Elastic Barrier: 一般化されたバリア型同期機構†

松 本 尚††

多くの共有メモリ型の密結合マルチプロセッサシステムが開発・実用化されている．しかし，密結合の利点を活かして複数のプロセッサを“密に”協調させて処理を行わせるには様々な問題点が残っている．その一つにプロセッサ間の同期の問題がある．共有メモリを介してプロセッサ間のデータの交換をする場合，特殊なアルゴリズムでない限り，生産者-消費者等のデータ依存や制御依存による順序関係を保存したり，共有資源の排他制御を行うために同期が必要となる．さらに，プロセッサ間の協調の度合いが強ければ強いほど同期が必要とされる頻度は多くなる．そこで，協調の度合いの強い，つまり粒度の細かい，並列処理を効率よく実行するためには，同期のためのオーバヘッドを十分に小さく抑える必要がある．この目的のために，細粒度タスクの静的スケジューリングと組み合わせて用いられる極めて軽い同期機構 Elastic Barrier（一般化されたバリア型同期機構）を考案した．本論文ではこの機構の構成と動作を説明し，さらに，機構の特徴を明らかにするために，広がりのあるバリア（Fuzzy Barrier）としての使用法，機構の能力拡張法，バリア同期を基本とするためのオーバヘッド削減能力の限界，同期機構とシステムのプリエンプション機構との整合性，他の同期機構との比較といった点について論じる．

1. はじめに

プロセッサ間の同期のオーバヘッドの削減はマルチプロセッサシステムにとって大きな問題の一つである．細粒度の並列性を利用する場合には，最適化（並列化）コンパイラ等で実行前にプロセッサに細粒度のタスクを静的にスケジューリングして，コンテクストスイッチ等のオーバヘッドを削除することが期待できる[1)~4)]．これまでに提案された同期のためのハードウェア支援機構[5)~8)]の多くは，プロセッサの仮想化を重視しているためソフトウェアによるこの静的スケジューリングを利用することを前提にしていない．このため，最適化コンパイラと組み合わせて高速性を追求するシステムにとって，それらはオーバヘッドがまだ大き過ぎた．静的なスケジューリングを前提にしていても，静的に見積もることの難しいデータ通信路の競合やキャッシュミスといった要因による遅延のため，さらにはコンパイラによる解析を簡略化するために同期機構は不可欠である．本論文では，簡単な付加ハードウェアで実現でき，細粒度タスクの静的スケジューリングと組み合わせて用いられる極めて軽い同期機構[4),9)]の構成と定性的特徴について述べる．

第2章で機構の前提条件としてプロセッサ資源のスケジューリングの方針を，第3章で **Elastic Barrier**（一般化されたバリア型同期機構）の構成と動作を述べ，第4章で本機構の広がりを持ったバリアとしての使用法を述べ，第5章でバリア同期を基本とするために生じる能力の限界に言及し，第6章で FIFO キューを用いたオーバヘッド削減能力の増強法を提示し，第7章でマルチジョブ実現のために必須であるシステムのプリエンプション機構との整合性を論じ，第8章で同じ目的のために考案された他の同期機構との比較を行い，第9章で簡単にまとめを述べる．

2. 機構の前提条件

マルチプロセッサ上のプロセッサ間の同期問題はプロセッサ資源のスケジューリングの問題と深く関係している．ここでは，要素プロセッサとして従来型のレジスタを持つプロセッサの使用を前提とする．このタイプのプロセッサではコンテクスト切替のコストが大きい．密に協調する複数の命令流を用いた細粒度の並列処理を考えているので，命令流間で同期が頻繁に必要になる．効率よく実行が行われるためには，同期を取りあう命令流（一つのプロセッサに割り当てられる命令の集まり，ここでは**シュレッド (shred)**[4),10)] と呼ぶ）は同時に実プロセッサに割り当てられ，コストの掛かるコンテクストスイッチ等のオーバヘッドなしに同期を取れる必要がある．このために以下の条件を課す．

OS がプロセッサ資源のスケジューリングを行う処理単位を**プロセス**と呼ぶことにする．プロセスは複数のシュレッドから構成され，スケジューリング時に OS にシュレッド数分のプロセッサの割り当てを要求する．OS は指定された台数の割り当てが可能になった時点で同時にプロセッサをプロセスに割り付ける．

† Elastic Barrier: A Generalized Barrier-Type Synchronization Mechanism by TAKASHI MATSUMOTO (IBM Research, Tokyo Research Laboratory).

†† 日本アイ・ビー・エム(株)東京基礎研究所

また，OS によってプリエンプトされる際も，同じプロセスに属するシュレッドは同時にプロセッサの割り当てを解かれる．本論文では，同じプロセスに属するシュレッド間の同期を扱う．プロセス間の同期は従来のように OS を介して，または共有メモリでビジーウェイトすることで達成される．なお今後，細粒度の処理単位を**タスク**と呼ぶことにする．プロセスは複数のタスクから構成され，プロセス内のタスクは静的にプロセッサにスケジューリングされている．プロセス内の同じ一つのプロセッサによって実行されるタスクの集合（つまり，一つのプロセッサに対する命令流）がシュレッドである．

3. Elastic Barrier：一般化されたバリア型同期機構

2章で述べた方針の下で，同じプロセスに属するすべてのシュレッドが同時に待ち合わせるバリア同期[11]を実現する機構は比較的簡単に構成できる．そこで，まずバリア型の同期機構について述べ，それを拡張し一般化する方法を述べて，Elastic Barrier の構成と動作について説明する．

3.1 バリア型同期機構

図1に軽い同期機構を含むシステムの全体構成を示す．ここでは便宜的にプロセッサ間の結合方式は共有バス方式とする．データ通信の競合を減らすため同期情報を伝達する**同期信号バス**を別に設け，各プロセッサをそのバスの一本のラインに対応させる．つまり，プロセッサ台数分の同期信号ラインからなる同期信号バスを設ける．プロセッサごとに同期コントローラが設けられ，これが同期信号バスを使って同期の成立を検出する．プロセッサはバリア同期地点になるとコントローラを使って自分に対応する同期信号ラインをアクティブにする．コントローラ内には同期を取るべき**プロセッサを登録する同期レジスタ**があり，同じプロセスに属するプロセッサが登録されている．このレジスタをマスクとして使って，コントローラは同期信号バスを監視し，同じプロセスに属するプロセッサに対応するすべての信号ラインがアクティブになる，つまり同期が成立するのを検出する．プロセッサはこの検出までは先の命令を実行しない．

