論　文

バリヤ同期除去による行列演算プログラムのアイドル時間の削減

米澤　直記[†a)]　　和田　耕一[††b)]

Reducing Idle Times on Matrix Programs by Eliminating Barrier Synchronizations

Naoki YONEZAWA[†a)] and Koichi WADA[††b)]

あらまし　バリヤ同期は，同期点の前までに発生したイベントが同期点までに完了すること，及びそれらのイベントの結果をすべてのプロセッサが等しく観察することを保証するために，プログラマによって並列プログラム中に記述される命令である．しかしながら，バリヤ同期は，(1) プロセッサ数が多くなるにつれて，バリヤ同期にかかるコストも増加する，(2) プロセッサのアイドル時間の増加をもたらす原因となる，といった性能面に悪影響を与える側面をもつ．この傾向は，PC クラスタ上に実現された OpenMP 実行環境などの，分散メモリ環境上の共有メモリ環境で，特に顕著に現れる．本論文では，バリヤ同期を除去するコンパイラアルゴリズムを新たに提案する．評価として，ヤコビの反復法，ガウスの消去法，及び SPLASH-2 LU を PC クラスタで実行し，本アルゴリズムの適用前後での実行時間の短縮率を測定した．その結果，(1) 本アルゴリズムを適用することによって，プロセッサ数が 4 以上の場合，常に実行時間が短縮し，(2) プロセッサ数が多いほど，バリヤ同期の除去の効果が高いことが明らかになった．

キーワード　バリヤ同期除去，OpenMP コンパイラ，PC クラスタ，配列範囲記述子，分散共有メモリ

1. まえがき

近年，共有メモリを想定したプログラミングのための標準仕様として，OpenMP [1] が注目されている．OpenMP では，既存の逐次プログラムに対して，並列実行する部分を指定する指示文を追加することで，簡潔にプログラムの並列化を実現することができる．しかしながら，OpenMP が主にターゲットとしているプラットホーム，すなわち物理的に共有メモリをもつ計算機は，PC クラスタと比較して，価格性能比，及びスケーラビリティの点で劣るという欠点をもっている．そこで，これまで，多くの研究者たちは，PC クラスタなどの分散メモリ環境に，仮想的に共有メモリ環境を実現するための機構を開発してきた [2]～[6]．その中の一つとして，我々は，共有メモリへのアクセスの解析による通信コードの生成，という方法で仮想的な共有メモリ環境を実現している [7]．具体的には，我々の開発しているコンパイラ Quaver では，共有変数へのアクセスを検出し，その変数の生産–消費関係を特定する．そして，その依存関係を満たすための通信コード，すなわち生産者が消費者に対して適切にデータを転送するコードを OpenMP プログラムに追加する．

一般に，分散メモリ環境では，通信にかかるコストが大きいため，通信が性能上のボトルネックになる傾向がある．そのため，通信オーバヘッドを削減，または隠ぺいするための数々の手法が提案されてきた．それらの提案として，データのキャッシング，データの最適配置，メモリコンシステンシモデルの緩和，通信時間と計算時間の重ね合わせ等が挙げられる．

本論文では，通信コスト削減手法として，バリヤ同期の除去に注目する．バリヤ同期とは，同期点の前までに発生したイベントが同期点までに完了すること，及びそれらのイベントの結果をすべてのプロセッサが等しく観察することを保証するために，プログラマによってプログラム中に記述される命令のことである．分散メモリ環境においては，バリヤ同期は，しばしば全対 1，1 対全のメッセージ交換として実装される．すなわち，同期点に到達したプロセッサは，バリヤ同期

† 神奈川大学理学部情報科学科，平塚市
Department of Information and Computer Science, Faculty of Science, Kanagawa University, Hiratsuka-shi, 259–1293 Japan
†† 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻，つくば市
Department of Computer Science, University of Tsukuba, Tsukuba-shi, 305–8571 Japan
a) E-mail: yonezawa@info.kanagawa-u.ac.jp
b) E-mail: wada@cs.tsukuba.ac.jp

を管理するプロセッサ（バリヤマスタ）に対して，同期点に到達した旨を通知し，一方で，バリヤマスタは全プロセッサからの通知がそろったことを確認し次第，全プロセッサに対して，同期点の次の命令を実行する許可を与える．そのため，プロセッサ数が多くなると，同期にかかるコストも増加することになる．更に，バリヤ同期はプロセッサのアイドル時間の増加をもたらす原因ともなる．負荷分散が不均衡であるときは，同期点に先に到着したプロセッサが他のプロセッサの到着を待たなければならないため，先に到着したプロセッサはアイドル状態になってしまう．プロセッサ間の負荷分散を均等にすることができた場合でも，ネットワーク資源への競合等の原因で，通信時間がランダムに遅延することにより，アイドル時間の増加が起こり得る．

本研究は，PC クラスタ向け OpenMP コンパイラにおいて，バリヤ同期を除去するコンパイラアルゴリズムを新たに提案し，行列演算プログラムに適用したときの有効性を評価することを目的とする．先述のとおり，我々が開発している OpenMP コンパイラ Quaver では，送受信コードを生成する．本論文で我々が提案する手法は，それらの送受信コードによって，イベント間の順序付け，すなわち共有メモリへのアクセスの順序付けが保証されるという事実を利用し，依存関係が静的に解析可能な場合に，バリヤ同期を送受信コードで置換するものである．Tseng が提案する手法 [8]，すなわちバリヤ同期で分割される，連続する二つのフェーズの間に依存関係がなければ，そのバリヤ同期を除去してもよいとする手法とは異なり，我々の手法では，たとえ依存関係があってもバリヤ同期を除去できる，という利点がある．ここで，フェーズとは，バリヤ同期とその次のバリヤ同期の間で実行されるコードの集合である，と定義する．また，バリヤ同期を除去した後も，これらの一連のコードの集合のことを，引き続きフェーズと呼ぶこととする．

以下，本論文は次のように構成される．**2.** では我々が開発している OpenMP コンパイラ Quaver について述べる．**3.** ではバリヤ同期の除去手法を提案し，続く **4.** では本手法を行列演算プログラムに適用したときの有効性を評価する．**5.** では関連研究について述べ，**6.** でまとめとする．

2. 配列範囲記述子を利用する OpenMP コンパイラ Quaver

本論文で提案する手法は，我々が開発している OpenMP コンパイラ Quaver の一機能として実装されている．したがって，提案手法の詳細を述べる前に，本章で，Quaver における解析方法について述べる．

2.1 並列処理に適した配列範囲記述子 quad

Quaver では，共有変数へのアクセスを検出し，依存関係を求める．そのため，コンパイラ内部で，共有変数へのアクセス情報を表現する必要がある．我々は，並列処理に頻出するアクセスパターンを簡潔に表現することが可能な配列範囲記述子として quad [9] を，Quaver の内部で利用している．

一般に，共有メモリ型プログラミング言語には，共有配列を均等に分割し，各プロセッサに割り当てるための記法が用意されていることが多い．OpenMP でも，ブロック割当，サイクリック割当，ブロックサイクリック割当を指定するための記法が用意されている(注1)．ブロックサイクリック割当されたデータに対するアクセスパターンは，図 1 (a) に示されるように，一定の長さのアクセス領域と一定の長さの非アクセス領域を交互に含むものとなる．また，分割された空間の境界外の近傍を含めてアクセスする問題に見られるように，割り当てられた領域よりも少し広い領域をアクセスする場合でも，同様のストライドアクセスとなる（図 1 (b)）．

quad は，4 個の整数 a，b，c，d から構成され，それぞれの整数は，

a：アクセスされる領域の，配列先頭からのオフセット

b：アクセス領域の長さ

c：非アクセス領域の長さ

d：繰返し数

を表す．

我々は，quad 間の積演算及び和演算を定義し，実装

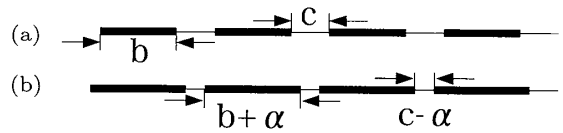


図 1　並列プログラムに典型的に出現するアクセスパターン
Fig. 1　Typical access patterns observed in parallel programs.

(注1)：より正確に記述すれば，OpenMP で分割する対象は，配列ではなく，配列をアクセスするループの実行である．しかしながら，ループの並列化を適切に指定することにより，各スレッドがアクセスする配列範囲を固定的に割り当てることが可能であるため，本文中では「配列を分割する」と表現する．

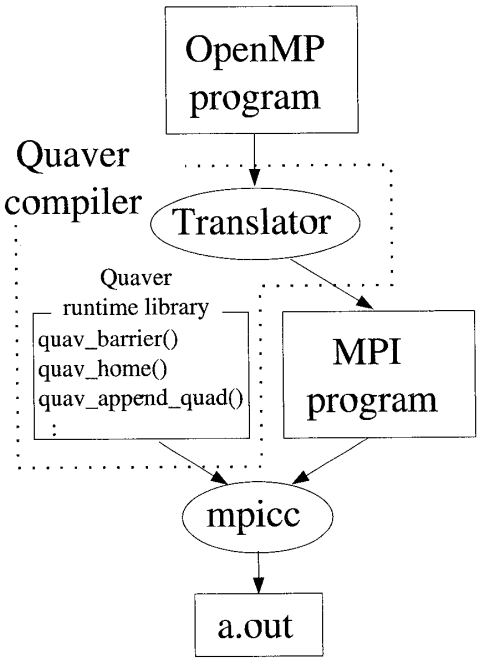


図 2　Quaver によるコンパイルの流れ
Fig. 2　The flow of compiling.

