                 Break (exit While loop)
23:             end if
24:         end while
25:         if N_A[i] > 0 then
26:             R[N_R] ← (d + 1, d + N_A[i], i, φ)
27:             N_R ← N_R + 1
28:             d ← d + N_A[i]
29:         end if
30:     end for
31:     while N_R ≠ 0 do
32:         N_R ← N_R − 1
33:         (h, ℓ, p, q) ← R[N_R]
34:         R_p[N_{R_p}] ← (h, ℓ, p, φ)
35:         N_{R_p} ← N_{R_p} + 1
36:     end while
37:     return R_0, ..., R_{k-1}, N_{R_0}, ..., N_{R_{k-1}}, L_0, ..., L_{k-1}
38: end procedure
```

3.7 Global Tree Contraction

区間の数は $\mathcal{O}(k)$ 個であり，各区間は前節の処理により，定数個の値に縮約される．最後に 1 コアを使って，これらの区間をマージし，1 つの縮約木を得る．詳細は省略する．

3.8 計算量の解析

$n \geq p^2$ の場合の計算量を解析する．まず，時間計算量について解析する．Local Tree Contraction for Pseudo Trees に関して，b^2 要素に対する処理は $\mathcal{O}(b \log b)$time で行える（処理済み縮約木へのマージ処理に $\mathcal{O}(b \log b)$time かかり，他は $\mathcal{O}(b \log b)$time でできる）．よって，このステップ全体では $\mathcal{O}\left(\frac{n}{k}\frac{b}{b^2}\right) = \mathcal{O}\left(\frac{n}{p}\right)$time となる．また，Local Tree Contraction for Contracted Trees に関しては，非可換演算子による Reduction として計算できるので，Koike, Sadakane [3] により提案されているパイプラインアルゴリズムを使用することにより，計算時間は $\mathcal{O}(\frac{n}{kb}) = \mathcal{O}(\frac{n}{p})$time となる．その他の処理は $\mathcal{O}(k)$time で計算できる．よって，時間計算量は $\mathcal{O}(\frac{n \log b}{p})$time となる．

次に I/O 計算量については，どの要素にも高々定数回しかアクセスせず，1 アクセスでは必ず b 要素を取得している．よって，$\mathcal{O}\left(\frac{n}{b}\right)$ となる．また，グローバルメモリ使用量は $\mathcal{O}(n)$ ワード，共有メモリ使用量は $\mathcal{O}(b^2)$ ワードである．よって多重度は $\mathcal{O}\left(\frac{M}{b^2}\right)$ となる．

4. 結論

本論文では，Tree reduction のための GPU アルゴリズムを提案した．Tree reduction は木に対する多くのクエリの一般化になっている．使用する演算子が部分縮約則を満たす時も，なお，多くの問題がカバーされる．本論文では演算子が部分縮約則を満たす時の GPU 向けアルゴリズムを提案し，I/O 計算量が最適となることを示した．今後は，実装評価を行ったのち，木上のパターンマッチングや確率伝播法などの様々な実用的な応用に対し，提案アルゴリズムを適用したい．

参考文献

[1] Patterson, D. A. and Hennessy, J. L.: *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition (2013).

[2] NVIDIA Corporation: NVIDIA CUDA C Programming Guide version 7.5 (2015).

[3] Koike, A. and Sadakane, K.: A Novel Computational Model for GPUs with Applications to Efficient Algorithms, *International Journal of Networking and Computing*, Vol. 5, No. 1, pp. 26–60 (online), available from ⟨http://www.ijnc.org/index.php/ijnc/article/view/96⟩ (2015).

[4] Skillicorn, D. B.: Structured Parallel Computation in Structured Documents, *Journal of Universal Computer Science*, Vol. 3, No. 1, pp. 42–68 (online), available from ⟨http://www.jucs.org/jucs_3_1/structured_parallel—⟩ (1997).

[5] Miller, G. L. and Reif, J. H.: Parallel Tree Contraction and Its Application, *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, Washington, DC, USA, IEEE Computer Society, pp. 478–489 (online), DOI: 10.1109/SFCS.1985.43 (1985).

[6] Morihata, A., Matsuzaki, K., Hu, Z. and Takeichi, M.: The Third Homomorphism Theorem on Trees: Downward & Upward Lead to Divide-and-conquer, *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, New York, NY, USA, ACM, pp. 177–185 (online), DOI: 10.1145/1480881.1480905 (2009).

[7] Dekel, E., Ntafos, S. and Peng, S.-T.: Parallel tree techniques and code optimization, *VLSI Algorithms and Architectures* (Makedon, F., Mehlhorn, K., Papatheodorou, T. and Spirakis, P., eds.), Lecture Notes in Computer Science, Vol. 227, Springer Berlin Heidel-

berg, pp. 205–216 (online), DOI: 10.1007/3-540-16766-8_18 (1986).

[8] Teng, S.-H. and Wang, B.: Parallel algorithms for message decomposition, *Journal of Parallel and Distributed Computing*, Vol. 4, No. 3, pp. 231 – 249 (online), DOI: http://dx.doi.org/10.1016/0743-7315(87)90035-9 (1987).

[9] Miller, G. L. and Reif, J. H.: Parallel Tree Contraction Part 2: Further Applications, *SIAM Journal on Computing*, Vol. 20, No. 6, pp. 1128–1147 (online), DOI: 10.1137/0220070 (1991).

[10] Miller, G. L. and H. Teng, S.: Tree-Based Parallel Algorithm Design, *Algorithmica*, Vol. 19, No. 4, pp. 369–389 (online), DOI: 10.1007/PL00009179 (1997).

[11] Rao Kosaraju, S. and Delcher, A.: Optimal parallel evaluation of tree-structured computations by raking (extended abstract), *VLSI Algorithms and Architectures* (Reif, J., ed.), Lecture Notes in Computer Science, Vol. 319, Springer New York, pp. 101–110 (online), DOI: 10.1007/BFb0040378 (1988).

[12] Cole, R. and Vishkin, U.: The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, *Algorithmica*, Vol. 3, No. 1-4, pp. 329–346 (online), DOI: 10.1007/BF01762121 (1988).

[13] Abrahamson, K., Dadoun, N., Kirkpatrick, D. and Przytycka, T.: A simple parallel tree contraction algorithm, *Journal of Algorithms*, Vol. 10, No. 2, pp. 287 – 302 (online), DOI: http://dx.doi.org/10.1016/0196-6774(89)90017-5 (1989).

[14] Morihata, A. and Matsuzaki, K.: A Practical Tree Contraction Algorithm for Parallel Skeletons on Trees of Unbounded Degree, *Procedia Computer Science*, Vol. 4, pp. 7 – 16 (online), DOI: http://dx.doi.org/10.1016/j.procs.2011.04.002 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011.

[15] Reif, J. H.: *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition (1993).

[16] Gibbons, A. and Rytter, W.: Optimal parallel algorithms for dynamic expression evaluation and context-free recognition, *Information and Computation*, Vol. 81, No. 1, pp. 32 – 45 (online), DOI: http://dx.doi.org/10.1016/0890-5401(89)90027-8 (1989).

[17] Robie, J. and Dyck, M.: XQuery 3.1: An XML Query Language (2014).

[18] Tarjan, R. E. and Vishkin, U.: Finding Biconnected Componemts And Computing Tree Functions In Logarithmic Parallel Time, *Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, SFCS '84, Washington, DC, USA, IEEE Computer Society, pp. 12–20 (online), DOI: 10.1109/SFCS.1984.715896 (1984).

[19] Matsuzaki, K., Hu, Z., Kakehi, K. and Takeichi, M.: SYSTEMATIC DERIVATION OF TREE CONTRACTION ALGORITHMS, *Parallel Processing Letters*, Vol. 15, No. 3, pp. 321–336 (2005).

[20] Matsuzaki, K.: Parallel programming with tree skeletons, PhD Thesis, The University of Tokyo, Japan (2007).

[21] Kakehi, K., Matsuzaki, K. and Emoto, K.: Efficient Parallel Tree Reductions on Distributed Memory Environments, *Proceedings of the 7th international conference on Computational Science, Part II*, ICCS '07, Berlin, Heidelberg, Springer-Verlag, pp. 601–608 (online), available from ⟨http://dx.doi.org/10.1007/978-3-540-72586-2_88⟩ (2007).

[22] Valiant, L. G.: A bridging model for parallel computation, *Commun. ACM*, Vol. 33, No. 8, pp. 103–111 (online), DOI: 10.1145/79173.79181 (1990).

[23] Emoto, K. and Imachi, H.: Parallel Tree Reduction on MapReduce, *Procedia Computer Science*, Vol. 9, pp. 1827 – 1836 (online), DOI: http://dx.doi.org/10.1016/j.procs.2012.04.201 (2012). Proceedings of the International Conference on Computational Science, ICCS 2012.

[24] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, USENIX Association, pp. 10–10 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1251254.1251264⟩ (2004).

Efficient GPU implementations for the Conway's Game of Life

Toru Fujita, Daigo Nishikori, Koji Nakano and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The Conway's Game of Life is the most well-known cellular automaton. The universe of the Game of Life is a 2-dimensional array of cells, each of which takes two possible states, alive or dead. The state of every cell is repeatedly updated according to those of eight neighbors. A cell will be alive if exactly three neighbors are alive, or if it is alive and two or three neighbors are alive. The main contribution of this paper is to develop several acceleration techniques for simulating the Game of Life. The key techniques for the simulation is to store a block of cells in registers of 32 threads in a warp of a CUDA block and to perform multiple-step simulation. We use a warp shuffle instruction, which allows us to exchange data stored in registers of threads in a warp, to transfer the current states stored in registers of other threads necessary to compute the next states. Further, since multiple-step simulation is performed, the number of CUDA kernel calls can be decreased. The experimental results show that, the best configuration of our GPU implementation can perform 1024-step simulation of 16384×16384 cells in 0.163 seconds on GeForce GTX TITAN X GPU. The best sequential algorithm using Intel Xeon X7460 CPU runs 58.3 seconds. Hence, our best GPU implementation has achieved a speed-up factor of 357 over the CPU implementation.**

I. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]–[4]. 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], [5]–[7]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], [9], 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 [10], since they have thousands 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* [8]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-96 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-12 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 *bank conflicts* of the shared memory access and *coalescing* of the global memory access [9]–[11].The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the shared memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the throughput between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory. Also, the latency of the global memory access is several hundred clock cycles, while that of the shared memory access is around 10 clock cycles [12]. Hence, we should minimize the memory access to the global memory to maximize the performance. Further, CUDA-enabled GPUs with Kepler [13] and Maxwell [14] architectures support *warp shuffle* instructions that directly exchanges data stored in registers of threads in the same warp [8]. It is faster than inter-thread communication by reading/writing the shared memory, Thus, we should use warp shuffle instructions whenever possible. Actually, it has been presented that warp shuffle instructions can accelerate the computation [15], [16].

The Conway's Game of Life was created John Horton Conway, a mathematician at Gonville and Caius College of the University of Cambridge [17], [18]. The universe of the Game of Life is an 2-dimensional array of *cells*, each of which takes one of two states, 1 (*alive*) and 0 (*dead*). The state of every cell is updated by the current states of the eight neighbors as follows:

1) (die) An alive cell becomes dead if it has fewer than two or more than three alive neighbors.
2) (born) A dead cell becomes alive if it has three alive neighbors.
3) (keep alive) An alive cell keeps alive if it has two or three alive neighbors.
4) (keep dead) A dead cell keeps dead if it has fewer than three or more than three neighbors.

Figure 1 illustrates the rules of the Game of Life. Originally the Conway's Game of Life assumes that the size of the 2-dimensional array is infinite. However, to store all the states in the memory, we assume that the universe is finite and the 2-dimensional array has $\sqrt{n} \times \sqrt{n}$ cells. Clearly, some neighbors of cells in the boundary of the 2-dimensional array do not exist. Sometimes, it is assumed that the states of such non-existent neighbors are always dead. In this paper, we assume that the 2-dimensional array is wrapper around to handle the boundary case. For example, the left neighbor of a

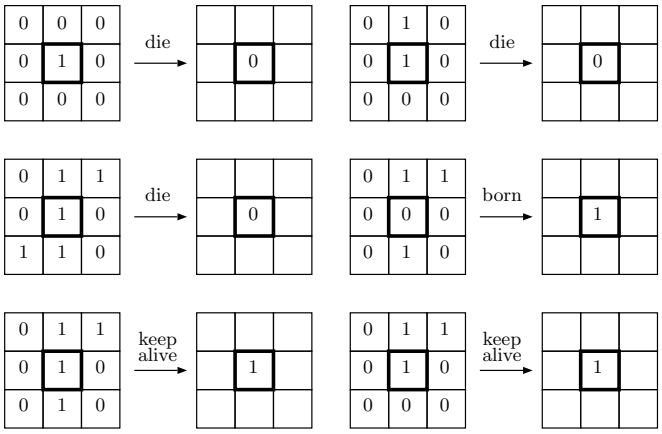


Fig. 1. The rules of the Game of Life

cell in the leftmost column is the rightmost cell in the same row.

It is easy to write a program for simulating the Game of Life if the state of a cell is stored in a word of data such as an 8-bit character and a 32-bit integer. However, for accelerating the simulation, it makes sense to use *bit-per-cell* arrangement [19] in which the state of a cell is stored as a bit of a word. For example, a 32-bit integer is used to store the states of 32 cells. A very sophisticated way to compute the next states of cells stored in a word by bitwise operations has been presented [20]. Also, the simulation of the Game of Life can be done by *stencil codes*, a class of kernels updating elements in an array according to some fixed pattern, called *stencil*. Hence, it is easy to implement the simulation using a framework of stencil computation. For example, it can be implemented on GPUs with few codes using stencil operations of MATLAB [21].

We are interested in how we can accelerate the simulation of the Game of Life using CUDA-enabled GPUs. However, as far as we know, there is no published technical paper aiming to accelerate the simulation. Very few papers presented GPU implementations of the simulation [22], [23], but their implementations are straightforward and did not aim to accelerate the simulation. On the other hand, there are a lot of web sites that present GPU implementations of the Game of Life. For example, bitwise logical operations for the bit-per-cell arrangement are used to compute the next states of cells [19]. Our implementations use the same technique. In addition, we developed a multiple-step simulation technique, which reduces memory access to the global memory. Also, we store the states of cells in registers of threads, and data transfer between registers is performed by a warp shuffle instruction. Using this techniques, we have obtained extremely fast GPU implementation for simulating the Game of Life using GPUs. For simulating the Game of Life with more than 1,000,000,000 cells, the best GPU implementation in [19] achieved 2.47×10^{10} updates per second on GeForce GTX 480 GPU. Our implementation performs 1024-step simulation of the Game of Life with 2^{28} cells in 0.163 seconds on GeForce GTX TITAN X GPU. Hence, it achieves 1.69×10^{12} updates per second and more than 68 times faster than the previously published implementation. GeForce GTX 480 and GTX TITAN X have 480 and 3072 processor cores running 1401MHz and 1000MHz, respectively. Thus, our implementation is much more efficient even if the difference of computing power of GPUs is taking into account.

This paper is organized as follows. In Section II, we first briefly explain the GPU architecture and CUDA programming model to understand GPU implementations of the Game of Life. Section III defines the Game of Life formally, and Section IV shows basic techniques to accelerate the simulation of the Game of Life including bit-per-cell arrangement and simulation using bitwise logical operations. Section IV also shows a straightforward implementation using the basic techniques on GPUs using the global memory. In Section V, we presents our new techniques for accelerating the simulation. We show that we can reduce the total number of bitwise operations if each thread computes the next states of cells in two words at the same time. We also present an idea of multiple-step simulation, which copies the states of cells to the shared memory and repeats the simulation several times. For further acceleration, we can store the states in registers of threads for multiple-step simulation, and the simulation can be performed by a warp shuffle instruction. Finally, Section VI shows experimental results. More specifically, we have implemented simulation algorithms of the Game of Life on the CPU and the GPU, and evaluated the running time. The experimental results show that our implementation is 357 times faster The experimental results show that our implementation is 357 times faster than the CPU implementation. Section VII concludes our work.

II. GPU ARCHITECTURE AND CUDA PROGRAMMING MODEL

This section briefly describes the GPU architecture and the CUDA programming model necessary to understand GPU implementations of the Game of Life. Please see [8] for the details.

Figure 2 (1) illustrates an architecture of CUDA-enabled GPUs. A GPU is a single-chip processor equipped with multiple *Streaming Multiprocessors (SMs)*, each of which has *processor cores*, *the shared memory* and *the register file*. The GPU processor is connected to *an off-chip memory*. For example, GeForce GTX TITAN X has 16 SMs[1] with 192 processor cores, a 96Kbyte shared memory, and a register file with 64K 32bit registers each. The off-chip memory can be accessed by all processor cores in all SMs, while the shared memory can be accessed only by processor cores in the same SM. Also, registers in a register file are assigned to a processor core, and they can be accessed only by the assigned processor core. The off-chip memory is quite large, say 12G bytes, but the memory access latency is quite large, say several hundred clock cycles. The memory access latency of the shared memory is around 10 [12] and that of registers in the register file is smaller. Hence, to accelerate the computation, we should minimize the global memory access. We should also use registers whenever possible.

When we develop programs running on GPUs, we can use CUDA programming model to support scalability. We assume that CUDA Compute Capability 5.2, which is available for

[1]Since the architecture of GeForce GTX TITAN X is Maxwell, its SM is particularly termed Maxwell Streaming Multiprocessor (SMM).

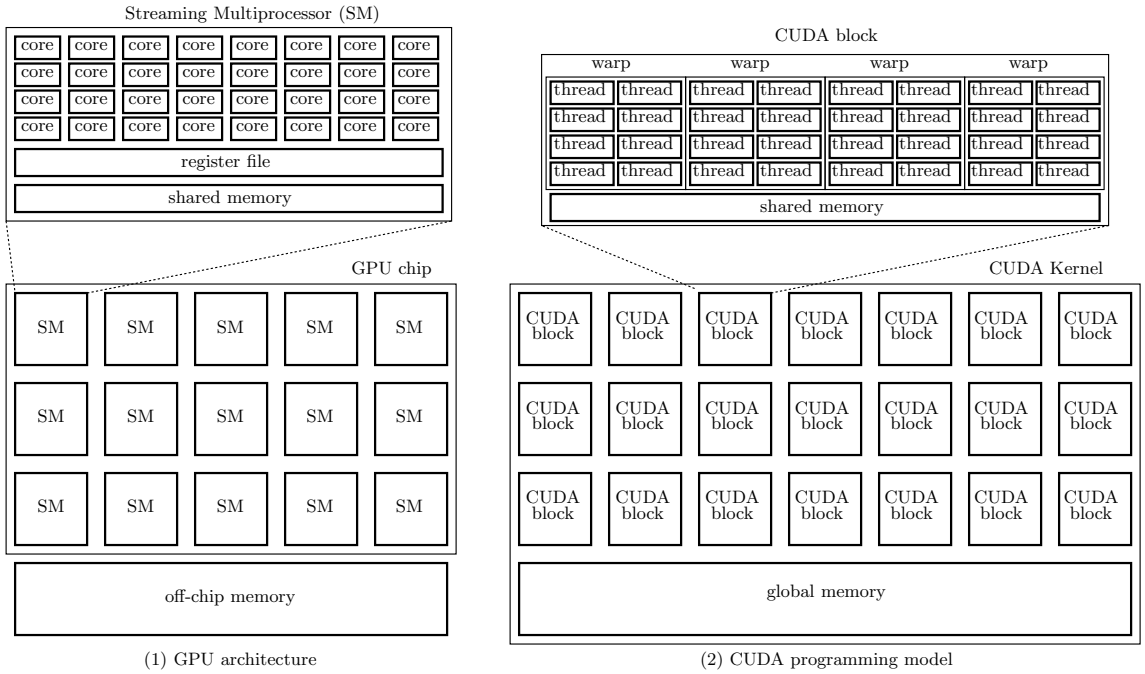


Fig. 2. GPU architecture and CUDA programming model

GeForce GTX TITAN X [24]. Usually, a CUDA program executed on the host computer invokes CUDA kernels one or more times. A CUDA kernel executes one or more CUDA blocks running on SMs of the GPU. CUDA blocks in a CUDA kernel are identical in the sense that they have the same number of threads executing the same program. Each CUDA block can have up to 1024 threads, and is dispatched to one of the SM of the GPU. Since the number of CUDA blocks can be more than the number of SMs in a single GPU, they are dispatched to SMs in turn. Also, it is possible that two or more CUDA blocks are executed in a single SM at the same time. Each SM can handle up to 32 CUDA blocks with total of 2048 threads at the same time. Since each SM has 128 processor cores, at most 128 threads among them can be active and work in parallel. In other words, each SM can have up to 2048 resident threads and 128 of them can be active on processor cores. A CUDA block can use *the shared memory*, which can be access by all threads in it. The shared memory of a CUDA block is implemented in the shared memory of the SM. Hence, its capacity is up to 96K bytes for CUDA compute capability 5.2 [8], and two ore more CUDA blocks can be arranged in the SM at the same time only if the total shared memory capacity is no more 64K bytes. All threads in all CUDA blocks can access *the global memory*, which is arranged in the off-chip memory (DRAM) of the GPU. Note that after all threads in a CUDA block terminates, data stored in the shared memory are lost, because the shared memory in an SM may be used for another CUDA block. If data stored in the shared memory must be referred later, they must be copied to the global memory on developer's own responsibility.

Threads in a CUDA block are partitioned into groups of 32 threads each called *warps*. It is guaranteed that 32 threads in the same warp execute the same instruction at the same time. Hence, if a CUDA block has at most 32 threads, they are executed synchronously. However, threads in different warps may not be executed at the same time. All threads in a CUDA block can call `__syncthreads()` for barrier synchronization if necessary. However the cost is `__syncthreads()` is not negligibly small. Hence, it makes sense to use a CUDA block with 32 threads for avoiding barrier synchronization using `__syncthreads()`, if we need to synchronize all threads in a CUDA block frequently. Also, to synchronize all threads in all CUDA blocks, we need to use separate CUDA kernel calls, because SMs in the GPU executes CUDA blocks in turn. The synchronization of all CUDA blocks are very costly, and we should minimize it.

Efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. To maximize the throughput between the GPU and the off-chip memory, the consecutive addresses of the global memory must be accessed at the same time. Hence, threads in a CUDA block should perform *coalesced access* when they access the global memory. Since the shared memory consists of 32 memory banks, memory access by 32 threads in a warp must be destined for distinct memory banks. In other words, *bank conflicts* by a warp should be avoided to maximize the shared memory access performance.

The communication between threads can be done through the global memory or the shared memory. Note that the communication between threads in different CUDA blocks in the same CUDA kernel call is not possible, because CUDA blocks may be dispatched to SMs in an arbitrary order. What threads in a CUDA kernel can do is to send data to threads in the following CUDA kernel by reading/writing the global memory

CUDA compute capability 3.0 and later supports *warp shuffle* instructions that permits exchanging of data stored in

registers in threads in a warp. The data exchange occurs at the same time for all active threads in a warp. For example,if `__shfl(a,i)` is executed by a CUDA block with a warp of 32 threads, the value of register a of thread i is returned. Since the data size for warp shuffle instructions must be 32 bits, two separate invocations are necessary to exchange 64-bit data. Warp shuffle instructions are more efficient than a conventional data exchanging method using write/read operations to the shared memory.

III. Conway's Game of Life and an conventional implementation

The universe of *the Conway's Game of Life* is a 2-dimensional array of cells, each of which takes one of two states, 1 (*alive*) or 0 (*dead*). For simplicity, we assume that the size of the array is $\sqrt{n} \times \sqrt{n}$. Let $u_0, u_1, \ldots$ denote 2-dimensional arrays such that u_0 stores the initial states, and each u_t ($t \geq 1$) is an array of cells after t-step transition. Let $u_t(i,j)$ denote the state of a cell at position (i,j) ($0 \leq i,j \leq \sqrt{n}-1$) . For simplicity, we assume that the 2-dimensional array is wrap around to handle the state of cells outside of the array. For example, the value of $u_t(i,-1)$ is that of $u_t(i,\sqrt{n}-1)$. Let $s_t(i,j)$ be the number of alive cells in eight neighbors, that is,

$$\begin{aligned} s_t(i,j) &= u_t(i-1,j-1)+u_t(i-1,j)+u_t(i-1,j+1) \\ &\quad +u_t(i,j-1)+u_t(i,j+1)+u_t(i+1,j-1) \\ &\quad +u_t(i+1,j)+u_t(i+1,j+1) \end{aligned} \quad (1)$$

The value of $u_t(i,j)$ ($0 \leq i,j \leq \sqrt{n}-1$) is determined by the following formula:

$$\begin{aligned} g_t(i,j) &= 1 \text{ (alive)} \quad \text{if } s_{t-1}(i,j)=3 \\ &\qquad \text{or } (s_{t-1}(i,j)=2 \text{ and } g_{t-1}(i,j)=1), \\ &= 0 \text{ (dead)} \quad \text{otherwise.} \end{aligned}$$

Hence, we can compute the value of $u_t(i,j)$ by the following Boolean formula:

$$\begin{aligned} u_t(i,j) &= (s_{t-1}(i,j)=3) \\ &\quad \vee (u_{t-1}(i,j) \wedge (s_{t-1}(i,j)=2)) \end{aligned} \quad (2)$$

We have two arrangements, *the word-per-cell* and *the bit-per-cell* arrangements for simulating the Game of Life not only on the GPU but also on the CPU. The word-per-cell arrangement is a conventional arrangement in which the state of each cell is stored in a word of the memory, such as a 32-bit integer or an 8-bit character. We assume that the initial states of cells are stored in the global memory of the GPU. For example, we can store the states $u_0(i,j)$ ($0 \leq i,j \leq \sqrt{n}-1$) of cells in a $\sqrt{n} \times \sqrt{n}$ 2-dimensional array of 8-bit characters. We use a CUDA kernel with n threads to compute the next states $u_1(i,j)$. For example, a CUDA kernel invokes $\frac{n}{32}$ CUDA blocks with 32 threads each. Each thread is assigned to a cell, and it evaluates formulas (1) and (2) to compute the next state $u_1(i,j)$ and write it in the global memory. Note that it is not possible to compute $u_2(i,j)$ by the same CUDA kernel, because threads in different CUDA blocks cannot communicate with each other. Thus, after a thread computes and writes $u_1(i,j)$, it must terminate. A CUDA kernel terminates when all threads complete the computation of next states of cells. After that, the same CUDA kernel to compute $g_2(i,j)$ is invoked. In other words, one CUDA kernel call is necessary to simulate one-step transition and thus, T CUDA kernel calls are performed for T-step simulation.

IV. Bit-per-cell arrangements, bitwise summing technique, and one-step simulation

The main purpose of this section is to show an efficient simulation of the Game of Life. The idea is to arrange the state of each cell in a bit of a word, and compute the next state by bitwise operations. These techniques has been presented and the CPU implementation has been shown in [20].

A. Bit-per-cell arrangements

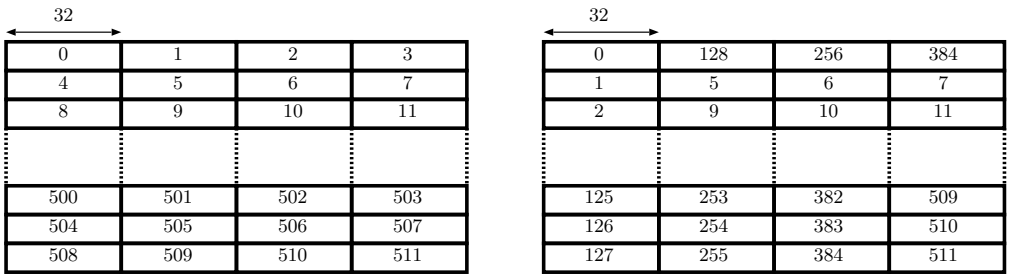
For more storage-efficient implementation of 2-dimensional array of cells, we can use *the bit-per-cell arrangement*, which arranges each cell to a bit of a word. For example, we use a 32-bit unsigned integer to store the states of consecutive 32 cells. As illustrated in Figure 3, consecutive 32 cells in the same row is arranged in a 32-bit word. In general, d consecutive cells in the same row is stored in a d-bit word and thus n cells are stored in a $\sqrt{n} \times \frac{\sqrt{n}}{d}$ array of d-bit words. We can have two address modes, *row-major* and *column-major*, to map addresses to these words. As shown in the figure, the row-major/column-major bit-per-cell arrangement maps addresses to words in row-major/column-major order, respectively. Since CUDA supports 32-bit and 64-bit words, it makes sense to set $d = 32$ or $d = 64$ when we implement the bit-per-cell arrangement in GPUs.

Note that we should use the column-major bit-per-cell arrangement although most of existing implementations use the row-major. For example, in a GPU implementation that we will show later, a block of 32×32 cells are operated in the same time. If we use the row-major order as illustrated in Figure 3 (1), the leftmost top block is arranged in stride addresses 0, 4, 8, ..., and 124. On the other hand, memory access is destined for coalesced addresses 0, 1, 2, ..., and 31, if we use the column-major order as illustrated in Figure 3 (2).

B. Bitwise summing technique

To simulate Game of Life stored in the bit-per-cell arrangements, we can retrieve the state of an individual cell by bitwise AND operation, compute the sum of neighbors by formulas (1) and (2) and write the next state by bitwise OR operation. However, this straightforward implementation of the bit-per-cell arrangement is not efficient. We should use *the bitwise summing technique*, which computes the bitwise sum of words by fundamental bitwise operations such as bitwise OR and bitwise AND. The original idea has been shown in [20]. We extend this idea for further acceleration.

Suppose that we have three d-bit words A, B, and C and we want to compute the sum of each bit. More specifically, let $A = a_{d-1}a_{d-2}\cdots a_0$, $B = b_{d-1}b_{d-2}\cdots b_0$, $C = c_{d-1}c_{d-2}\cdots c_0$. The goal is to compute the bitwise sum of A, B, and C, that is, the sum y_ix_i such that $y_i \cdot 2 + x_i = a_i + b_i + c_i$ for all i ($0 \leq i \leq d-1$). *The bitwise summing technique* computes such x_i and y_i as two words $X = x_{d-1}x_{d-2}\cdots x_0$, and $Y = y_{d-1}y_{d-2}\cdots y_0$. We will show that two words X and Y can be computed simply by bitwise XOR ($\oplus$), bitwise



(1) row-major bit-per-cell arrangement (2) column-major bit-per-cell arrangement

Fig. 3. Cell-per-bit arrangements of 128×128 array of 32-bit words

AND ($\wedge$), and bitwise OR ($\vee$). By simulating the full adder, we can compute X and Y in 7 bitwise binary operations as follows:

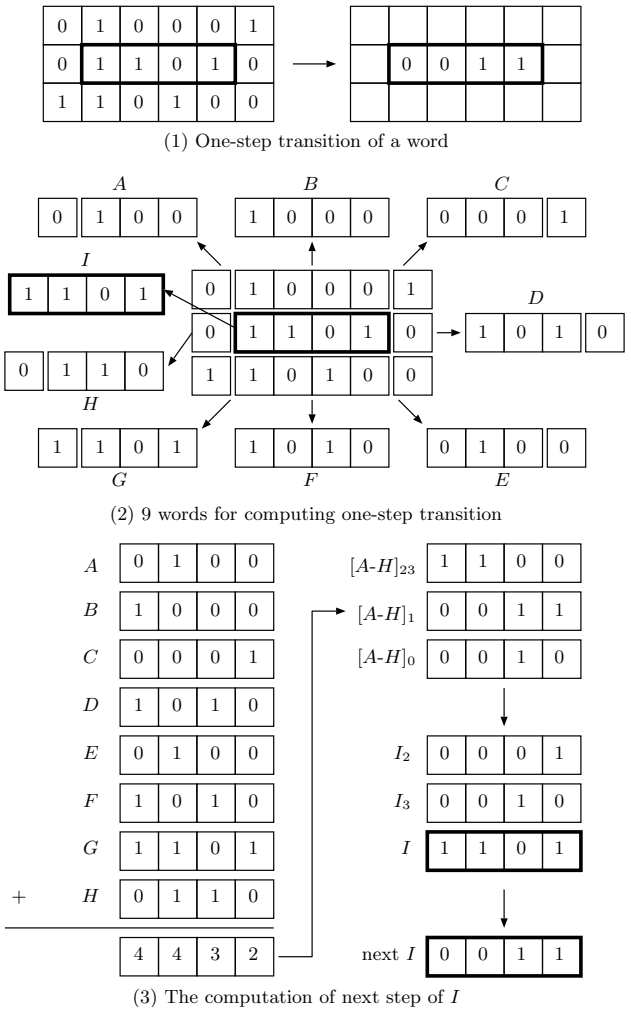
$$\begin{aligned} X &\leftarrow A \oplus B \oplus C, \\ Y &\leftarrow (A \wedge B) \vee (B \wedge C) \vee (C \wedge A). \end{aligned}$$

We can further reduce the number of bitwise operations to 5 if we use a temporal word T as follows:

$$\begin{aligned} T &\leftarrow A \oplus B, \\ X &\leftarrow T \oplus C, \\ Y &\leftarrow (A \wedge B) \vee (T \wedge C). \end{aligned}$$

To compute the next states of d cells stored in a d-bit word, the states of $2d+6$ neighboring cells are necessary as illustrated in Figure 4 (1), where $d = 4$. We store neighboring cells in eight words $A, B, \ldots, H$ as illustrated in Figure 4 (2), which are used to compute the next state of word I. For this purpose, we compute the bitwise sums as shown in Figure 4 (3) and obtain two words I_2 and I_3, where each bit of I_2 and I_3 is 1 if and only if the number of 1's in the corresponding position of eight words $A, B, \ldots, H$ is 2 and 3, respectively. Clearly, using I_2, I_3, and the current value of I, we can compute the next state of all cells in I. More specifically, $(I \wedge I_2) \vee I_3$ is the next state of each cell in I. Let $([A\text{-}H]_3, [A\text{-}H]_2, [A\text{-}H]_1, [A\text{-}H]_0)$ denote the bitwise sums of each bit of $A, B, \ldots, H$. Also, let $[A\text{-}H]_{23} = [A\text{-}H]_2 \vee [A\text{-}H]_3$. Clearly, $I_2 = 1$ if $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0) = (0, 1, 0)$ and $I_3 = 1$ if $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0) = (0, 1, 1)$. Hence, we can compute I_2 and I_3 from $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0)$.

We will show how I_2 and I_3 are computed. We first compute the bitwise sums of each of four pairs of two words. For example, by computing $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$, we obtain two bits $([AB]_1, [AB]_0)$ which represent the sum of A and B. Similarly, we can obtain $([CD]_1, [CD]_0)$, $([EF]_1, [EF]_0)$, and $([GH]_1, [GH]_0)$. After that, we compute the sum of pair $([AB]_1, [AB]_0)$ and $([CD]_1, [CD]_0)$, and obtain three bits $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0)$. This can be done by computing the sums from the least significant bit. Similarly, we obtain the sum $([E\text{-}H]_2, [E\text{-}H]_1, [E\text{-}H]_0)$. Finally, we compute the sum of $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0)$ and $([EH]_2, [EH]_1, [EH]_0)$ and obtain three bits $([AH]_{23}, [AH]_1, [AH]_0)$. From these three bits, the values of I_2 and I_3 can be obtained and then, the next states of I can be computed. The details of an algorithm, Algorithm



(1) One-step transition of a word

(2) 9 words for computing one-step transition

(3) The computation of next step of I

Fig. 4. The computation of the next states of 4 cells in a 4-bit word by Algorithm SINGLE-WORD

SINGLE-WORD that computes I_2, I_3, and the next state of I are as follows:

[Algorithm SINGLE-WORD]
1. $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$
2. $([CD]_1, [CD]_0) \leftarrow (C \wedge D, C \oplus D)$
3. $([EF]_1, [EF]_0) \leftarrow (E \wedge F, E \oplus F)$
4. $([GH]_1, [GH]_0) \leftarrow (G \wedge H, G \oplus H)$
// $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0]) \leftarrow ([AB]_1, [AB]_0)$
// $+([CD]_1, [CD]_0)$
5. $[A\text{-}D]_0 \leftarrow [AB]_0 \oplus [CD]_0$
7. $[A\text{-}D]_1 \leftarrow [AB]_1 \oplus [CD]_1 \oplus ([AB]_0 \wedge [CD]_0)$
8. $[A\text{-}D]_2 \leftarrow [AB]_1 \wedge [CD]_1$
// $([E\text{-}H]_2, [E\text{-}H]_1, [E\text{-}H]_0]) \leftarrow ([EF]_1, [EF]_0)$
// $+([GH]_1, [GH]_0)$
9. $[EH]_0 \leftarrow [EF]_0 \oplus [GH_0]$
10. $[EH]_1 \leftarrow [EF]_1 \oplus [GH]_1 \oplus ([EF]_0 \wedge [GH]_0)$
11. $[EH]_2 \leftarrow [EF]_1 \wedge [GH]_1$
// $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0]) \leftarrow ([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0])$
// $+([E\text{-}H]_2, [E\text{-}H]_1, [E\text{-}H]_0])$
12. $[A\text{-}H]_0 \leftarrow [A\text{-}D]_0 \oplus [E\text{-}H]_0$
13. $X \leftarrow [A\text{-}D]_0 \wedge [E\text{-}H]_0$
14. $Y \leftarrow [A\text{-}D]_1 \oplus [E\text{-}H]_1$
15. $[A\text{-}H]_1 \leftarrow X \oplus Y$
16. $[A\text{-}H]_{23} \leftarrow [A\text{-}D]_2 \vee [E\text{-}H]_2$
$\vee([A\text{-}D]_1 \wedge [E\text{-}H]_1) \vee (X \wedge Y)$
// $(I, I_2, I_3) \leftarrow (I, [A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0)$
17. $Z \leftarrow \overline{[A\text{-}H]_{23}} \wedge [A\text{-}H]_1$
18. $I_2 \leftarrow \overline{[A\text{-}H]_0} \wedge Z$
19. $I_3 \leftarrow [A\text{-}H]_0 \wedge Z$
20. $I \leftarrow (I \wedge I_2) \vee I_3$

Note that, when we compute $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0]) \leftarrow ([AB]_1, [AB]_0) + ([CD]_1, [CD]_0)$, the values of $([AB]_1, [AB]_0)$ and $([CD]_1, [CD]_0)$ can not be $(1, 1)$. Hence, $[A\text{-}D]_2$ can be computed by formula $[AB]_1 \wedge [CD]_1$.

Let us evaluate the total number of binary operations and unary operations performed in this algorithm for bit-per-cell arrangement. For computing $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$, two binary operations are performed. Thus, the sums of four pairs can be computed by 8 binary operations. Five binary operations are performed for computing the sum of two bits, $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0]) \leftarrow ([AB]_1, [AB]_0) + ([CD]_1, [CD]_0)$. This computation is executed twice, and thus, 10 binary operations are performed. For computing $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0])$, 9 binary operations are performed. Finally, (I, I_2, I_3) is computed in 5 binary operations and 2 unary operations. Thus, the total number of operations is $4 \times 2 + 2 \times 5 + 9 + 5 + 2 = 34$. Hence we have,

Lemma 1: The next states of cells stored in a word by the bit-per-cell arrangement can be computed in 34 operations.

Let us implement bitwise summing technique in the GPU. Since CUDA supports 32-bit and 64-bit bitwise operations, it makes sense to use a 32-bit or 64-bit integer to store 32 or 64 cells. Suppose that we use 64-bit integers to store cells. Each thread is assigned a word storing 64 cells, and it is responsible for computing the next states of these cells. We can invoke a CUDA kernel with $\frac{n}{64 \cdot 32}$ CUDA blocks with 32 threads each for n cells. Each word with 64 cells and 8 neighboring words are read by a thread assigned to it. The thread computes 8 words $A, B, \ldots, H$ from these words, and compute the next state of I by 34 operations. After that, it writes the resulting next states of I in the global memory and terminates. After all threads terminate, the CUDA kernel terminates. In this way, one-step simulation is performed by a single CUDA kernel call. The same CUDA kernel call is repeatedly performed T times to complete the T-step simulation.

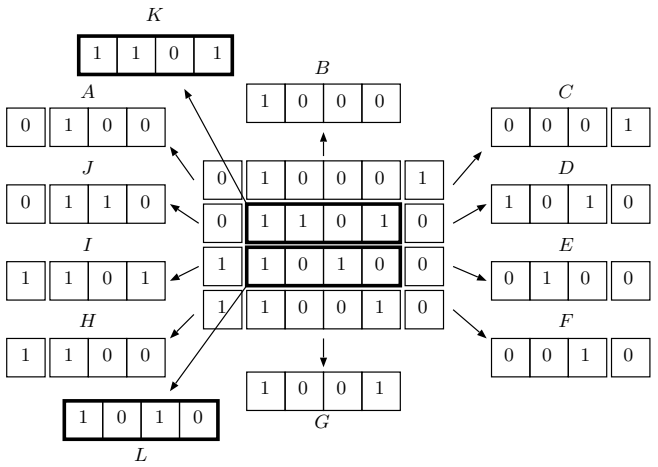


Fig. 5. Illustrating 12 words for computing next states of cells in two words by Algorithm DOUBLE-WORD

V. Our acceleration techniques

The main purpose of this section is to show our new ideas and techniques for further acceleration of simulation of the Game of Life.

A. Bitwise summing technique for two words

We can reduce the number of operations if next states of cells in two words are computed at the same time. If we just execute Algorithm SINGLE-WORD twice, we need 68 operations. We will show that it can be reduced to 59 operations by sharing the computation for two words. For this purpose, we partition the cells as illustrated in Figure 5. We compute the next states of cells in two words K and L in the figure at the same time. For updating K, the sum of words A, B, C, D, E, I, J, L is computed. Also, the sum of D, E, F, G, H, I, J, K is computed for word L. More specifically, we compute $([A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0])$ and $([D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0)$. Clearly, four words D, E, I, J are included in both sets of words. Hence, by computing the sum of these words first, we can reduce the total number of operations. Once we have $(K, [A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0])$, we can compute (K, K_2, K_3) where K stores next states of K, and each bit of K_2 and K_3 is 1 if and only if the number of 1's in the corresponding position of eight words A, B, C, D, E, I, J, L is 2 and 3, respectively. Similarly, we can obtain (L, L_2, L_3) using $(L, [D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0])$.

Using this idea, next states of cells in two words can be computed by Algorithm DOUBLE-WORD as follows:

[Algorithm DOUBLE-WORD]

1. $([DE]_1, [DE]_0) \leftarrow (D \wedge E, D \oplus E)$
2. $([IJ]_1, [IJ]_0) \leftarrow (I \wedge J, I \oplus J)$
3. $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$
4. $([CL]_1, [CL]_0) \leftarrow (C \wedge L, C \oplus L)$
5. $([FG]_1, [FG]_0) \leftarrow (F \wedge G, F \oplus G)$
6. $([HK]_1, [HK]_0) \leftarrow (H \wedge K, H \oplus K)$
7. $([DEIJ]_2, [DEIJ]_1, [DEIJ]_0]) \leftarrow ([DE]_1, [DE]_0)$
   $+([IJ]_1, [IJ]_0)$
8. $([ABCL]_2, [ABCL]_1, [ABCL]_0]) \leftarrow ([AB]_1, [AB]_0)$
   $+([CL]_1, [CL]_0)$
9. $([FGHK]_2, [FGHK]_1, [FGHK]_0]) \leftarrow ([FG]_1, [FG]_0)$
   $+([HK]_1, [HK]_0)$
10. $([A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0])$
    $\leftarrow ([ABCL]_2, [ABCL]_1, [ABCL]_0])$
    $+([DEIJ]_2, [DEIJ]_1, [DEIJ]_0])$
11. $([D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0])$
    $\leftarrow ([FGHK]_2, [FGHK]_1, [FGHK]_0])$
    $+([DEIJ]_2, [DEIJ]_1, [DEIJ]_0])$
12. $(K, K_2, K_3) \leftarrow (K, [A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0])$
13. $(L, L_2, L_3) \leftarrow (I, [D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0])$

Let us evaluate the total number of operations. Each of Lines 1-6 can be done in two binary operations. Lines 7-9 can be done in 5 binary operations each. Lines 10 and 11 can be performed in 9 binary operations each. Finally, lines 12 and 13 takes 5 binary operations and two unary operations. Thus, the total number of operations is $6 \times 2 + 3 \times 5 + 2 \times 9 + 2 \times 7 = 59$, and we have,

Lemma 2: The next states of cells stored in two words by the bit-per-cell arrangement can be computed in 59 operations

Similarly to the GPU implementation using the global memory, we can implement the algorithm for Lemma 2 in CUDA programming model. For example, a CUDA kernel with $\frac{n}{(64 \cdot 32 \cdot 2)}$ CUDA blocks with 32 threads each is repeatedly invoked. Each thread is responsible for computing the next states of two words. Since the memory access to the global memory can be shared for updating two words, we can further accelerate the computation.

