Vol.2012-MPS-87 No.4
2012/3/1

自己安定アルゴリズムの出力変化回数削減手法の提案

瓦谷 佳祐[†1]　山内 由紀子[†2]　伊藤 実[†1]

故障耐性を持つ分散アルゴリズムのひとつに，自己安定アルゴリズムがある．自己安定アルゴリズムでは，収束途中に各プロセスの出力変数が複数回変化する可能性がある．このような出力変数の変化は，自己安定アルゴリズムの出力を利用する外部のアルゴリズムに何度も再計算を行わせる可能性があり，様々なアルゴリズムの組み合わせによって実現されるような大規模なシステムでは大きな影響が出る可能性がある．本研究では，自己安定アルゴリズムの復旧途中の出力変化回数を高々2 回に削減することで故障の影響を抑制する手法を提案する．システムにおける計算機の数を n とすると，極大マッチング問題に対する既存のアルゴリズム [Chattopadhyay ら, 2002] では，全ての計算機の出力変化回数の総和は $O(n^2)$ である．提案アルゴリズムでは，極大マッチング問題における出力変化回数を $O(n)$ に削減する．

Reducing the number of output changes in self-stabilizing algorithms

Keisuke Kawaratani,[†1] Yukiko Yamauchi[†2] and Minoru Ito[†1]

One of the most versatile techniques for designing a fault-tolerant distributed system is self-stabilization. A self-stabilizing algorithm can make processes change their states many times during the recovery. These output changes are seen by the external systems and their effect may spread to other external systems. We propose a self-stabilizing maximal matching algorithm that guarantee each process changes its output at most twice during the recovery. The number of output changes during the recovery is $O(n^2)$ in the existing maximal matching algorithm proposed by [Chattopadhyay et al., 2002]. and $O(n)$ in the existing maximal independent set algorithm proposed by [Ikeda et al., 2002]. On the other hand, the proposed algorithms guarantees the number of output changes are $O(n)$.

1. はじめに

インターネット，携帯電話網，高度道路交通システム，多種多様な無線センサネットワークの普及に伴い，近年，様々な計算機システムが相互に連携した大規模かつ複雑な構造を持つシステムが増加している．このような計算機ネットワークは社会インフラとして今後益々重要になっていくと考えられ，可用性，頑健性，効率性，高性能性などを備えた計算機ネットワークの運用手法が必要とされている．本研究では，相互に連携する計算機システムが連携のために行う情報伝達に着目し，個々の計算機の故障に頑健な相互連携の手法を検討する．分散システム中の個々の計算機で故障が発生した場合にも，システム全体の処理が正しく行われることや，故障の影響からシステムが復旧することなどを目標とし，耐故障分散アルゴリズムについて多くの研究がなされている[1),2)]．

優れた故障耐性のひとつに自己安定性がある[3)]．どのような状況から実行を開始しても，やがてシステムがその目的を満たす状況に到達する（収束する）とき，分散アルゴリズムを自己安定であると言う．多くの自己安定アルゴリズムでは，収束途中にほぼすべての計算機が何度も再計算を行う．一般的に，分散アルゴリズムの外部のシステムや観測者への出力は個々の計算機が局所的に管理する出力変数に書き込まれる．多くの自己安定アルゴリズムでは，故障の影響により再計算が発生し，出力変数の値が何度も更新される．しかし，自己安定アルゴリズムによって実現されたシステムが相互に複雑に連携するシステムの一部に組み込まれた場合，出力が何度も変化すれば，連携するシステムにも再計算を引き起こし，やがてはシステム全体に大きな影響を及ぼす可能性がある．

自己安定アルゴリズムが収束に必要とする通信量を解明する手法として，文献 4) は単調収束性という概念を提案した．どのような実行においても，収束途中に各計算機が高々1 回しか計算を行わない時，この自己安定アルゴリズムを単調収束であると言う．文献 4) では，様々な分散問題に対して，単調収束性を実現するために必要な情報の量を計測する手法を提案している．しかし，単調収束な自己安定アルゴリズムの実現方法は示されていない．

本稿では，単調収束の概念をもとに，収束途中に出力変化の少ない自己安定アルゴリズムの設計手法を提案する．本稿では，極大マッチング問題について，各計算機における出力変

†1 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
†2 九州大学
Kyushu University

化回数を高々2 回に削減する手法を提案し，出力変化回数を $O(n^2)$ から $O(n)$ に削減する．

関連研究

山内らは，多くの自己安定アルゴリズムは収束するまでの間に不必要な出力の変化が起こることを指摘し，各計算機における高々1 回の状態変化により，システムの状況が初期状況から正当な状況へと変化していく単調収束性を提案した[4]．単調収束性により，出力変化回数を削減することで，システム外部に位置する外部システムやシステム観測者が不要な再計算を避けることができる．単調収束性では，各計算機における近傍の状態やトポロジー（近傍情報）と過去の状況における近傍情報の履歴（局所履歴）を各計算機がもつ情報量として定めており，様々なグラフ問題について情報量がどれだけ必要であるかを解析している．しかし，出力単調収束な自己安定アルゴリズムの具体的な実現方法については言及していない．

