並列計算機 Cenju-4 上でのソフトウェア ・キャッシュシステムの実装と評価 1

Implimentation and Evaluation of a Software Cache on Parallel Computer : Cenju-4

山内 聡

平成 12 年 2 月 15 日

内容梗概

近年、分散メモリ環境での並列処理がさかんに行われている。従来、分散メモリ環境ではメッセージ通信によるプログラムが一般的であった。しかし、メッセージ通信によるプログラムは、いつ、どのノードと、どのようなデータをやりとりするかを、明確に記述しなければならず、それらを直接的に記述する必要のない共有メモリ型のプログラムに比べて非常に困難である。

この欠点を補うために、物理的に分散配置されているメモリをユーザに対して 1 つの共有メモリに見せる、分散共有メモリ (DSM:Distributed Shared Memory) システムに関する研究がさかんである。

DSM の実現レベルには、ハードウェアとソフトウェアによるものがある。ハードウェアによるものは、NUMA(Non Uniform Memory Access) 型のメモリアーキテクチャのマルチプロセッサが対応する。これは、ハードウェアで共有メモリが実現されているため高速な通信が可能であるが、固定的なシステム構成のものが多く、アプリケーションの特徴を生かした高速化のための柔軟な対応は難しい。ソフトウェアによるものは、NORMA(NO Remote Memory Access) 型マルチプロセッサやワークステーションクラスタなどの上に、ソフトウェアで共有メモリを実現したものである。ソフトウェアが介在するため通信のオーバーヘッドが発生するが、アプリケーションの特徴を生かした最適化によりハードウェアによる実装を上回る性能を引出す可能性を持っている。一般に DSM では通信遅延を隠蔽するために、キャッシュシステムが必須である。DSM の実装と同様キャッシュシステムもハードウェアによる実装とソフトウェアによる実装が可能である。

本研究では、種々のソフトウェアキャッシュシステムを並列計算機 Cenju-4 に実装し、Cenju-4 のハードウェアキャッシュシステムとの比較による評価を行った。この際ソフトウェアキャッシュシステムでプロセッサ間の通信は、Cenju-4 の遠隔メモリアクセス機構ならびに遠隔関数呼び出し機能のみを用いて実装した。また、共有メモリ領域へのアクセスには最低一回の関数呼び出しが必要である。これに対し、ハードウェアキャッシュシステムでは、関数呼び出しを行うことなく共有メモリ領域のデータにアクセスが可能である。

今回実装したソフトウェアキャッシュシステムは、write-update 型と write-invalidate 型のコヒーレンスプロトコルの切り換えと、コヒーレンスの単位となるブロックサイズの変更をサポートする。

このソフトウェアキャッシュシステムとハードウェアキャッシュシステムに対し、行列サイズ $n \times n$ の行列積と LU 分解を実行しその性能を評価した。行列積による評価では、ハードウェア DSM のブロックサイズが 128B と固定であるのに対して、ソフトウェアキャッシュシステムでは、ブロックサイズを大きく設定することで、問

題サイズ n が 1024 以上ではソフトウェアキャアシュを用いる方が、高速に実行ができること示せた。しかし、n が 1024 未満の場合は、必要とされる通信量の減少とともにソフトウェアキャッシュのオーバーヘッドが顕在化し、ハードウェアキャッシュの方が高速となった。また、内積ガウス法による LU 分解では並列実行の単位が小さいため、ブロックサイズを大きくして通信量を減らすことができずソフトウェアキャッシュは十分な性能を出すことができなかった。また内積ガウス法の特徴を生かして、write-invalidate 型のコヒーレンスプロトコルから write-update 型へコヒーレンスプロトコルを変更することにより高速化が図れることを示そうとしたが、現在の実装では十分な最適化ができなかったためその効果を示すことができなかった。

今後の課題として、ソフトウェアキャッシュのオーバヘッドを減らすことがあげられ、またこのソフトウェアキャッシュシステムでは、ブロックサイズの変更とコヒーレンスプロトコルの選択しか実装できなかったため、さらに多くのキャッシュシステムの実装が考えられる。

1 Abstract

Recently, there has been much interest in parallel processing on distributed memory system. Formerly, programmer has used message passing in distributed memory system. With message passing, programmer must manage the data movement between processors, timing of communication and size of data. With shared memory, programmer is not aware of them. So, programming with message passing is more difficult than that with shared memory. In distributed shared memory system(DSM), programmer treats the distributed memory as a shared memory. DSM combines the simple programming model of shared memory with the scalability of distributed shared memory.

There are two level of DSM system implementations, hardware level and software level. Hardware DSM is multiprocessor of the NUMA(Non Uniform Memory Access) type. In hardware DSM, hardware manage the DSM protocol, so communication speed is high. But many hardware DSM is not flexible system about coherence protocol, block size, placement of shared memory, and so on. Software DSM is implemented by software on multiproseccor in NORMA (NO Remote Memory Access) type or workstation cluster. In software DSM, software manages the DSM protocol, so the communication speed is low. But with software implement, this DSM can manage the virtual shared memory flexibly.

In software DSM, many systems are implemented with cache system, to hide the communication delay among the distributed memories of each nodes, and there are many ways to implement the cache system.

We implemented software cache system on the parallel computer cenju-4. Cenju-4 supports the hardware DSM and remote memory access systems. With remote memory access system, programmer can access memory of other nodes without interrupting that nodes, but nodes cannot cache memory of other nodes. So we call this access non-cache hardware DSM. We implemented software cache system on this access. We can compare the software cache system with hardware.

