

スヌープキャッシュ制御機構の DOACROSS ループへの適用

松 本 尚[†]

共有メモリ共有バス型アーキテクチャは汎用性が高く比較的实施が容易である。また、スヌープキャッシュの発明により、大幅に共有バスのトラフィックが削減され、共有バス構成で接続できるプロセッサの台数が増大した。このために、共有メモリ共有バス型アーキテクチャは多くのマルチプロセッサシステムで採用されてきた。しかし、大規模数値計算や細粒度の並列処理といった用途では、スヌープキャッシュによるバストラフィックの削減効果があまり得られず、大きな性能向上は得られなかった。筆者は共有メモリ共有バス型のマルチプロセッサを高性能化する機構として、データのタイプによってスヌーププロトコルを切替えて通信を最適化するスヌープキャッシュ制御機構、およびこの機構と生産者-消費者型の同期機構とを統合した MISC 機構を提案してきた。これらの機構の定量的な効果を測定するために、実行駆動型のマルチプロセッサシミュレータを作成した。このシミュレータ上で、DOACROSS 型のループを例に挙げて、シミュレーション実験を行った。その結果、これらの機構が共有バスのアクセス回数削減とメモリアクセスのレイテンシ削減に大きな効果があることが確認された。また、依存距離の短い DOACROSS ループにおいて、MISC 機構による同期オーバヘッドの削減効果が大きな性能向上をもたらすことが確認された。

Effects of Dynamic Snoop-Protocol Switching on High-Speed Execution of Doacross Loops

TAKASHI MATSUMOTO[†]

In multiprocessor systems, the overheads caused by inter-processor communication and synchronization continue to be impediments to the efficient execution of parallel programs. To reduce these types of overhead in shared-memory/shared-bus multiprocessors, we proposed two innovative hardware mechanisms: the Inter-Cache Snoop Control Mechanism (ICSCM), which switches snoop-protocols dynamically to optimize shared-bus utilization, and the Mechanism for Integrated Synchronization and Communication (MISC), which extends the ICSCM to support producer-consumer-type synchronization efficiently. To investigate the performance improvements effected by these mechanisms, we have developed an execution-driven multiprocessor simulator and performed simulation experiments on it, using doacross loops as sample targets. In consequence of the experiments, we have realized remarkable effects of ICSCM/MISC on the speedup of doacross loops. This paper reports the structure of the simulator and the simulation methods, and discusses the results of the experiments.

1. はじめに

筆者は共有メモリ共有バス型マルチプロセッサの性能を向上させる機構として、スヌープキャッシュ制御機構（以下 ICSCM: Inter-Cache Snoop Control Mechanism）を提案してきた^{1),3),4)}。この機構はメモリ領域ごとに最適なスヌーププロトコルを、そこに割り当てられるデータの性質によって設定し、実行時のバストラフィックのたびにスヌーププロトコルを動的に切替えるものである。プロトコルの動的な切替のために、バストラフィック時にプロトコルのタイプがバス上のすべてのスヌープキャッシュに放送さ

れ、スヌープキャッシュはその情報に基づいて動作を決定する。実装の簡便さと仮想記憶への対応のために、メモリ管理機構と組み合わせてページ単位にプロトコルタイプの情報を付加することを提案した。さらに、動的なスヌーププロトコルの切替の下で有効に機能する all-read, allwrite, allread-write といった新しいタイプのプロトコルを提案した。これらのプロトコルは一回のバスアクセスで複数のキャッシュにデータを取り込ませることが可能であり、複数のプロセッサで共用される構造体データ（例えば、配列）へのアクセスのレイテンシを削減するのに効果がある。

また、ICSCM を拡張して、通信の高効率化のみな

[†] 東京大学理学部情報科学科
Department of Information Science, Faculty of
Science, The University of Tokyo

* 本論文ではレイテンシという単語でプロセッサがメモリシステム（キャッシュを含む）に読み出し/書き込み要求を出してからその要求が完了するまでの時間を指す。

らず、生産者-消費者型の同期のオーバーヘッドを削減し、同期をも効率良く実行する機構である MISC (Mechanism for Integrated Synchronization and Communication) の提案も行った^{2),4)}。これらの機構やプロトコルの有効性を定量的に評価するために、ハードウェア構成を詳細に記述した実行駆動のマルチプロセッサシミュレータを作成した。本論文では、このシミュレータの構成について説明し、これを用いて行った ICSCM と MISC の DOACROSS ループに対する定量的評価の結果を報告する。

2. スヌープキャッシュ制御機構

まず、新しいプロトコル、ICSCMの構成、ICSCMの生産者-消費者の同期問題への拡張である MISC について簡単に説明する。

2.1 新しいタイプのプロトコル

効率的にプロトコルを切り替える機構の下では、ある限られたタイプのデータにのみ有効なプロトコルも使用可能になる。そのようなプロトコルの例として all-read プロトコルを説明する。このプロトコルはバス・スヌープを行っているキャッシュが、他のキャッシュによる共有バスを介した read の際に、積極的にバス上のデータを取り込むプロトコルである。ただし、データを取り込むことでキャッシュ内の他のデータを共有メモリに書き戻す必要が生じる場合は、取り込み動作を行わないものとする。all-read プロトコルは以下のような場合に適している。全プロセッサが同じデータをほぼ同時期に参照する必要があるときに各プロセッサが個別にリードを行い、メモリからキャッシュにデータを張り付けるとすると、キャッシュ内にデータが取り込まれるまでに、データ数×プロセッサ数のバス・トラフィックが共有バスで発生する。all-read をこのデータに対するプロトコルとして用いれば、データ数だけのバス・トラフィックで済み効率が良い。

同様に、write 時に必ず共有バスをアクセスし、他のキャッシュにバス上の書き込みデータを取り込ませる all-write、read 時は all-read の動作を行い、write 時は all-write の動作を行う allread-write 等の新しいプロトコルも考えられる。なお、これらの新しいプロトコルは細区分することが可能である。例えば、all-read プロトコルは write 時に update 動作をする

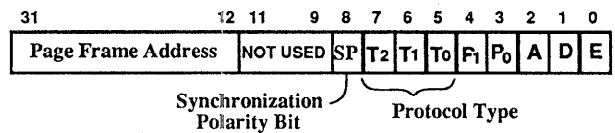


図 1 ICSCM のページエントリ (TLB) の構成
Fig. 1 A page-entry/TLB required for ICSCM/MISC.