している．積演算によって，アクセスされるデータ間の依存関係を調べて，データの消費プロセッサに転送すべきデータを求めることができる．また，和演算は，多数の quad を，それらによって表現されるアクセス領域に関する情報を保ったまま，より少ない quad に変換する．このことにより，quad のために消費されるメモリ領域を節約することができる．

2.2　Quaver によるコンパイルの流れ

Quaver は，図 2 に示すように，トランスレータと実行時ライブラリから構成される．トランスレータは，OpenMP プログラムを解析し，共有変数へのアクセスを検出する．検出されたアクセス領域は，quad で表現される．OpenMP ディレクティブを検出した場合は，ディレクティブに応じて，適切な実行時ライブラリ関数を挿入する．トランスレータによって生成されたコードは，`mpicc` によってコンパイルされ，PC クラスタで実行可能なファイルが生成される．

実行時には，同期点において，quad 間の積演算によって転送すべきデータが求められ，MPI_Send() 等によって送信されることになる．

2.3　Quaver 実行時ライブラリ

Quaver 実行時ライブラリは，以下の関数群から構成される．

- **quav_append_quad()** アクセス情報を表現する quad を，quad テーブルと呼ばれるメモリ領域に登録する．quad のほかに，アクセスタイプ（読出しまたは書込み），プロセッサ ID も引数として渡され，quad テーブルに登録される．
- **quav_divide_loop_and_append_quad()** 次の操作を 1 関数で行う：(1) ループ全体のアクセス情報，及びそのループを分割する OpenMP ディレクティブの情報をもとに，この関数の呼出しプロセッサがアクセスする配列範囲を求める，(2) その配列範囲を quad で表現する，(3) quav_append_quad() を呼び出す．現在のところ，OpenMP 仕様の `schedule` 節のうち，`static` にのみ対応している．
- **quav_assign_home()** この関数は，共有配列が宣言された後に呼び出される．配列を均等に分割し，部分配列を各プロセッサに割り当てる．部分配列を割り当てられたプロセッサを，その部分配列のホームプロセッサと呼ぶ．同期点のあと，ホームプロセッサには，最新データが存在することが保証される．
- **quav_home()** この関数は，典型的には，ディレクティブによって並列実行を指示されたループの本体で呼び出される．引数として受け取ったループ制御変数とプロセッサ ID をもとに，そのプロセッサがループ本体の計算をすべきかどうかを決定する．この関数によって，並列実行を指示されたループは，各プロセッサによって排他的に並列実行されることになる．
- **quav_barrier()** すべてのプロセッサがバリヤ同期に到着するまで，プロセッサの実行をブロックする．このバリヤ同期の後に読出しアクセスされるデータは，この関数内で特定され，消費プロセッサに転送される．この関数は，**1.** で述べたバリヤマスタを利用した同期方式を実装している．
- **quav_fetch()** 読み出されるデータをコンパイル時に特定することができなかった場合に，実行時に，読出しに関する quad をもとに，ホームプロセッサに読出し対象のデータを要求し，返答としてデータを受け取る．この関数を使用することにより，共有変数のアクセス解析が困難なプログラム，あるいはアクセスが動的に定まるプログラムも実行することが可能となる．なお，読み出されるデータがコンパイル時に特定される場合は，quav_barrier() が終了する時点までにデータが読出しプロセッサに到着するため，Quaver は quav_fetch() の呼出しコードを生成しない．

quav_barrier() と quav_fetch() では，プロセッサが，他プロセッサに要求を出し，その返答を待っている間に，他プロセッサから別の要求が到着した場合は，適

宜その要求を処理して返答する．

2.4 アクセス情報の抽出と通信コードの生成

OpenMP プログラムの解析時に，Quaver トランスレータは，共有変数へのアクセス箇所を特定する．アクセスが書込みの場合，そのアクセス情報を表現する quad を引数として，quav_append_quad() がその書込みアクセスの直後に生成される．一方，読出しアクセスの場合，quav_append_quad() は，当該データが生成される書込みアクセスの特定が可能な場合，その直後に生成される．書込みアクセスの特定が不可能な場合，データの読出しアクセスの直前に，quav_fetch() とともに生成される．なお，quad によって表現されるアクセス情報は，巻上げ (hoisting) という処理によって，quad の変形を伴いながら，可能な限り上位ブロックに巻き上げられる．巻上げ処理によって，quad が縮約され，実行時に生成される quad の数が少量に抑えられる．巻上げ処理の詳細については，**3.2** で述べる．

また，同期点においては，トランスレータは，quav_barrier() を生成する．

トランスレータによって，図 3 に示される OpenMP プログラムが解析された結果として生成されるコードを図 4 に示す．変換によって追加されたコードを斜体で表す．

例えば，このプログラムを 4 プロセッサで実行する場合には，全プロセッサが quav_divide_loop_and_append_quad() を呼び出すことによって，for ループ内で各プロセッサが書込みアクセスする配列範囲が，quad テーブルに登録される．

```
#pragma omp parallel for
for (i = 0; i < 16; i++)
    a[i] = i * i;
b = a[2] + a[3] + a[4] + a[5];
```

図 3 OpenMP プログラムの例
Fig. 3 An example of OpenMP program.

```
quav_divide_loop_and_append_quad(mypid, write, a(0, 16,
0, 1), static, nochunk);
#pragma omp parallel for
for (i = 0; i < 16; i++) {
    if (! quav_home(a, i, mypid)) continue;
    a[i] = i * i;
}
quav_append_quad(p0, read, a(2, 4, 0, 1));
quav_barrier();
if (mypid == p0)
    b = a[2] + a[3] + a[4] + a[5];
```

図 4 Quaver 実行時ライブラリの挿入
Fig. 4 Inserting Quaver runtime library.

ここでは，配列 a の連続する 4 個の要素に対して各プロセッサが書込みアクセスするので，例えばプロセッサ 1 は quad (4, 4, 0, 1) を登録することになる．

次の for ループでは，quav_home() によって，各プロセッサが for 文を排他的に分担して実行する．続く quav_append_quad() では，次フェーズにおいてプロセッサ 0 が読出しアクセスする配列範囲，すなわち quad (2, 4, 0 ,1) が quad テーブルに登録される．

quav_barrier() 内では，各プロセッサが，読出しアクセスと書込みアクセスに関する quad 間で積演算を施すことにより，データの依存関係を調べて，必要であれば，データを消費プロセッサに送信する．例えば，プロセッサ 1 は，プロセッサ 0 による読出しアクセスを表現する quad (2, 4, 0, 1) と，プロセッサ 1 による書込みアクセスを表現する quad (4, 4, 0, 1) との間の積演算を実行し，その結果として quad (4, 2, 0, 1) が求まる．この演算結果をもとに，プロセッサ 1 から a[4] と a[5] がプロセッサ 0 に送信される．プロセッサ 2 と 3 では，積演算の結果が空になるためデータは送信されない．

上の例では，書込みプロセッサとホームプロセッサが同一であるが，異なる場合，quav_barrier() 内で，書込みプロセッサからホームプロセッサへ最新データを転送する．書込みプロセッサは，quav_assign_home() によって与えられているホームプロセッサへの配列領域の割当情報をもとに，ホームプロセッサによる担当領域全体への read アクセスが起きるものとして扱う．すなわち，自プロセッサ以外のホーム担当情報と自プロセッサによる書込みアクセス情報との積演算を実施することによって，転送すべきデータとあて先プロセッサを特定する．

すべてのデータ通信が終了したあと，各プロセッサはバリヤマスタ（現在の Quaver の実装では，プロセッサ 0 が担当する）に対してバリヤに到着したことを伝える．バリヤマスタは，すべてのプロセッサがバリヤに到着したことを確認した後，各プロセッサに対して，次フェーズの実行を許可するメッセージを送信する．

3. バリヤ同期の除去手法

本章では，本論文で提案するバリヤ同期を除去するアルゴリズムについて述べる．

3.1 実行時ライブラリ関数の追加

後述するバリヤ同期の除去アルゴリズムにおいて，quav_barrier() と置換する関数群を，本節で定義する．

- **quav_append_quad_for_next_phase()** この関数は，quav_append_quad() と同様に，アクセス情報を quad テーブルに登録する．登録される quad は，この関数の呼出しプロセッサが次フェーズで読み出す配列範囲を表現し，次に述べる quav_send_recv_if_needed() で，受信すべきデータの総量を求めるために用いられる．
- **quav_send_recv_if_needed()** 書込みアクセス情報，ホーム担当情報，及び quav_append_quad_for_next_phase() で与えられた読出しアクセス情報をもとに，受信すべきデータの総量を求め，必要であればデータを受信する．また，quav_barrier() と同様に，消費プロセッサに送信すべきデータがあれば送信する．

quav_append_quad_for_next_phase() によって与えられた，自プロセッサの次フェーズにおける読出しデータが，quav_send_recv_if_needed() ですべて到着すれば，他プロセッサの進行状況に関係なく，自プロセッサは次フェーズに移行できる．このプロセッサ間依存の緩和が，我々の提案手法の本質である．

