分散共有メモリシステム上にソフトウェアによって構築されたキャッシュシステムの静的制御

南里豪志[†] 佐藤周行[††] 島崎眞昭[†,☆]

本研究では，分散共有メモリシステム上にソフトウェアによって構築されたキャッシュメモリシステムについて，キャッシュ制御命令をアプリケーションコードに挿入するプリプロセッサによるプログラム最適化の効果を調べる．本研究で利用するキャッシュメモリシステムは，並列C言語Split-Cのライブラリにより，各プロセッサのローカルメモリ上に構築されたものであり，ハードウェアの支援をまったく必要としない．これに対してプリプロセッサは，アプリケーションのデータ依存関係を解析し，その結果を利用してキャッシュ制御命令をアプリケーションコードに挿入する．また，ソフトウェアによって構築したキャッシュメモリと大域メモリで構成されるメモリ階層に対してRISC計算機におけるキャッシュメモリを考慮したプログラミング技術を適用することにより，ソフトウェアによるキャッシュメモリシステムの効率向上を図る．本稿では，このようなシステムを分散メモリ型並列計算機CM-5上に構築し，計測を行った．実験には，3種の基本的なアプリケーションを利用し，ソフトウェアによるキャッシュの効果，プリプロセッサによる効果，およびキャッシュメモリを考慮したプログラミング技術による効果について計測し評価した．その結果，16プロセッサ上の並列行列積において，ソフトウェアによるキャッシュ，プリプロセッサおよびキャッシュメモリを考慮したプログラミング技術を利用することにより，1024 × 1024 の行列における台数効果が13.7となった．

Effects of Static Management of Software-Implemented Cache Memory System on Distributed Shared Memory System

TAKESHI NANRI,[†] HIROYUKI SATO[††] and MASAAKI SHIMASAKI[†,☆]

We propose a pre-processor approach in which a pre-processor system inserts codes for managing software-implemented cache memory on distributed shared memory system into application programs. In our work, we used a software-implemented cache memory system implemented on a part of local memory on each processor, by using runtime library of parallel C language Split-C. In order to manage the software-implemented cache memory system, our pre-processor inserts codes which explicitly handle the cache system by using the information from the dependence analysis of the application program. Therefore the pre-processor decides optimal behabior of the software-implemented cache memory system. Furthermore, we examine the effectiveness of the array-blocking technique as a typical example of RISC-oriented optimizing programming techniques for cache memory. This technique is performed to increase the reuse of data which is copied to the software-implemented cache memory. In this paper we used the software-implemented cache memory system on a distributed memory parallel computer CM-5 for experiments. In the experiments, we evaluated the effect of software-implemented cache memory system, that of pre-processor and that of the array-blocking technique. As a result, with 16 processors and software-implemented cache memory system, the speed up ratio of array-blocked parallel matrix multiplication to serial is about 13.7 by using the pre-processor.

† 九州大学大型計算機センター
Computer Center, Kyushu University
†† 九州大学大学院システム情報科学研究科
Department of Computer Science and Communications Engineering, Kyushu University
☆ 現在，京都大学大学院工学研究科電気工学専攻
Presently with Department of Electrical Engineering, Kyoto University

1. 背　景

分散共有メモリシステムは，並列処理に関する近年の研究の中で最も注目を集めているものの1つである．このようなシステムは，分散メモリ型並列計算機の各プロセッサに大域メモリが分散されているため，大域メモリへのアクセスコストが高いことが問題となる．

この大域メモリへのアクセスコストを低減するため，近年提案されているほとんどの分散共有メモリシステムはキャッシュメモリシステムを用意している．分散共有メモリシステムは，プログラミングモデルとしては各プロセッサに共有された1つのメモリを提供するが，このようなキャッシュメモリシステムを利用することにより，実行モデルとしては，遅い大域メモリと速いキャッシュメモリによるメモリ階層を提供する．

分散共有メモリシステム上のキャッシュメモリシステムには，ハードウェアによるサポートを利用して構築されたものと，完全にソフトウェアのみで構築されたものがある．一般にハードウェアによるサポートを利用するキャッシュメモリシステムはキャッシュの管理に要するコストが低いため，ソフトウェアによって構築されたキャッシュメモリシステムと比較すると効率が良いが，実装はハードウェアに依存しており，異なるプラットフォーム上に実装することは困難である．これに対してソフトウェアによって構築されたキャッシュメモリシステムは，ハードウェアに依存しないものが多いため，様々なプラットフォーム上に構築できる．さらにキャッシュの制御をソフトウェア的に行えるため，アプリケーションに適したキャッシュ動作を選択することにより，キャッシュの管理に要するコストを低減することができる．そこで本研究では，分散共有メモリシステム上にソフトウェアによって構築したキャッシュメモリについて，アプリケーションに適したキャッシュ動作を選択するプリプロセッサによる効果を計測する．

