# WTC 方式を用いた分散プロセス制御方式*

六沢 一昭†
rokusawa@okilab.oki.co.jp
沖電気工業 (株) 総合システム研究所

市吉 伸行‡
ichiyoshi@icot.or.jp
(財) 新世代コンピュータ技術開発機構

## 概要

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、状態の変更を行なう方式を述べる。非同期メッセージ通信を行なう分散環境では、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。本方式では終了検出の方式である WTC (Weighted Throw Counting) 方式の応用によりこの問題を解決した。

## 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」を行ない、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なわれる環境では「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 (“通信中のメッセージ”と呼ぶ)」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 [4, 5] をプロセス管理に応用したものであり、ポーリングを行なうことなく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, 2, 3] には、WTC 方式を利用し「プロセスの存在を制御プロセスに伝える」ことにより強制終了 [1, 2] 及び実行の停止 / 再開 [3] を行な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。
- 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、プロセスの存在の伝達を必要とせず、メッセージの追い越しにも低コストで対処できる方式を述べる。計算モデルの定義の後 WTC 方式を説明し、文献 [1, 2, 3] の「プロセスの存在を伝える方式」を述べる。そして強制終了、実行の停止 / 再開、状態の変更方式を述べ、最後に「プロセスの存在を伝える方式」との比較を行なう。

## 2 計算モデル

- 有限個のプロセッサ (PE) があり、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なう。
- 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いつでも終了しうる。また同じ ID を持つ新しいプロセスをいつでも生成できる。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。
- 負荷分散などのために実行中のプロセスを他の PE へ送信することがある。停止中のプロセスは送信されない。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状態のプロセスを“通信中のプロセス”と呼ぶ。

*A Process Control Scheme for Distributed Processing Systems Using Weighted Throw Counting
†Kazuaki ROKUSAWA (Systems Laboratory, OKI)
‡Nobuyuki ICHIYOSHI (ICOT)

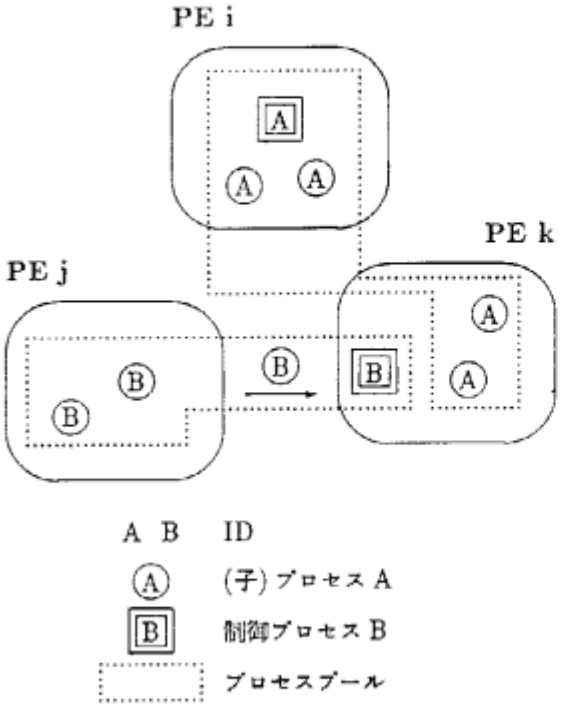


図 1: 計算モデル

## 3 WTC 方式

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。PE が受信したプロセスは同じ ID を持つサブプールに加えられる。同じ ID のサブプールが存在しなかった場合は新しくサブプールを生成する。

制御プロセスとサブプール及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、サブプール及び通信中のプロセスの重みは正の整数である。そして『重みの合計がゼロ』を満たすように管理する。この結果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はゼロになる。

サブプールは属するプロセスがすべて終了すると終了し、終了を示すメッセージ (%terminated) を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重みを運ぶ。%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の終了 (プロセスプールの終了) が検出される。

## 4 プロセスの存在を伝える方式

ここでは文献 [1, 2, 3] が述べている「プロセスの存在を制御プロセスへ伝える」方式を簡単に述べる。

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処理によって強制終了、実行の停止 / 再開を行なう。

(1) 記憶している PE へのみメッセージを送る。

(2) 処理中に %ready を受信したならばその送信元 PE へメッセージを送る。

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。

メッセージも通信中のプロセスと同様に重みを持ち、通信中の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を防いでいる。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なう。