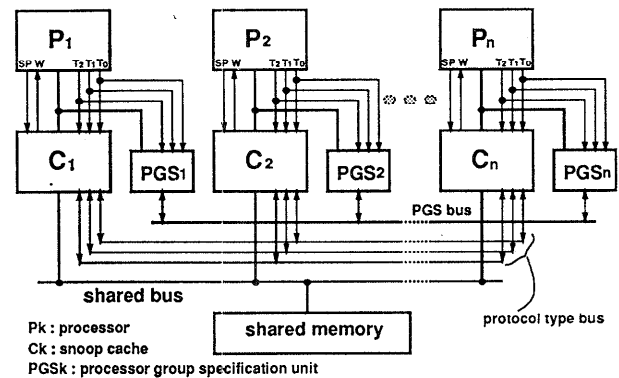


図 2 ICSCM を含むシステムの構成
Fig. 2 The organization of ICSCM/MISC.

allread-up, invalidate 動作をする allread-inv 等に細区分され、主記憶への書き戻しのタイミング等によってもっと細かく分類することも可能である。

2.2 ICSCM の構成

機構の実現方法については実現の容易さとメモリ管理・仮想記憶との親和性を考えて、従来型のプロセッサ (またはその専用 MMU) にすでにインプリメントされているページ管理機構を利用し、ページごとにプロトコルのタイプを表す情報を付加することにする。

図 1 に ICSCM のページエントリの例を示す。TLB (Translation Look-aside Buffer) も同じ構成を採る。P1, P0 はアクセス権を示すビット、A, D, E は仮想記憶の管理に用いられるビット、SP は後に述べる機構の拡張のためのビットである。そして、T2, T1, T0 が ICSCM の特徴であるページのプロトコルのタイプを表すビットである。つまり、このページに属するデータはどのタイプのプロトコルを使ってアクセスされるかを示す情報である。プロセッサはメモリ・アクセスする際に、このアクセスがどのプロトコルで処理されなければならないかを示すため、TLB のこれらのビットを外部に出力する。そして、図 2 のようにマルチプロセッサを構成し、キャッシュから共有バス上に対してアクセスが行われるときは、共有バスにもプロトコルのタイプを示す信号線が出力される。他の

キャッシュはこの信号線でプロトコルを選択しながらバス・スヌープ(バス監視)を行う。なお、TLBのフィールドの一部が外部に出力されていないプロセッサでは、ページ・フレーム・アドレスの上位の2~3ビットをプロトコルのタイプを表すビットとして流用できる可能性がある。コンパイラによるアクセス・パターンの解析やプログラマによるプロトコルタイプの指示等によって、変数やワークエリアをそれに適したプロトコルのタイプを持つページ内に割り振る。また、複数の論理ページを一枚の物理ページにマッピングするページエイリアシングの技法を使えば、同一のデータに対して複数のプロトコルを割り当てることができる。そして、論理アドレスを替えてアクセスすることで、同一のデータに対して複数のプロトコルをきめ細かく動的に切り替えることが可能である。

all-read タイプに類するプロトコルでは、無関係な仕事を実行しているプロセッサのキャッシュにデータを取り込ませないために、取り込みを行うキャッシュ群(プロセッサ群)を指定できることが望ましい。この機能の実現のため、キャッシュ群を指定する情報(識別番号)を伝えるためのプロセッサ群指定バスが共有バスに追加され、群指定情報を蓄えておいて all-read タイプ系のアクセス時に外部に出力するプロセッサ群指定回路がプロセッサごとに用意されている。そして、各キャッシュはスヌープ動作時に、バスのアクセスが all-read タイプ系であり、プロセッサ群指定バス上に自分のグループ識別番号が出力されているときのみデータ取り込みの動作を行う。

2.3 MISC の構成

生産者-消費者の同期を効率良く行うためにメモリ上でのfull/emptyビット^{5),6)}による同期機構とICSCMを持ったマルチキャッシュシステムを融合する。キャッシュのデータメモリと共有メモリにワードごとにfull/emptyを表す1ビットの同期ビットを付加する。この拡張によって、生産者-消費者タイプの同期をデータ転送と統合して行うことが可能となり、同期のオーバヘッドの削減とバスアクセス回数の削減が可能である。

まず、同期ビットに関する基本的な動作を説明する。MISCのメモリシステム上では同期ビットをデータの一部として扱い、同期ビットを含めたメモリワードをキャッシュする。プロセッサがあるメモリワードへ書き込みを行うと、キャッシュはそのワードの同期ビットを full にする。プロセッサがデータを読み込む場

合は、キャッシュはそのワードの同期ビットを調べ、fullであればデータをプロセッサに出力し、emptyであればプロセッサをメモリウェイトでサスペンドさせる。スヌープによるコンシステンシの保証により、プロセッサをサスペンドさせているキャッシュはサスペンドの原因となっているメモリワードへの他のプロセッサによるデータの書き込みを検出できる。そして、書き込みを確認したら、キャッシュはプロセッサのサスペンドを解除して処理を再開させる。この動作により、ソフトウェアによる付加的な処理なしに、データ転送(書き込み)前にメモリワードを読み出してしまいうことが防げる。

同期ビットの基本動作はプロトコル切替と無縁であるが、細粒度で生産者-消費者型のデータ転送を行うメモリワードに対して、消費者が1台のときは all-write プロトコル、消費者が複数の時は allread-write プロトコルを用いることがバスアクセス回数の最小化とレイテンシの削減に効果がある。そこで、同じタイプのメモリワードを集めて、ページを構成し all-write (または allread-write) プロトコルを割り当てる。また、すべてのメモリワードに対してこの同期チェックを実行することは同期ビットの初期化等のオーバヘッドのために非効率的である。そこで、同期のチェックを行うプロトコルと行わないプロトコルを用意する。さらに、メモリワードを再利用して繰り返し同期を行う場合、同期のたびに同期ビットを empty に初期化する必要がある。このオーバヘッドを除去するために、ページエントリに SP (Synchronization Polarity) ビットが用意され(図1参照)、full/empty と 0/1 の対応を規定している。プロトコル切替の場合と同様にページエイリアシングの技法により、SP ビットを反転させた二つの論理ページをメモリワードに割り当てれば、論理アドレスを切替えてアクセスすることで同期ビットの意味の反転が可能になるので初期化の必要はない。ページエイリアシングが行えない場合でも、配列等のまとまったデータであれば、ページ単位で SP ビットを反転することで初期化が可能である。

3. シミュレータの構成

3.1 全体構成

今回作成したマルチプロセッサシミュレータの構成について説明する。厳密な評価が可能なように、クロックレベルの実行駆動方式、つまりプロセッサモデルがグローバルクロックに従って実際に命令のフェッチ