本研究で利用するキャッシュメモリシステムは，分散メモリ型並列計算機上に構築された共有メモリ型並列C言語Split-C[8]のコンパイラおよびライブラリを利用して，各プロセッサのローカルメモリの一部に構築されている[19]．このようなキャッシュメモリシステムに対し本稿で述べるプリプロセッサは，必要なキャッシュ制御命令をアプリケーションコードに挿入することにより，制御を行う．この必要なキャッシュ制御命令は，アプリケーションに対してデータ依存解析および制御依存解析（以後，依存解析）を行うことにより挿入される．また，依存解析を行えない場合は実行時にランタイムライブラリが必要に応じてキャッシュ制御を行う．このようにしてアプリケーションの性質に適したキャッシュ制御を行う．

本研究のシステムは，依存解析の可否にかかわらずプログラマに対して，ソフトウェアによって構築したキャッシュメモリと大域メモリで構成される階層構造を提供する．この階層構造はRISC計算機におけるハードウェアキャッシュとメインメモリによる階層構造と同様のメモリモデルをプログラマに提供するため，RISC計算機に対するキャッシュ最適化技術が有効となる．そこで本研究では，RISC計算機における最も一般的なキャッシュ最適化技術の1つであるブロック化を適用したコードについて処理時間を計測し，ブロック化を適用しないコードの処理時間と比較することにより，本システムにおけるブロック化の有効性を示す．

本稿の構成は以下のとおりである．

まず2章で関連研究を紹介する．次に3章で本研究で利用するキャッシュシステムについて説明し，4章でキャッシュ制御命令を挿入するプリプロセッサの説明をする．また，5章でRISC計算機のキャッシュ最適化技術を紹介する．6章では本研究のシステムについて性能評価を行い，7章で結果を考察する．

2. 関連研究

分散共有メモリシステムにおけるプロセッサ間通信の最適化については数多くの研究がなされている[1),3),5),9)]．それらのほとんどは

- 通信の集合化による遅延の削減
- 冗長通信の除去
- 計算と通信のオーバラップ

のいずれかに分類される．これらの最適化は依存解析を十分に行える場合有効であるが，依存解析を行えない不規則なアクセスパターンを持つプログラムに適用することができない．

これに対し，インデックス配列による間接アクセスのため，アクセスパターンが不規則となる並列ループにおける通信の最適化技術として，実行時にアクセスパターンを解析し，その結果を利用して通信の最適化を行うinspector/executor法がある[13),14)]．

また，より複雑なアクセスパターンを持つプログラムに対しては，各プロセッサに用意したキャッシュを介して大域メモリにアクセスするシステムが利用される[16),20),22)]．分散共有メモリシステムDash[15)]やAlewife[7)]では，キャッシュにコピーされたラインの情報を“directory”と呼ばれる構造体により各プロセッサに分散して格納することによりキャッシュ制御を行う．さらにDashでは，各プロセッサでキャッシュメモリの一貫性を保つプロトコルとして，制限の緩いrelease consistencyプロトコルを提案している．Munin[6)]では，このrelease consistencyプロトコルに加え，multiple writerプロトコルによって複数のプロセッサが同時に同じブロックに書き込むことを可能にしている．これに対して分散共有メモリシステムTreadMarks[2)]

では，multiple writerプロトコルとさらに制限の緩いlazy release consistencyプロトコルを利用して，一貫性保持に要するコストの低減を図っている．また，中條ら[17]は，ワークステーションクラスタ上に構築した分散共有メモリシステムで，無効化に巡回型マルチキャストメッセージを用いるソフトウェア制御のコヒーレントキャッシュを実装している．これらのシステムは一部または全部がソフトウェアによって構築されているが，キャッシュへコピーするタイミングやコピーを行う単位となるサイズなどについてコンパイル時の情報を利用した最適化を行わない．これに対して本研究で利用したソフトウェアによるキャッシュシステムは，非常に簡潔で構築が容易であるうえ，静的に解析を行うことができるアプリケーションについては，コンパイル時の情報を利用して効率良く制御するよう最適化を行う．また，より複雑なシステムで採用されているプロトコルを本研究のシステムに利用することによってさらに効率の向上を図ることができる．ただしこれには文献4)，12)のような分散共有メモリシステムにおけるキャッシュメモリシステムのモデル化に関する研究が必要である．

また，ソフトウェアによるキャッシュの制御については，ソフトウェアによって制御できるキャッシュアーキテクチャを構築し，ソフトウェアとハードウェア双方から性能向上のための情報を提供するという研究がなされている[11]．

3. ソフトウェアによるキャッシュメモリシステム

3.1 ソフトウェアによるキャッシュメモリシステムの構造

本研究では，SPMD型並列C言語Split-C[8]のコンパイラ，およびランタイムルーチンを利用し，ソフトウェアのみによって構築されたキャッシュメモリシステムを利用する．このキャッシュメモリシステムは，**図1**のように，各プロセッサのローカルメモリの一部を用いて構築される．また，大域メモリからキャッシュメモリへのコピーはSplit-Cの大量通信を用いて実現される．

このように，分散メモリ型並列計算機のレベルで見た場合，大域メモリからキャッシュメモリへのコピー動作は，通信を集合化し，ローカルバッファへ大量通信することと同意である．分散メモリ型並列計算機におけるこのような通信の最適化については従来より研究されている[1]．本研究はこのような最適化とソフトウェアによるキャッシュメモリを統合することにより，

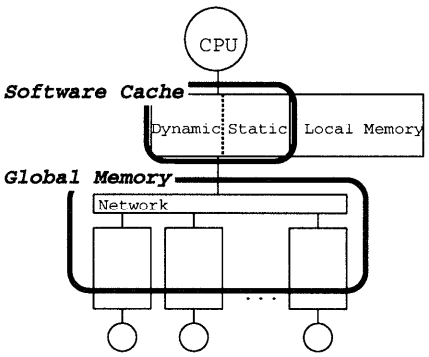


図1 ソフトウェアによるキャッシュメモリシステム
Fig. 1 Software implemented cache memory system.