自己安定アルゴリズムを適用する分散問題として，極大マッチング問題が挙げられる．Chattopadhyay らは，d デーモンのもと，匿名ネットワークに対して，収束時間が $O(n)$ ラウンド（$O(n^2)$ ステップ）で極大マッチング問題を解くアルゴリズムを提案した[5]．Chattopadhyay らのアルゴリズムでは，プロセスを識別するための ID を用いることによって，収束時間 $O(n)$ ラウンドを実現している．このアルゴリズムでは，各プロセスにおける出力変化回数は $O(n^2)$ である．

本研究では，極大マッチング問題に対してし，出力変化回数が $O(n)$ である自己安定アルゴリズムを提案する．

2. 諸 定 義

本章では，まず本稿で扱う分散システムを説明し，次に収束の単調性について述べる．

2.1 システムモデル

分散システムはネットワークとアルゴリズムから成る．プロセス集合 V，通信リンク集合 E で構成されるネットワークを無向グラフ $G = (V, E)$ と表す．2 つのグラフ G と G' について，G' のプロセス集合と通信リンク集合が共に G のプロセス集合と通信リンク集合の部分集合になっているとき，G' は G の**部分グラフ**であるという．G のプロセス集合 V の部分集合 G' を取り出して，両端点が G' に属する全ての通信リンクを通信リンク集合とする G の部分グラフを，**誘導部分グラフ**という．各プロセスは局所変数を管理する状態機械であり，アルゴリズムに従い，局所変数の値を更新する．プロセス $p, q \in V$ について，無向辺 $(p, q) \in E$ であるとき，p は q に隣接するといい，p は q の局所変数の値を遅延なく読むことができる．ただし，p は自身の局所変数の値しか書き換えることができない．

プロセス p の**状態**は，p の全ての局所変数の組みで定義される．各プロセスの局所変数は，**出力変数**，**入力変数**，**内部変数**から成る．出力変数は問題の定義に用いられる変数であり，システム外部の観測者が観測可能である．入力変数は問題の入力を表す．内部変数はそれ以外の局所変数である．各プロセスは出力変数，内部変数の値を書き換えることはできるが，入力変数の値を書き換えることはできない．プロセス p における出力変数を v_p とする．状況 C における v_p の値を $v_p|C$ と表す．プロセス p がとりうる状態の集合を S_p とすれば，システムのとりうる**状況**の集合は全てのプロセスの状態の直積で表される：$\mathcal{C} = \Pi_{p \in P} S_p$．ある状況 $C \in \mathcal{C}$ におけるプロセス p の状況を $S_p|C$ と表す．

各プロセス $p \in V$ はアルゴリズムの実行により自身の内部変数の値を書き換える．アルゴリズムは**ガード付きアクション**の有限集合として与えられる．各ガード付きアクションを $S : G \to A$ と表す．ガード G は $\{p\} \cup N_p$ の局所変数からなる論理式であり，A は p の局所変数の値を書き換えるステートメントである．ガードが真であるとき，そのガードは**有効であると**言う．プロセス p において，有効なガードが 1 つ以上存在するとき，p を**有効である**と言う．

本稿では，スケジューラとして c **デーモン**を想定する．各**計算ステップ**において， c デーモンは，1 つの有効なプロセスを選択し， 選択されたプロセス p は有効なガードアクションを実行することで出力変数や内部変数（局所変数）の値を更新する．また，本稿では弱公平なスケジューラを想定する．弱公平なスケジューラは，継続的に無限にしばしば有効であるプロセスを必ず有限時間内に選択する．

システムの**実行**を極大な状況の系列 $E = C_0, C_1, C_2, ...$ で表記する．ここで極大とは，E が無限系列であるか，もしくは有限系列であればその最後の状況は動作可能なプロセスが存在しない終端状況であるということを意味する．状況 C_{i+1} は C_i に 1 つの計算ステップを適用した結果得られる状況である．

ネットワークはグラフと見なせるので，グラフ理論で用いる用語をネットワークに対しても用いる．G におけるプロセス p の隣接プロセス集合を N_p と表す．ここで，p から離れたプロセスの記法を定義するために，N_p を N_p^1 と表す．$k \geq 2$ に対して，$N_p^k = N_p^{k-1} \cup \bigcup_{q \in N_p^{k-1}} N_q \backslash \{p\}$ とする．N_p^k の集合を p の k-**近傍**と表記する．また，p と $q (p \neq q)$ について，p と q の距離を $dist(p, q)$ で表す．$q \notin N_p^{k-1}$ かつ $q \in N_p^k$ が成り立つならば，$dist(p, q) = k$ である．p の k-近傍プロセスの集合は，p からの距離が k 以下であるプロセスの集合である．

プロセス p の**離心率**は $max_{q \in V}\{dist(p, q)\}$ で表される．グラフの**直径**は，離心率の最大値であり，d で表す．

