スタティックスケジューリングを用いたマルチプロセッサシステム上での無同期近細粒度並列処理

尾形 航[†] 吉田明正[†] 合田憲人[†]
岡本雅巳[†] 笠原博徳[†]

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近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込まねばならず，実行時の同期オーバヘッドが比較的大きいという問題があった．本論文ではマシンコード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命令実行タイミングを最適化し，すべての同期コードを除去することで同期オーバヘッドを低減する手法について提案する．また本手法を，ハードウェアアーキテクチャ面からサポートするよう設計された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．

Near Fine Grain Parallel Processing without Synchronization using Static Scheduling

Wataru Ogata,[†] Akimasa Yoshida,[†] Kento Aida,[†]
Masami Okamoto[†] and Hironori Kasahara[†]

A 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 codes by optimizing, or scheduling, execution timing of every instruction in machine clock level. Furthermore, it reports performance of the parallel processing without synchronization codes evaluated on an actual multiprocessor system OSCAR, which was designed to support the proposed scheme.

1. はじめに

科学技術計算の並列処理[1),2)]において，Fortran 自動並列化[10),11),18)]が従来のソフトウェア資産を活かしながら性能向上を図るために要求されている．

従来のマルチプロセッサ用自動並列化コンパイラでは，中粒度並列処理，すなわちループイタレーション間の並列性を利用する Do-All, Do-Across などのループ並列化手法[18)]が採用されている．このループ並列化技術は強力なデータ依存解析技術[18)]とプログラムリストラクチャリング技術[3)]の進歩と共に高度化しており，最近ではさまざまなループが自動的に並列化できるようになってきた．しかしイタレーション間に複雑なループキャリッドディペンデンスが存在するループや，ループ外部への条件分岐を持つループなど，効率良く並列化できないループが依然存在する．また，ループ外部の基本ブロックは並列化の対象とならない，基本ブロック，DO ループ，サブルーチン間の粗粒度並列性を利用できない，という問題があった．

この問題を解決し，さらにはプログラムの持つ並列性を最大限抽出してマルチプロセッサシステムの実効性能を向上させるための手法として従来のループ並列化に加え，粗粒度並列処理[2),11)]，近細粒度並列処理[16),14)]を階層的に組み合わせて多様な並列性を多くのプロセッサ上で効率良く利用することを可能とするマルチグレイン並列処理手法が提案されている[20)]．

ここで，粗粒度並列処理とは，ループ，基本ブロック，サブルーチン等をマクロタスク[2),19)]として定義し，それらの間の並列性を最早実行可能条件解析によ

† 早稲田大学理工学部
School of Science and Engineering, Waseda University

り抽出後，タスクを条件分岐等の実行時不確定性を考慮し実行時にプロセッサクラスタに割り当てる手法でありマクロデータフロー処理とも呼ばれる．

また，近細粒度並列処理手法は，マルチグレイン並列処理においてプロセッサクラスタに割り当てられたマクロタスクが中粒度並列処理を適用できない単一の基本ブロック，あるいはシーケンシャルループであったときにこれらをプロセッサクラスタ内で並列処理する手法である．これは，プロセッサ内でトレーススケジューリング，パーコレーションスケジューリング等を用いて複数基本ブロックにわたる命令レベルの並列性を抽出する VLIW やスーパースカラなどの命令レベル細粒度並列処理手法とは異なり，単一基本ブロック（マクロデータフロー処理によりマクロタスクとしてプロセッサクラスタに割り当てられる）内のステートメント（すなわち複数命令の集合）を近細粒度タスクとして定義し，これらの間の並列性を複数の PE 上で利用しようとする方式であり[14]，このマルチプロセッサ上での近細粒度並列処理の実現のためには，自動並列化コンパイラによる最適化だけではなく，異なる PE に割り当てられた近細粒度タスク間通信を効率良く行えるマシンの開発が不可欠である．さらに，近細粒度タスク間の同期オーバ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[4)~7),9)]の開発や，同期処理の回数を減らすためのスケジューリング結果を利用した冗長な同期の除去手法[12]，あるいは基本ブロックを複数のセクションに分割しセクションごとにバリア同期を取ることによりセクション内の同期コードを除去する手法[8)]などの開発が重要となる．このような近細粒度並列処理においては，基本ブロック中あるいはブロック間のすべての同期コードを除去し，すべてのデータ依存を満足させながら最小の同期オーバヘッドで実行することが1つの理想的な処理形態である．本論文ではマルチプロセッサ上でこのような理想的な並列処理を実現するためのコンパイル手法を提案する．

さらに本論文では，実マルチプロセッサシステム OSCAR (Optimally SCheduled Advanced multiprocessoR)[13]上で，提案する無同期並列処理手法の有効性を検証した結果についても述べる．