なお，書込みプロセッサとホームプロセッサが異なる場合は，**2.4** の quav_barrier() と同様に，quav_send_recv_if_needed() の終了までに，書込みプロセッサからホームプロセッサへ適切にデータが転送される．ただし，quav_barrier() のときは書込みプロセッサ側でのみ積演算が必要であったが，quav_send_recv_if_needed() では，最新データの到着を待つことができるように，ホームプロセッサも積演算を実施し，自分が受け取るべきデータ量を求める必要がある．

3.2 アクセス情報の巻上げ処理

1. でも述べたように，バリヤ同期はデータの生産と消費の順序付けを保証するためにプログラム中に挿入されるが，データの生産者から消費者へ直接データを転送する送受信コードによっても順序付けをすることが可能である．全プロセッサ間の順序付けを保証するバリヤ同期を除去し，各プロセッサ間の送受信コードで置換するためには，データの送信者が他のすべてのプロセッサへ送るべきデータを特定でき，なおかつデータの受信者が他のすべてのプロセッサから送られてくるデータを特定できなければならない．その際，送受信すべきデータは，書込みアクセスと読出しアクセスのすべてが特定できていれば求めることが可能である．このうち，書込みアクセスについては，送受信の時点では過去に発生したイベントなので，特定が可能である．一方，読出しアクセスについては，必ずしも特定できるとは限らない．

すなわち，一般に，あるバリヤ同期の除去が可能である条件は，そのバリヤ同期の直後のフェーズにおける読出しアクセスのすべて(注2)が，バリヤ同期までに判明していること，と言い換えることができる．そのため，バリヤ同期の除去アルゴリズムについて述べる前に，フェーズにおける共有変数へのアクセスの有無を判定することを容易にするために実行される，アクセス情報の巻上げ処理について述べる．巻上げ処理は，Quaver を構成するモジュールである，アクセスサマリ生成フィルタ（以下，フィルタ）によって実行される．

フィルタは，C で記述された OpenMP プログラムを解析し，アクセス情報を注釈として挿入する．フィルタは，まず，プログラムコードから解析木を生成し，プログラム中の共有変数への代入や参照から，変数 1 個へのアクセスを表現する quad を求めて，対応する解析木のノードに格納する．

次に，前段階の処理によって解析木の最下位レベルに分散配置されている quad を，ループや関数といったより上位レベルのブロックに巻き上げる．また，連続する文では，それぞれのアクセスサマリ同士の和演算により，アクセスサマリを縮約する．巻上げの際には，アクセス情報が損なわれないように適宜 quad を変形する．

例えば，図 4 の for 文では，quad (i, 1, 0, 1) がフィルタ内部で一時的に生成された後，巻上げ処理によって (0, 16, 0, 1) に変形され，for 文のアクセスサマリとして生成されている．

quad は，変数ごと，アクセスタイプ（読出しまたは書込み）ごとに区別して取り扱い，巻上げ処理は，同じ変数，及び同じアクセスタイプ同士で実施される．

関数間の巻上げ処理では，関数のアクセスサマリを求めておき，呼出し側の文脈に置き換える．関数のアクセスサマリが仮引数，定数，または大域変数で表現されている場合，関数間での巻上げが可能である．

これらの巻上げとそれに伴う quad の変形を解析木の上位に向かって繰り返すことによって，アクセスサマリが縮約されていくが，巻上げ処理は，同期を指示

(注2)：除去対象のバリヤ同期の直後のフェーズより後のフェーズにおける読出しアクセスは，除去対象のバリヤ同期とは別のバリヤ同期によって順序づけられるため，ここでは考慮する必要はない．

するOpenMPディレクティブを超えて実施されない．これは，アクセスサマリ生成の目的がデータの依存解析に用いることにあり，同期点を超えた巻上げが意味をなさないためである．

巻上げができないquadについては，下位の解析木ノードに格納されたまま保持される．一方，もしより下位のアクセス情報をすべての変数について巻き上げることができた場合，アクセスタイプごとに*all_read_hoisted*または*all_write_hoisted*というフラグをセットする．これらのフラグは，すべてのアクセス情報の巻上げが可能である限り上位のブロックに伝搬されていく．すなわち，あるレベルのブロックにおいて，例えば*all_read_hoisted*フラグがセットされている場合，(1) それより下位の読出しに関するquadがすべて巻き上げられており，(2) そのレベルに巻き上げられたquadを見れば，下位ブロックを含むコード内のすべてのアクセス情報を過不足なく入手できることを意味する．

3.3 バリヤ同期の除去アルゴリズム

前節の冒頭でも述べたとおり，あるバリヤ同期の除去が可能である条件は，そのバリヤ同期の直後のフェーズにおける読出しアクセスのすべてが，バリヤ同期までに判明していること，である．アクセスサマリは巻上げ処理によって既に生成されているため，バリヤ同期の除去アルゴリズムは最早，バリヤ同期を見つけて，同期点における*all_read_hoisted*フラグを調べることに集約される．図5に，本論文で提案するバリヤ同期の除去アルゴリズムの概要を示す．OpenMPのparallelディレクティブ等による暗黙的なバリヤ同期は，本アルゴリズムの適用前までにquav_barrier()に明示的に置換されているものとする．

同期点において，*all_read_hoisted*がセットされている場合，フィルタによって注釈として埋め込まれたアクセス情報から，quav_append_quad_for_next_phase()をバリヤ同期の前に生成する．また，quav_barrier()をquav_send_recv_if_needed()に置き換える．

*all_read_hoisted*フラグがセットされていない場合，バリヤ同期を除去することができないが，同期点までの巻上げが成功した変数については，quav_barrier()内で消費プロセッサに転送される．これにより，quav_fetch()の起動によるプロセッサストールを避けることができる．一方，同期点までの巻上げができなかった変数については，巻上げができた範囲のうちで最上の地点にquav_fetch()を生成する．これにより，巻上げを全くしない場合と比較して，quav_fetch()の実行回数を減らすことが可能となる．

なお，コンパイル時に，フェーズ間に依存関係がないことが分かった場合，Tsengのアルゴリズムと同様に，quav_barrier()を除去する．

3.4 バリヤ同期の除去例

図6は，図4のコードに対して，バリヤ同期の除去アルゴリズムを適用した例を示している．データを受け取るプロセッサはquav_append_quad_for_next_phase()によって，受け取るべきデータ量を計算することができる．この例では，プロセッサ0のみが次フェーズで読出しアクセスを行うので，この関数の呼出しもプロセッサ0のみが行う．quav_send_recv_if_needed()では，各プロセッサは，quav_barrier()のときと同様に，データを送信すべきかどうか調べて，積演算の結果が空でなければ送信する．そして，quav_append_quad_for_next_phase()から得られた情報により，受け取るべきデータ量が0になるまで，他プロセッサからのデータを受信する．

例えば，このプログラムを4プロセッサで実行

```
void eliminate_barrier(void)
{
  foreach (code_line) {
    if (found a quav_barrier() and all_read_hoisted) {
      generate quav_append_quad_for_next_phase();
      replace quav_barrier()
                with quav_send_recv_if_needed();
    }
  }
}
```

図5 バリヤ同期の除去アルゴリズム
Fig. 5 An algorithm for eliminating barrier synchronizations.

```
quav_divide_loop_and_append_quad(mypid, write, a(0,
16, 0, 1), static, nochunk);
#pragma omp parallel for
for (i = 0; i < 16; i++) {
    if (! quav_home(a, i, mypid)) continue;
    a[i] = i * i;
}
quav_append_quad(p0, read, a(2, 4, 0, 1));
if (mypid == p0) quav_append_quad_for_next_phase(p0,
read, a(2, 4, 0, 1));
quav_send_recv_if_needed(mypid);
if (mypid == p0)
    b = a[2] + a[3] + a[4] + a[5];
```

図6 バリヤ同期を除去された後のプログラム例
Fig. 6 An example code after eliminating a barrier synchronization.

