ブロードキャストとWTC方式を用いた分散プロセス制御方式†

六沢一昭†† 市吉伸行†††

本稿では，メッセージをブロードキャストすることによって，特定のプロセス群の強制終了，実行の停止／再開，状態の変更を行い，WTC方式の応用によりその完了を確認する方式を述べる．非同期メッセージ通信を行う分散環境では，送信はされたがまだ受信されていないメッセージが存在するため，大域的な状態の制御が困難である．WTC (Weighted Throw Counting) 方式は，プロセッサ内だけでなくネットワーク中のプロセスにも重みを持たせ重みの合計を制御プロセスが把握することにより，終了検出を行うものである．この方式では到着確認メッセージや観測を繰り返す必要がないため終了が効率よく検出できる．WTC方式を利用し「プロセスの存在するPEを制御プロセスが把握する」ことにより特定のプロセス群の状態制御を行う方式が提案されているが，「プロセスの存在を伝えるメッセージが必要」「メッセージの追い越しのある環境ではコストが高くなる」などの問題があった．本方式では，メッセージのブロードキャストと空のサブプールの導入によってこれらの問題を解決した．プロセスが広範囲のPEに渡るような分散実行に本方式は適している．

1. はじめに

本稿では，メッセージをブロードキャストすることによって，特定のプロセス群の「強制終了」「実行の停止／再開」「状態の変更」を行い，WTC方式の応用によりその完了を確認する方式を述べる．

たくさんのプロセス群を分散環境で実行する場合，特定のプロセス群の終了検出や強制終了，実行の停止／再開，状態の変更は基本的な処理である．しかしながら非同期メッセージ通信によって処理が行われる環境では「送信はされたがまだどのプロセッサにも受信されていないメッセージ」が存在する．このメッセージの存在は大域的な状態の制御を困難にさせている．

WTC方式[1]は，並列GCの方式である Weighted Reference Counting 方式[3),4)]をプロセス管理に応用した終了検出方式であり*，到着確認メッセージを用いる方式[10)]や観測を繰り返す方式[11),12)]とは異なった方式によりプロセス群の終了を効率良く検出することができる**．文献 1)，7)，8) には，WTC方式を利用し「プロセスの存在するPEを制御プロセスが把握する」ことにより強制終了[1),7)]および実行の停止／再開[8)]を行う方式が述べられているが，以下の問題点があった．

- プロセスの存在を伝えるメッセージが必要．
- プロセスの存在するPEを記憶することが必要．
- メッセージの追い越しのある環境ではコストが高くなる．

本稿では「プロセスの存在するPEを把握する必要のない」方式を述べる．まず計算モデルを定義し，本方式で利用しているWTC方式の解説を簡単に行い，「プロセスの存在するPEを把握する」方式の概略を説明する．そして本方式による強制終了，実行の停止／再開，状態の変更方式を述べ，最後に「プロセスの存在するPEを把握する」方式との比較を行う．

2. 計算モデル

本稿で述べる方式は「複数のプロセス群（プロセスプール）が同時に実行されている分散環境」に適用することを意図している．この分散環境は以下のようにモデル化される（図 1）．

- 有限個のプロセッサ (PE) が互いに結合されている．通信は非同期なメッセージによって行う．
- システムには有限個のプロセスプールがあり，異なったIDを持つ．プロセスプールは1つの制御プロセスと有限個の子プロセス（以下，“プロセス”と略す）からなる．
- プロセスには実行中と停止中の2つの状態がある．実行中のプロセスは，同じIDを持つ新しいプロセスや新しいIDを持つ新しいプロセスプールをいつでも生成でき，いつでも自発的に終了しうる．それに属するすべてのプロセスが終了するとプロセスプールは終了する．
- あるPEに存在する同じIDを持ったプロセス群

† A Scheme for State Change in a Distributed Environment Using Broadcast and Weighted Throw Counting by KAZUAKI ROKUSAWA (Systems Laboratory, Oki Electric Industry Co., Ltd.) and NOBUYUKI ICHIYOSHI (Institute for New Generation Computer Technology).

†† 沖電気工業(株)総合システム研究所
††† (財)新世代コンピュータ技術開発機構

* GC方式から終了検出方式の導出は文献 5) が述べている．

** 表現形式は異なるが本質的には同じ方式を文献 2) が述べている．

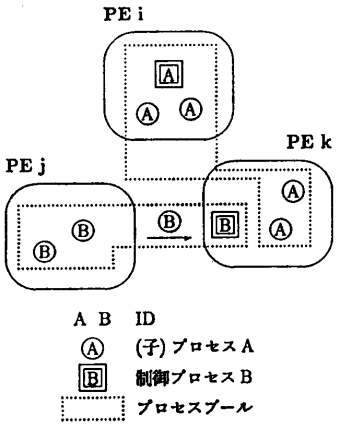

図 1 計算モデル
Fig. 1 Computation model.