2. ハードウェアサポート

マルチプロセッサシステム上でステートメントレベルの近細粒度並列処理を実現するためには，タスク間のデータ転送，同期オーバヘッドを最小化するための高機能コンパイラとそのコンパイラの性能を引き出すアーキテクチャサポートが必須である．

さらに本論文で提案する無同期近細粒度並列処理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厳密なコードスケジューリングと高度なアーキテクチャサポートが必要となる．この無同期並列処理を可能とするマルチプロセッサシステムでは，コンパイル時に近細粒度タスクの実行に要するコストをクロック単位で予測することを可能とし，さらにバスアクセス等のマシン各部の挙動もコンパイル時にすべて予測できねばならない．提案する並列処理手法の有効性評価に使用したマルチプロセッサシステム OSCAR は，上記の条件を満足し，厳密なコードスケジューリングをサポートするように設計されたマシンである．

2.1 OSCAR のアーキテクチャ

マルチプロセッサシステム OSCAR は**図1**に示すアーキテクチャを持ち，32 bit-RISC 型プロセッサ，分散共有メモリ（DSM，デュアルポートメモリ），ローカルメモリ（LM，プログラムメモリおよびデータメモリ），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3バンクの集中型共有メモリ（CM）に接続した構成となっている．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．

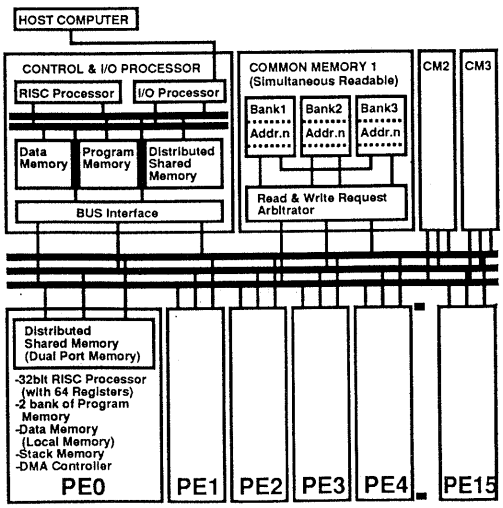


図 1 OSCAR のアーキテクチャ
Fig. 1 Architecture of OSCAR.

2.1.1 プロセッサエレメント

OSCAR の各プロセッサは，単精度浮動小数点の加減乗算も含めたすべての命令を 1 クロック（200 ns ただし現在は 333 ns に設定）で処理するほか，他の浮動小数点演算も固定クロック数で実行する．また OSCAR では，DRAM を使用した場合のリフレッシュによる命令の実行タイミングの変化を避けるため，PM，LM，CM には SRAM を使用している．

また，OSCAR は，

(1)DSM 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，各 PE は 3 本のプロセッサ間バスを介して他の PE 上の DSM または CM にデータを直接リード/ライトできる．

この際，他 PE 上の DSM へのデータ転送時間や，CM アクセスに要する時間，バスアクセス競合が生じたときの調停と遅延のクロック数も，コンパイル時に解析できるようになっている．これは，スタティックスケジューリングの精度を高めると共に，プログラムの実行をクロック単位で制御するコードスケジューリング機能を支援する設計である．

2.1.2 データ転送とハードウェアバス調停

OSCAR では，前述のような 3 種類のデータ転送モードを用いて，CM または他 PE 上の DSM への 1 ワードのデータのリード/ライトに 4 クロックを要する．この 4 クロックの内訳は，図 2 A の中ほどに示すように，

(1)バスに対するアクセス要求と調停
(2)使用バスの選択と獲得
(3)アドレス送出
(4)データ転送

となっている．

また，ここで 1 クロック目のバスアクセス調停時に使用される優先順位は，各 PE が 3 本のバスに対し個別に設定可能である．例えば PE 0 は，バス 1 に対して最高の優先順位，バス 2 に対して中程度の優先順位，バス 3 に対して最低の優先順位，のように異なるバスアクセス優先順位をディップスイッチにより設定することができる．現在はバスアービトレータの動作をコンパイラが簡単に解析できるようにするために，3 本のバス共に，番号の大きい PE から順に高い優先順位を割り付けている．

次に，バスアービトレータの動作について例を用いて具体的に説明する．まず，図 2 A-1 の例に示されるように複数のデータ転送命令（CM アクセスや DSM アクセス）が同時に発行されたときの OSCAR のバスの挙動は以下のとおりである．

1 クロック目で 3 台の PE がバスアクセス要求を同時に出し，その調停の結果，最も高い優先順位を持つ PE 3 が 2 クロック目に BUS 1 を獲得すると同時に BUS 2, 3 を解放する．その間，PE 2 と PE 1 はアクセス待ちとなる．3 クロック目に BUS 1 上で PE 3 が DSM のアドレスを送出する一方，次に優先順位の高い PE 2 は BUS 2 を獲得し，最も優先順位の低い PE 1 のアクセスはさらにもう 1 クロック待たされる．4 クロック目で PE 3 は BUS 1 上でデータ転送を行い，PE 2 は BUS 2 上で CM アドレスの送出，PE 3 は BUS 3 を獲得する．なお，OSCAR の CM は 3 本