する場合には，プロセッサ 1 は，**2.4** で示した例と同様に，quav_append_quad() で与えられる，プロセッサ 0 による読出し情報に基づき，a[4] と a[5] をプロセッサ 0 に送信した後，プロセッサ 1 が受け取るべきデータを計算する．プロセッサ 1 は quav_append_quad_for_next_phase() を呼び出していないことから，受け取るべきデータを求める quad 間演算の結果は空であるとみなし，受取り処理をせずに次フェーズに進むことができる．

プロセッサ 2 と 3 については，**2.4** と同様に，他プロセッサにデータを送信しない．また，プロセッサ 1 と同じ理由により，受け取るべきデータもないことから即座に次フェーズに進む．

プロセッサ 0 は，quav_divide_loop_and_append_quad() で内部的に生成される他プロセッサの書込みアクセスに関する quad と quav_append_quad_for_next_phase() で与えられる自プロセッサの読出しアクセスに関する quad との間の積演算から受け取るべきデータを求める．その結果，プロセッサ 1 から配列の 2 要素を受け取るべきことが判明する．その後，実際に，プロセッサ 1 から送られてくる配列の 2 要素を受信した後，次フェーズに進む．

ここで，受け取るべきデータを求める際には，それぞれの関数から与えられる情報のみが必要であり，他プロセッサとの通信は一切発生しないことに注意されたい．

図 4 の例と比較すると，図 7 に示すように，バリヤ同期を実現するためのメッセージの送受信が削減される点に加えて，データを受信しないプロセッサが，他プロセッサの進行状況に関係なく次フェーズの処理を開始することができる点が異なる．このことにより，アイドル時間が減少することが期待できる．

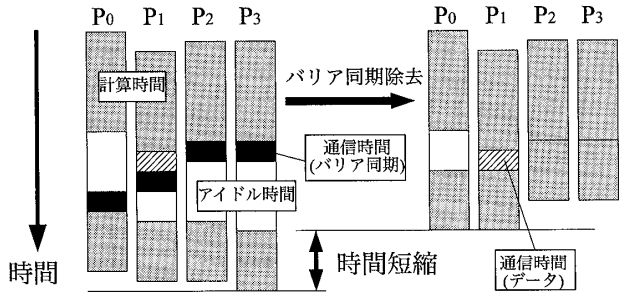


図 7　バリヤ同期除去による実行時間の短縮
Fig. 7　Reduction of execution time expected from barrier elimination.

4. 性能評価

本論文で提案する，バリヤ同期の除去手法を評価するため，アプリケーションプログラムを PC クラスタ上で実行し，バリヤ同期，すなわち quav_barrier() を除去しない場合の実行時間と比較した．また，バリヤ同期の除去を実現するために生じるオーバヘッドとして quad 間演算時間を測定し，実行時間に占める割合を評価した．更に，バリヤ同期の除去によって達成されたプロセッサのアイドル時間の短縮について定量的に評価した．

実験に使用した PC クラスタは，ギガビットイーサネットスイッチ（Dell 社 PowerConnect 5224）で接続された 16 台の PC から構成される．ギガビットイーサネットはジャンボフレームモードを利用しない状態で実験した．PC は，いずれも Intel 社 Xeon 2.8 GHz の CPU，1 GByte の主メモリ，Intel 社 82545GM ギガビットイーサネットコントローラを搭載している．また，各 PC で稼動するオペレーティングシステムとして Linux 2.6.8 を，PC クラスタ内のメッセージ通信ライブラリとして MPICH 1.2.6 を利用した．

4.1 アプリケーションプログラム

本評価では，アプリケーションプログラムとして，3 種類の行列演算プログラム，すなわちヤコビの反復法，ガウスの消去法，及び SPLASH-2 ベンチマーク [10] に所収されている LU を用いた．いずれのアプリケーションプログラムにおいても，計算対象の行列は，二重ポインタによって動的に確保した領域として実現しており，その要素は倍精度浮動小数点実数である．

4.1.1 ヤコビの反復法

ヤコビの反復法は，連立一次方程式 $\mathbf{Ax} = \mathbf{b}$ を解くアルゴリズムの一種であり，反復計算により $\mathbf{x}$ の値を徐々に真の解に収束させていく方法である．$k+1$ 番目の反復において $\mathbf{x}$ の各要素を求めるとき，k 番目の反復によって求められた $\mathbf{x}$ のすべての要素が必要となる．

本評価で用いたプログラムでは，各プロセッサはブロック分割された $\mathbf{x}$ の部分ベクトルを計算することで，性能向上を期する．この並列化手法では，$k+1$ 番目の反復において，各プロセッサが $\mathbf{x}$ の部分ベクトルの計算を開始する前に，他のすべてのプロセッサが k 番目の反復で生成した $\mathbf{x}$ の部分ベクトルが必要となる．このデータ間の依存関係を順番づけるため，反復するごとにバリヤ同期が実行される．

なお，本評価では，行列の大きさに関係なく，反復回数を 1,000 回とした．

ヤコビの反復法と同様のメモリアクセスパターンを示すアプリケーションの例として，タイムステップごとにオブジェクトあるいはエージェント間で情報を交換するシミュレーションが挙げられる．

4.1.2 ガウスの消去法

ガウスの消去法も，連立一次方程式を解くアルゴリズムであり，行列積算を繰り返すことによって $\mathbf{A}$ を上三角行列 $\mathbf{U}$ に変換する．$\mathbf{A}$ の k 列目の消去では，$\mathbf{A}$ の k 行が必要となる．

本評価で用いたプログラムでは，各プロセッサは均等に分割された $\mathbf{A}$ を対象とした計算を担当する．消去が進むにつれて，計算対象となる $\mathbf{A}$ の部分が徐々に小さくなっていくため，計算の分割方式として，行方向でのサイクリック分割を採用した．消去を実行するループの k 回目では，a_{kk} を用いて a_{ik} が消去され(注3)，他の a_{ij} についても a_{kk}，a_{kj}，a_{ik} を用いて計算される（ここで，$i > k$，$j > k$ である）．すなわち，各プロセッサが i 行目の各要素について計算するときは，$\mathbf{A}$ の k 行のうち，a_{kk} 以降の値が必要となる．このデータ間の依存関係を順番づけるため，1 列が消去されるごとに，バリヤ同期が実行される．

計算対象が徐々に小さくなっていくガウスの消去法と同様のメモリアクセスパターンを示すアプリケーションには，最小二乗問題の解法である QR 分解をはじめ，多くの密行列向けの計算法が存在する．

4.1.3 SPLASH-2 LU

SPLASH-2 ベンチマークに含まれる 2 種類の LU のうち，本評価では，“contiguous blocks” [11] を利用した．この実装法では，演算対象の行列の各要素のアドレスが，部分行列内で，すなわちブロック内で連続となる．

オリジナル版の SPLASH-2 LU で実装されている行列分割方法を，OpenMP で記述する場合，プログラムが複雑になるため，本評価では，簡潔な OpenMP プログラムが得られるように，オリジナル版を修正した．まず，並列化された状態にあるオリジナル版を，いったん逐次プログラムに書き直し，その逐次プログラムに対して，7 個の OpenMP プリミティブを追加することによって，再度並列化した．最終的に得られたプログラムでは，行列の行をブロックサイクリックによって分割している．

SPLASH-2 LU の計算範囲は，ガウスの消去法と同様に徐々に小さくなるが，ブロック化したことにより，計算範囲が小さくなる度合の粒度が大きくなった．

4.2 アプリケーションプログラムの基本特性

バリヤ除去の効果を論じる前に，比較のための基準として，アプリケーションプログラムの基本特性を調査した．ここでは，単一プロセッサでの実行時間，プログラム中でのバリヤ同期の出現個数と実行時の実施回数を示す．本論文では，各アプリケーションプログラムの計算を行うループの前後に，Intel 社の Pentium シリーズ及び互換プロセッサに用意されている RDTSC (Read Time Stamp Counter) 命令を挿入し，ループの実行に要したサイクル数を計測することによって実行時間を求めた．

4.2.1 単一プロセッサでの実行時間

表 1 に，単一プロセッサにおける，アプリケーションプログラムの実行時間を示す．ヤコビの反復法とガウスの消去法では，計算対象となる行列の大きさ N を，2,048，4,096，8,192 と変化させた．SPLASH-2 LU では，N を 4,096 及び 8,192 とし，いずれの場合もブロックの大きさを 64 とした．

ヤコビの反復法では，N を 2 倍にすると，実行時間はほぼ 4 倍となった．同様に，ガウスの消去法では 7.80 倍前後，SPLASH-2 LU では 7.97 倍となった．

4.2.2 バリヤ同期の出現個数及び実施回数

表 2 は，バリヤ同期の除去アルゴリズムを適用する前のプログラムを PC クラスタで実行した際に実施された，計算ループ中のバリヤ同期の回数を示している．また，表中，括弧内の数字はソースコード上でのバリ

表 1 単一プロセッサでの実行時間（秒）
Table 1 Sequential execution time.

	行列の大きさ		
	2,048	4,096	8,192
ヤコビの反復法	42.15	167.79	670.02
ガウスの消去法	13.96	108.56	852.77
SPLASH-2 LU	-	72.55	578.06

表 2 バリヤ同期の実施回数（括弧内は出現個数）
Table 2 The number of barrier synchronizations appeared in the application programs.

