社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report
2006－HPC－107 (29)
2006／8／1

適応的並列計算を支援するプロトコルの設計と正当性の証明

関谷 岳史[†] 田浦 健次朗[†] 近山 隆[†]

本稿では，プロセス間の接続トポロジーに制限がなく，全プロセスの任意の部分集合が脱退可能なモデルにおいて，プロセスが計算から安全に脱退するプロトコルを提案し，その正しさの証明を示す.「安全に」とはメッセージが脱退してしまったプロセス中や，脱退してしまったプロセス宛の接続中に取り残されない，ということである．本プロトコルではいくつかの仮定の下ではあるが，各プロセスが自律的に高い自由度で脱退のタイミングを決めることができる．実装に基づく実験では，中央サーバを用いた単純な直列化を行ったプロトコルに比べ，32 ノードが同時に脱退するのにかかる時間が約 1/2 であった.

Design and Correctness Proof of A Protocol that Supports Adaptive Parallel Processing

TAKESHI SEKIYA,[†] KENJIRO TAURA[†] and TAKASHI CHIKAYAMA[†]

We consider a protocol which allows participating processes to leave gracefully from computation, and we prove it partly. It works in the models which any connection topology is permitted and any subset of processes can leave. 'Gracefully' means that no message is left in closed connections or in processes that have left. With our protocol, processes can almost autonomously decide to leave although some assertions are needed. Experiment with implementation represents that our protocol has higher performance at speed which multiple processes leave at the same time than a naive serializing protocol which uses central server.

1. はじめに

並列計算や分散計算において，参加プロセスの集合やプロセス間の接続を動的に変化 (再構成) させながら計算を続行することは，資源の変化に適応した計算を行うための基礎である．たとえば広域分散環境や複数の利用者によって共有された環境においては，プロセス群をお互いに近くなるように配置したり，各時点で CPU/ネットワーク負荷が小さいノードに配置する必要がある．並列アプリケーションが実行時に，資源の状態に応じてそれを行うことが出来れば資源選択の柔軟性が高まり，資源の利用効率も向上する．

一方適応的な計算を正しく記述・実装することは容易ではない．実際そのような計算は資源状態によって異なる動作をするため，バグの再現が難しい．また，接続トポロジーの変化，特にプロセスの脱退を行うプロトコルは，通信と通信路の切断との競合によるメッセージの喪失やデッドロックなどの誤りを含みやすい．そのため実践的には，適応的計算は計算モデルやプロセス間の接続トポロジーを単純なもの (マスター・ワーカモデルなど) に限定して行われている．

本論文では，プロセス間の接続トポロジーに制限がなく，全プロセスの任意の部分集合が脱退しうるという一般的なモデルにおいて，プロセスが安全に脱退するプロトコルを提案し，その正しさを証明することを目的とする．このプロトコルでは，各プロセスが 3 節に述べる条件を満たす限り，任意の時点で自律的に脱退を希望でき，脱退を希望したプロセスはそれを確実に完了できる．本プロトコルは，その過程でメッセージを喪失・複製することはない．本論文は，これが可能なための本質的な条件と，その条件の下でそれを行うためのプロトコルを示し，その正しさを部分的に証明する．また，実装に基づくプロトコルの評価実験の結果を示す．

本論文の構成は以下のとおりである． 2 節で関連研究について述べる． 3 節で問題設定を正式に述べ，4 節で提案するプロトコルについて述べる． 5 節で実装に基づく実験結果について述べる．

2. 関連研究

Grid 環境や共有された環境における適応的な並列

† 東京大学
University of Tokyo

計算の必要性は多くの研究において指摘されている[1].適応的な並列計算においては，プロセスを脱退させる際のプロトコルが問題になるが，これに対していくつかの異なるスタンス，アプローチが存在する．

ひとつは，プロセスの移動や増減を，明らかに通信チャネル中にひとつもメッセージが存在しない時点に限定して行う方法であり，実装が単純であることが大きな利点である．科学技術計算などではループを並列実行し，ループの繰り返しの間でバリア同期がとられる場合が多く，このような応用で用いることができる[2),3)]．しかしながら，並列再帰呼び出しや，動的なwork stealingなどの負荷分散方法を用いるプログラム[4)]では，そのような時点は存在しないことが多い．