幅広いアプリケーションに対して高速化を図る．

3.2 ランタイムライブラリによるキャッシュメモリシステムの制御

本研究で利用するソフトウェアによるキャッシュメモリシステムでは，ランタイムライブラリによって，非常に簡単なキャッシュ制御が行われる．このキャッシュ制御に必要な情報は分散ディレクトリ方式[15]で管理される．また，プロセッサ間のキャッシュのコンシステンシプロトコルとしてはwrite backとwrite invalidateが用意され，プログラマはコンパイル時にマクロ変数を用いてアプリケーションに適したプロトコルを選択できる．

本研究のキャッシュへのコピーはSplit-Cのbulk_read命令を利用して行われる．これは他プロセッサの連続領域をローカルメモリに一括受信する通信命令で，受信が終了するまで次の処理に移らない．Split-Cには非同期に通信を行う命令も用意されているが，転送終了の時期が不確定であるため，本研究のキャッシュへのコピー手段としては利用できない．

4. プリプロセッサ

本研究で提案するプリプロセッサは，依存解析を行うことができるアプリケーションについて，コード中にキャッシュの制御命令を挿入する．このキャッシュ制御命令は，分散メモリ型並列計算機における通信の集合化と同様の方法で挿入される[18),21]．

たとえば，**図2**のようなSplit-Cによる並列LU分解プログラムに対してプリプロセッサを用いる．このプログラムに対して依存解析を行うと，ループjの各イタレーションで大域配列Aの領域のうち，プロセッサ$i\%PROCS$が所有する$A[i\%PROCS]$の$[i/PROCS][i]$から$[i/PROCS][N-1]$までの連続した領域を，各プロセッサが利用することが分かる．

```
double A[PROCS]::[N/PROCS][N];
for (i=0; i<N; i++)
  for (j=(i+1)+(i+1-MYPROC+PROCS)%PROCS;
       j<N; j+=PROCS)
    A[MYPROC][j/PROCS][i] /=
      A[i%PROCS][i/PROCS][i];
    for (k=i+1; k<N; k++)
      A[MYPROC][j/PROCS][k] -=
        A[MYPROC][j/PROCS][i] *
        A[i%PROCS][i/PROCS][k];
```

図2 並列LU分解（ソフトウェアによるキャッシュなし）
Fig. 2 Parallel LU decomposition (without software-implemented cache).

```
double StaticCache[N];
...
bulk_read(StaticCache,
          A[i%PROCS][i/PROCS][i],
          (N-i)*sizeof(double));
A[MYPROC][j/PROCS][i] /= StaticCache[0];
for (k=i+1; k<N; k++)
  A[MYPROC][j/PROCS][k] -=
    A[MYPROC][j/PROCS][i] *
    StaticCache[k-i];
```

図3 並列LU分解（プリプロセッサ）
Fig. 3 Parallel LU decomposition (with pre-processor).

そこでプリプロセッサは，**図3**のように，ループ j の最初にキャッシュへコピーを行い，その後の大域配列Aへのアクセスをソフトウェアによるキャッシュメモリへのアクセスに変換する．

また，静的依存解析の結果必要とされるソフトウェアによるキャッシュメモリの大きさが大きすぎる場合，キャッシュメモリを利用する範囲を分割することによって必要なキャッシュメモリの大きさを減少させなければならない．これは，**図4**のように，キャッシュ制御命令の内側のループをキャッシュサイズごとに実行することによって実現できる．

ソフトウェアによるキャッシュメモリシステムでは，ハードウェアのサポートを利用するキャッシュシステムと比較するとキャッシュリストの管理やコヒーレンス保持に要するコストが高い．これに対してプリプロセッサによるキャッシュ制御命令は依存解析の結果を利用して挿入されるため，参照される大域データのコピーが存在するキャッシュのアドレスに直接アクセスすることが可能である．また，データの生成と利用に関する情報を利用することにより，正しい値をキャッシュメモリにコピーできる．そのため，非常に低コストでキャッシュメモリシステムを管理できる．

5. RISC計算機のキャッシュ最適化技術

ソフトウェアによるキャッシュメモリシステムと大域メモリで構成されるメモリ階層は，RISC計算機におけるキャッシュメモリとメインメモリで構成されるメモリ階層と同じ構造である．

```
...
bulk_read(StaticCache,
          A[i%PROCS][i/PROCS][i],
          CacheSize);
A[MYPROC][j/PROCS][i] /= StaticCache[0];
for (k=i+1; k<i+CacheSize; k++)
  A[MYPROC][j/PROCS][k] -=
    A[MYPROC][j/PROCS][i] *
    StaticCache[k-i];
for (out_k=i+CacheSize+1; out_k<N;
     out_k+=CacheSize){
  bulk_read(StaticCache,
            A[i%PROCS][i/PROCS][out_k],
            CacheSize);
  for (k=out_k; k<min(N, out_k+CacheSize);
       k++)
    A[MYPROC][j/PROCS][k] -=
      A[MYPROC][j/PROCS][i] *
      StaticCache[k-out_k];
```

図4 CacheSizeでループ k を分割
Fig. 4 Distribute loop k with CacheSize.