	行列の大きさ		
	2,048	4,096	8,192
ヤコビの反復法	1,000 (1)	1,000 (1)	1,000 (1)
ガウスの消去法	2,047 (1)	4095 (1)	8,191 (1)
SPLASH-2 LU	-	128 (2)	256 (2)

(注3)：ただし，値を 0 とみなせばよいので，実際の計算は省略している．

ヤ同期の出現個数である．

ヤコビの反復法とガウスの消去法では，バリヤ同期は計算ループ中に 1 個のみ出現するが，実行時の実施回数は，ヤコビの反復法で反復回数を固定したため，N に関係なく一定であるのに対して，ガウスの消去法では N に比例して実施回数も増加した．一方，SPLASH-2 LU では，ループの繰返し回数は，N をブロックの大きさで割った数と等しく，1 回の繰返しごとに 2 回のバリヤ同期が挿入されているため，N によってバリヤ同期の実施回数が異なる．

4.3 バリヤ同期の除去アルゴリズムの効果

本論文で提案するバリヤ同期の除去アルゴリズムの有効性を検証する基準として，アルゴリズムを適用することによって削減できた，アプリケーションプログラム中のバリヤ同期の個数，及び PC クラスタにおけるアプリケーションプログラムの実行時間の短縮率を用いた．また，実行時間の短縮の主要因と予想されるアイドル時間の減少について，より詳細に評価した．

4.3.1 台数効果

図 8 から図 13 に，3 種のアプリケーションプログラムを PC クラスタに実行した際の台数効果を示す．バリヤ同期の除去前後の台数効果に加えて，比較のため，MPI 版プログラムについても示す．MPI 版プログラムでは，MPI_Bcast() や MPI_Allgather() 等の集合通信を可能な限り使用した．紙面の都合上，ヤコビの反復法とガウスの消去法の $N = 4{,}096$ の台数効果は割愛する．

いずれのアプリケーションプログラムでも，バリヤ同期を除去した場合の台数効果は，バリヤ同期除去前

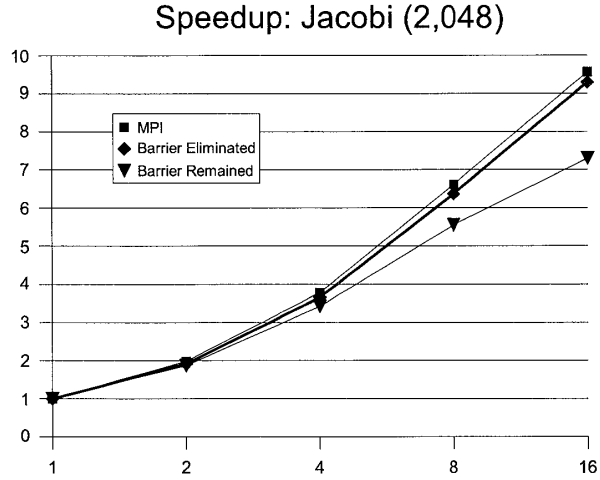


図 8 ヤコビの反復法の台数効果 ($N = 2{,}048$)
Fig. 8 Speedup of Jacobi method ($N = 2{,}048$).

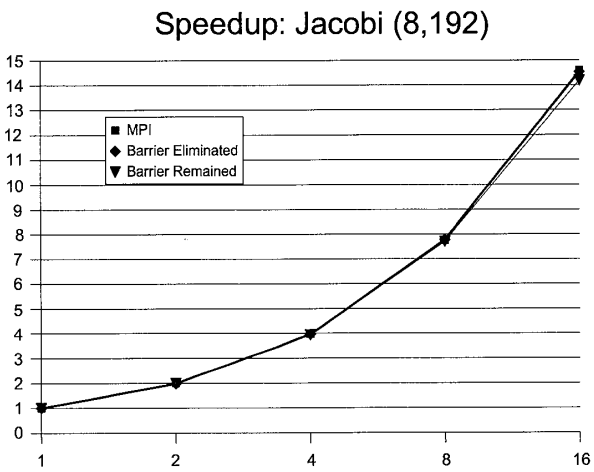


図 9 ヤコビの反復法の台数効果 ($N = 8{,}192$)
Fig. 9 Speedup of Jacobi method ($N = 8{,}192$).

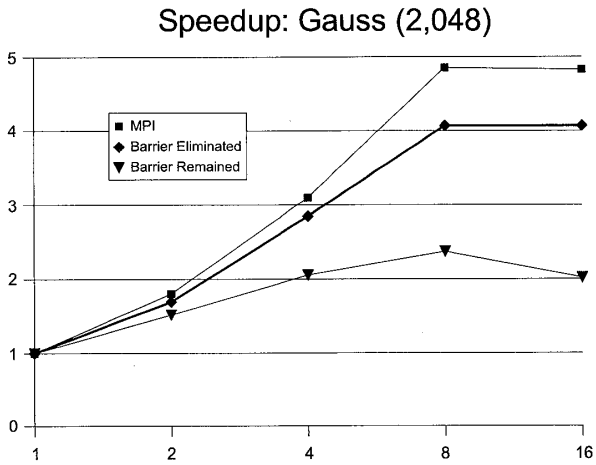


図 10 ガウスの消去法の台数効果 ($N = 2{,}048$)
Fig. 10 Speedup of Gaussian elimination ($N = 2{,}048$).

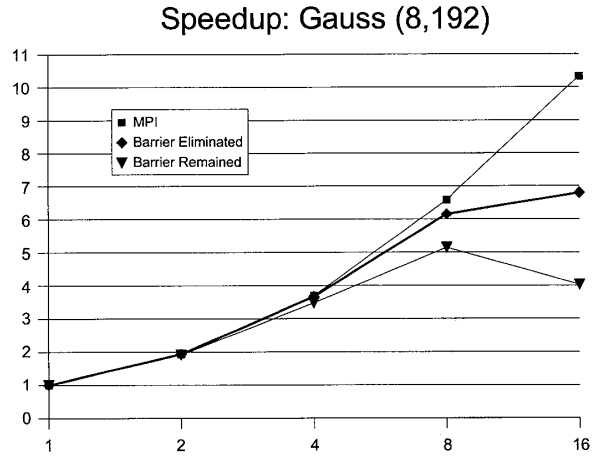


図 11 ガウスの消去法の台数効果 ($N = 8{,}192$)
Fig. 11 Speedup of Gaussian elimination ($N = 8{,}192$).

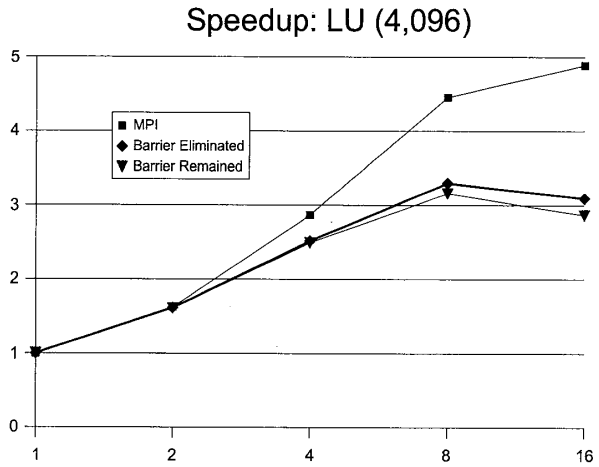


図 12 SPLASH-2 LU の台数効果 ($N = 4{,}096$)
Fig. 12 Speedup of SPLASH-2 LU ($N = 4{,}096$).

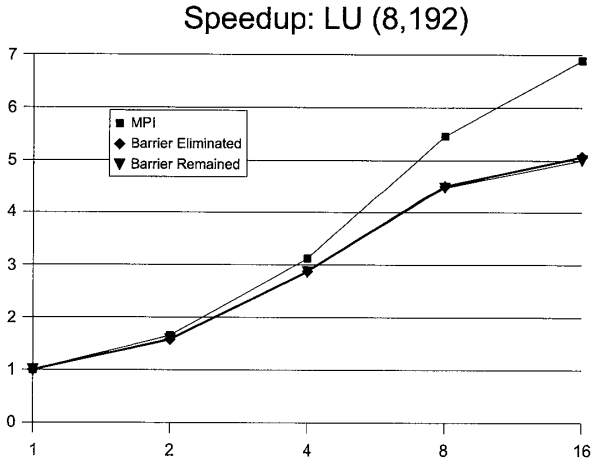


図 13 SPLASH-2 LU の台数効果 ($N = 8{,}192$)
Fig. 13 Speedup of SPLASH-2 LU ($N = 8{,}192$).