同期に関する情報は，プロセッサの命令コードに同期情報のためのフィールドまたはタグを新設するか，同期情報のためのプリフィックス命令を新設することで命令流中に埋めこまれる．この同期情報は『この情報が付加された命令の直前（または直後）で待ち合わせを行う必要がある』という同期要求を示す1ビットの情報で良い．待ち合わせのためにフラグ等をチェックする処理のオーバヘッドをなくすには，待ち合わせが成立するまでプロセッサをハードウェア的に，一時 wait 状態にして止めればよい．

3.2 ダミーの同期要求による一般化

一般の同期では，同じプロセスに属するシュレッド（プロセッサ）であっても，同期位置によって同期を必要とするシュレッドの組合せが変化する．その様子を**図2**に示す．図2では×印の点が本当に同期を必要とする地点であり，同期位置1ではシュレッド2を除いたすべてのシュレッドが待ち合わせを行う必要があることを示している．同期位置ごとに同期を取るシュレッドの組合せを示す情報と他のシュレッドの現在の同期位置を知る手段を持っていれば，指定されたシュレッド間だけで同期をとることが可能である．しかし，保持すべき情報量やプロセッサ間の結線のハードウェアを増加させないために，本機構では本来は待ち合わせの必要のない同期位置であっても，他に同期を

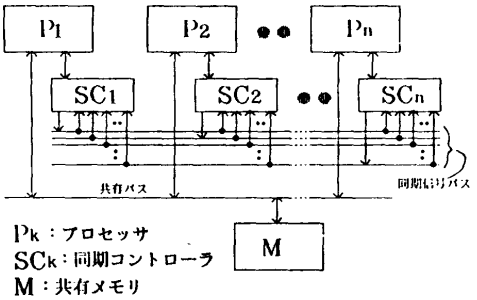

図1 同期機構を含むシステム
Fig. 1 The system including synchronization mechanism.

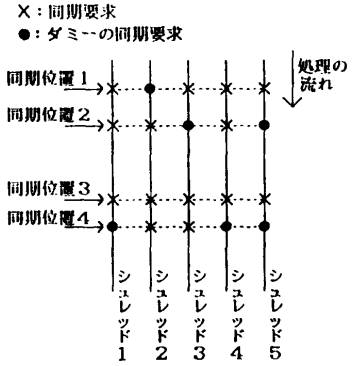

図2 ダミーの同期要求の挿入
Fig. 2 Insertions of dummy synchronization requests.

必要とするシュレッドがある位置にはダミーとして同期を要求する同期情報を挿入する（図2参照）．これにより一つのプロセスに属する全プロセッサが毎回待ち合わせを行えばよく，ハードウェア機構はバリア同期と全く同じ簡略なもので済む．

3.3 同期情報の拡張

3.4 節で述べるオーバヘッドの削減を可能にするために，同期情報を1ビットから2ビットつまり4種へ拡張する．4種の同期情報の動作と意味を以下に示す．

NONE： プロセッサは同期に関し何もしない．

RREQ： プロセッサは同期情報を出力後，同期コントローラから実行継続を許可する信号が返るまで命令実行中断状態になる．(Real REQuest)

APRV： プロセッサは同期情報を出力後，命令実行を続ける．他のプロセッサ上のタスクの実行終了を待つ必要のない場合，つまりダミーの同期要求や生産者・消費者の生産者側である場合の同期情報．意味は次回の同期成立の承認．(APpRoVal)

PREQ： プロセッサは同期情報を出力後，命令実行を続ける．RREQ の前に挿入して，後続の RREQ を予告するための同期情報．意味は次回の同期成立の先取り．(PreREQuest)

同期情報が RREQ の時のみプロセッサは同期コントローラから実行継続を許可する信号が返るまで実行中断状態になる．

3.4 カウンタによる機構の拡張

拡張された同期情報を使って，コントローラがプロセッサと独立に同期信号バスを管理して，オーバヘッドが発生する可能性を低減する．そのために，コントローラを三種のカウンタで拡張する．拡張された同期情報の使用法と三種のカウンタの紹介を兼ねて，これらを用いてオーバヘッドを削減する方法を例を使って説明する．なお今後，同期レジスタに登録されたプロセッサ群に対応する同期信号バスのすべてのラインがアクティブになることを同期条件の成立と表現する．

図3はダミーの同期要求に伴うオーバヘッドを削減する方法を示している．×印は RREQ が挿入されている場所を示し，シュレッド2はダミーの同期要求を同期位置2の付近に挿入する必要がある．同期情報を拡張する前の機構では，同期位置2に対応するシュレッド2上の地点を実行前に見積もり，RREQ をダミーとしてその地点(黒丸)に挿入する．しかしながら，実行時の予期できない要因により実際の同期位置2がずれて，プロセッサに不必要な待ちを生じさせる可能性がある．RREQ の代りに APRV を用い，図の白丸の位置に挿入すれば，オーバヘッドを避けることができる．シュレッド2の同期コントローラは同期位置1の同期の完了後すぐに APRV をプロセッサから受け取り，これにより同期位置2のために同期信号ラインをアクティブにする．一方，シュレッド2に対応するプロセッサはコントローラとは独立に命令の実行を継続する．同じシュレッド上で APRV が連続することを許すために，**同期承認カウンタ**が設けられ，同期条件未成立の APRV の数を計数する．同期承認カウンタはコントローラが APRV を受け取る度に1だけインクリメントされ，同期信号ラインをアクティベイトする度にデクリメントされる．そして，同期承認カウンタが0になるまでコントローラはアクティベイトを続ける．

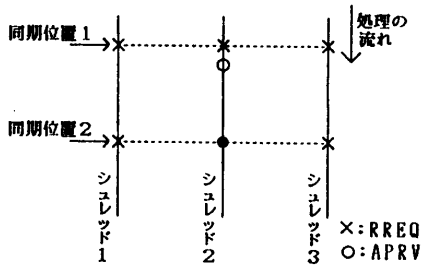

図3 ダミーの同期要求に伴うオーバヘッドの削減法
Fig. 3 Overhead reduction using APRV in the case of dummy requests.