(3a) サブプールが存在するならばメッセージが示す処理を行なう。例えば強制終了ならば、プロセスを終了させ自分の重みとメッセージの重みの合計を %terminated で制御プロセスへ返す。

(3b) サブプールが存在しない場合はメッセージの重みを制御プロセスへ送り返す。

## 5 強制終了

強制終了とは、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を終了させることである。

### 5.1 問題点

制御プロセスが「特定の ID のプロセスを終了させるメッセージ (%abort)」を単純にブロードキャストするだけでは不十分である。%abort 到着時に PE に存在したプロセスを終了させることはできるが、通信中であったプロセスの強制終了は不可能だからである。以下の操作を行なうとすべてのプロセスを終了させることが可能にはなる。

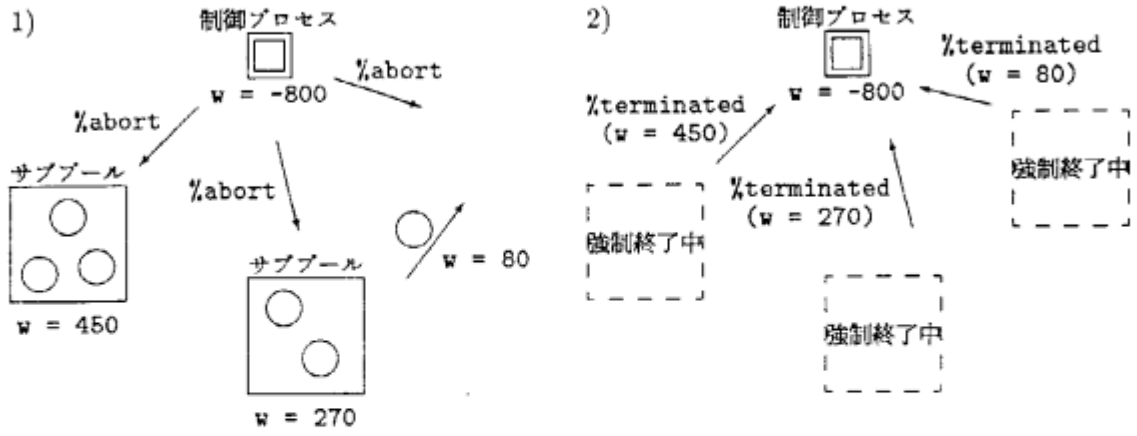


図 2: WTC 方式と強制終了処理 (%forget のブロードキャストの前まで)

%abort を受信したら指定された ID のプロセスプールが強制終了中であることを記憶する。そしてその後 同じ ID のプロセスを受信したら終了させる。

しかしこの方式ではプロセスプールの終了を検出できないため、以下の欠点がある。

- ID の再利用ができない。
- 強制終了中であることをすべての PE が永久に記憶し続けなければならない。

強制終了は何度でも行なうので特に後者は重大である。

## 5.2 WTC 方式を用いた強制終了方式

前述した問題は終了を検出できないことに起因していた。%abort をブロードキャストすることによって強制終了を行ない、WTC 方式を利用してその終了を検出する方式 (メッセージの追い越しのない環境を仮定) を以下に述べる。

### 制御プロセスの処理

**強制終了処理の開始** %abort をブロードキャストする。%abort は終了させるプロセスプールの ID を引数として持つ。重みは持たない。

**%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ったならばすべてのプロセスの終了が検出される[1]。

### サブプール側の処理

**%abort を受信すると** 指定された ID のサブプールを終了させ %terminated を制御プロセスへ送信する。そして「その ID のプロセスプールが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強制終了中であることの記憶だけを行なう。

**強制終了中の** プロセスを受信した場合は終了させ重みを %terminated で制御プロセスへ返す。

すべてのプロセスの終了を検出しても各 PE に残っている「強制終了中であることの記憶」を消去する必要がある。この消去を行なわないと「ある ID のプロセスプールが強制終了中であること」を PE は永久に忘れることができない。消去を行なうために制御プロセスはメッセージ (%forget) をブロードキャストする。%forget を受信すると[2] PE は記憶を消去し到着確認メッセージ (%ackForget) を制御プロセスへ送る。すべての PE からの到着確認メッセージの受信によって強制終了処理は完了する。