一般にメモリ階層では，より高速なメモリ上にデータをコピーすることによってメモリアクセス速度の向上を図るが，データを低速なメモリからコピーするコストが高いため，メモリ階層の効率を向上させるためにはコピー回数を減らす技術が必要である．そのため，RISC計算機における最適化技術のほとんどは，キャッシュメモリにコピーしたデータを再利用することを目的として，アプリケーションコードを変形することに主眼を置いている．そこで本研究ではこのような技術を並列アプリケーションプログラムに適用することにより，分散共有メモリシステム上にメモリ階層を構築した場合のグローバルメモリへのアクセス性能向上を図る．

ところで分散共有メモリシステムにおけるソフトウェアによるキャッシュメモリシステムを分散メモリ型並列計算機のレベルで見ると，参照の局所性を利用した通信の集合化と同じ効果を持つ．また，キャッシュにコピーされたデータを再利用することにより冗長な通信を削除できる．すなわち，分散メモリ型並列計算機におけるこれらの最適化はRISC計算機のメモリ階層モデルに帰着することができる．これにより，アクセスパターンの複雑なアプリケーションにおいても，通信の最適化を容易に行うことができる．ただし，これらの通信最適化技術と並んで頻繁に利用される最適化技術の1つである通信と計算のオーバラップは，プロセッサ間のキャッシュコンシステンシを保持することが困難になるため，本研究のシステムでは利用し

ない．

本稿では，キャッシュメモリ最適化技術としてブロック化を利用する．ブロック化は，プログラム中のループをキャッシュラインのサイズに合わせてブロック分割し，さらに同じブロックを連続して利用するようにループを変形する最適化技術である．

6. 評　　価

表1 に示すような環境でソフトウェアによるキャッシュシステムを構築して，評価を行った．今回の実験には九州大学大型計算機センターのCM-5を利用したが，各プロセッサに用意されているベクトルユニットは利用していない．プログラムはSplit-Cで作成した．

ここで，静的制御されるソフトウェアによるキャッシュとは，プリプロセッサが挿入するキャッシュ制御命令によって制御されるキャッシュであり，依存解析を行えないプログラムでは配置されない．これに対し，動的制御されるソフトウェアによるキャッシュとは実行時にランタイムライブラリによって制御されるキャッシュであり，本システムにつねに実装されている．

これらのソフトウェアによるキャッシュメモリは，各プロセッサが持つハードウェアによるキャッシュメモリと関係なく，キャッシュサイズやラインサイズを選択できる．ただしこのようなサイズの変更はシステムの再構築時に行う．

6.1 ソフトウェアによるキャッシュメモリシステムのアクセス性能

分散共有メモリシステム上に構築したソフトウェアによるキャッシュメモリシステムのメモリアクセス性能は**表2** のとおりである．

まず，大域メモリからの読み出しを1台のプロセッサのみが行う場合と，16台のプロセッサが同時に行う場合でメモリアクセスのオーバヘッドを計測した．1台のプロセッサのみが読み出しを行う場合，プロセッサ間ネットワークをその1台が占有でき，ネットワークにおける衝突は発生しない．一方，16台のプロセッサが同時に読み出しを行う場合，本実験では負荷を分散するため各プロセッサが自分の隣のプロセッサの所有するデータを読み出すようなプログラムを利用したが，プロセッサ間ネットワークが全プロセッサで共有されるのでネットワークにおける衝突が発生する．このため，表2に示すように衝突が発生しない場合に比べて衝突が発生する場合の方がメモリアクセスのオーバヘッドが大きい．

また，キャッシュヒット時とキャッシュを利用しない場合のメモリアクセスオーバヘッドの比は $R_H : R_G = 1 : 4$ となっている．これは，一般的なRISCの逐次計算機でハードウェアによるキャッシュを用いた場合の比率，$R_H : R_G = 1 : 8 \sim 1 : 16$ 程度[10]に比べ多少劣っているが，キャッシュシステムとして十分利用価値のある比率である．

表1 計測環境
Table 1 Evaluation environment.