について，その PE は終了の検出，強制終了，停止（停止中にする），および再開（実行中にする）ができる．

- 制御プロセスと各 PE（制御プロセスが存在する PE も含む）は互いにメッセージ通信することができる．例えば PE は自 PE 内に存在する特定 ID を持ったプロセス群の終了を伝えるメッセージを同じ ID を持つ制御プロセスへ送ることができる．また制御プロセスは自分と同じ ID を持つプロセスを強制終了させるメッセージを PE へ送信することができる．
- 負荷分散などのために PE は実行中のプロセスを他の PE へ送信することがある．「プロセスを送信する」とは「プロセスを運ぶメッセージを送信する」ことである．停止中のプロセスを送信することはできない．送信されてから受信されるまでの時間は不定である．送信されたがまだ受信されていないメッセージを“通信中のメッセージ”と呼び，そのメッセージが示すプロセスを“通信中のプロセス”と呼ぶ．

図1は，2つのプロセスプール（AとB）の存在する状態を示している．プロセスプールBを構成するプロセスのひとつは通信中のプロセスである．矢印(→)は通信中のメッセージと送信の方向を示す．

3. WTC 方式

ここでは WTC 方式を簡単に解説する．この方式は疎結合型並列計算機マルチ PSI[13]上の並列論理型言語 KL 1[14] 処理系実装[15]に用いている．

同一 PE に存在する同じ ID を持つプロセスの集合はサブプールを構成する．例えば図1で **PEi** 内に存在する2つのプロセスはひとつのサブプールを構成する．PE はその中に存在するサブプールを把握し，プロセスを受信すると同じ ID を持つサブプールに加える．同じ ID のサブプールが存在しない場合は新しく生成する．

WTC 方式は制御プロセスとサブプールおよび通信中のプロセスに“重み”* を持たせる．制御プロセスの重みは負の整数，サブプールおよび通信中のプロセスは正の整数である．そして『重みの合計がゼロ』を満たすように重みの割付けを行う．この結果サブプールと通信中のプロセスの重みの合計は制御プロセスによって把握され，すべてのプロセスが終了した時にのみ制御プロセスの重みがゼロになる．

PE がプロセスを他の PE へ送信する場合はそのプロセスに適当な重みを割り付けサブプールの重みはその分だけ減らす．送信されたプロセスの重みとサブプールの新しい重みはどちらも正の整数であり2つの合計は送信前のサブプールの重みに等しい．送信されたプロセスを PE が受信した場合はその重みを同じ ID を持つサブプールへ加える．同じ ID のサブプールが存在しない場合はサブプールを新たに生成しプロセスの重みをサブプールの重みの初期値とする．

属するプロセスがすべて終了したことを PE が検出するとそのサブプールを終了させ終了を示すメッセージ（%terminated）を同じ ID を持つ制御プロセスへ送る．%terminated は終了したサブプールが保持していた（正の整数の）重みと ID を運ぶ．%terminated を受信すると，制御プロセスは運ばれてきた重みを自分の（負の整数である）重みに加える．この操作により重みがゼロになったならば，そのプロセスプールに属するすべてのプロセスが終了しそのプロセスプールに係わる通信中のメッセージも存在しないことが検出される．

サブプールの重みが1になると2つの正の整数への分割が不可能なので PE はそのサブプールのプロセスを送信することができない．この場合，重みを要求するメッセージ (%request) を制御プロセスへ送信する．制御プロセスは %request を受信すると重みを供給するメッセージ（%supply）を返信する．%supply を受信する，あるいはプロセスの受信により重みが2以上になるとプロセスの送信が再開される．

図 2 は，ある ID を持つプロセスプールにおけるプ

* 文献 2) は“重み (weight)”を“credit”と表現している．

ロセスの送受信とサブプールの終了を示している．1) は **PEi** がサブプール i 中のプロセスを **PEk** へ送信した状態を示している．送信するプロセスには重み 80 を割り付けサブプールの新しい重みは 450 になった．送信前のサブプールの重みは 530 である．2) は 1) に続く状態を示している．**PEk** はプロセスを受信したが該当するサブプールが存在しなかったため，受信プロセス (のみ) を含む重み 80 のサブプールを生成した．また **PEj** のサブプールが終了し，サブプールが保持していた重み 270 を %terminated で制御プロセスへ返却した．

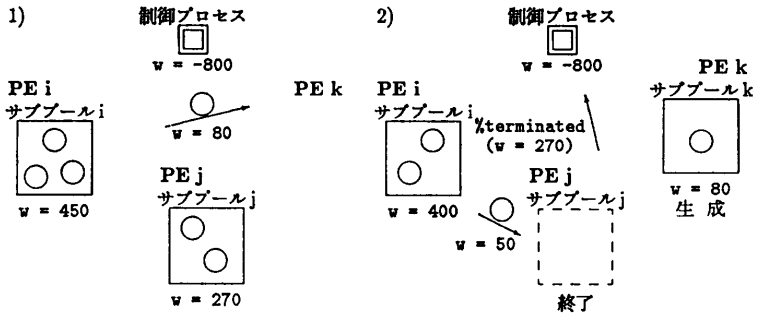

図 2 WTC 方式
Fig. 2 The WTC scheme.