[1] サブプールの存在しない PE に %abort が到着しても重みは返信されないので、この時点ではまだ通信中の %abort があるかもしれない。

[2] メッセージの追い越しがないため、この時点では %abort 受信済みが保証される。

### 5.3 メッセージの追い越しのある場合

考慮する必要があるのは「%forget による %abort の追い越し」のみである。

%abort より先に %forget を受信した場合は、%ackForget は送信せず %forget 受信を記憶する。%forget の後に %abort を受信した場合は、すべての記憶を消去し %ackForget を送信する。

## 6 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 / 再開処理はすべてのプロセスの実行中 / 停止中であることが保証された状態で開始するものとする。

### 6.1 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「実行中であるプロセス群の重み[3]」を示している。従って、「特定の ID のプロセスの実行を停止させるメッセージ (%stop)」を送信し、停止したプロセスの重み (正の整数) を %stopped で返信させてカウントし、カウントした重み (停止したプロセス群の重み) と制御プロセス重み (実行中であったプロセス群の重み) の和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かもしれない。
- 強制終了と同じく、すべてのプロセスの実行停止が確認されても受信されていない %stop がまだ存在するかもしれない。

前者により、%stop 送信後　制御プロセスは %stopped だけでなく %terminated を受信することもある。この %terminated が運ぶ重みは「停止する前に終了したプロセス群の重み」なので、制御プロセスの重みへ加えればよい。加えた結果、停止プロセス群の重みとの和がゼロになったならば、やはり実行の停止が確認される。

後者は到着確認メッセージ (%ackStop) によってすべての %stop の受信を確認する。

[3]正確には、「実行中であると制御プロセスが認識しているプロセス群の重み」である。

### 6.2 再開処理の考察

停止中のプロセスは送信されないため、すべてのプロセスが停止している状況では通信中のプロセスは存在しない。プロセスはすべて必ずどこかの PE に存在している。このため再開は、「特定の ID 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停止中のサブプールに到着するかもしれない。停止中のサブプールには必ず %start が送られてくるが、%start より先にプロセスが到着するかもしれないからである。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停止処理を行なってしまう。即ち受信したプロセスを停止させ重みを %stopped で制御プロセスへ送ってしまう。

上記の解決方法には以下の 2 つが考えられる。

- %stopped が送信されることを認め、制御プロセス側で無視する。
- %stopped を送信させない。

前者は「再開処理中の制御プロセスは %stopped を受信しても無視する」ことによって容易に実現できるが、無視するのは“安全な処理”ではない。例えば間違って %stopped が送信された場合の検出が不可能になってしまう。

後者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、言い換えると、自分より停止 / 再開の世代が「遅れている」か「進んでいる」かを区別する方式である。これは通信中のプロセスとサブプールに世代を設けることによって実現できる。

#### 世代管理

制御プロセスが世代を保持し、停止 / 再開の度に世代をひとつ進める。そして %stop 及び %start に世代を付加して送りサブプール側はその世代を記憶する。停止中のサブプールが同じ ID のプロセスを受信した場合、サブプールの世代が受信プロセスの世代より上ならばプロセスを停止させ %stopped を送信するが、サブプールの世代が下ならば %start を

受信した時と同様にサブプールの実行を再開する[4]。このため実行中のサブプールが %start を受信することもある。

停止 / 再開は、再開 / 停止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 3 種類の世代 (例えば $0 \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow \ldots$) を設けさえすればよい。

## 6.3 停止 / 再開処理

以下に処理を示す (メッセージの追い越しのない環境を仮定)。世代管理を行なうことにより、再開処理中に %stopped が送信されることを防いでいる。

### 制御プロセスの処理

制御プロセスは、%stopped で返信される「停止したプロセス群の重み」をカウントする“停止済みカウンタ”と、%ackStop 及び %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** 停止済みカウンタと ack カウンタをクリアし、世代をひとつ進め、%stop をブロードキャストする。%stop は実行を停止させるプロセスプールの ID と世代を引数として持つ。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**[5] 運ばれてきた重みを停止済みカウンタに加える。

**%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に加える。

**停止完了の検出** 停止済みカウンタの重みと制御プロセスの重みの和がゼロになり、%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 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 は実行を再開させるプロセスプールの ID と世代を引数として持つ。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、すべてのプロセスの実行が再開し、通信中の %start も存在しないことが検出される。