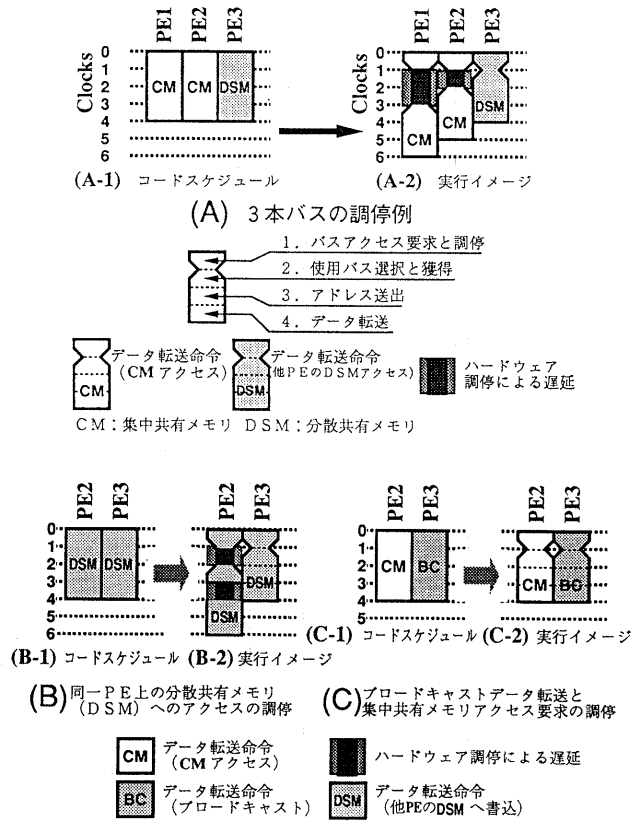


図 2 ハードウェアアービトレータの動作
Fig. 2 Example of hardware bus arbitration.

のバスに対応して3バンク構成となっており，各バスと CM バンクは互いに独立してアドレス送出とデータ転送を行うことができる．以下，5クロック目に PE 2 は BUS 2 を介して CM との間でデータ転送を行い，PE 3 は BUS 3 上でアドレスの送出，6クロック目に PE 3 はデータ転送を行う．この調停の様子を図2 A-2 に示す．図からわかるように，3PE のうち最も高い優先順位を持つ PE 3 はデータ転送命令発行から4クロック後に転送を終了するが，次に優先順位の高い PE 2 は1クロックの遅延を生じて命令発行から5クロック目に終了，3番目に優先順位の高い PE-1は2クロックの遅延を生じ6クロック目にデータ転送を終了する．

次に，複数の PE が同時にある特定の PE の DSM にアクセスする場合，例えば図2 B-1 のように PE 2 と PE 3 が PE 1 の DSM を同時にアクセスする場合，ハードウェアバスアービトレータの調停により，PE 2 は3クロック目で BUS 2 を獲得する．しかし，各 PE 上の DSM は，アドレス受理とデータ転送を同時に行えないので，PE 2 は4クロック目にもアクセス待ちとなり，5クロック目で DSM アドレス送出，6クロック目でデータ転送を行う．結局，PE 2 は計2クロックの遅延を生じ，図2 B-2 に示すとおり，同時に発行されたデータ転送命令のうち，PE 3 は4クロック目にデータ転送を終了するが PE 2 は6クロック目に終了する．

また，図2 C-1 のようにブロードキャスト転送と他のデータ転送（CM アクセス）が同時に発行された場合は，現在の OSCAR ではハードウェアバグを避けるために BUS 1 を用いたブロードキャスト転送しか許していないため，2クロック目に PE 3 が BUS 1 を獲得すると同時に PE 2 も BUS 2 を獲得することができ，バスアービトレータによる遅延が生じず，図2 C-2 のように両方とも4クロック後に命令を終了する．

3. 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 並列化コンパイラでは粗粒度，中粒度，近細粒度のすべての粒度で Fortran プロ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3**に示される論理的構造をとり，Phase1 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，近細粒度の各粒度で並列化を行う．ここで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，また近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うことを意味する．

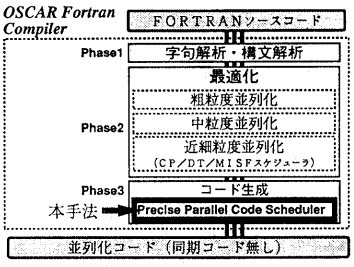


図 3 OSCAR Fortran コンパイラの構成
Fig. 3 A structure of OSCAR Fortran compiler.