終了検出に関しては多くの方式が提案されており，代表的なものに到達確認メッセージを用いる方式[10]と観測を繰り返す方式[11),12)]がある．本稿が仮定している計算モデルにこれらの方式を適用すると，前者は

- プロセスの送信数と同数の到着確認メッセージが必要になる．
- サブプールの終了が他のサブプールに依存する．

後者は

- 終了しているか否かの観測を何回でも繰り返してしまうおそれがある．
- 観測の間隔を小さくすると無駄な観測が増大し，間隔を大きくすると終了検出の遅れが大きくなる．

という問題がある．これに対して WTC 方式は以下の利点を持っている[9]．

- 到着確認メッセージが不要．
- サブプールは他のサブプールに依存することなく終了できる．
- プロセスの送受信そのものが終了検出操作を含んでいるため“観測”を行う必要がない．
- 終了検出の遅れは %terminated を送受信する時間だけで済む．

最適な重み割付け方法は応用に依存するが，前述した KL 1 処理系の実装では以下の戦略を採用している．

送信するプロセスには 2^{10} (サブプールの重み $\geq 2^{11}$ の場合) あるいはサブプールの重みの半分 (サブプールの重み $< 2^{11}$ の場合) を，%supply には 2^{20} を割り付ける．

一度 %supply を受信すると重みの補給を受けることなく 2^{10} 回程度プロセスを送信することができる．このため重みの要求はたとえ起きても高々 1 回ですむ．

4. プロセスの存在する PE を把握する方式と問題点

ここでは文献 1), 7), 8) が述べている「プロセスの存在する PE を制御プロセスが把握する」方式の概略を説明し問題点を指摘する．

4.1 方式の概略

サブプールを生成したら生成を伝えるメッセージ (%ready) を制御プロセスへ送信する．%ready を受信すると制御プロセスは送信元 PE を記憶し，%terminated を受信すると削除する．記憶している PE にはサブプールの存在することが期待される．

制御プロセスは以下の操作によって強制終了，実行の停止／再開を行う．

(1) 記憶している PE へのみ「強制終了などを起こすメッセージ (これを“制御メッセージ”と呼ぶ)」を送る．制御メッセージには重みを付け，制御プロセスの重みはその分だけ減らす．

(2) 処理中に %ready を受信したならばやはりその送信元 PE へ重み付きの制御メッセージを送る．

制御メッセージを受信すると PE は以下のいずれかの処理を行う．

(3a) サブプールが存在するならば制御メッセージが示す処理を行い，制御メッセージの重みとサブプールの重みの合計を制御プロセスへ送り返す．

(3b) サブプールが存在しない場合は制御メッセージの重みの返却のみを行う．

処理の開始時点で制御プロセスが把握していたサブ

プールは(1)によって処理され，開始後に検出したサブプールは(2)によって処理される．(3a) の後にサブプールが終了するとそのサブプールに関するすべての情報が PE から消滅する．この PE にプロセスが到着し再びサブプールが生まれると %ready が送信され(2)の制御メッセージの返信によって処理がなされる．

通信中のプロセスと同様に制御メッセージも重みを持つため通信中の制御メッセージを残したままプロセスプールが終了してしまうことはない．

4.2 問題点

この方式には「サブプールの生成を伝える」ことに起因する以下の問題点がある．

- サブプールの生成を伝えるメッセージ (%ready) が必要．
- 制御に必要な制御メッセージの数はプロセスのふるまいに依存する．最悪の場合は送受信されるプロセスと同じ数だけ %ready が送信され，それと同数の制御メッセージが送信されてしまう．
- 制御プロセスは PE ごとに情報を持ち %ready および %terminated の受信時に更新することが必要．
- メッセージの追い越しのある環境では，制御プロセスが %ready を受信する前に %terminated を受信したり，%terminated を (あるいは %ready を) 受信することなく複数の %ready を (あるいは %terminated を) 受信することが起こりうる．これに対処するには %ready と %terminated の受信数の差を把握することが必要であるが，これを実現するには PE と同じ数のカウンタが必要になるためコストが高くなる．

5. プロセスの存在する PE を把握しない方式

ここでは 4.2 節で述べた方式の問題点を解決する新しい方式を提案する．この方式はプロセスの存在する PE を制御プロセスが把握する必要がなく，またメッセージの追い越しの有無に係わらず全く同じ方式が適用できる．まず，サブプール生成の伝達を不要にする手段を考察し，次に強制終了，実行の停止／再開，状態の変更方式を述べる．

5.1 サブプール生成の伝達を不要にするには

サブプール生成の伝達を不要にできれば 4.2 節で述べた問題は起こらない．しかし伝達をやめると 2 つの問題が新たに生じてしまう．

まず，制御プロセスにはサブプールの存在する PE がわからない．このため制御メッセージを送るべき PE を特定することができないが，これはメッセージのブロードキャストによって解決できる．

もう 1 つの問題は，制御メッセージを受信し処理を行った後にサブプールが終了した場合のことである．そのような PE にプロセスが到着しても以下の理由により何も処理が行われない．