もうひとつは，メッセージが喪失することを許容し，アプリケーションレベルないし，送受信者間のend-to-endの通信プロトコルでこれに対処させる，というものである．これは，たとえば分散Cilk[5)]で用いられている．このようなスタンスは故障や下の通信レイヤによるメッセージロスを前提としているシステムでは必然かつ自然な方法であり，Ibis[1)]などのシステムでも用いられていると考えられる．

これらの多くの研究に対し，本研究では，各プロセスが非同期に，自由度の大きなタイミングで脱退でき，かつメッセージは決して失われず，複製もされないプロトコルを提案する．ただし，これにはプロセス故障はなく，下の通信レイヤによるメッセージの喪失はないという仮定が必要である．理論的には，故障がないことは，非同期メッセージパッシングシステムでメッセージが失われず複製もされないことを保証するための必要条件であり[6)]，この仮定は妥当である．実用的には，故障の存在する環境で本プロトコルを実行すれば，メッセージの喪失はすべて故障に起因するものであると仮定できる点が有用である．

3. 問題設定

3.1 基本的な仮定とプロトコルの性質

プロセス間は信頼性のある通信チャネルで接続されており，そのトポロジーは任意である．各通信チャネルはメッセージの順序を保存する (FIFO)．メッセージには本プロトコルにしたがって送信されるもの (プロトコルメッセージ) と，それとは独立に送信されるもの (ユーザメッセージ) がある．ユーザメッセージはあるプロセスによって送信され，受信したプロセスによって消費される．計算を実行中いくつかのプロセスが，適当な時点で脱退を「希望」し，脱退を「完了」するためのプロトコルを開始する．脱退を完了したプロセスはすべての活動を停止する．すなわち，ユーザメッセージやプロトコルメッセージを送信・受信しない．脱退を完了していないプロセスは任意の時点で新しいユーザメッセージを生産し，送信しうる．脱退を希望したプロセスはそれ以降ユーザメッセージを消費しないかもしれず，そのようなメッセージは，脱退を希望していないプロセスへ転送する．本論文では簡単のため，プロセスや，プロセス間チャネルの動的な追加はないものと仮定している．

脱退は，基本的に各プロセスが自律的に，任意の時点で希望してよい．しかし，極端な場合として全プロセスが脱退を希望した場合など，どんなプロトコルであっても成功し得ない場合が存在する．したがって脱退を希望するプロセスの集合や，そのタイミングについて，何らかの仮定をおく必要がある．本論文では以下を仮定する．

> 脱退を希望していないプロセスとそれらの間の接続からなるグラフを考える．このとき，計算中のいかなる時点においても，このグラフは連結である—(*)

その仮定の元で，我々の提案するプロトコルは以下の性質を満たす．

Safety: メッセージが失われることはない．つまりプロセスは，自分が持つ接続が，自分向きのメッセージを保持している間に脱退を完了することはない．また，メッセージを保持したまま脱退を完了することもない．

また，メッセージが複製されることもない．

Progress: 脱退を希望したプロセスは，有限ステップ (状態遷移) 後に脱退を完了することが出来る．とくに，プロトコルはデッドロックやライブロックを引き起こさない．

プロトコルの特徴は，プロセス間のチャネルやプロセスは信頼性があると仮定し，そのもとで，確認応答，再送，タイムアウトなどを用いずに，メッセージが失われないことを保証している点である．実践的な言葉で述べると，プロセス間のチャネルとしてTCPのような信頼性のあるプロトコルを用い，その上で，「正しいタイミングで」TCPの接続を閉じるプロトコルを提案していることに相当する．メッセージ送信は単にTCP接続上を，その信頼性を仮定して送るのみで，TCP実装の性能を容易に引き出すことが出来る．

