計算機アーキテクチャ 96－21
(1992. 10. 23)

マルチプロセッサシステム上の無同期細粒度並列処理

尾形 航 †, 岡本 雅巳 †, 本多 弘樹 ‡, 笠原 博徳 †, 成田 誠之助 †

†早稲田大学 理工学部 情報科 , 〒 169 東京都 新宿区 大久保 3-4-1

‡山梨大学工学部電子情報工学科 , 〒 400 山梨県 甲府市 武田 4-37

あらまし

マルチプロセッサシステム上でFortranプログラム中の基本ブロックを並列処理する手法として、従来よりコンパイル時のスタティックスケジューリングを用いた細粒度並列処理手法が提案されている。しかし、従来の方式ではタスク間のデータ依存に基づく先行制約を保証するため並列プログラム中に同期コードを埋めこまねばならず、その実行によるオーバーヘッドが比較的大きいという問題があった。本論文ではスケジューリングの精度を引き上げマシンクロックレベルでの命令実行の最適化を可能とすることにより、すべての同期コードを除去する事でオーバーヘッドを低減する手法について提案する。又、本手法を実マルチプロセッサシステムOSCAR上でインプリメントし、無同期実行の効果を検証した結果についても報告する。

和文キーワード 細粒度並列処理, スケジューリング, 並列化コンパイラ, マルチプロセッサ,Fortran プログラム, アーキテクチャサポート

Near Fine Grain Parallel Processing on a Multiprocessor System Without Synchronization

Wataru OGATA †, Masami OKAMOTO †, Hiroki HONDA ‡
Hironori KASAHARA †, Seinosuke NARITA †

†School of Science & Engineering,Waseda University, 3-4-1 Okubo Shinjuku-ku Tokyo, 169 Japan

‡Faculty of Engineering,Yamanashi University, 4-37 Takeda, Kofu-shi, Yamanashi, 400 Japan

Abstract

The near fine grain parallel processing scheme using static scheduling algorithms has been proposed to process a Fortran basic block in parallel on a multiprocessor system. However, the scheme suffers from relatively large synchronization overhead since synchronization codes must be inserted into a parallel machine code to satisfy precedence constraints caused by data dependences among tasks. To cope with this problem, this paper proposes a parallel code generation scheme which removes all synchronizations by optimizing, or scheduling, execution timing of every instrucrion in a machine clock level. Furthermore, it reports performance of the parallel processing without synchronization evaluated on an actual multiprocessor system OSCAR.

英文 key words Near fine-grain parallel processing, scheduling, parallelizing compiler, multiprocessor, Fortran, architectural support

1 まえがき

計算機の性能向上に対する要求は止まるところを知らず、これに応えるため並列処理[1][2][3][4]の研究が始められて久しい。とりわけFortranなどの手続型プログラムを自動的に並列化[6][7][18]して、マルチプロセッサ上で処理するアプローチはソフトウェア資産や既得のプログラミング技術を活かしながら並列処理による性能向上を享受できることから注目を集めている。

中でもDoループのイタレーションをタスクとして並列性を抽き出すDo-Across,Do-Allなどの中粒度並列処理が従来から大きな成果をあげており[18]、さらに最近ではマクロタスク間の並列性を引き出す粗粒度並列処理が活発に研究されている[4][19]。

一方、ステートメントをタスクとして並列性を抽き出す細粒度並列処理も、CP/DT/MISF[9]やDF/IHS[10]に代表される実用的なスタティックスケジューリング手法が開発され、その有効性が実マルチプロセッサシステム上で確認されている[9]。また最近では細粒度タスク間の同期オーバーヘッドを軽減するために、ハードウェアによる高速バリア同期機構[11][12][13][14]が提案されている。さらに、同期処理そのものの数を減らすために、スタティックスケジューリング結果を利用して冗長な同期を除去する手法[5]も研究されている。

さらに基本ブロックをいくつかのセクションに分けて、セクション内の同期コードを除去する手法が、現在バデュー大学で試みられている[15]。これは、セクションの前後にバリア同期を挿入して、セクション内の命令実行タイミングとスケジューリングのずれを一定の範囲内に保つ。そしてセクションの中ではスケジューリングから最悪のずれを見込んで、同期ポイントの間に充分な余裕をとりながらスタティックスケジューリングを行い、タスク間依存を保証しつつ同期コードを除去するという方式を取っている。

このようなマルチプロセッサ上での細粒度並列処理における最終目標は、全ての同期コードを除去する、すなわち無同期で実行しつつ全てのデータ依存を満足させることである。これを実現するためには、コンパイラによるクロックレベルの命令実行、データ転送タイミングのスケジューリングと、それを可能とするハードウェアサポートが求められる。本稿では、この無同期並列処理を可能にするために設計されたマルチプロセッサシステムOSCAR上で、基本ブロック中の全ての同期コードを除去するコンパイル手法について提案する。又、OSCARで無同期実行手法の実用性を検証した結果についても述べる。

2 OSCAR Fortran 並列化コンパイラ

本手法では、OSCAR上にインプリメントされている自動並列化コンパイラ、OSCAR Fortran並列化コンパイラ[16]を改良して無同期実行を実現する。

このOSCAR Fortran並列化コンパイラは粗粒度, 中粒度, 細粒度の全ての粒度でFortranプログラム中の並列性を抽出するマルチグレイン並列処理方式を実現している。OSCAR Fortran並列化コンパイラは図1に示される論理的構造をとり、**Phase.1**でソースコードの字句解析と構文解析を行い、中間言語に変換する。**Phase.2**で中間言語の最適化と、粗粒度, 中粒度, 細粒度の各粒度で並列化を行なう。粗粒度並列処理とは、基本ブロック,Doブロック,サブルーチンをマクロタスクとして並列性を利用することを意味し、中粒度並列処理は、Doループを解析してDo-All,Do-Across のようにループイタレーション間の並列性を抽出することを意味する。