- サブプールの終了によって制御メッセージの示す処理内容は消滅している．
- 制御プロセスに生成が伝わらないため制御メッセージも送信されない．

この問題は“空のサブプール”の導入によって対処することができる．サブプールが終了すると重みは返却するが，重みゼロのサブプール (これを“空のサブプール”と呼ぶ) を残し処理内容も記憶し続ける．また対応するサブプールの存在しない PE へ制御メッセージが到着した場合も空のサブプールを生成し処理内容を記憶する．空のサブプールがプロセスを受信すると，記憶している処理を行う．この結果，制御メッセージを 1 回受信するとメッセージの示す処理が PE に保持され，受信するすべてのプロセスに対して処理の行われることが保証される．

空のサブプールの存在は制御プロセスの重みに影響しない．このためプロセスプールの終了が検出されても一般に空のサブプールが PE に残っており，これらの消去が必要になる．制御プロセスは終了を検出したら「空のサブプールを消去するメッセージ (%forget)」をブロードキャストする．%forget を受信すると PE は (存在するならば) 空のサブプールを消去し，到着確認メッセージ (%ackForget) を返信する．すべての PE からの %ackForget の受信によってプロセスプールの終了は完了する．

5.2 強制終了

強制終了とは，制御プロセスが制御メッセージを送ることにより同じプロセスプールに属するすべてのプロセスを終了させることである．プロセスプールの終了は 3 章で述べた WTC 方式によって確認できるので，すべてのプロセスに終了の要求が届きさえすればよい．これはメッセージのブロードキャストと空のサブプールによって行うことができる．以下に「特定のプロセスを終了させるメッセージ (%abort)」をブロードキャストすることによって強制終了を行い，

WTC 方式を利用してその終了を検出する方式を述べる．

制御プロセスの処理

%abort の到着確認メッセージ（%ackAbort）をカウントする“ack カウンタ”を用意する．

強制終了処理の開始 ack カウンタをクリアし，%abort をブロードキャストする．%abort は重みを持たない．

%ackAbort を受信したら ack カウンタを +1 する．

%terminated を受信すると運ばれてきた重みを制御プロセスの重みに加える．

終了の検出 制御プロセスの重みがゼロになり，%abort と同数の %ackAbort を受信したならば，すべてのプロセスが終了し通信中のメッセージも存在しないことが検出される．

サブプール側 PE の処理

%abort を受信すると %ackAbort を返信し，指定されたサブプールを終了させ，重みを %terminated で制御プロセスへ返却する．PE には空のサブプールを残し，強制終了中であることを記憶する．空のサブプールの場合は %ackAbort を返信し強制終了中であることを記憶する．該当するサブプールが存在しない場合は %ackAbort を返信し，空のサブプールを生成して強制終了中であることを記憶する．

強制終了中の空のサブプールがプロセスを受信すると，プロセスを終了させ，重みを %terminated で制御プロセスへ返す．

終了を検出したならば制御プロセスは 5.1 節で述べたように %forget をブロードキャストする．すべての %ackForget の受信によって強制終了は完了する．

該当するサブプールの存在しない PE，あるいは空のサブプールを持つ PE に %abort が到着した場合は重みの返却が起こらない．このため制御プロセスの重みがゼロになっても通信中の %abort の存在する可能性がある．%ackAbort の返信と ack カウンタの操作は通信中の %abort の存在しないことを確認するものである．

5.3 停止／再開

停止／再開とは，制御プロセスがメッセージを送ることにより同じプロセスプールに属するすべてのプロセスの実行を停止／再開させることである．停止／再開処理は何回でも繰り返される点で前述した強制終了とは異なる．ここでは停止／再開したことの確認ができる方式を考える．そしてすべてのプロセスの実行中／停止中であることが確認された状態で停止／再開処理を開始するものとする．以下ではまず停止／再開処理それぞれの考察を行い次に操作内容を述べる．

5.3.1 停止処理の考察

停止処理開始時点での制御プロセスの重み（負の整数）は「実行中である（と認識されている）プロセス群の重み」を示している．したがって，まずカウンタを用意して制御プロセスの重みをコピーし，「特定のプロセスの実行を停止させるメッセージ (%stop)」を送信してプロセスを停止させ，停止したプロセスの重みのコピー（正の整数）をメッセージ（%stopped）で返信させカウンタに加えていき，カウンタの値がゼロになったならばすべてのプロセスの実行停止が確認できそうである．しかし以下のことを考慮する必要がある．

%stop を受信する前にサブプールが終了することもある．したがって %stop 送信後制御プロセスは %stopped だけでなく %terminated も受信することがある．停止したプロセスが終了することはないので，%terminated が運ぶ重みは %stopped と同じ「実行中であったプロセスの重み」である．このため %terminated の受信時には重みを制御プロセスの重みに加えるだけでなく %stopped を受信した時と同様にカウンタへ加える必要がある．加えた結果カウンタの値がゼロになったならば，やはり実行の停止が確認される．