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，クロックレベルで近細粒度タスクの実行タイミングを制御して，同期コードを挿入せずに OSCAR 用のコードを生成する．このコードは OSCAR 上でシングルジョブ形態で，割り込みがない状態で実行される．

以下に提案する無同期近細粒度並列処理の手法について説明する．

3.1 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，まず，ステートメントをタスクとして，タスク間のデータ依存解析を行い，**図4**のようなタスクグラフを生成する．

次に，この先行制約を満たしながら，リストスケジューリングの一種である CP/DT/MISF[2),14),15)]を用いてタスクを静的に PE に割り当てる．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．

Step 1 cp 長に従い各タスクのレベル l_i を決定する
Step 2 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

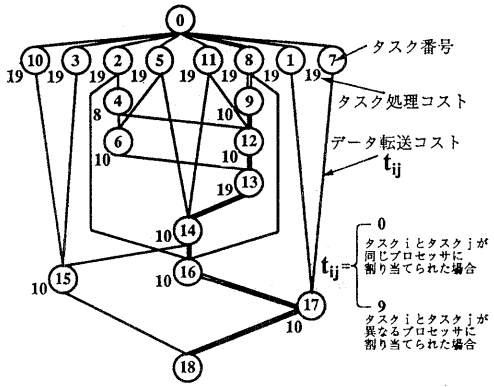


図 4 タスクグラフの例
Fig. 4 An example of task-graph.

Step 2.1 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

Step 2.2 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．

Step 2.3 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 Step 2.3 を繰り返し，なければ Step 2.4 を実行する．

Step 2.4 次のスケジューリング時間 $t_{current}$（少なくとも1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 Step 2.2 を繰り返す．

次に，各 PE に割り当てられたタスクを OSCAR の機械語命令に変換して各プロセッサ用のオブジェクトコードを生成する．ただし，この段階ではクロックレベルの厳密なスケジューリングを行っていないので，各タスク間のデータ依存を満たすために，タスク間でデータ同期（条件同期）をとる必要がある．

3.2 同期オーバヘッド

OSCAR 上で近細粒度並列処理を行う場合，タスク間でデータ同期をとるために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並列マシンコード中に挿入することにより同期を実現している．実行時にフラグ送信側 PE はバスを介し，4クロックをかけて受信側 PE の分散共有メモリ上に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．ただし，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メモリ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．

しかし，通常の演算命令が1クロックで実行されるのに比べると，送受合わせて最低7クロックを要するデータ同期は大きなオーバヘッドとなるうえ，フラグセットを行う際にバスを使用し，バスを介した通常のデータ転送を圧迫する．

また，現在の OSCAR Fortran コンパイラでは，基本ブロックやループ・ボディの最後にすべての PE が作業を終了していることを確認するため，バリア同期コードを挿入している．OSCAR では，バリア同期は専用ハードウェアを用いて最短7クロックで行われるが，しかし，これも実行時間が数 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大きなオーバヘッドとなる．

3.3 冗長な同期の除去

近細粒度並列処理における同期オーバヘッドを軽減するために，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されていることを利用して，冗長なデータ同期を除去する方法が従来より提案されている[2),12)]．

例えば，集中型共有メモリ上に同期フラグをおいて，**図5** Aのように 3PE に割り当てた 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が，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 は不要であり，また同期 Sync-2 を受信するときは，必ず Sync-1 は終了しているため，改めて Sync-2 の同期を

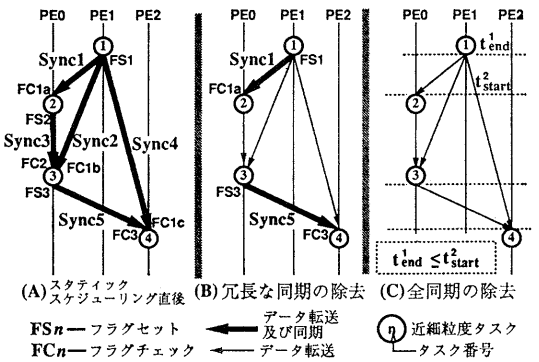


図 5 冗長な同期の除去と全同期の除去
Fig. 5 Elimination of redundant synchronization code and all synchronization code.

とる必要はない．同様に Sync-4 も削除できるので，最終的に図 5 B に示すように同期は 2 個（Sync-1 と Sync-5）に減りオーバヘッド軽減が達成できる．

3.4 全同期コードの除去

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すべての同期コードを除去することにある．例えば図 5 B の各タスクの実行開始・終了の時刻を調べてタスク 1 の終了時刻（t^1_{end}）がタスク 2 の開始時刻（t^2_{start}）以下（$t^1_{end} \leq t^2_{start}$）であることがわかっているなら，改めて同期を取る必要はない（図 5 C）．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コードスケジューリングすれば，先行制約を満たしながら同期コードを削除できる．