一時故障は，任意の個数のプロセスの局所変数の値を，同時に任意の値に書き換える．自己安定アルゴリズムは，一時故障が発生しない実行，つまり，各プロセスがアルゴリズムに従って内部変数を書き換え続ければ，どのような実行も，やがてシステムが目的とする状況にたどり着くことを保証している．つまり，最後の一時故障が発生した時点から，新たな一時故障が発生しなければ，システムはやがて正当な状況に到達し，問題を解くことが保証されている．

問題（タスク）P は，各プロセスの**出力変数**に対する論理式で定義される．問題の定義を満たす状況を，P の解状況と呼ぶ．

分散システムがアルゴリズム $\mathcal{A}$ を実行し，有限時間内に P の解状況をとるとき，$\mathcal{A}$ は問題 P を解くと言う．アルゴリズム $\mathcal{A}$ の状況 C が，C から始まるどのような実行においても，問題の定義を満たすとき，C を**正当な状況**と呼ぶ．正当な状況の集合を $\mathcal{C}_L$ と表す．

アルゴリズム $\mathcal{A}$ の任意の実行が必ず正当な状況を含む場合，アルゴリズム $\mathcal{A}$ は**自己安定**であるという．

本研究で扱う分散問題の定義を以下に示す．

極大マッチング問題． 極大マッチング問題では，各プロセス p の出力変数 m_p は，マッチングプロセスとして N_p のうちひとつのプロセスをさすポインタである．グラフにおけるマッチングとは，互いに隣接しない通信リンクの集合である．グラフ上にマッチングの上位集合がない場合，マッチングは極大であるという．このような通信リンクの集合を**極大マッチング**と呼ぶ．全ての隣接プロセスがマッチングを形成している場合，そのプロセスの出力であるポインタは nil を指すこととする．極大独立点集合問題における解状況は，マッチングプロセスの集合がそのグラフの極大マッチングを構成する状況である．

2.2 収束の単調性

単調収束を実現するための情報量の指標は，既存研究[4] における近傍情報と局所履歴を用いる．

定義 2.1 (近傍情報) ネットワークプロセス G におけるプロセス p の k-近傍情報 $view_p^k$ とは，$q \in N_p^k$ の各プロセスの状態と任意の G 上の N_p の誘導部分グラフである状況 C における p の k-近傍情報を $view_p^k$ と表す．

定義 2.2 (局所履歴) プロセス p における $history_p^k[0..\ell]$ は，$\ell+1$ 個のプロセス p の k-近傍情報の系列である．実行 $E = C_0, C_1, ...$ が与えられると，状況 C_i において，$history_p^k[0]$ は C_i の $view_p^k$ であり，$history_p^k[j]$ は C_{i-j} の $view_p^k$ である．$i < j$ の場合，$history_p^k[j]$ は未定義であり，$\perp$ と表す．

プロセス p における $view_p^k$ を k-近傍情報と表し，$history_p^k[0..\ell]$ を $(k,\ell)-$ 局所履歴と表す．

単調収束性の定義は，既存研究[4] と同様の定義とする．本研究では単調収束性の制約を緩和した出力単調収束性を提案し，その定義を以下に示す．

定義 2.3 (単調収束性) 自己安定アルゴリズム $A(T)$ が，任意の初期状況から開始する任意の実行 $E = C_0, C_1, ..., C_\ell, ..., (\ell = min\{j | C_j \in C_L\})$ 中で，各プロセス p の状態が変化する場合，その出力変数 v_p が $v_{p|C_\ell}$ に変化することを保証するとき，$A(T)$ を単調収束であるという．

定義 2.4 (出力単調収束性) 自己安定アルゴリズム $A(T)$ が，任意の初期状況から開始する任意の実行 $E = C_0, C_1, ..., C_\ell, ..., (\ell = min\{j | C_j \in C_L\})$ 中で，各プロセス p の出力が変化する場合，その出力変数 v_p が $v_p | C_\ell$ に変化することを保証するとき，$A(T)$ を出力単調収束であるという．

定義 2.5 (問題の局所性) 問題 P について単調収束性を実現するためには k-近傍情報と $(k,\ell)-$ 局所履歴が必要である場合，問題 P は $(k,\ell)-$ 局所である．

3. 単調収束性と出力単調収束性の実現可能性

3.1 単調収束性の実現不可能性

補題 3.1 直径が 3 以上のネットワークにおける $(1,0)$-局所でない問題に対して単調収束な自己安定アルゴリズムは存在しない．

証明 1 分散システムでは，隣接しているプロセスの情報しか読むことができず，また，内部変数を用いた計算なしに遠方のプロセスの情報を集めてくることはできない．単調収束性では，デーモンに選択されたプロセスが必ず正しい状態に出力変化しなければならないことから，離れたプロセスの情報を獲得するための内部変数を用いた計算を行うことがなく，隣接プロセスの情報のみで出力を決定しなければならない．近傍情報の定義から，k-近傍情報 $(k > 1)$ が必要なグラフ問題において単調収束性の実現は不可能である．
局所履歴が 1 以上であるグラフ問題については，局所履歴の定義から，過去の近傍情報を記録することが必要である．内部変数を用いれば，過去の近傍情報を記録することは可能であるが，初期状況において内部変数の値が任意であることから，出力変化を行なっていないにもかかわらず，出力変化を行ったという状態をもつプロセスが存在し得る．局所履歴の定義から，このようなプロセスが存在する状況における単調収束性は保証できない．局所履歴が 1 以上であるグラフ問題においても単調収束性の実現は不可能である．以上より，$(1,0)$-局所でないグラフ問題の単調収束性は実現不可能である． □