B. Multiple-step simulation using the shared memory

We can accelerate the computation if multiple steps simulation is performed on the shared memory. More specifically, a CUDA block is assigned to multiple words, say, 32 words. It copies words storing the cell states to the shared memory and simulates multiple steps on the shared memory. The resulting states are copied to the global memory.

If multiple-step simulation is performed in a block of 2-dimensional array, cells in the boundary of the block may not have correct states. More specifically, suppose that we have a block of $d \times d$ cells in a large 2-dimensional array of cells. Since we do not have the states of cells outside of the block, we simply assume that those cells always take state 0.

We can say that the boundary cells are *dirty* after one-step simulation in the sense that their states may not be correct, because at least one of neighboring cells of each boundary cell is not taken into account. Also, cells inside the boundary are *clean* in the sense that their states are guaranteed to be correct. After another step simulation, neighboring cells of the dirty cells, that is, the boundary cells of clean cells become

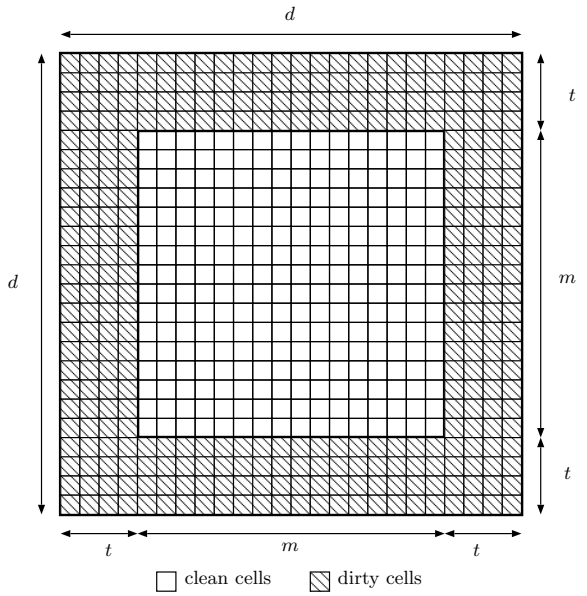


Fig. 6. Dirty cells

dirty. In general, cells in the distance t from the boundary become dirty after t-step simulation and $m \times m$ cells are clean, where $m = d - 2t$, as illustrated Figure 6.

To simulate multiple steps of all cells, the $\sqrt{n} \times \sqrt{n}$ 2-dimensional array in the global memory is partitioned into $\frac{\sqrt{n}}{m} \times \frac{\sqrt{n}}{m}$ *slices* of size $m \times m$ each as illustrated in Figure 7. Each slice is expanded by t cells for every direction, and we obtain a $d \times d$ block. A CUDA block is assigned to a block and performs t-step simulation using the shared memory. For this purpose, it copies the states of $d \times d$ cells in a block to the shared memory. Note that each row of $d \times d$ cells is stored in one or two d-bit words. Thus, we read at most $2d$ words to copy $d \times d$ cells from the global memory. In the shared memory, t-step simulation is performed. After that, the resulting states in the $m \times m$ slice are written in the global memory. Similarly, we need to perform write operations for at most $2m$ words to the global memory. Since this t-step simulation for all blocks must be completed before the next t-step simulation is performed. Hence, each t-step simulation must be performed by one CUDA kernel call and thus T-step simulation can be done by $\frac{T}{t}$ CUDA kernel calls.

Clearly, we should use the column-major bit-per-cell arrangement because cells in a block are arranged in neighboring addresses. More specifically, the d or $2d$ words in the global memory storing $d \times d$ cells are stride if we use the row-major bit-per-cell arrangement, On the other hand, the $2d$ words are in two consecutive addresses of length d each if the column-major bit-per-cell arrangement is used.

If Algorithm DOUBLE-WORD is implemented using the shared memory as it is, memory access to the shared memory has bank conflicts. The shared memory of Maxwell architecture has 32 memory banks with 32-bit width [25]. If we store 64-bit data in the shared memory, each of them are stored in two adjacent banks. In other words, a pair of two adjacent banks are used to store a 64-bit number. Hence, we can think that the shared memory has 16 memory banks with 64-bit

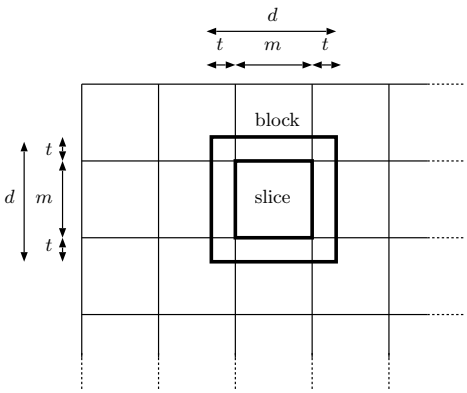


Fig. 7. A $d \times d$ slice and an $m \times m$ block in a large 2-dimensional array

width. Further, if each of 32 threads in a warp access to a 64-bit number in the shared memory, 16 threads in the first half warp try to access them, and then 16 threads in the second half warp do the same thing. Suppose that 64 64-bit numbers that constitute a block (Figure 8 (1)) and let $a(i)$ $(0 \leq i \leq 63)$ be the i-th 64-bit number. is stored in the 16 banks of the shared memory as it is (Figure 8 (2)). If Algorithm DOUBLE-WORD is executed using 32 threads in a warp, the first warp may access 64-bit numbers $a(2), a(4), a(6), \ldots, a(32)$. As we can see in Figure 8 (2), two numbers are in the same bank.

To avoid bank conflicts, we use the shift arrangement as illustrated in Figure 8 (3). In the shift arrangement, the second row and the fourth row are shifted by one. We can confirm that $a(2), a(4), a(6), \ldots, a(32)$ are arranged in distinct banks. The first warp also may access, sets of 16 numbers

- $a(0), a(2), a(4), \ldots, a(30)$, and
- $a(1), a(3), a(5), \ldots, a(31)$.

We can confirm that 16 numbers in each set are in distinct banks. Thus, we can avoid bank conflicts by the shift arrangement.

We can observe that, we should select an appropriate value of t $(1 \leq t < \frac{d}{2})$ for fixed n and d that minimizes the running time. We assume that the cost for computing the next state of d cells stored in a word is one unit. Also, let c be the cost of miscellaneous overhead for dispatching CUDA blocks and reading/writing the states of d cells in the global memory. Under this assumption, we can write that the cost of t-step simulation of a slice of size $m \times m$ is $t + c$. Hence, the cost of T-step simulation of $\sqrt{n} \times \sqrt{n}$ cells is:

$$\frac{T}{t} \times \frac{n}{m^2} \times (t + c) \quad = \quad \frac{nT(t + c)}{t(d - 2t)^2}$$

This cost is minimized when $4t^2 + 6ct - dc = 0$, that is,

$$t \quad = \quad \frac{\sqrt{9c^2 + 4dc} - 3c}{4d^2}.$$

Clearly, t is an increasing function of c and the value of t is in the range $[0, \frac{d}{6}]$. Intuitively, this is reasonable because the overhead c is large, we should minimize the number $\frac{T}{t}$ of CUDA kernel calls.

C. Further acceleration using warp shuffle

The memory access latency of the shared memory is not small [12]. Hence, if we can implement words of cells as registers, we can further accelerate the computation. We will show that t-step simulation can be done using registers without using the shared memory.

The algorithm is almost the same as in Subsection V-B, which uses the shared memory for t-step simulation. Instead of using the shared memory, we use registers which can be accessed faster than the shared memory. However, registers are assigned to a thread, and they can be accessed only by the assigned thread. Hence, we use a warp shuffle instruction, which copies registers of threads in the same warp, as illustrated in Figure 9. First, each thread copies two words storing cells from the global memory. For one-step simulation, each thread copies registers of two neighboring threads. After that, one-step simulation is performed for two words. This operation is repeated t times for t-step simulation. The resulting states of cells are copied from the registers to the global memory.

VI. Experimental results

The main purpose of this section to show the performance of algorithms for Game of Life.

We have evaluated the running time of 1024-step simulation for a 16384×16384 $(2^{14} \times 2^{14})$ array. The array is wrap-around in the sense that cells in the top-row and the bottom-row are neighbor. Also,the leftmost-column and the rightmost-column are neighbor. We have used GeForce GTX TITAN X and Intel Xeon X7460 CPU (2.66GHz) for the experiment. GeForce GTX TITAN X has 16 streaming multiprocessors with 192 cores each.

Table I shows the running time of straightforward implementations, for the word-per-cell and the bit-per-cell. In the word-per-cell, we have used 8-bit unsigned characters to store the states cells. In other words, a 2-dimensional array of 16384×16384 unsigned characters are used and evaluated formulas (1) and (2) to obtain the next states. The CPU implementation of the word-per-cell is obvious. The CPU computes the next state of every cell one by one. To implement the word-per-cell in the GPU, each cell is assigned one thread. More specifically, a CUDA kernel computing 1-step transition invokes 2^{23} CUDA blocks with 32 threads each. The 2-dimensional array storing the states of cells are arranged in the global memory. Each thread reads the states of cells necessary compute the next state of an assigned cell. It computes the next cell by formulas (1) and (2) and writes the resulting state in the global memory. Note that, a CUDA kernel call can compute only 1-step transition and thus 1024 CUDA kernel calls are necessary to compute the states after 1024 steps.

We have used 64-bit unsigned long long integers for the bit-per-cell arrangement. Hence, $16384 \times 16384 = 2^{28}$ cells are implemented in $16384 \times 256 = 2^{22}$ words. To see the difference of performance of Algorithms SINGLE-WORD and DOUBLE-WORD, we have implemented both algorithms. To compute the next states of all cells, the CPU executes SINGLE-WORD 2^{22} times. It also need to execute DOUBLE-WORD 2^{21} times for 1-step simulation. To compute the next states

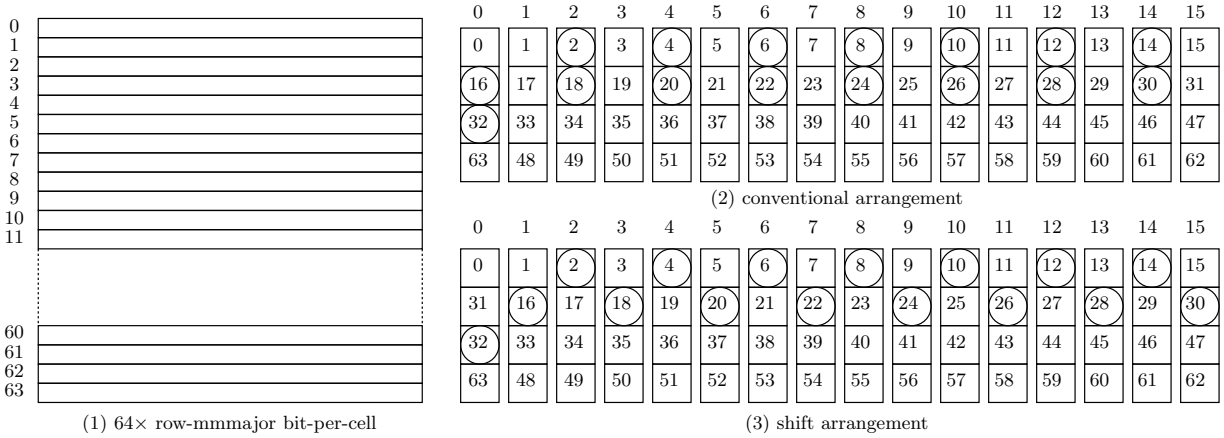


Fig. 8. The padding technique to avoid bank conflicts

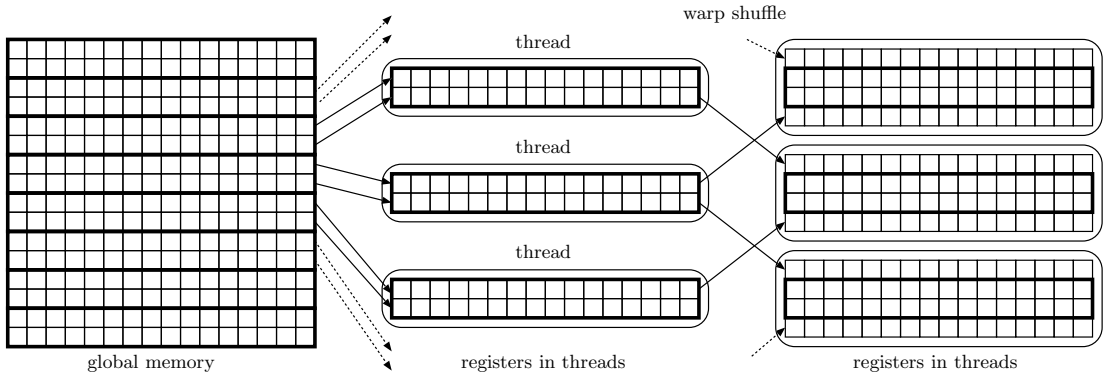


Fig. 9. Copying words storing cells using a warp shuffle instruction

of all cells by Algorithm SINGLE-WORD, a CUDA kernel of 2^{17} CUDA blocks of 32 threads each is invoked. Also, for 1-step simulation by Algorithm DOUBLE-WORD, 2^{16} CUDA blocks of 32 threads each are used. These 1-step simulations are repeated 1024 times for 1024-step simulation.

From Table I, the GPU implementations can accelerate the computation with a speed-up factor of more than 100 for the word-per-cell arrangement and the bit-per-cell-arrangement using Algorithm SINGLE-WORD. Since the state of one cell is stored using 8 bits in the word-per-cell, we can expect that an implementation of the bit-per-cell is 8 times faster than that of the word-per-cell Quite surprisingly, the bit-per-cell implementation can be more than 20 times faster than the word-per-cell implementation. This is because memory access to 8-bit words is not efficient in 64-bit processor architecture Thus, we should not use word-per-cell arrangement and must use bit-per-cell arrangement for 64-bit words. Further, we can see that Algorithm DOUBLE-WORD on the CPU is much faster than Algorithm SINGLE-WORD. On the other hand, Algorithm DOUBLE-WORD on the GPU does not achieve an improvement over Algorithm SINGLE WORD. This is because a straightforward implementation of Algorithm DOUBLE-WORD involves stride memory access to the global memory, while that of Algorithm SINGLE-WORD does not.

For further acceleration, we implemented multiple-step simulation with bit-per-cell arrangement using the shared memory and the registers on the GPU. Since we want to avoid barrier synchronization using `__syncthreads()`, we use CUDA blocks with one single warp of 32 threads each. Also, we implemented simulation of the Game of Life for a block with 32×32 cells and with 64×64 cells as follows:

32×32 block: A block of size 32×32 is implemented using 32 32-bit unsigned integers, each of which stores the states of 32 cells. A CUDA block with 32 threads is assigned 32×32 cells. Each thread computes t-step transition of 32 cells stored in a 32-bit unsigned integer by repeating Algorithm SINGLE-WORD.

64×64 block: A block of size 64×64 is implemented using 64 64-bit unsigned long long integers, each of which stores the states of 64 cells. Since a warp of 32 threads are used for 64 words, we execute SINGLE-WORD twice or DOUBLE-WORD once to compute 1-step transition. Each thread repeat this t times to complete t-step transition.

To find the best value of the number t of steps computed by a single CUDA kernel call, we evaluated the running time for $t = 2, 4, 8$, and 16. Recall that the 2-dimensional array of size 16384×16384 is partitioned into $\frac{16384}{m} \times \frac{16384}{m}$ slices of size $m \times m$ each where $m = d - 2t$ and $d = 32$ for 32×32 blocks and $d = 64$ for 64×64 blocks. Hence, it makes no sense to perform 16-step simulation for 32×32 blocks, because $m = d - 2t = 0$.

Table II shows the running time of 1024-step simulation

TABLE I. THE RUNNING TIME (IN SECONDS) OF CPU IMPLEMENTATION AND GPU IMPLEMENTATION (GLOBAL MEMORY)

	word-per-cell	bit-per-cell	
		SINGLE-WORD	DOUBLE-WORD
Intel Xeon X7460 CPU	2151	84.8	58.3
Nvidia GeForce GTX TITAN X	119.3	0.574	0.672
speed-up	111	147	86.8

TABLE II. THE RUNNING TIME (IN SECONDS) OF GPU IMPLEMENTATIONS OF MULTIPLE-STEP SIMULATION

	GPU (shared memory)			GPU (register+warp shuffle)		
	32×32 block	64×64 block		32×32 block	64×64 block	
steps	SINGLE-WORD	SINGLE-WORD	DOUBLE-WORD	SINGLE-WORD	SINGLE-WORD	DOUBLE-WORD
2	0.607	0.439	0.385	0.543	0.390	0.379
4	0.482	0.301	0.237	0.386	0.216	0.194
8	0.850	0.356	0.248	0.545	0.197	0.163
16	-	0.762	0.511	-	0.377	0.295

of the Game of Life with 16384×16384 cells. In most cases, implementations of 64×64 blocks are faster than that of 32×32 blocks, because 64-bit memory access can maximize the memory access bandwidth for the global memory and the shared memory. Also, implementations using Algorithm DOUBLE-WORD are faster than the shared memory implementations except for a few exceptions. From the table, 8-step simulation with 64×64 block using Algorithm DOUBLE-WORD runs 0.163 seconds, which is the minimum over all implementations that we have developed.

VII. CONCLUSION

The main purpose of this paper presents several techniques for accelerating the simulation of the Conway's Game of Life. In particular, we have presented techniques of (1) sharing the sum computation for two words, (2) multiple-step simulation, and (3) register with warp shuffle instructions. The best implementation performs 1024-step simulation of 16384×16384 cells in 0.163 seconds on GeForce GTX TITAN X GPU. This implies that it achieves 1.69×10^{12} updates per second, which is more than 68 times faster than previously presented best implementation.

REFERENCES

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

[2] V. Podlozhnyuk, "Image convolution with CUDA," July 2007.

[3] Y. Takeuchi, D. Takafuji, Y. Ito, and K. Nakano, "Ascii art generation using the local exhaustive search on the GPU," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 194–200.

[4] Y. Zhang, J. L. Recker, R. Ulichney, I. Tastl, and J. D. Owens, "Plane-dependent error diffusion on a GPU," in *Proc. SPIE*, vol. 8295, Jan. 2012.

[5] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[6] P. Steffen, R. Giegerich, and M. Giraud, "GPU parallelization of algebraic dynamic programming," in *Proc. of International Conference on Parallel Processing and Applied Mathematics: Part II*, Sept. 2009, pp. 290–299.

[7] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "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.

[8] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 7.0," Mar 2015.

[9] ——, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[10] 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.

[11] K. Nakano, "Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models," *IEICE Trans. on Information and Systems*, vol. E96-D, no. 12, pp. 2626–2634, 2013.

[12] S. Okamoto, Y. Ito, K. Nakano, and J. L. Bordim, "Thorough evaluation of GPU shared memory load and store instructions,," in *Proc. of International Symposium on Computing and Networking*, Dec. 2015, pp. 614–616.

[13] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Kepler GK110, whitepaper," 2012.

[14] NVIDIA Corportaion, "NVIDIA GeForce GTX980 whitepaper," 2014.

[15] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," in *Proc. International Symposium on Computer Architecture*, 2012, pp. 49–60.

[16] Y. Yang, Z. Guan, H. Sun, and Z. Chen, "Accelerating RSA with fine-grained parallelism using GPU," in *Proc. of Information Security Practice and Experience (LNCS)*, vol. 9065, 2015, pp. 454–468.

[17] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, Oct. 1970.

[18] A. Adamatzky, *Game of Life Cellular Automata*. Springer, 2015.

[19] M. Fisher, "Conway's Game of Life on GPU using CUDA," Mar 2013. [Online]. Available: http://www.marekfiser.com/Projects/Conways-Game-of-Life-on-GPU-using-CUDA

[20] N. Tsuda, "Acceleration of Game of Life by the bit operation (bit-board)," December 2012, in Japanese. [Online]. Available: http://vivi.dyndns.org/tech/games/LifeGame.html

[21] MathWorks, "Stencil operations on a GPU," 2015. [Online]. Available: https://www.mathworks.com/examples/parallel-computing/398-stencil-operations-on-a-gpu

[22] K. S. Perumalla and B. G. Aaby, "Data parallel execution challenges and runtime performance of agent simulations on GPUs," in *Proc. of Spring Simulation Multiconference*, 2008, pp. 116–123.

[23] M. Bailey and S. Cunningham, "A hands-on environment for teaching GPU programming," in *Proc. of SIGCSE Technical Symposium on Computer Science Education*, 2007, pp. 254–258.

[24] NVIDIA Corporation, "Geforce GTX TITAN X," 2015. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x

[25] ——, "Tuning CUDA applications for Maxwell," 2015.

A Parallel Algorithm for LZW decompression, with GPU implementation

Shunji Funasaka, Koji Nakano, and Yasuaki Ito

Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashihiroshima 739-8527 Japan

Abstract. The main contribution of this paper is to present a parallel algorithm for LZW decompression and to implement it in a CUDA-enabled GPU. Since sequential LZW decompression creates a dictionary table by reading codes in a compressed file one by one, its parallelization is not an easy task. We first present a parallel LZW decompression algorithm on the CREW-PRAM. We then go on to present an efficient implementation of this parallel algorithm on a GPU. The experimental results show that our parallel LZW decompression on GeForce GTX 980 runs up to 69.4 times faster than sequential LZW decompression on a single CPU. We also show a scenario that parallel LZW decompression on a GPU can be used for accelerating big data applications.

Keywords: Data compression, big data, parallel algorithm, GPU, CUDA

1 Introduction

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [4]. 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. 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.

There is no doubt that data compression is one of the most important tasks in the area of computer engineering. In particular, almost all image data are stored in files as compressed data formats. There are basically two types of image compression methods: *lossy* and *lossless* [9]. Lossy compression can generate smaller files, but some information in original files are discarded. Hence, decompression of lossy compressed images does not generate files identical to the original images. On the other hand, lossless compression creates compressed files, from which we can obtain the exactly same original files by decompression. Hence, lossless compression can be used far more than images. In this paper, we focus on LZW compression, which is one of the most well known patented lossless

compression method [11] used in Unix file compression utility "compress" and in GIF image format. Also, LZW compression option is included in TIFF file format standard [1], which is commonly used in the area of commercial digital printing. However, LZW compression and decompression are hard to parallelize, because they use dictionary tables created by reading input data one by once. In [10], a CUDA implementation of LZW compression has been presented. But, it achieved only a speedup factor less than 2 over the CPU implementation using MATLAB. Also, several GPU implementations of dictionary based compression methods have been presented [6, 8]. As far as we know, no parallel LZW decompression using GPUs has not been presented. In particular, decompression may be performed more frequently than compression; each image is compressed and written in a file once, but it is decompressed whenever the original image is used. Hence, we can say that LZW decompression is more important than the compression.

The main contribution of this paper is to present a parallel algorithm for LZW decompression and the GPU implementation. We first show that a parallel algorithm for LZW decompression on the CREW-PRAM [2], which is a traditional theoretical parallel computing model with a set of processors and a shared memory. We will show that LZW decomposition of a string of m codes can be done in $O(L_{\max} + \log m)$ time using $\max(k, m)$ processors on the CREW-PRAM, where $L_{\max}$ is the maximum length of characters assigned to a code. We then go on to show an implementation of this parallel algorithm in CUDA architecture. The experimental results using GeForce GTX 980 GPU and Intel Xeon CPU X7460 processor show that our implementation on a GPU achieves a speedup factor up to 69.4 over a single CPU.

Let us consider the following scenario to use LZW compression and decompression. Suppose that we have a set of bulk data such as images or text stored in a storage of a host computer with a GPU. A user gives a query to the set of bulk data and all data must be processed to answer the query. To accelerate the computation for the query, data are transferred to the GPU through the host computer and they are processed by parallel computation on the GPU. For the purpose of saving storage space and data transfer time, data are stored in the storage as LZW compressed format. If this is the case, compressed data must be decompressed using the host computer or using the GPU before the query processing is performed. We will show that, since LZW decompression can be done very fast in the GPU by our parallel algorithm, it makes sense to store compressed data in a storage and to decompress them using the GPU.

2 LZW compression and decompression

The main purpose of this section is to review LZW compression/decompression algorithms. Please see Section 13 in [1] for the details.

The LZW (Lempel-Ziv & Welch) [12] compression algorithm converts an input string of characters into a string of codes using a string table that maps strings into codes. If the input is an image, characters may be 8-bit integers. It

reads characters in an input string one by one and adds an entry in a string table (or a dictionary). In the same time, it writes an output string of codes by looking up the string table. Let $X = x_0x_1 \cdots x_{n-1}$ be an input string of characters and $Y = y_0y_1 \cdots y_{m-1}$ be an output string of codes. For simplicity of handling the boundary case, we assume that an input is a string of 4 characters a, b, c, and d. Let S be a string table, which determines a mapping of a string to a code, where codes are non-negative integers. Initially, $S(a) = 0$, $S(b) = 1$, $S(c) = 2$, and $S(d) = 3$. By procedure AddTable, new code is assigned to a string. For example, if AddTable(cb) is executed after initialization of S, we have $S(cb) = 4$. The LZW compression algorithm is described as follows:

[LZW compression algorithm]
1 for $i \leftarrow 0$ to $n-1$ do
2 if($\Omega \cdot x_i$ is in S)
3 $\Omega \leftarrow \Omega \cdot x_i$;
4 else
5 Output($S(\Omega)$); AddTable($\Omega \cdot x_i$); $\Omega \leftarrow x_i$;
6 Output($S(\Omega)$);

In this algorithm, Ω is a variable to store a string. Also, "$\cdot$" denotes the concatenation of strings/characters.

Table 1 shows how the compression process for an input string *cbcbcbcda*. First, since $\Omega \cdot x_0 = c$ is in S, $\Omega \leftarrow c$ is performed. Next, since $\Omega \cdot x_1 = cb$ is not in S, Output($S(c)$) and AddTable(cb) are performed. More specifically, $S(c) = 2$ is output and we have $S(cb) = 4$. Also, $\Omega \leftarrow x_1 = b$ is performed. It should have no difficulty to confirm that 214630 is output by this algorithm.

Table 1. String table S, string stored in Ω, and output string Y for $X = cbcbcbcda$

i	0	1	2	3	4	5	6	7	8	-
x_i	c	b	c	b	c	b	c	d	a	
Ω	-	c	b	c	cb	c	cb	cbc	d	a
S	-	$cb:4$	$bc:5$	-	$cbc:6$	-	-	$cbcd:7$	$da:8$	-
Y	-	2	1	-	4	-	-	6	3	0

Next, let us show LZW decompression algorithm. Let C be *the code table*, the inverse of string table S. For example if $S(cb) = 4$ then $C(4) = cb$. Initially, $C(0) = a$, $C(1) = b$, $C(2) = c$, and $C(3) = d$. Also, let $C_1(i)$ denote the first character of code i. For example $C_1(4) = c$ if $C(4) = cb$. Similarly to LZW compression, the LZW decompression algorithm reads a string Y of codes one by one and adds an entry of a code table. In the same time, it writes a string X of characters. The LZW decompression algorithm is described as follows:

[LZW decompression algorithm]

```
1  Output(C(y_0));
2  for i ← 1 to n − 1 do
3      if(y_i is in C)
4          Output(C(y_i)); AddTable(C(y_{i−1}) · C_1(y_i));
5      else
6          Output(C(y_{i−1}) · C_1(y_{i−1})); AddTable(C(y_{i−1}) · C_1(y_{i−1}));
```

Table 2 shows the decompression process for a code string 214630. First, $C(2) = c$ is output. Since $y_1 = 1$ is in C, $C(1) = b$ is output and AddTable(cb) is performed. Hence, $C(4) = cb$ holds. Next, since $y_2 = 4$ is in C, $C(4) = cb$ is output and AddTable(bc) is performed. Thus, $C(5) = bc$ holds. Since $y_3 = 6$ is not in C, $C(y_2) \cdot C_1(y_2) = cbc$ is output and AddTable(cbc) is performed. The reader should have no difficulty to confirm that *cbcbcbcda* is output by this algorithm.

Table 2. Code table C and the output string for 214630

i	0	1	2	3	4	5
y_i	2	1	4	6	3	0
C	-	4 : cb	5 : bc	6 : cbc	7 : $cbcd$	8 : da
X	c	b	cb	cbc	d	a

3 Parallel LZW decompression

This section shows our parallel algorithm for LZW decompression.

Again, let $X = x_0x_1 \cdots x_{n-1}$ be a string of characters. We assume that characters are selected from an alphabet (or a set) with k characters $\alpha(0), \alpha(1)$, ..., $\alpha(k-1)$. We use $k = 4$ characters $\alpha(0) = a$, $\alpha(1) = b$, $\alpha(2) = c$, and $\alpha(3) = d$, when we show examples as before. Let $Y = y_0y_1 \cdots y_{m-1}$ denote the compressed string of codes obtained by the LZW compression algorithm. In the LZW compression algorithm, each of the first $m-1$ codes $y_0, y_1, \ldots, y_{m-2}$ has a corresponding AddTable operation. Hence, the argument of code table C takes an integer from 0 to $k + m - 2$.

Before showing the parallel LZW compression algorithm, we define several notations. We define pointer table p using code table Y as follows:

$$p(i) = \begin{cases} \text{NULL} & \text{if } 0 \le i \le k-1 \\ y_{i-k} & \text{if } k \le i \le k+m-1 \end{cases} \quad (1)$$

We can traverse pointer table p until we reach NULL . Let $p^0(i) = i$ and $p^{j+1}(i) = p(p^j(i))$ for all $j \ge 0$ and i. In other words, $p^j(i)$ is the code where we reach from code i in j pointer traversing operations. Let $L(i)$ be an integer satisfying

$p^{L(i)}(i) = \text{NULL}$ and $p^{L(i)-1}(i) \neq \text{NULL}$. Also, let $p^*(i) = p^{L(i)-1}(i)$. Intuitively, $p^*(i)$ corresponds to the dead end from code i along pointers. Further, let $C_l(i)$ $(0 \leq i \leq k+m-2)$ be a character defined as follows:

$$C_l(i) = \begin{cases} \alpha(i) & \text{if } 0 \leq i \leq k-1 \\ \alpha(p^*(i+1)) & \text{if } k \leq k+m-2 \end{cases} \tag{2}$$

It should have no difficulty to confirm that $C_l(i)$ is the last character of $C(i)$, and $L(i)$ is the length of $C(i)$. Using C_l and p, we can define the value of $C(i)$ as follows:

$$C(i) = C_l(p^{L(i)-1}(i)) \cdot C_l(p^{L(i)-2}(i)) \cdots C_l(p^0(i)). \tag{3}$$

Table 3 shows the values of p, p^*, L, C_l, and C for $Y = 214630$.

Table 3. The values of p, p^*, l, C_l, and C for $Y = 214630$

i	0	1	2	3	4	5	6	7	8	9
$p(i)$	NULL	NULL	NULL	NULL	2	1	4	6	3	0
$p^*(i)$	-	-	-	-	2	1	2	2	3	0
$L(i)$	1	1	1	1	2	2	3	4	2	-
$C_l(i)$	a	b	c	d	b	c	c	d	a	-
$C(i)$	a	b	c	d	cb	bc	cbc	$cbcd$	da	-

We are now in a position to show parallel LZW decompression on the CREW-PRAM. Parallel LZW decompression can be done in two steps as follows:

Step 1 Compute L, p^*, and C_l from code string Y.
Step 2 Compute X using p, C_l and L.

In Step 1, we use k processors to initialize the values of $p(i), C_l(i)$, and $L(i)$ for each i $(0 \leq i \leq k-1)$. Also, we use m processors and assign one processor to each i $(k \leq i \leq 2k+m-1)$, which is responsible for computing the values of $L(i), p^*(i)$, and $C_l(i)$. The details of Step 1 of parallel LZW decompression algorithm are spelled out as follows:

```
[Step 1 of the parallel LZW decompression algorithm]
1  for i ← 0 to k − 1 do in parallel // Initialization
2      p(i) ← NULL; L(i) = 1; C_l(i) ← α(i);
3  for i ← k to k + m − 1 do in parallel // Computation of L and p*
4      p(i) ← y_{i−k}; p*(i) ← y_{i−k};
5      while(p(p*(i)) ≠ NULL)
6          L(i) ← L(i) + 1; p*(i) ← p(p*(i));
7  for i ← k to k + m − 2 do in parallel // Computation of C_l
8      C_l(i) ← α(p*(i + 1));
```

Step 2 of the parallel LZW decompression algorithm uses m threads to compute $C(y_0) \cdot C(y_1) \cdots C(y_{m-1})$, which is equal to $X = x_0 x_1 \cdots x_{n-1}$. For this

purpose, we compute the prefix-sums of $L(y_0), L(y_1), \ldots, L(y_{m-2})$ using $m-1$ processors. In other words, $s(i) = L(y_0)+L(y_1)+\cdots+L(y_i)$ is computed for every i $(0 \leq i \leq m-1)$. For simplicity, let $s(-1) = 0$. After that, for each i $(0 \leq i \leq m-1)$ $L(y_i)$ characters $C_l(p^{L(y_i)-1}(y_i)) \cdot C_l(p^{L(y_i)-2}(y_i)) \cdots C_l(p^0(y_i))(= C(y_i))$ are copied from $x_{s(i-1)}$ to $x_{s(i)-1}$. Note that, the values of $p^0(y_i), p^1(y_i), \ldots, p^{L(i)-1}(y_i)$ can be obtained by traversing pointers from code i. Hence, it makes sense to perform the copy operation from $x_{s(i)-1}$ down to $x_{s(i-1)}$.

Table 4 shows the values of . By concatenating them, we can confirm that $X = cbcbcbcda$ is obtained.

Table 4. The values of $L(y_i)$, $s(i)$, and $C(y_i)$ for $Y = 214630$

i	0	1	2	3	4	5
y_i	2	1	4	6	3	0
$L(y_i)$	1	1	2	3	1	1
$s(i)$	1	2	4	7	8	9
$C(y_i)$	c	b	cb	cbc	d	a

Let us evaluate the computing time. Let $L_{\max} = \max\{L(i) \mid 0 \leq i \leq k+m-1\}$. The for-loop in line 1 takes $O(1)$ time using k processors. Also, while-loop in line 5 is repeated at most $L(i) \leq L_{\max}$ times for each i. Hence, for-loop in line 3 can be done in $O(L_{\max})$ time using m processors. It is well known that the prefix-sums of m numbers can be computed in $O(\log m)$ time using m processors [2]. Hence, every $s(i)$ is computed in $O(\log m)$ time using $m-1$ processors. After that, every $C(y_i)$ with $L(y_i)$ characters is copied from $x_{s(i)-1}$ down to $x_{s(i-1)}$ in $O(L_{\max})$ time using m processors. Therefore, we have

Theorem 1. *The LZW decomposition of a string of m codes can be done in $O(L_{\max} + \log m)$ time using $\max(k, m)$ processors on the CREW-PRAM, where k is the number of characters in an alphabet.*

4 GPU implementation

The main purpose of this section is to describe a GPU implementation of our parallel LZW decompression algorithm. We focus on the decompression of TIFF image file compressed by LZW compression. We assume that a TIFF image file contains a gray scale image with 8-bit depth, that is, each pixel has intensity represented by an 8-bit unsigned integer. Since each of RGB or CMYK color planes can be handled as a gray scale image, it is obvious to modify gray scale TIFF image decompression for color image decompression.

As illustrated in Figure 1, a TIFF file has *an image header* containing miscellaneous information such as ImageLength (the number of rows), ImageWidth (the number of columns), compression method, depth of pixels, etc [1]. It also

has *an image directory* containing pointers to the actual image data. For LZW compression, an original 8-bit gray-scale image is partitioned into *strips*, each of which has one or several consecutive rows. The number of rows per strip is stored in the image file header with tag RowsPerStrip. Each Strip is compressed independently, and stored as the image data. The image directory has pointers to the image data for all strips.

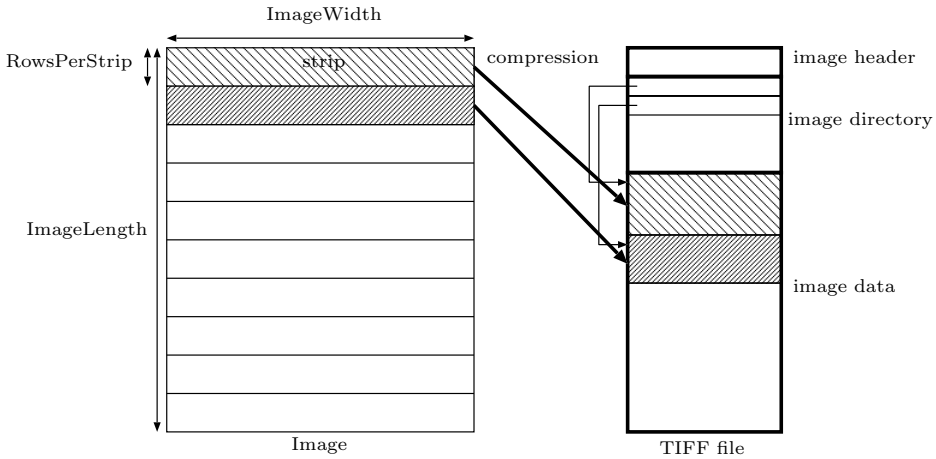


Fig. 1. An image and TIFF image file

Next, we will show how each strip is compressed. Since every pixel has an 8-bit intensity level, we can think that an input string of an integer in the range $[0, 255]$. Hence, codes from 0 to 255 are assigned to these integers. Code 256 (ClearCode) is reserved to clear the code table. Also, code 257 (EndOfInformation) is used to specify the end of the data. Thus, AddTable operations assign codes to strings from code 258. While the entry of the code table is less than 512, codes are represented as 9-bit integer. After adding code table entry 511, we switch to 10-bit codes. Similarly, after adding code table entry 1023 and 2037, 11-bit codes and 12-bit codes are used, respectively. As soon as code table entry 4094 is added, ClearCode is output. After that, the code table is re-initialized and AddTable operations use codes from 258 again. The same procedure is repeated until all pixels in a strip are converted into codes. After the code for the last pixel in a strip is output, EndOfInformation is written out. We can think that a code string for a particular strip is separated by ClearCode. We call each of them *a code segment*. Except the last one, each code segment has $4094 - 511 + 1 = 3584$ codes. The last code segment for a strip may have codes less than that.

In our implementation, a CUDA block with 1024 threads is assigned a strip. A CUDA block decompresses each code segment in the assigned strip one by

one. More specifically, a CUDA block copies a code segment of a strip stored in the global memory to a shared memory. After that, it performs Step 1 of the parallel LZW decompression. Tables for p, p^*, L, and C_l are created in the shared memory. We use 16-bit unsigned short integer for each element of the tables. Since each table has at most 4096 entries, the total size of tables is $4 \times 4096 \times 2 = 32$Kbytes. Since the capacity of the shared memory is 48Kbytes [7], this is possible. Since the table has 4095 entries, 1024 threads compute them in four iterations. In each iteration, 1024 entries of the tables are computed by 1024 threads. For example, in the first iteration, $L(i)$, $p^*(i)$, and $C_l(i)$ for every i ($0 \leq i \leq 1023$) are computed. After that, these values for every i ($0 \leq i \leq 1023$) are computed. Note that, in the second iteration, it is not necessary to execute the while-loop in line 5 until $p(p^*(i)) \neq$ NULL is satisfied. Once the value of $p^*(i)$ is less than 1024, the final resulting values of $L(i)$ and $p^*(i)$ are computed using those of $L(p^*(i))$ and $p^*(p^*(i))$. Thus, we can terminate the while-loop as soon as $p^*(i) < 1024$ is satisfied.

After the tables are obtained, the prefix-sums of s is computed in the shared memory for Step 2. Finally, the strings of characters of each code are written in the global memory. The prefix-sums can be computed by parallel algorithm for GPUs [3, 5].

5 Experimental results

We have used GeForce GTX 980 which has 16 streaming multiprocessors with 128 processor cores each to implement parallel LZW decompression algorithm. We also use Intel Xeon CPU X7460 (2.66GHz) to evaluate the running time of sequential LZW decompression.

We have used three gray scale images with 4096×3072 pixels (Figure 2), which are converted from JIS X 9204-2004 standard color image data. They are stored in TIFF format with LZW compression option. We set RowsPerStrip= 16, and so each image has $\frac{3072}{16} = 192$ strips with $16 \times 4096 = 64$k pixels each. We invoked a CUDA kernel with 192 CUDA blocks, each of which decompresses a strip with 64k pixels. Table 5 shows the compression ratio, that is, "original image size: compressed image size." We can see that "Graph" has high compression ratio because it has large areas with constant intensity levels. On the other hand, the compression ratio of "Crafts" is small because of the small details. Table 5 also shows the running time of LZW decompression using a CPU and a GPU. In the table, T_1 and T are the time for constructing tables and the total computing time, respectively. To evaluate time T_1 of sequential LZW decompression, OUTPUT in lines 4 and 6 are removed. Also, to evaluate time T_1 of parallel LZW decompression on the GPU, the CUDA kernel call is terminated without computing the prefix-sums and writing resulting characters in the global memory. Hence, we can think that $T - T_1$ corresponds to the time for for generating the original string using the tables. Clearly, sequential/parallel LZW decompression algorithms take more time to create tables for images with small compression ratio because they have many segments and need to create tables many times.

Also, the time for creating tables dominates the computing time of sequential LZW decompression, while that for writing out characters dominates in parallel LZW decompression. This is because the overhead of the parallel prefix-sums computation is not small. From the table, we can see that LZW decompression for "Flowers" using GPU is 69.4 times faster than that using CPU.

"Crafts" "Flowers" "Graph"

Fig. 2. Three gray scale image with 4096×3072 pixels used for experiments

Table 5. Experimental results (milliseconds) for three images

images	compression ratio	sequential(CPU)			parallel(GPU)			Speedup ratio
		T_1	$T-T_1$	T	T_1	$T-T_1$	T	
"Crafts"	1.67 : 1	90.4	18.0	108	0.847	1.30	2.15	50.2 : 1
"Flowers"	2.34 : 1	70.9	16.6	87.5	0.541	0.719	1.26	69.4 : 1
"Graph"	38.0 : 1	27.2	19.3	46.5	0.202	1.59	1.79	26.0 : 1

Let us discuss the performance of three practical scenarios as follows:
Scenario 1: Non-compressed images are stored in the storage. They are transferred to the GPU thorough the host computer.
Scenario 2: LZW compressed images are stored in the storage. They are transferred to the host computer, and decompressed in it. After that, the resulting non-compressed images are transferred to the GPU.
Scenario 3: LZW compressed images are stored in the storage. They are transferred to the GPU through the host computer, and decompressed in the GPU.

The throughput between the storage and the host computer depends on their bandwidth. For simplicity, we assume that their bandwidth is the same as that between the host computer and the GPU. Note that since the bandwidth of the storage is not larger than that of the GPU in many cases, this assumption does not give advantage to Scenario 3. Table 6 shows the data transfer time for non-compressed and compressed files of the three images. Clearly, the time for non-compressed files is almost the same, because they have the same size. On

the other hand, images with higher compression ratio take fewer time, because their sizes are smaller. It also evaluates the time for three scenarios. Clearly, Scenario 3, which uses parallel LZW decompression in the GPU, takes much fewer time than the others.

Table 6. Estimated running time (milliseconds) of three scenarios

images	compression ratio	Data Transfer non-compressed	Data Transfer compressed	Scenario 1	Scenario 2	Scenario 3
"Crafts"	1.67 : 1	3.79	2.44	7.58	110	4.59
"Flowers"	2.34 : 1	3.83	1.73	7.66	89.2	2.99
"Graph"	38.0 : 1	3.80	0.167	7.60	46.7	1.96

6 Conclusion

In this paper, we have presented a parallel LZW decompression algorithm and implemented in the GPU. The experimental results show that, it achieves a speedup factor up to 69.4. Also, LZW decompression in the GPU can be used to accelerate the query processing for a lot of compressed images in the storage.