解釈実行を行い、スヌープキャッシュや共有メモリやバスアビータといったシステムの構成要素もグローバルクロックに従って厳密に動作をシミュレートする方式を採用している。プロセッサモデルの命令セットは R 3000⁹⁾の命令セットを一部拡張 (fetch & inc, fetch & dec, exchange, compare & exchange 命令の追加) して使用している。ただし、今回のシミュレーションでは拡張された部分の命令は使用していない。プロセッサのパイプラインは2段で1段目が命令フェッチ、2段目が解釈実行となっており、2段目の時間コストは命令ごとに設定可能である。プロセッサはレジスタ 32 本の他に命令キャッシュを内蔵している。ICSCM および MISC のためにメモリをセグメントによって管理しており (現在ページ管理機構は未実装)、セグメントごとにキャッシュプロトコルを指定できる。スヌープキャッシュは命令・データ混在型で、キャッシュのブロック単位で転送を行うため、ブロック幅が共有バスのデータ幅よりも広い場合は、バースト転送によって共有バス上のデータの転送を行う。また、ICSCM に対応するために、複数のプロトコルを実装しており (現在 10 種類)、プロセッサからのプロトコルタイプの出力によりプロトコルを使い分ける。さらに、シミュレータは MISC への対応のためにメモリシステムにはワード (4 byte) ごとに同期ビットが用意され、プロセッサのアドレス出力の上位2ビットを MISC による同期を行うか否かを示すビットと SP ビットに割り当てている。プロセッサ内蔵キャッシュとスヌープキャッシュは共にキャッシュサイズ、ブロックサイズ、ウェイト数、キャッシュメモリのアクセスタイムが設定可能である。バスアビータのバス調停は先着順で、同一クロックサイクル内に到着したバス要求に対しては、バス使用ごとに優先度をプロセッサ間でローテーションすることで公平化を図っている。共有バスはアドレス出力から所要データ転送まで1バスサイクルで行うインタロック方式で、1バスサイクルのクロック数は共有メモリまたはキャッシュメモリのアクセスタイムで決定される。

なお、このシミュレータはオブジェクト指向言語 COB⁸⁾を使用して記述され、基本的な構成部品 (プロセッサ、キャッシュ等) がオブジェクトとして記述されているため、プロセッサ台数やアーキテクチャ (部品の組合せ方法) を容易に変更してシミュレーションができる点に特長がある。

3.2 スヌーププロトコル実装方式

ICSCM のキーとなるスヌープキャッシュへの複数のプロトコルの実装について解説を行う。

このシミュレータのスヌープキャッシュのモデルは内部状態がハードウェアによる実装の内部状態と一致するように記述されている。allread 等の新しいタイプのプロトコルも従来のスヌーププロトコルを統一的に記述できる MOESI モデル⁹⁾の枠内で実装でき、キャッシュブロックの状態を表すタグは 3 bit (5 状態) である。スヌープキャッシュのタグ部分は高性能の実装のためにはデュアルポートメモリで実現される必要があり、この条件が満たされることを仮定した。さらに、本シミュレータではスヌープキャッシュのデータ部分もデュアルポートメモリで実装されているものと仮定した。これは、シングルポートメモリを用いた場合、allread 系のプロトコルは update 系のプロトコルと同様に、キャッシュのデータ部分に対してバスからのトランザクションとプロセッサからのトランザクションが競合して性能が低下する可能性があるためである。単純な invalidate プロトコルと比べてハードウェアコストが増大するが、この投資に値する性能向上がもたらされるかどうかを調べるのが本シミュレーションの目的の一つである。

また、スヌープキャッシュ間でプロトコル切替のために伝達する情報はプロトコルタイプそのものである必要はなく、現在のバストランザクションに対しては2ビットの情報があれば済む。つまり、スヌーピング動作において、ライトでキャッシュヒット時に invalidate と update の区別と、ライトとリード共にキャッシュミス時に積極的にデータを取り込むか否かの区別の2ビット分である。ただし、この方式を用いる場合はデータのプロトコルタイプとキャッシュブロックの状態からこの信号線を作成するためのロジックが必要になる。本シミュレータではインクリメンタルにプロトコルを増設する際に、追加実装が容易となるようにプロトコルタイプを伝達する方式を採用している。

正確さを期するために、性能評価で使用する4種類のプロトコルについて、シミュレータのスヌープキャッシュに実装された動作を記述しておく。invalidate プロトコルは Illinois プロトコル、update プロトコルは Firefly プロトコルとして実装されている¹⁰⁾。allwrite プロトコルは、プロセッサのメモリ読み出し時は通常のキャッシュ動作で、書き込み時はライトスルー (ただし、キャッシュミス時は fetch-on-write)

動作で、前述のように他のキャッシュにバス上のデータを取り込ませる。allread-write プロトコルは、プロセッサのメモリ書き込み時は allwrite プロトコルと同じ動作で、読み出しでかつキャッシュミスによるバスを使ったデータフェッチ時に、他のキャッシュにバス上のデータを取り込ませる。

4. 性能評価

4.1 シミュレーションの方式と仮定

今回のシミュレーションに用いたサンプルプログラムを図3(a)に示す。これはイテレーション間で依存関係を持つ典型的な DOACROSS ループである。並列実行時の同期を極力減らすために、逆依存関係を代入先の配列を変更することで解消し、図3(b)のように変形する。なお、この変形は一般的な変数のリネーミング操作で達成される。

この変形後のプログラムをイテレーションごとの並列実行を可能とするために図4のように並列化する。図中において NPROC は並列に実行するプロセッサの数（並列実行中は固定されている）、myid は各プロセッサに固有の識別子で実行に参加するプロセッサに0, 1, 2, 3... と識別番号が与えられる。iter 0, iter 2, iter 3 はイテレーション間の同期を行う変数へのポインタである。このプログラムでの同期の方法¹⁾を簡単に説明する。プロセッサごとに同期変数が用意され、イテレーションの計算値の代入（生産）終了後に、その同期変数（iter 0 が指している）に現在のイテレーションカウンタの値を代入する。他のイテレーションで計算される値を参照（消費）すべき地点では、まずその値を計算するプロセッサの同期変数（iter 2 と iter 3

```
for (i = 3; i < N-2; i++)
    x[i] = x[i] + w1 * x[i+2] + w2 * x[i+1]
        - w3 * x[i-3] - w4 * x[i-2];
```

(a)

```
c[0] = x[0];
c[1] = x[1];
c[2] = x[2];
for (i = 3; i < N-2; i++)
    c[i] = x[i] + w1 * x[i+2] + w2 * x[i+1]
        - w3 * c[i-3] - w4 * c[i-2];
```

```
c[N-2] = x[N-2];
c[N-1] = x[N-1];
```

(b)

図3 サンプルプログラム

Fig. 3 Sample target program.

```
setup(NPROC, myid, &iter0, &iter2, &iter3);
for (i = myid; i < 3; i += NPROC) {
    c[i] = x[i];
    *iter0 = i;
}
for (; i < N-2; i += NPROC) {
    temp = x[i] + w1 * x[i+2] + w2 * x[i+1]
    while (*iter3 < i-3);
    temp = temp - w3 * c[i-3];
    while (*iter2 < i-2);
    c[i] = temp - w4 * c[i-2];
    *iter0 = i;
}
for (; i < N; i += NPROC) {
    c[i] = x[i];
    *iter0 = i;
}
```

図4 同期変数を使って並列化されたプログラム

Fig. 4 Parallelized program using sync. variables
(for ICSCM).