また細粒度並列処理は基本ブロック内のステートメントをタスクとして並列処理を行なうことを意味する。

Phase.3では、OSCAR用の並列マシンコードを出力する。提案するコード生成手法ではOSCARのハードの挙動を考慮しながら、クロックレベルでタスクのタイミングを調整して同期コードを挿入せずにOSCAR用のコードを生成し、無同期処理を実現する。

以下に本手法の対象であるOSCARのアーキテクチャと、細粒度並列処理の手法について説明する。

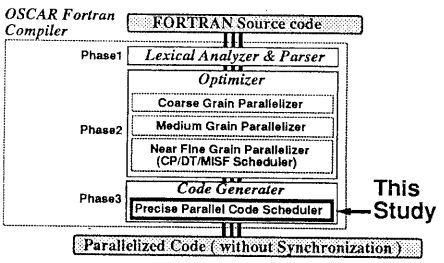

図 1: OSCAR Fortran 並列化コンパイラの構成

2.1 OSCARのアーキテクチャ

マルチプロセッサシステムOSCAR(図2)は、32bit-RISC型プロセッサ,分散共有メモリ(DPM, デュアルポートメモリ), ローカルメモリ(LM, プログラムメモリ及びデータメモリ), ダイレクトメモリアクセスコントローラ(DMAC)で構成されるプロセッサエレメント(PE)16台を、3本のバスで集中型共有メモリ(CM)に接続する構成をとっている。CPU内レジスタ間演算は、単精度浮動小数点演算の加減乗算も含めて全て1クロック($200ns$但し現在は$333ns$に設定)で処理される。浮動小数点除算は逆数近似法を用いて、10クロックで実行される。

また、OSCARでは、並列化コンパイラがPE間通信用のコードを生成するに当たって、複数の通信方式の中から通信の速度、通信コスト、通信相手の数を考慮して最適な方式を選べるように[1]DPMを用いた1PE対1PE直接データ転送, [2]1PE対全PEのブロードキャスト転送, [3]CMを用いた間接転送、の3つの転送モードをサポートしており、いずれもバスを介して他のPE上のDPMまたはCMにデータを送る。これらの転送モードを使用するためのバスアクセス命令は原則として4クロックを要する。また複数のPEがバスアクセスを同時に要求した場合には設定された優先順位に従ってハードウエア調停回路によって3本のバスに振り分けられる。さらに、各バスにはバリア同期を高速に処理するハードウェアが備え付けられている。

また、OSCARに採用されているRISCプロセッサは全ての命令を演算データによらず固定クロック数で実行するほか、他PE上のDPMにデータを送るのに要する時間や、CMをアクセスするのにかかる時間、バスアクセスが衝突したときの調停と遅延などのクロック数も固定である。これは、プログラムの動作をクロック単位で調整し、データ依存を満足させながら同期コードを除去する無同期実行を可能にするための設計である。

2.2 OSCAR Fortranの近細粒度並列処理

OSCAR Fortran並列化コンパイラは基本ブロックを近細粒度レベルで並列化するにあたって、ステートメントをタスクとしてタスク間の並列性を抽出する。

例えば図3のプログラムでは各ステートメントがタスクとして定義され、データ依存により生じる、タスク間の先行制約は図4の様なタスクグラフで表わされる。図4ではタスク10,3,2などは

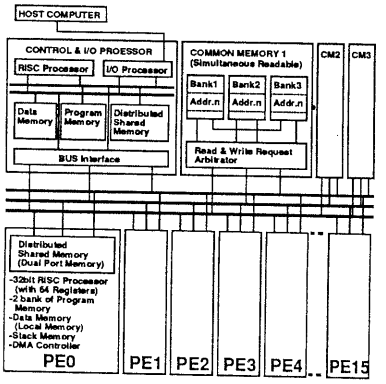

図 2: OSCAR のアーキテクチャ

互いに先行制約がないので同時に別の PE に割り当てられ並列実行することが可能であり、タスク 15 は先行制約により、10,3 を終了しないと開始できないことを表す。

次に、この先行制約を満たしてタスクを静的に PE に割り当てる。この割り当てにはリストスケジューリングの一種である CP/DT/MISF が用いられている。CP/DT/MISF は、タスクグラフから得られるクリティカルパス長とデータ転送時間などの情報を基にタスクに優先順位をつけ、アイドルプロセッサにレディタスクを静的に割り当てる方式である。

具体的には、各タスクのクリティカルパス長 (CP)、データ転送コスト (DT)、直接後続タスク数 (MISF) の 3 つをパラメータとして以下の手順で割り当てる。

Step1 cp 長に従い各タスクのレベル l_i を決定する

Step2 データ転送を考慮したリストスケジューリングを実行する

Step2.1 スケジューリング時刻 $t_{current} = 0$ として $t_{current}$ におけるレディタスクを見つけ、アイドルプロセッサに割り当てる

Step2.2 最もレベルの高いレディタスク (t_{ready}個) を、アイドルプロセッサ (m_{idle}個) に割り当てる可能な組み合わせに対して、最もデータ転送の少ない割り当てを選ぶ。

Step2.3 もし、組み合わせの候補が複数あれば直属の後続タスク数の最も多いレディタスクを選択してアイドルプロセッサに割り当てる。まだアイドルプロセッサがあれば Step2.3 を繰り返し、なければ Step2.4 を実行する。

Step2.4 次のスケジューリング時間 $t_{current}$ (少なくとも一つのプロセッサがアイドル状態になる最も早い時刻) を算出し、レディタスクがある限り Step2.2 を繰り返す。

```
<< LU Decomposition >>
 1) u12 = a12 / l11
 2) u24 = a24 / l22
 3) u34 = a34 / l33
 4) l54 = -l52 * u24
 5) u45 = a45 / l44
 6) l55 = a55 - l54 * u45
<< Forward Substitution >>
 7) y1 = b1 / l11
 8) y2 = b2 / l22
 9) b5 = b5 - l52 * y2
10) y3 = b3 / l33
11) y4 = b4 / l44
12) b5 = b5 - l54 * y4
13) y5 = b5 / l55
<< Backward Substitution >>
14) x4 = y4 - u45 * y5
15) x3 = y3 - u34 * x4
16) x2 = y2 - u24 * x4
17) x1 = y1 - u12 * x2
```