並列計算機	CM-5
プロセッサ数	16
メモリ	32 MByte/Proc
静的制御されるソフトウェアによるキャッシュのサイズ	512 KByte/Proc
動的制御されるソフトウェアによるキャッシュのサイズ	512 KByte/Proc
ラインサイズ	128 Byte

表2 メモリアクセス基本性能
Table 2 Performance of memory access.

readアクセスオーバヘッド（μsec）

衝突	R_L	R_G	R_H	R_M
無	1.2	16.2	4.3	287.0
有		18.7	5.1	370.0

R_L：ローカルデータのread
R_G：大域データのread
R_H：ソフトウェアによるキャッシュのヒット時のread
R_M：ソフトウェアによるキャッシュのミス時のread

writeアクセスおよびコヒーレンスプロトコルのオーバヘッド（μsec）

W_L	W_G	C_{Coh}
0.6	1.8	0.6

W_L：ローカルデータのwrite
W_G：大域データのwrite
C_{Coh}：ソフトウェアによるキャッシュのコヒーレンスプロトコルのコスト

アクセスオーバヘッド比率

$R_H : R_G$	$W_L : W_G$
1:4	1:3

6.2 並列アプリケーションにおける性能評価

本研究ではベンチマークとしてMM（行列積），CH（コレスキー分解），GE（ピボット選択を行うガウス消去法）という3個の並列アプリケーションプログラムを利用した．また，それぞれのアプリケーションについて，明示的にブロック化を施したプログラムを作成し，ブロック化を施さないプログラムと比較した．それぞれの特徴は次のようになる．

MM MMでは各行列は行方向にblock分割され，owner computes ruleによって並列に計算される．MMは静的依存解析が可能であるためプリプロセッサによってキャッシュ制御命令が挿入される．さらに，ループによる依存関係がないため，計算

を行う前に必要なデータをすべて一括してキャッシュにコピーすることにより，計算中のキャッシュミスをなくすことができる．しかし，キャッシュサイズに制限があるため，プリプロセッサはループをキャッシュサイズで分割する．ここですべてのプロセッサが同じ領域にアクセスすると，その領域を所有するプロセッサに負荷が集中し，性能が低下する．これは，各プロセッサが別の領域にアクセスするような変形によって回避することができる．そこでMMにブロック化を施した後，各プロセッサが別のブロックにアクセスするような変形を行い，実験に利用した．

CH CHでは行列は行方向にcyclic分割され，owner computes ruleによって並列に計算される．CHも静的依存解析が可能であるためプリプロセッサによってキャッシュ制御命令が挿入される．しかし，CHでは外側ループによる依存関係があるため，プリプロセッサはそのループの内側のループで必要となるデータを集め，外側のループの各イタレーションの最初に一括してキャッシュにコピーする．

この並列プログラムに対するブロック化は，**図5**のように列方向のブロック単位で計算を行うようにループを変形することによって行う．この変形により，ブロック化されたCHは以下のようなアルゴリズムで計算が進行する．

(1) block1を分解
(2) block1を利用して右側の領域の値を計算
(3) block2を分解
(4) block2を利用して右側の領域の値を計算
...

GE GEでも行列は行方向にcyclic分割され，owner computes ruleによって並列に計算される．本研究で用いたGEのプログラムでは，配列を間接的にアクセスするため静的な依存解析は不可能である．そのため，動的制御されるソフトウェアによるキャッシュシステムが利用される．

この並列プログラムに対するブロック化もCHと同様に，**図6**のように列方向のブロック単位で計算を行うようにループを変形する．

各アプリケーションについてそれぞれ以下のような処理時間を計測する．

Seri：シリアル

Para：並列（ソフトウェアによるキャッシュメモリシステムなし）

Dyna：並列（動的制御されるソフトウェアキャッシュメモリシステムを利用）

Stat：並列（プリプロセッサを利用）

Bloc：並列（ブロック化を適用したアプリケーションを利用）

これらを用いて各アプリケーションに対する本研究のシステムの性能を評価する．

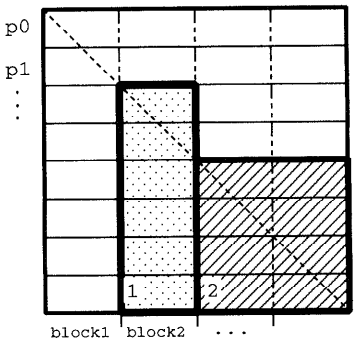


図5 ブロック化されたコレスキー分解
Fig. 5 Blocked Cholesky Decomposition.

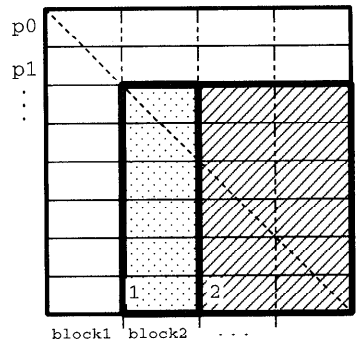


図6 ブロック化されたガウス消去法
Fig. 6 Blocked Gaussian Elimination.

6.2.1 行 列 積

表3に16プロセッサでの結果を示す．ソフトウェアによるキャッシュメモリシステムを用いない場合（*Para*），1024×1024の行列における処理時間は*Seri*の約2倍である．これは，ソフトウェアによるキャッシュメモリシステムを用いない行列積では，プロセッサ間通信が$O(n^3)$の頻度で発生するためである．また，ソフトウェアによるキャッシュメモリシステムを利用した場合の台数効果*Seri*/*Dyna*も1.47であった．これに対してプリプロセッサを用いると台数効果*Seri*/*Stat*は7.46に向上した．

また，ブロック化を施したプログラムの処理時間（$Bloc + Stat$）から計算した台数効果（$\frac{Seri}{Bloc+Stat}$）は13.7となった．**図7**に1024×1024の行列における

表3 行列積の実行時間（16 processors）
Table 3 Execution time of matrix multiplication (16 processors).