References

1. Adobe Developers Association: TIFF Revision 6.0 (June 1992), http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf
2. Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press (1988)
3. Harris, M., Sengupta, S., Owens, J.D.: Chapter 39. parallel prefix sum (scan) with CUDA. In: GPU Gems 3. Addison-Wesley (2007)
4. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann (2011)
5. Nakano, K.: Simple memory machine models for GPUs. In: Proc. of International Parallel and Distributed Processing Symposium Workshops. pp. 788–797 (May 2012)
6. Nicolaisen, A.L.V.: Algorithms for Compression on GPUs. Ph.D. thesis, Tecnical University of Denmark (Aug 2015)
7. NVIDIA Corporation: NVIDIA CUDA C programming guide version 6.5 (Aug 2014)
8. Ozsoy, A., Swany, M.: Culzss: Lzss lossless data compression on cuda. In: Proc. of International Conference on Cluster Computing. pp. 403–411 (Sept 2011)
9. Sayood, K.: Introduction to Data Compression, Fourth Edition. Morgan Kaufmann (2012)
10. Shyni, K., Kumar, K.V.M.: Lossless LZW data compression algorithm on CUDA. IOSR Journal of Computer Engineering pp. 122–127 (2013)
11. Welch, T.: High speed data compression and decompression apparatus and method. US patent 4558302 (Dec 1985)
12. Welch, T.A.: A technique for high-performance data compression. IEEE Computer 17(6), 8–19 (June 1984)

Asynchronous enzymatic numerical P systems for the compare-and-exchange and sorting

T. Shiiba A. Fujiwara
Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology
680-4 Kawazu, Iizuka, Fukuoka, 820-8502, Japan

Email: k232033t@mail.kyutech.jp, fujiwara@cse.kyutech.ac.jp

Abstract

We consider asynchronous EN P systems, which are computational models inspired by the structures and behaviors of living cells, for compare-and-exchange and sorting. We first propose an asynchronous EN P system for the compare-and-exchange, and show that the asynchronous EN P system works in $O(1)$ sequential and parallel steps. We next propose an asynchronous EN P system for sorting of n numbers, and show that the asynchronous EN P system works in $O(n)$ parallel steps and $O(n^2)$ sequential steps.

1. Introduction

A number of next-generation computing paradigms have been considered due to limitation of silicon-based computation. In the next-generation computing paradigms, natural computing, which works using natural materials for computation, has considerable attention. As one of the natural computing, a numerical P system, which is inspired from structures of living cells and economics, has been introduced in [1]. In addition, an enzymatic numerical P system [2] (EN P system, for short) is also a model such that a variable called enzyme is used to promote evolution programs.

A number of EN P systems have been proposed for some operations. For example, an EN P system for the compare-and-exchange has been proposed in [3]. However, synchronous application of programs is assumed in the EN P system with maximum parallelism. The maximal parallelism means that all applicable programs are applied synchronously.

On the other hand, there is obvious asynchronous parallelism in the cell biochemistry. The asynchronous parallelism means that all programs are independently applied with different speed. Since all objects in a living cell basically works in asynchronous manner, the asynchronous parallelism must be considered to make the EN P system more realistic model.

In the present paper, we first propose asynchronous EN P system for the compare-and-exchange, and show that the asynchronous EN P system works in $O(1)$ parallel steps and $O(1)$ sequential steps.

We next propose an asynchronous EN P system for sorting n numbers using the asynchronous EN P system for compare-and-exchange as sub-systems. The asynchronous EN P system works in $O(n)$ parallel steps and $O(n^2)$ sequential steps.

2. Asynchronous EN P system

The EN P systems and the sets used in the system are defined as follows.

$$\Pi_{ENP} = (m, H, \mu, (V_1, P_1, V_1(0)), (V_2, P_2, V_2(0)), \cdots, (V_m, P_m, V_m(0)), V_O))$$

- m : m is the number of membranes.
- H: H is a set of labels for membranes. (We assume that a membrane labeled 1, which is called the skin membrane, is the outermost membrane, i.e., the skin membrane contains all of the other membranes.)
- μ: μ is membrane structure that consists of m membranes.
- V_i: V_i is a set of numerical variables in the membrane labeled i.
- P_i: P_i is a set of evolution programs in the membrane labeled i .
- $V_i(0)$: $V_i(0)$ is a set of initial values of variables in the membrane labeled i.
- V_O: V_O is a set of output variables.

In this paper, we assume that V_O is included in the outermost region of the system.

We next formally define a k-th evolution program $pr_{k,i}$ as follows.

$$pr_{k,i} = \{F_{k,i}(y_{1,i}, y_{2,i}, \cdots, y_{k,i}) | e_{j,i} \rightarrow c_{k,1}|v_1 + c_{k,2}|v_2 + \cdots + c_{k,n_i}|v_{n_i}\}$$

In the above expression, $F_{k,i}(y_{1,i}, y_{2,i}, \cdots, y_{k,i})$ is called a *production function* and computed arguments $y_{1,i}, y_{2,i}, \cdots, y_{k,i} \subset V_i$, and $c_{k,1}|v_1 + c_{k,2}|v_2 + \cdots + c_{k,n_i}|v_{n_i}$ is called a *repartition protocol*. $\{v_1, v_2, \cdots, v_{n_i}\}$ is a set of variables in the region and in neighboring regions, which are outside and inside regions. In addition,

$e_{j,i}$ is called *enzyme*, where j is an integer running from 1 to $|V_i|$. In case that $e_{j,i} > min\{y_{1,i}, y_{2,i}, \cdots, y_{k,i}\}$, the enzyme works as a catalyst, and then, the repartition protocol allocates an output of the production function to the variables according to coefficients $\{c_{k,1}, c_{k,2}, \cdots, c_{k,n_i}\} \subset N$.

In the present paper, we assume that the EN P system is asynchronous, i.e., any numbers of applicable programs are applied in each step of computation. In other words, the asynchronous EN P system can be executed sequentially, and also can be executed maximally in parallel.

3. Compare-and-Exchange

The input of compare-and-exchange is a pair of two values p and q, and the output is also a pair of two values x and y such that $x = \min\{p,q\}$, $y = \max\{p,q\}$. An idea of Π_{COMP}, which is an asynchronous EN P system for the compare-and-exchange, is as follows. We utilize a feature that an evolution program is applied if and only if a value of an enzyme in the program is greater than all numerical variables in a production function in the program. We compare two input values p and q by setting the two values to numerical variables and enzymes. The exchange operation is executed if a value of the enzyme is smaller than a value of the numerical variable.

The following is an outline of computation of the asynchronous EN P system Π_{COMP}.

Step 1: Subtract 0.1 from a value of p to avoid tie situations.

Step 2: Set p to numerical variable n_1 and enzyme e_1, and set q to numerical variables n_2 and enzyme e_2.

Step 3: Compute a larger value x using the following evolution programs. (The programs are simplified to shorten the description.) Compute a smaller value y similarly.

$$n_1|_{e_2} \rightarrow 1|x, \;\; n_2|_{e_1} \rightarrow 1|x$$

Step 4: Output x and y as a result of the compare-and-exchange.

We obtained the following theorem for the proposed EN P system Π_{COMP}.

Theorem 1: The asynchronous EN P system Π_{COMP}, which executes the compare-and-exchange for two numbers, works in $O(1)$ parallel steps and $O(1)$ sequential steps, using $O(1)$ variables, $O(1)$ membranes, and $O(1)$ evolution programs.

4. Sorting

An idea of Π_{SORT}, which is an asynchronous EN P system for sorting, is based on odd-even transposition sort [4]. We assume that input of sorting is n values $V_0, V_1, \cdots, V_{n-1}$. An outline of the EN P system is as follows.

Step 1: Repeat the following two sub-steps, (1-1) and (1-2), $\frac{n}{2}$ times and, and them, output the values.

(1-1) Perform the compare-and-exchange for (V_{2i}, V_{2i+1}) $(0 \leq i \leq \frac{n}{2} - 1)$ in $\frac{n}{2}$ membranes.

(1-2) Perform the compare-and-exchange for (V_{2i-1}, V_{2i}) $(1 \leq i \leq \frac{n}{2} - 1)$ in $\frac{n}{2}$ membranes.

We obtained the following theorem for the proposed EN P system Π_{SORT}.

Theorem 2: The asynchronous EN P systems Π_{SORT}, which executes sorting for n numbers, works in $O(n)$ parallel steps and $O(n^2)$ sequential steps, using $O(n)$ variables, $O(n)$ membranes, and $O(n)$ evolution programs.

5. Conclusions

We proposed asynchronous EN P systems for compare-and-exchange and sorting. As a future work, we are considering asynchronous EN P systems using the fewer number of membranes and programs.

Acknowledgments

This research was supported by JSPS KAKENHI, Grand-in-Aid for Scientific Research (C), 24500019.

References

[1] G. Pãun and R. Pãun. Membrane computing and economics: Numerical P systems. Fundamenta Informaticae, Vol, 73, pp. 213-227, 2006.

[2] A. Pavel, O. Arsene and C. Buiu. Enzymatic numerical P systems - a new class of membrane computing systems. 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), pp. 1331-1336, 2010.

[3] S. Maeda, A. Fujiwara, Enzymatic numerical P systems for basic operations and sorting, 7th International Conference on Soft Computing and Intelligent System, 2014.

[4] N. Haberman, Parallel neighbor-sort (or the glory of the induction principle). AD-759 248, National Technical Information Service, US Department of Commerce, Tech. Rep., 1972.

Asynchronous membrane computing for the compare-and-exchange and sorting

Yutaka Nishida Akihiro Fujiwara
Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology
Iizuka, Fukuoka, 820-8502, Japan
Email: k232060y@mail.kyutech.jp, fujiwara@cse.kyutech.ac.jp

***Abstract*—Recently, membrane computing, which is a computational model based on cell activity, has considerable attention as one of new paradigms of computations. In the membrane computing, the asynchronous parallelism must be considered to make the membrane computing more realistic.**

In the present paper, we propose asynchronous P systems that execute a compare-and-exchange operation and sorting. We first propose an asynchronous P system for the compare-and-exchange operation of two binary numbers of m bits. The P system works in $O(m)$ steps by using $O(m)$ types of objects, a constant number of membranes and evolution rules of size $O(m)$. We next propose an asynchronous P system for sorting of n binary numbers of m bits by using the above asynchronous P system as a sub-system. The P system works in $O(mn^2)$ steps by using $O(mn)$ types of objects, a constant number of membranes, and evolution rules of size $O(mn)$.

I. Introduction

A number of next-generation computing paradigms have been considered due to limitation of silicon-based computational hardware. As an example of the computing paradigms, natural computing, which works using natural materials for computation, has considerable attention. A membrane computing, which is a computational model inspired by the structures and behaviors of living cells, is a representative of the natural computing.

A basic feature of the membrane computing was introduced by in [1] as a P system. The P system consists mainly of membranes and objects. A membrane is a computing cell, in which independent computation is executed, and may contain objects and other membranes. Each object evolves according to evolution rules associated with a membrane in which the object is contained.

The P system and most variants have been proved to be universal [2], and several P systems have been proposed for solving NP problems [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] since a exponential number of membranes can be created in a polynomial number of steps on the P system. In addition, P systems for basic operations, such as logic or arithmetic operations, have been proposed in [13], [14] to apply membrane computing in a wide range problems.

However, synchronous application of evolution rules is assumed on the above P systems with the maximal parallelism, which is a main feature of the P systems. The maximal parallelism means that all applicable rules in all membranes are applied synchronously.

On the other hand, there is obvious asynchronous parallelism in the cell biochemistry. The asynchronous parallelism means that all objects may react on rules with different speed, and evolution rules are applied to objects independently. Since all objects in a living cell basically works in asynchronous manner, the asynchronous parallelism must be considered to make P system more realistic model.

For considering asynchronous parallelism, a number of P systems have been proposed in [15], [16], [17], [16], [18]. As an example, two asynchronous P systems [17] have been proposed for solving SAT and Hamiltonian cycle problem, and a number of P systems [18] have been proposed for graph problems. The P systems solve NP problems in a polynomial number of parallel steps. In addition, another asynchronous P system [16] has been proposed for computing arithmetic operations and factorization.

As complexity of the asynchronous P system, we consider two kinds of numbers, which are a number of sequential steps and a number of parallel step. The numbers of sequential steps is a number of executed steps in case that rules are applied sequentially, and the number of parallel steps is a number of executed steps with maximal parallelism.

In the present paper, we propose asynchronous P systems that executes a compare-and-exchange operation and sorting, which are basic operations for computation. We first propose an asynchronous P system for the compare-and-exchange operation of two binary numbers of m bits. The P system works in $O(m)$ sequential and parallel steps by using $O(m)$ types of objects, a constant number of membranes and evolution rules of size $O(m)$.

We next propose an asynchronous P system for sorting of n binary numbers of m bits by using the above asynchronous P system as a sub-system. The P system works in $O(mn^2)$ sequential and parallel steps by using $O(mn)$ types of objects, a constant number of membranes, and evolution rules of size $O(mn)$.

II. Preliminaries

A. Computational model for membrane computing

Several models have been proposed for membrane computing. We briefly introduce a basic model of the P system in this subsection. The P system consists mainly of membranes and objects. A membrane is a computing cell, in which

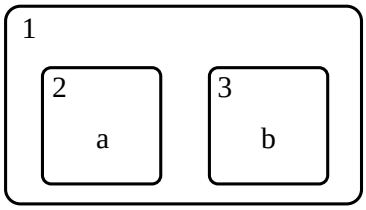


Fig. 1. An example of membrane structure

independent computation is executed, and may contain objects and other membranes. In other words, the membranes form nested structures. In the present paper, each membrane is denoted by using a pair of square brackets, and the number on the right-hand side of each right-hand bracket denotes the label of the corresponding membrane. An object in the P system is a memory cell, in which each data is stored, and can divide, dissolve, and pass through membranes. In the present paper, each object is denoted by finite strings over a given alphabet, and is contained in one of the membranes.

For example, $[[a]_2[b]_3]_1$ and Figure 1 denote the same membrane structure that consists of three membranes. The membrane labeled 1 contains two membranes labeled 2 and 3, and the two membranes contain objects a and b, respectively.

Computation of P systems is executed according to evolution rules, which are defined as rewriting rules for membranes and objects. All objects and membranes are transformed in parallel according to applicable evolution rules. If no evolution rule is applicable for objects, the system ceases computation.

Now, we formally define a P system and the sets used in the system as follows.

$$\Pi = (O, \mu, \omega_1, \omega_2, \cdots, \omega_m, R_1, R_2, \cdots, R_m)$$

O: O is the set of all objects used in the system.
ω_i: Each ω_i is a set of objects initially contained only in the membrane labeled i.
R_i: Each R_i is a set of evolution rules that are applicable to objects in the membrane labeled i.

In the present paper, we assume that input objects are given from the outside region into the outermost membrane, and computation is started by applying evolution rules. We also assume that output objects are sent from the outermost membrane to the outside region. In membrane computing, several types of rules are proposed. In the present paper, we consider five basic rules of the following forms.

(1) Object evolution rule: $[\ a\ ]_h \rightarrow [\ b\ ]_h$
where $h \in H$ and $a, b \in O$. Using the rule, an object a evolves into another object b. (We omit the brackets in each evolution rule for cases that a corresponding membrane is obvious.)

(2) Send-in communication rule: $a[\ ]_h \rightarrow [\ b\ ]_h$
where $h \in H$, and $a, b \in O$. Using the rule, an object a is sent into the membrane, and can evolve into another object b.

(3) Send-out communication rule: $[\ a\ ]_h \rightarrow [\ ]_h b$
where $h \in H$, and $a, b \in O$. Using the rule, an object a is sent out of the membrane, and can evolve into another object b.

(4) Dissolution rule: $[\ a\ ]_h \rightarrow b$
where $h \in H$, and $a, b \in O$. Using the rule, the membrane, which contains object a, is dissolved, and the object can evolve into another object b. (The outermost membrane cannot be dissolved.)

(5) Division rule: $[\ a\ ]_h \rightarrow [\ b\ ]_h[\ c\ ]_h$
where $h \in H$, and $a, b \in O$. Using the rule, the membrane, which contains object a, is divided into two membranes that contain objects b and c, respectively.

We assume that each of the above rules is applied in a constant number of biological steps. In the following sections, we consider the number of steps executed in a P system as the complexity of the P system.

B. Maximal parallelism and asynchronous parallelism

In the standard model in membrane computing, which is a P system with maximal parallelism, all of the above rules are applied in a non-deterministic maximally parallel manner. In one step of computation of the P system, each object is evolved according to one of applicable rules. (In case there are several possibilities, one of the applicable rules is non-deterministically chosen.) All object, for which no rules applicable, remain unchanged to the next step. In other words, all applicable rules are applied in parallel in each step of computation.

On the other hand, we propose asynchronous P systems, which assume that evolution rules are applied in fully asynchronous manner. In the asynchronous P systems, any number of applicable evolution rules is applied in each step of computation. In the other words, the asynchronous P system can be executed sequentially, and also can be executed in maximal parallel manner.

The reason why we assume the asynchronous parallelism in this paper is based on the fact that every living cell acts independently and asynchronously. Since the standard P system ignores the asynchronous feature of living cells, the asynchronous P system is a more realistic computation model for cell activities.

We now show an example for difference between the P system with maximal parallelism and the asynchronous P system. We define P system Π and the sets used in the system as follows.

$$\Pi = (O, \mu, \omega_1, R_1)$$

- $O = \{a, b, c, d, e\}$
- $\mu = [\ ]_1$
- $\omega_1 = \phi$
- $R_1 = \{a \rightarrow b, bc \rightarrow d, c \rightarrow e\}$

We now show an example of the computation of the P system Π. Let us assume that input objects aac are given into the membrane from the outside region.

We first consider a computation of Π on the standard P system. In the initial state, the applicable rules are $a \rightarrow b$ and

$c \to e$, and the two rules are applied in parallel with maximal parallelism. Then, input objects are evolved into bbe after the first computation step in the P system. Since the object bbe cannot be evolved using evolution rules in R_1, the computation on the P system is halted.

We next consider a computation of Π on the asynchronous P system. In the initial state of the asynchronous P system, the applicable rules are $a \to b$ and $c \to e$, and the two rules are applied asynchronously. Then, the input objects aac can be evolved into bbe, abe, bbc, abc or aae in the first step of the computations. In this case, objects bbc and abc can be evolved into bd and ad in the second step of the computation, respectively.

Therefore, a number of executions are possible in the asynchronous P system, and the evolution rules in the standard P system, which assumes a maximal parallel manner, may not work in an asynchronous parallel manner.

In the asynchronous P system, all evolution rules can be applied completely in parallel, which is the same as the conventional P system, or all evolution rules can be applied sequentially. We define the number of steps executed in the asynchronous P system in the maximal parallel manner as the number of parallel steps. We also define the number of steps in the case that the applicable evolution rules are applied sequentially as the number of sequential steps. The numbers of parallel and sequential steps indicate the best and worst case complexities for the asynchronous P system. In addition, the proposed asynchronous P system must be guaranteed to output a correct solution in any asynchronous execution.

C. Representation of binary numbers with objects

In this subsection, we describe a unified representation of a binary number with objects. The representation is similar to the binary notation of [14], and one object corresponds to one bit of a binary number. Therefore, we use $O(mn)$ objects to denote n binary numbers of m bits. In addition, the representation enables the addressing feature, i.e., each binary number is stored in a given address.

Let $V_{i,m-1}, V_{i,m-2}, \cdots, V_{i,0}$ be m Boolean values stored in address i. In case that the values denote a non-negative integer V_i, the following expression holds.

$$V_i = \sum_{j=0}^{m-1} V_{i,j} \times 2^j$$

.

We use the following m objects to denote a binary number of m bits. In the objects, A_i and B_j denote the address and the bit position, respectively, in which each value is stored.

$$\langle A_i, B_{m-1}, V_{i,m-1}\rangle, \langle A_i, B_{m-2}, V_{i,m-2}\rangle, \cdots, \langle A_i, B_0, V_{i,0}\rangle$$

The above objects are referred to as memory objects. For example, the following four memory objects denote a binary number 1000, which is stored in address 1.

$$\langle A_1, B_3, 1\rangle, \langle A_1, B_2, 0\rangle, \langle A_1, B_1, 0\rangle, \langle A_1, B_0, 0\rangle$$

III. Compare-and-exchange

In this section, we present an asynchronous P system for the compare-and-exchange operation of two binary numbers of m bits. The input of the compare-and-exchange operation is a pair of two values p, q, and the output of the operation is also a pair of two values x, y such that $x = \min\{p, q\}$ and $y = \max\{p, q\}$. We first explain an input and an output of the compare-and-exchange operation for the P system, and then, show an outline and details of the P system with an example. Finally, we discuss time complexity of the proposed P system.
‘

A. Input and output

An input and an output of the compare-and-exchange operation are expressed using memory objects described in Section 2.

We assume that two input binary numbers of m bits is stored in addresses p and q. The following two sets of objects are given as an input in the outermost membrane.

$$\langle A_p, B_{m-1}, V_{p,m-1}\rangle, \langle A_p, B_{m-2}, V_{p,m-2}\rangle, \cdots, \langle A_p, B_0, V_{p,0}\rangle$$
$$\langle A_q, B_{m-1}, V_{q,m-1}\rangle, \langle A_q, B_{m-2}, V_{q,m-2}\rangle, \cdots, \langle A_q, B_0, V_{q,0}\rangle$$

An output of the compare-and-exchange operation, which is a pair of two binary numbers stored in addresses x and y, is also given as sets of memory objects as follows.

$$\langle A_x, B_{m-1}, V_{x,m-1}\rangle, \langle A_x, B_{m-2}, V_{x,m-2}\rangle, \cdots, \langle A_x, B_0, V_{x,0}\rangle$$
$$\langle A_y, B_{m-1}, V_{y,m-1}\rangle, \langle A_y, B_{m-2}, V_{y,m-2}\rangle, \cdots, \langle A_y, B_0, V_{y,0}\rangle$$

B. An asynchronous P system for the compare-and-exchange

We first explain an overview of the asynchronous P system for the compare-and-exchange operation. The membrane structure used in the computation is the outermost membrane only such that $[\ ]_1$.

The computation of the P system consists of the following 3 steps.

- Step 1: Find the most significant bit, which is the left-most bit position such that Boolean values of two binary numbers p and q differs. Then, compute the relation between p and q from the most significant bit.
- Step 2: In case that $p \geq q$, copy all Boolean values of p and q to memory objects that denotes y and x, respectively. In the other case, copy all Boolean values of p and q to memory objects that denotes x and y, respectively.
- Step 3: Send out memory objects that denote x and y from the outermost membrane.

We now explain outline of each step of the computation. In Step 1, the most significant bit is searched from the higher bit to the lower bit applying the following two sets of evolution rules.

$$\begin{aligned} R_{1,1,1} \quad = \quad & \{\langle A_p, B_i, V\rangle\langle A_q, B_i, V\rangle\langle CB, i\rangle \\ & \to \langle A_p, B_i, V\rangle\langle A_q, B_i, V\rangle\langle CB, i-1\rangle \\ & \mid V \in \{0,1\}, 1 \leq i \leq m-1\} \end{aligned}$$

$$
\begin{aligned}
R_{1,1,2} \;=\; & \{\langle A_p, B_i, 1\rangle\langle A_q, B_i, 0\rangle\langle CB, i\rangle \\
& \quad \rightarrow \langle A_p, B_i, 1\rangle\langle A_q, B_i, 0\rangle\langle GTE\rangle \\
& \quad \mid 0 \le i \le m-1\} \\
& \cup\{\langle A_p, B_i, 0\rangle\langle A_q, B_i, 1\rangle\langle CB, i\rangle \\
& \quad \rightarrow \langle A_p, B_i, 0\rangle\langle A_q, B_i, 1\rangle\langle LT\rangle \\
& \quad \mid 0 \le i \le m-1\} \\
& \cup\{\langle A_p, B_0, V\rangle\langle A_q, B_0, V\rangle\langle CB, 0\rangle \\
& \quad \rightarrow \langle A_p, B_0, V\rangle\langle A_q, B_0, V\rangle\langle GTE\rangle \\
& \quad \mid V \in \{0,1\}\}
\end{aligned}
$$

In the above evolution rules, object $\langle CB, i\rangle$ denotes current bit position for comparison of two Boolean values. In case that two Boolean values are equal, except for the lowest bits, evolution rules in $R_{1,1,1}$ is applied, and the comparison is moved to the lower bit position. In the other case, one of objects $\langle GTE\rangle$ and $\langle LT\rangle$, which denote "greater than or equal to" and "less than" respectively, is created according to evolution rules in $R_{1,1,2}$.

In Step 2, Boolean values that denote p and q are copied to memory objects that denote x and y according to the results of the comparison in Step 1. Step 2 is executed applying the following set of evolution rule.

$$
\begin{aligned}
R_{1,2} \;=\; & \{\langle GTE\rangle \rightarrow \langle EX, m-1\rangle, \langle LT\rangle \rightarrow \langle CP, m-1\rangle\} \\
& \cup\{\langle A_p, B_i, V_p\rangle\langle A_q, B_i, V_q\rangle\langle EX, i\rangle \\
& \quad \rightarrow \langle A_x, B_i, V_q\rangle\langle A_y, B_i, V_p\rangle\langle EX, i-1\rangle, \\
& \quad \langle A_p, B_i, V_p\rangle\langle A_q, B_i, V_q\rangle\langle CP, i\rangle \\
& \quad \rightarrow \langle A_x, B_i, V_p\rangle\langle A_y, B_i, V_q\rangle\langle CP, i-1\rangle \\
& \quad \mid V_p, V_q \in \{0,1\}, 1 \le i \le m-1\} \\
& \cup\{\langle EX, -1\rangle \rightarrow \langle CB, m-1\rangle, \\
& \quad \langle CP, -1\rangle \rightarrow \langle CB, m-1\rangle\}
\end{aligned}
$$

In the above $R_{1,2}$, object $\langle EX, i\rangle$ executes exchange of two input values p and q, i.e. the object copies values p and q to y and x from the higher bit to the lower bit. On the other hand, object $\langle CP, i\rangle$ similarly copies two input values p and q to x and y. At the end of Step 2, object $\langle CB, m-1\rangle$ is created for initializing the object for the next compare-and-exchange operation.

In Step 3, memory objects, which denote x and y, are sent out from the outermost membrane applying the following set of send-out communication rules.

$$
\begin{aligned}
R_{1,3} \;=\; & \{[\langle A_x, B_i, V_x\rangle\langle A_y, B_i, V_y\rangle]_1 \\
& \quad \rightarrow [\,]_1\ \langle A_x, B_i, V_x\rangle\langle A_y, B_i, V_y\rangle \\
& \quad \mid V_x, V_y \in \{0,1\}, 0 \le i \le m-1\}
\end{aligned}
$$

Note that Step 3 may be executed before finishing Step 2 because we assume the asynchronous P system. In any execution of the P system, the P system output correct results at the end of computation because sets of evolution rules is designed to be applied sequentially.

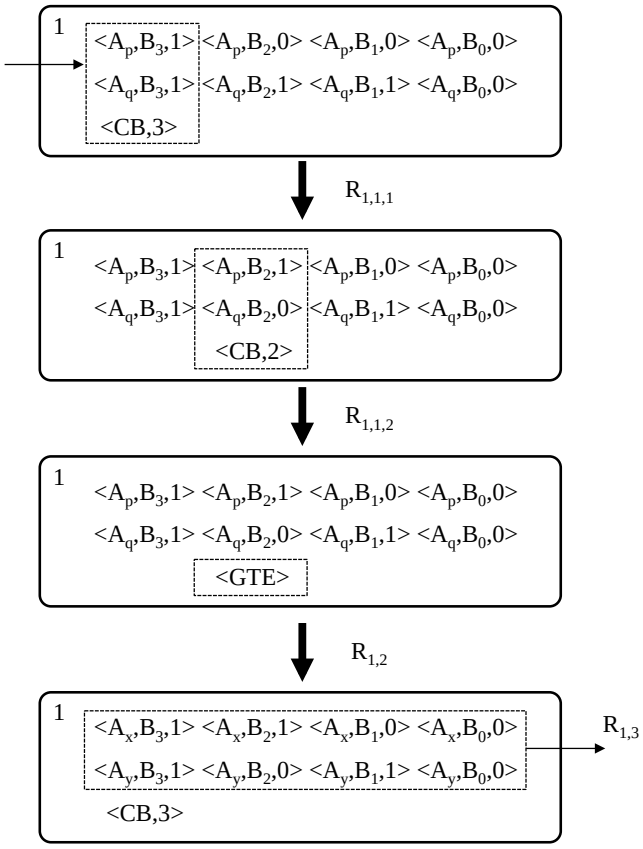


Fig. 2. An example of execution of Π_{ce}

We now formally define P system π_{ce} that executes the compare-and-exchange operation for two binary numbers of m bits.

$$\Pi_{ce} = (O, \mu, \omega_1, R_1)$$

$$
\begin{aligned}
O \;=\; & \{\langle A_p, B_i, V_p\rangle, \langle A_q, B_i, V_q\rangle, \\
& \quad \langle A_x, B_i, V_x\rangle, \langle A_y, B_i, V_y\rangle \\
& \quad \mid V_p, V_q, V_x, V_y \in \{0,1\}, 0 \le i \le m-1\} \\
& \cup\{\langle CB, i\rangle \mid 0 \le i \le m-1\} \\
& \cup\{\langle EX, i\rangle, \langle CP, i\rangle \mid -1 \le i \le m-1\} \\
& \cup\{\langle GTE\rangle, \langle LT\rangle\} \\
\mu \;=\; & [\,]_1 \\
\omega_1 \;=\; & \{\langle CB, m-1\rangle\} \\
R_1 \;=\; & R_{1,1,1} \cup R_{1,1,2} \cup R_{1,2} \cup R_{1,3}
\end{aligned}
$$

Figure 2 illustrates an example of an execution of the P system Π_{ce}. In the example, two input binary numbers are $p = 1100$ and $q = 1010$, and the memory objects that denote p and q are given from the outside region into the P system. Then, the comparison for the first bit is executed applying $R_{1,1,1}$, and the comparison is moved the the next bit.

In the next step, a set of rule $R_{1,1,2}$ is applied, and object $\langle GTE\rangle$ is created in the membrane, and the object enables

$R_{1,2}$. After application of $R_{1,2}$, exchanged values are sent out from the outermost membrane applying $R_{1,3}$.

C. Complexity of the P system

We now consider complexity of the proposed P system Π_{ce}. Both of the number of sequential and parallel steps in Step 1 and Step 2 are $O(m)$ because the steps are executed sequentially. The number of sequential and parallel steps in Step 3 is $O(m)$ and $O(1)$, respectively, because the step can executed in parallel.

Since the number of types of objects in the P system is $O(m)$ and the number of kinds of evolution rules is also $O(m)$, we obtain the following theorem for Π_{ce}.

Theorem 1: An asynchronous P system Π_{ce}, which executes compare-and-exchange operation of two binary numbers of m bits, works in in $O(m)$ sequential and parallel steps using $O(m)$ types of objects and evolution rules of size $O(m)$. □

IV. Sorting

We next show an asynchronous P system for sorting n binary numbers of m bits. An idea of the P system is based on the odd-even transposition sort [19], and the P system Π_{ce}, which is described in previous section, is used as a sub-system. We first explain an input and an output of sorting for our P system, and then, show an outline and details of our P system. Finally, we discuss time complexity of the proposed P system.

A. Input and output

We assume that input of sorting is n binary numbers $V_0, V_1, \cdots, V_{n-1}$, and also assume that the numbers are stored in addresses $X_0, X_1, \cdots X_{n-1}$. The binary numbers are given as a set of memory objects given below.

$$\begin{aligned}
&\langle X_0, B_{m-1}, V_{0,m-1}\rangle\langle X_0, B_{m-2}, V_{0,m-2}\rangle \cdots \langle X_0, B_0, V_{0,0}\rangle\\
&\langle X_1, B_{m-1}, V_{1,m-1}\rangle\langle X_1, B_{m-2}, V_{1,m-2}\rangle \cdots \langle X_1, B_0, V_{1,0}\rangle\\
&\qquad\qquad\vdots\\
&\langle X_{n-1}, B_{m-1}, X_{n-1,m-1}\rangle\langle X_{n-1}, B_{m-2}, X_{n-1,m-2}\rangle \cdots\\
&\qquad\qquad\cdots\langle A_{n-1}, B_0, V_{n-1,0}\rangle
\end{aligned}$$

We also assume that output of sorting is a set of binary numbers stored in addresses $Y_0, Y_1, \cdots Y_{n-1}$. The memory objects for output are given below.

$$\begin{aligned}
&\langle Y_0, B_{m-1}, V_{0,m-1}\rangle\langle Y_0, B_{m-2}, V_{0,m-2}\rangle \cdots \langle Y_0, B_0, V_{0,0}\rangle\\
&\langle Y_1, B_{m-1}, V_{1,m-1}\rangle\langle Y_1, B_{m-2}, V_{1,m-2}\rangle \cdots \langle Y_1, B_0, V_{1,0}\rangle\\
&\qquad\qquad\vdots\\
&\langle Y_{n-1}, B_{m-1}, V_{n-1,m-1}\rangle\langle Y_{n-1}, B_{m-2}, V_{n-1,m-2}\rangle \cdots\\
&\qquad\qquad\cdots\langle Y_{n-1}, B_0, V_{n-1,0}\rangle
\end{aligned}$$

B. An asynchronous P system for sorting

We first explain an overview of the asynchronous P system for sorting. The membrane structure used in the computation consists of three membranes such that $[\ [\ ]_{ce\_odd}\ [\ ]_{ce\_even}]_1$, where membranes *ce_odd* and *ce_even* is a P system for the compare-and-exchange operation for pairs of binary numbers. The computation of the P system consists of the following steps.

Step 1: Repeat the following two sub-steps, (1-1) and (1-2), $\frac{n}{2}$ times, and send out obtained memory objects from the outermost membrane.

(1-1) Send memory objects $V_0, V_1, \cdots V_{n-1}$ into membrane *ce_odd*. Then, execute the compare-and-exchange operation for $\frac{n}{2}$ pairs given below. (We call the the compare-and-exchange step *odd exchange step*.)

$$(V_{2i}, V_{2i+1}) \quad (0 \le i \le \frac{n}{2} - 1)$$

The above memory strands are sent out from the membrane after the odd exchange step.

(1-2) Send memory objects that denote $V_0, V_1, \cdots V_{n-1}$ into membrane *ce_even*. Then, execute the compare-and-exchange operation for $\frac{n}{2}$ pairs given below. (We call the compare-and-exchange step *even exchange step*.)

$$(V_{2i-1}, V_{2i}) \quad (1 \le i \le \frac{n}{2} - 1)$$

The above memory strands are also sent out from the membrane after the even exchange step.

We now explain an outline of each step of the computation. First of all, we consider two P systems, Π_{ce_odd} and Π_{ce_even}, which execute the odd and even exchange steps. Since each pair of compare-and-exchange operation is executed independently from the other pairs, Π_{ce_odd} and Π_{ce_even} are obtained by modifying Π_{ce}. We assume that these two P systems are two membranes *ce_odd* and *ce_even*.

We next explain the other steps for the P system. Since we assume an asynchronous P system, we must consider how to execute the above step synchronously, i.e., (1-1) and (1-2) must not be executed simultaneously.

In (1-1), input objects are moved into membrane *ce_odd*. This step is executed applying the following evolution rules.

$$\begin{aligned}
R_{1,1,1} \;=\; &\{\langle X_i, B_j, V\rangle\langle M_O, i, j\rangle[\ ]_{ce_odd}\\
&\quad\to [\langle X_i, B_j, V\rangle\langle M_O, i, j\rangle]_{ce_odd}\\
&\quad \mid V \in \{0,1\}, 0 \le i \le n-1, 0 \le j \le m-1\}\\
&\cup\{[\langle M_O, i, j\rangle]_{ce_odd} \to [\ ]_{ce_odd}\langle M_O, i, j+1\rangle\\
&\quad \mid 0 \le i \le n-1, 0 \le j \le m-2\}\\
&\cup\{[\langle M_O, i, m-1\rangle]_{ce_odd}\\
&\quad\to [\ ]_{ce_odd}\langle M_O, i+1, 0\rangle\\
&\quad \mid 0 \le i \le n-1\}\\
&\cup\{\langle M_O, n, 0\rangle\langle C, k\rangle \to \langle M_E, 0, 0\rangle\langle C, k+1\rangle\\
&\quad \mid 0 \le k \le n-1\}
\end{aligned}$$

In the above $R_{1,1,1}$, object $\langle A_i, B_j, V_{i,j}\rangle$ is moved into membrane *ce_odd* using object $\langle M_O, i, j\rangle$. The object $\langle M_O, i, j\rangle$ first moves memory objects stored in address 0, next moves memory objects stored in address 1, and so on. After all memory objects are moved into membrane *ce_odd*, the object is set to $\langle M_E, 0, 0\rangle$, which is an object used to move memory object into membrane *ce_even*. In addition, object

$\langle C, k\rangle$ is used to count the number of steps, and the object is incremented at the end of (1-1).

After moving memory objects into membrane *ce_odd*, the compare-and-exchange operation is executed in the membrane, and all memory objects are asynchronously sent out from membrane *ce_odd*.

In (1-2), input objects are moved into membrane *ce_odd* using the following $R_{1,1,2}$, which is similar to $R_{1,1,1}$.

$$\begin{aligned} R_{1,1,2} &= \{\langle X_i, B_j, V\rangle\langle M_E, i, j\rangle[\ ]_{ce\_even} \\ &\quad \to [\langle X_i, B_j, V\rangle\langle M_E, i, j\rangle]_{ce\_even} \\ &\quad \mid V \in \{0,1\}, 0 \le i \le n-1, 0 \le j \le m-1\} \\ &\cup\{[\langle M_E, i, j\rangle]_{ce\_even} \\ &\quad \to [\ ]_{ce\_even}\langle M_E, i, j+1\rangle \\ &\quad \mid 0 \le i \le n-1, 0 \le j \le m-2\} \\ &\cup\{[\langle M_E, i, m-1\rangle]_{ce\_even} \\ &\quad \to [\ ]_{ce\_even}\langle M, i+1, 0\rangle \\ &\quad \mid 0 \le i \le n-1\} \\ &\cup\{\langle M_E, n, 0\rangle\langle C, k\rangle \to \langle M_O, 0, 0\rangle\langle C, k+1\rangle \\ &\quad \mid 0 \le k \le n-1\} \end{aligned}$$

After $\frac{n}{2}$ times executions of (1-1) and (1-2), sorting is completed, and object $\langle M_O, 0, 0\rangle$ and $\langle C, n\rangle$is obtained. Then, all memory objects are sent out applying the following $R_{1,2}$.

$$\begin{aligned} R_{1,2} &= \{\langle M_O, 0, 0\rangle\langle C, n\rangle \to \langle S, 0, 0\rangle\} \\ &\cup\{\langle X_i, B_j, V\rangle\langle S, i, j\rangle \\ &\quad \to \langle Y_i, B_j, V\rangle\langle S, i, j+1\rangle \\ &\quad \mid V \in \{0,1\}, 0 \le i \le n-1, 0 \le j \le m-2\} \\ &\cup\{\langle X_i, B_{m-1}, V\rangle\langle S, i, m-1\rangle \\ &\quad \to \langle Y_i, B_{m-1}, V\rangle\langle S, i+1, 0\rangle \\ &\quad \mid V \in \{0,1\}, 0 \le i \le n-1\} \\ &\cup\{[\langle Y_i, B_j, V\rangle]_1 \to [\ ]_1\langle Y_i, B_j, V\rangle \\ &\quad \mid V \in \{0,1\}, 0 \le i \le n-1, 0 \le j \le m-1\} \\ &\cup\{\langle S, n, 0\rangle \to \langle M_O, 0, 0, 1\rangle\} \end{aligned}$$

We now formally define P system Π_{sort} that executes sorting for n binary numbers of m bits.

$$\Pi_{sort} = (O, \mu, \omega_1, \omega_{ce\_odd}, \omega_{ce\_even}, R_1, R_{ce\_odd}, R_{ce\_even})$$

$$\begin{aligned} O &= \{\langle X_i, B_j, V\rangle, \langle Y_i, B_j, V\rangle \mid V \in \{0,1\}, \\ &\qquad 0 \le i \le n-1, 0 \le j \le m-1\} \\ &\quad \cup\{\langle M_O, i, j\rangle, \langle M_E, i, j\rangle \\ &\qquad \mid 0 \le i \le n, 0 \le j \le m-1\} \\ &\quad \cup\{\langle C, k\rangle \mid 0 \le k \le n\} \\ &\quad \cup\{\langle S, i, j\rangle \mid 0 \le i \le n, 0 \le j \le m-1\} \\ \mu &= [\ [\ ]_{ce\_even}[\ ]_{ce\_odd}]_1 \\ \omega_1 &= \{\langle M_O, 0, 0, 1\rangle\} \\ R_1 &= R_{1,1,1} \cup R_{1,1,2} \cup R_{1,2} \end{aligned}$$

(ω_{ce_odd}, ω_{ce_even}, Π_{ce}. R_{ce_odd} and R_{ce_even} are omitted.)

C. Complexity of the P system

We now consider complexity of the proposed P system Π_{sort}. The numbers of parallel and sequential steps in (1-1) and (1-2) are $O(m)$ parallel steps and $O(mn)$ sequential steps because compare-and-exchange operations are executed for $\frac{n}{2}$ pairs of binary numbers. The other steps, which sequentially move memory objects, works in $O(nm)$. Since (1-1) and (1-2) is repeated $\frac{n}{2}$ times, time complexity of the P system is $O(mn^2)$ sequential and parallel steps.

The number of types of objects in the P system is $O(mn)$, and the number of kinds of evolution rules is also $O(mn)$. Therefore, we obtain the following theorem for Π_{sort}.

Theorem 2: An asynchronous P system Π_{sort}, which sorts n binary numbers of m bits, works in $O(mn^2)$ sequential and parallel steps using $O(mn)$ types of objects and evolution rules of size $O(mn)$. □

V. Conclusions

We proposed asynchronous P systems for the compare-and-exchange operation and sorting. The proposed P systems are fully asynchronous, i.e. any number of applicable rules may be applied in one step of the P systems. The first P system for the compare-and-exchange operation works in $O(m)$ sequential and parallel steps for two binary numbers of m bits, and the second P system for sorting works in $O(mn^2)$ sequential and parallel steps for n binary numbers of m bits. Although the number of steps is not small as well-known sorting algorithms, the proposed P system shows that the basic operations can be executed on the asynchronous P system.

As our future work, we are considering reduction of the numbers of parallel steps on the proposed asynchronous P systems.

Acknowledgments

This research was partially supported by JSPS KAKENHI, Grand-in-Aid for Scientific Research (C), 24500019.

References

[1] G. Păun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000.
[2] ——, *Introduction to Membrane Computing*. Springer, 2006.
[3] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and F. J. Romero-Campero, "A uniform solution to SAT using membrane creation," *Theoretical Computer Science*, vol. 371, no. 1-2, pp. 54–61, 2007.
[4] V. Manca, "DNA and membrane algorithms for SAT," *Fundamenta Informaticae*, vol. 49, no. 1-3, pp. 205–221, 2002.
[5] L. Q. Pan and A. Alhazov, "Solving HPP and SAT by P systems with active membranes and separation rules," *Acta Informatica*, vol. 43, no. 2, pp. 131–145, 2006.
[6] G. Păun, "P systems with active membranes: Attacking NP-complete problems," *Journal of Automata, Languages and Combinatorics*, vol. 6, no. 1, pp. 75–90, 2001.
[7] M. J. Pérez-Jiménez, A. Romero-Jiménez, and F. Sancho-Caparrini, "A polynomial complexity class in P systems using membrane division," *Journal of Automata, Languages and Combinatorics*, vol. 11, no. 4, pp. 423–434, 2003.
[8] C. Zandron, C. Ferretti, and G. Mauri, "Solving NP-complete problems using P systems with active membranes," in *Unconventional Models of Computation*, 2000, pp. 289–301.
[9] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez, "A fast P system for finding a balanced 2-partition," *Soft Computing*, vol. 9, no. 9, pp. 673–678, 2005.

[10] M. J. Pérez-Jiménez and A. Riscos-Núñez, “A linear-time solution to the knapsack problem using P systems with active membranes,” *Membrane Computing*, vol. 2933, pp. 250–268, 2004.
[11] ——, “Solving the subset-sum problem by P systems with active membranes,” *New Generation Computing*, vol. 23, no. 4, pp. 339–356, 2005.
[12] M. J. Pérez-Jiménez and F. Romero-Campero, “Solving the BIN PACKING problem by recognizer P systems with active membranes,” in *The Second Brainstorming Week on Membrane Computing*, 2004, pp. 414–430.