が指している)の中身を調べ、自分が参照しようとしているイテレーションの値を越えていれば参照を実行し、越えていなければ越えるまでビジーウェイトを行う。図中の setup(...) は正しいプロセッサの同期変数を指すように iter 0, iter 2, iter 3 を初期化する処理を示す。

なお、このプログラムでは生産者-消費者の同期しか存在しないため、MISC を利用する場合は図4の中に見られた同期処理がすべて必要ない。結局 MISC を用いる場合の並列化されたプログラムは図5となり、配列 c で同期ビットを利用した同期を行う。

このようにして作成された並列化版のプログラム(図4および図5)に対して、R 3000 用の市販の最適化コンパイラが出力したコードを参照しながら、ハンドコンパイルにより ICSCM (または MISC) を使用するアセンブルコードを作成し、シミュレーションに使用した。プログラムのデータはすべて 4 byte 長で、データのメモリへのアロケーションは配列、同期変数

```
for (i = myid; i < 3; i += NPROC)
    c[i] = x[i];
for (; i < N-2; i += NPROC)
    c[i] = x[i] + w1 * x[i+2] + w2 * x[i+1]
        - w3 * c[i-3] - w4 * c[i-2];
for (; i < N; i += NPROC)
    c[i] = x[i];
```

図5 明示的な同期処理のない MISC 用プログラム

Fig. 5 Parallelized program using full/empty bits
(for MISC).

ともに 4 byte ごとに順次にスペースを空けることなく配置した。ただし、配列 x と配列 c の先頭および最初の同期変数では同一キャッシュブロック内で他のデータと混在しないようにアラインメントを取った。純粋なループの実行効率を測定するため、同一条件でループ回数を 200 と 400 に設定して、二回の実行時間（総クロック数）を測定し、その時間差から単位時間あたりの処理イテレーション数（処理能力）を求めた。

シミュレータの設定可能なパラメータのうち、シミュレーションにおいて固定したパラメータの設定内容を以下に列挙する。

- 共有バスは一本のみで、バス幅が 64 bit と設定した。特に断らない限り 1 アクセスあたり 1 転送の非バースト転送を行うものとし、共有バス上で 3 クロックのコストがかかると仮定した。この時のスヌープキャッシュのブロックサイズは 8 byte である。また、比較で用いるバースト転送時は 1 アクセス 4 転送（ブロックサイズ 32 byte）で、共有バス上で 6 クロックのコストがかかると設定した。
- キャッシュの構成はプロセッサ内蔵の命令キャッシュが 4 KB、スヌープキャッシュが 16 KB、共に 2 ウェイ。データの load および store はバスアクセスを伴わない場合が 2 クロックで、バスアクセスを伴う場合は 2 クロックに加えてバスを獲得してから 4 クロック（バースト転送時で 7 クロック）のレイテンシがある。
- プロセッサの命令コストはレジスタ演算命令が 1 クロック、ただし、乗除算は 4 クロック、分岐命令が 2 クロック、load および store

はデータのアクセスが終了するまでサスペンドと設定した。

4.2 シミュレーションの結果と考察

4.2.1 ICSCM におけるプロトコル間の特性比較

図 4 のプログラムを ICSCM（バスは非バースト転送）で実行した場合のプロセッサ台数の増加に伴うスピードアップを示すグラフを図 6 に示す。横軸はプロセッサ台数を表し、縦軸は 100 クロックで処理したループのイテレーションの回数を表す。図中の ALLREAD_WRITE, ALLWRITE, UPDATE, INVALIDATE が配列 x と c に対して使用したプロ

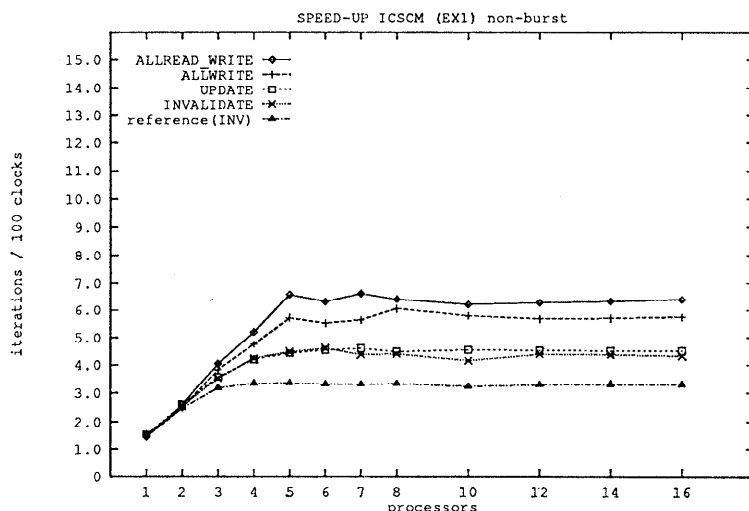


図 6 ICSCM におけるプロトコルの影響
Fig. 6 Effects of snoop protocols in ICSCM.

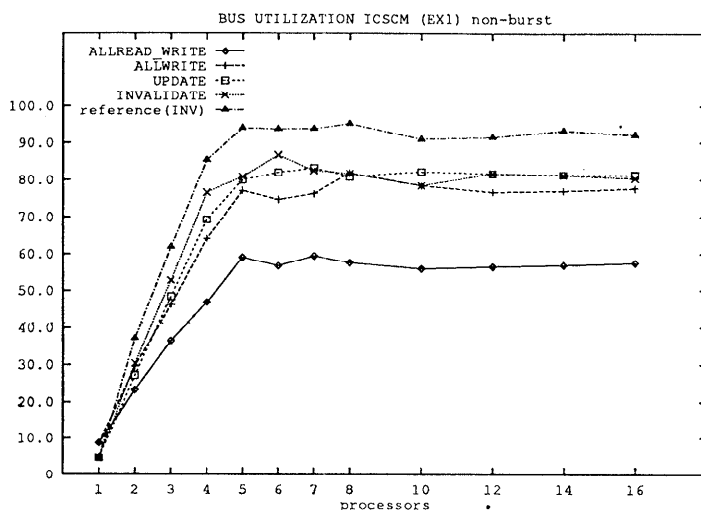


図 7 図 6 における共有バスの使用率
Fig. 7 Bus utilization in the case of Fig. 6.