3.2 出力単調収束性

単調収束性の実現は不可能である．単調収束性では内部変数を用いて計算できないことが原因で実現不可能であることから，本論文では，内部変数を用いた計算を可能にするため，単調収束性の条件を緩和した出力単調収束性を提案する．

出力単調収束性では，各プロセスが高々1 回の出力変化をすることにより，正しい状況へと収束する．ここで注目すべき点は，単調収束性では各プロセスがデーモンによって選択されたとき，必ず正しい状況での出力に変化しなければならないことに対し，出力単調収束性では，各プロセスがデーモンによって選択されたとき，正しい状況での出力に出力変化するための情報がない場合は出力変化をせず，その後再びデーモンに選択されたときに出力変化をしてもよいということである．このように，出力単調収束性では出力変化を保留することが可能であることから，出力変化するまでの間に，内部変数を用いて隣接プロセス以外のプロセスの情報を擬似的に読むことが可能である．

しかし，内部変数を用いた計算が可能である出力単調収束性についても，全域木作成問題，極大独立点集合問題，極大マッチング問題に対して実現は不可能である．以下の 3.3 では，極大マッチング問題について，出力単調収束性が不可能であることを証明する．

3.3 出力単調収束性の実現不可能性

以下では，極大マッチング問題に対して出力単調収束な自己安定アルゴリズムが存在しないことを示す．

証明は以下のアイデアに基づいている．各プロセスはガード付きアクションの実行により自身の状態を変更するが，変更後の状態は変更前の自身の状態，また隣接プロセスの状態（つまり，直接読むことができる変数の値）にのみ依存している．したがって，グラフ G におけるある単調収束な実行で，プロセス p が $view_p^1 = v_p^1$ で出力変数 V_p の値を x に変更するとすれば，p はどのようなグラフにおいても，$view_p^1$ が v_p^1 であれば出力変数の値を常に x に設定する．しかし，上記の問題では局所的な状況が同一であっても，$V_p \neq x$ であるような正しい状況しかないグラフ G' が必ず存在する．つまり，問題の解が N_p^1，またそれらのプロセスの状況のみにから決められないことに原因がある．この結果は[4] においても問題の局所性として議論されている．上記の問題はいずれも $(1,0)$-局所ではないことに注意されたい．

補題 3.2 直径が 3 以上のグラフに対する極大マッチング問題を解く出力単調収束な自己安定アルゴリズムは存在しない．

証明 2 任意のグラフに対する出力単調収束な自己安定アルゴリズム $\mathcal{A}$ が存在するとする．

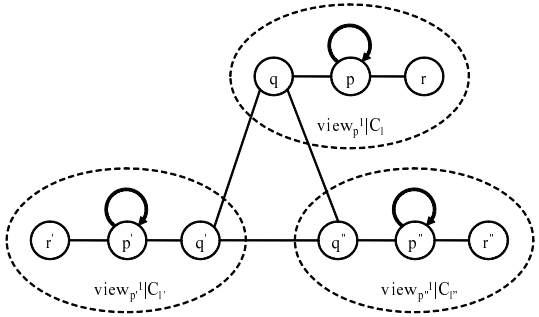

図 1 グラフ G^*

あるグラフ $G = (V, E)$ を想定する．$\mathcal{A}$ を実行すれば，どのような実行 $E = C_0, C_1, \ldots$ も出力単調収束である．実行 E における正当な状況を C_ℓ とする．ここで，C_ℓ において，どの隣接プロセスともマッチングしていないプロセス p が存在し，$m_p|C_0 \neq m_p|C_\ell$ としても一般性を失わない．よって，p は E の接頭辞 $C_0, C_1, \ldots, C_\ell$ において出力変数を変更する．このときの状況遷移を $C_k \to C_{k+1}$ とする．つまり，$view_p^1|C_k$ において，プロセス p は常に有効であり，c デーモンに選択されれば，出力変数を $m_p|C_\ell$ に変更する．

また，同様なプロセス p，p' が存在する他のグラフ G'，G'' が存在するとしても一般性を失わない．ここで，p'，p'' が出力変数を変更する状況遷移をそれぞれ $C_{k'} \to C_{k'+1}$，$C_{k''} \to C_{k''+1}$ とする．