[13] A. Leporati, C. Zandron, and G. Mauri, “Solving the factorization problem with P systems,” *Progress in Natural Science*, vol. 17, no. 4, pp. 471–478, 2007.
[14] A. Fujiwara and T. Tateishi, “Logic and arithmetic operations with a constant number of steps in membrane computing,” *International Journal of Foundations of Computer Science*, vol. 22, no. 3, pp. 547–564, 2011.
[15] R. Freund, “Asynchronous P systems and P systems working in the sequential mode,” in *International workshop on Membrane Computing*, 2005, pp. 36–62.
[16] T. Murakawa and A. Fujiwara, “Arithmetic operations and factorization using asynchronous P systems,” *International Journal of Networking and Computing*, vol. 2, no. 2, pp. 217–233, 2012.
[17] H. Tagawa and A. Fujiwara, “Solving SAT and Hamiltonian cycle problem using asynchronous p systems,” *IEICE Transactions on Information and Systems (Special section on Foundations of Computer Science)*, vol. E95-D, no. 3, 2012.
[18] K. Tanaka and A. Fujiwara, “Asynchronous P systems for hard graph problems,” *International Journal of Networking and Computing*, vol. 4, no. 1, pp. 2–22, 2014.
[19] N. Haberman, “Parallel neighbor-sort (or the glory of the induction principle),” AD-759 248, National Technical Information Service, US Department of Commerce, Tech. Rep., 1972.

セッション7

ロボット１

Agreement among mobile robots in the three-dimensional Euclidean space: The plane formation problem and the pattern formation problem

Yukiko Yamauchi *

Kyushu University, Japan.

Abstract. We consider a swarm of autonomous mobile robots each of which is anonymous and oblivious (memory-less), and synchronously executes the same algorithm. The *plane formation problem* requires the robots to land on a common plane without forming any multiplicity from a given initial configuration and the *pattern formation problem* requires the robots to from a given target pattern from an initial configuration. We first investigate the pattern formation problem for oblivious fully-synchronous (FSYNC) robots moving in the three dimensional Euclidean space (3D-space), and characterize the problem by showing a necessary and sufficient condition for the robots to form a target pattern F from an initial configuration P. The pattern formation problem in the two dimensional Euclidean space (2D-space) has been characterized by Yamashita and Suzuki (TCS 2010) and Fujinaga et al. (SICOMP 2015). They showed a necessary and sufficient condition based on the notion of *symmetricity* of an initial configuration that shows the symmetry that the robots can never break. The symmetricity of $\rho(P)$ of positions of robots P is intuitively the order of the cyclic group of the initial configuration. It has been shown that the oblivious FSYNC robots can form a target pattern F from an initial configuration P if and only if $\rho(P)$ divides $\rho(F)$. We extend the notion of symmetricity to 3D-space by using the *rotation groups* that is defined by a set of rotation axes and their arrangement. We define the symmetricity $\varrho(P)$ of positions of robots in 3D-space as the set of rotation groups formed by rotation axes that the robots can never eliminate. We show the following necessary and sufficient condition for the pattern formation problem which is a natural extension of existing results of the pattern formation problem in 2D-space: The oblivious FSYNC robots in 3D-space can form a target pattern F from an initial configuration P if and only if $\varrho(P) \subseteq \varrho(F)$. We will show the impossibility by showing the worst case arrangement of local coordinate systems of robots, that does not allow the robots to break their symmetricity. For the possibility cases, we present a pattern formation algorithm for oblivious FSYNC robots. As a corollary of the main result, we can show a necessary and sufficient condition for the robots to form a plane: The oblivious FSYNC robots in 3D-space can from a plane if and only if $\varrho(P)$ is a cyclic group or a dihedral group. We then show a plane formation algorithm for oblivious FSYNC robots.
Keywords. FSYNC model, mobile robots in three dimensional Euclidean space, plane formation problem, pattern formation problem, rotation group, symmetry breaking.

* Corresponding author. Address: 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan. Fax: +81-92-802-3637. Email: yamauchi@inf.kyushu-u.ac.jp

複数のエージェントによるオンライン木探索アルゴリズム

八神貴裕 * 山内由紀子 † 来嶋秀治 † 山下雅史 †
* 九州大学大学院システム情報科学府
† 九州大学大学院システム情報科学研究院

1 序論

1.1 はじめに

本稿では，グラフ内を逃げ回る侵入者を探索者が移動しながら発見するグラフ探索問題について考える．この問題は Parsons[3] が，洞窟ではぐれた友人を見つけ出すためには何人の探索者が必要で，それぞれがどのように行動すべきかという問題を洞窟をグラフとみなして考察したことから生じた問題である．この問題では，侵入者は常にグラフ内を動き回っている．探索者は侵入者がどこにいるのか知ることができず，侵入者の移動速度も分からない．これらのことは，一度探索した場所に再び侵入者が訪れる可能性があることを意味する．

グラフ探索問題は，探索者を石と見なしてグラフ上の石置きゲームとしてモデル化できる [2, 5]．このモデルではグラフの形を知っているプレイヤーがグラフの外から石を動かして探索する．このようにプレイヤーが探索開始時に既にグラフの形を知っている探索を**オフライン探索**と呼ぶ．移動する探索対象を発見する問題は，建物の警備ロボットや，火事，地震などの災害が起こった際に，建物に逃げ遅れた人がいないか探すロボットの開発に役立つと考えられる．しかし，このようなロボットは探索を行う建物の構造を探索開始前から必ず知っているとは限らない．そこで本稿では，グラフの形を知らない複数の**非同期エージェント**が実際にグラフ内に入りグラフを探索するモデルを考える．このように探索開始時にグラフの形が与えられていない探索を**オンライン探索**と呼ぶ．また，オンライングラフ探索のモデルとしては，一人のプレイヤーが探索者として実際にグラフの中に入り，侵入者の移動を制限するバリケードを用いて探索するバリケードモデルも考えられる [6]．石置きゲームのモデルは，グラフの外にいるプレイヤーの指示で石と見なされている探索者が自らグラフ内を移動していると考えられる．バリケードモデルでは，グラフの中にいる探索者が自身では移動できないバリケードを運ぶ点で異なる．バリケードモデルについては，任意の木について最小のバリケード数で探索可能なオンラインアルゴリズムが存在する [6]．

本稿では，次のようなモデルを扱う．各エージェントはメモリを持ち，それぞれ異なる頂点から探索を開始する．本稿ではエージェントが個別の ID を持つ場合と，ID を持たない場合の二つを考える．これらのモデルを用いて，連結かつ閉路を持たないグラフである木を最小のエージェントで探索するオンラインアルゴリズムを提案する．アルゴリズムの流れは，まず全てのエージェントが同じ頂点に集合し，その後リーダーとなるエージェントを 1 人決める．そしてそのリーダーをバリケードモデルの探索者，それ以外のエージェントをバリケードと見なして探索を行う．本稿では特に，エージェントが 1 つの頂点に集合し，リーダー（探索者）を決定する部分について述べる．

1.2 関連研究

石置きゲームのモデルの一つに辺探索モデルが存在する [2]．辺探索モデルにおける石（探索者）を

ガードと呼ぶことにする．G を辺探索モデルで探索するときの最小のガード数を $es(G)$ と表す．問題の入力はグラフ G である．問題の目的は $es(G)$ と，探索手順を与えることである．許されるガードの操作は，ガードを頂点に置く，ガードを頂点から取り除く，ガードをある頂点から隣接する頂点へ辺を通過して移動するの 3 つである．このモデルではガードが自走している．

次に，バリケードモデルのオンライン探索について紹介する．このモデルは 1 人のエージェントがバリケードを使って探索をしていると考えることもできる．グラフ G をバリケードモデルで探索するときの最小のバリケード数を $r(G)$ と表す．探索者とバリケードはメモリを持つ．探索者は探索開始時グラフの形を知らないと仮定する．問題の入力はグラフ G とバリケードの個数 r，探索者の初期位置 $v \in V$ である．探索者は r 個のバリケードを持って探索を始める．探索者はグラフ G が r 個のバリケードで探索可能か否か判断し，可能であれば探索を完了することが目的である．このモデルではバリケードは自ら移動することはできず，探索者が持ち運ぶことで移動する．

探索者は，隣接する頂点に辺を通過して移動する，頂点にいるときバリケードの設置や回収をする，今いる頂点のバリケードのメモリの読み出しと書き込み，の 4 つの動作を組み合わせて探索を行う．バリケードモデルに関して，次の結果を既に得ている．

定理 1 ([6])**.** 任意の木 T について，$es(T) = r(T)+1$ が成立する．

文献 [6] で提案されているオンライン木探索アルゴリズム TSB は，探索に必要なバリケード数に関して最適である．定理 2 は任意の木 T について，オンラインでもオフラインでも探索に必要となる最小のバリケード数は同じであることを意味する．

定理 2 ([6])**.** r 個のバリケードを用いる TSB で木 T の探索に失敗するならば，r 個のバリケードを用いて T をオフライン探索するアルゴリズムは存在しない．

1.3 主結果

本稿では次の結果を示す．任意の木 T のオンライン探索について，エージェントが ID を持つ場合は，エージェント数は $es(T)$ と一致する．また，エージェントが ID を持たない場合，提案アルゴリズムではエージェント数は $es(T)+1$ 必要になる場合がある．

2 準備

ここでは，モデルと問題の定義と，既存の研究の紹介を行う．

2.1 グラフに関して

本稿では単純かつ連結な無向グラフを扱う．特に平面に埋め込まれた連結かつ閉路を持たない木 $T = (V, E)$ について議論する．V は木 T の頂点集合で，E は木 T の辺集合を表す．

以下，本稿で扱うグラフの詳細な定義を行う．頂点 v の次数は $\deg(v)$ で表す．グラフの各頂点，各辺は固有のラベルを持たない．しかし，グラフ内の各頂点 $v \in V$ に接続する辺に対してラベル付け関数の集合 $\Lambda = \{\lambda_v | v \in V\}$ によって，反時計回りに 0 から $\deg(v) - 1$ までのポート番号が付けられている．以降，ラベル付け関数の集合 Λ をラベルと呼ぶ．ラベル付け関数 $\lambda_v(e)$ は頂点 v から見た接続する辺 e のポート番号を表す．頂点 $v \in V$ から任意の頂点までの距離の最大値を離心数と呼ぶ．V の頂点で離心数が最小の頂点を中心と呼ぶ．また，エージェントは頂点や辺にメッセージを残すことはできない（頂点や辺にはメモリがない）．この問題では，辺は道路のように見なして，エージェントが辺の上にいる場合は，辺のどの位置にいるかを考えることにする．

2.2 グラフ探索問題

オフラインの**グラフ探索問題**の入力はグラフ $G = (V, E)$ であり探索に必要なエージェント数と各エー

ジェントの行動について考える．オンライン探索では，問題の入力はグラフ G とエージェント数 k とエージェントの初期位置 $S \subseteq V$ ($|S| = k$) である．k 人のエージェントは異なる頂点からそれぞれ同じアルゴリズムを開始し，各エージェントは G を探索可能であるか否か判断し，可能であれば探索を完了することを目的とする．

時刻 $t \in [0, \infty)$ での侵入者の位置を連続関数 $i(t)$ で表す．$i(t)$ はグラフ G の頂点あるいは辺上のある点を表す．グラフ G をエージェント k 人で探索可能であるとは，任意の関数 $i(t)$ に関して，ある時刻 τ に少なくとも 1 人のエージェントが $i(\tau)$ 上にいるような非同期エージェントの探索アルゴリズム $\mathcal{A}$ が存在することである．言い換えると，侵入者が G 内をどのように移動したとしても，ある時刻 τ で必ずエージェントが侵入者を捕まえることができることを意味している．

2.3 複数のエージェントのオンライン探索モデル

エージェントが個別の ID を持つ場合と ID を持たない場合の 2 つを考える．G を ID を持つ複数のエージェントがオンライン探索できる最小のガード数を $as_o(G)$ と表す．G を ID を持たない複数のエージェントがオンライン探索できる最小のガード数を $as_a(G)$ と表す．また，エージェントはそれぞれメモリを持ち，非同期的に行動する．全てのエージェントは異なる頂点から同じアルゴリズムを開始すると仮定する．また，エージェントは探索開始時，G の形や頂点数，G 内のエージェントの総数を知らない．

各エージェントは探索中次のことを知ることができる．

- エージェントが辺 e を通過して v に到達したときの $\lambda_v(e)$．つまり通過してきた辺のポート番号．
- 同じ頂点にいる他のエージェント数
- 他のエージェントと辺ですれ違ったこと．辺上の同一点上にいることを**すれ違う**と言うことにする．このとき，すれ違ったエージェントの移動方向も分かる．

各エージェントは次の動作ができる．

- 頂点 u から隣接する頂点 v に辺 (u, v) を通過して移動する．辺を移動するエージェントは常に同じ速度で移動する（辺の途中で立ち止まることはない）．
- 同じ頂点上のエージェント及び辺ですれ違ったエージェントと情報の受け渡し

エージェントの動作に関して，Baba らは木状ネットワークでエージェントが集合する問題を議論している [1]．Baba らはエージェントに関して，同じ辺に他のエージェントがいたとしてもそのことを知ることはできず，エージェント同士の情報の交換もできないと仮定していた．しかし，グラフ探索問題では洞窟や建物をグラフと見なしているので，頂点や辺でエージェントの能力を分けることはしない（グラフを空間と考えているから）．そのため，このモデルでは辺ですれ違ったエージェントを知ることができるのは自然な仮定であると考える．

2.4 集合問題

本稿では木における複数のエージェントの**集合問題**を以下で定義する．

定義 1. 木 $T = (V, E)$ 内の全てのエージェントがある 1 つの頂点 $g \in V$ に集まる問題を集合問題と呼ぶ．また，このような頂点 g を集合点と呼ぶ．

本稿ではエージェントが集合した後で木の探索を行う．つまり，エージェントが集合できたことが分かる必要がある（つまり集合問題の終了判定）．提案アルゴリズムでは，集合のためのアルゴリズムが停止しない場合がある（詳細は第 3 章と第 4 章）．

集合問題を解く上で，木の**対称性** (symmetry) が重要であることが知られている [1]．

[1] 木 T が対称性を持つための必要十分条件は，[4] の用語を用いれば，対称度 (symmetricity)$\sigma(T) = 2$ となることである．

定義 2 ([1]). ラベル Λ を持つ木 T が対称性を持つことは以下の 3 つの条件を満たす関数 $f : V \to V$ が存在することである.

1. 任意の $v \in V$ について, $v \neq f(v)$ が成立する.

2. 任意の $u, v \in V$ について, u が v に隣接するとき, かつそのときに限り, $f(u)$ は $f(v)$ に隣接する.

3. 任意の辺 (u, v) について, $\lambda_u((u, v))$ は $\lambda_{f(u)}((f(u), f(v)))$ と等しい.

図 1 に対称性を持たない木と対称性を持つ木の例を示す [1].

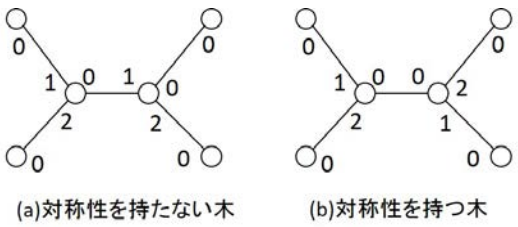


図 1: 対称性を持たない木と対称性を持つ木

Baba ら [1] は, ID を持たず, 互いに情報交換のできない複数のエージェントが 1 つまたは隣接する 2 つの頂点に集合する問題について議論している. ラベル Λ を持つ任意の木 T が中心を 1 つだけ持つとき, または, 対称性を持たないとき, エージェントは 1 つの頂点に集まることができる. 中心を 2 つ持ち, かつ対称性を持つとき, エージェントは隣り合う 2 つの頂点に集まることができる. そのアルゴリズム Gathering の概要を説明する.

Gathering ではエージェントは, 基本手順と呼ばれる手順でグラフ内を移動する. この基本手順は木 T の深さ優先探索をもとにした手順である. エージェントが頂点 v に到達したときに通過したポートを i とする. エージェントは次に $\lambda_v(e) = (i+1) \mod \deg(v)$ を満たす辺 e に進む. また, 本稿ではある頂点 v から基本手順でグラフの全ての頂点, 辺を訪れて再び v に戻ってくることを**走査**と呼ぶ.

Gathering は 3 つのフェーズから成る. フェーズ 1 では, 各エージェントは葉に到達するまで基本手順で移動する. 到達した葉を s とする. フェーズ 2 では, s から基本手順を用いて木全体を走査する. このとき, 木 T の形とラベル Λ の情報を集める. 最後にフェーズ 3 では, フェーズ 2 で集めた情報をもとにグラフの中心を計算する. 中心が 1 つの場合は, 全てのエージェントはその頂点に集まる. 中心が 2 つの場合でも, Λ を持つ木 T が対称性を持たないなら, 2 つの中心は区別できるので, エージェントは 1 点に集まることができる. しかし, 中心が 2 つかつ, Λ を持つ木 T か対称性を持つなら, 2 つの中心は区別できず, 各エージェントは 2 つの中心のどちらかに移動するため 1 つの頂点に集まることができない場合がある. 本稿では ID を持つエージェントの集合問題に関しては, 2 つの中心を結ぶ辺上でのエージェント同士の情報の受け渡しや ID の比較を行うことで, 全てのエージェントが 1 つの頂点に集合する.

3 ID を持つエージェントのオンライン木探索アルゴリズム

今回提案する探索アルゴリズムの概略は, まずラベル Λ を持つ木 T 内のエージェントが 1 つの頂点に集合し, エージェント数を確認する. その後, バリケードモデルの探索者となるエージェントを 1 人選び, 残りのエージェントをバリケードとして T の探索を行う. 本稿では, T 上の k ($k \geq 2$) 人のエージェントが T の 1 点で集合する方法について特に具体的に説明する. エージェントの集合に関する部分は Baba ら [1] の Gathering アルゴリズムを基にする. Gathering は木の 1 つまたは, 隣り合う 2 つの頂点に集まるアルゴリズムであったが, ここではエージェントの ID と互いの情報交換を用いることで 1 つの頂点に集まり, 全てのエージェントが集合できていることを確認するまでを行う.

探索アルゴリズムは以下の 3 つのステップに分かれる.

(移動ステップ) 各エージェントはGatheringのフェーズ1とフェーズ2を行い木TとラベルΛの情報を集める．そして集合点gを計算しそれぞれ移動する．
(確認ステップ)gに2人以上のエージェントが集まるとIDが最小のエージェントa_{min}を選び，g以外の頂点や辺にエージェントがいるか確認するために木Tを基本手順を用いて一度だけ走査する．全てのエージェントがgに集まるまで確認ステップの先頭に戻り，a_{min}の選択とTの走査を繰り返す．
(探索ステップ)gに集合したエージェントからリーダーを1人選び，バリケードモデルの探索者，その他のエージェントをバリケードとしてTを探索する．

3つのステップのうち，エージェントの集合は移動ステップと確認ステップで行う．移動ステップと確認ステップにおいて，ほとんどの移動は基本手順によって行う．

このアルゴリズムではエージェントはメモリ$state$の値に従って行動する．まず，$state$の取りうる値とそれぞれの意味を説明する．

Explore

アルゴリズム開始時の値．この状態のエージェントは全て集合点gの位置を知らない．

Select

ラベルΛを持つ木Tが中心を2つ持ちかつ対称性を持つ場合，1人のエージェントの計算だけでは集合点gを決定できない．その場合，$state =$ Selectに変更しgを決定する．

Stop

集合点gまで到達し停止している状態

Check

g以外の頂点や辺にエージェントがいないか確認しているときの状態

Wait

集合点gにいるエージェントのうち，$state =$ Checkでないエージェントはこの状態になり，$state =$ Checkのエージェントがg以外の頂点や辺にエージェントがいないか確認している間待機する．

$state =$ Exploreの場合，エージェントは集合点gを知らない．$state =$ Selectの場合は，木T，ラベルΛの情報全て持っているが，gの位置を知っているエージェントと知らないエージェントがいる．それ以外の場合はgの位置と木T，ラベルΛの情報を全て持っている．$state =$ Explore, Selectのエージェントは$state =$ Stop, Check, Waitのエージェントと同じ頂点にいるとき，またはすれ違う際に，gや木Tの形の情報を受け取ることで，一部の手順を省略することができる．

以下，移動ステップと確認ステップの詳細を説明する．

3.1 移動ステップ

初期状態で各エージェントの$state$はExploreである．エージェントはGatheringのフェーズ1，フェーズ2を行い，木Tの形とラベルΛを求める．そして，ラベルΛを持つ木Tが中心を1つしか持たない場合，および，中心が2つかつ対称性を持たない場合は，Gatheringのフェーズ3によって集合点gを唯一に決定することができる．エージェントはgへ移動し，gに$state =$ Waitのエージェントがいれば自身の$state$もWaitに変更する．それ以外の場合は$state$をStopに変更する．そして確認ステップに進む．

中心が2つかつ対称性を持つ場合は，フェーズ1で初めて到達した葉sからの距離が大きい方の中心c'へ基本手順で移動する．c'へ移動するまでに他のエージェントから集合点gの位置を受け取れなかった場合，$state$をSelectに変更し，2つの中心とその間の辺を往復し続ける．$k \geq 2$であるので，少なくとも2人のエージェントが2つの中心とその両方を端点に持つ辺に集まる．$state =$ Selectかつ，gの情報を持たないエージェント同士がすれ違ったとき（もしくは同じ頂点にいるとき），IDを比較しIDの最も大きいエージェントが進もうとしている（もし

くは今いる）頂点を集合点 g に決定する．ただし，$state$ を Select に変更した後一度も移動していないエージェントだけで g は決定しない（図 2）．また，辺上で同じ方向に移動しているエージェントのみで g を決定することはしない（図 3）．これらの場合に g を決定すると，異なる g を持つエージェントが存在する可能性があるからである．例えば，図 2 はエージェント a_1 と a_2 が同時に中心の一方 c_a に到達し，それと同時に a_3 と a_4 がもう一方の中心 c_b に到達し，それぞれのエージェントが $state$ を Select に変更した直後を表している．a_1, a_2, a_3, a_4 は g を知らないと仮定する．このとき，c_a と c_b には g を知らないエージェントが 2 人ずついるが，このまま g を決定すると，a_1 と a_2 は c_a，a_3 と a_4 は c_b を g に選択してしまう．

図 2: g を決定しない場合の例 1

次に図 3 は図 2 の状況から，a_1 と a_2 が同時に c_a から移動を開始し，a_3 と a_4 も同時に c_b から移動を開始した状況である．このとき，a_1 と a_2，a_3 と a_4 は辺上の同じ点にいる（情報の受け渡しが可能）が，ここで g を決定してしまうと，a_1 と a_2 は c_b を，a_3 と a_4 は c_a を g に選択してしまう．

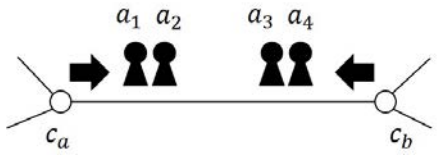


図 3: g を決定しない場合の例 2

g を決定した後は，g へ移動し，g に $state = \text{Wait}$ のエージェントがいれば自身の $state$ も Wait に変更する．それ以外の場合は $state$ を Stop に変更する．そして確認ステップに進む．

集合点 g を知っているエージェントは g を知らないエージェントに g の情報や木 T，ラベル Λ の情報を教える．g を教えてもらったエージェントは自身で g を計算する必要がなくなるので，移動ステップの一部を省略できる．より具体的には，もし g を知らないエージェントが g を知っているエージェントと同じ頂点にいたり，すれ違ったりした場合は，g や木 T，ラベル Λ の情報を受け取り，g へ基本手順で移動することで移動ステップの一部を省略する．

移動ステップは，T 内の全てのエージェントは同じ頂点を g に決定できることを保証する．

3.2 確認ステップ

g には移動ステップで必ず 2 人以上のエージェントが集まる．$state = \text{Stop}$ であるエージェントが 2 人以上になると，ID が最小のエージェント a_{min} を選ぶ．そして，a_{min} の $state$ を Check，それ以外の頂点 g のエージェントの $state$ は Wait とする．

a_{min} は 1 度だけ T 全体を基本手順で走査する．このとき，すれ違ったエージェントの有無を記録しておく．T の走査を終えた後，すれ違ったエージェントがいれば確認ステップの先頭に戻り，最小の ID を持つエージェントを選んで，再び木全体を走査することを繰り返す．もしいなければ全てのエージェントは g に集まっている．

確認ステップの正当性に関して，次の補題が成立する．

補題 1. a_{min} が基本手順で T 全体を走査したとき，頂点 g 以外ですれ違うエージェントがいなければ，全てのエージェントは g に集まっている．

3.3 探索ステップ

確認ステップで，全てのエージェントは木の 1 つの頂点に集合している．集合したエージェントの中から，ID が最小のエージェントを選び，そのエージェントをバリケードモデルの探索者，残りのエージェントをバリケードとみなして，バリケードモデルの

オンライン探索アルゴリズムを実行する．このアルゴリズムより，以下の補題が導かれる．

補題 2. 任意の $as_o(T) \geq 2$ である木 T について，$as_o(T) \leq r(T) + 1$ が成立する．

エージェント数が 1 のときについても考えなければならない．$k = 1$ のとき，提案アルゴリズムでは，このエージェントは T 内のエージェントは自身 1 人なのか，それ以上存在しているのか知ることができない．エージェント 1 人で探索できる木（つまり道グラフ）の場合は，木の形を求めた時点で探索が成功していることが分かる．よって，与えられた木 T が道グラフであった場合は，木の形を求めた時点で探索成功とし，アルゴリズムを停止すればよい．しかし，T がエージェントが 1 人では探索できない木であった場合，木内に存在する 1 人のエージェントは集合アルゴリズムを終了することができない．以上より次の定理が成立する．

定理 3. ラベル Λ を持つ任意の木 T について，$as_o(T) = es(T)$ である．

4 IDを持たないエージェントのオンライン木探索アルゴリズム

基本的なアルゴリズムの方針は ID を持つ場合と似ている．アルゴリズムの流れは，まずはエージェントが 1 つの頂点に集合し，その後リーダーを決めてバリケードモデルのアルゴリズムで探索する．ここで問題になるのはリーダーの決め方である．エージェントが ID を持っているときは ID の大小関係でリーダーを決めることができたが，エージェントが ID を持たない場合この方法は使えない．ここでは，各エージェントは異なる頂点からアルゴリズムを開始するという仮定より，アルゴリズム開始時からエージェントが集合点 g に集まるまでに通過したポート番号の列 P を記録し，それを ID のように用いることにする．より正確には，頂点 v に到達したとき，どのポート番号の辺を通過してきたかを記録する．P の比較でエージェントを区別したり，P を辞書式順序で並べたときに先頭に来るエージェントを 1 人選択したりすることができる．

しかし，次の条件 C を満たす場合は，この提案アルゴリズムではエージェントは 1 点に集合できない場合がある．この条件は定義 2 の条件を拡張したものである．

（条件 C）以下の 4 つを満たす関数 $f : V \to V$ が存在する．

1. 任意の $v \in V$ について，$v \neq f(v)$ が成立する．

2. 任意の $u, v \in V$ について，u が v に隣接するとき，かつそのときに限り，$f(u)$ は $f(v)$ に隣接する．

3. 任意の辺 (u, v) について，$\lambda_u((u, v))$ は $\lambda_{f(u)}((f(u), f(v)))$ と等しい．

4. 任意の $v \in V$ について，初期状態で v にエージェントがいるとき，かつそのときに限り，$f(v)$ にもエージェントがいる．

条件 C を満たす場合，辺上で P の比較を行ったとき異なるエージェント同士の P が同じ場合がある．そのため，P の比較を行ったエージェント間の大小関係が定まらない場合がある．

ID を持たない複数のエージェントによる探索アルゴリズムも移動ステップ，確認ステップ，探索ステップの 3 つに分けられる．各ステップの詳細を説明する．

4.1 移動ステップ

基本的には，エージェントが ID を持つ場合と同じである．ただし，エージェントが ID の比較を行っていた部分に関しては，通過したポートの列 P を比較する動作に置き換える．具体的には，ID が最小もしくは最大のエージェントを選ぶ際は，P の辞書式順序比較し並べ，先頭に来るエージェントを選択する．

注意すべき点は，エージェントが P の比較を行った際，辞書式順序で先頭に来るエージェントが 1 人

に定まらない場合があることである（図 4）．そのため，異なる中心を集合点に選ぶエージェントがいる可能性がある．最終的に全てのエージェントが 1 つの頂点に集合できるか否かは確認ステップの最後で判断することにし，移動ステップでは，エージェントが 2 つの頂点に集まっていてもよいことにする．

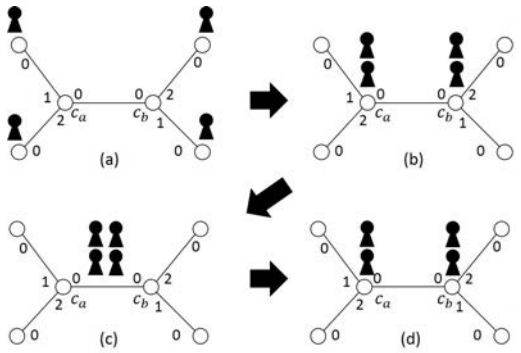


図 4: 1 つの頂点に集合できない例

図 4(a) は条件 C を満たす木とエージェントの初期配置の例である．図 4(b) は全てのエージェントが同時に $state = \text{Select}$ になった直後の状態を表す．その後，全てのエージェントが同時にもう一方の中心に移動しようとすると，2 つの中心を端点に持つ辺で 4 人のエージェントが同時にすれ違う (図 4(c))．このとき，4 人のエージェントの P を辞書式順序で並べると先頭 2 つとその後ろの 2 つの P はそれぞれ同じ値であるので，この 4 人では集合点 g を決定できない．それぞれが頂点に到達すると，c_a と c_b はどちらとも $state = \text{Select}$ かつ，$state = \text{Select}$ に変更されて少なくとも一度は移動しているエージェントが 2 人いるので，それぞれのエージェントは今いる頂点を集合点 g に決定してしまう．

4.2 確認ステップ

確認ステップでは T が (i) 中心が 1 つまたは，中心が 2 つかつ対称性を持たない場合と，(ii) 中心が 2 つかつ対称性を持つ場合で内容が変わる．確認ステップに入ったエージェントは既に木 T とラベル Λ を知っているので，(i) と (ii) は区別できる．まず (i) の場合について考える．(i) の場合，全てのエージェントは各自の計算で同じ集合点 g を計算できる．エージェントが ID を持つ場合のアルゴリズムの確認ステップと同様の手順で全てのエージェントが g に集合できているか確認する．ただし，エージェントは ID を持たないので，通過したポートの列 P を用いて a_{min} を決定する．

(ii) の場合，移動ステップで求めた集合点（これは中心の一方）を g，もう一方の中心を g' とする．(i) と同じ理由で，通過したポートの列 P を用いて，a_{min} を決定し，T を走査する．a_{min} が再び g に戻ってくるまでに g と g' 以外で他のエージェントと同じ頂点にいたり，すれ違ったりしなければ，T 内のエージェントは全て g と g' に集まっている．g や g' 以外の頂点，辺にエージェントがいるなら，a_{min} を選び，T の走査を繰り返す．また，確認ステップで T を走査したとき，g' にエージェントがいるか否か分かる．g' にエージェントがいないのであれば，探索ステップへ進む．もし，g' にエージェントがいるなら，最後に再び a_{min} を選ぶ．選ばれたエージェントは g' に移動し，g に集まったエージェント数と g の全てのエージェントの P を辞書式順序で並べ，先頭から順につなげた列 $\mathcal{P}$ を伝える．エージェント l 人の P からなる $\mathcal{P}$ は $\mathcal{P} =<$（1 人目の P）;（2 人目の P）; ... ;（l 人目の P）$>$ のように構成する．

ここまでは g' に集まったエージェントも同じ手順を行っている．g と g' のそれぞれに集まっているエージェント数が分かると，集まったエージェント数が少ない方の頂点にいるエージェントはもう一方の中心に移動する．しかし，エージェント数 k が偶数のとき，g と g' でエージェント数が等しい場合がある．このときは互いの $\mathcal{P}$ を比較し，辞書式順序で先頭に来る方が，もう一方の中心に移動する．エージェントが 1 点に集まると探索ステップに進む．

ただし，アルゴリズム開始時に次の条件 C を満たすときは，g に集まったエージェントで求めた列 $\mathcal{P}$ と g' に集まったエージェントから受け取った列 $\mathcal{P}$ が一致する場合があり，そのときは集合点を 1 つに決

定できない．例えば，図 4(d) の後，c_a と c_b に集まったエージェントは，T を一度だけ走査し他のエージェントがいないことを確認すると，c_a と c_b のエージェントは互いにエージェント数と $\mathcal{P}$ を伝えるが，エージェント数はどちらも 2 人で，$\mathcal{P}$ も同じ値であることから，1 点に集合することができない．この場合は $k/2$ 人のエージェントの 2 つのグループがそれぞれ探索ステップに進む．

また，特に $k = 2$ かつ条件 C を満たす場合，2 人のエージェントは $state$ = Select のままで確認ステップを終了できないことがある．

4.3 探索ステップ

集合したエージェントの中から，P が辞書式順序で先頭に来るエージェントを選ぶ．選ばれたエージェントを探索者，残りの人のエージェントをバリケードと見なして，木 T の探索を行う．

提案アルゴリズムではエージェント数が偶数の場合のみ 1 点に集合できないことがある．エージェント数が奇数であれば必ず 1 つの頂点に集合できることから，次の定理が導かれる．

定理 4. ラベル Λ を持つ任意の木 T について，$as_a(T) \leq es(T) + 1$ が成立する．

5 まとめ

複数のエージェントによるオンラインの木探索アルゴリズムを提案した．任意の木 T について，エージェントが ID を持つ場合は，エージェント数は $es(T)$ で十分であることを示した．一方，エージェントが ID を持たない場合，エージェント数は $es(T) + 1$ で十分であることを示した．今後の課題は，ID を持たないエージェントのオンライン探索について，任意の木 T を $es(T)$ 人のエージェントで探索することができるか否か考察する．また，複数のエージェントによる一般のグラフのオンライン探索アルゴリズムを考える．

参考文献

[1] D. Baba, T. Izumi, F. Ooshita, H. Kakugawa and T. Masuzawa, ”Linear time and space gathering of anonymous mobile agents in asynchronous trees.”, Theoretical Computer Science 478, pp. 118–126, 2013.

[2] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou, “The complexity of searching a graph”, Journal of the ACM 35(1), pp. 18–44, 1988.

[3] T. D. Parsons, “Pursuit-evasion in a graph”, In *Proceedings of the Theory and Applications of Graphs*, pp. 426–441, 1976.

[4] M. Yamashita, and T. Kameda,”Computing on anonymous networks. I. Characterizing the solvable cases”, *Parallel and Distributed Systems*, IEEE Transactions on 7(1), 69–89, 1996.

[5] 山下雅史, “捜索—移動する対象を探索する”, 室田一雄編, 離散構造とアルゴリズム III, 近代科学社, pp. 115–162, 1994.

[6] 八神貴裕, 山内由紀子, 来嶋秀治, 山下雅史, “バリケードを用いたオンライン木探索について”, 一般社団法人情報処理学会九州支部 火の国情報シンポジウム 2015 論文集, 4A-1, 2015.

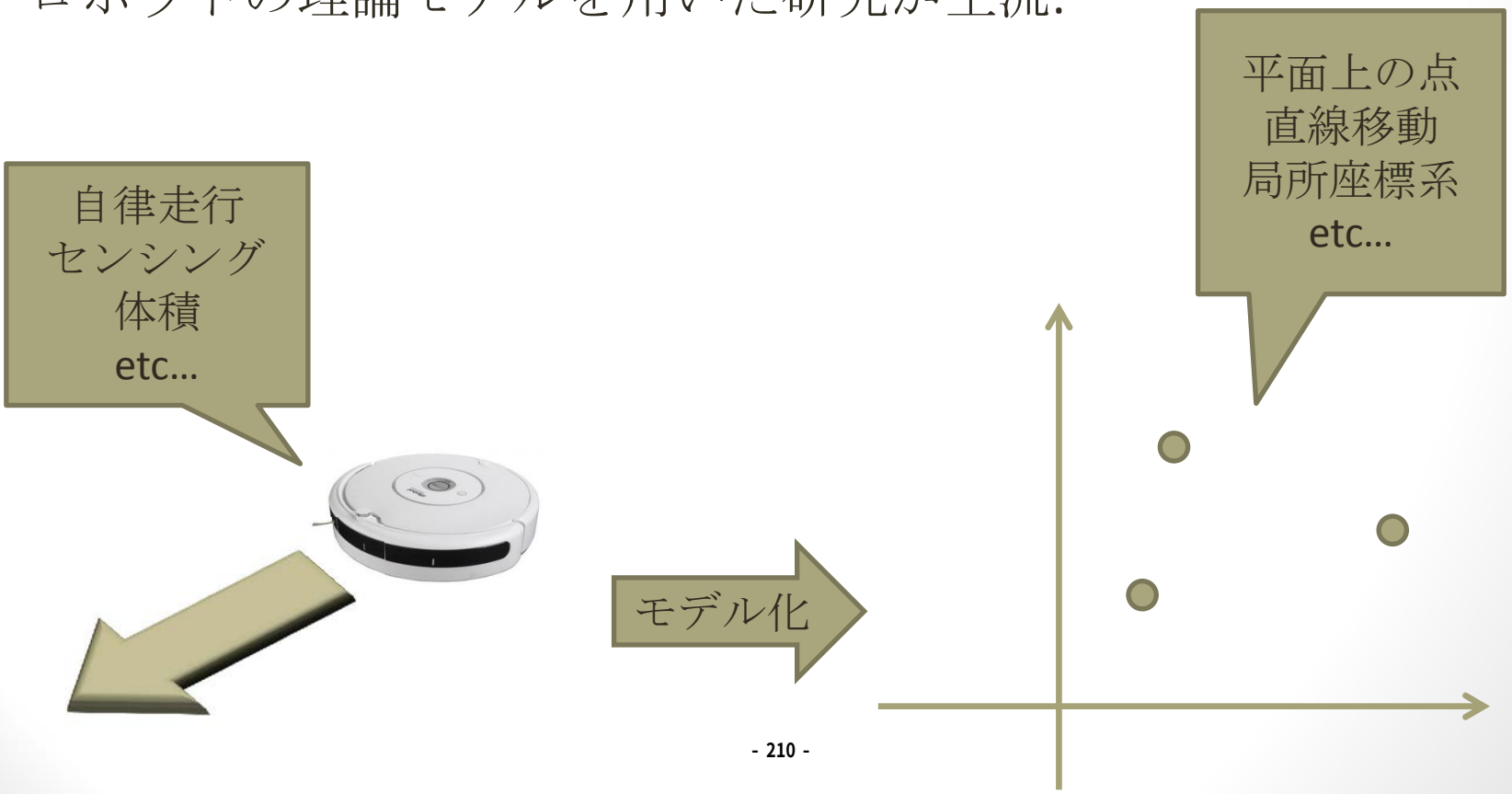
状態を持つ自律分散ロボット群を用いた集合問題の可解性について

寺井 智史　　法政大学大学院　◎
和田幸一　　法政大学
片山喜章　　名古屋工業大学大学院

はじめに

- 自律分散ロボット群の研究

　自律的に動作し,全体としては協調的に行動する.
ロボットの理論モデルを用いた研究が主流.



はじめに

■研究背景

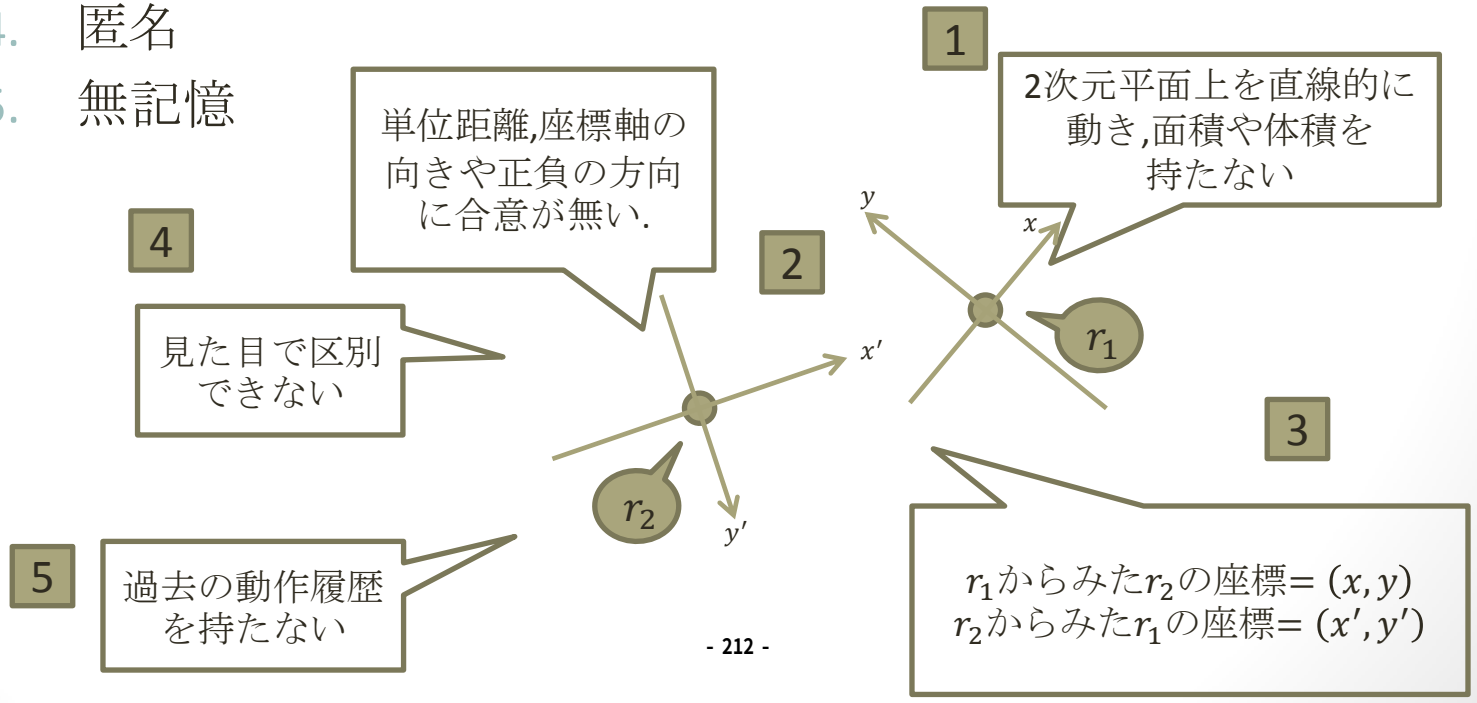
最も単純なモデルでは非可解な問題を解くために状態$(light)$を持つロボットモデルが考案された.

■研究概要

本研究では,ロボットに状態を持たせたとき,どの程度計算能力が向上するのかを調べる.