トコルを表している。なお、この4本のグラフの測定では、同期変数に対して常に update プロトコルを使用している。reference (INV) に対応するグラフでは、同期変数も含めてすべてのデータに対するプロトコルを invalidate に固定した場合を示す。使用したサンプルプログラムはイテレーション間の依存距離が2のものであるが、イテレーションをオーバーラップして実行することにより、allread-writeプロトコルではプロセッサ5台で1台の場合の4倍以上のスピードアップが得られている。また、市販のマイクロプロセッサでよく用いられる invalidate プロトコルに固定した場合と比較して、約2倍の性能が得られている。図7は図6に対応するシミュレーションにおける単位時間あたりの共有バスの使用率(占有率)を示している。図より、reference (INV) 以外のケースにおいては、共有バスにまだ余裕があることが判り、イテレーションのオーバーラップによる高速化が限界まで達成されていることが判る。reference (INV) 以外のこれら4本のグラフに対応するシミュレーション間での性能差の主原因は、使用プロトコルの差に伴うキャッシュのヒット率の差とそれに基づくレイテンシの差である。また、図8はループを200イテレーション分だけ実行するのに伴う共有バスの使用時間(バスアクセスごとの占有クロック数の総和、バスの使用回数に比例)を示す。図から、allread-write プロトコルのバス使用回数の最小化の効果の大きさが把握できる。

4.2.2 MISC におけるプロトコル間の特性比較

同期に対する最適化を施したMISC(バスは非バースト転送)で同じ処理内容の図5のプログラムを実行した場合について、プロセッサ台数の増加に伴うス

ピードアップを示すグラフを図9に示す。グラフの表記法は図6と同じである。同期ビットの使用により同期のオーバーヘッドが削減され、イテレーションの中でオーバーラップして実行可能である処理の割合が増加したため、allread-write プロトコルではプロセッサ7台で約6倍、10台で約7倍のスピードアップが得られることが判る。図9のグラフの振舞いに関して、シミュレーションで得られた他の統計データ(バスの使用率と使用時間、キャッシュごとのバス使用回数、アクセスタイプごとのバス使用回数等)も参考にして、少し詳しく分析した結果を述べる。allread-write プロ

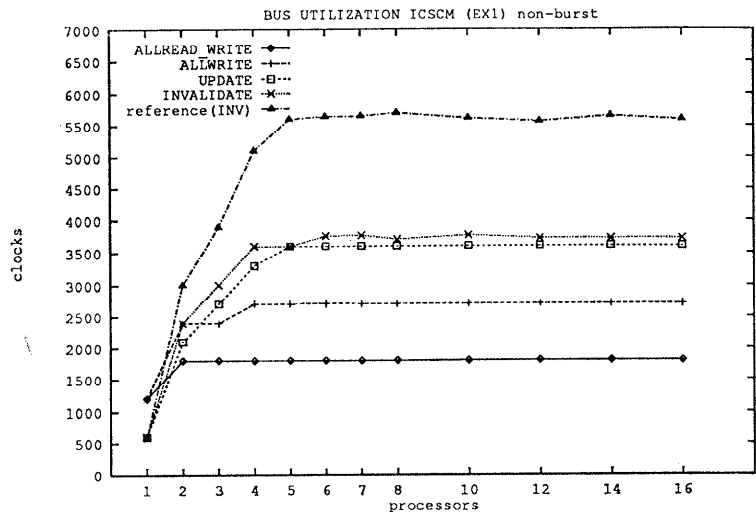


図8 図6における共有バスの使用時間
Fig. 8 Total bus occupation time for 200 iterations in the case of Fig. 6.

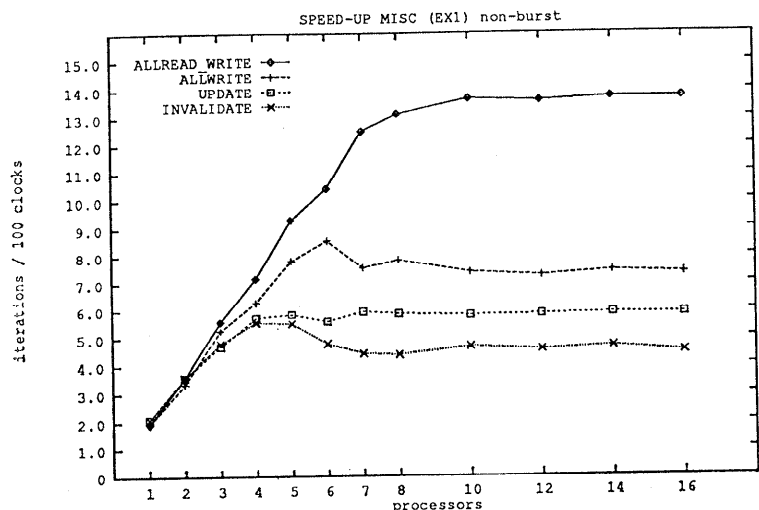


図9 MISC におけるプロトコルの影響
Fig. 9 Effects of snoop protocols in MISC.

トコルはグラフが飽和した時点では、共有バスの使用率が 80% 程度でまだ余裕があり、図 6 と同様にイテレーションの実行を限界までオーバーラップさせた高速化が達成されている。他のプロトコルのグラフに関しては、性能が飽和する地点での共有バスの使用率が 95~100% であり、共有バスも飽和している。invalidate プロトコルの性能がプロセッサ台数 4~8 の回りでダウンする理由は、「生産者-消費者の関係にある通信において消費者のキャッシュへのデータの読み出しが先行し（同期ビットで待たされる）、生産者による書き込みがこの先行した読み出しのキャッシュエントリを無効化し、消費者は実行を継続するために再度キャッシュにデータを読み出す」という現象が増加して共有バスアクセスが増加したためである。allwrite プロトコルの性能がプロセッサ台数 6~12 の回りでダウンする理由は、台数が少ない範囲では生産者の書き込みが先行して消費者がキャッシュ内からデータを読み出すことが可能であったが、台数が増えると消費者の読み出しが先行し始めて、共有バスアクセスが増加したためである。今述べたようにプロセッサ台数が増え過ぎると、allwrite プロトコルの典型的なメリットは消える。しかし、図が示すように性能は update プロトコルよりも常に優れている。この理由は書き込み時のキャッシュミスに対する fetch on write 動作と false sharing¹²⁾にある。本シミュレーションでは 1 個のキャッシュブロックに連続した 2 個の配列要素が割り当てられている。生産者の書き込みは常に新しい配列要素に対して行われるが、allwrite プロトコルが使われる場合、ブロック内の配列要素のうち一方（後から書き込まれる要素）の書き込み時にはキャッシュ内に当該ブロックはすでにフェッチされており、新たに共有メモリからブロックをフェッチするコストが掛からない。また、allread-write プロトコルの読み込み時も false sharing は他のプロセッサのキャッシュに前もってデータを割り付ける役割を果たし有利に働いている。このように、本シミュレーションでは false sharing が性能を引き上げる方向に作用

している。

図 10 に共有バスにバースト転送を採用した場合の図 9 に対応するグラフを示す。比較のために、非バースト転送の結果である図 9 から allrw-nonbst プロトコルに対応するグラフを図内に allrw_nonbst として表示してある。性能が飽和する以前は非バースト転送の場合と有意な性能差はないが、性能が飽和してくるとどのプロトコルについても非バースト転送の場合よりも性能が悪くなっている（図 9 参照）。サンプルプログラムのような Doacross ループの場合、1 イテレーションの間に生産者-消費者型同期の消費が 2 回、生産が 1 回あるような粒度の細かい並列処理である。このため、キャッシュのブロックサイズを大きくしてバースト転送によってバスの転送能力を大きくするよりも、1 回のバス転送時間が短いことの方が、レイテンシが小さくバスの占有率が少ないため、重要であることがわかる。ただし、バス幅を広げて転送能力を拡大する場合は、レイテンシやバス転送時間を増やすことなくキャッシュのブロックサイズを大きくできるので、性能向上に効果がある。この場合、ブロックサイズが大きくなったことにより、前述の false sharing によるヒット率の向上の恩恵にあずかることもできる。