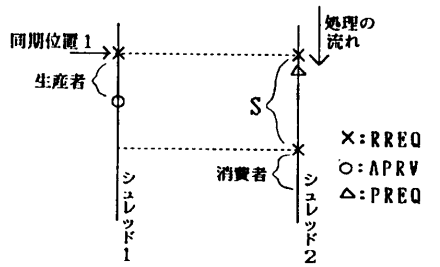

図4 生産者-消費者の依存に伴うオーバヘッドの削減法
Fig. 4 Overhead reduction using APRV and PREQ in the case of producer-consumer type synchronization.

図4は生産者-消費者の依存を守る同期に伴うオーバヘッドの削減を示している．シュレッド1上の APRV（白丸）が生産者に対応し，シュレッド2の二番目の RREQ（×印）が消費者を表している．共有メモリ上でデータの受渡を行う場合，生産者は対応する

消費者が処理を始めるのを待つ必要がないので，生産の完了を示す同期情報として APRV を使う．PREQ を使用せず，シュレッド1の生産者が処理を先に終了した場合，この同期に対応する同期条件の成立はシュレッド2の二番目の RREQ の地点まで待たされる．しかし，生産者-消費者の同期の性質から考えると，同期条件は図の領域Sのどこで成立しても構わない．そして，同期条件が早目に成立すればするほど，オーバヘッドが削減できる可能性は増える．そこで，図の三角印の地点に PREQ を挿入し，同期条件が PREQ と対応する RREQ で囲まれる範囲の任意の地点で成立可能にする．シュレッド2の同期コントローラは PREQ を受け取ると同期信号ラインをアクティベイトし，一方 APRV のときと同様にプロセッサは命令の実行を継続する．プロセッサは命令の実行が RREQ に到達したときに初めて，継続する命令が実行可能かコントローラに問い合わせる．同期条件がその間に成立していれば，コントローラは命令続行の許可をすぐにプロセッサに出力する．もし成立していなければ，コントローラはプロセッサを同期条件が成立するまで wait 状態で待たせる．さらに，同期条件の成立順序が保たれる範囲で，RREQ と対応する PREQ の間に他の同期情報が挿入可能なように，二つのカウンタを用いて拡張する．この拡張により同期条件が成立する範囲が広げられるので，オーバヘッドの生じる可能性がより低減できる．カウンタのひとつは同期条件未成立の PREQ の数を計数する**同期予告カウンタ**で，もうひとつは RREQ の前に PREQ によって成立した同期条件の数を計数する**同期成立カウンタ**である．同期予告カウンタは同期承認カウンタとまったく同じ動作（APRV を PREQ で置き換えた動作）を行う．同期成立カウンタはコントローラが APRV 以外による同期条件の成立を検出したときに1だけインクリメントされ，プロセッサに実行継続許可の信号を出力する度にデクリメントされる．同期成立カウンタが0ならば，実行継続許可の信号を出力しない．

3.5 同期コントローラの構成

図5に同期コントローラの構成とプロセッサとの結線を示す．プロセッサは前述の同期情報をそれが付加された命令の実行直前に SSIG0(Synchronization SIGnal), SSIG1 の信号線を通して，同期コントローラに対して出力する．同期情報が RREQ の時のみプロセッサは同期コントローラから SACK (Synchronization ACKnowledge) 信号が返るまで実行中断状態になる．プロセッサが同期情報のためのプリフィックス命令または命令コード内の新しいフィールドを解釈する時間的コストは，プロセッサ内に複数命令のディスパッチ回路と同期情報の解釈回路または命令解読と並行して動作する解釈回路を追加することにより消去することができる．

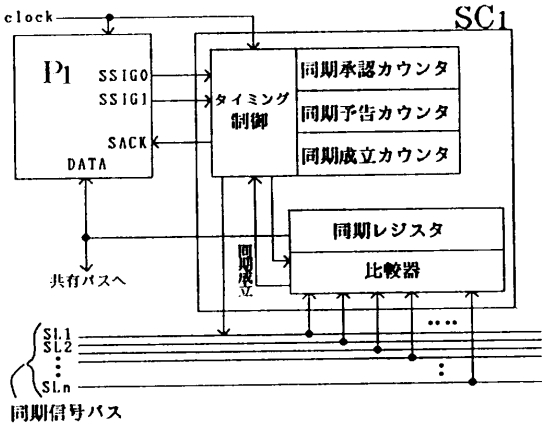

図5 同期コントローラの構成
Fig. 5 Synchronization controller for Elastic Barrier.

図6に同期コントローラの動作アルゴリズムを示す．簡単のため同期コントローラは共通クロックで動作していると仮定し，クロックごとに同期情報の受付と同期条件の成立のチェックを行い，図中の一連の動作を実行する．理解を容易にするために同期情報を受けとった場合と同期条件が成立した場合に分けて記述してあるが，実際には1クロック内の動作として一つにまとめられる．activate(SL) は同期信号ラインをアクティブにする動作で，すでにアクティブになっている場合は何もしない．同様に，nagate(SL) は同期信号ラインをインアクティブにする動作である．output(SACK) はプロセッサに SACK を出力する動作，waiting(SACK) はプロセッサが SACK 待ちの wait 状態の時に真になる論理関数を示す．