Size	*Seri*	*Para*	*Dyna*	*Stat*	*Bloc + Stat*
256	15.1	36.0	2.44	1.51	1.19
512	127.5	290.4	55.4	11.3	9.34
768	408.8	1012.5	236.8	39.1	34.4
1024	1091.4	2339.5	739.7	146.2	92.1

(sec.)

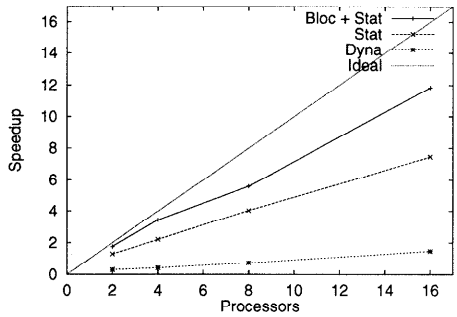


図7 行列積の台数効果（1024 × 1024）
Fig. 7 Speedup of matrix multiplication (1024 × 1024).

表4 行列積の処理速度（16 processors）
Table 4 Execution time of matrix multiplication (16 processors).

Size	*Seri*	*Para*	*Dyna*	*Stat*	*Bloc + Stat*
256	2.22	0.93	13.8	22.4	28.2
512	2.10	0.92	4.85	23.8	28.7
768	2.22	0.89	3.82	23.2	26.3
1024	1.96	0.92	2.90	14.7	23.3

(MFlops)

台数効果を示す．

行列積の計算量は $2N^3$ で近似できる．そこで，行列積を用いた場合の本システムの処理速度 $= \frac{2N^3}{elapsedtime}$ を**表4**に示す．*Seri*, *Para* と *Dyna*, *Stat*, *Bloc+Stat* を比較することにより，キャッシュを用いない場合の処理速度は行列サイズに依存しないが，キャッシュを用いる場合の処理速度は行列サイズが大きくなると低下することが分かる．これは，行列サイズがキャッシュサイズに対して大きくなり，キャッシュミスが頻繁に起こるためである．

6.2.2 Cholesky 分解

表5に16プロセッサでのCHの計測結果を示す．

行列サイズが1024の場合，ソフトウェアによるキャッシュメモリシステムを用いたときの台数効果は0.94，プリプロセッサを用いたときの台数効果は2.5であった．これに対してブロック化を適用することにより，台数効果は5.33に向上した．

図8に1024×1024の行列における台数効果を示す．

表5 コレスキー分解の実行時間（16 processors）
Table 5 Execution time of Cholesky Decomposition (16 processors).

Size	*Seri*	*Para*	*Dyna*	*Stat*	*Bloc + Stat*
256	2.4	11.9	3.42	1.59	1.30
512	20.5	98.2	22.1	9.66	6.90
768	65.1	322.1	66.2	30.3	11.5
1024	179.1	756.5	190.3	71.7	33.8

(sec.)

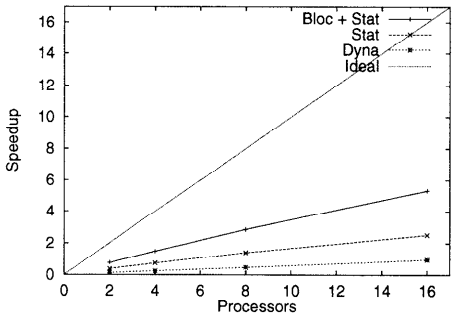


図8 コレスキー分解の台数効果（1024 × 1024）
Fig. 8 Speedup of Cholesky Decomposition (1024 × 1024).

6.2.3 ガウス消去法

GEの計測結果を**表6**に，台数効果を**図9**に示す．

行列サイズが1024のとき，ソフトウェアによるキャッシュメモリシステムを用いてもGEの台数効果は1.11であった．これに対してブロック化を利用すると，台数効果が6.42に向上した．

7. 議　　論

7.1 プリプロセッサおよびブロック化の効果

プリプロセッサによる効果を**表7**に，ブロック化による効果を**表8**に示す．

本研究で用いたキャッシュシステムはすべてソフトウェアで構築されている．このため，ハードウェアキャッシュに比べて制御コストが高く，コンパイラによる最適化が必要である．しかし，すべてソフトウェアで構築されているため，キャッシュ動作の制御もソフトウェアによって行うことができる．十分に依存解析が行えるアプリケーションについては，プリプロセッサによって必要なデータを事前にコピーする制御命令をコードに挿入することが可能であるため，キャッシュリストの探索やキャッシュコヒーレンスの保持といった管理が不要となる．よって表7の *Dyna/Stat* に示すように，プリプロセッサを利用することによりMMで4.9～5.0倍CHでも2.3～2.6倍の大幅な速度向上がみられる．

表6 ガウス消去法の実行時間（16 processors）
Table 6 Execution time of Gaussian elimination (16 processors).

Size	*Seri*	*Para*	*Dyna*	*Bloc + Dyna*
256	5.0	51.2	6.0	2.01
512	42.3	399.2	43.5	11.3
768	136.4	1335.4	148.7	29.1
1024	370.2	3186.9	332.2	56.9

(sec.)