本手法で用いる OSCAR は，クロックレベルのコードスケジューリングをサポートするように設計されているためコンパイル時にマシンコード実行の様子を正確に予測できる．この特徴を最大限に活かして，タスクの実行開始・実行終了タイミング，バスアクセスタイミング，プロセッサ間データ転送開始・転送終了タイミングさらにはバリア同期タイミングなどを，コードスケジューリング時に決定してすべての同期を除去することを可能とする．

4. 無同期近細粒度並列処理コードの生成

コード生成ルーチンは 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，データ転送タイミング，バスの競合による遅延などを推定し，各タスク，各命令の実行タイミングをクロック単位で厳密にスケジュールし，無同期並列実行マシン語プログラムを生成する．

4.1 タスク・命令の実行時刻決定

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，おのおのの命令およびタスクの実行開始時刻と終了時刻を決定していく．

また，複数 PE によるバスアクセス命令の競合が生じる場合，同期フラグの送信や受信ポイント，バリア同期ポイントについては，それぞれ下記のように処理を行う．

4.2 コンパイラによるバスアクセス調停

バスアクセスについては，コード生成ルーチンがハードウェア，バスアービトレータの動作を考慮して，以下の手順で制御を行う．

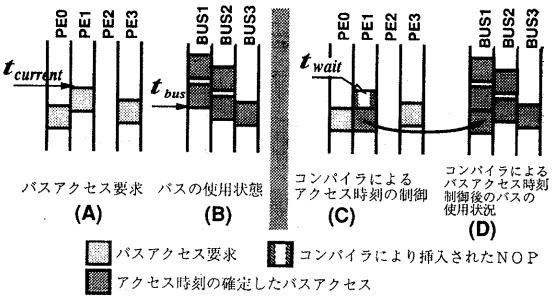


図 6 コンパイラによるバスアクセスの調停
Fig. 6 Arbitration of bus-access by compiler.

図 6 A，B は，それぞれある時刻 $t_{current}$ における PE からのバスアクセス要求（データ転送命令）タイミング（A）と，そのときのバスの利用状況（B）の例を示す．この状態で，まずコードスケジューラは，最も早くバスアクセス要求の出ている PE 1 のバスアクセスをスケジュールすることを考える．この図の場合，時刻 $t_{current}$ ではバスが 3 本とも塞がっているので，図 6 B に示すようにバスが空く時刻（t_{bus}）を計算し，図 6 C の t_{wait} の時間分だけ WAIT コード，すなわち NOP を挿入して，t_{bus} までアクセスを待たせてから PE 1 にバスアクセスを行わせる（図 6 D）．

また，図 2 で示したように複数の PE のバスアクセス命令が同時に発行された場合には，ハードウェアアービトレータがバスアクセスをウェイトさせるのでコードスケジューラはハードウェアアービトレータの調停結果と同様になるように WAIT コードを挿入し，バスアクセス競合が生じないように制御する．

以上のようにコードスケジューラはハードウェアの動作を厳密に推定して，複数の PE に割り当てられたデータ転送命令を見渡しながらコードスケジューリングを行う．

4.3 同期除去手順

図 7 では 2 台の PE に 4 つのタスク X，Y，A，B が割り当てられた場合を例として，データ同期コードを挿入する場合と，データ同期コードを除去する場合の比較を示す．図では PE 1 にタスク X と A，PE 2 にタスク Y と B が割り当てられ，タスク X とタスク B の間にデータ依存があるが本手法では以下の操作で同期コードを挿入せずに機械語命令を生成する．

まず各 PE に割り当てられたタスクをスキャンし，PE 1 でタスク X の次のデータ同期ポイント（データ送信，WriteDSM）に到達した場合には，従来は図 7 A-1，B-1 に示すように同期コード Sync-S（同期フラ

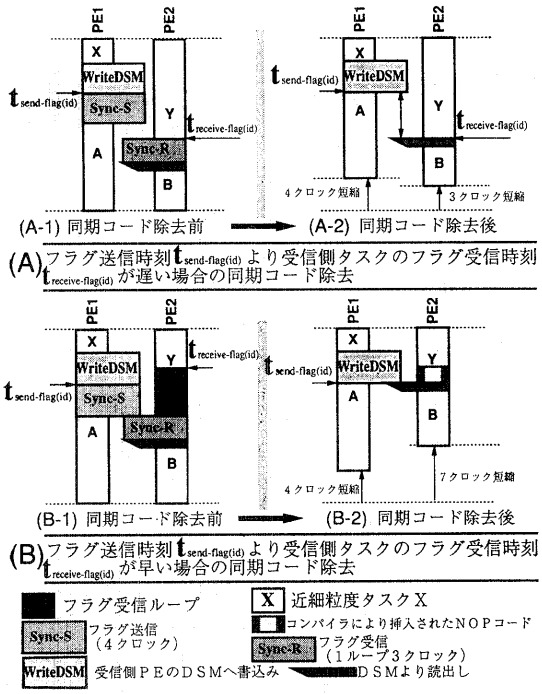


図 7 データ同期コードの除去
Fig. 7 Elimination of data synchronization codes.