図 3: 基本ブロックの例

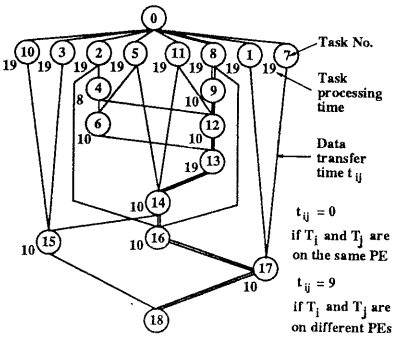

図 4: タスクグラフの例

このスケジューリング結果に従い、各タスクを OSCAR の命令に変換してオブジェクトコードを生成する。ただし、ここでスケジューリングに用いるタスク処理時間やデータ転送時間は OSCAR 上での実値であるが CP/DT/MISF では厳密にデータ転送のタイミングを定めることができないため、実際のコード実行タイミングはスケジューリングの結果からずれる。

このタイミングのずれのため、従来のインプリメントでは各タスクの先行制約を満たすために、タスク間でデータ同期 (条件同期) をとる必要があった。このデータ同期のため OSCAR では分散共有メモリ上にフラグ変数をおき、これを操作するフラグ転送コードを挿入して同期をとっている。フラグ送信側 PE はバスを介して 4 クロックで、受信側 PE の分散共有メモリ上のフラグをセットする。一方受信側 PE はフラグ状態を監視するためフラグ読込・チェック・条件分岐、の 3 命令の実行を繰り返し受信動作を行う。但し、この受信動作は PE 内のビジーウェイトで済むので集中共有型マルチプロセッサで問題となるビジーウェイトによるバスバンド幅の低下は起こらない。

以上のように OSCAR 上では、送受合わせて最低 7 クロックかけてデータ同期を実現している。

又、現在の OSCAR Fortran コンパイラでは、基本ブロックやループ・ボディの最後に全ての PE が作業を終了している事を確認するため、バリア同期をとるコードを挿入している。OSCAR ではこのバリア同期は専用ハードウェアを用いて最短 7 クロックで行なわれ、同期後、次のブロック、あるいは次のイタレーションの実行を開始する。

2.3 冗長な同期の除去

データ同期及びバリア同期に要する時間は本来の演算とは関係ないオーバーヘッドの要因となる。例えば OSCAR では、データ依存を保証するデータ同期に最低 7 クロックを要してしまい、通常の演算命令が 1 クロックで処理を行うのを考えると、大きなオーバーヘッドとなることがわかる。またフラグセットを行なう際にバスを使用することからバスの帯域を下げ、バスを介した通常のデータ転送を圧迫する。

又、バリア同期を取るには、最後の PE がバリア同期ポイントに到着してから 7 クロックを要し、実行時間が数 10 クロック程度の小さな基本ブロックやループ・ボディ部に適用すると相対的にかなり大きなオーバーヘッドとなる。

この同期のオーバーヘッドを軽減するために、筆者等はスケジューリング結果とタスクの先行制約がコンパイル時に決定していることを利用して冗長なデータ同期コード、すなわちフラグ

セット及びフラグチェックのコードを除去する方法を従来より提案している [4][5]。例えば、集中型共有メモリ上に同期フラグをおいて図5A の様に 3PE に割り当てられた 1~4 のタスクを並列処理する場合、通常は太線エッジで示される 5 回の同期を取る必要がある。しかし、同一 PE 上に割り当てられたタスク 2 と 3 の間の同期 Sync-3 は明らかに不要であり、フラグセット FS2 とフラグチェック FC2 は削除できる。次に同期 Sync-2 が処理される時は、必ず Sync-1 の処理及びタスク 2 の処理は完了しているため、改めて Sync-2 の同期をとる必要はない。同様にして Sync-4 は Sync-1 と Sync-5 が完了しているため削除することが可能となる。これらを踏まえて、フラグチェック FC1b,1c を除くことができる。このような冗長なフラグ除去を行なうと、図5B に示されるように Sync-1 と Sync-5 の 2 個だけに同期が減り大幅なオーバーヘッド軽減が達成できる。

2.4 全同期の除去

本論文で提案する手法の目的はタスク間データ依存を満たしながら全ての同期コードを除くことにある。例えば図5B のタスクの実行開始と終了の時刻を調べてタスク 1 が終了 (t^1_{end}) してからタスク 2 が開始 (t^2_{start}) されると判っている ($t^1_{end} \leq t^2_{start}$) なら、改めて同期を取る必要はない (図5C)。さらにすすめて、タスク 1 が終了してからタスク 2 が開始されるようにスケジューリングすることにより、先行制約を満たしながら同期コードを除くことが可能である。

しかし一般的なマルチプロセッサシステムでは命令パイプラインやキャッシュの状況によっても影響を受け、厳密に命令の実行時間を予想するのは非常に難しい、すなわちプログラムの動作を予め予測することが困難であるため、このような同期コードの除去は不可能であった。

しかし本手法では OSCAR のスタティックスケジューリングサポート機能を最大限に活かして、タスクの実行開始, 終了タイミング、バスアクセスタイミング、プロセッサ間データ転送開始, 終了タイミングを決定して全ての同期を除くことを可能とする。