また，同期機構が命令の実行順序を保証するため，同期コントローラには以下の制約がある．プロセッサから同期情報が入力されても，プロセッサがそれ以前に実行した命令に伴うデータ通信が通信路の遅延等で完了していないときは，同期コントローラはその通信が完了するまで同期信号バス上の自分のラインをインアクティブからアクティブに変更する動作は行わない．つまり，この制約で先に実行されたデータの通信を同期のための信号が追い越すことがないことを保証している．

```
0.プロセス実行開始時 (初期化)
CCP = 0; CAP = 0; CPR = 0; negate(SL);

1.同期情報受け付け時
swich (SSIG) {
case RREQ:
    if (CCP > 0) {
        CCP--;
        output(SACK);
    } else activate(SL);
    break;
case APRV:
    CAP++;
    activate(SL);
    break;
case PREQ:
    CPR++;
    activate(SL);
    break;
}

2.同期条件成立時
if (CAP > 0) {
    CAP--;
    if ((CAP == 0) && (CPR == 0))
        negate(SL);
} else {
    if (CPR > 0) {
        CPR--;
        if (waiting(SACK)) output(SACK);
            else CCP++;
        if (CPR == 0) negate(SL);
    } else {
        output(SACK);
        negate(SL);
    }
}

CCP: 同期成立カウンタ
CAP: 同期承認カウンタ
CPR: 同期予告カウンタ
SL: コントローラの同期信号ライン
```

図 6 同期コントローラ動作アルゴリズム
Fig. 6 Algorithm of synchronization controller's action.

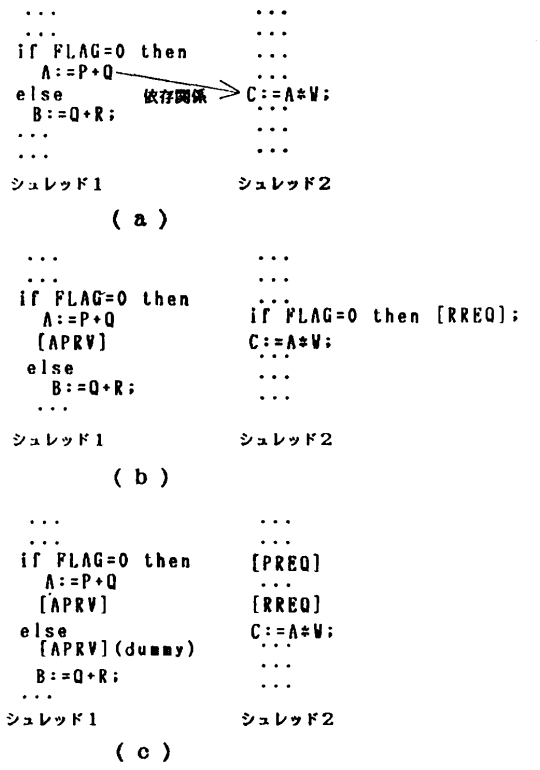

図 7 分岐がある場合の同期情報の挿入法 [RREQ], [APRV], [PREQ] は各同期情報の挿入場所を示す.
Fig. 7 Insertion methods of synchronization information in the case of existence of conditional branches.

3.6 同期情報の挿入パターンの性質

Elastic Barrier の機構はバリア同期を基本としており，同期コントローラは同期レジスタに登録されたグループ（プロセス）内で同期信号バス上に同期ごとにバリアを張っている．つまり，コントローラは必ずグループに対応するすべての同期信号ラインがアクティブになることを確認している．このため，同一プロセス内のシュレッドは，同じ回数だけ同期信号ラインを ON/OFF する必要がある．よって，プロセッサが RREQ に出会う回数と APRV に出会う回数の合計はシュレッド間で同期ごとに揃っていなければならない．

分岐については，すべてのシュレッドがバリア同期して同一条件で分岐する場合を除いて，プロセッサがシュレッド内の任意のパスを実行しても，同一回数だけ同期信号ラインが ON/OFF されるように同期情報を挿入する必要がある．つまり，**図 7**(a) のような場合，条件が成立したときだけ，データの依存関係からシュレッド1と2の間で同期が必要である．このように条件によって同期の必要性が変化する場合は，図7(b)のように同じプロセス内のすべてのシュレッドで同一の条件判断を行わせるか，図7(c)のように同期の必要のないパスにもダミーの同期要求 (APRV) を挿入する必要がある．

RREQ が常に対応する PREQ を持つことにすると，命令ごとに同期をとるような場合，PREQ のタグを付けるための NOP (No OPeration) 命令や意味のない PREQ プリフィックス命令を挿入する必要がでる．このため，3.5 節のアルゴリズムは RREQ が必ずしも対応する PREQ を持つ必要がないように作成されている．ただし，PREQ は必ず同じシュレッド内に対応する RREQ を持ち，PREQ と対応する RREQ の間には PREQ を持たない RREQ は存在できない．

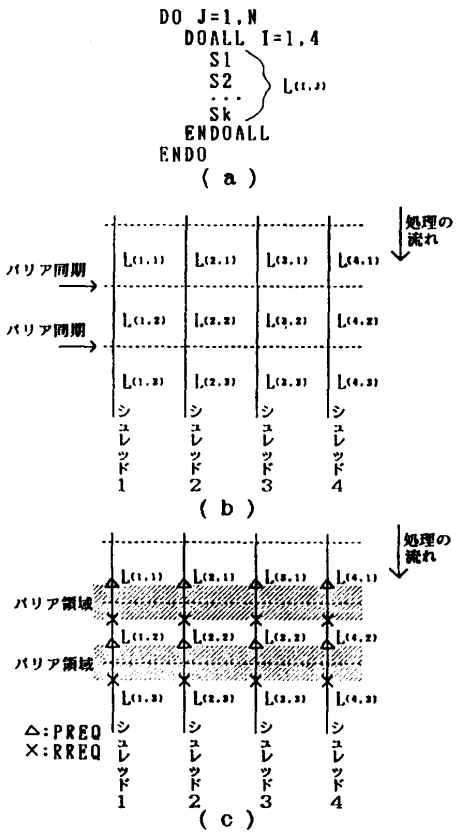

図 8 ファジーバリアとしての使用法
Fig. 8 The usage as Fuzzy Barrier.