4.2.3 依存距離と性能との相関関係

本小節では、最高性能を示す allread-write プロトコルに議論の焦点を当てることにする。そして、使用する図では allread-write プロトコルのグラフと参考

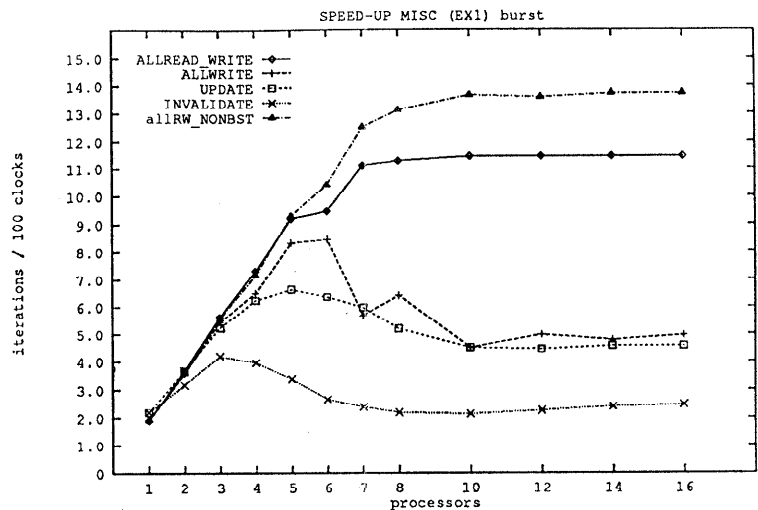


図 10 MISC のバースト転送時のプロトコルの影響

Fig. 10 Effects of snoop protocols in MISC (burst transfer).

としてすべてのアクセスを invalidate プロトコルに固定したグラフのみを表示する。ループイテレーション間の依存距離の差が性能に及ぼす影響を調べるために、1 イテレーションの計算内容を

$$c[i] = x[i] + w1 * x[i+2] + w2 * x[i+1] \\ - w3 * c[i-K-1] - w4 * c[i-K];$$

と変更して、依存距離 K を 1~4 まで変化させたシミュレーションを行った (これまで議論したシミュレーションは $K=2$ に相当する)。同期を同期変数で行う場合を図 11 に示し、同期を MISC の同期ビットで行う場合を図 12 に示す。図中の allRW (K) は allread-write プロトコルで依存距離 K を、inv(K) は invalidate プロトコルで依存距離 K のシミュレーションの結果を示し、他の記法は図 6 と同じである。図 12 の allRW (3) と allRW (4) のスピードアップが鈍いが、これは共有バスの飽和によるものである。ちなみに、16 プロセッサで allRW (3) で約 90%、allRW (4) ではほぼ 100% 共有バスを使用していた。参考までに図 12 に対応する共有バスの使用クロック数のグラフを図 13 に示す。allread-write プロトコルの共有バスの使用時間は依存距離、プロセッサ台数に無関係で (図中の 1 本の水平直線)、invalidate プロトコルに比べ非常に優れていることがわかる。

さらに粒度の細かい DO-ACROSS ループの高速化の可能性について調べるために、1 イテレーションの計算内容を

$$c[i] = x[i] + w2 * x[i+1] \\ - w4 * c[i-K];$$

に変更し、処理量を約 35 クロックに減らしたシミュレーションを行った。MISC の同期ビットで同期を行った場合の結果を図 14 に示す。図中の記法は図 12 と同じである。allRW (2)、

allRW (3)、allRW (4) のグラフがプロセッサ 8 台以上で完全に飽和している。この理由は共有バスの飽和によるものでバスの使用率がほぼ 100% に達していたためである。ちなみに、10 台で allRW (1) が 45%、inv (1) が 85%、他のケースはすべてほぼ 100% であった。しかし、このように細粒度の DOACROSS ループであっても、invalidate プロトコルのみのマルチプロセッサ (同期ビットはサポートされている) と比べて、MISC を allread-write プロトコルで利用すれば 2~3 倍といった有意なスピードアップが共有バス型マルチプロセッサで得られることが確認された。

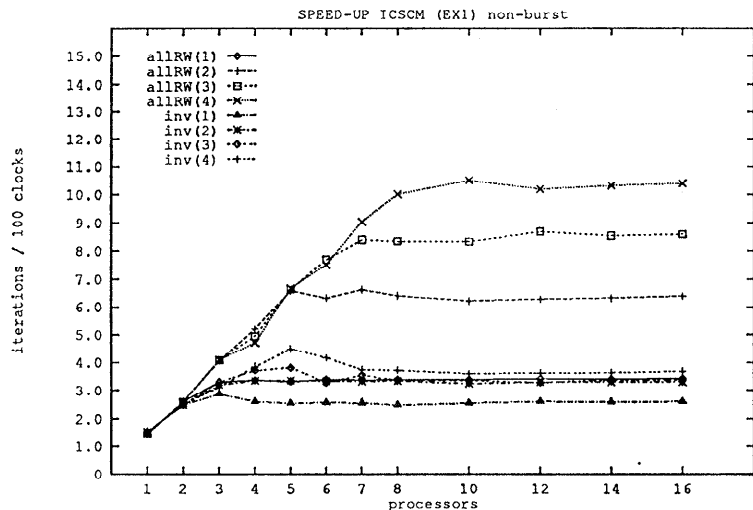


図 11 ICSCM における依存距離の影響

Fig. 11 Effects of dependency distances in ICSCM.

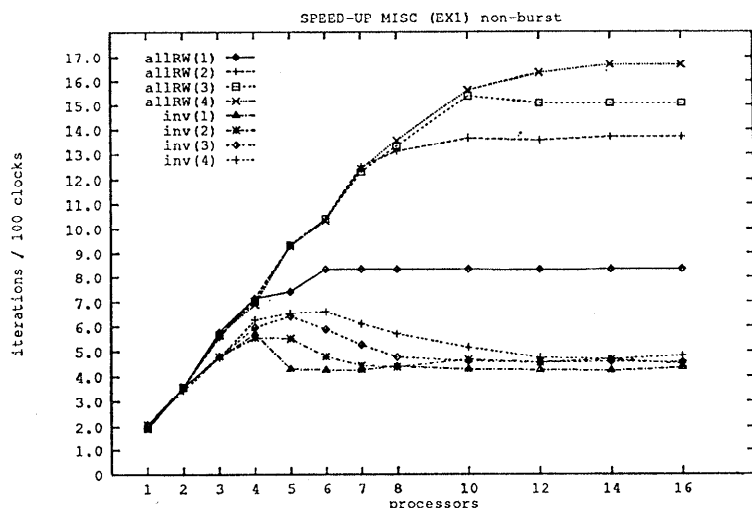


図 12 MISC における依存距離の影響

Fig. 12 Effects of dependency distances in MISC.