### サブプール側の処理

**%stop を受信すると** %ackStop を返信する。そして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、サブプールの重みを %stopped で制御プロセスへ送り、停止中であること及び世代を記憶する。サブプールの重みは操作しない。%stopped の送信は「この重みの分だけプロセスが停止した」ことを意味する。該当するサブプールが存在しない場合は停止中であること及び世代の記憶だけを行なう。

**停止中のサブプールが** 同じ ID のプロセスを受信した場合はサブプールの世代と受信プロセスの世代を比較する。サブプールの世代が上ならばそのプロセスを停止させ重みを %stopped で制御プロセスへ送る。サブプールの世代が下ならばサブプールの実行を再開する。尚、受信プロセスの重みをサブプールへ加える処理は通常通り行なう。

**%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) を再開させる。該当するサブプールが存在しない場合、及びサブプールが既に実行中の場合は %ackStart の返信のみ行なう。

停止処理中に %terminated を受信し重みの加算を行なったところゼロになる (すべてのプロセスの終了が検出される) ことが考えられる。%stop を受信する前にもサブプールの終了する可能性があるからである。この場合は各 PE に「停止中であることの記憶」が残っているので、強制終了と同様に %forget をブロードキャストし記憶を消去する。そしてすべての %ackForget の受信によってプロセスプールの終了が確認される。

## 6.4 メッセージの追い越しがある場合

1 回の停止あるいは再開処理においては追い越しの影響はない。また、停止処理に係わる制御メッセー

[4] 再開は %start を待ってもよい。

[5] %stopped は停止処理中にのみ受信しうる。

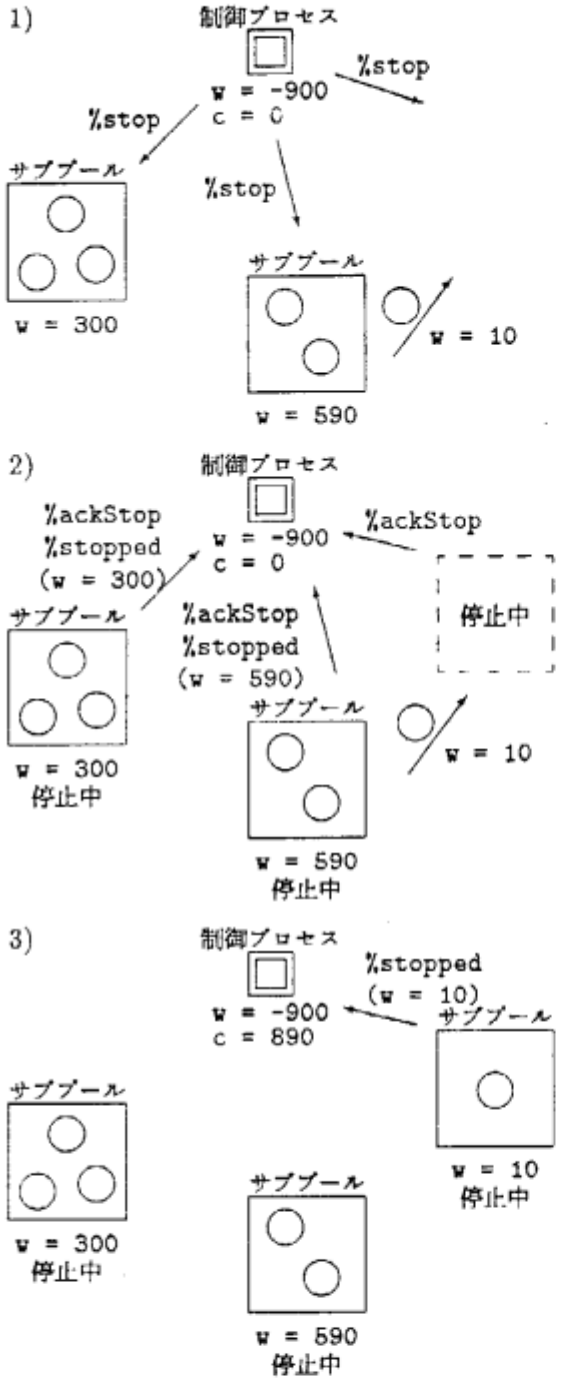


図 3: 停止処理

ジと再開処理に係わる制御メッセージは混じることがないのでやはり追い越しは起こらない。このため強制終了と同様に「%forget による %stop の追い越しのみ考えればよい。