5.3.2 再開処理の考察

停止中のプロセスは送信されないため，停止が確認されている状況では通信中のプロセスは存在せず必ずどこかの PE に存在している．このため再開は，「特定のプロセスの実行を再開させるメッセージ（%start)」をブロードキャストしその到着を確認するだけでよい．ただし以下のことを考慮する必要がある．

%start を受信するとプロセスは実行中となり送信も再開されるが，送信されたプロセスは「まだ(%start が到着していないため）停止中であるサブプール」に到着するかもしれない．**図 3** は %start のブロードキャストとそれに続く状況を示している．サブプール j (の存在する PE) は %start 受信後にプロセスを送信したが，このプロセスはサブプール i （の存在する PE) へ %start より先に到着するかもしれない．停止中のサブプールには再開処理の始まったことがわからないので，受信したプロセスを停止させ重みを返却し

てしまう．

この再開処理中の重みの返却（%stopped の送信）を防ぐには，受信したプロセスが「まだ停止していないプロセス」か「既に再開したプロセス」かをサブプールが判別できなくてはならない*．これは以下に示す世代管理によって実現できる．

世代管理

制御プロセスとサブプールが世代を保持し，送信するプロセスに付加する．停止／再開処理の開始時に制御プロセスが世代をひとつ進め，%stop および %start には新しい世代を付加して送る．受信したプロセスの世代がサブプールの世代より進んでいるならば「既に再開したプロセス」であり，遅れているならば「まだ停止していないプロセス」である．

停止／再開は，実行中／停止中であることを確認してから開始するので同時に2世代しか存在しない．このため世代管理を行うには3種類の世代（例えば 0→1→2→0→…）を設ければ十分である．

5.3.3 停止／再開操作

制御プロセスの処理

「まだ実行中であるプロセス群の重み」を保持する“実行中カウンタ”と，%ackStop および %ackStart をカウントする“ack カウンタ”を用意し，さらに停止/再開の世代を記憶する．

停止処理の開始　制御プロセスの重みを実行中カウンタにコピーし，ack カウンタはクリアし，世代をひとつ進め，%stop をブロードキャストする．%stop は新しい世代を運ぶ．重みは持たない．

%ackStop を受信したら ack カウンタを +1 する．

%stopped を受信したら** 運ばれてきた重みを実行中カウンタに加える．

%terminated を受信したら運ばれてきた重みを制御プロセスの重みと実行中カウンタに加える．

停止完了の検出　実行中カウンタの値がゼロになり，%stop と同数の %ackStop を受信したならば，すべてのプロセスの実行が停止し，通信中の %stop と %ackStop も存在しないことが検出される．

再開処理の開始　ack カウンタをクリアし，世代をひとつ進め，%start をブロードキャストする．%start は新しい世代を運ぶ．重みは持たない．

%ackStart を受信したら ack カウンタを +1 する．

再開完了の検出　%start と同数の %ackStart を受信したならば，すべてのプロセスが実行を再開し，通信中の %start と %ackStart も存在しないことが検出される．

サブプール側 PE の処理

%stop を受信すると %ackStop を返信し，指定されたサブプール（内のすべてのプロセスの実行）を停止させ，重みのコピーを %stopped で制御プロセスへ送り（サブプールの重みはそのままである），停止中であることの記憶と世代の更新を行う．空のサブプールの場合は %ackStop を返信し，停止中であることの記憶と世代の更新を行う．該当するサブプールが存在しない場合は %ackStop を返信し，空のサブプールを生成して停止中であることおよび世代の記憶を行う．

%start を受信すると %ackStart を返信し，指定されたサブプール（内のすべてのプロセスの実行）を再開させ，世代の更新を行う．サブプールが既に実行中の場合は %ackStart の返信のみ行う．空のサブプールの場合は %ackStart を返信し，実行中であることの記憶と世代の更新を行う．該当するサブプールが存在しない場合は %ackStart を返信し，空のサブプールを生成して実行中であ

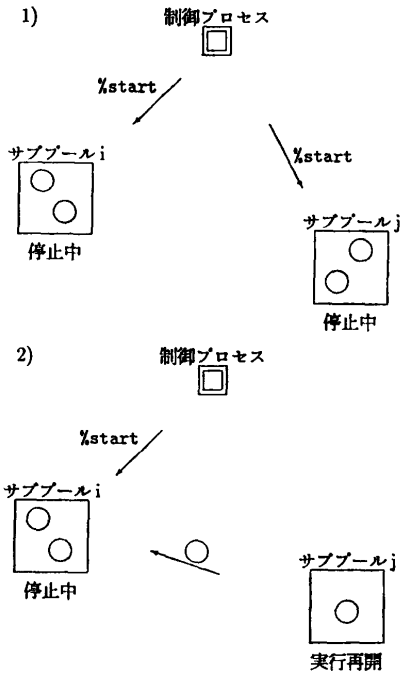

図 3　プロセスが先に到着するかもしれない
Fig. 3　A process may arrive earlier.