バリア同期についても同様で、全ての PE が同時にバリア同期ポイントに到達するようにスケジューリングすることにより、バリア同期処理をとらずに済むことになる。

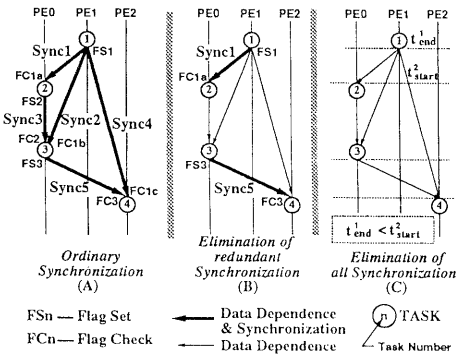

図 5: 冗長な同期の除去と全同期の除去

3 無同期細粒度並列処理コードの生成

本手法では CP/DT/MISF によるスタティックスケジューリング結果を用いた上で、データ転送タイミング、バスの競合による遅延などを考慮し、各タスク、各命令のタイミングをクロック単位で厳密に定めるコードスケジューリングを用い、無同期実行プログラムを作成する。

この無同期実行を行うために、クロックレベルのコードスケジューリングと、同期ポイントでのタイミング調整を行う操作を、基本ブロック [17] ごとに施す。また、基本ブロック間についても可能な限り無同期実行を適用する。

3.1 タスクの実行時刻の決定

クロックレベルのコードスケジューリングを行うために、基本ブロックの先頭から各タスクとそれを構成する命令をスキャンし、個々の命令の処理時間を累算しながら命令とタスクの実行時刻を定めていく。

まず基本ブロックの最初のタスクの実行開始時刻は 0 であり、当然その中の 1 番目の命令実行開始時刻も 0 である。そして次命令の実行開始時刻は前命令実行開始時刻に前命令の実行時間を加えたものとし、また、タスク内の最後の命令の終了時刻はそのタスクの終了時刻となる。

次に、述べる無同期並列コード生成手法を述べる。ただし、現状では、インプリメントの都合上一度同期コードを含んだマシンコードを出力させて、タイミングを調整し同期コードを除去する手順をとっているため、以下に示す手順では、挿入してある同期コードを除去する操作が含まれている。しかし、最終インプリメントでは同期は最初から挿入しない予定であるので、この操作は不要となる。その場合には手順中の同期コードは同期ポイントの意を表すこととなる。

処理する基本ブロックの番号を bn、PE 番号を pe、その PE に割り当てられた i_{pe} 番目の命令の実行開始時刻を $op_{time}(bn, pe, i_{pe})$、その命令実行にかかる時間を $op_{exec}(bn, pe, i_{pe})$ とする。また、処理にあたって、各 PE の現在の処理時刻を $t_{current}(pe)$ で表現すると、基本ブロック bn の同期コードを除く手順は以下のようになる。

Step.A-1 すべての PE について処理時刻を $t_{current}(pe) = 0$
処理命令を $i_{pe} = 0$ とし、現在処理の対象とする PE の番号を $pe = 0$ とする。

Step.A-2 ある命令 i_{pe} の実行開始時刻は、
$op_{time}(bn, pe, i_{pe}) = t_{current}(pe)$ とし、PE の現時刻を
$t_{current}(pe) = t_{current}(pe) + op_{exec}(bn, pe, i_{pe})$ で更新する。これを、バスアクセス命令, フラグ送信コード, フラグ受信コード, バリア同期, 基本ブロック bn の終端のいずれかに達するまで繰り返す ($i_{pe} = i_{pe} + 1$)。

Step.A-2.1 フラグ送信コードに達したら **Step.C-1** へ。

Step.A-2.2 フラグ受信コードに達し、かつフラグ番号 $flagid$ に対応する送信時刻 t_{flagid} が定義されていたら **Step.C-2** へ。

Step.A-3 Step.A-2 のループを停止したら次の PE ($pe = pe + 1$) について Step.A-2 の操作を繰り返す。全 PE について処理を終了したら、Step.A-4 へ。

Step.A-4 バスアクセスを行なう PE があれば **Step.B-1** へ。

Step.A-5 全 PE がバリア同期に達しているならばバリア同期除去のため Step.D-1 へすすむ。また全 PE がブロックの終端に達していたならこのブロックの処理は全て終了しており、終端の処理のため Step.E-1 を実行する。まだブロックの処理の途中のものがあれば、Step.A-2 より処理を繰り返す。

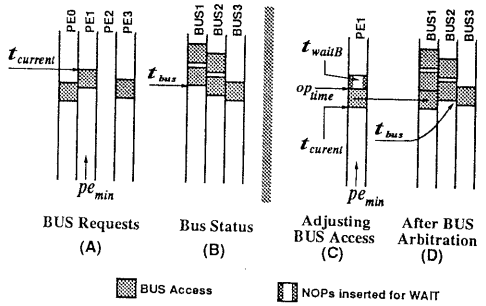

図 6: バスアクセスの調整

3.1.1 バスアクセス競合と調整

全 PE の状態を調べ、現在バスアクセスを待っているものを探し出し、その PE のバスアクセス命令の実行開始, 終了時刻を決定するとともに、バスの利用状況を更新する。Step.B-1〜3 に従って、順次バスに割り当ててタイミングを定める (図 6)。

Step.B-1 バスの利用状態を調べ、バスが最も早く空く時刻 t_{bus} (図 6B) を決定する。

Step.B-2 現在バスアクセスを待つ PE のうち、最も $t_{current}(pe)$ の小さい pe_{min} を選択する (図 6A, この場合 $pe_{min} = PE1$)。pe_{min} が複数あれば、その中で最大の PE 番号 pe_{min} を選択する。

Step.B-2.1 選択した pe_{min} の現処理時刻について $t_{current}(pe_{min}) < t_{bus}$ であれば、$t_{waitB} = t_{bus} - t_{current}(pe_{min})$ なる時間 t_{waitB} だけ WAIT コード (通常は NOP コード) を挿入して (図 6C)、現処理時刻を $t_{current}(pe_{min}) = t_{bus}$ と更新する。