4. バリア同期のオーバヘッドの緩和

Elastic Barrier を使用して，同じプロセス内のすべてのシュレッドが RREQ を発行すれば，各シュレッドの RREQ の発行地点でバリア同期が行える．このとき，RREQ を発行した点で厳密にバリア同期をする必要があるならば，オーバヘッドは同期機構の電気的な信号の遅延程度である．しかし，R. Gupta が述べているように[12]バリア同期といっても通常はある一点だけで待つ必要はなく，ある広がりのある領域内で一回の同期が成立すればよい．例を用いて説明する．**図 8**（a）のように DOALL（イテレーション間に実行順序の制約のない DO ループ）と DO（順次）の二重ループになったプログラムの場合，内側の DOALL の各イテレーションを一つのプロセッサ（シュレッド）に割り当てて並列化できる．このとき，外側の DO の順次性を保証するため，DOALL のイテレーションの最後にバリア同期が必要になる（図 8（b））．しかし，イテレーション間で依存する配列変数の生成と参照がそれぞれ図 8（c）の三角印と×印の

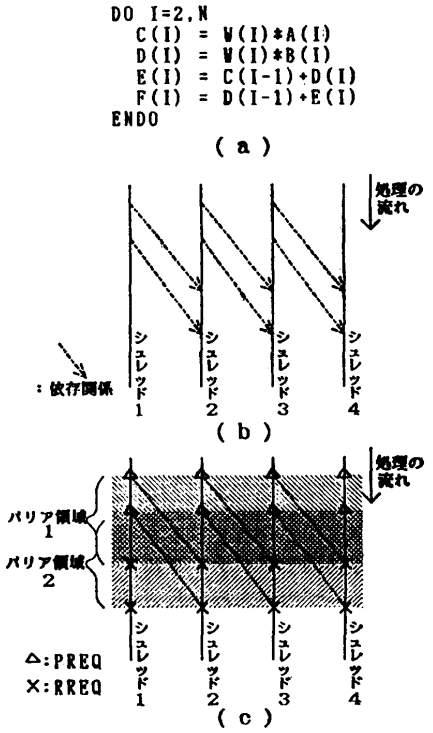

図 9 重なりあるバリア領域
Fig. 9 Overlapping barrier regions.

地点でなされているとすると，この両地点で挟まれた斜線の領域内の任意の時点で一回だけ同期が成立すれば十分である．バリア同期を行う時点がプログラム上の一点ではなく，幅を持っているので，オーバヘッドが吸収される可能性が高くなる．この種のバリア同期（Fuzzy Barrier）も本同期機構で PREQ と RREQ を使って簡単に実現できる．図 8（c）の三角印に PREQ を×印に RREQ を挿入して，広がりのあるバリア領域を PREQ と RREQ で囲めば，Fuzzy Barrier が実現できる．さらに，カウンタの効果によって，Elastic Barrier が R. Gupta の提案している Fuzzy Barrier の機構[12]よりもオーバヘッドを削減できる可能性が高くなる場合を示す．**図 9**（a）のようなループに対し，イテレーションごとにプロセッサを割り当てると図 9（b）のような Lexically Forward の依存関係を持つ．この場合，Elastic Barrier では図 9（c）のように広がりのあるバリア領域のオーバラップが可能で，バリア領域を広げることができる．

5. オーバヘッド吸収能力の限界

本機構はバリア同期を基本としており，グループとして登録されているプロセッサに対応するすべての同

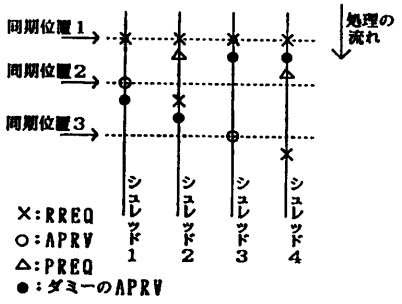

図 10 独立した二つの依存の同期
(図は実行前の見積時の処理の流れ)
Fig. 10 Two independent requests for synchronization.
(This figure shows estimation of process flow at compilation time.)

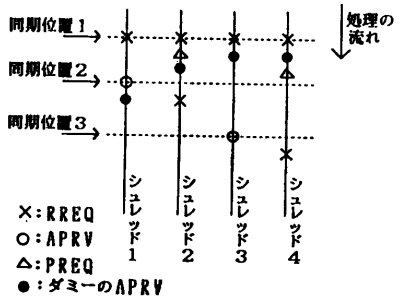

図 11 PREQ と RREQ の間に APRV が存在する場合
(図は実行前の見積時の処理の流れ)
Fig. 11 The case of APRV existence between PREQ and RREQ.

期信号線が各同期でアクティブになる必要がある．このことはグループ内で実行される同期の順序が決められていることを意味する．そこで，実行前に見積もった同期の順序が狂うほど大幅な遅延が実行時に発生した場合は，オーバヘッドが生じる可能性がある．

この点について**図 10** を用いて説明する．図 10 は同期情報を挿入するために処理の流れを見積もった様子を示している．シュレッド 1 と 2 およびシュレッド 3 と 4 がそれぞれ生産者-消費者の関係にあり，コンパイル時の見積ではシュレッド 1 の '生産' (計算結果の共有変数への代入等) がシュレッド 3 の生産より早く完了すると判断された場合の同期情報の挿入状態を示している．つまり，シュレッド 1 の生産は図中の同期位置 2 の丸印で完了し，シュレッド 3 は同期位置 3 の丸印で完了すると見積もられた．ところが，コンパイル時に予見できなかった要因によって，シュレッド 1 の実行に同期位置 1 と 2 の間で大幅な遅延が生じ，シュレッド 1 がシュレッド 3 と 4 の同期のためのダミーの APRV (黒丸) に到達する前に，シュレッド 3 と 4 がそれぞれ APRV (白丸) と RREQ (×印) に到達したとする．このとき，シュレッド 2 も二つ目の RREQ (×印) で実行を中断している．シュレッド 4 はシュレッド 3 との依存関係は解決しているが，シュレッド 1 およびシュレッド 2 がダミーの APRV に到達するまで待たされることになる．シュレッド 1 (か 2) と 4 の間にこの後すぐに生産者-消費者の関係があれば，たとえシュレッド 4 がこの時点では待たずに先へ進めたとしても，すぐにシュレッド 1，2 を待つ必要が生じる．このため，シュレッド 4 の待ちが処理全体としてのオーバヘッドと即座に断定できるわけではない．ただ，シュレッド 1，2 と 4 の間で同期がしばらく必要でなければ，この不必要なシュレッド 4 の待ちが他のシュレッドにも影響を及ぼし全体的なオーバヘッドとなる可能性は高くなる．つまり，シュレッド 1，2 と 4 は実はある程度独立に実行可能であるにもかかわらず，同じ同期のグループに属していることがオーバヘッドの生じる可能性を高めている．