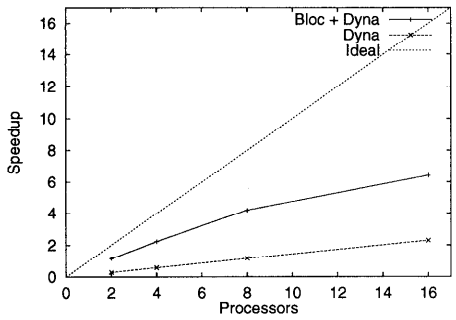


図9 ガウス消去法の台数効果（1024 × 1024）
Fig. 9 Speedup of Gaussian elimination (1024 × 1024).

また，表 8 に示すように，プリプロセッサを用いた場合 ($\frac{Stat}{Bloc+(Stator Dyna)}$)，用いない場合 ($\frac{Dyna}{Bloc+(Stator Dyna)}$) の双方でブロック化が有効である．特にプリプロセッサを用いない場合のブロック化を施さないプログラムの台数効果 (*Seri*/*Dyna*) は，MM で 1.5～2.3，CH で 0.92～0.94，GE で 0.97～1.11 と低く，並列処理の効果がまったく得られないが，ブロック化を施すことにより台数効果が MM で 11.9～13.7，CH で 2.97～5.30，GE で 3.74～6.51 となり，並列処理の効果を得ることができるようになる．

このようにソフトウェアによるキャッシュシステムを利用する場合，実用的な性能を得るためにはブロック化をはじめとした，キャッシュの利用効率をあげるような何らかの最適化技術が必要である．

7.2 ピボット選択を行う並列 LU 分解

本稿の実験において利用したピボット選択を行う並列 LU 分解は，インデックス配列による間接アクセスを行うため，静的な依存解析は行うことができない．これに対し，実行時に判明したインデックス配列の値を利用し，データの生成，参照関係を解析することによって通信の集合化や通信時間の隠蔽を行う inspector/executor という方法[14]が研究されている．このような方法を本研究におけるシステムのランタイムライブラリに加えることにより，不規則問題に対する性能を向上させることが期待できる．

表7 プリプロセッサによる効果
Table 7 Effect of pre-processor.

Code	Size	$\frac{Seri}{Stat}$	$\frac{Para}{Stat}$	$\frac{Dyna}{Stat}$
MM	512	11.3	25.7	4.91
MM	1024	7.46	16.0	5.06
CH	512	2.12	10.2	2.29
CH	1024	2.50	10.6	2.65

表8 ブロック化による効果
Table 8 Effect of array-blocking.

Code	Size	$\frac{Seri}{Dyna}$	$\frac{Seri}{Bloc+Stat}$	$\frac{Dyna}{Bloc+Stat}$	$\frac{Stat}{Bloc+Stat}$
MM	512	2.30	13.7	5.93	1.21
MM	1024	1.48	11.9	8.03	1.58
CH	512	0.92	2.97	3.20	1.4
CH	1024	0.94	5.30	5.63	2.12
Code	Size	$\frac{Seri}{Dyna}$	$\frac{Seri}{Bloc+Dyna}$	$\frac{Dyna}{Bloc+Dyna}$	-
GE	512	0.97	3.74	3.85	-
GE	1024	1.11	6.51	5.84	-

8. まとめ

本稿では，分散メモリ型並列計算機上にソフトウェアによって構築されたキャッシュメモリシステムについて，最適な動作を決定するプリプロセッサの効果を3種のベンチマークプログラムにより計測，評価した．また，RISC 計算機におけるキャッシュ最適化技術を適用することにより，さらに高い性能が得られることを示した．

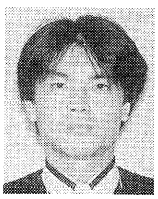
本研究で利用したソフトウェアによるキャッシュメモリシステムは簡潔な方法で構築されているため，様々な分散メモリ型並列計算機や，ワークステーションクラスタ上での実装が容易である．また，より効率の良いキャッシュメモリシステムが分散共有メモリ上で開発されているが，ソフトウェア的に制御できるよう実装されていれば，本研究を応用することによってアプリケーションに適したキャッシュ動作の制御を行うことができる．

参考文献

1) Amarasinghe, S. and Lam, M.: Communication Optimization and Code Generation for Distributed Memory Machines, *Proc. Conf. on Programming Language Design and Implementation*, ACM, pp.126–138 (1993).
2) Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations, *Computer*, Vol.29, No.2, pp.18–28 (1996).