グセット）を挿入したが，本手法では，そのフラグの番号を $flag(id)$ とし，また，そのフラグ送信時刻を $t_{send-flag(id)}$ として記録する．

一方，PE 2 でタスクB実行開始前のデータ同期ポイントに到達した場合，従来は図7 A-1，B-1 に示される同期コード Sync-R（フラグチェック）を挿入していたが，本手法では，直前のタスクYの終了時刻をフラグの受信開始時刻 $t_{receive-flag(id)}$ とし，チェックすべきフラグの送信時刻 $t_{send-flag(id)}$ と比較して図7 A-1 のように，

$$t_{send-flag(id)} \leq t_{receive-flag(id)}$$

であれば図7 A-2 に示すとおりタスクXからタスクBに送られたデータの DSM からのロードを $t_{receive-flag(id)}$ に開始させるようスケジューリングする．

逆に，図7 B-1 に示すように

$$t_{send-flag(id)} > t_{receive-flag(id)}$$

であれば図7 B-2 のようにタスクYと次のタスクBの間に WAIT コードを挿入し，$t_{send-flag(id)}$ の時刻まで，タスクBの開始を遅らせる．

これにより従来フラグ転送コードを挿入していた場合に比べて送受信合わせて7クロックのオーバヘッドが除去される．

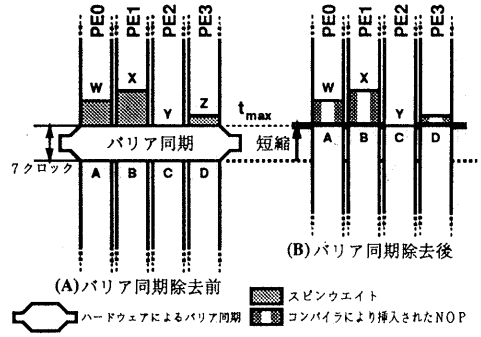


図 8 バリア同期除去
Fig. 8 Elimination of barrier-synchronization code.

4.4 バリア同期の除去

図8 Aは基本ブロックの途中にバリア同期がある例を示す．本手法では以下の手順でこれを除去する．まず，図8 Aのバリア同期の直前にあるタスク（W, X, Y, Z）の中で最後にバリア同期ポイントに到着する PE（この場合はタスクYを実行する PE 3）がそのタスクを終了する時刻 t_{max} を調べる．次に他の PE のタスク W, X, Z の後に WAIT コード（NOP）を追加して，全 PE が図 8 B のように時刻 t_{max} にバリア同期ポイントに到達するようコードスケジューリングを行い，バリア同期コードを除去する．これにより，バリア同期オーバヘッドを約7クロック短縮することができる．

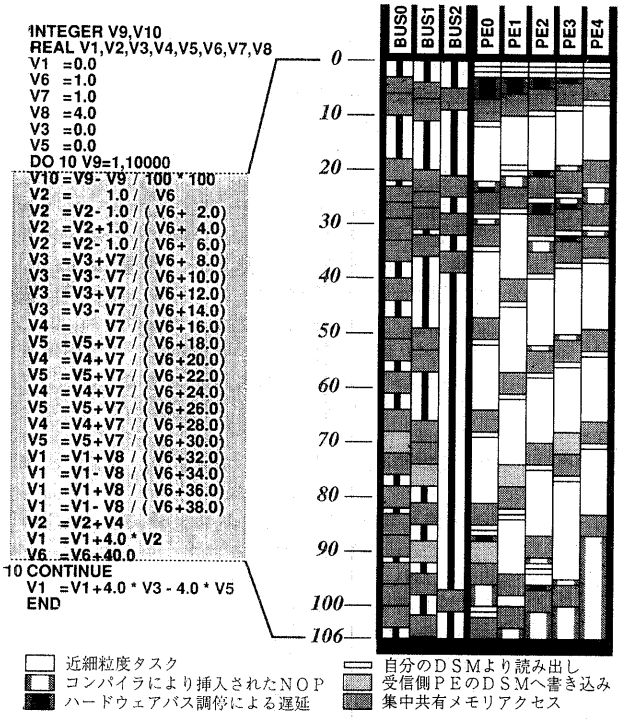
以上の 4.1〜4.4 節の操作で実際にクロックレベルの厳密なコードスケジューリングを行った例を**図 9** Bに示す．この例では図 9 A のプログラムの Do ループのボディ部を PE 5 台で並列処理するものとして，スケジューリングを行っている．

このコードスケジューリング手法の計算量は，ある基本ブロックに含まれる近細粒度タスクの数を n，プロセッサの数を m とすると，コードスケジューリングに要するコストのオーダは $O(n \times m)$ であり，コンパイル時に多大な負担は生じない．例えば図9 Bのようなコードスケジューリング結果は WS（HP 社 Apollo 400 シリーズ）上で1秒以下で得られる．

5. 無同期実行の性能評価

本章では，提案する無同期近細粒度並列処理手法を OSCAR 上でインプリメントし，実際に Fortran プログラムに提案手法を適用して性能評価を行った結果について述べる．

図9 Aに示すプログラムは円周率計算を行うもの



(A) 例題プログラム1　　(B) コードスケジューリング結果

図 9 例題プログラムとそのコードスケジューリング
Fig. 9 Example program and its scheduled parallel code.