Step.B-2.2 そして、バスアクセス命令の実行開始時刻を定めるため、$op_{time}(bn, pe_{min}, i_{pe}) = t_{current}(pe_{min})$ とし、pe の現時刻を $t_{current}(pe_{min}) = t_{current}(pe_{min}) + op_{exec}(bn, pe_{min}, i_{pe})$ と更新する (図 6C の op_{time} と $t_{current}$)。そして、次のバスアクセス命令実行開始時刻決定に備えてバスが空く時刻を調べて t_{bus} を決定する (図 6D)。

Step.B-3 バスアクセスのあるあいだ Step.B-2 を繰り返したあと、$pe = 0$ として Step.A-2 より繰り返す。

3.1.2 同期除去手順

フラグ送信、並びに受信コードの除去は、図 7に示す手順で行なう。フラグ転送コードを除去することにより、同期に要したオーバーヘッドがなくなり、処理時間が短縮される。具体的に除去するには、以下のとおり。

Step.C-1 まず、図 7a のタスク X の次のフラグ送信コード Sync-S について、フラグの番号 $flagid$ を調べて送信時刻 t_{flagid} を現処理時刻 $t_{flagid} = t_{current}(pe)$ と記録し (図 7b)、フラグ送信コードを除いてタスク A について Step.A-2 よりループを繰り返す。

Step.C-2 図 7a,c のタスク Y の次のフラグ受信コード Sync-R の場合、フラグの番号 $flagid$ を調べ、t_{flagid} と $t_{current}(pe)$ を比較する。

Step.C-2.1 $t_{flagid} \leq t_{current}(pe)$ であればそのまま受信コードを除去し (図 7b)、タスク B について Step.A-2 より処理を続行する。

Step.C-2.2 逆に、$t_{flagid} > t_{current}(pe)$ であれば t_{flagid} の時刻まで、待たなければならない (図 7d)。具体的には、$t_{waitf} = t_{flagid} - t_{current}(pe)$ の時間 t_{waitf} だけ、タスク Y と次のタスク B の間に WAIT コードを挿入する。そして、$t_{current}(pe) = t_{flagid}$ と更新して、受信コードを除去し、タスク B について Step.A-2 より処理を続行する。

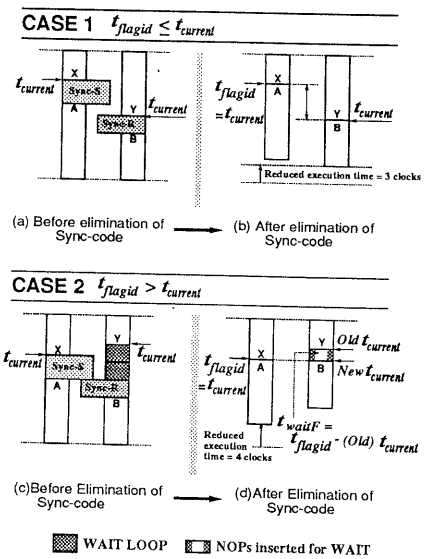

図 7: データ同期コードの除去

3.1.3 バリア同期の除去

Fortran プログラムの基本ブロックや Do ループイタレーションの途中にあるバリア同期 (図 8a) は、下記の手順で、タスク (X,Y,Z) の終了時刻を揃えて (図 8b の X',Y,Z') 取り除く。これにより、基本ブロックの実行時間を全体で約 7 クロック短縮する。

Step.D-1 全 PE の中で最も最後にバリア同期ポイントに到着する PE と、その時刻 t_{max} を調べるため、全 PE の $t_{current}(pe)$ のうち、最も時刻が遅い pe_{max} を選んで (図 8a, この場合はタスク Y の PE)、$t_{max} = t_{current}(pe_{max})$ とする。この t_{max} をバリア同期の時刻と定める。

Step.D-2 全ての PE が、時刻 t_{max} でバリア同期ポイントに到達するように、それぞれの PE につき現処理時刻 $t_{current}(pe)$ との差 $t_{waitR(pe)} = t_{max} - t_{current}(pe)$ を算出し、$t_{waitR}(pe)$ クロック分 WAIT コードを追加し、バリア同期箇所で全 PE を揃える (図 8b)。

Step.D-3 $pe = 0$ として、Step.A-2 より処理を再開する。

3.1.4 基本ブロックの終端処理

Step.A〜D の処理を行うことで基本ブロックから同期コードを除去できるが、そのままでは各 PE の最後のタスク (図 8c の X,Y,Z) の終了時刻が、まちまちとなる。連続した基本ブロックを無同期で実行する準備として、全 PE の基本ブロックの終了時刻を揃える (図 8d の X',Y,Z')。手順は前項のバリア同期除去と類似であるが、後続基本ブロックの頭で PE の実行タイミングを揃えるバリア同期を挿入する必要が無くなり、連続した基本ブロックを無同期で実行するために有効である。

Step.E-1 基本ブロック終了時刻 t_{max} を決定するため、全 PE の $t_{current}($ のうち、最も時刻が遅い PE を選んで (pe_{max}、この場合にはタスク Y の PE)、$t_{max} = t_{current}(pe_{max})$ とする (図 8c)。

Step.E-2 全 PE を基本ブロック終了時刻 t_{max} で実行を終了させるため、各 PE について $t_{waite}(pe) = t_{max} - t_{current}(pe)$ を算出し、$t_{waite}(pe)$ クロック分 WAIT コードを各 PE の基本ブロックの最後に追加する (図 8d)。

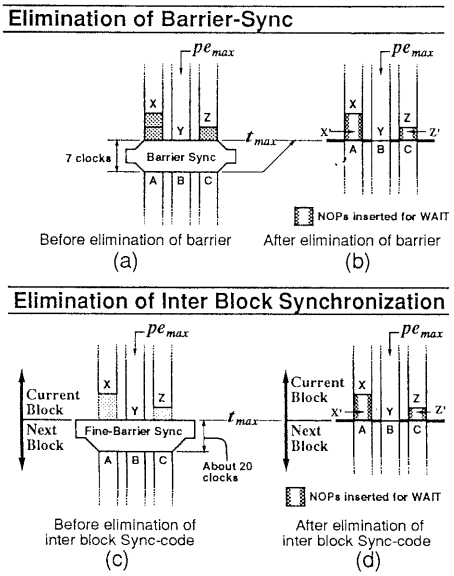

図 8: バリア同期除去 (上) とブロック終端の処理 (下)