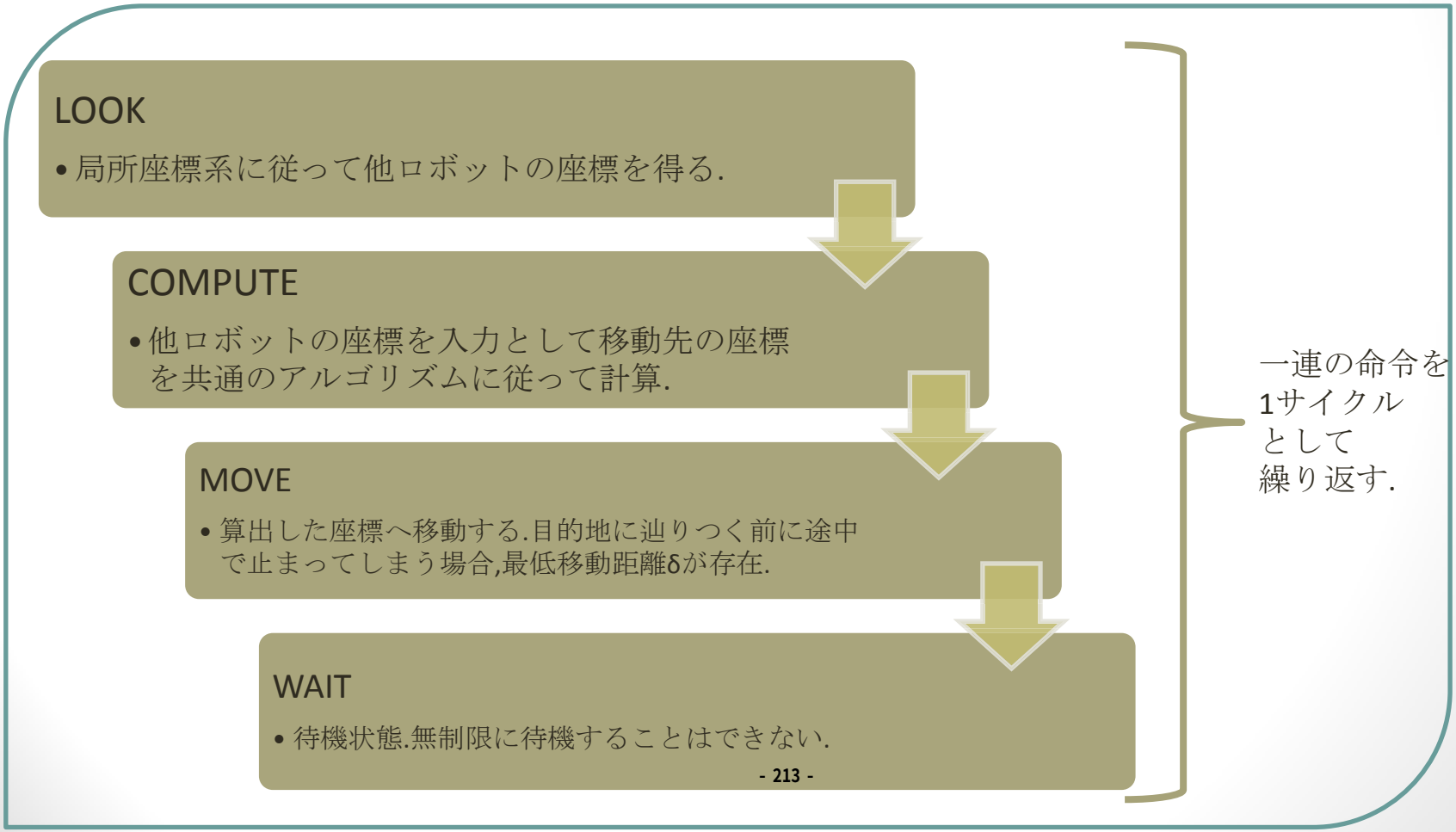
理論モデル

- 基本的なモデルの紹介
  - 仮定
    1. 点として扱う
    2. 局所座標系を持つ
    3. 他ロボットの位置を認識できる
    4. 匿名
    5. 無記憶



理論モデル

- 基本的なモデルの紹介
  - 動作



スケジュール

スケジュールによっては目的の行動をとれない(問題を解けない)ことがある.

そこで

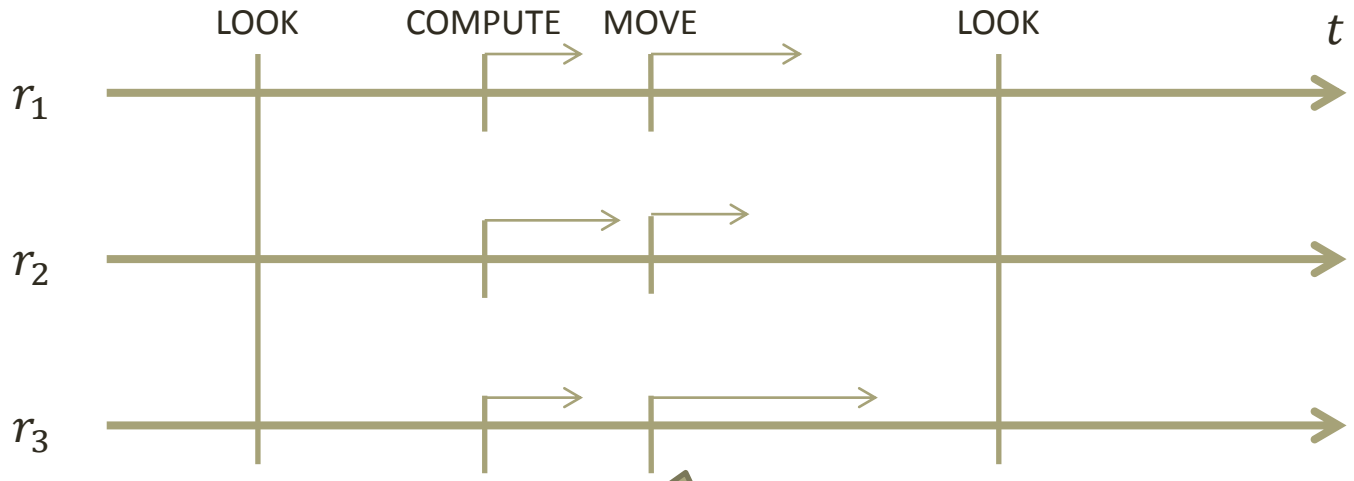
- $FSYNC(Fully\ syncronous)$
- $SSYNC(Semi\ syncronous)$
- $ASYNC(Asyncronous)$

の3種類のスケジュールがよく使われる.

スケジュール

- FSYNC

各ロボットの動作を時系列に沿って並べると…

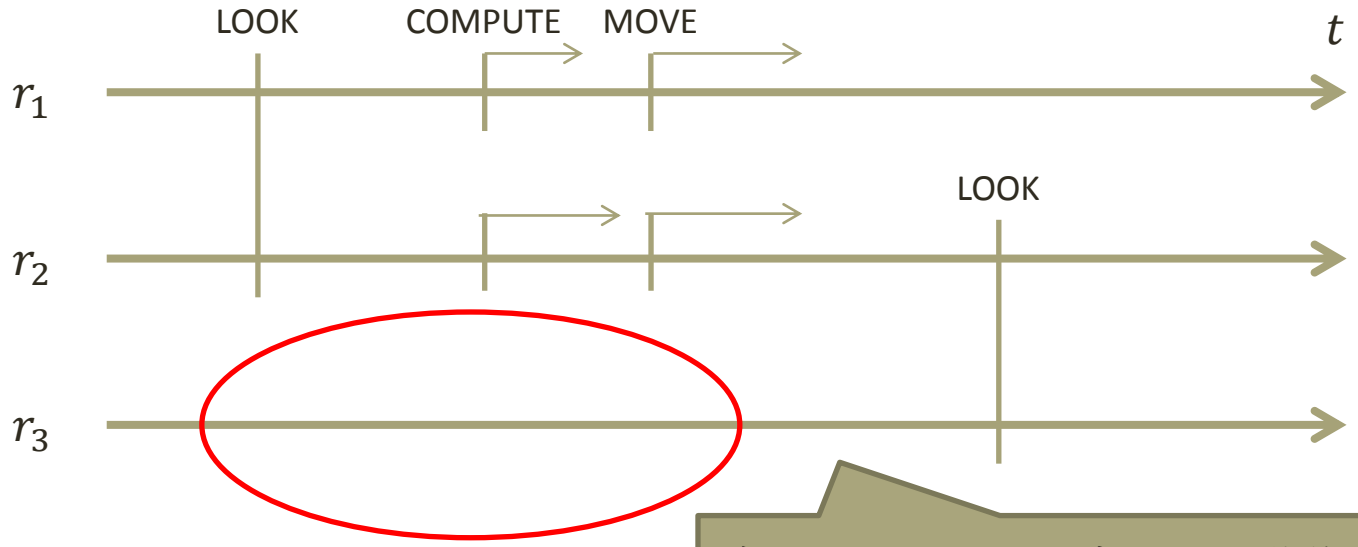


全サイクルで
全ロボットが同期して
動作する.

スケジュール

- SSYNC

各ロボットの動作を時系列に沿って並べると…

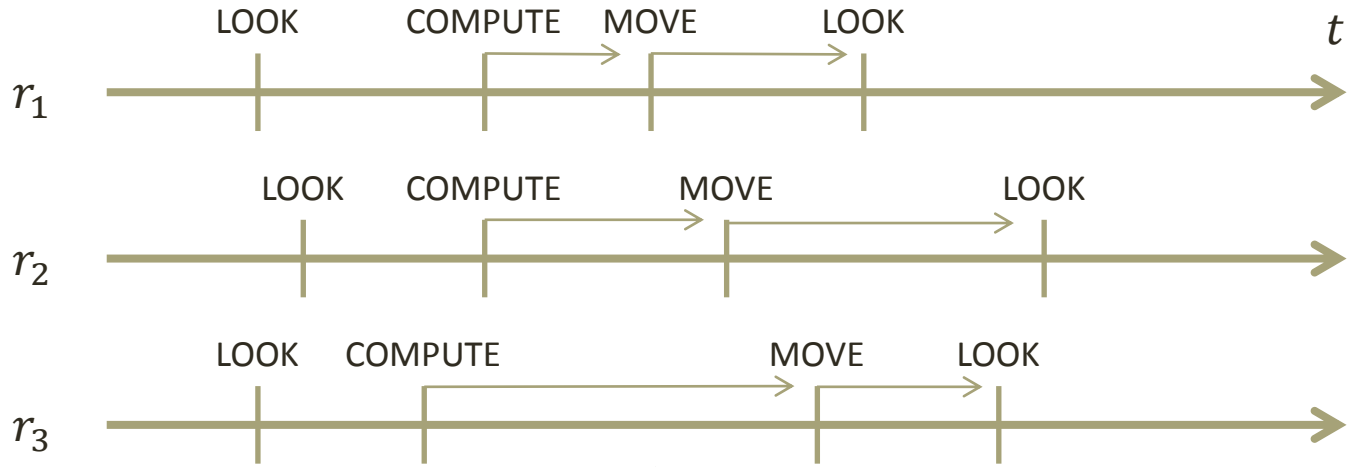


各サイクルでロボットの動作は同期しているが,サイクルを実行しないロボットが存在している.

スケジュール

- ASYNC

各ロボットの動作を時系列に沿って並べると…



まったく同期していない

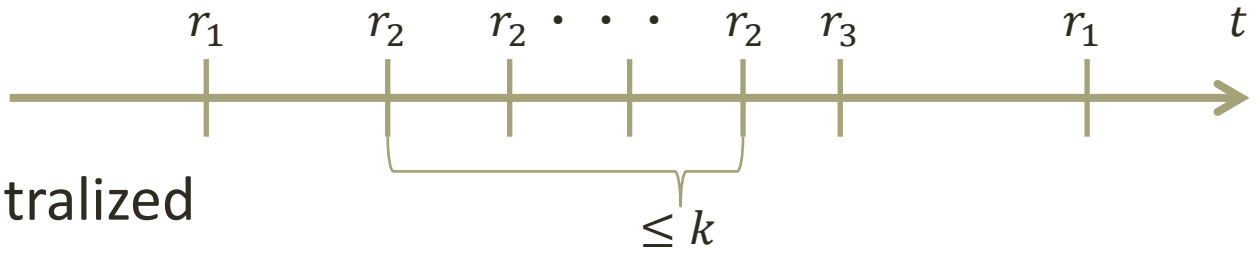
スケジュール

今回用いるスケジュールは前提としてfairであることを仮定する.

- fair
  - どのロボットも無限回動作

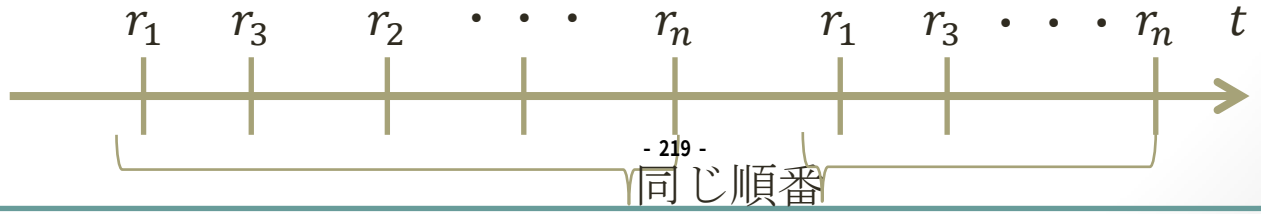
特殊なスケジュール

- k-bounded
  - 任意のロボットが2回動作する間に他ロボットが高々k回動作



- centralized
  - 必ず1台ずつ動作

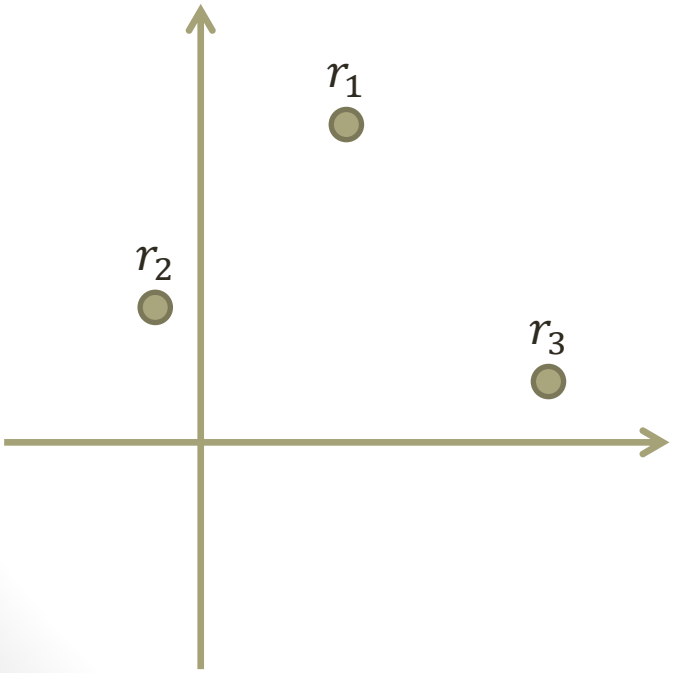
- round-robin
  - 1-bounded centralizedに等しい



問題定義

- 集合問題

$n(\in \mathbb{N})$台のロボットが,任意の初期配置から予め決められていない1点に集まることができるか,という問題.



スケジュール	可解性
ASYNC	×
SSYNC	×
FSYNC	○

状態を持つロボット

- 状態($light$)

  ロボットの内部状態を記録できる定数ビットの記憶領域.
  状態の可視性によって以下のようなモデル[1]に分ける.

	自分の状態	相手の状態
$full-light$	○	○
$internal-light$	○	×
$external-light$	×	○

引用した論文内では
internal-light→FSTATE , external-light→FCOMMと紹介

[1]P.Flocchini , N.Santoro , G.Viglietta , M.Yamashita , Rendezvous of Two robots with Constant Memory , 20th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2013) , Lecture Notes in Computer Science 8179, pp 189-200.

状態による可解性の拡張 $(n = 2)$

[2]

スケジュール	可解性
centralized	○
k-bounded($k \geq 1$)	×

[3]

スケジュール	*full*	*internal*	*external*
ASYNC	≤ 12	?	12
SSYNC	≤ 6	6	≤ 12
FSYNC	1	1	1

スケジュール	*full*	*internal*	*external*
ASYNC	≤ 3	?	3
SSYNC	2	3	3
FSYNC	1	1	1

※赤字はδの知識あり

[2] X D'efago , M Gradinariu , P Julien , C St'ephane , M Philippe , R Parv'edy , Fault and Byzantine Tolerant Self-stabilizing Mobile Robots Gathering — Feasibility Study — , 20th International Symposium, DISC 2006, Lecture Notes in Computer Science , 4167 , pp 46-60.
[3] P.Flocchini , N.Santoro , G.Viglietta , M.Yamashita , Rendezvous of Two robots with Constant Memory , 20th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2013) , Lecture Notes in Computer Science 8179, pp 189-200.

状態による可解性の拡張 $(n \geq 3)$

[2]

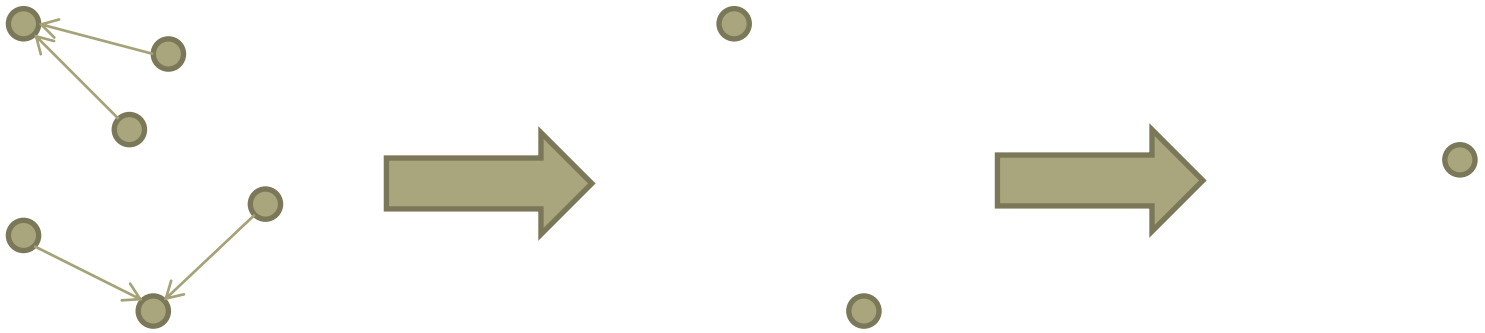
スケジュール	可解性
2-bouded centralized	×(distinct)
round-robin	×(SS)

スケジュール	*full*	*internal*	*external*
SSYNC	3	?	2
centralized	≤ 3	?	2
round-robin	≤ 3	2	≤ 2

※赤字は移動が厳密な場合

アルゴリズム概要

初期配置から2点もしくは1点に集まることが出来るアルゴリズム[4]と,2台の集合問題のアルゴリズムを拡張することで3台以上の集合問題を解くアルゴリズムを考えた.



[4] T Izumi, Y Katayama, N Inuzuka, and K Wada, Gathering Autonomous Mobile Robots with Dynamic Compasses: An Optimal Result, 21st International Symposium on Distributed Computing (DISC 2007), Lecture note in Computer Science 4731, pp 298-312,

今後の課題

状態無しでは非可解な仮定のいくつかに対して状態を持たせることで可解になることを示した.引き続き様々な仮定での可解性を調べていく。

- 目標
  - ロボットに状態を持たせたとき,どの程度能力が上がるのかを明らかにする.

自律分散ロボット群の理論モデルに対する実機シミュレーションに関する研究

法政大学大学院　田邉太一
法政大学　　和田幸一
名古屋工業大学　片山喜章

研究背景

自律分散ロボット群の研究では理論モデルが主体で実際に動作させるということが少なく、実現が難しい部分がある

⇒理論モデルを実現する際にどう差を少なくするか

⇒実際のロボットの動作を理論モデルの参考にする

研究内容

アルゴリズムとスケジュールを記述し、スケジュールに従ってロボットを動作させるための動作制御と記述したスケジュールに従ってロボットが動作しているか確認を行うシステムの作成

理論モデル上のロボットの動作

- ロボットはWAIT,LOOK,COMPUTE,MOVEをサイクルとして繰り返す
  - ◆WAIT

    待機状態
  - ◆LOOK

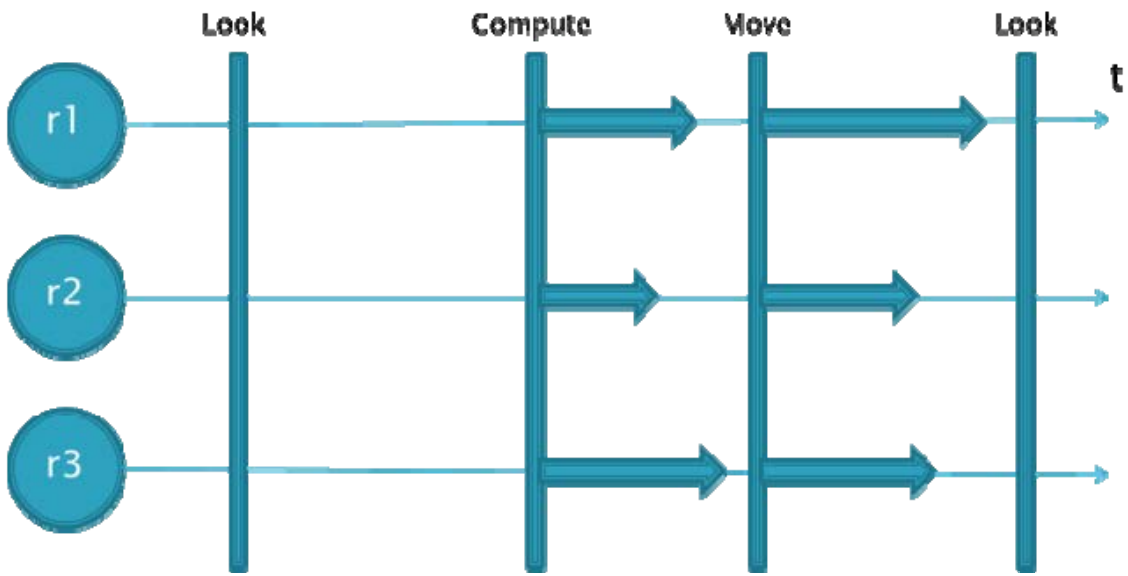
    計測するロボットを原点として、視界に有るロボットの座標を計測する
  - ◆COMPUTE

    LOOKで得た情報を元に目的地を計算する
  - ◆MOVE

    目的地へ移動する

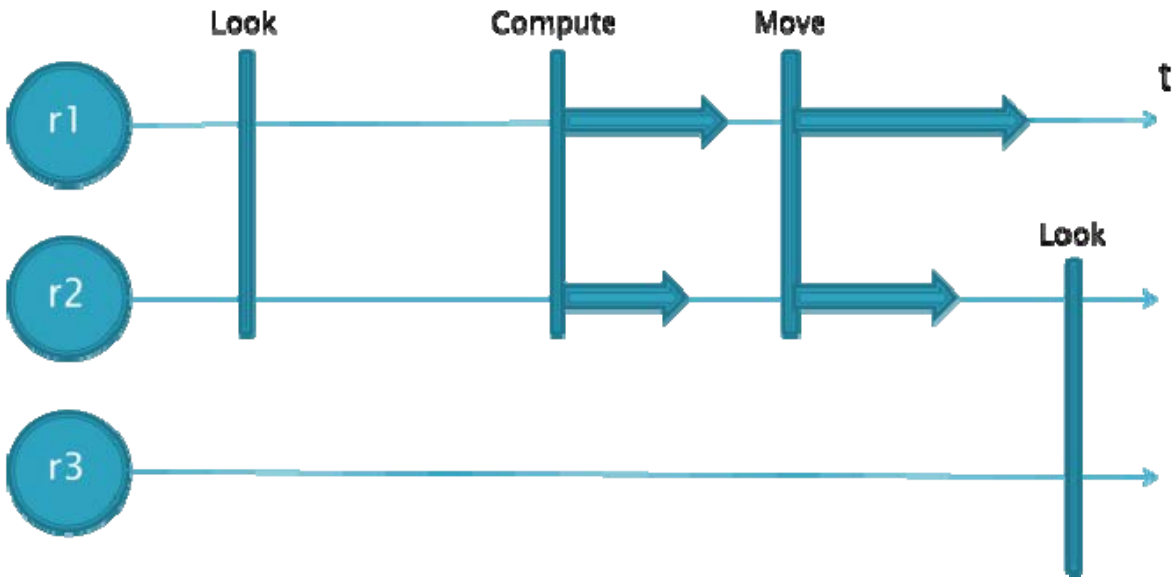
理論モデル上のロボットの同期

- FSYNC（完全同期）



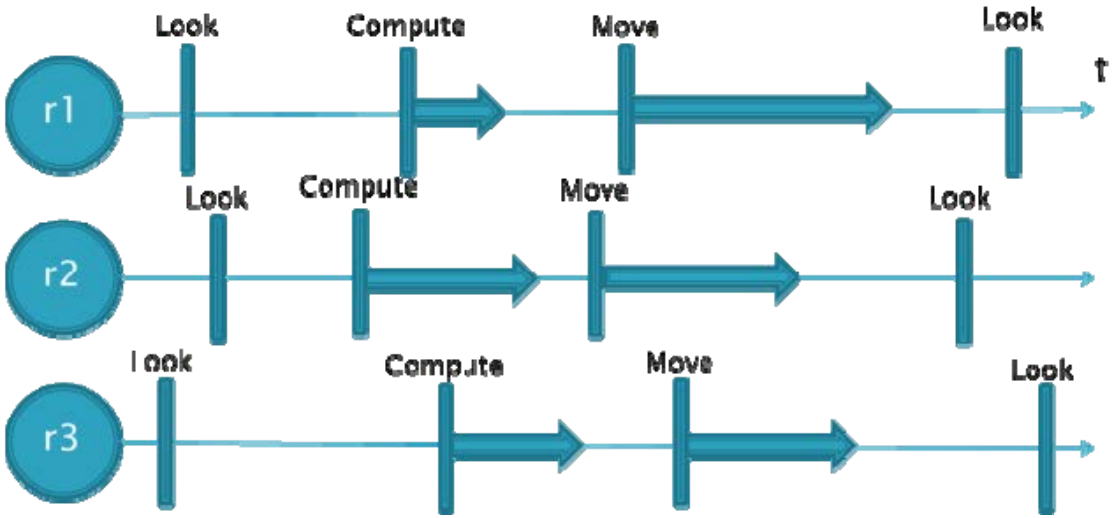
理論モデル上のロボットの同期

- SSYNC（半同期）



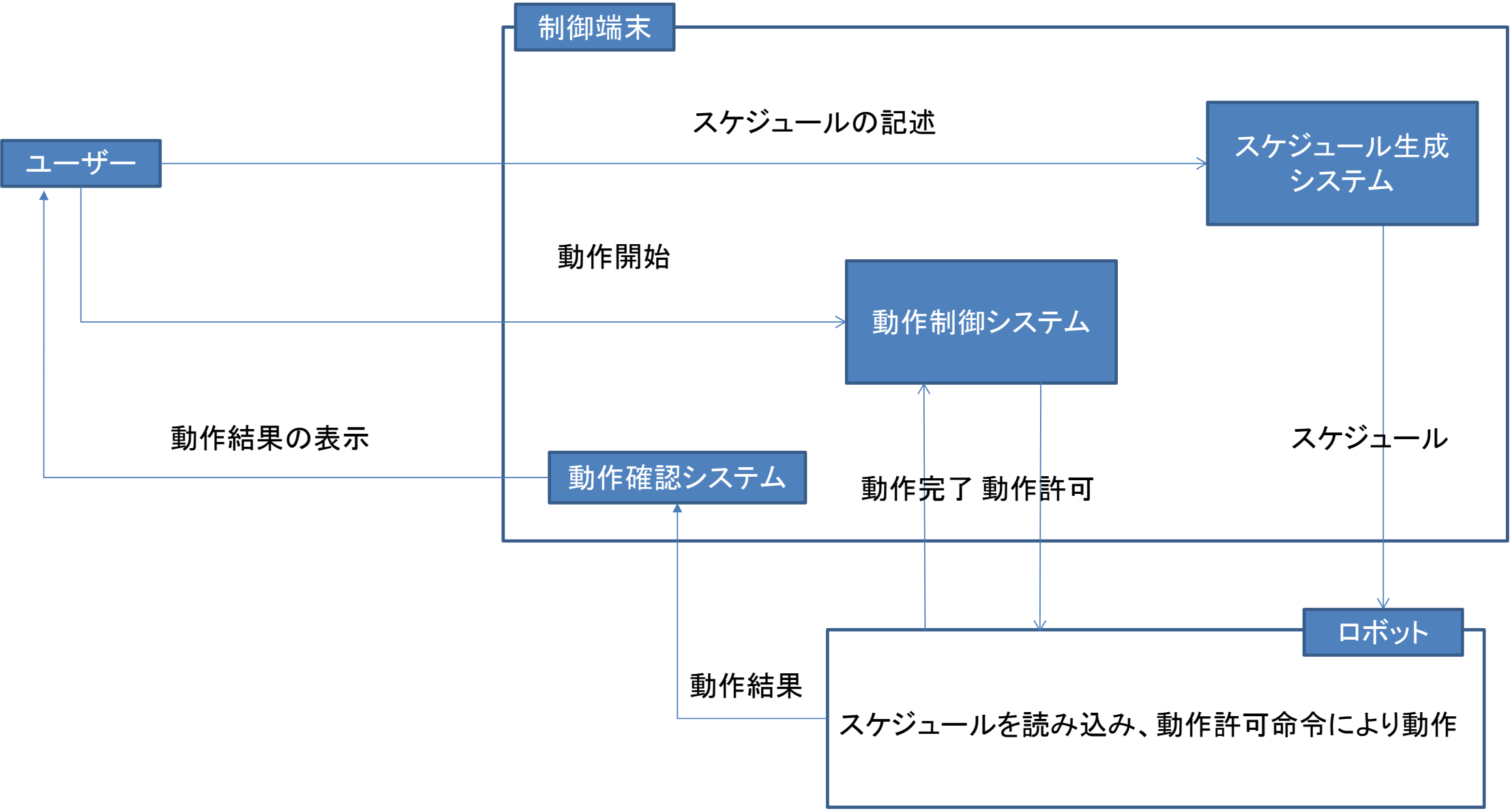
理論モデル上のロボットの同期

- ASYNC（非同期）



システムの各機能

- スケジュール生成システム
  - ユーザーから入力された情報をもとにスケジュールを作成しロボットに送る
- 動作制御システム
  - ロボットの動作の同期をとる際に使用する
- 動作確認システム
  - ロボットから受けとった動作完了信号（ロボットの実動作時間）からロボットの時間ごとの動作を表示する
- ロボット
  - 受け取ったスケジュールと送信制御に従って動作し、動作が終了したら制御端末に送信する


制御端末
スケジュールの記述
ユーザー
スケジュール生成
システム
動作開始
動作制御システム
動作結果の表示
スケジュール
動作確認システム
動作完了 動作許可
ロボット
動作結果
スケジュールを読み込み、動作許可命令により動作

ロボットの実現

理論モデルのロボットを以下のもので実現する

- Kinect
  - ◆観察
- PC
  - ◆計算
- ルンバ,Turtlebot
  - ◆移動
- 色紙
  - ◆ロボットの認識

ロボットの認識

Kinectで周囲の画像を取得し,ある程度の大きさの色の塊を発見した場合ロボットとして色の塊の中心をロボットの座標とし,それまでに回転した角度と距離を取得する

動作の実現

- 動作
  - ◆LOOK

    ロボットを１回転させ、Kinectで他のロボットを発見するまでに
    かかった角度と距離を計測する
  - ◆COMPUTE

    LOOKで得た結果をもとに目的地までの角度と距離を計算する
  - ◆MOVE

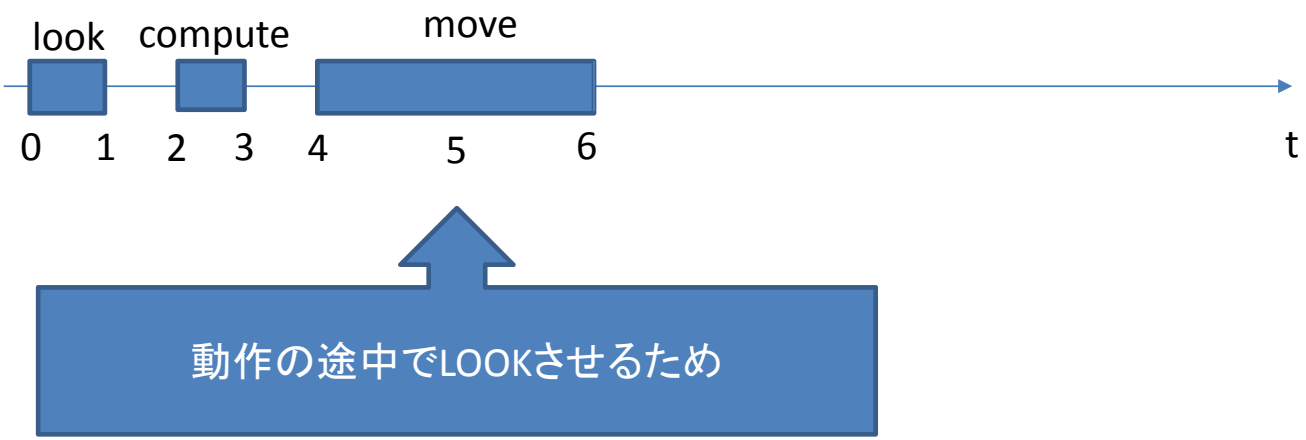
    COMPUTEで得た角度ロボットを回転させ、計算した距離移動する

スケジュール生成システム

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16(単位時間)
robot1																	
robot2																	
robot3																	
robotn																	

単位時間の定義

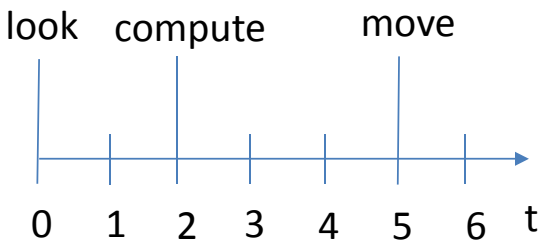
単位時間は1つの動作を開始してから終了するまでを1とする
またMOVEの動作を分割して1とすることもできる



スケジュールの記述形式

サイクルの記述形式は以下のようにする

サイクル名	
動作	次の動作まで待機する単位時間



サイクルA	
look	1
compute	2
move	

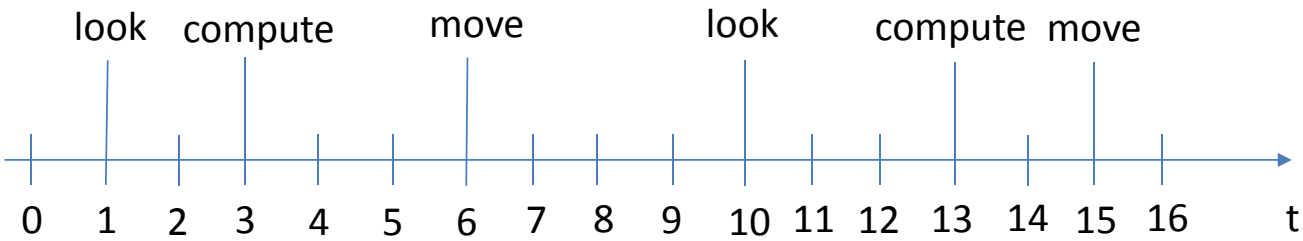
スケジュールの記述形式

スケジュールの記述形式は以下のようにする

サイクル名	
動作	次の動作まで待機する単位時間

スケジュール
次のサイクルまで待機する単位時間、またはサイクル名
サイクル名（繰り返し数）または各動作
繰り返しは複数のサイクルでもできる

スケジュールの記述例



サイクルA	
look	2
compute	3
move	4

サイクルB	
look	3
compute	2
move	1

スケジュール
1
サイクルA（1）
サイクルB（1）

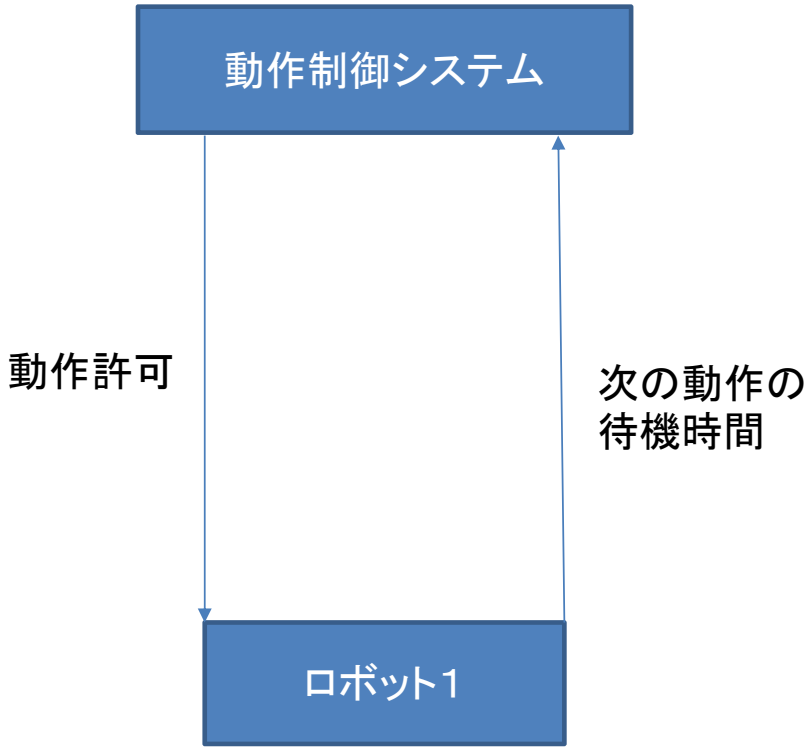
動作制御システム

動作制御システム

・動作制御システムの動作
各ロボットに動作を行う許可を出す

・動作制御システムが記録するもの
1.受け取った次の動作までの待機時間
2.ロボットから動作完了信号を受け取ったか

・動作許可の送信条件
全てのロボットから動作完了を受け取り、待機時間が0のロボットがある場合に待機時間が0のロボットのみに動作許可を送信する



動作制御システムのアルゴリズム

ロボット側

```
While(動作開始命令=false){
}
If(スケジュールの先頭=待機時間){
        制御システムへ送信
}
While(目的達成=false){
        if(動作許可命令=true){
                動作実行
                待機時間を送信
        }
}
```

動作制御システム側

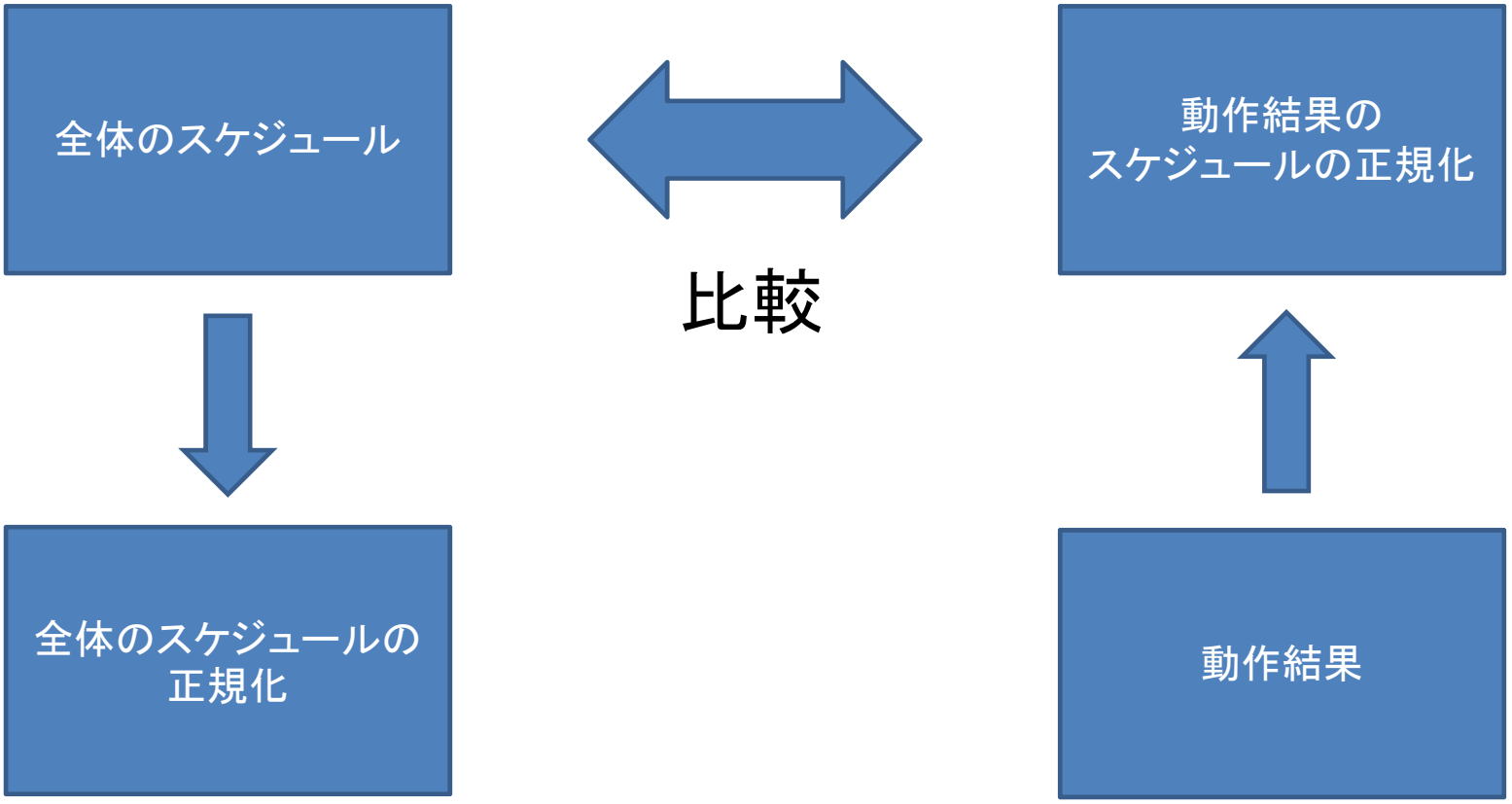
```
動作開始命令を送信
While(待機時間=false){
}
While(目的達成=false){
        if(待機時間=0){
                動作許可命令を送信
                待機時間-1
        }
        else{
                待機時間-1
        }
}
```

動作確認システム

ロボットが動作許可命令を受け取ってから動作完了命令を送信するまでの時間をロボットが動作を行った時間とする

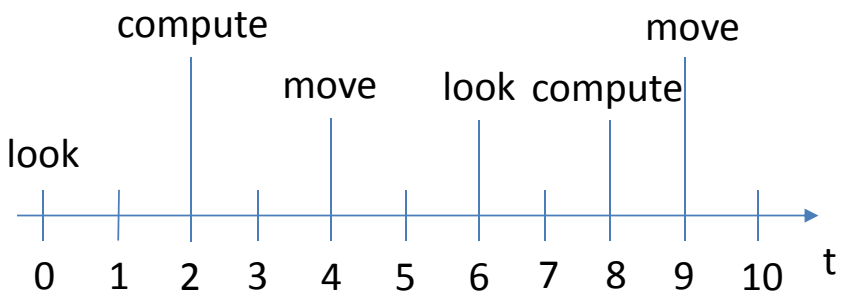
ロボットが目的を達成した時に全てのロボットから動作確認システムに全ての動作を行った時間を送信し、それを元にスケジュールを正規化して記述されたスケジュールに従って動作していたか確認する

動作結果の確認方法

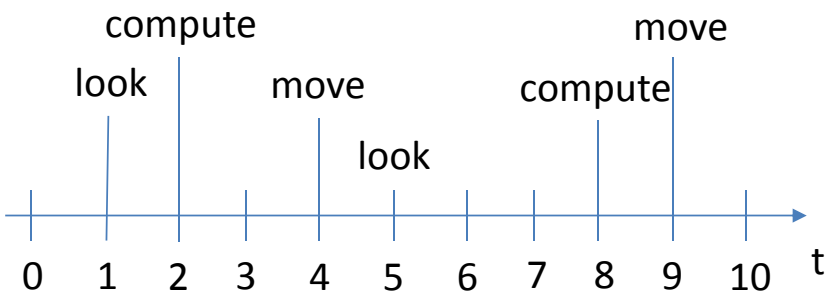


記述したスケジュールの正規化

スケジュール1と2の全てのLOOKに対して,任意のiに対してi番目のLOOKをした時のロボットの配置が等しければロボットの配置が同じ場合スケジュール1と2を同じものとみなす

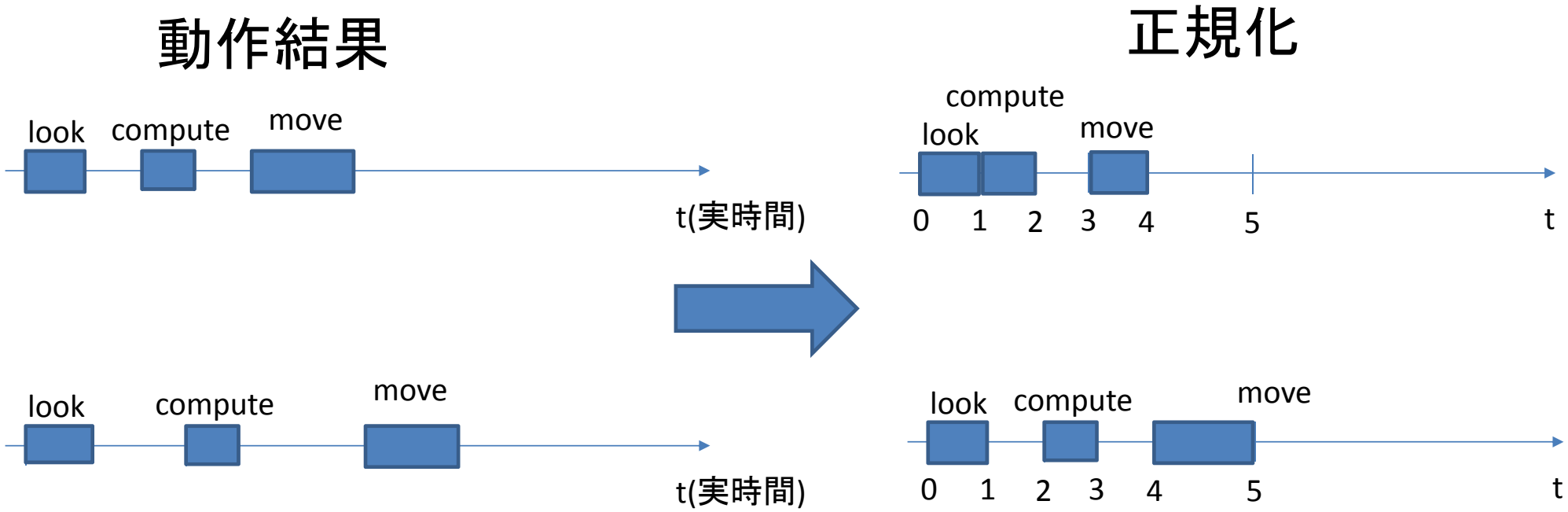


スケジュール1



スケジュール2

動作結果に対するスケジュールへの正規化



スケジュールの実現

ロボットの動作結果からスケジュールを正規化し、記述したスケジュールと正規化されたスケジュールが等価であれば記述したスケジュールが実現されたとする

記述したスケジュール　　動作結果

今後の課題

進捗状況

- ロボットの実装は終わっている
- システムの作成
- スケジュールの等価の定義の設定

セッション８

ロボット２

個体群プロトコルによる1-区間連結性の模倣

片岡　大輝　泉　泰介

近年，インターネットやモバイルのデバイス，スマートフォンなどが普及している中で，移動体通信による分散システムは見時間身近な存在になりつつある．移動体通信のシステムのモデル化における大きな要因の一つは時間とともにネットワークの構造（トポロジ，参加ノード）が変化しうることであるが，近年，そのようなシステムの動的変化をモデル化した分散アルゴリズムの通信モデルが提案されている．特に代表的なモデルの２つは個体群プロトコルモデルと，1-区間連結動的グラフのモデルである．

個体群プロトコルは，n 台のエージェントが互いに通信を行うモデルである．個体群プロトコルにおいては，スケジューラは１対のエージェントを選択し，選択されたエージェント対は互いに情報をやり取りすることで自身の状態を更新する．

動的グラフのモデルにおいて，個々の時刻における通信は，通常のメッセージパッシングシステムと同様であるが，ネットワークのトポロジは時刻とともに変化する．ただし，任意のトポロジ変化を許したモデルを考えた場合，多くの問題は自明に非可解となるため，一般に変化に対して何らかの制約が仮定として課される．１-区間連結動的グラフは各時刻においてシステムのトポロジ全体が連結であることのみを仮定する．またメッセージの送信において，伝えるメッセージは辺の次数によらず決まるものとする．

これらのモデルはいずれも近年精力的に研究されており，その計算能力に関して多くの結果が独立に得られている．一方で，計算機の能力を等しくした場合における，モデル間の能力差についてはあまり検討されていなかった．そこで，本研究では，その検討の一つとして，確率的スケジューラのもとでの個体群プロトコルモデルによる１区間連結性のシミュレートを行うことを検討する．個体群プロトコルモデルは，動的グラフの文脈において，各時間においてシステム中のノード１対のみが互いに通信可能なモデルであるとみなすことができる．

本研究では，1-区間連結動的グラフのモデルの上のｒラウンドの実行を，確率的スケジューラのもとでの個体群プロトコルモデル上で，$O(r^2 nlogn)$ ステップでシミュレート可能ということを示した．$O(r^2 nlogn)$ より少ないステップで解けるのかが今後の課題である．

正六角形グリッド上でのファットロボットの集合問題について

法政大学大学院　白川遥平
法政大学　和田幸一
名古屋工業大学　片山喜章

研究背景

- 自律分散ロボット群

自律的に動作する複数のロボット各々が協調的に動作することにより全体でひとつの問題を達成させる

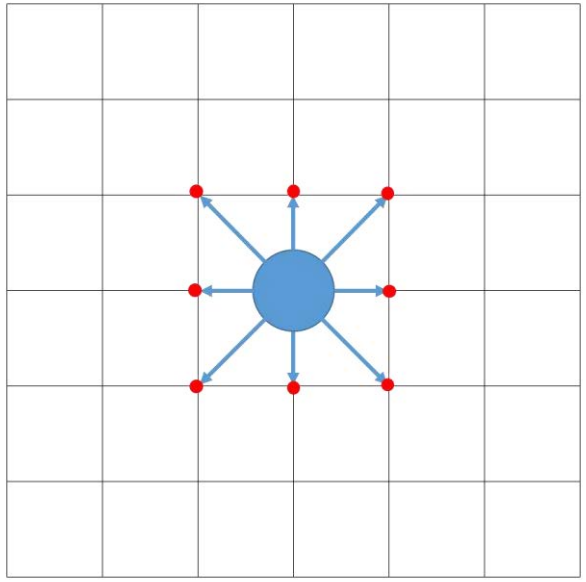
⇒集合問題

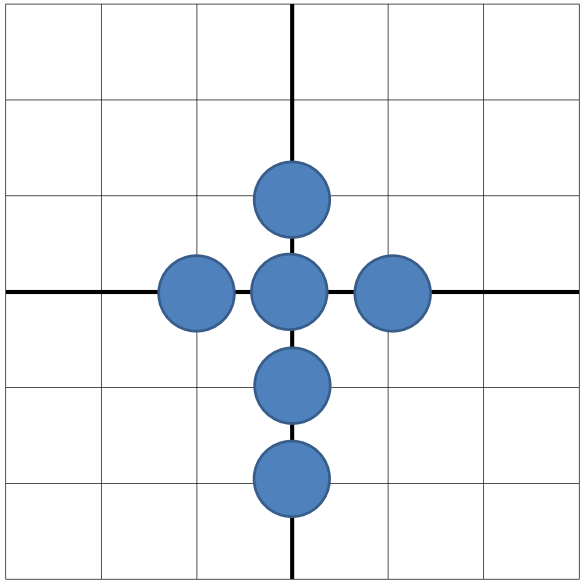
先行研究

格子平面上における集合問題

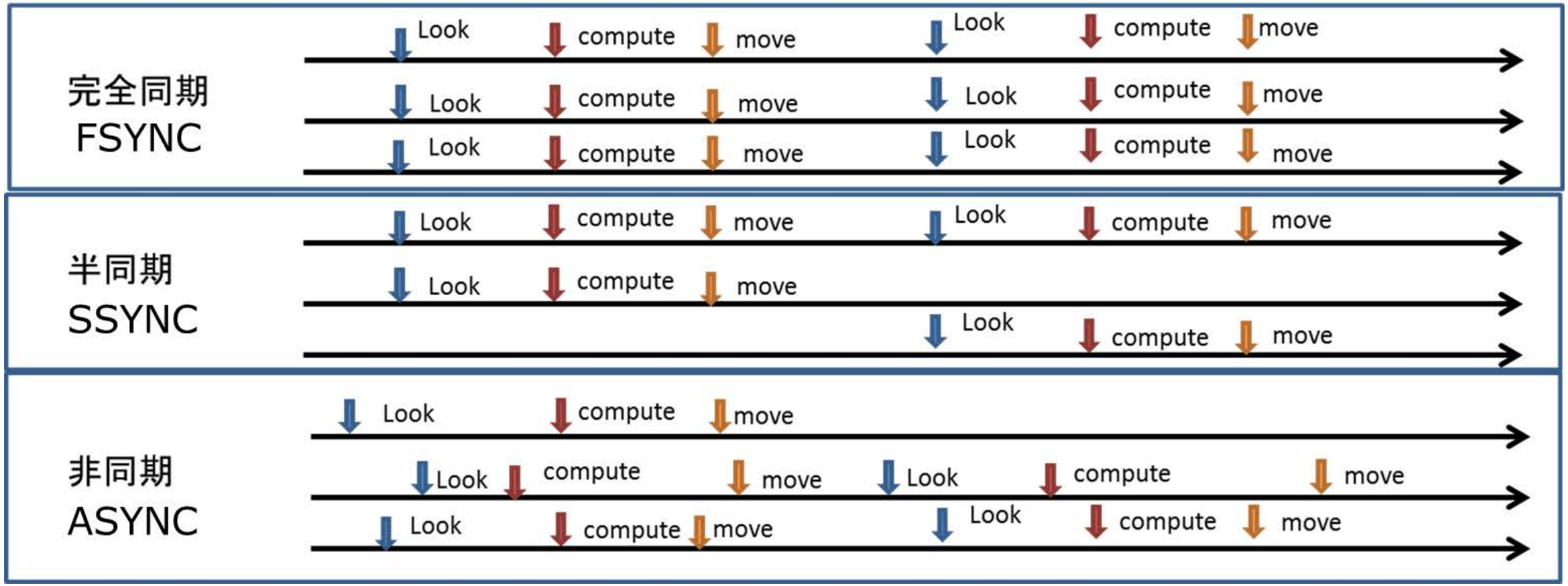
ロボットを平面上の点ではなく円盤で表すファットロボット

移動可能な位置と視野範囲

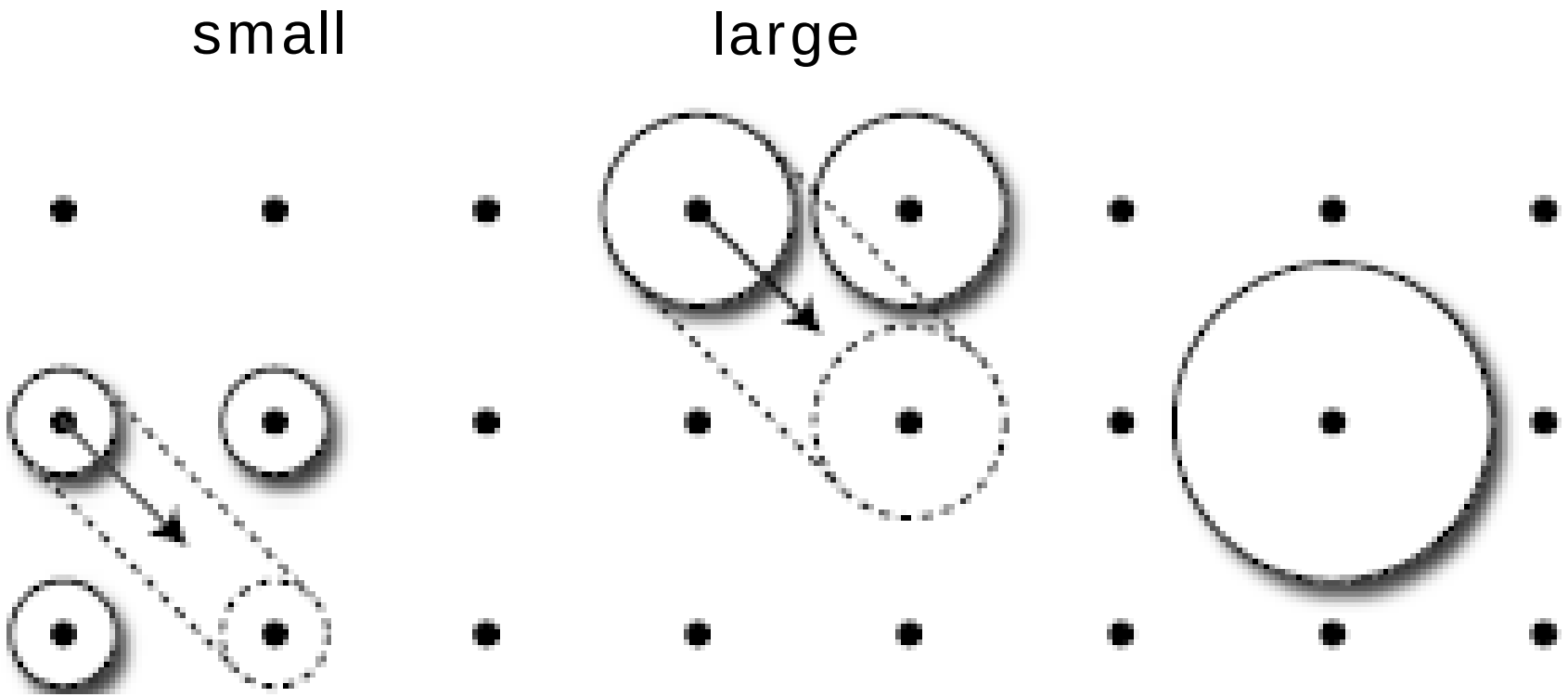




同期



ロボットの大きさ



ロボットのモデル