* 再開処理中の重みの返却を認め制御プロセス側で無視する方式は危険であり追い越しのあるシステムには適用できない[9]．

** %stopped は停止処理中にのみ受信しうる．

ることおよび世代を記憶する．

停止中のサブプールがプロセスを受信した場合はその重みをサブプールへ加え世代を比較する（サブプールは停止中，プロセスは実行中なので，世代は必ず異なる）．サブプールの世代が先行しているならばプロセスを停止させプロセスの重みのコピーを %stopped で制御プロセスへ送る．受信プロセスの世代が先行している場合はサブプールの世代を更新し実行を再開する．

該当するサブプールが存在しない，あるいは空のサブプールが存在する場合は %stop が到着しても重みのコピーの返却は起こらない．このため実行中カウンタがゼロになっても通信中の %stop が存在する可能性がある．%ackStop の返信と ack カウンタの操作は通信中の %stop が存在しないことを確認するためのものである．

図 4 は停止処理を示している．1) は処理の開始であり，制御プロセスの重み (w＝−900) を実行中カウンタへコピーし (c＝−900)，%stop をブロードキャストした．2) は %stop 受信時の処理を示している．%ackStop と %stopped が送信され，停止中である空のサブプールが生まれた．通信中のプロセスが存在するため，この時点で通信中の %ackStop と %stopped がすべて受信されてもまだ停止処理は完了しない．3) では停止中の空のサブプールにプロセスが到着し重みのコピーが返却された．この重みが制御プロセスに到着すると実行中カウンタがゼロになり停止処理が完了する．

%stop が到着する前にすべてのプロセスが終了してしまうこともある．この場合は %stopped はひとつも返信されない．%terminated を受信し重みの加算を行うと制御プロセスの重みと実行中カウンタの値が両方ともゼロになりプロセスプールの終了が検出される．この時は %stop と同数の %ackStop を受信し通信中のメッセージがすべてなくなるのを待ってから %forget をブロードキャストする．すべての %ackForget 受信によってプロセスプールの終了が完了する．

5.4 状態の変更

5.3 節で述べた停止／再開は，「状態を実行中と停止中の２つに限り」「停止中のプロセスは送信されず終了もしない」という特徴を利用した状態変更処理と言うこともできる．ここではこれを一般化し「プロセスの送信や終了の制限がなく」「任意の数の状態を扱

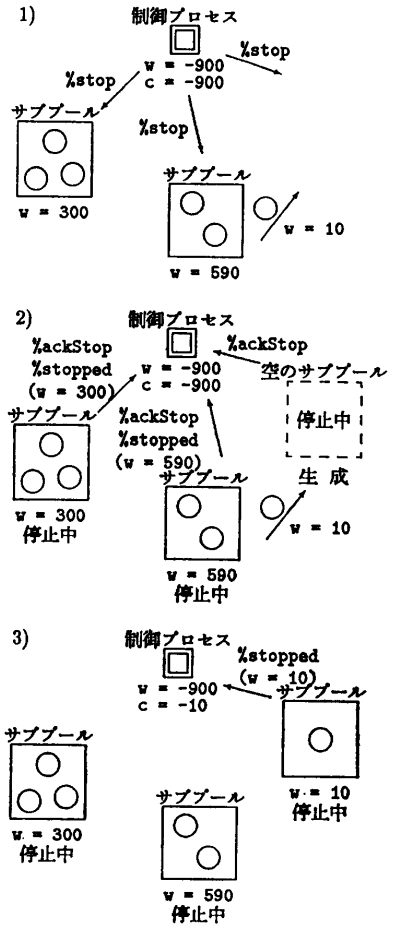

図 4 停止処理
Fig. 4 Stop operations.

うことのできる」状態変更方式を述べる．ここでも状態変更の完了したことを確認できる方式を考え，ある状態への変更は，すべてのプロセスが前の状態であることが確認された状況で開始するものとする．

5.4.1 停止処理と異なる点

状態の変更は以下に示すように基本的には停止処理と同じ操作で行うことができる．

状態変更処理開始時点での制御プロセスの重みは「前の状態であるプロセス群の重み」を示している．カウンタを用意して制御プロセスの重みをコピーし，状態変更を行うメッセージ %change を送信する．%change を受信するとサブプールは状態を変更して「新しい状態に変わったプロセス群の重み」のコピーを %changed で返却する．%changed を受信すると重みをカウンタに加えていき，値がゼロになったならばすべてのプロセスの状態変更が確認できる．しかし以下の２点を考慮する必要がある．

プロセスはいつでも送信されるので，サブプールが自分と異なる状態を持つプロセスを受信した場合，そのプロセスは「まだ変更されていない」のかもしれないし「既に変更されている」のかもしれない．「まだ変更されていない」プロセスを受信した場合は，プロセスの状態をサブプールの状態に変更し，プロセスの重みのコピーを %changed で制御プロセスへ送る必要がある．一方「既に変更された」プロセスならば，サブプールの状態を変更し重みのコピーを送る必要がある．このように異なった処理を行うので受信プロセスの未変更／変更済みが判別できなくてはならない．