より優れ，MPI より劣る結果となった．

MPI との比較やバリヤ同期の除去の効果を論ずる前に，アプリケーションプログラムの負荷分散の状況を明らかにするため，本節では，バリヤ同期の除去前の結果について吟味する．

ヤコビの反復法では，$N = 8{,}192$ のとき，16 プロセッサで 14.19 倍の台数効果が得られ，SPLASH-2 LU では，性能向上に鈍化が見られるものの 16 台時に 8 台時よりも高い台数効果が得られた．一方，ガウスの消去法では，16 プロセッサでは，8 プロセッサよりも台数効果が悪化する結果となった．この現象の理由として，以下の 3 点が考えられる：(1) ヤコビの反復法は，反復ごとに各プロセッサが更新する $\mathbf{x}$ の大きさが不変であるため，負荷分散が均等であることから高い台数効果が得られた．(2) ガウスの消去法では，計算対象となる行列の範囲が，計算の進行とともに 1 行 1 列ずつ小さくなる．この挙動が，計算途中の行列の大きさがプロセッサ数で割り切れない状況を頻繁に発生させ，結果として負荷分散の不均衡を招き，台数効果を悪化させた．(3) ヤコビの反復法では反復回数を 1,000 回としたためバリヤ同期の回数も 1,000 回と少なかったのに対して，ガウスの消去法では，行列の大きさとほぼ同じ回数のバリヤ同期が必要であるため，バリヤ同期がオーバヘッド要因となって，台数効果の悪化をもたらした．(4) SPLASH-2 LU は，ガウスの消去法と同様に計算対象となる行列の大きさが小さくなり，負荷分散の不均衡が生じるプログラムであるものの，少ないバリヤ同期が通信起動回数を少なく抑えた結果，16 台時にオーバヘッドとして顕在化しなかったものと考えられる．

以下，本評価では，プログラム間の特性の違い，すなわち，(1) ヤコビの反復法は負荷分散が静的かつ均等であり，ガウスの消去法及び SPLASH-2 LU は負荷分散が動的に変化すること，(2) バリヤ同期の回数に関して，SPLASH-2 LU は比較的少なく，ガウスの消去法は多いことの 2 点に着目し，これらのアプリケーションプログラムにおける，我々の提案手法の効果を定量的に評価する．

4.3.2 バリヤ同期の除去数

先に表 2 で示したバリヤ同期のうち，バリヤ同期の除去アルゴリズムによって除去することができた数について調査した．

いずれのアプリケーションプログラムにおいても，読出しアクセス情報の巻上げが可能であり，quav_fetch() は生成されなかった．その結果，計算ループ中に出現するバリヤ同期を除去することができ，実行時においてバリヤ同期は一切実行されなかった．

4.3.3 実行時間の短縮率

本評価では，バリヤ同期の除去アルゴリズムを適用したプログラム，及び適用する前のプログラムの実行時間を測定し，実行時間の短縮率を求めた．ここで，実行時間の短縮率は，バリヤ同期の除去アルゴリズムの適用前，適用後の実行時間をそれぞれ T_b，T_e としたときに，$100 \times (1 - T_e/T_b)$ で求められる百分率として定義される．

図 14 から図 16 に，それぞれのアプリケーションプログラムの短縮率を示す．いずれのアプリケーションプログラムでも，(1) 2 プロセッサで実行した，SPLASH-2 LU ($N = 8{,}192$) を除き，短縮率が正，すなわち，バリヤ除去の効果が得られ，(2) プロセッサ数が増加

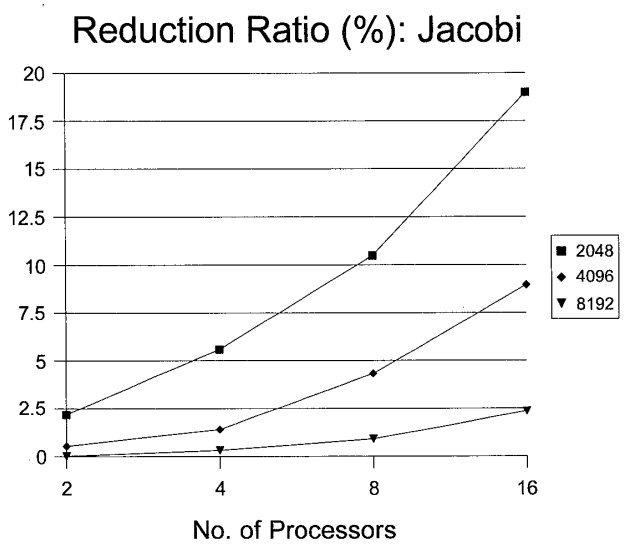


図 14　バリヤ除去手法による実行時間の短縮率：ヤコビの反復法

Fig. 14　Reduction ratio of execution time: Jacobi method.

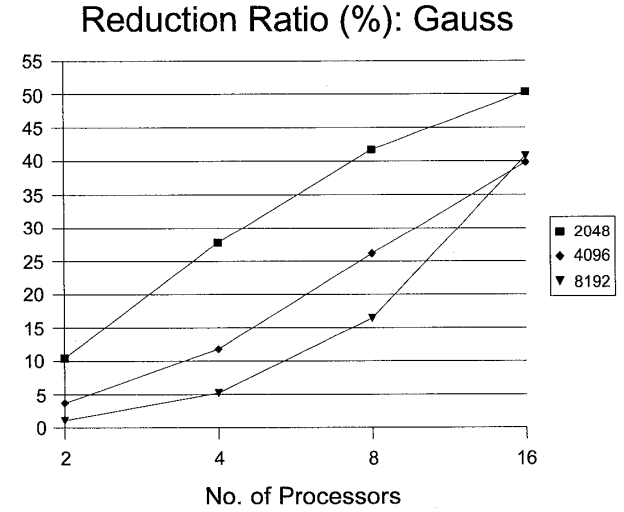


図 15　バリヤ除去手法による実行時間の短縮率：ガウスの消去法

Fig. 15　Reduction ratio of execution time: Gaussian elimination.

するにつれて，短縮率も増加する傾向が見られた．

ヤコビの反復法では，$N = 2{,}048$ のとき，16 台で 19.00%の短縮となったが，$N = 8{,}192$ では 2.36%にとどまった．しかしながら，この短縮により，$N = 8{,}192$ のときの 16 台での台数効果は，バリヤ同期の除去前に 14.19 倍だったのに対して，14.53 倍に向上した．

ガウスの消去法では，$N = 2{,}048$ のとき，16 台で 50.36%，$N = 8{,}192$ のとき 40.79%と高い短縮率が得られた．台数効果も $N = 8{,}192$ のときの台数効果は，バリヤ同期の除去前には 16 台で 4.03 倍と，8 台時の 5.14 倍から悪化していたのに対し，バリヤ同期の除去後

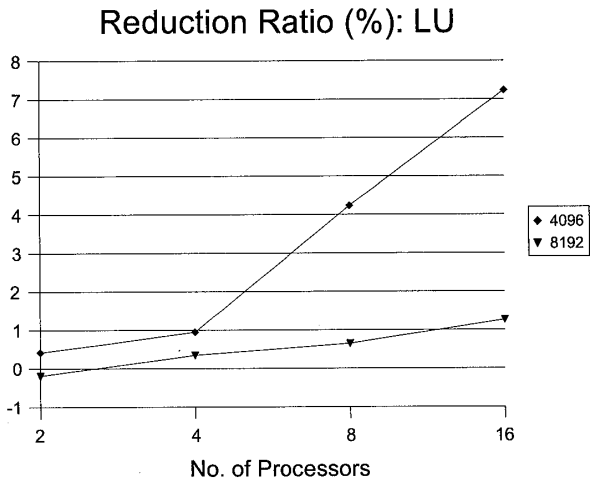


図 16　バリヤ除去手法による実行時間の短縮率：SPLASH-2 LU

Fig. 16　Reduction ratio of execution time: SPLASH-2 LU.

表 3　MPI との台数効果の比較（MPI を 100 とする．16 台，$N = 8{,}192$）

Table 3　Normalized speedup (MPI = 100, 16 processors, $N = 8{,}192$).

	除去前	除去後
ヤコビの反復法	99.56	97.22
ガウスの消去法	68.81	38.97
SPLASH-2 LU	73.60	72.67

には，8 台時の 6.15 倍から，16 台の 6.80 倍に向上した．

SPLASH-2 LU では，$N = 8{,}192$ のプログラムを 2 プロセッサで実行するとき，バリヤ同期の除去による効果が得られなかった．これは，(1) 2 プロセッサ時にはもともとバリヤ同期のコストが小さいことに加えて，(2) 他のアプリケーションプログラムと比較して，ループの繰返しごとに登録する quad 数が多いため，バリヤ同期の除去のために必要な quad 間演算がオーバヘッドとなったと考えられる．この点については，**4.3.5** で更に詳しく述べる．

いずれのアプリケーションプログラムでも，問題サイズが小さいほど，短縮率が大きい傾向が見られた．この理由として，バリヤ同期を除去する前のコードでは，問題サイズが小さいほど，あるバリヤ同期から次のバリヤ同期に到達するまでの時間が短く，相対的にバリヤ同期を実施するためのオーバヘッドが大きくなり，そのため，バリヤ同期を除去する効果が大きくなることが考えられる．

4.3.4　MPI との比較

16 プロセッサ，$N = 8{,}192$ の場合に，MPI の台数効果を 100 としたときの相対値を表 3 に示す．

ヤコビの反復法では，負荷が均等のため，アイドル時間が少なく，バリヤ同期の除去前の数値が 97.22 と MPI に迫る結果となった．

ガウスの消去法では，除去前の数値が 38.97 にとどまったが，除去後には 68.81 と改善している．ただし，MPI との比較においては大きな開きが認められる．この原因として，MPI では MPI_Bcast() を使用することによって，多台数時の通信オーバヘッドを軽減できたことが考えられる．

SPLASH-2 LU も同様に，集合通信の使用の有無によって，MPI との差が開いたものと考えられる．

4.3.5 バリヤ同期の除去に伴うオーバヘッド

バリヤ同期を除去したコードを実行する際には，quav_append_quad_for_next_phase() で登録された quad をもとに，quav_send_recv_if_needed() 内で他プロセッサから受け取るべきデータ総量を求めるため quad 間演算が必要となる．本項では，この quad 間演算にかかった時間を評価する．