## 7 状態の変更

6 章でのべた「停止 / 再開」は、状態数を 2 つに限り「停止中のプロセスは送信されない」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。本章ではこれを一般化し「プロセス送信の制限はなく任意の数の状態を扱うことのできる」状態変更方式を述べる。尚、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保証された状態で開始するものとする。

### 7.1 停止処理と異なる点

状態の変更は基本的に停止と同じ処理で行なうことができる。即ち、制御プロセスが新しい状態を運ぶメッセージ %change を送信し、サブプールは状態を変更して重みを %changed で返信する。「状態の変更されたプロセス群の重み」と「前の状態であったプロセス群の重み」の和がゼロになったならば変更が完了する。しかし停止処理と異なりプロセスは状態が変更されても送信されるので、以下に示すように再開処理と同様の問題が発生する。

> ある状態のサブプールが状態の異なるプロセスを受信した場合、どちらの状態が新しいのかわからない。

この問題は停止 / 再開と同様に「世代を管理する」ことによって解決できるが、以下に示す方式でも解決可能である。

### 7.2 世代管理を行なわない方式

自分の状態と異なるプロセスを受信した場合、サブプールは必ず %changed を送信する。ただし %changed は重みだけでなく状態も運ぶ。従って %changed の送信は「この重みの分だけこの状態に変更した」という意味になる。

制御プロセスは %changed を受信すると自分の状態と比較する。同じならばこれは「制御プロセスの要求の通り状態が変更された」ことを示すので、「変更されたプロセス群の重み」に加算する。異なっている場合は「一旦は制御プロセスの要求通り状態変更がなされたプロセスが再び前の状態に戻ってしまった[6]」ことを示すので「変更されたプロセス群の重み」

[6] これは、%change 受信によって状態を変更したプロセスが送信され、まだ %change を受信していないサブプールに到着した場合である。

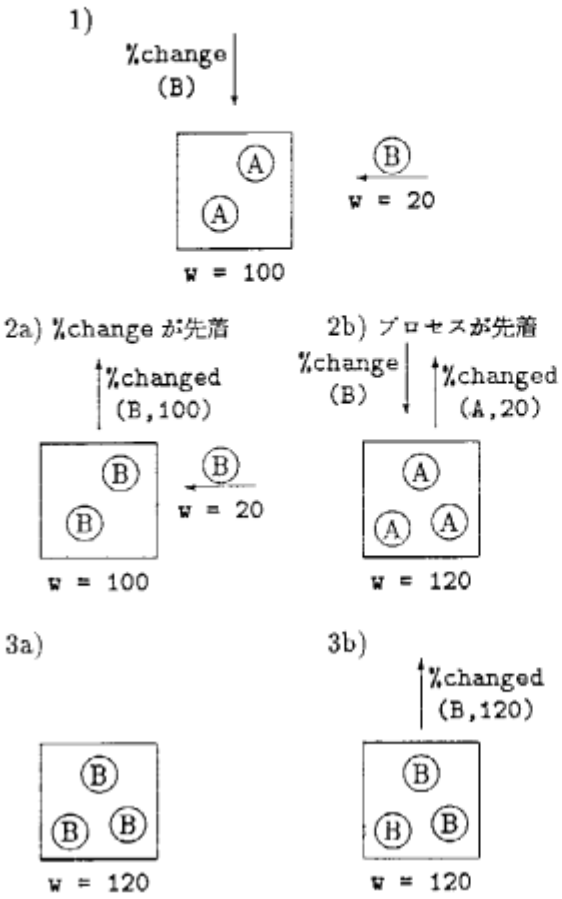


図 4: 状態変更 (%ackChange は省略)

から減算する。この場合は新しい状態への変更を行なう %change が必ず %changed の送信元プロセッサへ到着し、戻ってしまった分の重みを補うだけの重みが %changed で送られてくる。そしてすべてのプロセスの状態変更が完了した時のみ「変更されたプロセス群の重み」と「前の状態であったプロセス群の重み」の和がゼロになる。

しかしこの方式はメッセージの追い越しがあると使えない。新しい状態への変更を示す %changed が状態の逆戻りを示す %changed を追い越すと、変更が完了しないうちに 2 つの重みの和がゼロになる可能性があるからである。