3.2 自律的な脱退を許すプロトコルの本質的な限界

さて，そもそも上で述べた仮定 (*) を満たすためにもなんらかの協調動作が必要なので，本プロトコルは，各プロセスが完全に任意の時点で，独立に脱退を希望

することを許すわけではない．本論文では，提案するプロトコルとは別の何らかの条件によって，上記 (*) は満たされるという仮定のもと，上で述べた性質を証明する．

実際にどのようにして仮定 (*) を満たすかについては，トポロジーや一度に脱退するプロセス集合に関する事前知識に応じていくつかの方法がある．また，上で述べた仮定は理論的にも妥当のものであると考える．実際，仮定 (*) が満たされず，かつ,「脱退を希望したプロセスは必ず脱退できる」というプロトコルは，残されたプロセスを複数の連結成分に分断する可能性がある．ひとたびこの状態に陥れば，もはや任意のプロセス集合が，メッセージを失うことなく脱退できないことは明らかである (ひとつの連結成分の全プロセスが脱退を希望した場合)．

つまり，メッセージを喪失せず，かつ脱退を希望したプロセスが必ず脱退できることを保証するプロトコルは，以下のどれかを選択する必要がある．

(1)　なんらかの協調動作によって，脱退を希望していないプロセスが連結でなくなるような状態に決して到達しないことを保証する．

(2)　脱退を希望していないプロセスが連結でなくなるような状態に到達した場合，新しくチャネルを追加して連結性を回復する．

(3)　脱退を希望していないプロセスが連結でなくなるような状態に到達した場合，いくつかのプロセスを，その場の命令伝播や事前合意 (このプロセスが脱退したら自動的にこれらも脱退する，など) によって，強制的に脱退させる．具体的には，ひとつの連結成分のみが残るように，他の全ての連結成分の，全てのプロセスを，早急に脱退させる．

選択肢 3 はたとえば，プロセス間の接続をツリーにした際に，中間ノードの脱退開始にともなって，その下のノードも全て脱退するように強制することなどに相当する．我々のプロトコルはこのような強制的な脱退方針を，プロトコルの本質的な部分は一切変更することなく容易に組み込むことが可能であり，5 節の実装ではこの方法が実装されている．

4. 安全な脱退を支援するプロトコル

4.1 プロトコルの詳細

詳細なアルゴリズムを示したものが **図 1** である．変数には以下のようなものがある．

- p, q: ポート
- m : メッセージ
- *leave* : 脱退フラグ (脱退を希望しているとき 1，そうでないとき 0)
- **parent**[i] : i に対して親のプロセスとつながっているポート
- **children**[i] : i に対して子のプロセスとつながっているポートの集合
- M : メッセージの一時保管場所

ポートとはチャネルの終端点のことであり，チャネルは二つのポートの間を結んでいる．ポートの状態は **open** と **closed** の 2 種類である．

また，メッセージにはユーザメッセージの DATA(m)，プロトコルメッセージの SEEKROOT，ROOT[i]，CANCEL[i]，ACK がある．

プロトコルの基本アイディアは脱退を希望したプロセスとその隣接プロセスで，脱退を希望していないプロセスを root とした，tree 構造を作ることである．tree の leaf のプロセスから順に，ユーザメッセージを root に向かって転送 (押し付け) しながら脱退していく．このようにするとユーザメッセージは脱退するプロセス中に取り残されることなく，root のプロセスに集められる．しかし，この基本アイディアだけでは，一度 root になったプロセスが脱退を希望したときにうまくいかない場合があることが分かったため，複数の root(すなわち **parent**[]) を保持しておき，いずれかの root を選んでそこに向かってメッセージを転送する．

脱退の流れを簡単に示すと次のようになる．脱退を希望したプロセス ($leave = 1$ のプロセス) は SEEKROOT を隣接するすべてのプロセスに対して送る．SEEKROOT を受け取った脱退を希望していないプロセス ($leave = 0$ のプロセス)i は ROOT[i] を折り返し送る．ROOT[i] を受け取った $leave = 0$ のプロセスは受け取ったポートを **parent**[i] とし，それ以外のポートすべてに ROOT[i] を送り，**children**[i] に加える．**children**[i] のポートからはいずれ ROOT[i] もしくは ACK が帰ってくるので，そのポートは **children**[i] からはずし，場合によっては **closed** にする．そして，**children**[i] $= \emptyset$ になると i の tree からの脱退 (Event close_leaf) が生じるか，もしくは完全な脱退 (Event finish_leaving) の準備が完了したことになる．プロトコルメッセージ CANCEL[i] は i の tree の root のプロセスが ROOT[i] を送ってしまった後になって，脱退を希望したときに送り出すことで一度構築された i の tree を取り消すメッセージである．