$N = 8{,}192$，プロセッサ数を 16 とした場合に，プロセッサ 0 で必要となった上記の quad 間演算にかかった時間は，実行時間全体のうち，ヤコビの反復法で 0.02%，ガウスの消去法で 0.07%，SPLASH-2 LU で 0.14%と，いずれも小さく抑えることができた．アプリケーションプログラムでこの比率が異なる理由として以下の 2 点が挙げられる：(1) 実行時間当りの quad 間演算回数は，ヤコビの反復法と比較して，ガウスの消去法は 3.01 倍，SPLASH-2 LU は 6.63 倍多い．多くなる原因として，ガウスの消去法では，ループの繰返し数が多いこと，SPLASH-2 LU では，1 回当りに登録する quad の個数が多いことが考えられる，(2) プロセッサ間で依存関係にあるデータが，ヤコビの反復法では一次元配列であるのに対して，ガウスの消去法及び SPLASH-2 LU では二次元配列である．多次元配列のアクセス範囲を表現する場合，quad はそれぞれの次元に 1 個の quad を使用する [9] ため，多次元配列を使用するアプリケーションプログラムでは，quad を格納するためのメモリや quad 間演算に要するコストが大きくなる．

4.3.6 アイドル時間の減少効果

あるプロセッサが，あるフェーズにおける計算を終了し，次のフェーズに移行する際に，他プロセッサの計算の終了を待たなければならない場合，そのプロセッサはアイドル状態となる．ここでは，フェーズでの計算の終了時点から次のフェーズの開始時点までの経過時間を計測することによって，図 7 で示したようなアイドル時間の短縮を定量的に評価した．

図 17 から図 19 に，各アプリケーションプログラムの実行時に発生した，フェーズ間の移行にかかった

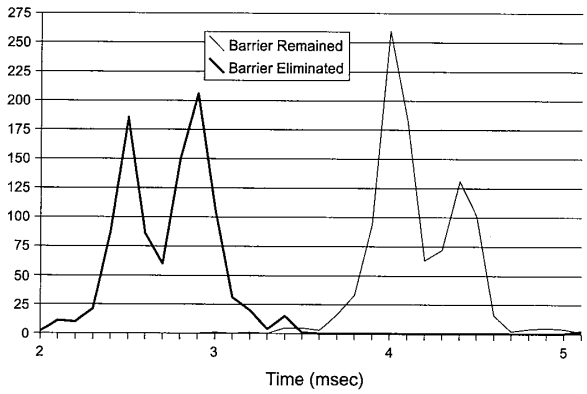


図 17 アイドル時間の度数分布：ヤコビの反復法 ($N = 8{,}192$)
Fig. 17 Frequency distribution of idle time: Jacobi method.

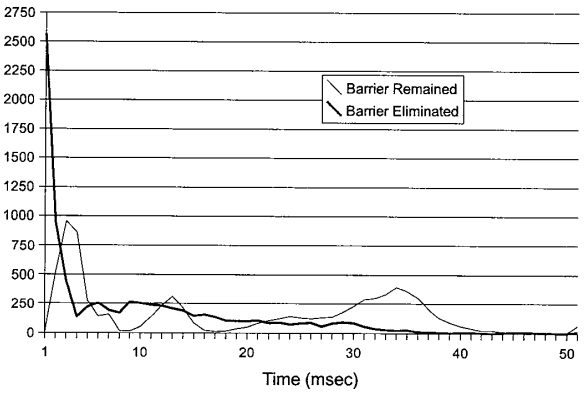


図 18 アイドル時間の度数分布：ガウスの消去法 ($N = 8{,}192$)
Fig. 18 Frequency distribution of idle time: Gaussian elimination.

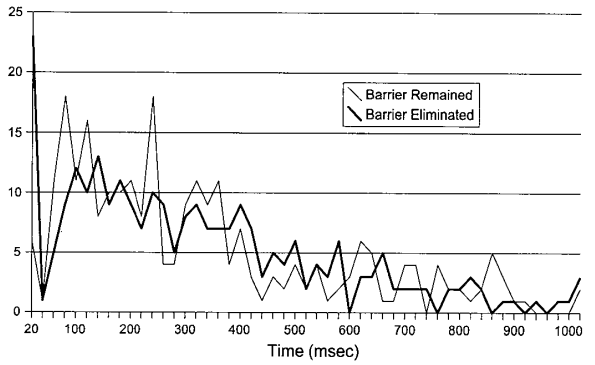


図 19 アイドル時間の度数分布：SPLASH-2 LU ($N = 8{,}192$)
Fig. 19 Frequency distribution of idle time: SPLASH-2 LU.

表 4 アイドル時間の平均と中央値（ミリ秒）
Table 4 Means and medians of the idle times.

	平均		中央値	
	除去前	除去後	除去前	除去後
ヤコビの反復法	4.31	3.20	4.03	2.74
ガウスの消去法	18.89	8.29	19.48	4.06
SPLASH-2 LU	320.89	314.67	239.71	257.24

時間，すなわちアイドル時間の度数分布を示す．また，表 4 にアイドル時間の平均と中央値を示す．いずれのアプリケーションプログラムにおいても，$N = 8{,}192$，プロセッサ数 16 のときに計測した．また，今回の評価では，いずれのアプリケーションプログラムでも，各プロセッサが同様の挙動を示したことから，プロセッサ 0 で得られた数値を使用した．具体的には，ガウスの消去法と SPLASH-2 LU では，負荷分散が動的に変化するが，問題をサイクリックに分割しているため，全プロセッサが同様の挙動を周期的に示すこととなった．一方，ヤコビの反復法では，負荷分散が均衡であるため，全プロセッサが同様の挙動を示した．

ヤコビの反復法では，バリヤ同期の除去アルゴリズムの適用に関係なく 5 ミリ秒以内にほぼすべてのフェーズ間移行が終了した．ただし，より詳細に見れば，適用前は 3.5 ミリ秒以内に終了した移行は全 1,000 回のうち 1.20%にとどまったのに対して，適用後には 99.80%となった．

ガウスの消去法における全 8,191 回のフェーズ間移行のうち，バリヤ同期のアルゴリズムの適用前では 1 ミリ秒以下で終了した移行は皆無であったのに対して，適用後は 2,568 回 (31.35%) であった．また，30 ミリ秒以下では適用前が累計で 5,503 回 (67.18%) であったのに対して，適用後は 7,895 回 (96.39%) であった．

SPLASH-2 LU では，図 19 に示すように，ほぼ拮抗した結果となったが，アイドル時間の総計は，バリヤ同期の除去前には 82.15 秒であったのに対し，除去後は 80.56 秒と 1.94%短縮した．

4.3.7 考　　察

以上の結果をまとめると，本評価により，(1) いずれのアプリケーションプログラムにおいても，行列の大きさに関係なく，ほぼすべての場合にバリヤ同期の除去の効果が得られたこと，(2) プロセッサ数が多いほど，バリヤ同期の除去の効果が高かったこと，(3) バリヤ同期の除去を実現するために必要な quad 間演算時間は，プロセッサ数が多い場合でも小さく抑えることができたこと，(4) アプリケーションプログラム間で，バリヤ同期の除去効果の傾向が大きく異なったことが明らかとなった．

最後に挙げた，アプリケーションプログラム間で，傾向が大きく異なる理由として，以下の 3 点が挙げられる．

- SPLASH-2 LU は，先に表 2 に示したように，バリヤ同期の回数が少ないため，バリヤ同期の除去効果が限定的であった．このことから，本手法は，バリヤ同期の回数が多いアプリケーションプログラムに対して，より効果的であると考えられる．
- ガウスの消去法は，他のアプリケーションプログラムと比較して，バリヤ同期の回数が多かったため，除去の効果が大きかったといえる．加えて，ガウスの消去法は，前述のとおり負荷分散が動的に変化するため，バリヤ同期を除去する前のプログラムでは，他プロセッサの同期点への到着を待つプロセッサと，他プロセッサを待たせるプロセッサが次々に入れ換わることになる．すなわち，あるバリヤ同期で待っていたプロセッサが，次のバリヤ同期では他プロセッサを待たせることになる．この状況が発生するプログラムで，バリヤ同期を除去した場合，計算が終了したプロセッサは，他プロセッサの計算の終了を待たずに，送るべきデータを他プロセッサに送ることができる．このような非同期的な 1 対 1 の送受信が適切に行われることにより，システム全体のアイドル時間が削減できたと考えられる．
- ヤコビの反復法は，負荷分散が均等であることから，ネットワーク遅延などの外的要因が生じない限りプロセッサのアイドル時間が生じにくい．このため，バリヤ同期を除去しても，大きな短縮率が得られなかったものと考えられる．ただし，ヤコビの反復法でも，バリヤ同期の除去アルゴリズムの適用により，ほぼすべてのアイドル時間を 3.5 ミリ秒以下に抑えることができた．これは，バリヤ同期を除去することによって，バリヤマスタを介さずに次のフェーズに移行できることが寄与したと考えられる．