- 集合するロボットの台数を知るかどうか

集合を達成した時の一番遠いロボットまでの距離Lmaxがわかる

- パラメータ

sync∈{async,ssync,fsync}

num∈{known,unknown}

size∈{small,large}

- 集合問題を解くアルゴリズムのクラスをGA(sync,num,size)で定義する

座標系に対する知識

- 共通の座標系を持つ

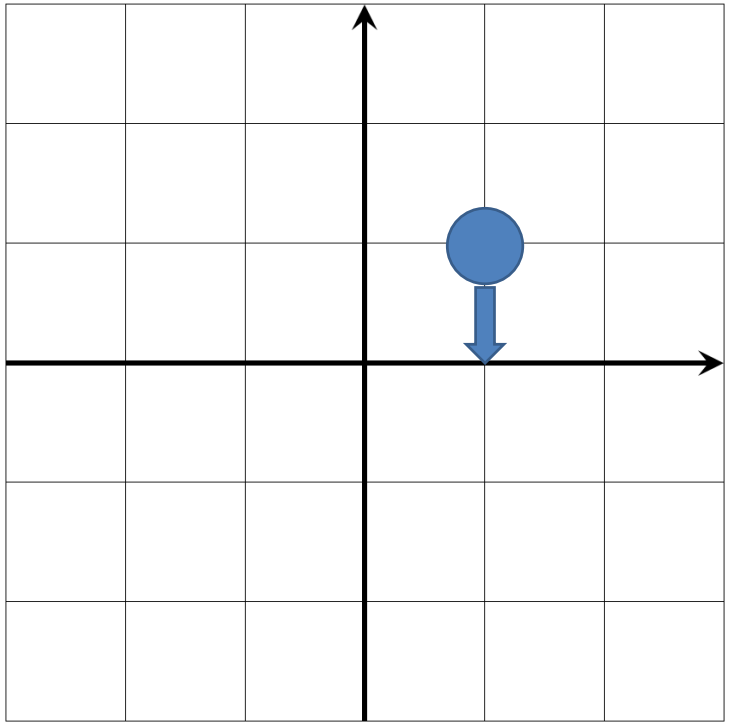
・共通の単位長さ

・共通の原点

・共通の座標軸の方向

・共通の正負の向き

(1,1)から(1,0)への移動



座標系に対する知識

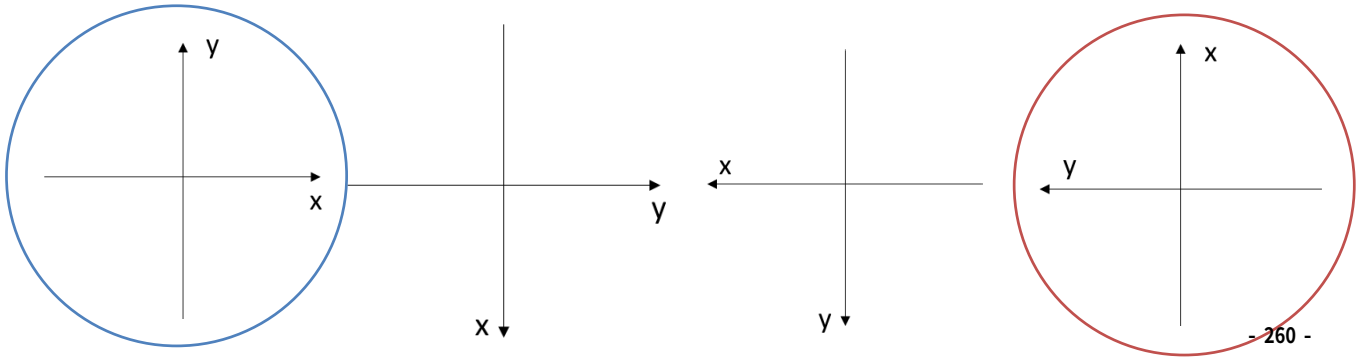
- 共通の座標系を持たない

・共通の単位長さ

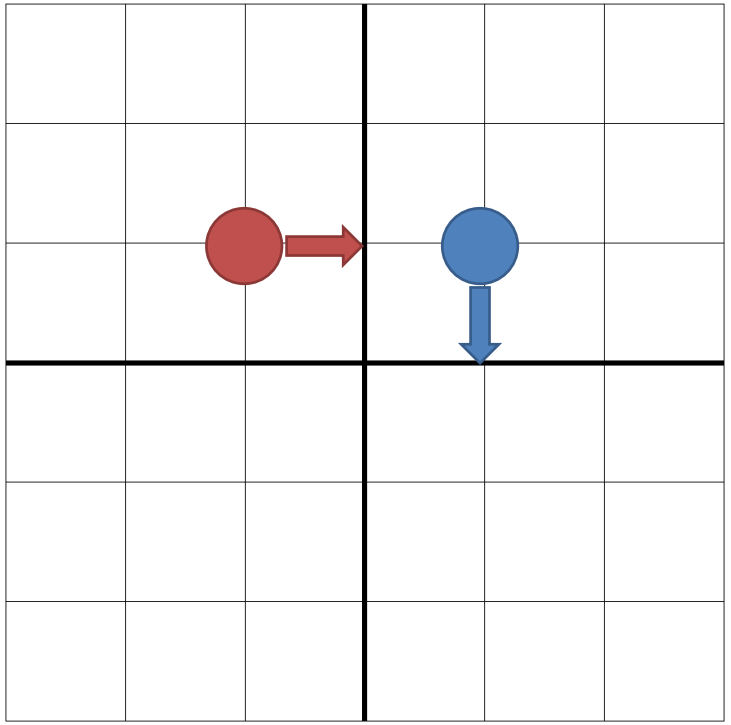
・共通の原点

・共通の座標軸の方向

・右回りに対する合意



(1,1)から(1,0)への移動



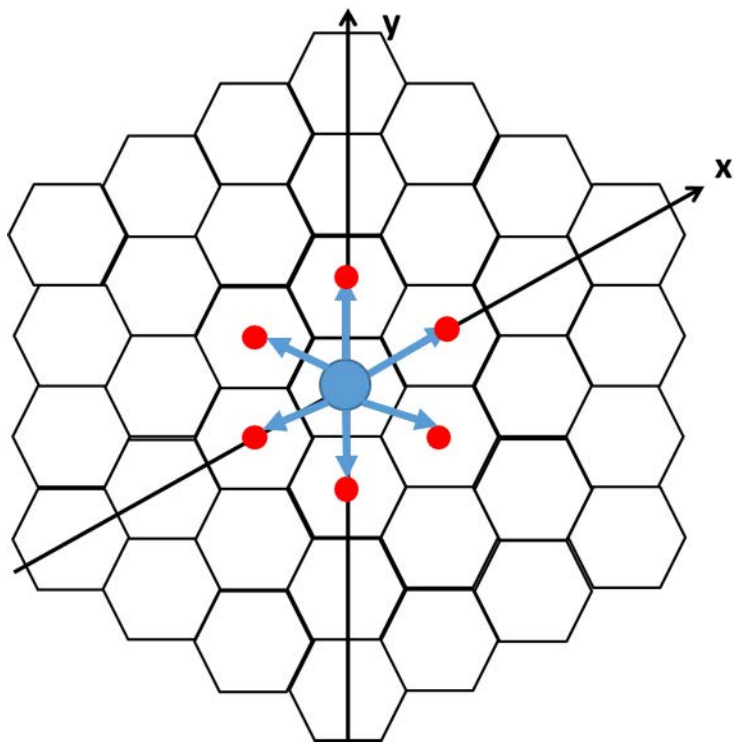
二次元直交座標系に対する結果

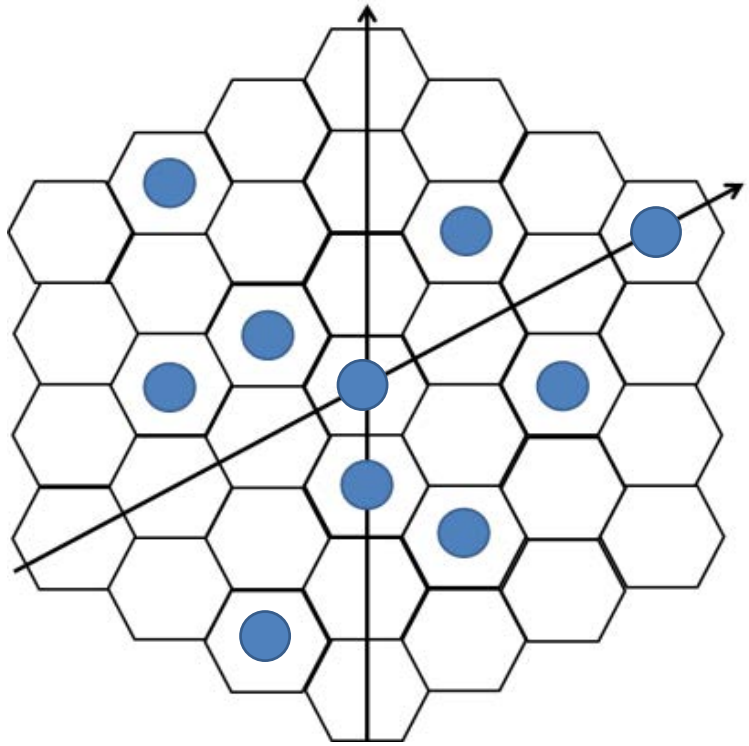
二次元直交座標 共通座標系有り		small	large
async	unknown	Spiral	非可解
	known	Spiral	SpiralSpread
ssync	unknown	Spiral	PullSlide
	unknown	Spiral	SpiralSpread PullSlide

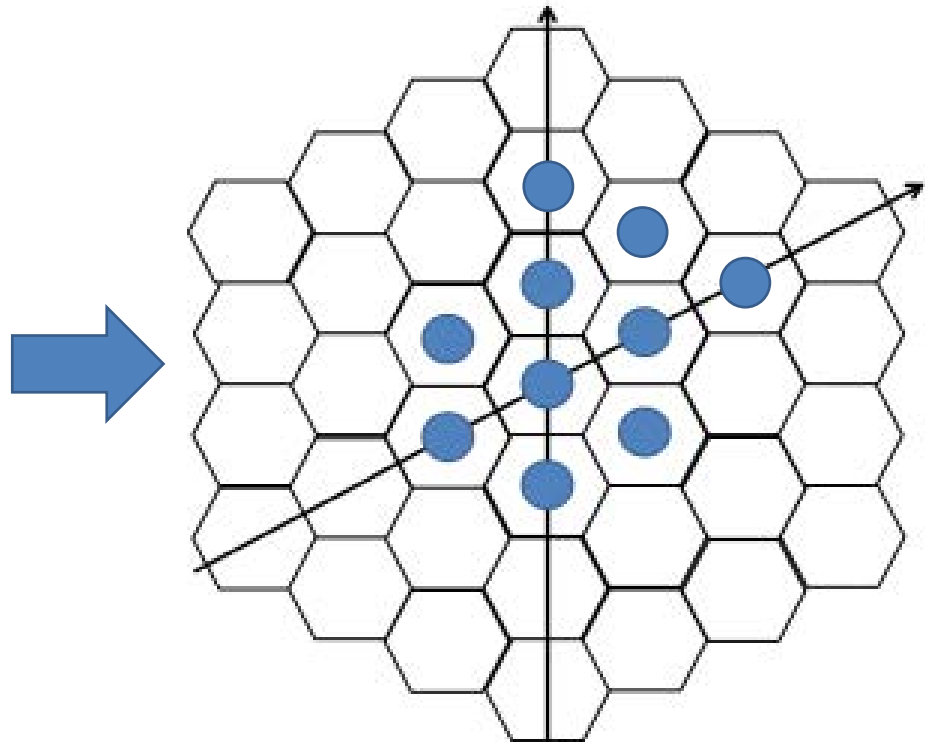
二次元直交座標 共通座標系なし		small	large
async	unknown	未解決	非可解
	known	未解決	未解決
async	unknown	未解決	未解決
	known	GridPullSpin2	GridPullSpin3

正六角形グリッド上の集合問題

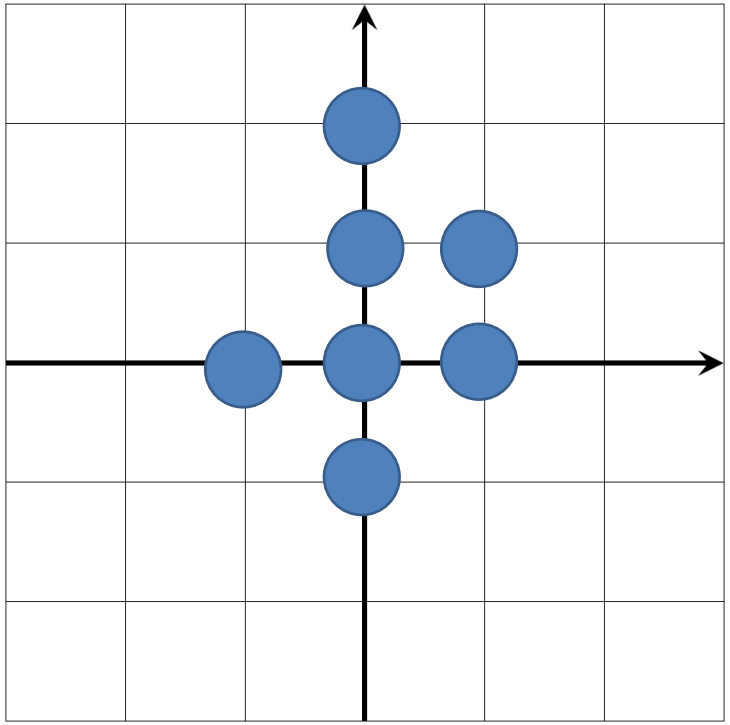
移動可能な位置と視野範囲

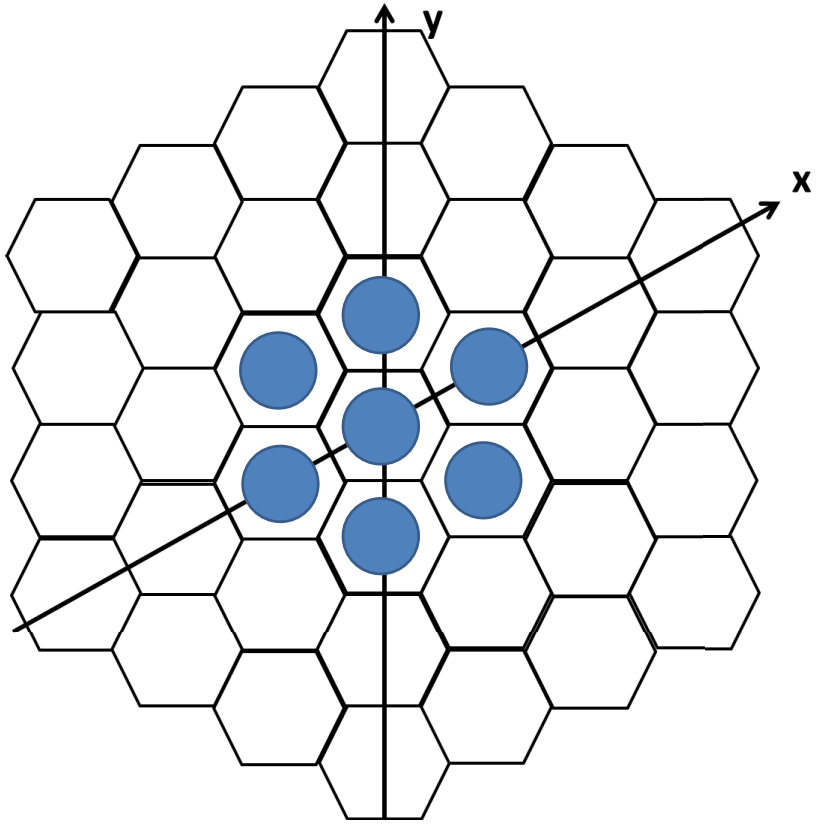






ロボット7台での集合



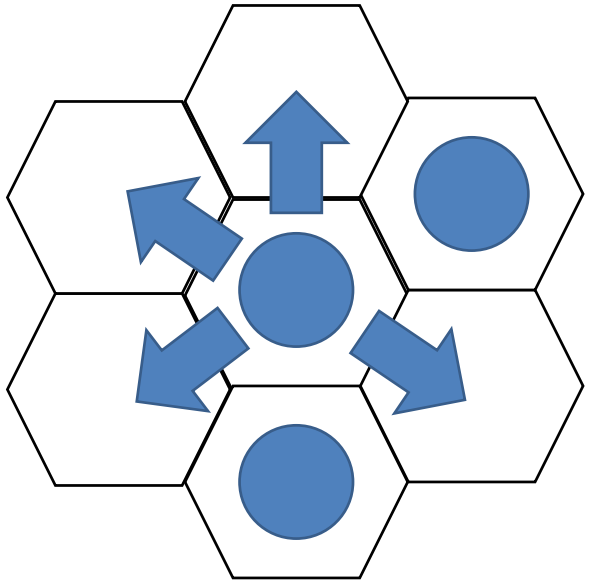


六角形グリッド上では二次元直交座標系に比べより密に集合できる

ロボットのモデル

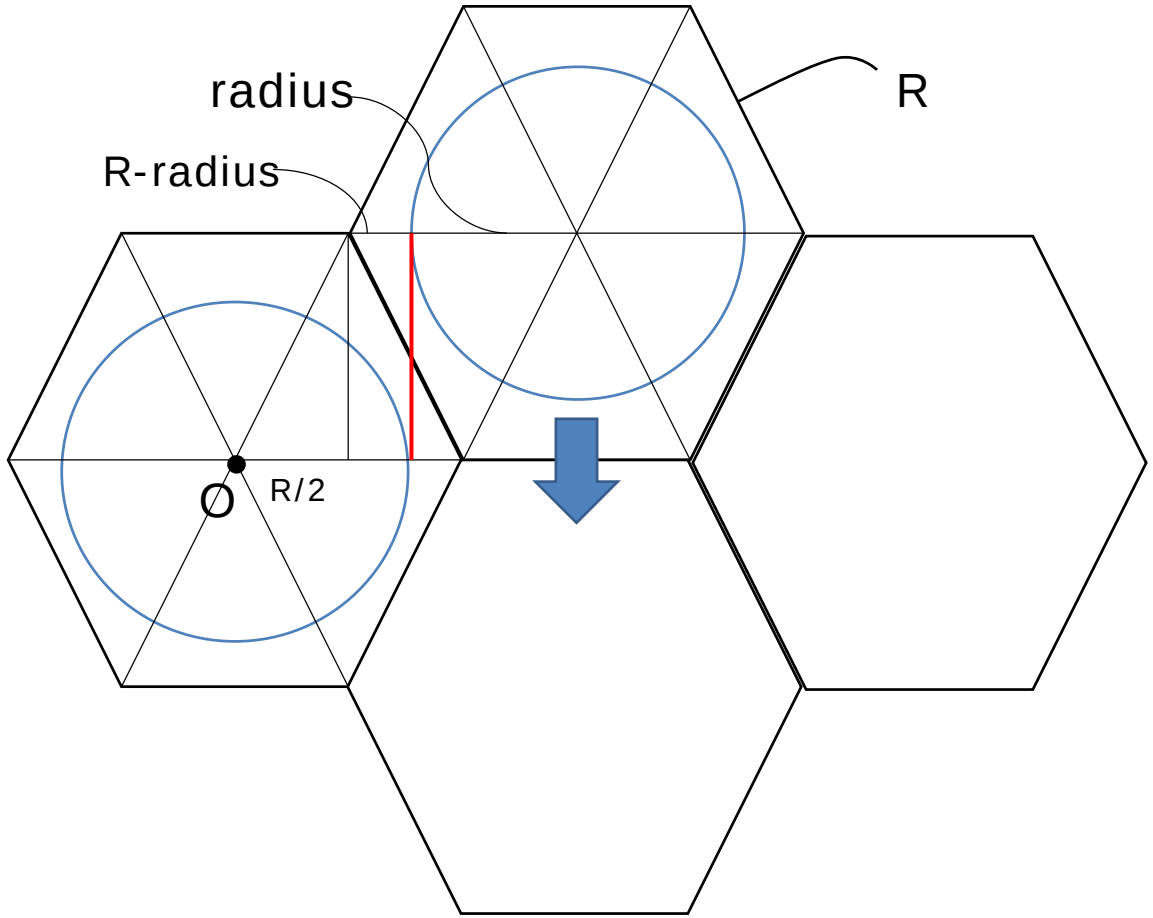
ロボットの半径radiusによって2つのモデルを定義する

1.自由に動ける場合(モデルsmall)



モデルsmallでは,ロボットの大きさが十分に小さくロボットはロボットが存在しない点に自由に移動できる

自由に動ける場合の条件



・正六角形の一辺の大きさをR

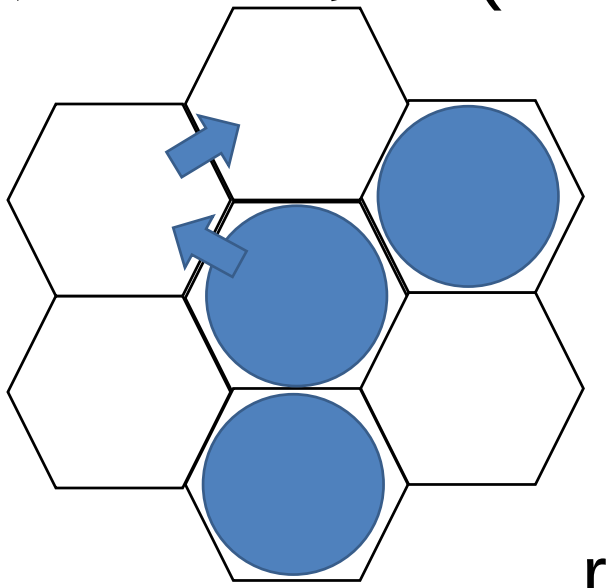
・ロボットの半径の大きさをradius

radius < R/2+R-radius

Small:radius < 3R/4

ロボットのサイズ

2.自由に動けない場合(モデルlarge)



モデルlargeでは,あるロボットは他のロボットが存在する周囲の点へは他のロボットに接触するため移動することができない。

radius < √3R/2

laege: 3R/4 ≦ radius < √3R/2

正六角形グリッドにおける座標系に対する知識

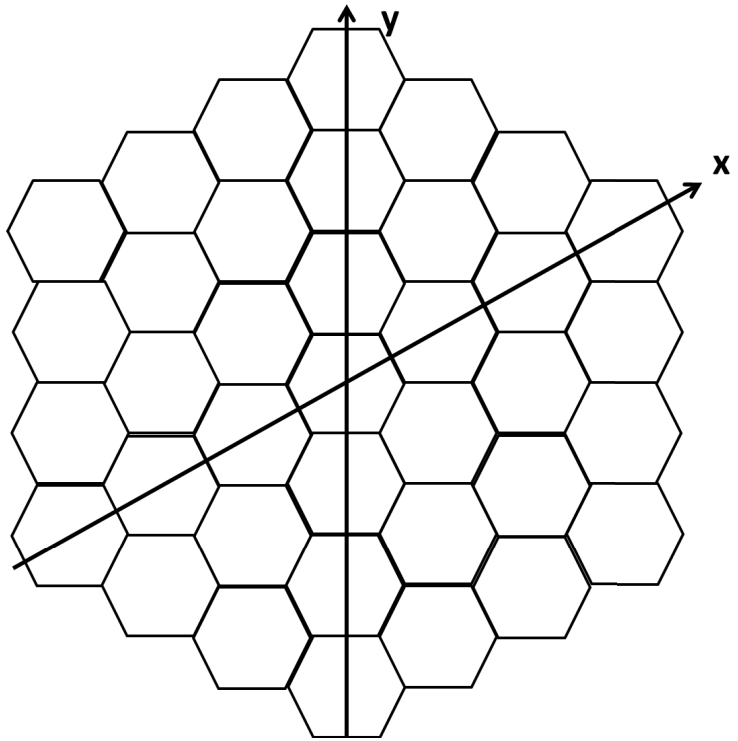
- 共通の座標系を持つ

・共通の単位長さ

・共通の原点

・共通の座標軸の方向

・共通の正負の向き



正六角形グリッドにおける座標系に対する知識

- 共通の座標系を持たない

・共通の単位長さ

・共通の原点

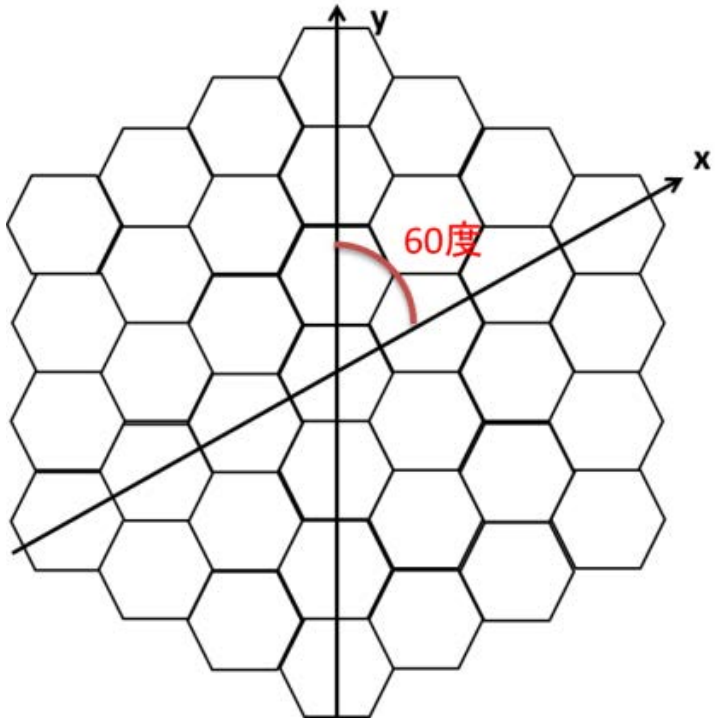
・時計回りの回転を正として

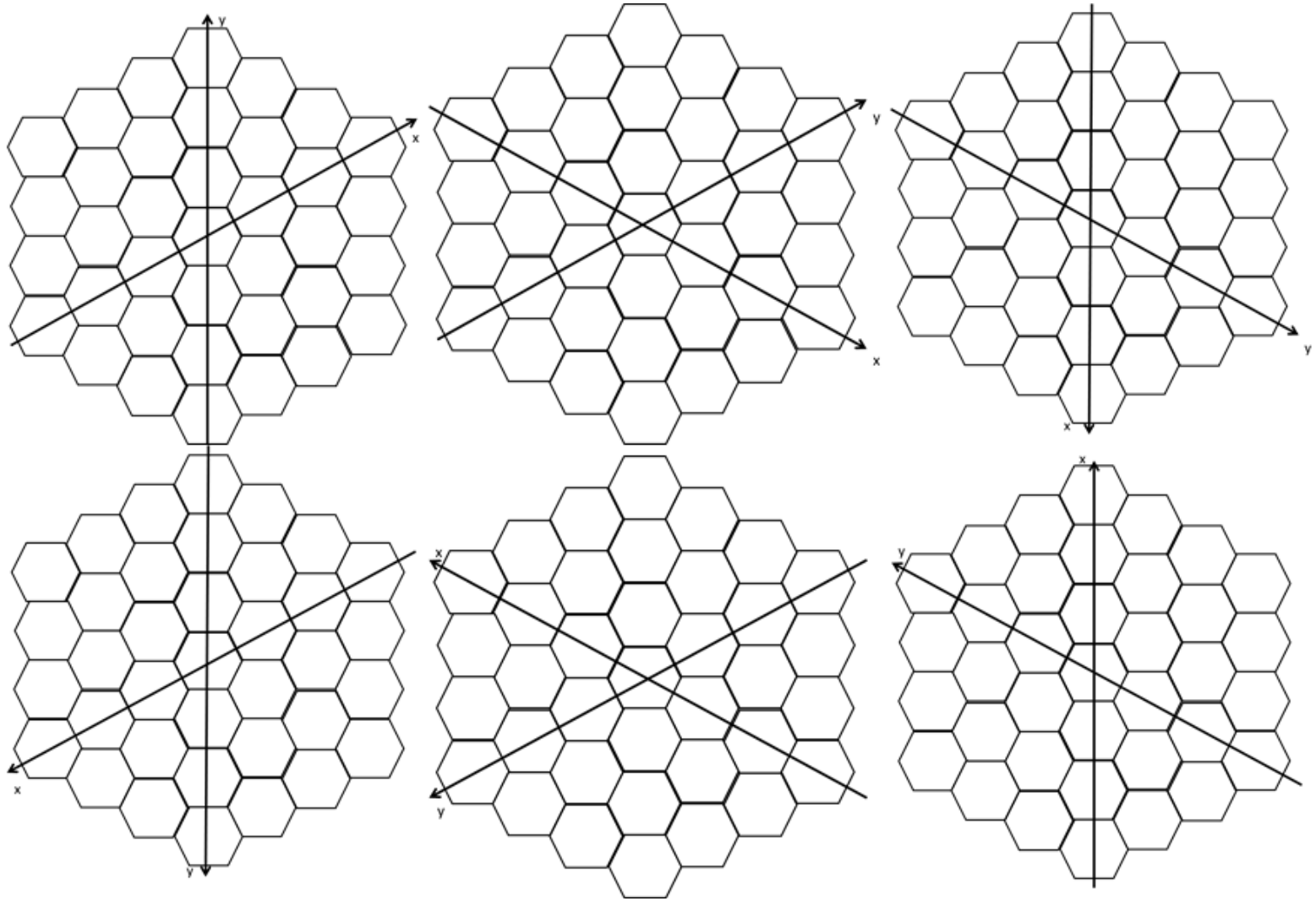
y→x

+60度

そのときの

x軸,y軸それぞれの方向を固定





六角形グリッドにおける共通の座標系なしの場合の総パターン

正六角形グリッドに対する結果

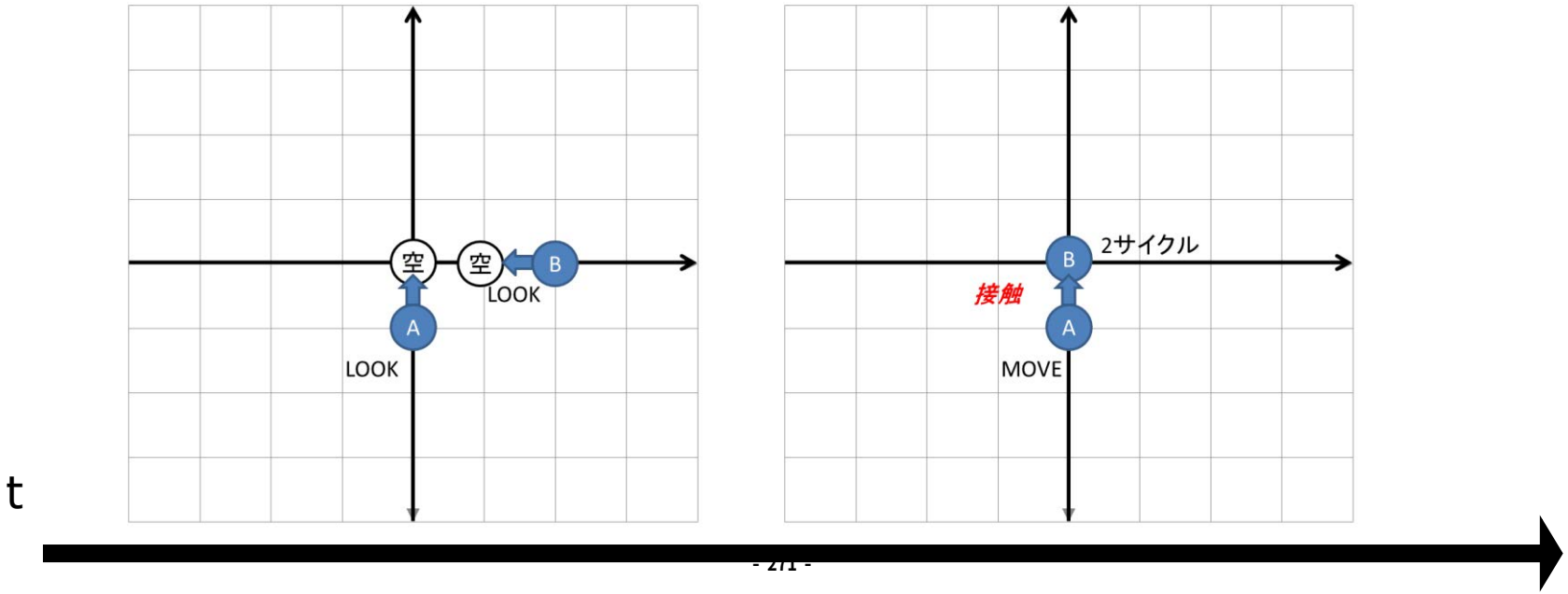
正六角形グリッド 共通の座標系有り		small	large
async	unknown	hexagonSpiral	非可解
	known	hexagonSpiral	視野範囲1では非可解
ssync	unknown	hexagonSpiral	視野範囲1では非可解
	known	hexagonSpiral	視野範囲1では非可解

正六角形グリッド 共通の座標系なし		small	large
async	unknown	未解決	非可解
	known	未解決	視野範囲1では非可解
ssync	unknwon	未解決	視野範囲1では非可解
	known	hexagonPullSpin	視野範囲1では非可解

非同期での問題点

Spiral∈GA(async,unknown,small)

非同期で2つ以上の点から同一の点への移動があると･･･

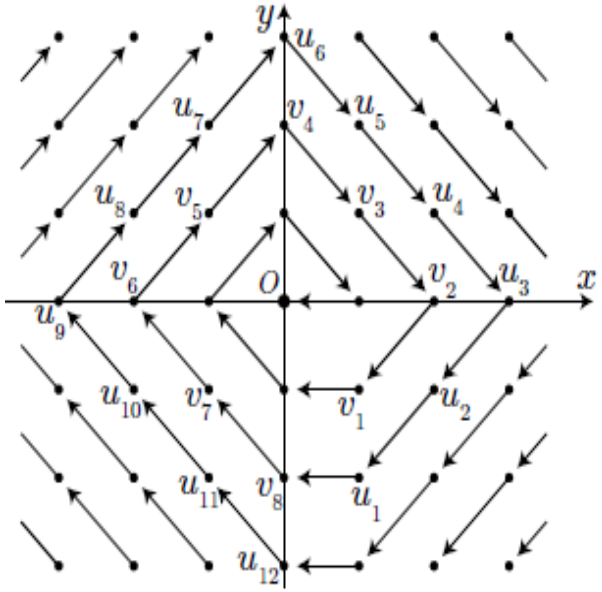


二次元直交座標での解決策

Spiral∈GA(async,unknown,small)

解決策

先行研究では2次元直交座標系上でロボットの通りを一本道にし、移動先にロボットがいないときのみ移動させることで問題を解決した



アルゴリズム 1 点 $v = (v_x, v_y)$ 上のロボット r 上のアルゴリズム *Spiral*

1: **Predicate**:
2: $Spin(r) \equiv (v \neq 1 \vee y > 0) \wedge (e_v \notin C)$
3: $Upper(r) \equiv (v = 1) \wedge (y \leq 0) \wedge ((v_x - 1, v_y) \notin C)$
4: $Stay(r) \equiv (v = O) \vee (\neg Spin(r) \wedge \neg Upper(r))$
5: **Actions**:
6: Spin :: $Spin(r)$ → 行き先を e_v に
7: Upper :: $Upper(r)$ → 行き先を $(v_x - 1, v_y)$ に
8: Stay :: $Stay(r)$ → 移動しない

アルゴリズム Spiral

正六角形グリッド 共通の座標系有り		small	large
async	unknown	**Spiral**	
	known	Spiral	
ssync	unknown	Spiral	
	known	Spiral	

汎用アルゴリズム

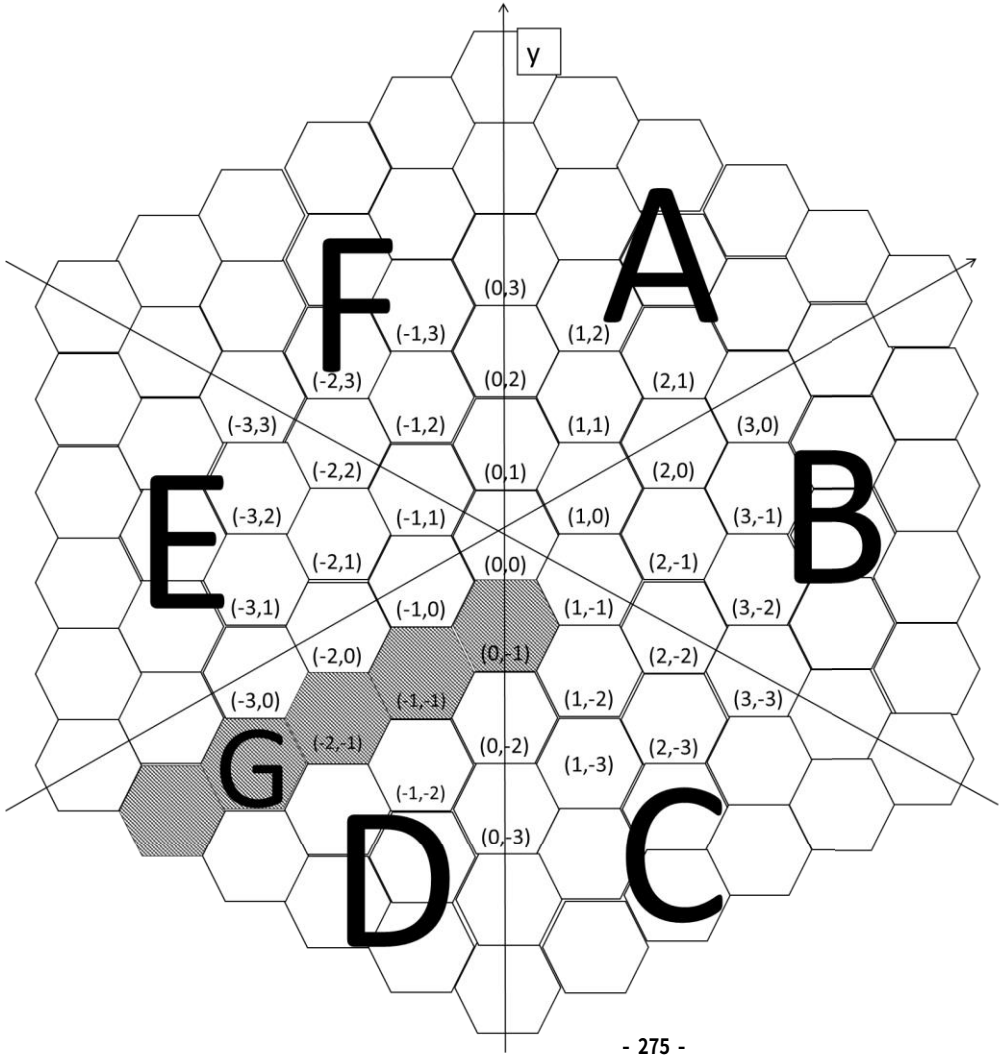
同様の方法で二次元直交座標以外でも原点が存在するし一般的な二次元平面上といった条件の元でロボットのモデルが(async,unknown,small)の時,集合を達成する汎用アルゴリズムを設計した

汎用アルゴリズム　GeneralAlgorithm

1: Predicate
2:Spin(r)≡特定の点に向かうための条件
3:Upper(r)≡距離を縮めるための条件
4: stay(r)≡ロボットが動かないための条件
5: Action
6: Spin::Spin(r)→特定の点に向かって右回りに移動
7: Upper::Upper(r)→距離を1縮める移動
8: stay::stay(r)→移動しない

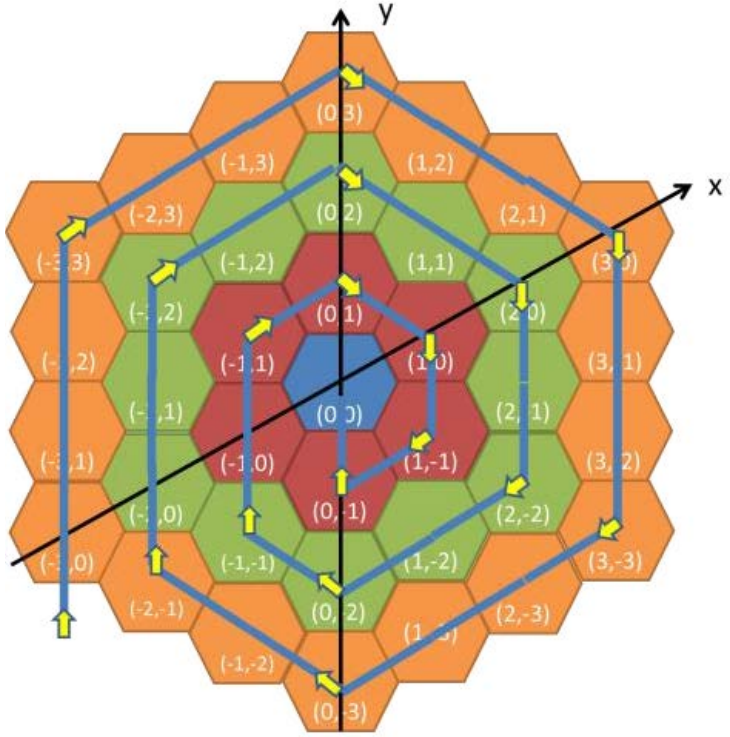
必要な要素
・共通の原点
・右周りに対する合意
・点を区別することができ
・距離を計算できる

右回りの移動の実現


y
A
B
C
D
E
F
G
(0,3)
(1,2)
(2,1)
(3,0)
(-1,3)
(-2,3)
(0,2)
(1,1)
(2,0)
(3,-1)
(-1,2)
(-3,3)
(0,1)
(1,0)
(2,-1)
(3,-2)
(-1,1)
(-2,2)
(-3,2)
(0,0)
(1,-1)
(2,-2)
(3,-3)
(-1,0)
(-2,1)
(-3,1)
(0,-1)
(1,-2)
(2,-3)
(-1,-1)
(-2,0)
(-3,0)
(0,-2)
(1,-3)
(-1,-2)
(-2,-1)
(0,-3)

六角形グリッドのアルゴリズム

アルゴリズムとその移動パターン



hexagonspiral
1: Predicate
2: up(r) ≡ [(E∨G) ∧{(vy+1) ∉C}
3:rightup(r) ≡ F∧{(vx+1) ∉C}
4: rightdown(r) ≡ A∧{(vx+1,vy-1) ∉C}
5: down(r) ≡ B∧{(vy-1) ∉C}
6: leftdown(r) ≡ C∧{(vx-1) ∉C}
7: leftup(r) ≡ D∧{(vx-1,vy+1) ∉C
8: stay(r) ≡ (v=0)∨
(¬up(r)∧¬rightup(r)∧¬rightdown(r)∧¬down(r)∧¬leftdown(r)∧¬leftup(r))
9: Action
10: up :: up(r) → 行き先を(vx,vy+1)に
11: rightup :: rightup(r) → 行き先を(vx+1,vy)に
12: rightdown :: rightdown(r) → 行き先を(vx+1,vy-1)に
13: down :: down(r) → 行き先を(vx,vy-1)に
14: leftdown :: leftdown(r) → 行き先を(vx-1,vy)に
15: leftup :: leftup(r) → 行き先を(vx-1,vy+1)に
16: stay :: stay(r) → 移動しない

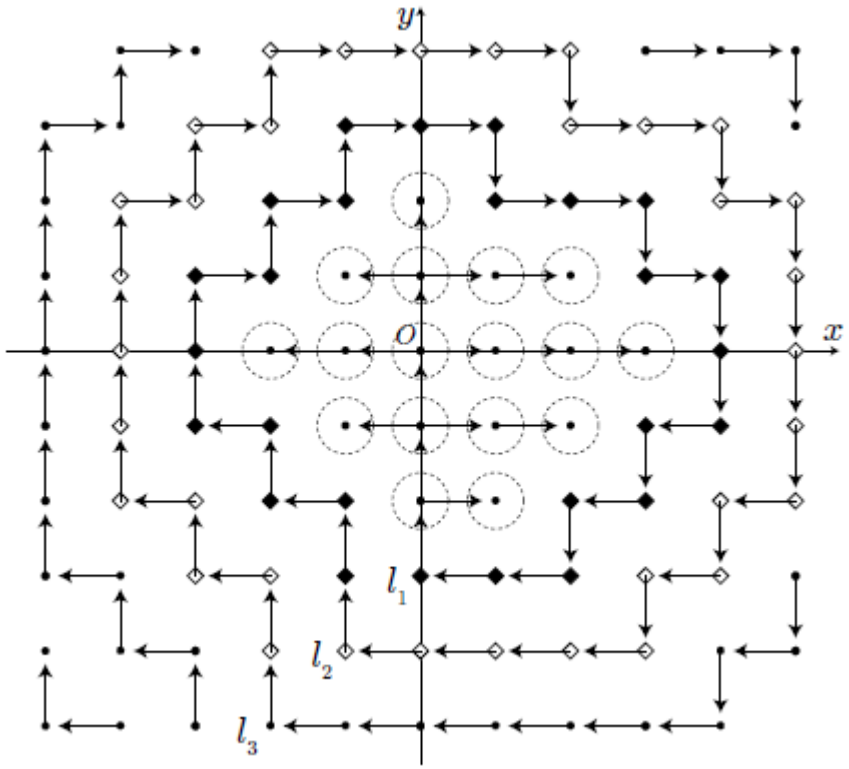
定理
hexagonSpiral∈GA(async,unknown,small)

アルゴリズム hexagonSpiral

正六角形グリッド 共通の座標系有り		small	large
async	unknown	**hexagonSpiral**	
	known	hexagonSpiral	
sync	unknown	hexagonSpiral	
	known	hexagonSpiral	

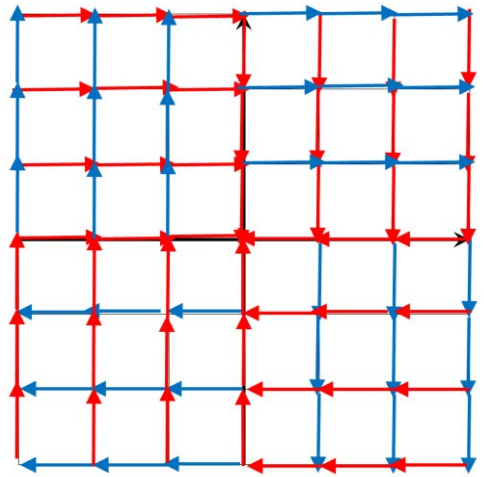
二次元直交座標系において ロボットサイズLargeの時

SpiralSpread∈GA(async,known,large)



集合ポイントの周囲で一方通行を行う

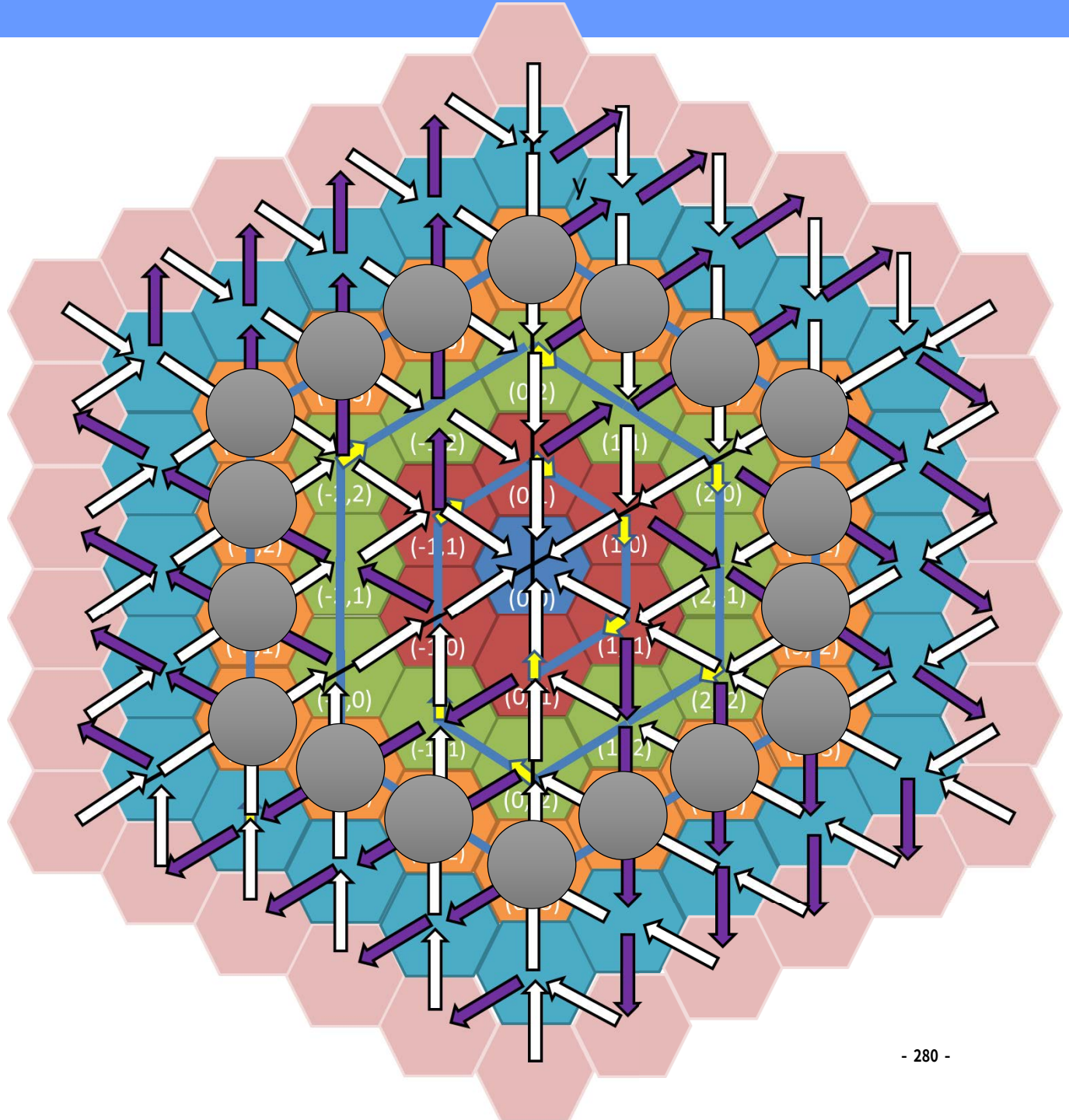
PullSlide∈GA(ssync,unknown,large)



距離を縮める移動Pull
距離を一度離し次のPullで先の右回りの点に移動できる位置に移動するSlide

ロボットサイズLargeに対する結果

二次元直交座標 共通座標系有り		small	large
async	unknown		
	known		**SpiralSpread**
ssync	unknown		**PullSlide**
	unknown		SpiralSpread PullSlide

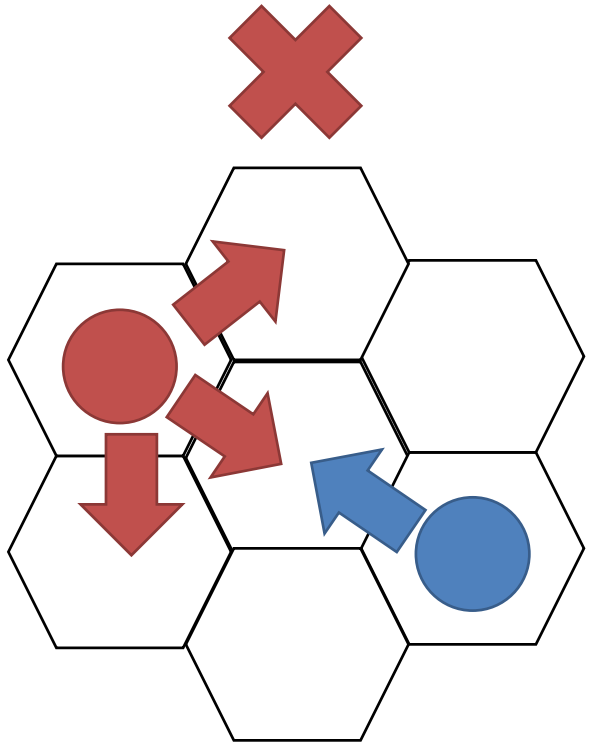


先の移動パターンを正六角形グリッド上で適用しようとすると・・・

同様のパターンのアルゴリズムを作るだけでは集合問題を達成できない

※
GA(async,known,large)
GA(ssync,unknown,large)
が存在しないとはかぎらない

ロボットの視野範囲1で集合が達成できないことの証明

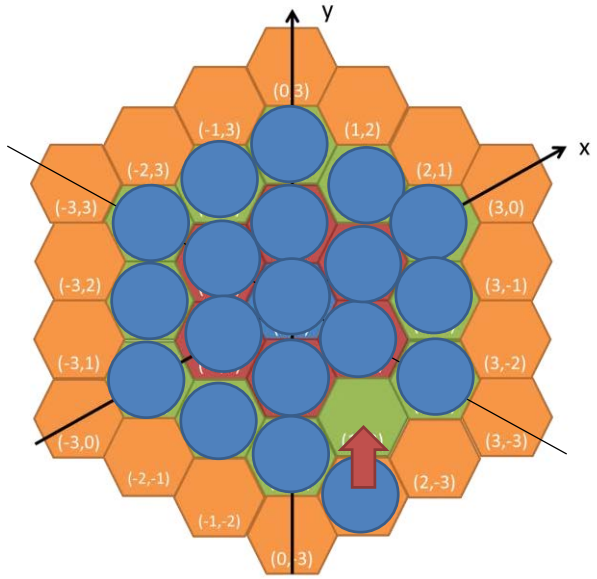


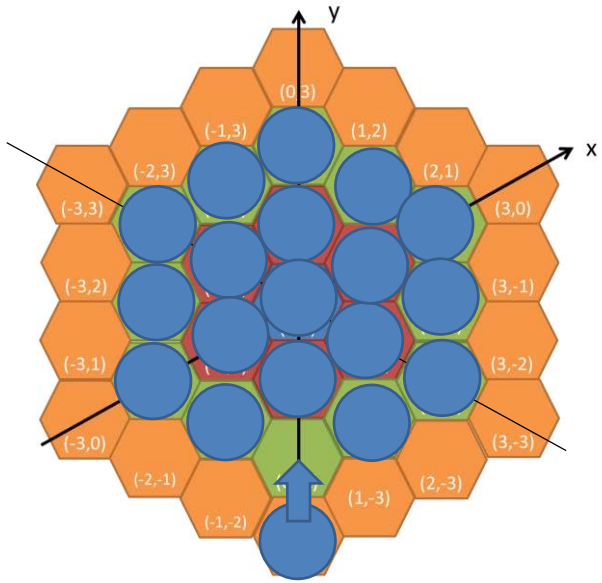
(ssync,known,large)の場合

直線状の距離が2離れた位置において片方のロボットが距離を近づける移動をする可能性がある場合

もう片方のロボットは距離を縮めるまたは同一距離への移動は衝突回避することができない

ロボットの視野範囲1で集合が達成できないことの証明

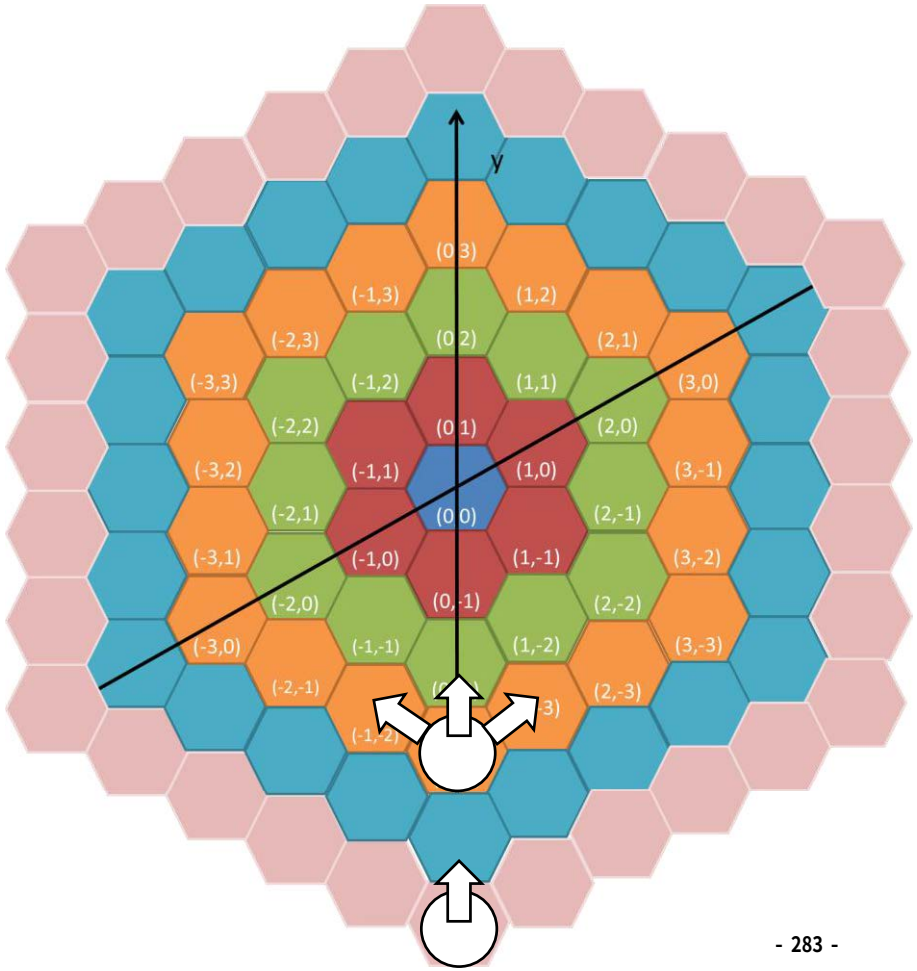




ロボットサイズLargeの時,軸上または|vx|=|vy|となる点以外の点1つを除いてLmaxとLmax-1における全ての点が集合を達成しているとする時,その点にロボットが入ることができず集合が達成できない

よって各距離において必ず1つは軸上において距離を縮める必要がある

ロボットの視野範囲1で集合が達成できないことの証明

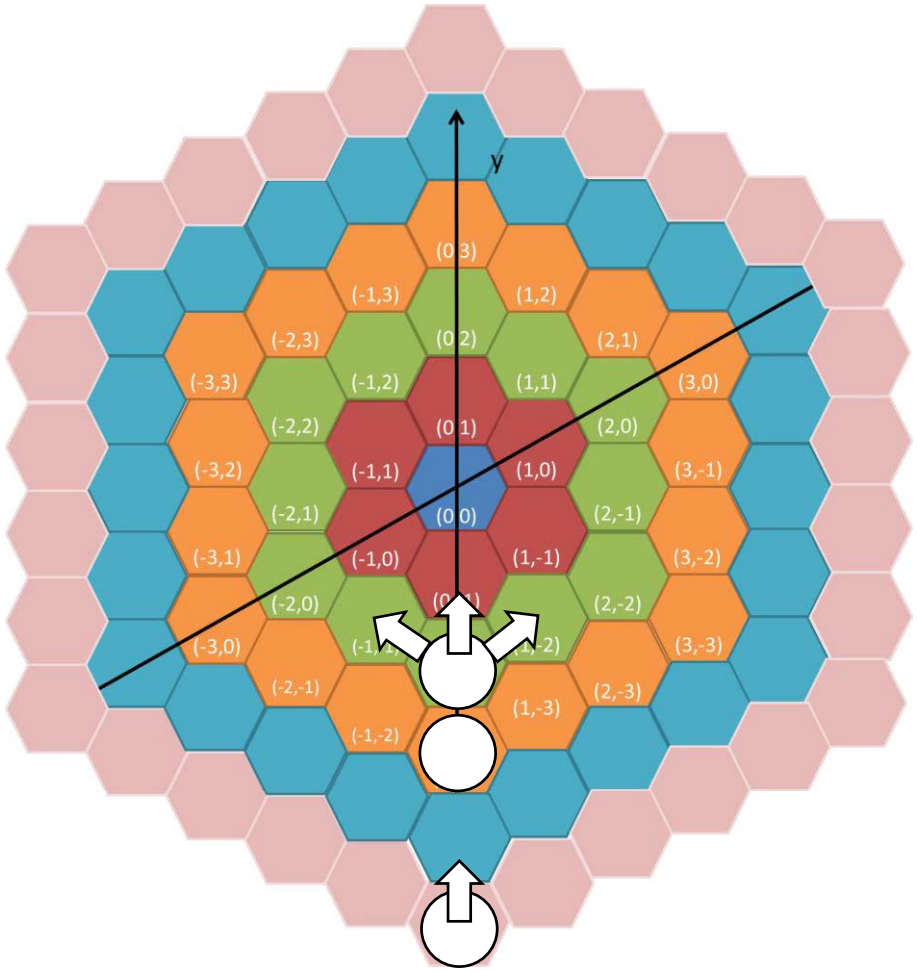


軸上または|vx|=|vy|となる点において距離を近づける移動をする点があると

そこから直線状で原点への距離が2近づいた点において前方3方向への移動しかできない

集合を達成するためにはこのロボットが移動する必要がある

ロボットの視野範囲1で集合が達成できないことの証明

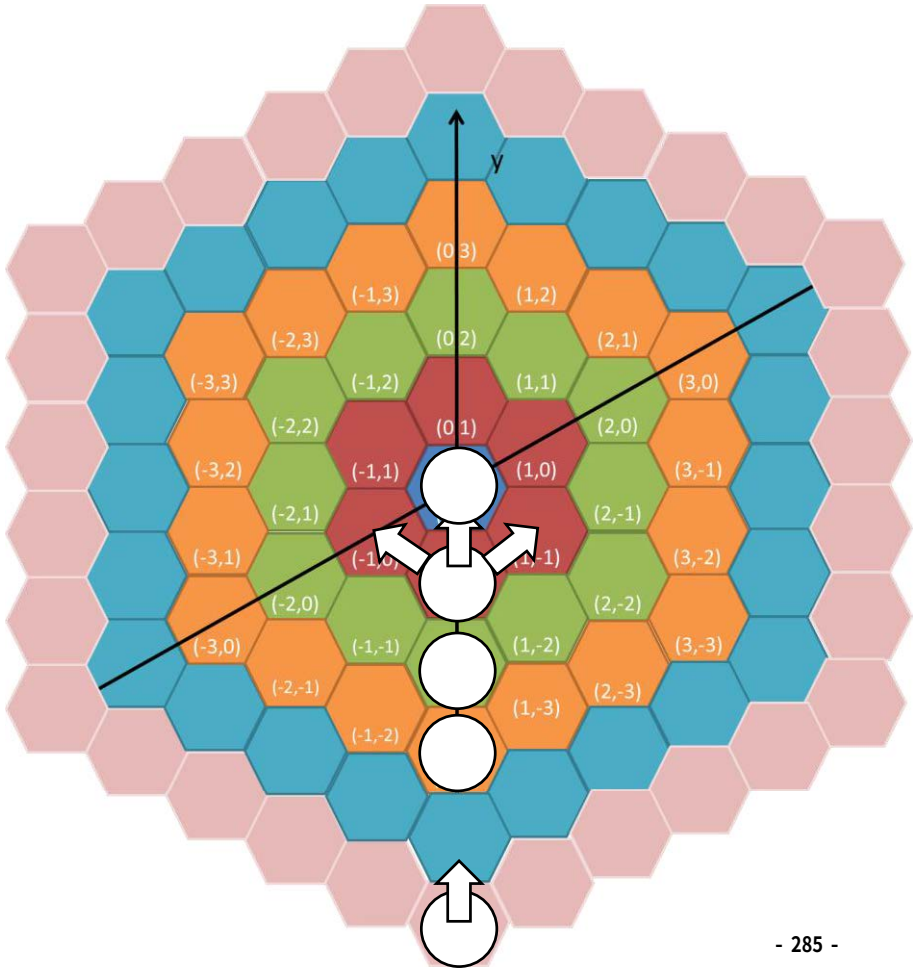


どの方向へ移動する場合でも先ほどの点から原点への距離が1近づいた位置においてロボットがいてはならない

その位置のロボットは前方3方向のいずれかに移動する必要がある

またこのロボットが移動するためにも原点への距離が1近づいた位置においてロボットが前方3方向のいずれかに移動する必要がある

ロボットの視野範囲1で集合が達成できないことの証明



これが原点まで繰り返される

全ての位置にロボットがあった時、直線状のロボットは移動することができないので集合を達成することができない

よって,ロボットの視野範囲1では集合を達成することができない□

ロボットサイズLargeでの非可解性

正六角形グリッド 共通の座標系有り		small	large
async	unknown		
	known		視野範囲1では非可解
ssync	unknown		視野範囲1では非可解
	known		視野範囲1では非可解

二次元直交座標 共通座標系なし		small	large
async	unknown		
	known		
ssync	unknown		
	unknown	GridPullSpin2	

正六角形グリッド 共通座標系なし		small	large
async	unknown		
	known		
async	unknown		
	known	hexgonPullSpin	

正六角形グリッド 共通の座標系有り		small	large
async	unknown	hexagonSpiral	非可解
	known	hexagonSpiral	視野範囲1では非可解
ssync	unknown	hexagonSpiral	視野範囲1では非可解
	unknown	hexagonSpiral	視野範囲1では非可解

正六角形グリッド 共通の座標系なし		small	large
async	unknown	未解決	非可解
	known	未解決	視野範囲1では非可解
ssync	unknown	未解決	視野範囲1では非可解
	unknown	hexagonPullSpin	視野範囲1では非可解

まとめ

- 今後の課題

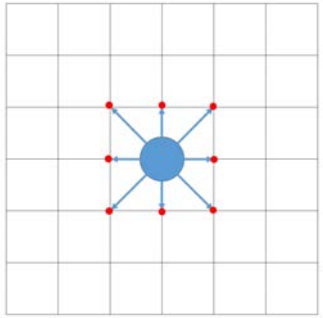
・表における未解決のモデルに対するアルゴリズムを考える

・largeにおける視野範囲2以上での可解性を調べる

まとめ

- 二次元直交座標系と六角形グリッド上でのロボットサイズlargeにおける可解性に違いの理由

二次元直交座標系において距離2となる斜めの位置が視野範囲に含まれている



⇒

・二次元直交座標系において距離1の範囲のみを視野範囲としたとき結果が変わる可能性について検討

非同期モデルのロボットによる、半同期モデルのシミュレートについて

法政大学　奥村太加志
法政大学　和田幸一
名古屋工業大学　片山喜章

研究内容

- 自律型分散ロボットについて、非同期(ASYNC)モデルのロボットで半同期(SSYNC)モデルをシミュレートする方法について考える。

- ASYNCモデルのロボットに、新たに機能を追加することで実現する。
  - 状態の表示

- SSYNCモデルのシミュレートを実現するSIMプロトコルについて、2台の場合では状態数を削減できるかを考える。

ロボットのモデル

- システムはロボットの集合で構成される。
  - それぞれに計算能力、移動能力がある。
  - それぞれは点とみなす。
- センサーによって周囲のロボットを観測可能
- 匿名性
  - 外見によって識別できず、すべて同じプロトコルを実行する。
- 自律性
  - 集中制御はない。
- 他のロボットに情報を伝える直接の通信手段はない。

ロボットのモデル

- ロボットはactive , inactiveの2つの状態がある。

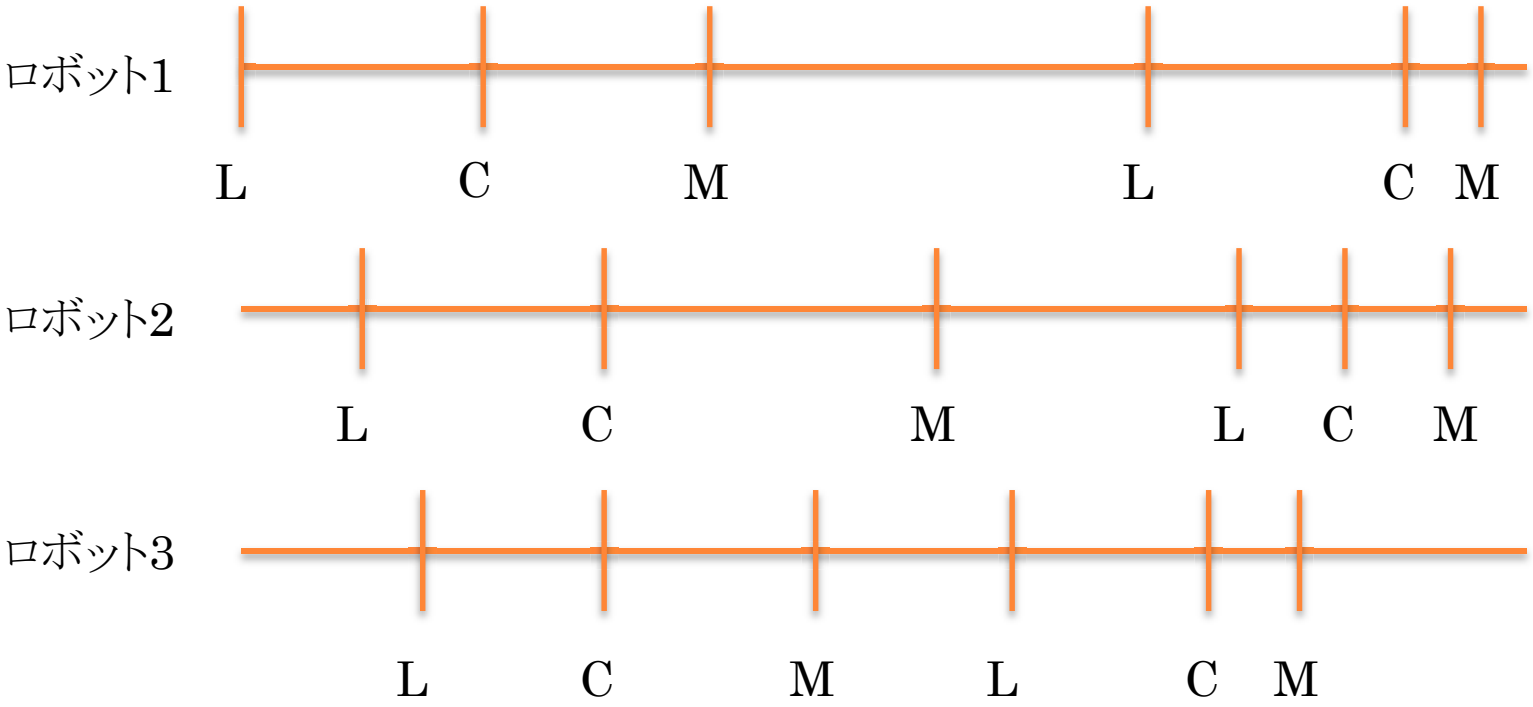
- active
  - Look-Compute-Move(LCM)サイクル実行
    - Look : 周囲の観測
    - Compute : アルゴリズム実行、行き先への計算
    - Move : 目的地へ移動

- inactive
  - 休止状態

スケジュール

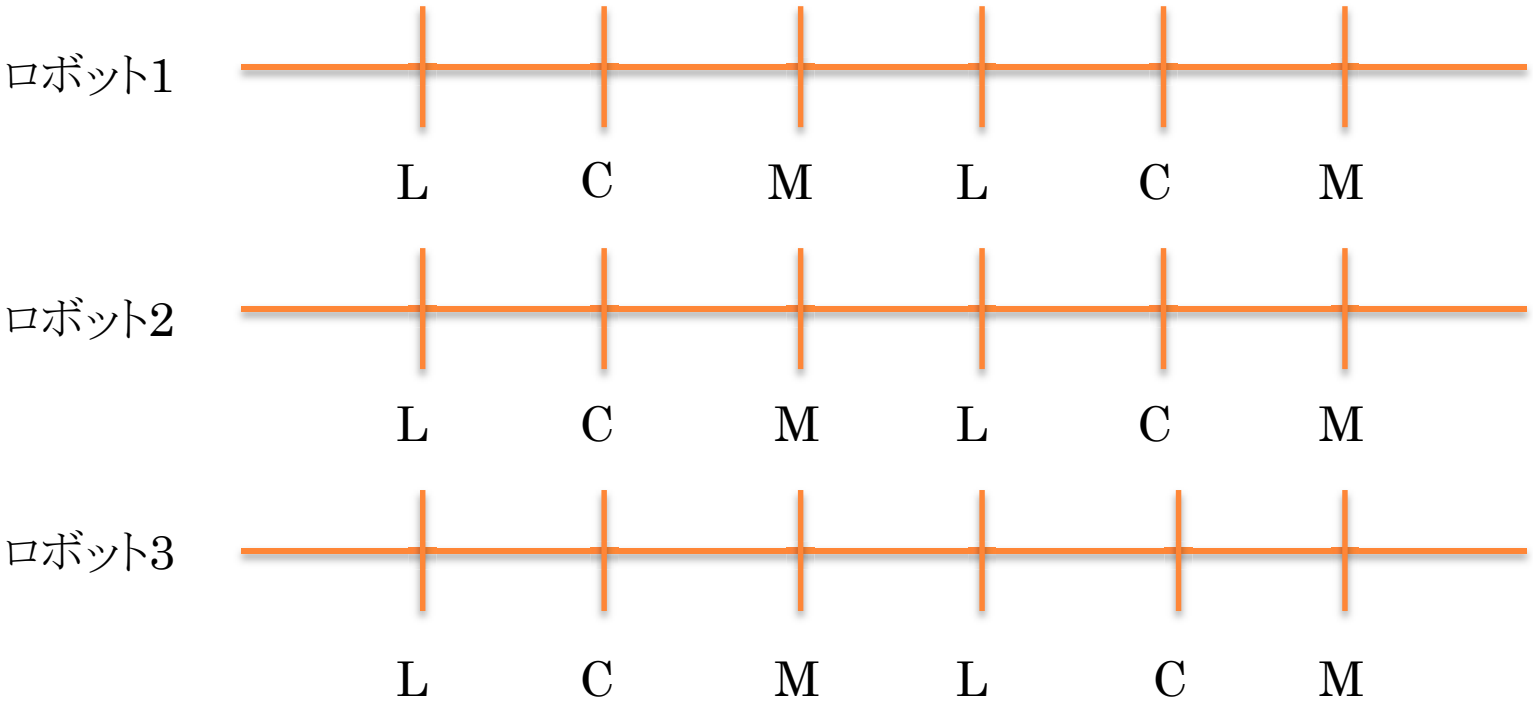
- 非同期(ASYNC)
  - ロボットは独立して行動する。
  - LCMサイクルの行動が共通でない

L : Look