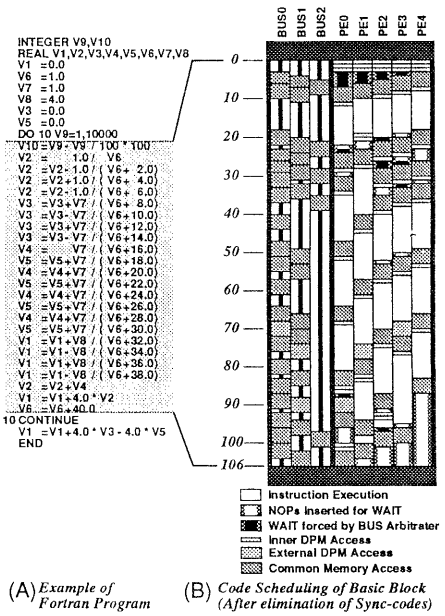

図 9: 例題 1(A) とループボディのコードスケジューリング (B)

以上の Step.A〜E 手順を用いて、実際にクロックレベルの厳密なコードスケジューリングを行なった例を図 9B に示す。この例では図 9A の Do ループのボディ部を PE5 台で並列処理するものとして、スケジューリングを行なっている。

3.2 連続したブロックの無同期実行

基本ブロックをスケジューリング通りに動作させるためには、プログラム中の全基本ブロックの先頭に実行のタイミングを合わせるバリア同期の挿入が必要となるが、この方式では同期オーバーヘッドが大きくなる。ここではこれを避ける手法について提案する。

Fortran の簡単な構文、例えば Do 文は図 10に示されるように 3 つの基本ブロック A,B,C と 2 つのラベル L1,L2、そして条件分岐 JC と (無条件) 分岐 JA に展開される。スケジューリングの結果、個々の基本ブロックは 1 クロックの誤差もなく同時に先頭から実行し、やはり同時に実行を終了すると仮定して、図について説明する。図のプログラムの動作を説明すると、プログラムの先頭でバリア同期をとり、基本ブロック A(初期設定) の実行を時刻 0 クロックで同時に開始して 10 クロック目で終了する。全 PE はラベル L1 を通過し、10 クロック目に同時にブロック B を開始する。又、基本ブロック B を 20 クロック目で全 PE が同時に終了した後、全 PE で揃って条件分岐 JC を 20 クロック目から 1 クロックをかけて処理し、分岐する場合には 21 クロック目に揃ってブロック D へ移る、また分岐しない場合は 21 クロック目に全 PE は同時にブロック C の実行を開始する。

この例で見る通り、連続する基本ブロックの間にラベル行があるとき、ラベル行は PE 間で実行のタイミングを変化させる事が無いので、直前の基本ブロックを同時に終了するなら、次の基本ブロックは全く同時に実行を開始すると考えてよい。又、分岐命令がある場合でも、直前のブロックを同時に終了するなら、全 PE が同じタイミングで分岐命令を開始し、同じタイミングで分岐処理をして、次の基本ブロックを全 PE が同じタイミングで開始することになり、PE 間のタイミングに影響を与えない。

このように、Do ループ中の基本ブロックの開始と終了のタイミングを全 PE で揃えておけば、基本ブロックの接続点がラベル行, 分岐命令のいずれの場合でも全 PE の実行タイミングは狂わないので、次の基本ブロックの先頭で改めて同期を取り直す必要が無い。そして、基本ブロックの中の同期コードも除去しているので、Do ループ全体を全く同期無しで実行することが可能である。

より複雑な構文でも、基本ブロックがラベル行と分岐命令で接続された形を取る限りにおいて状況は同じであり、ほとんどの同期コードを取り除いて、プログラムを無同期で実行することが可能である。

4 無同期実行の性能評価

本章では、提案する無同期細粒度並列処理手法を OSCAR 上でインプリメントし、性能評価を行なった結果について述べる。

4.1 スケジューリングと実行結果の対応

まず、図 9に示した基本ブロックを用いて PE 間の実行タイミングがあってるかを調べ、OSCAR の実際の動作がコンパイラにより生成されるタイミングチャートと同じであることを確認する。具体的には図 9の基本ブロックを 10000 回繰り返し実行し、実際の実行時間を測定し、ブロックのスケジューリングから算出された予想実行時間と比較した。結果は表 1 (表中の値は 1 ループあたりの実行時間) の通りで、表に示すように実測値と推定値の誤差が約 $120ns$ 含まれているが、これは 1 マシンクロック $333ns$ より小さい値であり、実行タイミングのずれではなく計測ルーチンの誤差であることがわかる。

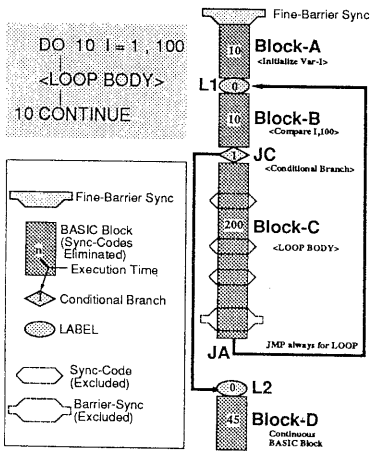

図 10: 連続した基本ブロックの処理

表 1: スケジューリングと実行結果の対応 (1 ループ)

PE 台数	スケジューリング クロック数	推定時間 [μs]	実測時間 [μs]
2	235	78.255	78.350
3	185	61.605	61.764
4	158	52.614	53.782
5	113	37.629	37.850
6	110	36.630	36.852