ここで，図 1 のような新たなグラフ G^* を考える．G^* において，p において $view_p^1|C_k$，p' において $view_p^1|C_{k'}$，p'' において $view_p^1|C_{k''}$ であるような状況から始まる実行を考える．このとき，c デーモンが p，p'，p'' を順番に指定すれば，図 1 のようになる．この後，$\mathcal{A}$ のどのような実行においても，q，q'，q'' は出力変数を変更して極大マッチングを作成する．この 3 プロセスのうち，少なくとも 2 プロセスは互いにマッチングを作成する．しかし，残る 1 プロセスは隣接プロセスがどのプロセスともマッチングしていないにもかかわらず，マッチングを形成することができない．すなわち，$\mathcal{A}$ は極大マッチングを形成できない．したがって，このような $\mathcal{A}$ は存在しない．想定したグラフ G^* において，$\mathcal{A}$ は存在しないことは，r，r'，r'' の存在の有無に依存しない．したがって，グラフの直径が 3 以上のネットワークにおいて極大マッチング問題を解く出力単調収束な自己安定アルゴリズムは存在しない． □

4. 提案手法

提案手法では，出力変化回数を抑えるために，リセットの概念を用いる．本研究においてリセットとは，内部変数の値を一定の値に整えることを意味する．本章で提案するアルゴリズムでは，リセットを行い，内部変数の値を整えた後に出力単調収束に問題を解くことで,.各プロセスの出力変化回数が高々2 回となることを実現する．

極大マッチング問題では，既存研究のアルゴリズムにおける全ての計算機における出力変化回数の総和は $O(n^2)$ である．本研究における提案アルゴリズムでは，各プロセスの出力変化回数が高々2 回となることから，全ての計算機における出力変化回数の総和を $O(n)$ にする．

4.1 極大マッチング問題に対する提案アルゴリズム

極大マッチング問題に対してリセットに基づくアルゴリズムを提案する．

提案アルゴリズムの概要を以下に述べる．極大マッチング問題に対する提案アルゴリズムは主に 4 つのアルゴリズムから構成される．1 つ目は矛盾チェックアルゴリズムであり，各プロセスにおける状態を矛盾状態かどうか判別する．2 つ目はリセットアルゴリズムであり，矛盾状態のプロセスのみをリセット完了状態にする局所的なリセットを行う．そして，リセット完了状態のプロセスは，3 つ目の内部極大マッチングアルゴリズムによって，内部変数を使用して極大マッチングを構成し，4 つ目の出力更新アルゴリズムによって内部変数が構成した極大マッチングを出力変数に反映する．

出力変化回数を削減するために，出力変化の有無を記録することが必要である．出力変化の有無を記録するために，与えられた実行 $E = C_0, C_1 \ldots$ と E 中の状況 C_i について，プロセス p が $C_0, C_1, \ldots C_{i-1}$ 中で出力を変更するとき，C_i で p は決定状態であると定義し，出力を変更していないときは未決定状態であると定義する．

極大マッチング問題に対するリセットアルゴリズムでは，プロセス p における出力が隣接プロセス q を指す場合，$m_p = q$ と表すこととする．プロセス p がマッチングに含まれない場合，$m_p = nil$ と表す．また，各プロセス p は内部変数に $decide_p$，$fault_p$，$reset_p$，$pref_p$ を持つ．$decide_p$ はプロセス p の決定状態・未決定状態を表す．$decide_p = 1$ の場合，プロセス p は決定状態であり，$decide_p = 0$ の場合，プロセス p は未決定状態である．$fault_p$ はプロセス p が問題に矛盾しているかどうかを表す．$fault_p = 1$ の場合，プロセス p は問題に矛盾している状態であり，$fault_p = 0$ の場合，プロセス p は問題に矛盾していない状態である．$reset_p$ はプロセス p のプロセスのリセットに関する状態を表す．$reset_p = comp$ はリセットを完了した状態を表し，$reset_p = wait$ はリセットを待機している状態，$reset_p = que$ はリセットの実行を要求された状態を表す．プロセス p において，内部極大マッチングアルゴリズムで極大マッチングを構成する内部変数を $pref_p$ とする．プロセス p における $pref_p$ が指す相手がいない場合，$pref_p = nil$ と表す．

提案アルゴリズムの正当な状況を以下に示す．

$$L = \forall p \in V : ((m_p = q \wedge fault_p = 0 \wedge decide_p = 1 \wedge reset_p = comp)$$
$$\wedge(q : q \in N_p \wedge m_q = p \wedge decide_q = 1 \wedge fault_q = 0 \wedge reset_q = comp))$$
$$\vee((m_p = nil \wedge fault_p = 0 \wedge decide_p = 1 \wedge reset_p = comp)$$
$$\wedge(\forall q : q \in N_p \wedge m_q = r \wedge decide_q = 1 \wedge fault_q = 0 \wedge reset_q = comp)$$
$$\wedge(r : r \in N_q \wedge r \neq p))$$

アルゴリズムの詳細を説明するために，プロセスの状態を正常状態，故障状態，リセット待機状態，リセット要求状態，正常状態 2，リセット完了状態の 6 つに分類する．この 6 つの状態は互いに重複しておらず，全ての内部状態を網羅している．プロセスの状態と内部変数の値の関係を表 1 に示す．

表 1 プロセス p の状態

		$decide_p = 0$	$decide_p = 1$
$reset_p = comp$	$fault_p = 0$	リセット完了状態	正常状態
	$fault_p = 1$		矛盾状態
$reset_p = wait$	$fault_p = 0$	リセット待機状態	
	$fault_p = 1$		
$reset_p = que$	$fault_p = 0$	リセット要求状態	正常状態 2
	$fault_p = 1$		矛盾状態