C : Compute
M : Move



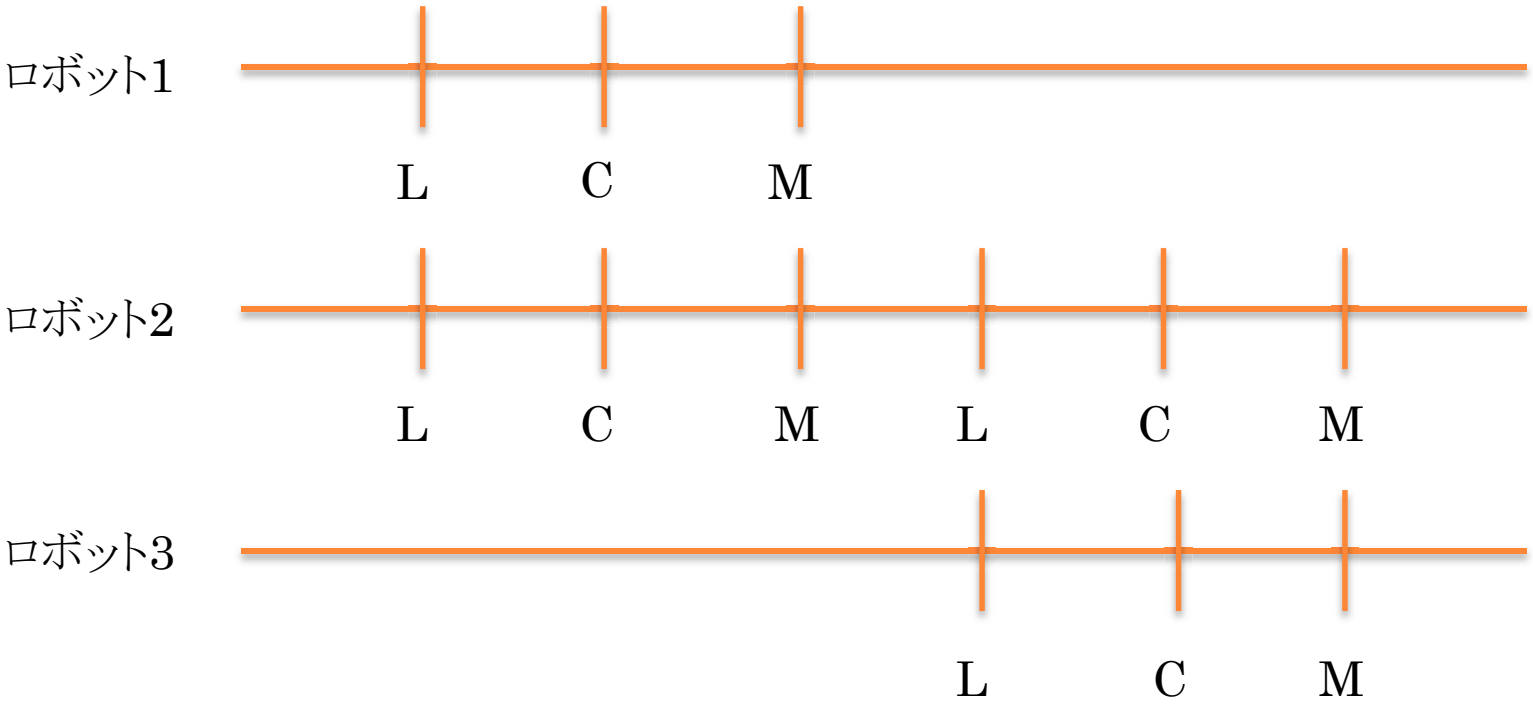
スケジュール

- 全同期(FSYNC)
  - すべてのロボットのLCMサイクルの行動が共通



スケジュール

- 半同期(SSYNC)
  - すべて、または一部のロボットの行動が一致



各モデルの関係性

- FSYNCモデルで解ける問題の集合：$FSYNC$
- SSYNCモデルで解ける問題の集合：$SSYNC$
- ASYNCモデルで解ける問題の集合：$ASYNC$

- 各モデルの計算能力は

$$FSYNC \supset SSYNC \supset ASYNC$$

ASYNCモデルの機能追加

- ASYNCモデルのロボットに、SSYNCモデルをシミュレートできるよう、自身の状態を表示できる機能を追加する。

- ライトによる状態の表示
  - 多色の表示が可能
  - k状態を持つASYNCモデルで解ける問題の集合を$ASYNC^k$と示す。

- n台のASYNCモデルのロボットで解ける問題の集合を$ASYNC(n)$と示す。

従来の結果

- SSYNCモデルでは解けないが、状態を持つASYNCモデルでは解ける問題がある。
  - 例:2台のロボットの集合問題

$$ASYNC(2)^k \setminus SSYNC(2) \neq \emptyset$$

- SSYNCモデル、ASYNCモデルにライトをつけた場合

$$ASYNC(n)^k = SSYNC(n)^k$$

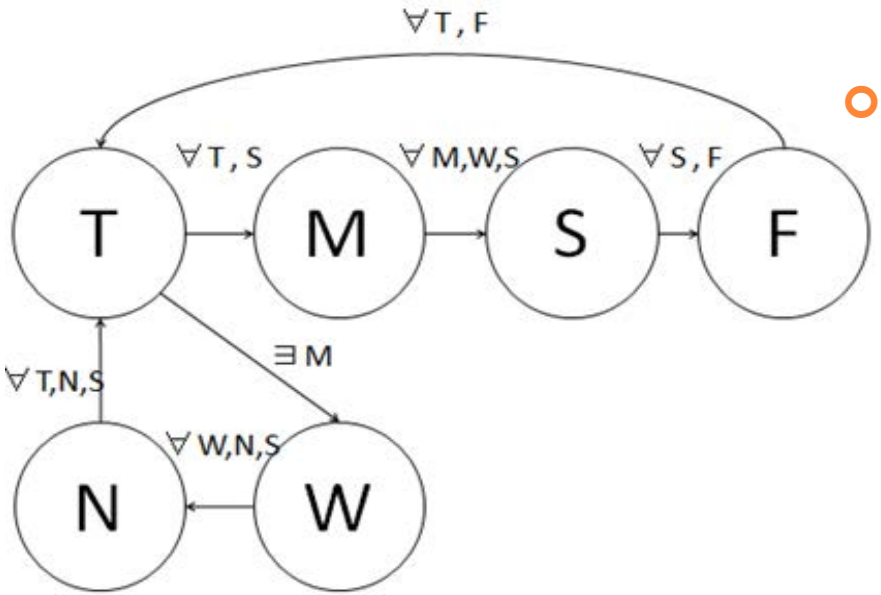
- $ASYNC^6$モデルは、SSYNCモデルと同等である。
  - SIMプロトコル

$$ASYNC(n)^6 \supseteq SSYNC(n)$$

出典:Shantanu Das , Paola Flocchimi , Giuseppe Prencipe , Nicola Santoro , Masafumi Yamashita (2012)"The Power of Lights : Synchronizing Asynchronous Robots using Visible Bits" 2012 32nd IEEE International Conference on Distributed Computing Systems pp.506-515

SIMプロトコル

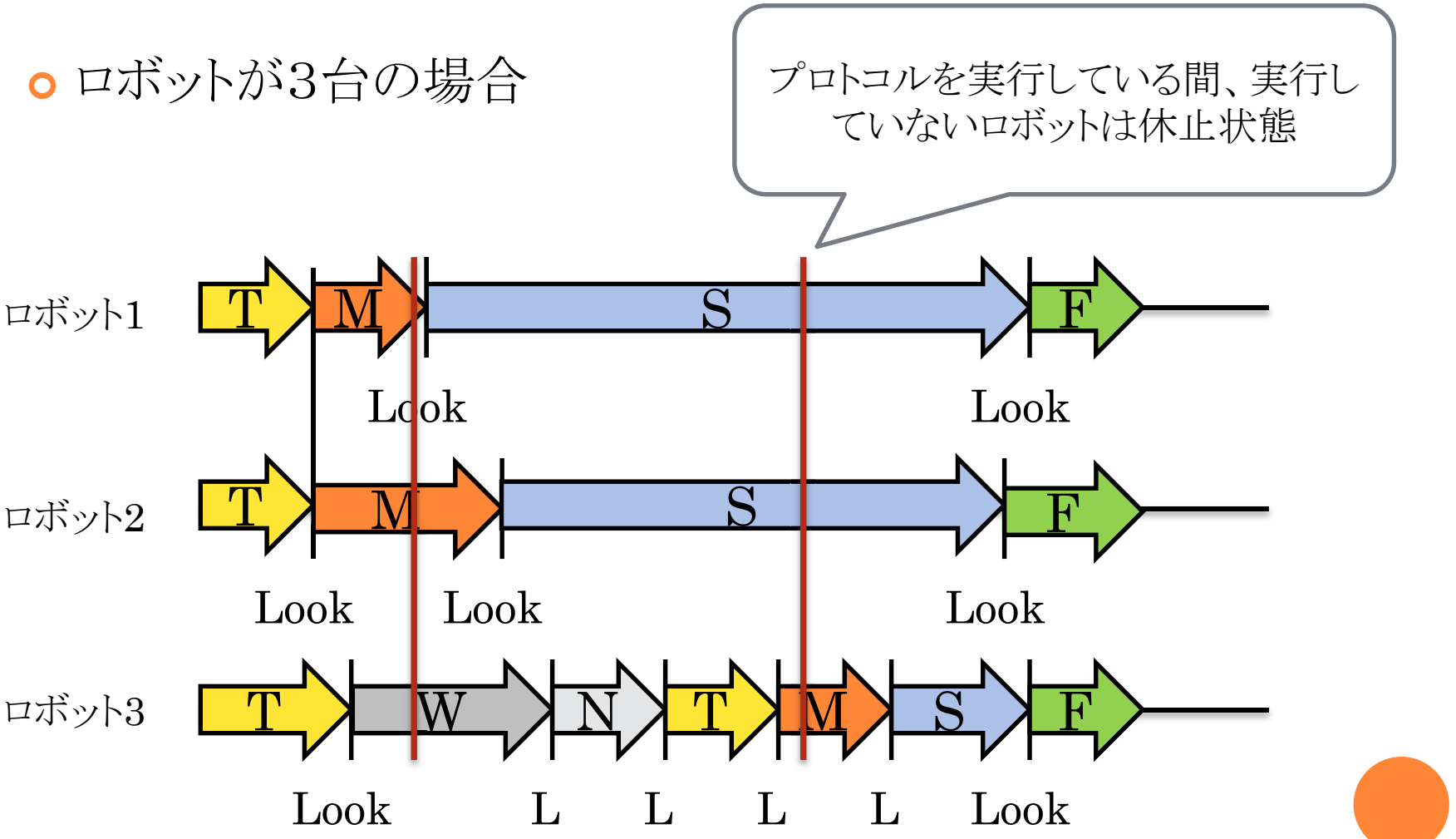
- 6状態を使用する。
  - T(rying) , M(ove) , S(topped) , F(inished) , W(aiting) , N(ext)
  - はじめ、すべてのロボットは状態T
  - 状態遷移はCompute時に行われる。
  - 状態TからMに状態遷移するとき、プロトコルを実行



- Mega-Cycle
  - すべてのロボットが、初期状態Tから最終状態Fになるまでのサイクル
  - SIMでは、Mega-Cycle間に必ず一度、プロトコルを実行する。

SIMプロトコル

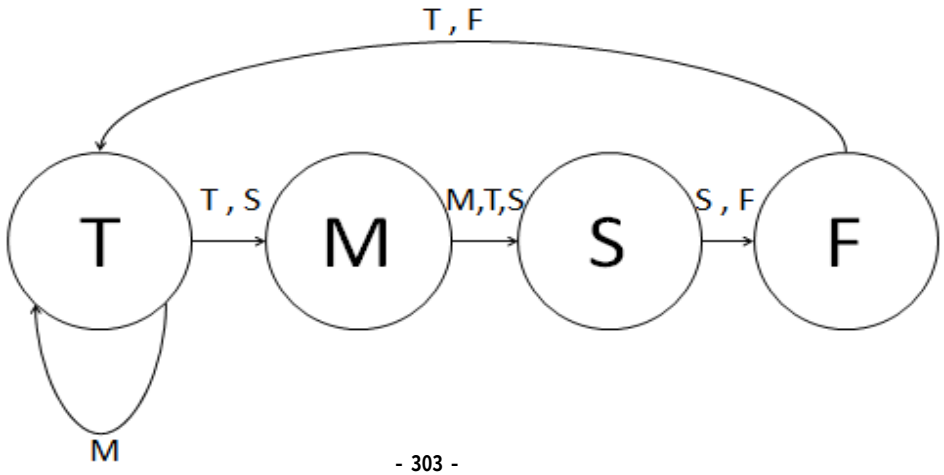
- ロボットが3台の場合



ロボット2台の場合のSIM

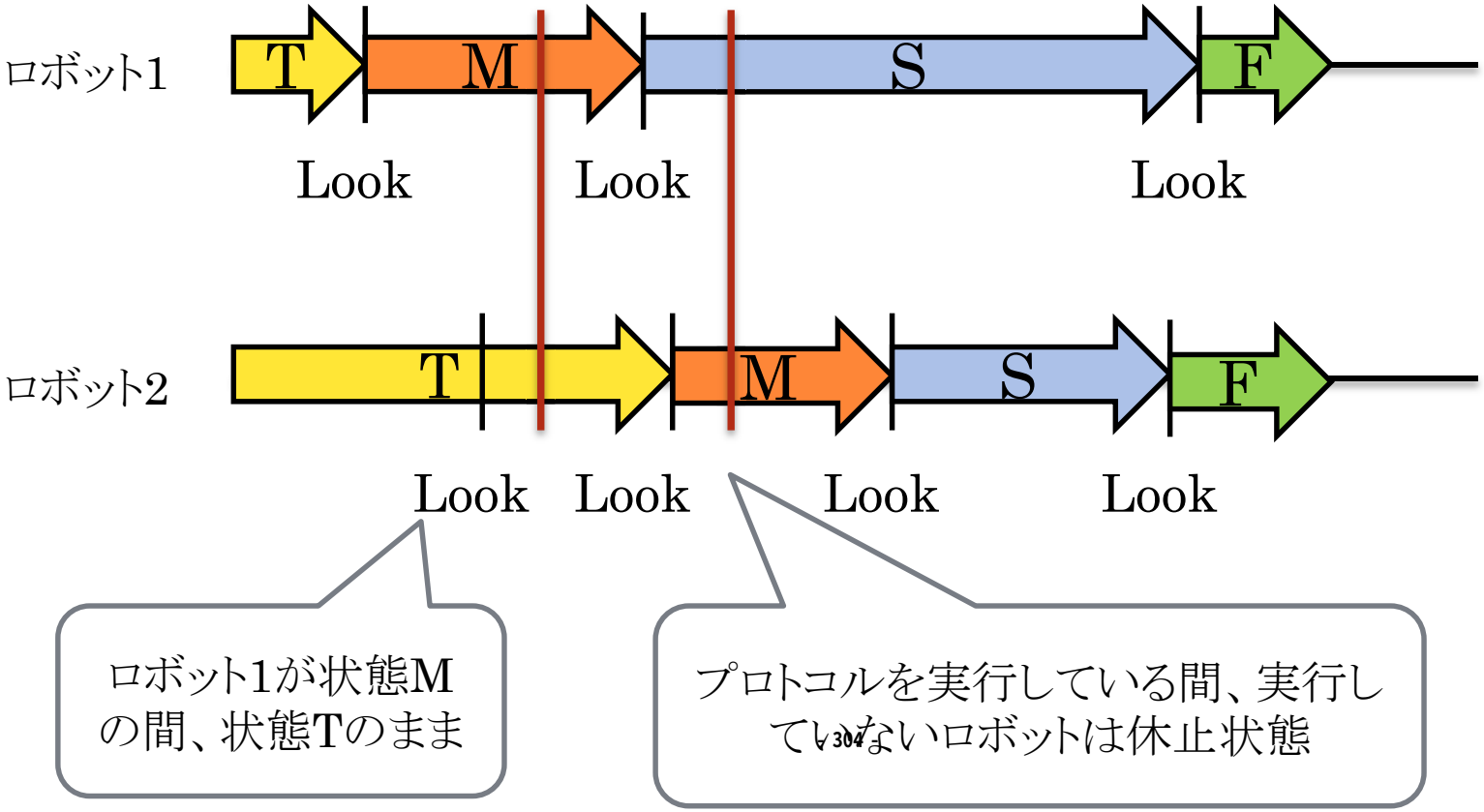
- ロボット2台に限定した場合、SIMプロトコルの状態数の削減を考える。

- 6状態→4状態の削減が可能
  - 4状態:T , M , S , F
  - 状態遷移はCompute時に行う。
  - 状態T→Mに遷移するとき、プロトコルを実行する。



ロボット2台の場合のSIM

- 改良したSIMでも、Mega-Cycle間で2台のロボットは必ず一度プロトコルを実行する。



ロボット2台の場合のSIM

- 今回の結果より、

$$ASYNC(n)^6 \supseteq SSYNC(n)$$

  - n=2の場合は、

$$ASYNC(2)^k \setminus SSYNC(2) \neq \emptyset$$より

$$ASYNC(2)^4 \supset SSYNC(2)$$

今後の課題

- 今回は、ASYNCモデルのロボットは、自身と相手の両方の状態を見ることができた。
  - full-light

- 状態の見え方に制限を加えた場合について
  - internal
  - external

	自身の状態	相手の状態
full-light	参照可	参照可
internal	参照可	参照不可
external	参照不可	参照可

キラリティのない分散ロボットの平面形成問題

冨田祐作　山内由紀子　来嶋秀治　山下雅史
九州大学

1　はじめに

本研究では空中，水中，宇宙空間などの 3 次元空間中を自律的に移動するロボット群の分散制御を考える．2 次元平面上でのロボット群の分散制御に関する研究は数多く存在しているため，3 次元空間中のロボットを，2 次元平面上に乗せることは有用であると考えられる．山内ら [1] は，3 次元空間中の無記憶完全同期ロボット群に対し，全てのロボットの局所座標系が右手系に統一されている (**キラリティがある**) という仮定の下での平面形成問題の必要十分条件を示した．

本研究では，ロボットの持つ局所座標系に着目し，右手系と左手系が混在している (**キラリティがない**) 場合を考え，3 次元空間中のキラリティのない無記憶完全同期ロボット群の平面形成問題の十分条件を示す．本稿では，山内ら [1] の平面合意条件を満たさないロボット群の配置であっても，ロボット群にキラリティが無ければ，平面を合意可能な場合が存在することを示す．

2　諸定義

2.1　ロボットのモデル

各ロボットは，3 次元空間中の点であると仮定し，観測，計算，移動からなるサイクルを同期的に実行する**完全同期モデル**であると仮定する．さらに，各ロボットは**無記憶**であり，自分自身や他のロボットが一つ前のサイクルに行った動作を記憶することができない．また，各ロボットは移動途中に停止せず，各ロボットの移動経路は重なっても良いとする．またロボットは自身の局所座標系で観測を行うとし，本研究では局所座標系に関していずれの仮定も置かない．

2.2　平面形成問題

平面形成問題とは全てのロボットがひとつの平面を合意し，合意した平面へ着地する問題である．文献 [1] ではロボットは重複無く，異なる点に着地することを平面形成の条件としている．本研究では，重複を避けて着地不可能な場合が存在するという問題を解決するために，着地の際の 2 点重複を許す．

2.3　対称性の群

山内ら [1] はロボット群を点集合とみなし，回転操作がなす群である**回転群**を用いることでロボットの配置の対称性を分類している．本研究でもまた，ロボットの配置の対称性に着目するが，キラリティのない仮定において回転群により対称性を分類しようとすると，右手系と左手系が鏡映対称性であることを原因とする問題が生じる．そこで，回転操作と鏡映操作が成す群である**対称性の群**を用いてロボットの配置に関する対称性を分類する．回転群と対称性の群は，シェーンフリース記号 [3] を用いて表記することができ，対称性の群は回転軸の種類と本数，鏡映面の存在の有無から全 17 種類に分類される．これは回転群を表す記号と添え字から表記され，巡回群 C，二面体群 D，回映群 S，正四面体群 T，正八面体群 O，正十二面体群 I，と添え字 $\{v, h\}$ を用いて表記される．添え字 v は回転数が最大である軸 (**主軸**) に対して平行方向に鏡面があることを示し，h は主軸に対して垂直方向に鏡面があることを示し，添え字が無い場合は鏡面が無いことを示す．回映群 S は回転群には含まれない群であり，これは回転操作と鏡映操作を同時に行う回映操作，を要素とする群である．また，対称性の群には部分群の関係が存在するものがあり，例えば T は O の部分群である．これを $T \prec O$ と表記することとする．本研究ではロボット全体の配置 P の対称性の群を $\iota(P)$ で表す．回転群と対称性の群の一覧については表 1 に示す．

2.4 $\iota(P)$-分割

文献 [1] で定義される $\gamma(P)$-分割，と同様に定義される $\iota(P)$-分割を定義する．ロボットの配置 P を $\iota(P)$ の群の作用によって軌道に分割できるが，この軌道空間を $\{P_1, P_2, ..., P_m\}$ とし，P の $\iota(P)$-**分割**と呼ぶ．各 P_i の任意の 2 つの要素は $\iota(P)$ の操作によって相互に入れ替えることができ，これを**頂点推移性**があると呼ぶ．頂点推移性を持つ各 P_i 内のロボットは後述する ι **に対するローカルビュー**を用いることで同じ観測結果を持つ．$\iota(P)$ は P を観測する局所座標系に依らないため，各ロボットは $\iota(P)$-分割を計算でき，ι に対するローカルビューを用いることで文献 [1] と同じく全ロボットで $\{P_1, P_2, ..., P_m\}$ を一意に順序づけることができる．

2.5 局所座標系に関する回転群 ω

P に属する全てのロボットの配置と局所座標系が，回転群 A の全ての元を作用させても一致するとき，そのうち最大の位数の群 A を $\omega(P) = A$ とする．すなわち，ω は局所座標系に関する対称性を示す．

2.6 ι に対するローカルビュー

文献 [1] ではローカルビューと呼ばれる観測方法を定め，ローカルビューが同一な集合に回転群を用いて分割することで，平面合意のアルゴリズムの道具としている．キラリティのある仮定下でのローカルビュー [1] は以下のように定められる．

まず**経度**, **緯度**, **高度**を考える．初期配置 $P = \{p_1, p_2, ..., p_n\}$ の配置 p_i を全体座標系でのロボット r_i の位置とする．P は平面に含まれておらず，最小包含球の中心 $b(P)$ について $b(P) \notin P$ と仮定する．最大の空の球 $L(P)$ は中心が $b(P)$ であり，内側に P の点を含まない球である．また $L(P)$ は少なくとも P の 1 つの点を表面に含む．r_i は $L(P)$ を**地球**として捉え，直線 $p_i b(P)$ を**地軸**として捉える．線分 $\overline{p_i b(P)}$ と $L(P)$ の交点を北極 NP_i とする．そのときロボット r_i は地軸上に無いロボットを子午線ロボット r_{m_i} を選ぶ．r_{m_i} の選び方は後述する．線分 $\overline{p_{m_i} b(P)}$ と $L(P)$ との交点を MP_i とする．$L(P)$ 上の NP_i から始まり，MP_i を含む大きな半円を**本初子午線**とする．r_i の**局所観察**をこの地球を中心とした経度，緯度，高度を用いて変換する．ロボット $r_j \in P$ の位置を高度 h_j(範囲 [0,1])，経度 θ_j(範囲 [0,2π))，緯度 ϕ_j(範囲 [0,π]) で表す．$L(P)$ 上の点を高度 0，$B(P)$ 上の点を高度 1 とする．MP_i の経度を 0 とする．また，経度の正の方向は反時計回りとする．NP_i の緯度を 0, 赤道を $\pi/2$, 南極を π とする．p_j の位置を $p_j^* = (h_j, \theta_j, \phi_j)$ と表記する．この位置の比較として辞書式順序と同様の方法で，順序付けを行う．二つの位置 (h, θ, π) と (h', θ', π') があるとき, 以下の条件の時のみ $(h, \theta, \pi) < (h', \theta', \pi')$ と順序付ける．

$$
\begin{aligned}
h &< h', or \\
h &= h' \, and \, \theta < \theta', \, or \\
h &= h', \theta = \theta' \, and \, \pi < \pi'
\end{aligned}
$$

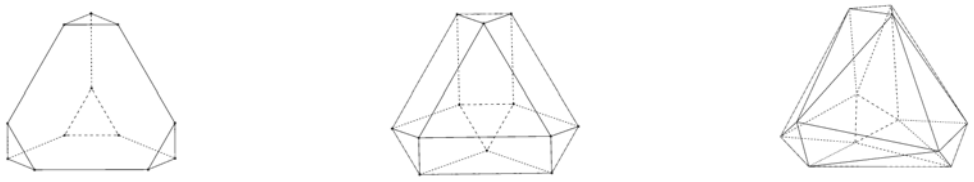
$V_i^* = \langle p_i^*, p_{m_i}^*, p_{j_1}^*, p_{j_2}^*, ..., p_{j_{n-2}}^* \rangle$ を p_j^* の位置を並べたリストとする.($p_{j_k}^* < p_{j_{k+1}}^*$)．これを r_i のローカルビューとする．子午線ロボットはローカルビュー V_i^* が最も小さい順序になるように選ばれる．

しかし，この座標系によらない観測方法であるローカルビューをキラリティのないロボット群に適応すると，ロボットが右手系であるときは経度の正方向は全体座標系 (右手系) から見て**反時計周り**，左手系であるときは**時計周り**，となり経度の正方向が一意に定まらない．すなわち $\iota(P)$-分割された部分集合である $\{P_1, P_2, ..., P_m\}$ を一意に順序付けることができなくなる可能性がある．よって $\iota(P)$-分割した各部分集合のローカルビューが一致するように，以下のような変更を加え ι に対するローカルビューの定義を定める．

ローカルビュー [1] を定めようとしたとき，経度の正の方向を反時計周りを正とするか，時計周りを正とするかによってローカルビューが異なる場合は小さくなる方をローカルビューとする．

3 キラリティのないロボット群の平面合意問題の十分条件

文献 [1] では，ロボットの対称性が回転群で C_n または D_n，すなわち対称性の群で対称性が C_n，C_{nv}，C_{nh}，D_n，D_{nd}，D_{nh} であると平面合意可能であることが示されている．ここで山内ら [1] の平面合意条件に含まれない初期配置 P であっても，キラリティがない場合，あるアルゴリズムによってロボットを移動させた後の配置 P' の対称性の群 $\iota(P')$ を平面合意可能な対称性に変換可能な場合が存在することを次の補題 1 にて示す．



(a) (b) (c)

図 1: 12 台 $\iota(P) \in \{T, T_v, T_h\}$ 対称性を持つ配置例

補題 1. P は頂点推移性を持ち，$|P| = 12$ であるとする．また n_R を P に含まれる右手系の台数，n_L を左手系の台数とする．$\iota(P) \in \{T, T_v, T_h\}$ であるとすると，このとき $(n_R > 0)$ かつ $(n_L > 0)$ であるならば無記憶完全同期ロボット群は，$\iota(P') \in \{C_n, C_{nv}, C_{nh}, D_n, D_{nv}, D_{nh}\}$ を満たす配置 P' に P を変換できる．

補題 1 の条件にあてはまる具体的な配置は図 1 である．補題 1 にて，$\iota(P)$ の条件が $\{T, T_v, T_h\}$ である理由を述べる．先に述べたように補題 1 は山内ら [1] の示した平面合意不可能な初期配置 P であっても，キラリティがない場合平面合意可能になる場合が存在することを示している．そのため補題 1 の条件はキラリティのある仮定において平面合意不可能である条件を示しているが，キラリティのある仮定 [1] において平面合意不可能な配置に属する対称性は，部分群として T を含んでいるという性質があり，回転群 T は対称性の群における $\{T, T_v, T_h\}$ の部分群であるため，対称性が $\{T, T_v, T_h\}$ であるとき平面合意不可能な場合が存在する．よって，キラリティのある仮定において平面合意不可能な配置を，対称性の群で表した $\{T, T_v, T_h\}$ が補題 1 の条件となっている．この平面合意不可能な対称性としてあげられる $\{T, T_v, T_h\}$ を平面合意可能な配置に変換できる場合を補題 1 にて示している．

この補題 1 にて使用される P を変換するアルゴリズム ψ はある前提条件を持つ入力に対し，各ロボットが実行するアルゴリズムであり，

Z 軸の負の方向を最小包含球の中心 $b(P)$ に向け，そのロボット r_i から最も近い 3 回回転軸から ϵ 手前の地点に移動する．その後最も近い 3 回回転軸を中心に，反時計回りに $1/6\pi$ 回転する．

という動作を主に行うものであり，また ψ が仮定する前提条件は次の二つを両方満たすものである．

1. ロボットから最も近い 3 回回転軸が 1 本のみ存在．
2. $\iota(P) \succeq T$

ただし，実際はロボット群が持つ最小包含球の表面上 (図 2) に移動した方が，ロボットの重複を回避できるため，実際の ψ は (Algorithm3.1) とする．

また補題 1 と $\iota(P)$-分割を用いることで定理 1 が証明される．なお，定理 1 は山内ら [1] の平面形成の条件が成立しない場合の平面形成可能性である．

Algorithm 3.1 ψ *at* r_i

記法：

P：全体座標系 Z_i での配置．
$B(P)$：P を包括する最小包含球．
$b(P)$：最小包含球 $B(P)$ の中心．
$rad(B(P))$：最小包含球 $B(P)$ の半径．
$\iota(P)$：対称性の群における P の対称性．
r_i：P に属するロボットのうち自分自身を指す．
$n_{r_{ib}}$：r_i からもっとも近い 3 回回転軸と，
　　$B(P)$ との 2 つの交点のうち
　　r_i から距離の小さい方の交点．