1 クロック=333ns

これよりスタティックコードスケジューリングの結果と、OSCAR での実行の結果は同一であることが確認され、プログラムの動作をクロック単位で最適化し、実際にそのとおりに動作させることが可能であることが確かめられた。

4.2 評価

実際に OSCAR 上で Fortran プログラムを処理し、実行した結果を以下に示す。図 11及び 12のグラフ中に示される 3 種類の線のうち、細実線は細粒度並列処理を行なった基本ブロック中の同期を全く除去しない場合、点線は冗長なフラグ転送を除去 [5] した場合、そして太実線は提案する方法でプログラムを無同期実行する場合の結果をそれぞれ示す。また、図 11のグラフ中の数字は基本ブロック中でとられる同期の回数を示している。

例題 1 は円周率を級数計算で求め

$$\pi = 4 \cdot tan^{-1}1 \simeq \sum_{n=1}^{200000} \left\{ \frac{1}{2n-1}(-1)^{n-1} \right\}$$

を 20 項ずつシーケンシャルループに展開し 10000 回のループを実行する。本手法を用いて基本ブロック中の同期及び基本ブロック間の同期も省いて実行した実行時間を図 11の太線に示す。この場合はループに入る前に 1 回同期を取った後、ループを繰り返している間はバリア同期はもちろん、一切の同期を取らずに実行している。無同期実行により先行制約を保証できなければ、データ転送が正しく行なわれず演算結果に重大な過ちが引き起こされるが、同期を除去しない場合, 冗長な同期を除去した場合, 本手法で全ての同期を除いた場合の、いずれでもプログラムの演算結果は

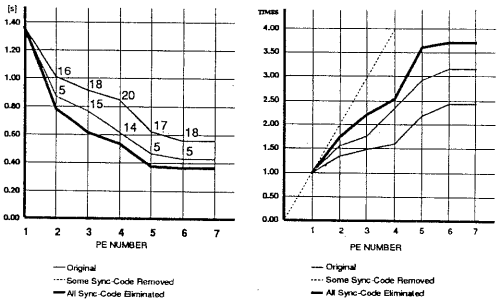

図 11: 例題 1 の実行時間と台数効果

表 2: 例題 2 の実行時間 [s]

PE 台数	同期除去前		冗長なデータ同期の除去後		本手法による無同期実行	
1	6.4842		6.4842		6.4842	
2	4.1634	(26)	3.9741	(16)	3.7684	(0)
3	2.9649	(25)	2.8702	(19)	2.7358	(0)
4	2.3457	(27)	2.2959	(22)	2.2345	(0)
5	2.1813	(33)	2.1216	(28)	2.0884	(0)
6	1.9765	(33)	1.9539	(28)	1.8560	(0)

カッコ内は基本ブロック中のデータ同期の数

全く等しかった。よって本手法で同期を全て除いても、タスク間の先行制約を正しく満たした実行が行なわれていると確認する。

次に、PE3 台を用いて細粒度並列処理を施した実行時間を見ると、図 11 に示すように全く同期を除去しない場合は、イタレーション当たりデータ同期 18 個とバリア同期を含めて実行時間が 0.9263s だったのに対し、本手法を用いて無同期で実行した場合は 0.6175s に短縮されており、同期コードを除くことにより実時間で 0.3088s の短縮、並列処理性能は約 33.3%の向上が達成されたことがわかる。なお、冗長なデータ同期のみを除去した場合は実行時間が 0.7702s であり、全く同期を除去しない場合に対して実時間で 0.1527s の短縮、並列処理性能は 19.8%の向上である。以上より本手法を用いることによって実行時間を大きく短縮できることが確かめられた。

又、並列処理性能で見ると、無同期並列処理手法により全く同期を除去しない場合に比べて約 30%、冗長なデータ同期を除いた場合に比べて約 10%程度向上している。

例題 2 は 60 × 60 非零要素 2.8%のスパース行列演算をループフリーコードで実行するもので、行列演算部分の基本ブロックは 96 ステートメントからなる。この基本ブロックの実行を 10000 回繰り返して実行時間を測定した。

このプログラムを細粒度並列化して、同期を除去しなかった場合, 冗長なデータ同期を除去した場合, 本手法を用いて無同期実行した場合の実行時間を表 2に示す。

この例でも同期を除去しなかった場合, 冗長なフラグ転送のみを除去した場合, そして本手法を用いて無同期実行した場合のいずれも、演算結果は全て等しく、タスク間の先行制約を保証しながら無同期実行が行なわれていることが確かめられた。

演算時間については、PE2 台を使用した場合に、最も削減効果が出ており、同期を除去しなかった場合に比べて、本手法を用いて 26 個のデータ同期とバリア同期を除いた結果、実時間で 0.3950s 実行時間を短縮し、並列処理性能は 9.5%向上している。この例題 2 では例題 1 に比べて約 4 倍のステートメントを基本ブロック中に含むが、除去したデータ同期の数としては 2 倍以下である。そ

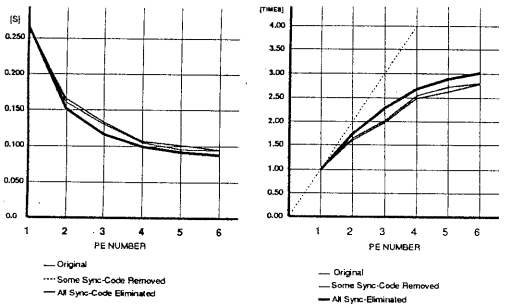

図 12: 例題 3 の実行時間と台数効果

のため、例題 1,PE2 台で 16 個のデータ同期を除去したことによる並列処理性能向上が 22.3%であるのに対し、一見劣っている。しかし実時間での短縮では例題 1 の 0.2258s 短縮に対して例題 2 では 0.3950s 短縮であり、除去したデータ同期の数に相応の実行時間短縮がなされていることがわかる。