4.2.4 DOALL ループへの適用の可能性

依存距離を変化させたシミュレーションの結果 (図 11, 図 12, 図 14) より, 以下のような推論が可能である. 依存距離を離していくと DOACROSS ループの性質は, DOALL ループ (イテレーション間に依存がないループ) に近付いていく. このことから考えて, データパラレルタイプの数値計算プログラムによく見られる DOALL ループについても, allread-write プロトコルが使用できる ICSCM を搭載したマルチプロセッサは, データのアクセスパターンによっては大幅なスピードアップを達成できる可能性がある.

この推論を裏付けるために 1 イテレーションの計算内容を

$$\begin{aligned} c[i] = & x[i] + w1 * x[i+2] \\ & + w2 * x[i+1] \\ & - w3 * c[i] \\ & - w4 * c[i]; \end{aligned}$$

と変更して, シミュレーションを行った. イテレーション間に依存がないため, 同期は行わない. また, $w4 * c[i]$ の項はイテレーションあたりの処理量を他のシミュレーションと揃えるために故意に挿入しており, 最後の 2 項を結合則で一回の乗算にまとめるような最適化は行っていない.

シミュレーション結果を図 15 に示す. 図中の記法は図 6 に準じている. allRW_BURST と inv_BURST は, 比較のためにバースト転送を行った場合における, allread-write と invalidate プロトコルに対応するグラフを示している. DOACROSS の場合と同じプロセッサへのイテレーションの割り当てを行っているため false sharing が発生し, allread-write プロトコルでは読み出しによる共有バス使用回数が減って性能が向上する. よって, ループボディでより多くの配列を参照する場合 (本シミュレーションでは x

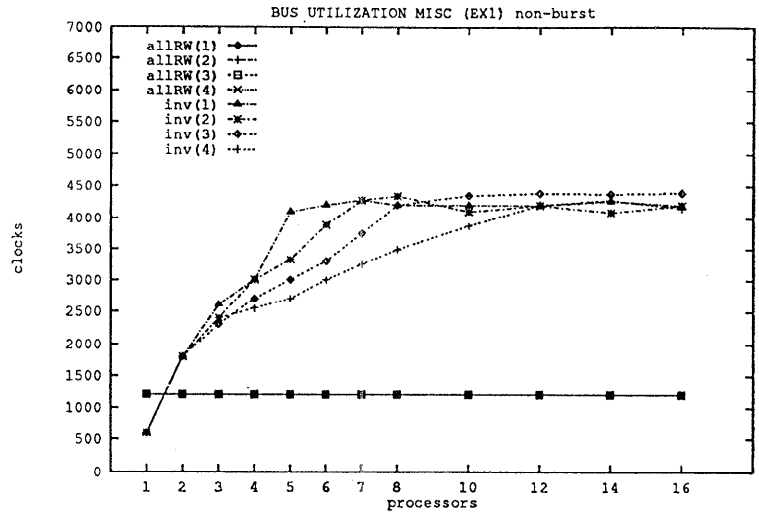


図 13 図 12 における共有バスの使用時間

Fig. 13 Total bus occupation time for 200 iterations in the case of Fig. 12.

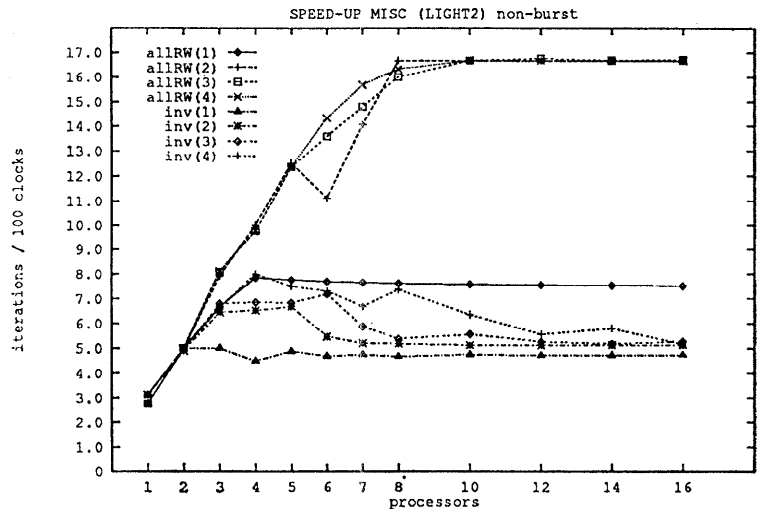


図 14 MISC における依存距離の影響 (より細粒度の場合)

Fig. 14 Effects of dependency distances in MISC with a lighter loop body.

と c の 2 種類) は, 他プロトコルに対する allread-write プロトコルの効用がより顕著になる. ただし, 配列への書き込みに対しては false sharing が悪い方向に作用し, 書き込みの度に共有バスが使用される. この書き込みに対する false sharing の悪影響のために, バースト転送時の一回のバス使用時間の長さの影響が顕著となり, バースト転送時の性能が非バースト転送時の性能を下回っている (図 15 参照).

イテレーションのプロセッサへの割り当て方式を false sharing が存在しないようにキャッシュのブ

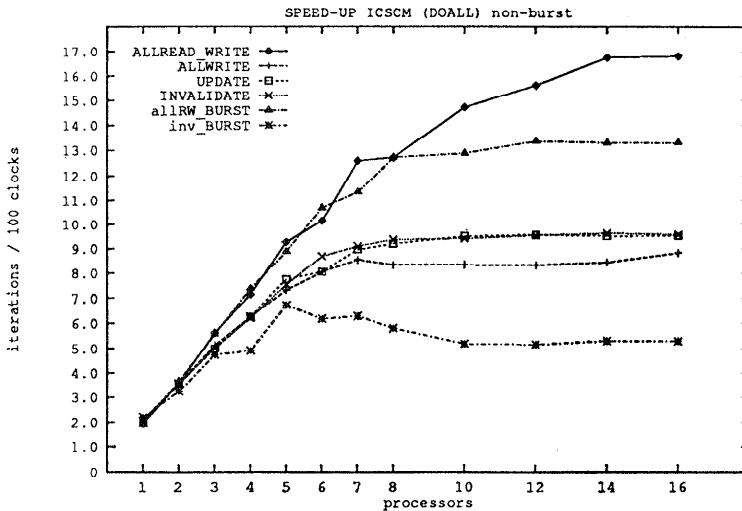


図 15 DOALL ループにおけるプロトコルの影響

Fig. 15 Effects of snoop protocols in ICSCM using DOALL loop.