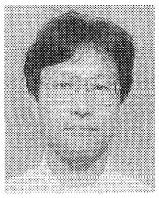
3) Anderson, J. and Lam, M.: Global Optimizations for Parallelism and Locality on Scalable Parallel Machines, *Proc. Conf. on Programming Language Design and Implementation*, ACM, pp.112–125 (1993).
4) Archibald, J. and Baer, J.-L.: Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, *ACM Trans. Comp. Syst.*, Vol.4, No.4, pp.273–298 (1993).
5) Bruening, U., Giloi, W. and Wchroeder-Preikshat, W.: Latency Hiding in Message Passing Architectures, *Proc. 8th Intl. Parallel Processing Symposium*, pp.704–709 (1994).
6) Carter, J., Bennett, J. and ZWENEPOEL, W.: Techniques for Reducing Consistency-rclatcd Communication in Distributed Shared-memory Systems, *ACM Trans. Comp. Syst.*, Vol.13, No.3, pp.205–243 (1995).
7) Chaiken, D., Kubiatowicz, J. and Agarwal, A.: LimitLESS Directories: A Scalable Cache Coherence Scheme, *Proc. Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp.224–234 (1991).
8) Culler, D.E., Dussequ, A., Goldstein, S.C., Krishnamurthy, A., Lumetta, S., Luna, S., von Eicken, T. and Yelick, K.: Introduction to Split-C, Technical Report, University of California, Berkeley (1993).
9) Hanxleden, R. and Kennedy, K.: GIVE-N-TAKE – A Balanced Code Placement Framework, *Proc. Conf. on Programming Language Design and Implementation*, ACM, pp.107–120 (1994).
10) Hennessy, J. and Patterson, D.: *Computer Architecture: A Quantitative Approach*, Chapter 8, Morgan Kaufmann (1990).
11) Hill, M., Larus, J., Reinhardt, S. and Wood, D.: Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors, *ACM Trans. Comp. Syst.*, Vol.11, No.4, pp.300–318 (1993).
12) Kattner, R., Eger, M. and Schloer, C.: Modeling Cache Coherence Overhead with Geometric Objects, *CONPAR 1994* (1994).
13) Koelbel, C. and Mehrotra, P.: Compiling Global Name-space Parallel Loops for Distributed Execution, *IEEE Trans. of Parallel and Distributed Syst.*, Vol.2, No.4, pp.440–451 (1991).
14) 窪田昌史，三吉郁夫，大野和彦，森眞一郎，中島浩，富田眞治：不規則アクセスをともなうループの並列化コンパイル技法—Inspector/Executor アルゴリズムの高速化，情報処理学会論文誌，Vol.35, No.4, pp.532–541 (1994).
15) Lenoski, D., Laudon, J., Kourosh, G., Weber, W., Gupta, A., Hennessy, J., Horowitz, M. and Lam, M.: The Stanford Dash Multiprocessor, *Computer*, Vol.25, No.3, pp.63–79 (1989).
16) Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Comp. Syst.*, Vol.7, No.4, pp.321–359 (1989).
17) 中條拓伯，藏前健治，金田悠紀夫，前川禎男：ソフトウェア DSM におけるコヒーレント・キャッシュシステムの実装と評価，情報処理学会論文誌，Vol.36, No.7, pp.1719–1727 (1995).
18) 南里豪志，佐藤周行，島崎眞昭：非同期通信によって自動最適化を行う並列化コンパイラの設計，九州大学大型計算機センター計算機科学研究報告 (1994).
19) 南里豪志，佐藤周行，島崎眞昭：ソフトウェアキャッシュの制御を行う最適化コンパイラによる分散共有メモリシステムのメモリ性能改善，並列処理シンポジウム JSPP'96 論文集，pp.331–338 (1996).
20) Nitzberg, B. and Lo, V.: Distributed Shared Memory: A Survey of Issues, and Algorithms, *Computer*, Vol.24, No.8, pp.52–60 (1991).
21) Sato, H., Nanri, T. and Shimasaki, M.: Using Asynchronous and Bulk Communication to Construct an Optimizing Compiler for Distributed-memory Machines with Consideration Given to Communication Costs, *Proc. 1995 ACM Intl. Conf. on Supercomputing*, pp.185–189 (1995).
22) Stumm, M. and Zhou, S.: Algorithms Implementing Distributed Shared Memory, *Computer*, Vol.23, No.5, pp.54–64 (1990).

(平成 8 年 9 月 17 日受付)
(平成 9 年 2 月 5 日採録)



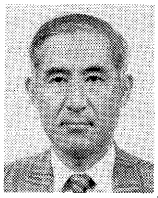
南里 豪志（正会員）

1969 年生．1993 年九州大学工学部情報工学科卒業．1995 年同大学大学院工学研究科修士課程修了．1996 年九州大学大型計算機センター助手，現在に至る．並列化コンパイラの研究に従事．日本ソフトウェア科学会会員．



佐藤　周行（正会員）

1962年生．1985年東京大学理学部情報科学科卒業．1990年同大学大学院博士課程修了．1990年九州大学大型計算機センター講師，1992年同助教授，1996年同大学大学院システム情報科学研究科助教授，現在に至る．理学博士．プログラム意味論，並列化コンパイラの研究に従事．ACM，日本ソフトウェア科学会各会員．



島崎　眞昭（正会員）

1943年生．1966年京都大学工学部電子工学科卒業．1971年同大学大学院工学研究科博士課程単位習得退学．1971年同大学工学部助手（情報工学科），同大学助教授を経て，1989年九州大学教授（大型計算機センター），1997年京都大学教授（工学研究科電気工学専攻），現在に至る．工学博士．1974～1975年ニューヨーク大学クーラント研究所アソシエイト・リーサーチ・サイエンティスト．スーパーコンピューティング，計算機ソフトウェア，計算科学の研究に従事．著書「スーパーコンピュータとプログラミング」．電子情報通信学会，電気学会，日本応用数理学会，日本ソフトウェア科学会，ACM，IEEE各会員．