4.1.1 矛盾チェックアルゴリズム

矛盾チェックアルゴリズムでは，各プロセスにおける状態を矛盾状態かどうか判別する．正常状態，リセット要求状態，正常状態 2 のプロセスに対して，問題に矛盾する場合はこのアルゴリズムによって矛盾状態に，問題に矛盾しない場合は正常状態にする．矛盾を表す論理式を以下に定義する．

アルゴリズム 4.1 矛盾チェックアルゴリズム

プロセス p のアクション

// 正常状態 → 矛盾状態
S_{11} : $decide_p = 1 \wedge fault_p = 0 \wedge reset_p = comp \wedge conf_p$
$\longrightarrow fault_p := 1$
// リセット要求状態 → 矛盾状態
S_{12} : $decide_p = 0 \wedge reset = que \wedge conf_p$
$\longrightarrow decide_p := 1 \wedge fault_p := 1$
// リセット要求状態 → 正常状態 2
S_{13} : $decide_p = 0 \wedge reset_p = que \wedge \neg conf_p$
$\longrightarrow decide_p := 1 \wedge fault_p := 0$
// リセット待機状態 (隣接が全て正常状態) → 矛盾状態
S_{14} : $reset_p = wait \wedge (\forall q : q \in N_p \wedge decide_q = 1 \wedge fault_q = 0 \wedge reset_q = comp)]$
$\longrightarrow decide_p := 1 \wedge fault_p := 1 \wedge reset := comp$
// 正常状態 2(隣接にリセット待機状態なし) → 正常状態
S_{15} : $decide_p = 1 \wedge fault_p = 0 \wedge reset_p = que \wedge \neg conf_p$
$\wedge\neg(\exists q : q \in N_p \wedge reset_q = wait)$
$\longrightarrow reset_p := comp$
// 正常状態 2(隣接にリセット待機状態なし) → 矛盾状態
S_{16} : $decide_p = 1 \wedge fault_p = 0 \wedge reset_p = que \wedge conf_p$
$\wedge\neg(\exists q : q \in N_p \wedge reset_q = wait)$
$\longrightarrow fault_p := 1 \wedge reset_p := comp$

$conf_p = (m_p = q \wedge fault_p = 0 \wedge decide_p = 1 \wedge reset_p = comp$
$\wedge\neg(q : q \in N_p \wedge m_q = p \wedge decide_q = 1 \wedge fault_q = 0 \wedge reset_q = comp))$
$\vee(m_p = nil \wedge fault_p = 0 \wedge decide_p = 1 \wedge reset_p = comp$
$\wedge(\forall q : q \in N_p \wedge m_q = r \wedge decide_q = 1 \wedge fault_q = 0 \wedge reset_q = comp)$
$\wedge(r : r \in N_q \wedge r \neq p))$

矛盾チェックアルゴリズムのガード付きアクションをアルゴリズム 3.1 に示す.

4.1.2 リセットアルゴリズム

リセットアルゴリズムは，矛盾状態のプロセスを未決定状態にするアルゴリズムである．リセットを実行する過程を以下に示す.

(1) 矛盾状態プロセスをリセット待機状態にする（矛盾チェックアルゴリズム）
(2) リセット待機状態プロセスに隣接する全てのプロセスはリセット要求状態をとる
(3) リセット要求状態のプロセスは矛盾チェックアルゴリズムにより自身の状態をチェックする（矛盾チェックアルゴリズム）
(4) 矛盾している場合は 1 に戻る．矛盾していない場合は正常状態 2 となり，正常状態 2 のプロセスに隣接するリセット待機状態のプロセスがリセットされる
(5) リセット待機状態のプロセスは，隣接にリセットされた状態のプロセスをもつ場合，リセットされる

アルゴリズム 4.2 リセットアルゴリズム

プロセス p のアクション

// 矛盾状態 → リセット待機状態
S_{21} : $decide_p = 1 \wedge fault_p = 1 \wedge (reset_p = comp \vee reset_p = que)$
$\longrightarrow reset_p := wait$
// 正常状態 (隣接プロセスにリセット待機状態が存在) → リセット要求状態
S_{22} : $decide_p = 1 \wedge fault_p = 0 \wedge reset_p = comp \wedge (\exists q : q \in N_p \wedge reset_q = wait)$
$\longrightarrow decide_p := 0 \wedge reset_p := que$
// リセット待機状態 (全隣接プロセスがリセット待機状態またはリセット完了状態
// または正常 2 状態) → リセット完了状態
S_{23} : $reset_p = wait \wedge (\forall q : q \in N_p$
$\wedge((reset_q = wait) \vee (decide_q = 1 \wedge reset_q = comp)$
$\vee(decide_q = 1 \wedge fault_q = 0 \wedge reset_q = que)))$
$\longrightarrow decide_p := 0 \wedge reset_p := comp$

リセットアルゴリズムのガード付きアクションをアルゴリズム 3.2 に示す.