ロックサイズを考慮して決定する、すなわち書き込みが行われる配列がブロックサイズ単位でまとまるようにプロセッサに割り当てる場合は、書き込み時のバス使用回数を減らすことができる。具体的には、バスを使用した書き込みはライトバック時にブロック単位で主記憶に書き戻されるケースのみになる。しかし、この方式を採用すると DOALL ループと DOACROSS ループで異なったイテレーション割り付け方式が必要となり、コンパイル技法が複雑になる。また、以下のようにループボディに複数の代入文が含まれ、一部のステートメントのみがイテレーション間の依存関係を持っている場合、

$$\begin{aligned}
 c[i] &= x[i] + w1 * x[i+2] + w2 * x[i+1] \\
 &\quad - w3 * c[i]; \\
 d[i] &= c[i] + w4 * x[i] + w5 * x[i+1] \\
 &\quad - w6 * d[i-3] - w7 * d[i-2];
 \end{aligned}$$

並列処理を行うためには、ループを DOALL ループと DOACROSS ループに分離して実行する必要がある。上記のような例の場合、ステートメント間で同じ配列を参照しているため、読み出しによる共有バスの使用回数およびレイテンシの削減の観点からはループを分離せずに実行した方が有利である。DOALL ループにおいても DOACROSS ループにおけるプロセッサへのイテレーションの分配方式と同じ方式で性能を引き出すことが可能である場合は、この議論が成立し、ループを分離せずに効率の良い並列実行が可能である。allread 系プロトコルの使用はこの高性能化への

第一歩と言える。なお、DOALL ループでは頻繁にキャッシュコンシステンスを保つ必要はないので、書き込み時の false sharing によるバスの使用回数の増大は、本来回避可能な問題のはずである。この問題の解決が合理的なハードウェア量とコンパイル技法で（できればマルチジョブ環境で）可能であるかどうかの検討は高性能化への次のステップであり、今後の課題である。

5. おわりに

DOACROSS 型のループは細粒度並列実行によってしか高

速化ができない処理である。そして、そのタイプのループを高速並列実行可能にすることは、大規模なアプリケーションの並列実行において、処理時間のボトルネックになる可能性の高い部分を高速化することにつながり、実用的な意義も大きい。本論文で採用したような DOACROSS ループでは、invalidate プロトコルを持つマルチプロセッサで並列実行しても、バスの飽和や同期のオーバーヘッドのため、単一プロセッサによる実行に対して小さなスピードアップしか得られない。しかし、スヌープキャッシュ制御機構 (ICSCM) の下で allread-write プロトコルを用いれば、イテレーション当たりのバスアクセスが大幅に削減できる。しかも、キャッシュのヒット率が向上するためレイテンシも少なくできる。特に、本論文で用いたような配列計算では、allread-write プロトコルのバスアクセス最小化の効果が非常に大きいことが判った。そして、同期変数を介したソフトウェアによる同期を行っても、単一プロセッサによる実行 (図 9 のプロセッサ 1 台に相当) に比べて、イテレーション間の依存距離をかなり上回る倍率のスピードアップが可能であった。また、ICSCM に同期機能を拡張した MISC を用いれば、同期のためのバスアクセスと同期オーバーヘッドがさらに削減できて、より大きなスピードアップ (本論文の条件下では ICSCM のみの場合の 2 倍程度) が得られた。本論文では、ベクタ化による高速化ができない DOACROSS ループを例に挙げて、ICSCM および MISC を採用した密結合マルチプ

ロセッサの有効性を示した。一般に共有バスやキャッシュは大規模数値計算と相性が悪いとされてきたが、スヌープキャッシュを制御して通信を最適化することで、低コストで実現できる共有バス型マルチプロセッサを数値計算に効率良く適用できる可能性があることを示した。

今後の課題として、本論文で使用したシミュレータを利用して本格的なアプリケーションに対して、ICSCM および MISC の効果を測定することが挙げられる。

謝辞 サンプルプログラムのアセンブルコードおよび実験結果のグラフ表示ツールの作成に御協力をいただいた日本 IBM 東京基礎研究所の森山孝男氏に感謝いたします。

参 考 文 献

- 1) 松本 尚: 細粒度並列実行支援機構, 情報処理学会計算機アーキテクチャ研究会報告 No. 77-12, pp. 91-98 (1989).
- 2) 松本 尚ほか: スヌープキャッシュを用いて通信と同期を統合する機構, 電子情報通信学会コンピュータシステム研究会報告, CPSY 90-42, pp. 25-30 (1990).
- 3) 松本 尚: 細粒度並列実行支援マルチプロセッサの検討, 情報処理学会論文誌, Vol. 31, No. 12, pp. 1840-1851 (1990).
- 4) Matsumoto, T. et al.: MISC: A Mechanism for Integrated Synchronization and Communication Using SnooP Caches, *Proc. of the 1991 Int. Conf. on Parallel Processing*, Vol. 1, pp. 161-170 (Aug. 1991).
- 5) Jordan, H. F.: Performance Measurement on HEP—A Pipelined MIMD Computer, *Proc. 10th Int. Symp. on Computer Architecture*, pp. 207-212 (Jun. 1983).
- 6) Arvind and Iannucci, R. A.: Critique of Multiprocessing von Neumann Style, *Proc. 10th Int. Symp. on Computer Architecture*, pp. 426-436 (Jun. 1983).
- 7) 日本電気: μ PD 30300 (VR 3000) 32ビット・マイクロプロセッサ ユーザーズ・マニュアル アーキテクチャ編, 日本電気 (1989).
- 8) 小野寺民也, 上村 務: COB によるオブジェクト指向プログラミング, *Computer Today*, No. 40, pp. 26-38 (1990).
- 9) Sweazey, P. and Smith, A. J.: A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus, *Proc. 13th Int. Symp. on Computer Architecture*, pp. 414-423 (Jun. 1986).
- 10) Archbald, J. and Baer, J.-L.: Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, *ACM Trans. Computer Systems*, Vol. 4, No. 4, pp. 273-298 (1986).
- 11) Midkiff, S. P. and Padua, D. A.: Compiler Generated Synchronization for Do Loops, *Proc. 1986 Int. Conf. on Parallel Processing*, pp. 544-551 (Aug. 1986).
- 12) Torrellas, J. et al.: Measurement, Analysis and Improvement of the Cache Behavior of Shared Data in Coherent Multiprocessors, Technical Report CSL-TR-90-412, Stanford Univ. (Feb. 1990).

(平成 4 年 9 月 16 日受付)
(平成 5 年 1 月 18 日採録)



松本 尚 (正会員)

1962 年生. 1985 年東京大学工学部計数工学科卒業. 1987 年大阪市立大学大学院理学研究科物理学専攻修士課程修了. 日本アイ・ビー・エム(株)東京基礎研究所研究員を経て, 1991 年 11 月より東京大学理学部情報科学科助手. 並列計算機アーキテクチャ, オペレーティングシステム, 最適化コンパイラに関する研究に従事. 他にニューラルネットワーク, 学習, 力の超統一理論等に興味を持つ. 1990 年本学会学術奨励賞受賞. 電子情報通信学会, 日本ソフトウェア科学会, ACM 各会員.