このオーバヘッドを低減するための方法の一つとして，同期レジスタをユーザが更新可能にし，ユーザのプログラム実行中に動的に同期を取り合うグループを変更するという方式[13]が考えられる．

6. FIFO キューを使った機構の拡張

3 章で述べた Elastic Barrier の機構はコントローラ内部で同期承認カウンタと同期予告カウンタによって同期条件未成立の APRV と PREQ の回数をそれぞれ管理している．しかし，シュレッド内の APRV と PREQ の出現順序までは記憶していない．このため，PREQ とペアの RREQ の間に APRV を挿入すると不必要なプロセッサの待ちを生じる可能性がある．**図 11** を用いて具体的に説明する．図 11 は 5 章で用いた図 10 とプログラムの意味的には同じ状況を表している．違いはシュレッド 2 のダミーの APRV の位置が PREQ と RREQ のペアの間に移動している点だけである．この移動があっても同期信号線を ON にする PREQ と APRV のシュレッド上の順序は不変なので，同期は保証される．しかし，同期による動作タイミングは変化する可能性がある．シュレッド 1 が同期位置 2 の APRV に到達したとき，遅延が生じなければ，シュレッド 2 は PREQ とダミーの APRV の両方を通過している．そこで，同期承認カウンタと同期予告カウンタが共に 1 になっている．同

期位置2の APRV によって同期条件が成立した後，3.5 節のアルゴリズムに従えば，シュレッド2の同期コントローラでは同期承認カウンタが0にデクリメントされ，同期成立カウンタはインクリメントされない．このためシュレッド2の RREQ は3回目の同期条件が成立するまで（見積どおりならシュレッド3が同期位置3の APRV に到達するまで）通過できないことになる．シュレッド2におけるこの不必要な待ちは，同期コントローラが APRV と PREQ の出現順序を管理していないことに起因している．

同期承認カウンタと同期予告カウンタの代りに FIFO キューを用いて，同期条件が成立するごとにキューの先頭の同期情報に応じた動作を行うことにする．APRV と PREQ に到達する度に，その同期情報をキューに入力する．そして，同期条件が成立した時に，キューから同期情報を1個取り出し，PREQ であれば同期成立カウンタをインクリメントし，APRV であればカウンタの操作はしない．取り出した後も，まだキューが空でなければ，同期信号線を次の同期条件の成立のために再び ON にする．図 11 の例の動作を説明すると，シュレッド2においては PREQ, APRV の順で同期情報に到達し，この順番でキューに入力される．同期位置2においてシュレッド2のキューの先頭は PREQ なので，同期条件の成立によって同期成立カウンタがインクリメントされ，シュレッド2の RREQ は3回目の同期条件が未成立でも通過できる．このように不必要な待ちが削減できる．以上からわかるように，FIFO キューを用いれば不必要の待ちの発生する危険性なしに，ペアの PREQ と RREQ の間に APRV が挿入可能，つまり，RREQ からより離れた地点に PREQ が挿入可能である．

FIFO キューの方がカウンタよりもハードウェア的な構造が複雑なので，コントローラの動作周波数を決める際のクリティカル・パスになる危険性がある．そのため，キューとカウンタのどちらのアプローチを取るかは，ハードウェア量（および動作周波数）と機能とのトレードオフである．以下の議論では Elastic Barrier の機構はカウンタの方式を採用しているものと仮定する．

7. Elastic Barrier の下でのプリエンプション方式

Elastic Barrier の大きな特徴の一つにマルチジョブとの整合性の良さがある．つまり，OS の下で Elastic Barrier を利用したプロセスをプリエンプトし，解放されたプロセッサを他のプロセスに割り当て，実行させることが可能である．このプリエンプション方式について述べる．ただし，プロセスはコンピュテーションインテンシブで周辺装置の入出力は頻度が少なく，プリエンプションはプロセッサのマシンサイクルに比べて十分に長いタイムスライスによって主に起こり，プロセス切替のオーバヘッドはシステムの性能上は無視できると仮定する．

OS によるプロセッサ資源の管理を前提としており，マルチジョブ，マルチユーザの環境がシステムとして実現されることを仮定している．このため，OS はプロセッサ資源の再スケジューリングのために各プロセッサに対して外部割り込みを発行する機能を持ち，プロセスは実行の途中でプリエンプトされる．ただし，Elastic Barrier は極力オーバヘッドを減らすために，同期待ちが必要なプロセッサをハードウェア的に wait 状態で待たせることにしているので，OS によるプリエンプションを常に可能にするためには，wait 状態でも受け付けられる外部割り込みが必要である．

プロセスが実行途中でプリエンプトされるので，外部割り込み時に同期機構の内部状態を含んだコンテクストを実ハードウェアに対して退避／復旧する必要がある．しかし，実行中のプロセスには複数のプロセッサと複数の同期コントローラが割り当てられており，これらのハードウェアに関するコンテクストをある瞬間で完全に切り出すことは困難である．つまり，割り込みを受け付けて内部状態をフリーズする時点がプロセッサやコントローラごとにずれる可能性がある．

この時間のずれが許容できるようにプリエンプション方式を構成する必要がある．この時，問題となるのは以下の点である．Elastic Barrier の機構はプロセッサとコントローラ，そしてコントローラ間で信号の送受を行っているが，オーバヘッドやハードウェア量の観点から，これらの信号の送受にはハンドシェイク等の確認作業がない．このため，プロセッサやコントローラごとの待避される内部状態の時間のずれは，これらの間の信号通信に矛盾を生じる可能性がある．すなわち，信号の送り手は送信済みの状態で内部状態の退避が行われ，その信号の受け手は未受信の状態で内部状態の退避が行われる可能性がある．この状況に陥るとプロセスの再実行は不可能である．