4.1.3 内部極大マッチングアルゴリズム

内部極大マッチングアルゴリズムでは，リセット完了状態のプロセスにおいて，内部変数を用いて既存のアルゴリズム[6] を計算し，極大マッチングを内部変数によって構築する．既存アルゴリズムのガード付きアクションをアルゴリズム 3.3 に示し，内部極大マッチングアルゴリズムのガード付きアクションをアルゴリズム 3.4 に示す.

4.1.4 出力更新アルゴリズム

出力更新アルゴリズムでは，内部変数を用いて構成した極大マッチングの結果を出力変数に反映する．出力更新アルゴリズムのガード付きアクションをアルゴリズム 3.5 に示す.

4.1.5 極大マッチング問題に対する提案アルゴリズムの正しさ

プロセスの状態については，正常状態，故障状態，リセット待機状態，リセット要求状態，正常状態 2，リセット完了状態の 6 つに分類する．状態遷移図を図 2 に表す.

補題 4.1 任意の初期状況からはじまる任意の実行 $E = C_0, C_1, \ldots$ にも，やがてそれ以

アルゴリズム 4.3 既存アルゴリズム

プロセス p のアクション

M_p : $pref_p = nil \wedge pref_q = p$
$\longrightarrow pref_p := q$

S_p : $pref_p = nil \wedge (\forall r \in N_p : pref(r) \neq p) \wedge pref_q = nil$
$\longrightarrow pref_p := q$

U_p : $pref_p = q \wedge pref_q \neq p \wedge pref_q \neq nil$
$\longrightarrow pref_p := nil$

アルゴリズム 4.4 内部極大マッチングアルゴリズム

プロセス p のアクション

// M_p が成り立つ場合 → $pref_p$ は q を指す
S_{31} : $decide_p = 0 \wedge reset_p = comp \wedge \neg(\exists q : q \in N_p \wedge reset_q = wait)$
$\wedge(M_p : decide_p = decide_q = 0 \wedge reset_p = reset_q = comp)$
$\longrightarrow pref_p := q$

//S_p が成り立つ場合 → $pref_p$ は q を指す
S_{32} : $decide_p = 0 \wedge reset_p = comp \wedge \neg(\exists q : q \in N_p \wedge reset_q = wait)$
$\wedge(S_p : decide_p = decide_q = 0 \wedge reset_p = reset_q = comp)$
$\longrightarrow pref_p := q$

//U_p が成り立つ場合 → $pref_p$ は nil とする
S_{33} : $decide_p = 0 \wedge reset_p = comp \wedge \neg(\exists q : q \in S_p \wedge reset_q = wait)$
$\wedge(U_p : decide_p = decide_q = 0 \wedge reset_p = reset_q = comp)$
$\longrightarrow pref_p := nil$

//$pref_p$ の指すプロセスが (正常状態または正常状態 2) かつ出力変数が p を指していない場合
// → $pref_p$ は nil とする
S_{34} : $decide_p = 0 \wedge reset_p = comp \wedge \neg(\exists q : q \in S_p \wedge reset_q = wait)$
$\wedge(pref_p = q \wedge decide_q = 1 \wedge (reset_q = comp \vee reset_q = que) \wedge m_q \neq q)$
$\longrightarrow pref_p := nil$

降どのプロセスも状態を変更しない正当な状況 C_ℓ が存在する

証明 3 状況 $C_0, C_1, C_2, \ldots C_{\ell'} \ldots, C_\ell$ を考える．

- $C_{\ell'}$
  - $C_{\ell'} \neq C_\ell$ である
  - $C_{\ell'}$ 以降ガード付きアクションが実行されない

アルゴリズム 4.5 出力更新アルゴリズム

プロセス p のアクション

//内部変数が q(リセット完了状態) とマッチング → 出力は q を指す
S_{41} : $decide_p = 0 \wedge reset_p = comp \wedge pref_p = q \wedge pref_q = p \wedge \neg(\exists q : q \in N_p \wedge reset_q = wait)$
$\longrightarrow m_p := q \wedge decide_p := 1 \wedge reset_p := comp \wedge fault_p := 0$

//内部変数が q(正常状態) とマッチング → 出力は p を指す
S_{42} : $decide_p = 0 \wedge reset_p = comp \wedge pref_p = q \wedge pref_q = p \wedge m_q = p$
$\wedge\neg(\exists q : q \in N_p \wedge reset_q = wait)$
$\longrightarrow m_p := q \wedge decide_p := 1 \wedge reset_p := comp \wedge fault_p := 0$

//全ての隣接プロセスの出力が他のプロセスを指す場合 → 出力は p を指す
S_{43} : $decide_p = 0 \wedge reset_p = comp \wedge (\forall q : q \in N_p \wedge m_q = (r : r \in N_q \wedge r \neq p))$
$\wedge\neg(\exists q : q \in N_p \wedge reset_q = wait)$
$\longrightarrow m_p := nil \wedge decide_p := 1 \wedge reset_p := comp \wedge fault_p := 0$