で，オリジナルループに対しループアンローリングを適用し，24ステートメントから成るループボディを生成している．ここでは，このループを1万回繰り返し性能評価を行った．

これを，PE 台数を変化させながら近細粒度並列処理した場合の実行時間を**図10**に示す．このグラフ中に示される3種類の線は，上から順に，異なる PE 間の同期を全く除去しない場合（細実線），冗長なフラグ転送を除去[12]した場合（鎖線）と，提案する無同期並列処理方式の場合（太実線）の結果をそれぞれ示す．なお，図10のグラフ中の数字は基本ブロックの1回の実行中でとられるデータ同期（フラグ送信 S-n，フラグ受信 R-n）の回数を示している．また，図には表されていないが，基本ブロックの終端でバリア同期を1回処理している．

無同期実行では，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．無同期実行で先行制約を保証できなければ，データ転送が正しく行われず演算結果に重大な過ちが引き起こされるが，同期除去前，冗長な同期除去，本手法による無同期実行の，いずれでもプログラムの演算結果は全く等しかった．よって本手法で同期をすべて除去しても，タスク間の先行制約を満足しつつ実行が行われていることがわかる．

次に，PE 3 台を用いて近細粒度並列処理を行った場合の実行時間を見ると，全く同期を除去しない場合は，図 10 に示すように基本ブロックの実行1回につきフラグ送信 18 個とフラグ受信 38 個，バリア同期1個の処理を含めて，実行時間が 92.63 μs だったのに対し，本手法で無同期実行した場合は 61.76 μs に短縮されており，同期コードの除去により実時間で 30.87 μs の短縮が得られ，約 33.3% の速度向上が達成されたことがわかる．同様にして，PE 6 台の場合には，同期除去前に対して，本手法を適用した場合には実時間で 19.26 μs だけ実行時間が短縮され，34.3 % の速度向上が得られている．

この例では，無同期並列処理手法を適用することにより同期除去前に比べて平均30%，冗長な同期除去に比べて約10% 程度速度が向上することが確認された．

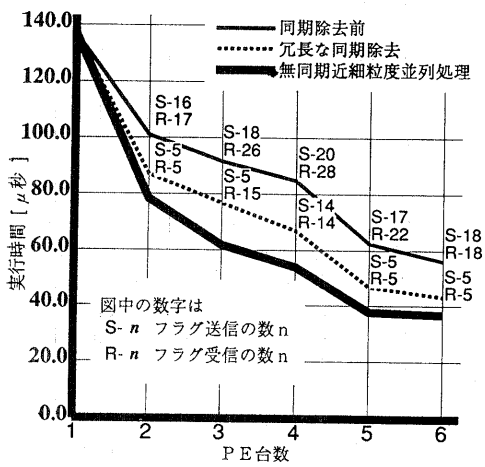


図 10 例題の OSCAR 上での並列実行時間と無同期近細粒度並列処理の効果
Fig. 10 Parallel execution time of example on OSCAR and effectiveness of near fine grain parallel processing without synchronization.

6. まとめ

本論文ではクロックレベルの高精度スタティックコード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，基本ブロック中のすべての同期コードを除去する並列コード生成手法について提案した．

また，実際にマルチプロセッサシステム OSCAR 上で提案手法をインプリメントし，バスアクセスを含めた命令実行のクロックレベルでの最適化と，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，基本ブロックの近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．

今後の課題としては無同期近細粒度並列処理用コードスケジューリング時のプロセッサ間データ転送順序最適化による実行時間の短縮およびマルチグレイン並列処理と無同期近細粒度並列処理の融合があげられる．

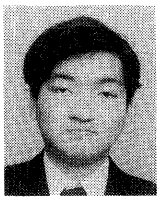
謝辞 本研究に使用したマルチプロセッサシステム OSCAR の製作，メンテナンスをしていただいた，富士ファコム制御(株)の皆様に，感謝いたします．