例題 3 は市販のシステム解析用ツールプログラム"MATRIXx"の生成した Fortran ソースプログラムを処理した例である。このプログラムではシステムの出力値を 500$Step$ にわたって計算している。

例題 1 や例題 2 に比べて複雑なプログラム構造をとっているが、無同期実行で実行して、正しい計算結果を出しており、複雑なプログラムでも、基本ブロック間で同期を取らずに正しく実行していることが確認された。

実行時間は図 12に示す通りで、例えば、PE3 台を使用して、同期を除かない場合に比べて本手法で無同期実行した場合は実時間で 0.0213s 短縮し、並列処理効果は 16.6%向上している。

実行時間を図 12で見ると、冗長なデータ同期を取り除いた場合を示す点線が、同期を全く除去していない場合の細実線とほぼ同じカーブを描き、あまり効果を発揮していないのに対し、本手法を用いた無同期実行の実行時間を示す太実線は大きく他の 2 線を下回っており、無同期実行による顕著な実行時間短縮の効果が確かめられた。

以上より、クロックレベルのスタティックコードスケジューリングを用いて、基本ブロック中から同期コードを削除したプログラムが、タスクの先行制約を保証しながら正常に動作することが確認された。そして多少複雑なプログラム構造でも、基本ブロックの出入り口で全 PE の実行タイミングが揃っている限り改めて同期を取る必要がない事も確認された。これによりプログラムの先頭で 1 回同期を取るだけで後は全く同期を取らずに実行することも今後可能になると考えられる。

5 まとめ

本稿ではクロックレベルでの高精度なスタティックスケジューリング手法を用いて、マルチプロセッサシステム OSCAR 上で、基本ブロック中の全ての同期コードを除去するコンパイル手法について提案した。

又、実際に稼動するマルチプロセッサシステム OSCAR を用いてこれらの手法をインプリメントし、スタティックスケジューリングの精度を上げ、バスアクセスを含めた命令実行をクロックレベルで最適化する高精度のコードスケジューリングのアルゴリズムと、それをサポートするマルチプロセッサアーキテクチャの利用により、基本ブロックの細粒度並列処理を無同期で実行でき、実行時間を顕著に短縮できることが確認された。今後の研究課題としては、バスアクセスの順序とタイミングを最適化し、さらに細粒度並列処理を高速化することが挙げられる。

参考文献

[1] K.Hwang,F.A.Brigges:"Computer Arghitecture And Parallel Processing", McGRAWHILL(1984)

[2] 村岡洋一:"並列処理", 昭晃堂 (1986)

[3] 富田眞治, 末吉敏則:"並列処理マシン", オーム社 (1989)

[4] 笠原博徳:"並列処理技術", コロナ社 (1991)

[5] 笠原博徳, 藤井稔久, 本多弘樹, 成田誠之助: "スタティック・マルチプロセッサ・スケジューリング・アルゴリズムを用いた常微分方程式求解の並列処理", 情処論文誌,pp.1060-1070(Oct.1987)

[6] 笠原博徳:"最適化並列コンパイラ技術の現状", 信学誌 Vol.73 No.3 pp.258–266(Mar.1990)

[7] 本多弘樹, 岡本雅巳, 合田憲人, 笠原博徳: "Fortran プログラム粗粒度タスクの OSCAR における並列実行方式", 信学論 VOL.J-75-D-I NO.8,pp.526–535(AUG.1992)

[8] 笠原博徳, 成田誠之助, 橋本親:"OSCAR のアーキテクチャ", 信学論 D,VOL.J71-D,8,pp.1440-1445(AUG.1988)

[9] H.Kasahara,H.Honda,S.Narita,"Parallel Processing of near fine grain tasks using static scheduling on OSCAR", Proc.IEEE SuperComputing'90,pp,856–864(Nov. 1990)

[10] 笠原博徳, 成田誠之助: "マルチプロセッサ・スケジューリング問題に対する実用的な最適解及び近似アルゴリズム", 信学論 Vol.J67-D No.7,pp.792–799(July 1984)

[11] M.T.O'Keefe,H.G.Dietz: "Hardware Barrier Synchronization Static Barier MIMD(SBM)", International Conference on Parallel Processing,Vol.I,pp.35–42(1990)

[12] M.T.O'Keefe,H.G.Dietz: "Hardware Barrier Synchronization Dynamic Barrier MIMD(DBM)" International Conference on Parallel Processing,Vol.I,pp.43–46(1990)

[13] Y.Birk,P.B.Gibbons,J.L.C.Sanz,D.Sporoker:"A simple mechanism for efficient barrier synchronization in MIMD machines", International Conference on Parallel Processing,Vol.II,pp.195–198(1990)

[14] C.J.Beckmann,C.D.Polychronopoulos: "Fast Barrier Synchronization Hardware", Proc.Super computing 90,IEEE, pp.180–189(1990).

[15] A.Zaafrani,H.G.Dietz,M.T.O'Keefe: "Static Scheduling for Barrier MIMD Arghitectures", International Conference on Parallel Processing,Vol.II,pp.187–194(1990)

[16] 本多弘樹, 水野聡, 笠原博徳, 成田誠之助: "OSCAR 上での Fortran プログラム基本ブロックの並列手法", 信学誌 Vol.73-D-I pp.756–766(Sep.1990)

[17] A.V.Aho,R.Sethi,J.D.Ullman,"Compilers:Principles,Techniques ,and Tools",Addison-Wesley(1986)

[18] Padua D.A.AndWolfe M.J.:"Advanced Compiler Optimization for Super Computers", C.ACM,29.12,pp.1184–1201(Dec.1986)

[19] 笠原博徳, 合田憲人, 吉田明正, 本多弘樹: "Fortran マクロデータフロー処理のマクロタスク生成手法", 信学論 Vol.J75-D-I No.8,pp.511–525(Aug.1992)