4.2 プロトコルの正当性の証明

3 節で述べたふたつの満たすべき性質のうち，前者のみ証明が完成している．後者の証明については今後

```
01 Event user_send:
02   user generates a new message m ⇒
03     if p ∉ children[id] and p.state = open:
04       p.send(DATA(m))

05 Event forwarding:
06   #M > 0 and p ∉ children[id]
     and p.state = open:
07     M := M − {m}
08     p.send(DATA(m))

09 Event want_to_leave:
10   leave = 0 ⇒
11     leave := 1
12     forall p (p.state = open):
13       if p ∈ children[id]:
14         p.send(CANCEL(id))
15       else:
16         p.send(SEEKROOT)

17 Event user_receive:
18   received(DATA(m)) from p ⇒
19     if leave = 0:
20       deliver DATA(m) to user
21     else:
22       M := M + {m}

23 Event receive_SEEKROOT:
24   received(SEEKROOT) from p ⇒
25     if leave = 0:
26       p.send(ROOT(id))
27       children[id] := children[id] + {p}

28 Event receive_CANCEL:
29   received(CANCEL(i)) from p ⇒
30     if p = parent[i]:
31       p.send(CANCEL(i))
32       parent[i] := NULL
33       forall q ∈ children[i]:
34         q.send(CANCEL(i))
35     else if p ∈ children[i]:
36       children[i] := children[i] − {q}
37     else if id = i:
38       p.send(CANCEL(i))

39 Event receive_ROOT:
40   received(ROOT(i)) from p ⇒
41     if leave = 0 and i ≠ id
42       p.send(ROOT(i))
43     else if p ∈ children[i]:
44       children[i] := children[i] − {p}
45       if ∀j(q ∉ children[j] and q ≠ parent[j])
46         q.send(ACK)
47         q.state := closed
48     else if i ≠ id and leave = 1:
49       parent[i] := p
50       forall q (q ≠ p):
52         q.send(ROOT(i))
51         children[i] := children[i] + {q}

53 Event receive_ACK:
54   received(ACK) from p ⇒
55     forall i (p ∈ children[i]):
56       children[i] := children[i] − {p}
57     p.state := closed

58 Event close_leaf:
59   ∃i, j (parent[i] ≠ NULL
     and parent[j] ≠ NULL
     and children[i] = {})⇒
60     parent[i].send(ROOT(i))
61     p := parent[i]
62     parent[i] := NULL
63     if ∀i(p ∉ children[i] and p ≠ parent[i])
64       p.send(ACK)
65       p.state := closed

66 Event finish_leaving:
67   (leave = 1) and (#M = 0)
     and ∃i (parent[i] ≠ NULL and children[i] = {}
     and ∀j ≠ i (parent[j] = NULL)) ⇒
68     parent[i].send(ACK)
69     parent[i].state := closed
70     parent[i] := NULL
```

図 1 プロトコルの詳細
Fig. 1 Our Protocol

の課題とし，ここでは証明なしで Theorem を示すにとどめる．

Lemma 1. プロセス A, B のポート p_A, p_B は直接チャネルで繋がっているとする．ある i が存在して，$p_A = \texttt{parent}[i]$ ならば，$p_B \in \texttt{children}[i]$ である．

Proof. parent[i] にポートが代入されるのは Event receive_ROOT 中の 49 行目のみである．この ROOT[i] は 26 行目，または 52 行目で送られたものであるので，ポートは直後に $p \in \mathit{children}[i]$ になる．よって，$p_A = \texttt{parent}[i]$ になった時点では $p_B \in \texttt{children}[i]$ であることが分かる．次に，$p_A = \texttt{parent}[i]$ が保たれている間は $p_B \in \texttt{children}[i]$ が保たれることを示す．$p_B \in \texttt{children}[i]$ でなくなる可能性があるのは以下の場合である．