## 7.3 状態変更処理

世代管理を行なわない状態変更処理 (メッセージの追い越しのない環境を仮定) を以下に示す。

### 制御プロセスの処理

制御プロセスは、%changed で返信される「状態を変更したプロセス群の重み」をカウントする“変更済みカウンタ”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** 変更済みカウンタと ack カウンタをクリアし %change をブロードキャストする。%change は状態の変更を行なうプロセスプールの ID と新しい状態を引数として持つ。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** %changed が示す状態と制御プロセスの状態 (新しい状態) を比較する。同じならば運ばれてきた重みを変更済みカウンタに加算し、異なっているならば減算する。

**%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に加える。

**変更完了の検出** 変更済みカウンタの重みと制御プロセスの重みの和がゼロになり、%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 も存在しないことが検出される。

### サブプール側の処理

**%change を受信すると** %ackChange を返信する。そして指定されたサブプール (内のすべてのプロセス) を %change が指定する状態に変更し、サブプールの重みと新しい状態を %changed で制御プロセスへ送り、新しい状態を記憶する。サブプールの重みは操作しない。該当するサブプールが存在しない場合は新しい状態の記憶だけを行なう。

**同じ ID で状態の異なる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、重みとサブプールの状態を %changed で制御プロセスへ送る。受信プロセスの重みをサブプールへ加える処理は通常通り行なう。

この処理方式を用いて状態変更を行なうと、一般に「サブプールは存在しないが新しい状態を記憶し

ている」PE [7]が存在する。このPEは重みを持たないのでWTC方式による終了検出には影響しない。このため制御プロセスの重みがゼロになった(すべてのプロセスの終了が検出された)場合は、記憶のみを持つPEが存在するかもしれないので、強制終了と同様に%forgetをブロードキャストしてその記憶の消去を行なう必要がある。%ackForgetをすべて受信してプロセスプールの終了が確認できる。

### 7.4 メッセージの追い越しがある場合

停止/再開と同様に世代管理を行なえばよい。上述した世代管理を行なわない方式では7.2で述べたようにメッセージの追い越しに対応できない。

## 8 プロセスの存在を伝える方式との比較

制御プロセスがプロセスの制御を行なうために%abortなどのメッセージをどのPEへ送信するかについて以下の2つが考えられる。

- サブプールの存在するPEへのみ送信する。
- すべてのPEへ送信する。

4章で述べたプロセスの存在を伝える方式[1, 2, 3]は前者であり、サブプールの生成を制御プロセスへ伝えることにより実現している。無駄なメッセージ送信は起きないが[8]以下の問題がある。

- プロセスの存在を伝えるメッセージが必要。最悪の場合は送信されるプロセスと同じ数だけ必要になる。
- メッセージの追い越しのある環境では、サブプールの生成を伝えるメッセージと終了を伝えるメッセージの追い越しに対処するため、PEと同数のカウンタが制御プロセスに必要となりコストが高くなる。

一方、本方式は後者であり無駄なメッセージ送信は起きるが、以下の利点がある。

- プロセスのふるまいに依存せずPE数に比例した数のメッセージで済む。
- サブプールの生成を伝えないため追い越しの影響を受けるメッセージ送信が少なく、追い越しのある環境でも低コストで対処できる。

メッセージ遅延時間の差が問題にならない規模の計算機システムで、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を行なうシステムに本方式は適している。

## 9 おわりに

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、状態の変更を行ない、WTC方式の応用によりその完了を確認する方式を述べた。本方式は、以下の特長を持つ。

- プロセスのふるまいに依存することなく処理がなされる。
- メッセージの追い越しにも低コストで対処できる。

## 参考文献


[1] K. Rokusawa, N. Ichiyoshi, T. Chikayama and H. Nakashima, "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems," ICPP, 1988.

[2] 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"、情報処理学会第36回全国大会7H-2, 1988年3月.

[3] 川合英夫、仲瀬明彦、今井明、後藤厚宏、六沢一昭、"並列推論マシンにおけるKL1の実行制御方式 - 分散ゴール管理の課題と対策 -"、情報処理学会第74回計算機アーキテクチャ研究会82-2, 1990年4月.

[4] P. Watson and I. Watson, "An efficient garbage collection scheme for parallel computer architectures," PARLE, 1987.

[5] D. I. Bevan, "Distributed garbage collection using reference counting," PARLE, 1987.


[7] このPEでサブプールの生成/終了が起きると記憶も消滅する。

[8] サブプール終了後にメッセージの到着することがあるため、無駄なメッセージ送信の起きることもある。