サブプールはいつでも自発的に終了するので，制御プロセスは，状態変更前に送信された %terminated だけでなく変更後に送信された %terminated も受信する．前者は前の状態のうちに終了したプロセス群の重みを持っているので %changed を受信した時と同様にカウンタに加える必要がある．一方，後者は新しい状態に変わってから終了したプロセス群の重みを運ぶので，カウンタに加えてはいけない．このように異なった処理を行う必要があるので，制御プロセスは2つを判別できなくてはならない．**図5**は2種類の %terminated を示している．1) の %terminated は状態変更前に送信されたものなのでカウンタに加える (c= −700→ −200) が，2) の%terminated は状態変更後に送信されたものなのでカウンタに加えてはいけない．

上記の判別は，再開処理における「まだ停止していないプロセス」か「既に再開したプロセス」の判別と同様に，世代を管理し %change および %terminated が世代を運ぶことによって可能になる*．

5.4.2 状態変更操作

制御プロセスの処理

「前の状態であるプロセス群の重み」をカウントする“旧状態カウンタ”と，%ackChange をカウントする“ack カウンタ”を用意する．

状態変更処理の開始　制御プロセスの重みを旧状態カウンタにコピーし，ack カウンタはクリアし，世代をひとつ進め，%change をブロードキャストする．%change は新しい状態と新しい世代を運ぶ．重みは持たない．

%ackChange を受信したら ack カウンタを +1 する．

%changed を受信したら運ばれてきた重みを旧状態カウンタに加算する．

遅れている世代の %terminated を受信したら運ばれてきた重みを制御プロセスの重みと旧状態カウンタの両方に加える．

同じ世代の %terminated を受信した場合は運ばれてきた重みを制御プロセスの重みにのみ加える．

変更完了の検出　旧状態カウンタの値がゼロになり，%change と同数の %ackChange を受信したならば，すべてのプロセスが新しい状態に変更され，通信中の %change と %ackChange も存在しないことが検出される．

サブプール側 PE の処理

%change を受信すると %ackChange を返信し，以下の処理を行う．

- %change の世代がサブプールの世代より進んでいるならば，サブプールの重みのコピーを %changed で制御プロセスへ送り（サブプールの重みはそのままである），サブプールの状態変更を行い，世代を更新する．空のサブプールの場合は %changed の送信は行わず状態変更と世代更新のみを行う．
- 同じ世代の %change の場合，これは「既に

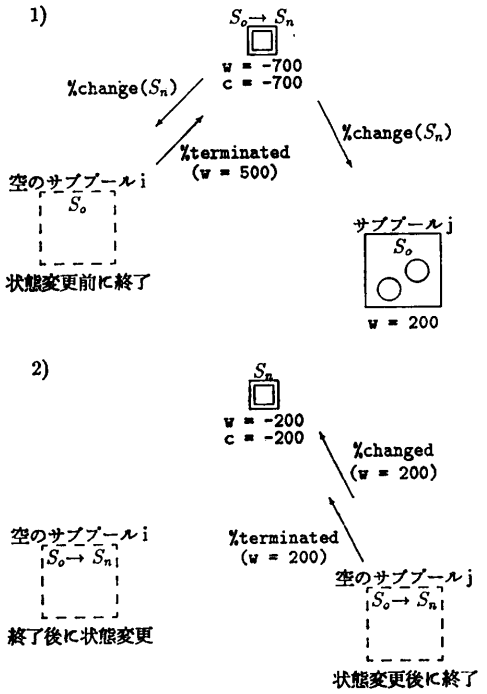

図5　状態変更後に送信された %terminated も受信する

Fig. 5　The controlling process may receive a %terminated sent after changing.

* メッセージの追い越しがなければ世代管理を行わなくとも %change および %terminated が状態を運ぶことによって判別は可能である[6]．

変更されたプロセス」の受信によってサブプールの状態変更が既になされている場合なので，何も行わない．

- 該当するサブプールが存在しない場合は空のサブプールを生成し新しい状態と世代を記憶する．

遅れている世代のプロセスを受信した場合は，プロセスの状態をサブプールの状態に変更し，重みをサブプールへ加えるとともに %changed で制御プロセスへ送る．

進んでいる世代のプロセスを受信した場合は，サブプールの状態変更と世代の更新を行い，サブプールの重みのコピーを %changed で制御プロセスへ送った後，プロセスの重みをサブプールへ加える．

%ackChange の返信と ack カウンタの操作は，停止処理と同様，通信中の %change が存在しないことを確認するためのものである．

また停止処理と同様に，%change が到着する前にすべてのプロセスが終了してしまうこともある．%terminated の受信のみによって制御プロセスの重みと旧状態カウンタの値が両方ともゼロになる．この時も停止処理中の場合と同様に %change と同数の %ackChange を受信し通信中のメッセージがすべてなくなるのを待ってから %forget をブロードキャストする．すべての %ackForget 受信によってプロセスプールの終了が完了する．

6. プロセスの存在する PE を把握する方式との比較

本方式は「プロセスの存在する PE を把握する」方式と比べて以下に示す利点を持っている．

- サブプール生成の伝達が不要．
- プロセスのふるまいには依存せず，PE に比例した数の制御メッセージで制御ができる．
- 制御プロセスは PE ごとの情報を持つ必要がない．
- メッセージの追い越しのある環境でも同一の方式が適用できる．