- (line 36) この行が実行されるには p_A が CANCEL[i] を送信する必要があるが，それは 31 行目のみで，直後に $p_A = \texttt{parent}[i]$ でなくなる．
- (line 44) この行が実行されるには p_A が ROOT[i] を送信する必要があるが，それは 60 行目のみでやはり，直後に $p_A = \texttt{parent}[i]$ でなくなる．
- (line 56) この行が実行されるには p_A が ACK を送信する必要があるが，それは 64 行目と 68 行目でいずれの場合も，直後に $p_A = \texttt{parent}[i]$ でなくなる．

以上から示された． □

Lemma 2. ユーザメッセージは closed なポートにやってくることはない．

Proof. ポートが closed になるには以下の場合があるので各場合について調べる．

- (line 47) Lemma 3 より示される．
- (line 57) ACK を送るのは 46, 64, 68 行目であるが，いずれの場合もそのあとポートを closed にしている．closed のポートからメッセージが送信されることはないため，ACK を受け取った後にはなにもメッセージが来ないことが分かる．
- (line 65, line 69) Lemma 1 よりこのポートと対になるポート q は children[i] に含まれている．また，ACK を受け取ったあとには closed になるので q からユーザメッセージが送られることはない．

以上から示された． □

Lemma 3. 47行目のポート p' にはメッセージはやってこない．

Proof. 47 行目では直前に ROOT[i] を受け取っているので，対になるポート p が ROOT[i] を送ったあとにユーザメッセージを送らないことを示せばよい．

- (line 26, line 52) 直後に $p \in \mathit{children}[i]$ になるため，CANCEL(i) を p' から受け取らない限りユーザメッセージは送られない．
- (line 42) ここで受け取った ROOT(i) の前には SEEKROOT を受け取っているはずである．$\mathit{leave} = 0$ であるので，Event receive_SEEKROOT により，$p \in \mathit{children}[i]$ である．よって，CANCEL(i) を受け取らない限りユーザメッセージは送られない．
- (line 60) 47 行目では $p' \notin \texttt{children}[i]$ であるので Lemma 1 より $p \neq \texttt{parent}[i]$ である．よって，$\exists j(p \in \texttt{children}[j])$ であれば，ユーザメッセージは送られず，かつ，いずれ p' の送った ACK (46 行目) が届くため，ユーザメッセージは送られないまま closed になる．もし $\forall j(p \notin \texttt{children}[j])$ であれば，p は 65 行目で closed になるため，やはりユーザメッセージは送られない．

以上より示された． □

Theorem 1 (Safety)**.** ユーザメッセージはプロトコル中で失われたり，複製されたりしない．失われないとはすなわち，ユーザメッセージがチャネルに取り残されたり，隣接するチャネルがすべて closed なポートであるようなプロセス (脱退を完了したプロセス) に取り残されることはない．

Proof. Lemma 2 より，ユーザメッセージがチャネル中に取り残されることはない．また，Event finish_leaving は $\#M = 0$ でないと生起しないため，脱退を完了したプロセス中に取り残されることもない．また，いずれのユーザメッセージも一度のみ送られるので，ユーザメッセージが複製されることもない． □

Theorem 2 (Progress)**.** $\mathit{leave} = 1$ のプロセスはいずれ *Event finish_leaving* が生起する．

5. 実装と評価

5.1 脱退にかかる時間

提案手法の特徴として脱退するタイミングの自由度が高く，複数の脱退を高い並列度で実行可能なことが上げられる．そこで，naive な直列化を行って複数プロセスの脱退を調停するプロトコル (排他制御プロトコル) と，本論文で提案したプロトコルで，脱退にかかる時間の比較実験を行った．実験環境には istbs クラスタ (Xeon 2.4GHz (70 nodes) + Xeon 2.8GHz

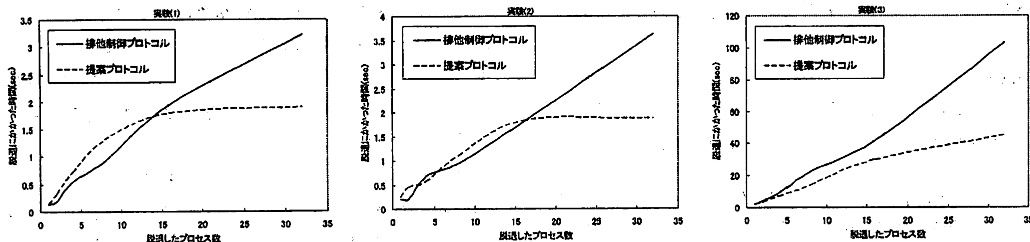

図 2 脱退するプロセス数と脱退にかかる時間
Fig. 2 Number of Leaving Nodes and Time Taken to Leave

(122 nodes)) 上の 64 ノードを用いた.