この問題点の解決策を提示する．まず，コントロー

ラとプロセッサ間でレース条件が起こらないように，割り込みの受付を順次化する．つまり，外部割り込みをコントローラ側で受け付け，プロセッサの割り込みのアサートはコントローラを介して行う．コントローラが割り込みを受け付けた時点で，コントローラはプロセッサからの RREQ の受け付けを中止する．つまり，RREQ が入力されても内部状態や同期信号ラインの状態を変えず，SACK も出力しない．RREQ の受け付けを中止した後に，プロセッサの割り込みをアサートする．

次に，同期機構内の内部状態が安定したことを確認する方法を提示する．プロセス内に含まれる同期機構を全体的に見た場合，外部入力はプロセスに属するプロセッサからの同期情報のみである．ところで，外部割り込みによって起動される割り込み処理ルーチンの先頭部はコンテクストの退避等を行うスケジューラのプログラムの一部である．そして，ここで同期コントローラのカウンタの退避／復旧やレジスタの設定も行われるので，このルーチン内ではプロセッサ間の同期に Elastic Barrier を使用することはない．そこで，プロセスに属する全プロセッサが割り込みルーチンに切り替わってしまえば，新たな同期情報入力はなくなり，同期機構の内部状態は固定化する．つまり，割り込みルーチン内でコントローラの内部状態を退避するまえに，プロセスに属していた全プロセッサの間で Elastic Barrier の機構を用いずに（共有メモリ等で）バリア同期を行えばよい．

以上，プロセス切替に伴うコントローラの内部状態の退避に関する注意点を中心に述べたが，内部状態の再設定についても同様の注意が必要である．上述の方式に沿ったプリエンプション時の各プロセッサにおける割り込み処理の一例を以下に示す．

1．同期コントローラに外部割り込みが入力されると，コントローラは以後 RREQ の受け付けの中止，つまり同時に SACK の出力を中止し，プロセッサの INT をアサートする．ただし，他の APRV，PREQ の同期情報は受け付け，内部的な処理を続行する．
2．プロセッサが割り込みを受け付け，割り込みルーチンに制御を移す．
3．プロセスの再割り当て時に内部状態を復旧するのに必要なプロセッサ内のコンテクストをメモリへ退避．
4．同一プロセスに属していたプロセッサと共有メモリを介してバリア同期．
5．同期コントローラ内のコンテクスト（カウンタ）をメモリへ退避．
6．同期コントローラのカウンタを初期化（ゼロクリア）し，INT をネゲートして，RREQ の受け付け中止を解除．
7．同一プロセスに属していたプロセッサと共有メモリを介してバリア同期．
8．次に実行するプロセスのプロセッサとコントローラ用のコンテクストを獲得．
9．新しいプロセス用に同期コントローラの同期レジスタを設定．
10．新しいプロセスに属するプロセッサと共有メモリを介してバリア同期．
11．同期コントローラのコンテクスト（カウンタ）を復旧．
12．新しいプロセスに属するプロセッサと共有メモリを介してバリア同期．
13．プロセッサのコンテクストを復旧．
14．割り込みルーチンから抜け，新しいプロセスのシュレッドに制御を移す．

8. 他の細粒度タスク用の静的同期機構との比較

細粒度のタスクの同期を扱い，かつタスクが静的にプロセッサにスケジューリングされていることを前提とする同期機構に前出の Fuzzy Barrier の機構と静的実行順序制御機構[14),15)] がある．これらとの比較を簡単に述べる．

Elastic Barrier は Fuzzy Barrier の機構とは独立に考案されたものであるが，見方によっては Fuzzy Barrier を APRV という新しい同期情報と三種のカウンタでより一般化かつ拡張したものととらえることができる．バリア同期だけではなく一般の依存関係に基づく同期を効率良く行うために，これらの一般化と拡張がなされている．Elastic Barrier では単に APRV を挿入すればシュレッドが独立に実行できるような同期に対しても，Fuzzy Barrier では APRV と同期承認カウンタがないために，必ず一回ごとに RREQ に相当する wait を伴う可能性のある同期情報を挿入する必要がある．このため，オーバヘッドの生じる可能性も高く，同期情報の挿入位置を決定するための手間も多い．また，Fuzzy Barrier にはカウンタに

よる拡張がないため，同期の制御部をプロセッサと独立に弾力を持たせて（elastic に）動作させる能力が低い．ハードウェア量的には，Elastic Barrier は同期情報として2ビット必要であるが，Fuzzy Barrier は APRV に相当するものがなく，RREQ には必ず対応する PREQ を持たせているため，奇数回目の同期要求は PREQ として偶数回目は RREQ として使用して，同期情報を1ビットで済ませている．また，Elastic Barrier はプロセッサごとに3個のカウンタ（ビット幅はデザイン時の決定事項であるが，実用的には4ビット程度）がハードウェア的な増分として必要である．ただし，物理的な IC のパッケージの境界を越えてマルチプロセッサを構成する場合に大きな問題となる要素プロセッサ間の同期信号のための結線数は，共にプロセッサ台数と同じ本数で済む．