$\widehat{r_i n_{r_{ib}}}$：$B(P)$ 上を通る r_i と $n_{r_{ib}}$ を結ぶ弧．
ϵ：$\epsilon = rad(B(P))/2$

前提条件：

r_i から最も近い 3 回回転軸が 1 本に定まる．
P は頂点推移性を持ち，$\iota(P) \succeq T$

アルゴリズム：

Z 軸の負の方向を最小包含球の中心 $b(P)$ に向ける．
$\widehat{r_i n_{r_{ib}}}$ 上で，$n_{r_{ib}}$ からの弧上の距離が ϵ 手前である点に移動し (図 2)，もっとも近い 3 回回転軸を中心に反時計回りに $1/6\pi$ 回転する．
(図 3 はロボットが移動する際に通る円周を表す．)

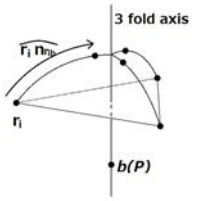


図 2: ψ(Algorithm3.1) による球面移動のイメージ

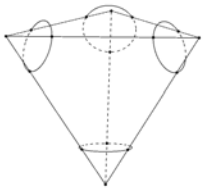


図 3: ψ による移動範囲

定理 1. $P(0)$ を任意の初期配置，$\iota(P(0))$ を $P(0)$ の対称性の群とする．$\{P_1, P_2, \cdots, P_m\}$ を $\iota(P(0))$-分割とする．P_i に含まれる右手系の台数を n_{i_R}, 左手系の台数を n_{i_L} と表す．$^{\forall}i, (|P_i| \in \{12, 24, 48, 60, 120\})$ であるとする．このとき $^{\exists}i, ((n_{i_R} > 0) \wedge (n_{i_L} > 0) \wedge ((|P_i| = 12) \vee (\omega(P_i) \not\succ T)))$ が成り立つならば無記憶完全同期ロボット群は平面を合意可能である．

文献 [1] の平面形成の条件と定理 1 により，キラリティのないロボット群の 2 点重複を許す平面形成問題の十分条件を示すことができる．

定理 2. $P(0)$ を任意の初期配置とし，$\{P_1, P_2, \cdots, P_m\}$ を $\iota(P(0))$-分割とする．P_i に含まれる右手系の台数を n_{i_R}，左手系の台数を n_{i_L} と表す．キラリティのない無記憶完全同期ロボット群は以下の条件を少なくとも *1* つ満たすとき，着地の際 *2* 台のロボットの重複までを許すと平面形成可能である．

1. $\iota(P) \not\succ T$,
2. $\{\iota(P) \succeq T\} \wedge \{^{\exists}i, (|P_i| \notin \{12, 24, 48, 60, 120\})\}$,
3. $\{\iota(P) \succeq T\} \wedge$
   $\{^{\forall}i, (|P_i| \in \{12, 24, 48, 60, 120\})\} \wedge$
   $\{^{\exists}j, ((n_{j_R} > 0) \wedge (n_{j_L} > 0) \wedge (\omega(P_j) \not\succ T))\}$.

この定理を実現するアルゴリズムを具体的に与えることで，定理 2 は証明可能である．

4　今後の課題

定理 2 はキラリティのないロボット群に対する平面形成問題の十分条件であるため，今後必要条件についても検討する．さらに，今回は平面への着地の際に 2 点重複を許しているが，重複を回避するための手法条件を検討することも課題である．

参考文献

[1] Y. Yamauchi, T. Uehara, S. Kijima, and M. Yamashita, Plane Formation by Synchronous Mobile Robots in the Three Dimensional Euclidean Space, The 29th International Symposium on Distributed Computing (DISC 2015) (to appear).

[2] 下川航也，平澤美可三，松本三郎，丸本嘉彦，村上　斉 (訳) : 多面体 新装版，数学書房 (2014).

[3] 今野豊彦 : 物質の対称性と群論，共立出版 (2014).

[4] GeoGebra <http://www.geogebra.org/download?ggbLang=ja>

表 1: 対称性一覧 [1][2][3]

回転群	対称性の群	2 回回転軸	3 回回転軸	4 回回転軸	5 回回転軸	n 回回転軸	水平鏡映面	垂直鏡映面	位数
C_1	C_1	0	0	0	0	0	×	×	1
	C_s	0	0	0	0	0	×	○	2
	C_i	0	0	0	0	2 回回映軸	×	×	2
C_n	C_n	0	0	0	0	1	×	×	n
	C_{nv}	0	0	0	0	1	×	○	2n
	C_{nh}	0	0	0	0	1	○	○	2n
D_n	D_n	n	0	0	0	1	×	×	2n
	D_{nv}	n	0	0	0	1	×	○	4n
	D_{nh}	n	0	0	0	1	○	○	4n
$C_{n/2}$	S_n	0	0	0	0	n 回回映軸	×	×	n
T	T	3	4	0	0	0	×	×	12
	T_d	3	4	0	0	0	×	○	24
	T_h	3	4	0	0	0	○	○	24
O	O	6	4	3	0	0	×	×	24
	O_h	6	4	3	0	0	○	○	48
I	I	15	10	0	6	0	×	×	60
	I_h	15	10	0	6	0	○	○	120

限られた視界を持つ自律ロボット群による線分被覆問題

門出顕宏 * 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
† 九州大学大学院システム情報科学研究院

1 はじめに

限られた視界を持つ自律ロボット群において，その視界で線分を被覆するアルゴリズム SCA (Segment Cover Algorithm) を提案する．本研究で用いたロボットシステムのモデルは，(1) 無記憶 (2) 完全同期 (3) 匿名 (4) 視野の制限有の自律分散モデルである．ロボットは，自身を原点とした局所座標系を持ち，その動作は，周囲のロボットの位置を観測 (Look)，移動先を計算 (Compute)，移動 (Move) による一連のサイクルの繰返しからなる．

既存の研究において，この問題が求解可能であるためのロボットモデルの条件を幾つか提案している．Cohen ら [1] は，視界の制限のないロボットによる線分上の等間隔配列問題を解決するアルゴリズムを示している．Eftekhari ら [2] は，侵入者を感知するセンサーの半径が r で，他のロボットを確認することができる視界半径が $2r$ であるロボットについて線分被覆問題を解決している．本稿では，各ロボットが自身の持つ視界の範囲を知らないという条件を新たに加え，[2] でいうところの侵入者を感知するセンサーと他のロボットを確認することができる視界の半径が等しいとしている．これらの条件下で，線分をロボットの視界で被覆するのに十分なロボットの台数に対し，いくらか多くロボットを投入することで，この問題を解く．ロボットを多く投入することについては，後に議論する．また，ロボットは与えられた線分の両端をロボットと同様に認識するものとする．

本研究は，一次元空間でのロボットによる監視を目標としている．被覆のためのロボットの台数に余分を持たせるのは，自身の視界範囲を知らないためである．このような制約を要求するのは，ロボットシステムをよりロバストなシステムにするためである．例えば，各ロボットの視界半径に差が生じた場合，アルゴリズム SCA では，視界半径を情報として用いていないため，線分被覆問題を解決可能である．

ここで，ロボットの可視グラフ (Visivility graph) という概念を導入しておく．ロボットを**頂点**とみなし，相互に確認できるロボットどうしを**辺**で結んだものをロボットの可視グラフと言う．つまり，可視グラフにおいて，**隣接**しているロボット対は，相互に確認しているロボットの対である．

2 問題

与えられた被覆する線分の長さを L とし，ロボットの視界を半径 V の円とする．このとき，線分被覆問題を解くとは，任意の初期配置から，半径 V の視界を持つ n 台のロボットを等間隔に配列し，それら視界で長さ L の線分を被覆することを言う．ただし，各ロボットは目標の点に収束し，終端配置においてすべてのロボットは連結であることを条件とする．

3 アルゴリズム

ロボットは視界半径 V を知らず，メモリを持たないため，アルゴリズムに使用できる情報は，視界内にいる他のロボットとの距離のみである．ロボットの台数 n は，$n \geq \frac{L}{V} - 1$ を満たせば，線分全体を視界で被覆するのに十分であるが，アルゴリズム SCA では $n > \frac{m}{m-2}\frac{L}{V} - 1$ のロボットを投入することで線分被覆問題を解決する．ただし，$m \geq 2$ はアルゴリズムが与えるパラメータであり，$m \geq 2$ としているのはロボットどうしの衝突を避けるためである．どのようにして衝突を避けているかについては，次章に

て説明する．

与えられた線分の一端の座標を $x = 0$，他端の座標を $x = L$ とし，$x = 0$ にもっと近いロボットから順番に，$r_1, r_2, ..., r_n$ とし，投入されたロボットすべての集合を $R = \{r_1, r_2, ..., r_n\}$ とする．このようにしてロボットに与えた順番は，時間によらず不変である．ロボットは視界内に存在するすべてのロボットを認識可能であるが，ロボット r_i がアルゴリズムで使用するのは，ロボット r_{i-1}, r_{i+1} との距離で，自身から見てロボットが (1) 両側に確認できる場合 (2) 片側にのみ確認できる場合 (3) どちら側にも確認できない場合で分けている．各ロボットが同一のアルゴリズム SCA をそれぞれ用いることで，任意の初期配置について，ロボットどうしの衝突なしに線分の被覆を実現する．

アルゴリズムにより，視界内の線分上に，自身を原点とする一次元座標系を各ロボットに定義させる．

3.1 両側にロボットが確認できる場合

自身の両隣のロボットを結ぶ線分の中点へ移動する．

局所座標系においては，原点からの距離が最も小さい正方向のロボットの座標を x_{i+1} とし，原点からの距離が最も小さい負方向のロボットの座標を x_{i-1} ととする．このとき，$(x_{i-1} + x_{i+1})/2$ へ移動する．

3.2 片側にのみロボットが確認できる場合

自身の隣のロボットとの距離を d とすると，自身とそのロボットを結ぶ直線上の点へ d/m だけそのロボットから遠ざかるように移動する．

局所座標系においては，原点からの距離が最も小さいロボットの座標を x_{i+1} とし，$-x_{i+1}/m$ へ移動する．

3.3 どちら側にもロボットが確認できない場合

ロボットは今いる点で静止する．

4 アルゴリズムの正当性

まず，ロボットどうしの衝突の問題について，衝突が発生しないことを明確にしておく．ロボット r_i について，前章の 3.1 より，自身の両側にロボットが確認できる場合は，明らかに衝突しない．3.3 より，どちらにもロボットが確認できないとき，自身は移動しない．r_{i+1}, r_{i-1} との距離は，どちらも V より大きく，この 2 台の移動距離はそれぞれ高々 V/m である．ここで，$m \geq 2$ より，衝突は起こらない．最後に，片側にのみロボットが確認できる場合，r_{i-1} が確認できているとすると，r_{i+1} と最も接近するのは，r_i, r_{i+1} ともに V/m だけお互いに近づくように移動するときである．ここで，r_i と r_{i+1} との距離は V より大きく，$m \geq 2$ より，$V/m + V/m \leq V$ であるから，衝突は起こらない．以上より，すべての場合において衝突は発生しないことが言える．仮に，$m < 2$ とすると，衝突が発生することがわかる．

次に，実際にアルゴリズム SCA で，線分被覆問題を解決できることを示す．

定理 1. $n > \frac{m}{m-2}\frac{L}{V} - 1$ を満たすとき，アルゴリズム SCA により，n 台のロボットで線分被覆問題を解決できる．

定理 1 は，$n > \frac{m}{m-2}\frac{L}{V} - 1$ の下でアルゴリズム SCA を実行した場合に成り立つ以下の補題 3, 4 より証明される．

与えられた線分の一端の座標を $x = 0$，他端の座標を $x = L$ とする．時刻 t におけるロボットの間隔を $x = 0$ の方から，$d_0(t), d_1(t), ..., d_n(t)$ とし，$D(t) = \{d_i(t) | i = 0, 1, ..., n\}$ とする．$D(t)$ のうち，最小のものを $d_{\min}(t)$ とする．このとき，以下の補題 1, 2 が成り立つ．

補題 1. 任意の時刻 t について，$d_{\min}(t+1) \geq d_{\min}(t)$ が成り立つ．

補題 2. $d_{\min} < \frac{m-2}{m}V$ のとき，$d_{\min}(t+1) > d_{\min}(t)$ となる時刻 t が存在する．

$d_{\min} < \frac{m-2}{m}V$ は常に成り立つため，以上の補題 1, 2 より，以下が成り立つ．

補題 3. すべてのロボットは，それぞれの両隣のロボットと相互に確認できる状態に到達する．r_1, r_n については，それぞれ線分の一端と r_2，他方の線分の一端と r_{n-1} を確認できる状態に到達する．

時刻 t におけるロボットの可視グラフを無向グラフ $G(t) = (R', E(t))$ で定義する．また，線分に

おける $x = 0, L$ の点 (線分の両端点) をそれぞれ，r_0, r_{n+1} とする．$R' = R \cup \{r_0, r_{n+1}\}$ は，すべてのロボットと線分の両端点の集合であり，時刻 t において，r_i, r_{i+1} が相互に確認できるかつそのときに限り，$(r_i, r_{i+1}) \in E(t)$ を定義する．自身の両側にロボットが確認できる場合に適用されるアルゴリズムにより，次のことが言える．

補題 4. 可視グラフ $G(t)$ が一つの連結成分となった時刻を $t = t_c$ とすると，$t \geq t_c$ において，$G(t+1) = G(t)$ であり，ロボットの位置は線分の両端点を含むすべてのロボットが等間隔になる点へ収束する．

5 余剰台数の必要性

図 1 に 1 台のロボットで線分を被覆する場合を示す．図 2 には，アルゴリズム SCA の (2) 片側にのみ確認できる場合のロボットの動きを示す．

アルゴリズム SCA は，長さ $L = 2V$ の線分が与えられたとき，1 台のロボットでは線分被覆問題を解決できない．更に，$\frac{2m+1}{m+1}V < L < 2V$ の線分が与えられた場合においても，1 台のロボットでは線分被覆問題を解決できない．$d_1(t+1) = d_2(t)$ すなわち，$d_1(t) + \frac{d_1(t)}{m} = L - d_1(t)$ となるとき，ロボットの動きは周期的になり，収束しない．また，$d_1(t) > \frac{m}{2m+1}L$ のとき，$d_1(t) > d_2(t)$ となり，これもロボットの動きは収束しない．$d_1(t) < \frac{m}{2m+1}L$ $(d_2(t) < \frac{m}{2m+1}L)$ のとき，$d_1(t) < d_2(t+1)$ $(d(t+1) > d_2(t))$ が常に成り立つ．よって，$L - \frac{m}{2m+1}L = \frac{m+1}{2m+1}L \leq V$ であれば，(2) 片側のみ確認できる場合に適用されるアルゴリズムによって，与えられた線分の両端点が確認できる領域へ必ず進入できる．

このように，1 台のロボットにアルゴリズム SCA を適用して線分を被覆させる場合にも，$L < 2V$ の条件では解決できないことがわかる．

6 おわりに

本研究は，一次元空間におけるロボットの監視を目的としている．これを今回と同じ条件下で二次元空間へ拡張し，平面図形のロボットの監視へと繋げたいと考えている．

参考文献

[1] R. Cohen, and D. Peleg, “Local spreading algorithms for autonomous robot systems,” Theoretical Computer Science, **399**, pp.71–82, 2008.

[2] M. Eftekhari, P. Flocchini, L. Narayanan, J. Opatrny, and N. Santoro, “Distributed Barrier Coverage with Relocatable Sensors.” Proc. of the 21st International Colloquium on Structural Information and Communication Complexity, pp.235–249, 2014.

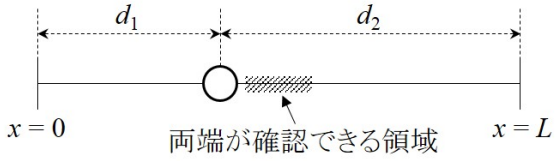


図 1　1 台のロボットによる線分被覆

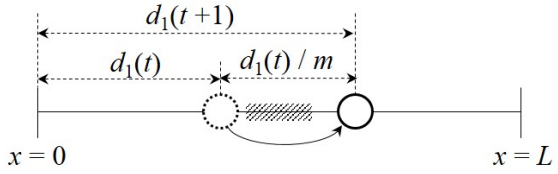


図 2　アルゴリズム SCA によるロボットの動き