実験内容は次のとおりである. 各プロセスはランダムに選んだ 5 つのプロセスへコネクションを張る. 次に, $n(=1,2,4,8,16,32)$ プロセスが同時刻 (各マシンの時計に基づくため, 多少の誤差がある) に脱退を希望する. 脱退を希望してから, 完了するまでの時間を各プロセスで測定し, もっとも時間がかかったプロセスについてその値を測定値とする. 本プロトコルの実装では, 脱退を希望するとプロトコルに従いプロトコルメッセージをやり取りし, 脱退を完了する. 一方, 排他制御プロトコルの実装では, 脱退を希望するとあらかじめ決めておいた中央サーバに問い合わせを行い, 脱退の許可がでた時点で通信路上・プロセス中にメッセージがないことを簡単なプロトコルで確認し, 脱退する. 中央サーバは同時にひとつのプロセスが脱退を行うように調停する. 実験は各 n について 20 回ずつ行い, 平均値を求めた. 実験は (1) 通信路にまったくメッセージがない場合, (2) 各通信路に 2Byte のユーザメッセージが平均 2 つある場合, (3) 通信路にメッセージはないが, 脱退するプロセスは 10MByte の計算状態を他のプロセスに送信する必要がある場合をそれぞれ行った.

実験結果は図 **2** のようになった. 排他制御プロトコルのほうでは脱退するプロセスが増えるに連れて, 線形に時間も増加しているのに対して, 本プロトコルでは増加の割合を小さく抑えられている. $n=32$ プロセスのときで約半分程度の時間で脱退が完了している. これは排他制御プロトコルでは同時にひとつのプロセスのみ脱退を許されるため, $O(n)$ の時間がかかるのに対し, 本プロトコルでは同時に複数のプロセスが脱退を行えるためである.

また, 本実験中で脱退にともなうメッセージの喪失は起きなかったため, この実験の環境・接続トポロジーにおいてはプロトコルの正しさが実験的に示せた.

6. おわりに

本稿では 3 節で述べた仮定のもとではメッセージの喪失なく安全な脱退を可能にするプロトコルを提案し, その正当性を部分的に証明した. また, 実装に基づく実験によって多数のプロセスが同時に脱退した場合の脱退にかかる時間を評価し, 排他制御プロトコルに比べて高い性能を示すことを確認した.

今後の課題として, 本プロトコルの上にアプリケーションを構築し, その評価を行うことがあげられる.

参考文献

1) Wrzesinska, G., van Nieuwpoort, R., Maassen, J. and Bal, H. E.: Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid., *Proc. of 19th International Parallel and Distributed Processing Symposium*, Denver, CA, USA (2005).
2) 松原正純, 鈴木和宏, 勝野昭: 自律コンピューティングに向けた HPC 向け動的負荷分散機構, 先進的計算基盤システムシンポジウム SACSIS (2003).
3) Weatherly, D.B., Lowenthal, D.K., Nakazawa, M. and Lowenthal, F.: Dyn-MPI: Supporting MPI on a Nondedicated Cluster of Workstations, *IEEE/ACM Supercomputing 2003 (SC '03)* (2003).
4) Mohr, E., Kranz, D. A. and Halstead, Jr., R. H.: Lazy Task Creation: A Techinque for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).
5) Blumofe, R. D. and Lisiecki, P. A.: Adaptive and Reliable Parallel Computing on Networks of Workstations, *Proc. of the USENIX Annual Technical Conference*, pp. 133–147 (1997).
6) Tel, G.: *Introduction to Distributed Algorithms*, Cambridge Univ Pr (Txp) (2001).