静的実行順序制御機構は Elastic Barrier の用語で説明すると，APRV と RREQ に相当する同期情報を持ち，それらの同期情報には同期相手に関する情報が含まれ，APRV（実行の継続を許可するトークン）は指定されたプロセッサに対して発行され，RREQ も指定されたプロセッサからの APRV を待つのに用いられる．このように，同期相手が一対一で指定される点が，最大の相違点である．機構としては，プロセッサ間を双方向に完全結合で結び，結合枝ごとにカウンタを設け，APRV の多重発行を許している．依存関係上で先行するプロセッサが後続するプロセッサへの結合枝に APRV を発行し，その結合枝のカウンタをインクリメントし，先行側のプロセッサは実行を継続する．後続側のプロセッサは RREQ を用いて，その結合枝のカウンタの値を調べ，0でなければデクリメントして実行を継続し，0であれば先行側から APRV が発行されるまで実行を中断して待つ．この順序制御機構では任意の2台のプロセッサ間での同期トークンの受渡しが可能なので，5章で述べたような独立した先行関係間になんらかの順序関係を導入する必要はない．一方，Elastic Barrier では自然に行うことができるバリア同期等の3台以上のプロセッサが関連する同期を行うのにコストが掛かる．複数のトークンを発行する必要があるので，同期情報で複数のプロセッサを指定可能にするか，複数の同期情報をひとつの同期に対して挿入する必要がある．ハードウェア量的には，同期情報が NONE, APRV, RREQ の2ビットに加えて，同期相手のプロセッサを指定するビットが必要である．プロセッサ台数を n とすると，プロセッサを一台のみ指定するなら $\log(n)$ ビット，任意の組合せで指定するなら n ビットだけプロセッサの指定に必要である．さらに，全体でカウンタが $n(n-1)$ 台，同期のためのプロセッサ間の結線数が $2n(n-1)$ 本（ただし，係数の2はカウンタの溢れを検出する線を含む）必要になる．

また，Elastic Barrier と Fuzzy Barrier は同期信号ラインにマスクをかけることで，同期信号パスのグループ分けが可能である．このことから，プロセッサ台数を指定するスケジューリングによって，容易かつ同時に複数のプロセスをマルチプロセッサ上で処理できる．これに対して，順序制御機構は同期相手のプロセッサを一意に指定するため，複数のプロセスを同時に処理するためにはプロセッサの指定の仮想化が必要で，ハードウェアで仮想プロセッサと実プロセッサの対応をとる機構を用意する必要がある．

9. おわりに

密結合型マルチプロセッサ上で細粒度の並列実行を効率よく実現するための Elastic Barrier（一般化されたバリア型同期機構）の構成と動作について述べた．この機構は同期を取り合うプロセッサ群内での同期の発生順序が静的に決定できれば，オーバヘッドなしに（多くとも電気的な信号の遅延程度で）同期を取ることができる．また，Elastic Barrier の広がりのあるバリアとしての使用法，オーバヘッド吸収能力の限界，FIFO キューを用いた機構の拡張法，他の同期機構との比較を提示し，機構の特徴を明らかにした．さらに，Elastic Barrier を使用したシステムにおいて，要素プロセッサが wait 状態でも受付け可能な外部割り込みを持っていれば，その割り込みを用いて実行途中のプロセスのコンテクストの退避／復旧が可能で，時分割によるマルチジョブが実現できることを示した．

謝辞 いつも御討論いただいている森山孝男氏と渦原茂氏，研究の機会を与えてくださった上村務氏と鈴木則久所長に感謝いたします．また，著者を研究生として御指導いただいている東京大学工学部の田中英彦教授，貴重な助言を提示された査読者の方々に深く感謝いたします．

参考文献

1) Ellis, J. R.: *Bulldog: A Compiler for VLIW Architectures*, The MIT Press (1986).
2) Cytron, R. G.: Doacross: Beyond Vectoriza-

tion for Multiprocessors, *Proc. 1986 Int. Conf. Parallel Processing*, St. Charles, IL, pp. 836-844 (Aug. 1986).
3) 本多弘樹ほか：OSCAR 上での Fortran プログラム基本ブロックの並列処理手法，信学論（D），Vol. J73-D-I, No. 9, pp. 756-766 (Sep. 1990).
4) 松本　尚：細粒度並列実行支援マルチプロセッサの検討，情報処理学会論文誌，Vol. 31, No. 12, pp. 1840-1851 (1990).
5) Teixeira, T. J. and Gurwitz, R. F.: Stellix: UNIX for a Graphics Supercomputer, *Proc. of the Summer 1988 USENIX Conf.*, pp. 321-330 (Jun. 1988).
6) Beck, B. et al.: VLSI Assist for a Multiprocessor, *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 10-20 (Oct. 1987).
7) Polychronopoulos, C. D.: Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design, *IEEE Trans. Comput.*, Vol. 37, No. 8, pp. 991-1004 (Aug. 1988).
8) 工藤知宏，天野英晴：ATTEMPT の同期機構，第 38 回情報処理学会全国大会論文集，pp. 1462-1463 (Mar. 1989).
9) 松本　尚：細粒度並列実行支援機構，情報処理学会計算機アーキテクチャ研究会報告，No. 77-12, pp. 91-98 (Jul. 1989).
10) 森山孝男ほか：粒度を考慮したマルチプロセッサの資源管理，情報処理学会計算機アーキテクチャ研究会報告，No. 83-18, pp. 103-108 (Jul. 1990).
11) Arenstorf, N. S. and Jordan, H. F.: Comparing Barrier Algorithms, *Parallel Computing*, Vol. 12, No. 2, pp. 157-170, North-Holland (Nov. 1989).
12) Gupta, R.: The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors, *Proc. Third Int. Conf. on ASPLOS*, pp. 54-63 (Apr. 1989).
13) 松本　尚：一般化されたバリア型同期機構の諸問題について，並列処理シンポジウム JSPP '90 論文集，pp. 49-56 (May 1990).
14) 有田隆也ほか：PN プロセッサにおけるフロー制御方式について，第 39 回情報処理学会全国大会論文集，pp. 1896-1897 (Oct. 1989).
15) 高木浩光ほか：問題が持つ先行関係のみを保証する高速な静的実行順序制御機構の構成法，並列処理シンポジウム JSPP '90 論文集，pp. 57-64 (May 1990).

（平成 2 年 9 月 3 日受付）
（平成 3 年 2 月 12 日採録）

松本　尚（正会員）

1962 年生．1985 年東京大学工学部計数工学科卒業．1987 年大阪市立大学大学院理学研究科物理学専攻修士課程修了．同年より日本アイ・ビー・エム（株）東京基礎研究所に勤務．現在，研究生として東京大学大学院工学研究科に在籍．並列計算機アーキテクチャ，オペレーティングシステム，並列化コンパイラに関する研究に従事．他にニューラルネットワーク，学習，力の超統一理論等に興味を持つ．本学会学術奨励賞受賞．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．