5. 関 連 研 究

Tseng は，共有メモリ型マルチプロセッサで実行する SPMD プログラムでのバリヤ除去アルゴリズムを提案し，Stanford DASH [12] 上で性能を評価している [8]．また，バリヤ同期を，カウンタを用いた同期に置換するアルゴリズムも提案している．この手法では，データの生産プロセッサが，ある変数の値を変化させ

るまで消費プロセッサが待つことで，バリヤ同期よりも小さなオーバヘッドで，プロセッサ間の同期を実現している．この手法は，DASH が提供する共有メモリの存在を前提とする手法である，といえる．

Dwarkadas らは，ソフトウェア分散共有メモリ (Distributed Shared Memory：DSM) システムにおいて，バリヤ同期を，データの生産者から消費者にデータを送信する Push 関数に置換するコンパイラ手法を，他のコンパイラ手法とともに提案している [13]．Dwarkadas らの手法は，以下の点で我々の手法と異なる：

- Dwarkadas らの手法は，配列範囲記述子として Bounded Regular Section [14] を採用しているが，問題領域がブロックサイクリック分割される場合，並列処理に適した配列範囲記述子 quad [9] を採用している我々の提案手法と比べて，記述子の個数が多くなるため，実行性能が劣ると考えられる．
- Quaver では，データはいずれかのホームプロセッサに属しているため，同期点以降，読出しプロセッサは，書込み履歴を把握することなく，ホームプロセッサからのみデータを受け取ればよい．我々の提案手法では，この方式の利点を生かして，バリヤ同期の除去の可否を判断する際，必ずしも書込みアクセス情報を登録しない．この点で，読出し及び書込みアクセス情報を常に必要とする Dwarkadas らの手法に比べて単純であるといえる．
- Dwarkadas らが評価に用いた 6 個のアプリケーションプログラムのうち，バリヤ同期の置換手法を適用できたのは 2 個のみであり，そのいずれにおいても限定的な効果しか得られない，という結論に至っている．また，本論文での評価と異なり，プロセッサ数を変化させたときの評価をしていない．

2003 年以降に発表された，コンパイラによる解析結果を利用して実現されたソフトウェア DSM に関するいくつかの論文 [15], [16] においても，Tseng や Dwarkadas らの論文を関連研究として触れるにとどまっており，バリヤ除去機能の実装や詳細な評価はなされていない．

6. むすび

本論文では，共有メモリ型プログラムの実行時に，プロセッサのアイドル時間を増大させるバリヤ同期のコストに焦点を当て，プログラム中に出現するバリヤ同期を除去するアルゴリズムを提案した．本手法は，配列範囲記述子を用いてメモリアクセス範囲を特定し，プロセッサ間のデータ依存関係を解析し，データを生成するプロセッサからデータを消費するプロセッサに対してデータを転送するメッセージをもってプロセッサ間の同期を達成する点，すなわちデータの依存関係を利用してバリヤ同期を除去する点が，従来の手法と異なっている．

評価として，本手法を適用した，3 種類の行列演算プログラムを PC クラスタで実行し，本手法を適用しない場合と実行時間を比較した．その結果，プロセッサ数が 16 のとき，行列のサイズ N に対して，ヤコビの反復法で 2.36% ($N = 8{,}192$; 台数効果 14.53 倍) から 19.00% ($N = 2{,}048$; 台数効果 9.29 倍)，ガウスの消去法で 39.81% ($N = 4{,}096$; 台数効果 5.53 倍) から 50.36% ($N = 2{,}048$; 台数効果 4.07 倍)，SPLASH-2 LU で 1.26% ($N = 8{,}192$; 台数効果 5.07 倍) から 7.23% ($N = 4{,}096$; 台数効果 3.09 倍) の短縮率が得られた．アプリケーションプログラムの特性により，大小はあるものの，ほぼすべての場合にバリヤ同期の除去によって実行時間が短縮した．また，プロセッサ数が多いほどバリヤ同期の除去の効果が高いこと，バリヤ同期の除去を実現するために必要なオーバヘッドは，除去によって達成される時間短縮と比較して小さく抑えられることが明らかになった．プロセッサ数が 2 のとき，短縮率が負になった場合があったが，多プロセッサ数で高い効果が得られるため，本手法の瑕疵にはならないと考える．

以上のことから，本手法は，プロセッサ数の増加につれて増加するバリヤ同期のコストの問題を軽減することに寄与できる，と結論づけられる．

本論文の評価を通して得られた，行列演算プログラムに本手法を適用したときの有効な結果を受け，今後の課題として，本手法を，多様なアプリケーションプログラムに適用し，より多台数の PC から構成される PC クラスタ上で評価実験を行うことが考えられる．本論文で示した結果から，より多くのプロセッサ数を用いた場合でも，更に実行時間が減少することが予想される．具体的には，(1) 一般に，並列処理では，プロセッサ数が増えるに伴い，それぞれのプロセッサが担当するアプリケーション演算の量は小さくなる．このとき，プロセッサ数の増加がバリヤ同期の絶対時間を増加させると同時に，アプリケーション演算時間に対するバリヤ同期の相対的な時間をも増加させる傾向がある．(2) 一方，本論文での実験結果から問題サイズが小さいときに，短縮率が良い傾向が観察された．こ

れらの2点から，本手法は，プロセッサ数が多くなるほど，高い短縮率が得られることが予想される．そのため，32台ないし1,024台程度のPCから構成されるPCクラスタを用いても，本論文で提案するバリヤ除去手法が有効であるか検証したいと考えている．また，メモリアクセスのパターンやタイミングが本論文の評価で用いたアプリケーションプログラムとは異なるプログラム，あるいはコンパイラによる静的解析が困難なプログラムを用いた性能評価を通して，本手法が有効に働く問題範囲をより明確する必要があると考える．

文　献

[1] OpenMP Architecture Review Board, "OpenMP application program interface," version 2.5 ed., 2005.

[2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," Computer, vol.29, no.2, pp.18–28, 1996.

[3] H. Lu, OpenMP on Networks of Workstations, PhD Thesis, Rice University, 2001.

[4] S.-J. Min, A. Basumallik, and R. Eigenmann, "Supporting realistic OpenMP applications on a commodity cluster of workstations," Proc. WOMPAT 2003, pp.170–179, 2003.

[5] L. Huang, B. Chapman, Z. Liu, and R. Kendall, "Efficient translation of OpenMP to distributed memory," Proc. ICCS 2004, pp.408–413, 2004.

[6] R. Eigenmann, J. Hoeflinger, R.H. Kuhn, D.A. Padua, A. Basumallik, S.-J. Min, and J. Zhu, "Is OpenMP for grids?," Proc. 16th International Parallel and Distributed Processing Symposium, 2002.

[7] N. Yonezawa, K. Wada, and T. Ogura, "*Quaver*: OpenMP compiler for clusters based on array section descriptor," Proc. IASTED International Conference on Parallel and Distributed Computing and Networks, pp.234–239, 2005.

[8] C.-W. Tseng, "Compiler optimizations for eliminating barrier synchronization," Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, 1995.

[9] 米澤直記，和田耕一，"並列処理に適した配列範囲記述子quadの提案と評価，" 情処学論，vol.46, no.5, pp.1274–1286, 2005.

[10] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," Proc. 22nd Annual International Symposium on Computer Architecture, pp.24–37, 1995.

[11] S.C. Woo, J.P. Singh, and J.L. Hennessy, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors," ACM SIGPLAN Notices, vol.29, no.11, pp.219–229, 1994.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford DASH multiprocessor," Computer, vol.25, no.3, pp.63–79, 1992.

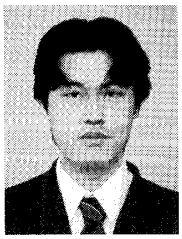
[13] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, "An integrated compile-time/run-time software distributed shared memory system," ASPLOS, pp.186–197, 1996.

[14] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," IEEE Trans. Parallel Distrib. Syst., vol.2, no.3, pp.350–360, 1991.

[15] S.-J. Min, A. Basumallik, and R. Eigenmann, "Optimizing openMP programs on software distributed shared memory systems," Int. J. Parallel Program., vol.31, no.3, pp.225–249, 2003.

[16] N.P. Manoj, K.V. Manjunath, and R. Govindarajan, "CAS-DSM: A compiler assisted software distributed shared memory," Int. J. Parallel Program., vol.32, no.2, pp.77–122, 2004.

（平成19年6月1日受付，10月29日再受付）



米澤　直記

平11筑波大学大学院博士課程工学研究科中途退学．同年筑波大学電子・情報工学系助手を経て，平15より神奈川大学理学部情報科学科特別助手．分散共有メモリシステム，並列プログラミング環境に関する研究に従事．IEEE，ACM，情報処理学会各会員．

和田　耕一　（正員）

昭59神戸大学大学院自然科学研究科博士課程了．同年，同大大学院自然科学研究科助手．昭62筑波大学電子・情報工学系講師，助教授を経て平11教授，現在に至る．平4～5カナダ，ビクトリア大学客員研究員．並列分散処理とアーキテクチャ，並列シミュレーション，マルチメディア情報処理に関する研究に従事．学術博．IEEE，ACM各会員．