なお，この研究の一部は，文部省科学研究費補助金奨励研究 (B) 05452354 および (C) 05680284 による．

参考文献

1) 富田眞治，末吉敏則：並列処理マシン，オーム社，東京 (1989)．
2) 笠原博徳：並列処理技術，コロナ社，東京 (1991)．
3) Wolfe, M.: *Optimizing Supercompilers for Supercomputers*, MIT Press (1989).
4) O'Keefe, M. T. and Dietz, H. G.: Hardware Barrier Synchronization Static Barrier MIMD (SBM), *International Conference on Parallel Processing*, Vol. I, pp. 35-42 (1990).
5) O'Keefe, M. T. and Dietz, H. G.: Hardware Barrier Synchronization Dynamic Barrier MIMD (DBM), *International Conference on Parallel Processing*, Vol. I, pp. 43-46 (1990).
6) Birk, Y., Gibbons, P. B., Sanz, J. L. C. and Soroker, D.: A Simple Mechanism for Efficient Barrier Synchronization in MIMD Machines, *International Conference on Parallel Processing*, Vol II, pp. 195-198 (1990).
7) Beckmann, C. J. and Polychronopoulos, C. D.: Fast Barrier Synchronization Hardware, *Proc. Super Computing 90*, IEEE, pp. 180-189 (1990).
8) Zaafrani, A., Dietz, H. G. and O'Keefe, M. T.: Static Scheduling for Barrier MIMD Architectures, *International Conference on Parallel Processing*, Vol. II, pp. 187-194 (1990).
9) 有田五次郎，末吉敏規：FIFO キューを同期手段とする並列プログラムについて(3)実行管理機構，情報処理学会論文誌，pp. 838-846 (1983).
10) 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 (1990).
11) 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 (1992).
12) 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，情報処理学会論文誌，Vol. 28, No. 10, pp. 1060-1070 (1987).
13) 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. J 71-D, No. 8, pp. 1440-1445 (1988.)
14) Kasahara, H., Honda, H. and Narita, S.: Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR, *Proc. IEEE SuperComputing '90*, pp. 856-864 (1990).
15) 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. J 67-D, No. 7, pp. 792-799 (1984).
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処理手法，信学誌，Vol. 73-D-I, No. 9, pp. 756-766 (1990).
17) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers : Principles, Techniques, and Tools*, Addison-Wesley (1986).
18) Padua D. A. and Wolfe M. J.: Advanced Compiler Optimization for Super Computers, *C. ACM*, Vol. 29, No. 12, pp. 1184-1201 (1986).
19) 笠原博徳，合田憲人，吉田明正，岡本雅巳，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. J 75-D-I, No. 8, pp. 511-525 (1992).
20) 吉田明正，岡本雅巳，合田憲人，尾形 航，本多弘樹，笠原博徳：OSCAR Fortran マルチグレインコンパイラ，情報処理学会研究会報告 92-PRG-9-10, pp. 71-78 (1992).

(平成5年9月16日受付)
(平成6年2月17日採録)



尾形　航（正会員）

昭和42年生．平成3年早稲田大学理工学部電気工学科卒業．平成5年同大大学院修士課程修了．現在，同大学院博士課程在学中．並列実行方式，近細粒度並列処理手法，計算機アーキテクチャの研究に従事．

吉田　明正（正会員）

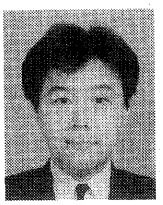
1968年生．1991年早稲田大学理工学部電気工学科卒業．1993年早稲田大学大学院修士課程修了．1993年より日本学術振興会特別研究員．現在，早稲田大学大学院博士課程在学中．並列実行方式，並列化コンパイラ等の研究に従事．電子情報通信学会会員．

合田　憲人（正会員）

平成2年早稲田大学理工学部電気工学科卒業．平成4年同大大学院修士課程修了．現在，同大学院博士課程在学中．平成4年同大情報科学研究教育センター助手，並列化コンパイラ，マルチプロセッサスーパーコンピュータの研究に従事．電子情報通信学会会員．

岡本　雅巳（正会員）

昭和41年生．平成2年早稲田大学理工学部電気工学科卒業．平成4年同大大学院修士課程修了．現在，同大学院博士課程在学中．平成5年同大情報科学研究教育センター助手．並列化コンパイラ，並列処理方式の研究に従事．電子情報通信学会会員．



笠原　博徳（正会員）

昭和55年早稲田大学理工学部電気工学科卒業．昭和60年同大学院博士課程修了．工学博士．昭和58年～60年早稲田大学理工学部助手．昭和60年カリフォルニア大バークレー短期客員研究員，日本学術振興会第1回特別研究員．昭和61年早稲田大学電気工学科専任講師．昭和63年同助教授．平成3年情報学科助教授，現在に至る．平成元年～2年イリノイ大学 Center for Supercomputing R & D 客員研究員．昭和62年 IFAC World Congress 第1回 Young Author Prize 受賞．主な著書「並列処理技術」（コロナ社）．電子情報通信学会，電気学会，シミュレーション学会，ロボット学会，IEEE, ACM 等の会員．