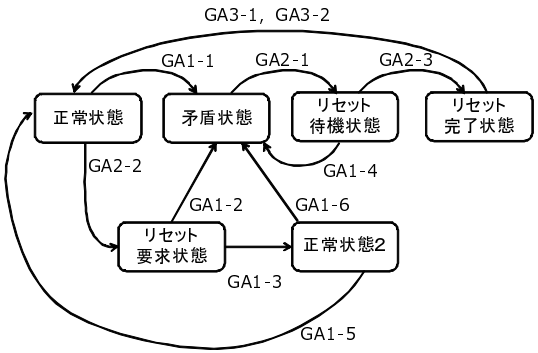

図 2 状態遷移図

背理法により $C_{\ell'}$ が存在しないことを示す．$C_{\ell'}$ が存在すると仮定する．図 2 に示した通り，全て状態に対してガード付きアクションが存在することから，有効なガードを持つプロセスがシステムの中に 1 つでも存在するとき，そのようなプロセスの中の少なくとも 1 つが対応するガードを実行できる．このことにより $C_{\ell'}$ 以降ガード付きアクションが必ず実行され，$C_{\ell'}$ は存在しない．これは仮定に矛盾し，$C_{\ell'}$ が存在しない．全てのプロセスの状態が $C_{\ell'}$ になることなく正常状態に到達することから，どのような状況もやがて C_ℓ に到達する．□

補題 4.2 提案アルゴリズムにおける各プロセスの出力変化回数の上限は 2 である．

証明 4 まず，提案アルゴリズムでは，リセット待機状態，リセット完了状態を経てリセット完了状態に遷移したプロセスは，高々1 回の出力変化でそれ以降状態を変更しない正常状態となることを以下に証明する．

リセット完了状態のプロセスは，内部極大マッチングアルゴリズムによって，内部変数が極大マッチングを構成する．この内部変数が極大マッチングを構成するアルゴリズムは，既存のアルゴリズムを用いていることから，内部変数は必ず極大マッチング問題の正当な状況に収束することが保証できる．さらに，リセット完了状態の 2 つのプロセスの内部変数が互いを指している場合，これらのプロセスの極大マッチングを構成する内部変数はそれ以降変化することはない．

内部変数が構成した極大マッチングにおいて，互いを指し合うプロセスは，出力更新アルゴリズムによって内部変数の値が出力変数に反映されるので，リセット完了状態から高々1 回の出力変化でそれ以降状態を変更しない正常状態となる．

内部変数が構成した極大マッチングにおいて，nil を指すプロセスは，隣接プロセス $\{q \mid q \in N_p \wedge m_q \neq q\}$ がそれ以降状態を変更しない正常状態であるので，S_{43} により状態変化した後，それ以降状態変化することはない．

したがって，リセット待機状態，リセット完了状態を経てリセット完了状態に遷移したプロセスは，高々1 回の出力変化でそれ以降状態を変更しない正常状態となることが示された．

次に，リセット待機状態を経ずにリセット完了状態となったプロセスの出力が 2 回変化する例を示す．初期状況でリセット完了状態のプロセス p が存在し，p の隣接プロセス $\{q \mid q \in N_p\}$ 全ての出力が p 以外のプロセスを指している場合，p は S_{43} により nil に出力変化する場合が存在する．このとき，初期状況は任意であることから q が問題に矛盾している場合は存在する．p が nil に出力変化した後に，やがて問題に矛盾している隣接プロセス q がリセット待機状態に遷移した場合，p は S_{11} によって矛盾状態になる．上記で示したように，矛盾状態のプロセス p は高々1 回の出力変化でそれ以降状態を変更しない正常状態となることから，合計して高々2 回の出力変化でそれ以降状態を変更しない正常状態となる．

以上より，提案アルゴリズムにおける各プロセスの出力変化回数の上限は 2 であることが示された． □

5. むすび

本研究では，出力単調収束性の実現が不可能であることから，各計算機における出力変化を高々2 回に抑えるアルゴリズムを提案した．本研究では，極大マッチング問題を扱った．極大マッチング問題における既存研究のアルゴリズムでは，全ての計算機における出力変化回数の総和が $O(n^2)$ であり，提案アルゴリズムでは，全ての計算機における出力変化回数の総和を $O(n)$ に削減することができた．

参考文献

1) 増澤利光, 山下雅史：適応的分散アルゴリズム, 共立出版 (2010).
2) Kuhl,J.G. and Reddy,S.M.: Distributed fault-tolerance for large multiprocessor systems, ISCA '80 Proceedings of the 7th annual symposium on Computer Architecture (1980), pp. 23–30.
3) Dijkstra,E.W.: Self-stabilizing systems in spite of distributed control, Object Oriented Real-Time Distributed Computing (2008), pp. 363–369.
4) Yamauchi,Y. and Tixeuil,S.: Monotonic stabilization, In proceedings of the 14th International Conference on Principles of Distributed Systems (OPODIS 2010), pp. 475–490.
5) Chattopadhyay,S., Higham,L. and Seyffarth,K.: "Dynamic and self-stabilizing distributed matching", Proceedings of the twenty-first annual symposium on Principles of distributed computing (2002), pp. 290–297.
6) Tel,G.: *Introduction to Distributed Algorithms*, Cambridge University Press (2000).