一方，以下の欠点もある．

- 一部の PE にのみプロセスが存在するような状況では大部分の制御メッセージが無駄になってしまう．
- プロセスプールの終了後に空のサブプールを消去する後処理が必要．

このため本方式は，「後処理が問題にならない程度の計算量を持つ」応用の「同じプロセスプールに属するプロセスが広範囲の PE に存在する」ような分散実行に適している．

また本方式は制御メッセージに対して到着確認メッセージを送信しているが，「プロセスの存在する PE を把握する」方式と同様に制御メッセージに重みを持たせることによって省略可能である．

7. おわりに

プロセスの存在する PE を把握する方式の問題点を検討し，それを解決する方式を提案した．本方式は，制御メッセージをブロードキャストして特定のプロセス群の強制終了，実行の停止／再開，状態の変更を行い，WTC 方式の応用によりその完了を確認する．このため制御プロセスは PE ごとの情報を持つ必要がなく，プロセスのふるまいに依存せずに処理を行うことができる．

参考文献

1) Rokusawa, K., Ichiyoshi, N., Chikayama, T. and Nakashima, H.: An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems, *Proc. ICPP: International Conference on Parallel Processing*, Vol. I, pp. 18-22 (1988).
2) Mattern, F.: Global Quiescence Detection Based on Credit Distribution and Recovery, *Inf. Process. Lett.*, Vol. 30, No. 4, pp. 195-200 (1989).
3) Watson, P. and Watson, I.: An Efficient Garbage Collection Scheme for Parallel Computer Architectures, *Proc. PARLE: Parallel Architectures and Languages Europe*, LNCS 259, Vol. II, pp. 432-443 (1987).
4) Bevan, D. I.: Distributed Garbage Collection Using Reference Counting, *Parallel Computing*, Vol. 9, No. 2, pp. 179-192 (1989).
5) Tel, G. and Mattern, F.: The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes, *Proc. PARLE: Parallel Architectures and Languages Europe*, LNCS 505, Vol. I, pp. 137-149 (1991).
6) 六沢一昭，市吉伸行：WTC 方式を用いた分散プロセス制御方式，第3回並列処理シンポジウム JSPP '91, pp. 437-444 (1991).
7) 六沢一昭，市吉伸行，瀧 和男，吉田かおる，稲村 雄，中島 浩：並列処理における PE 間に渡るゴールの重みつき参照カウントを用いた管理方

式，第 36 回情報処理学会全国大会論文集，7 H-2 (1988).
8) 川合英夫，伸瀬明彦，今井　明，後藤厚宏，六沢一昭：並列推論マシンにおける KL 1 の実行制御方式—分散ゴール管理の課題と対策—，第 74 回情報処理学会計算機アーキテクチャ研究会，82-2 (1990).
9) 六沢一昭，中島克人，市吉伸行，稲村　雄：大域 GC における WTC 方式を用いたマーキング終了検出，第 43 回情報処理学会全国大会論文集，5 L-11 (1991).
10) Shavit, N. and Francez, N.: A New Approach to Detection of Locally Indicative Stability, *Proc. 13th ICALP: International Colloquium on Automata, Languages and Programming*, LNCS 226, pp. 344-358 (1986).
11) Dijkstra, E. W., Feijen, W. H. J. and van Gasteren, A. J. M.: Derivation of a Termination Detection Algorithm for Distributed Computations, *Inf. Process. Lett.*, Vol. 16, No. 5, pp. 217-219 (1983).
12) Mattern, F.: Algorithms for Distributed Termination Detection, *Distributed Computing*, Vol. 2, pp. 161-175 (1987).
13) Taki, K.: The Parallel Software Research and Development Tool: Multi-PSI System, *Programming of Future Generation Computers*, pp. 411-426, Elsevier Science Publishers B. V., North-Holland (1988).
14) Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *Comput. J.*, Vol. 33, No. 6, pp. 494-500 (1990).
15) Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Chikayama, T.: Distributed Implementation of KL 1 on the Multi-PSI/V 2, *Proc. ICLP: International Conference on Logic Programming*, pp. 436-451 (1989).

(平成 3 年 7 月 31 日受付)
(平成 4 年 1 月 17 日採録)

六沢　一昭（正会員）

1958年生．1981年早稲田大学理工学部電子通信学科卒業．1983年同大学院理工学研究科修士課程修了．同年沖電気工業(株)入社．現在同社総合システム研究所勤務．1986～1990年(財)新世代コンピュータ技術開発機構に出向．並列論理型言語処理系，並列プログラミングの研究に従事．電子情報通信学会会員．

市吉　伸行（正会員）

1979 年東京大学理学部情報科学科卒業．1981 年同大学院修士課程修了．1982 年(株)三菱総合研究所入社．1984 年より 1 年間米国バテル研究所にて人工知能を研修．1987 年より(財)新世代コンピュータ技術開発機構へ出向．論理型言語処理系，大規模並列プログラムの研究開発に従事．