In this software cache system, programmer can select the coherence protocol, write-update or write-invalidate, and programmer change the coherence unit size. We evaluate this software cache with matrix multiplication and LU programs. In matrix multiplication, because of the software overhead, software DSM don't show better performance in case of smaller matrix size than 1024×1024, but because of the non flexibility of hardware DSM and the flexibility of software DSM, software DSM shows better performance, when the matrix size is larger than 1024×1024.

In LU decomposition, the size of parallel

processing unit is small. And software cache doesn't show better performance than hardware DSM. Because software cache can not use the good performance of communication in big block size, And write-update protocol shows better performance in theory, but write-update protocol did not show better performance in my result, It is because the update protocol of software cache system has many overheads.

We will decrease the overhead of software cache system. With changing memory consistency model, the the overhead will decrease. And in this software cache system, we can only select the block size and coherence protocol, so we will implement and evaluate more cache system.

目 次

1 Abstract **2**

2 はじめに **3**

3 研究の背景 **4**
- 3.1 メモリアーキテクチャの分類 . . 4
- 3.2 分散共有メモリシステム 4
- 3.3 DSM でのキャッシュシステム . 5
- 3.4 DSM の高速化 5
  - 3.4.1 メモリコンシステンシモデル 5
  - 3.4.2 管理粒度 5
  - 3.4.3 キャッシュコヒーレンス 6
- 3.5 Cenju-4 の概要 6
  - 3.5.1 Cenju-4 の構成 6
  - 3.5.2 Cenju-4 上の並列プログラミングライブラリ . . 6
  - 3.5.3 Paralib/CJ4 7
  - 3.5.4 Cenju-4 の分類 7

4 ソフトウェア ・キャッシュシステム **7**
- 4.1 ソフトウェア ・キャッシュの構成 7
- 4.2 キャッシュの管理 8
- 4.3 コヒーレンスプロトコル 9
- 4.4 排他制御 10
- 4.5 ユーザインターフェイス 10
- 4.6 ソフトウェア ・キャッシュの動作 11

5 ソフトウェア ・キャッシュ評価 **11**
- 5.1 行列積を用いた評価 11
  - 5.1.1 キャッシュブロックサイズによる比較 11
  - 5.1.2 プロトコルの違いによる評価 12
  - 5.1.3 CC-NUMA/CJ4 との比較 12
  - 5.1.4 考察 13
- 5.2 LU 分解を用いた評価 13
  - 5.2.1 キャッシュブロックサイズによる比較 13
  - 5.2.2 プロトコルの違いによる評価 14
  - 5.2.3 CC-NUMA/CJ4 との比較 14
  - 5.2.4 考察 15

6 おわりに **15**

2 はじめに

近年、分散メモリシステムでの並列計算が注目されている。並列計算機上でのプログラミングモデルには、ノード間でのデータの受け渡しをメッセージ通信により行うものと共有メモリを介して行うものがある。メッセージ通信によるプログラミングでは、ノード間で通信が行われる度に、それぞれのノードで、いつ、どのノードと、どのようなデータをやりとりするかを、明確に記述しなければならない。一方、共有メモリ型のプログラミングモデルでは、それらの直接的な記述は必要なく、ユーザーはメモリ操作に伴なうノード間通信を意識せず他ノードとデータを共有することができるため容易な記述が可能である。

分散共有メモリシステムとは、物理的に分散配置されているメモリをユーザに全体として１つの共有メモリに見せるシステムで、ユーザは分散メモリシステム上で共有メモリによる容易なプログラムが可能になる。分散共有メモリの実現には、ハードウェアによる方法とソフトウェ

アによる方法がある。ハードウェアによる方法とは、NUMA 型のマルチプロセッサに対応し、ソフトウェアによる方法とは、NORMA 型のマルチプロセッサ上やワークステーションクラスタ上でソフトウェアで DSM システムを実現する方法である。

ハードウェアによるものもソフトウェアによるものも、共有メモリが分散配置されているので、メモリアクセスが不均一となる。そのため共有メモリの管理が重要である。ハードウェアによるものは高速な通信が可能であるが、メモリの管理は固定的で実行アプリケーションの特性を生かすような柔軟な対応は難しい。一方、ソフトウェアによるものは柔軟なメモリ管理が可能で、実行アプリケーションの特性を生かすことができるが、多くの場合通信に OS の支援が必要であるため、通信がボトルネックとなっていた。

本研究では、並列計算機 Cenju-4 上にソフトウェア ・キャッシュシステムを実装した。Cenju-4 では、CC-NUMA 型のハードウェアでキャッシュコヒーレンスをとる DSM システムと、他ノードのメモリアクセスをキャッシュする機構がハードウェアで提供されていない DSM と呼べる機構を提供している。このシステムは他ノードのメモリに直接アクセスできる遠隔メモリアクセス機構で、ユーザレベルで高速なアクセスが可能である。この遠隔メモリアクセス機構にキャッシュシステムを実装することにより、様々なキャッシュシステムを評価することができ、さらにソフトウェア ・キャッシュシステムとハードウェアによる高速 DSM 機構の比較が可能である。

3 研究の背景

3.1 メモリアーキテクチャの分類

マルチプロセッサシステムは、メモリの物理構成について、大きく次の 3 つに分類することができる。

(1) UMA(Uniform Memory Access) 型 (図 1(a))

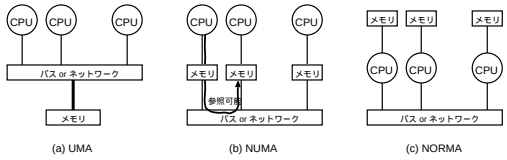

図 1: メモリアーキテクチャ

共有メモリ方式で、どのプロセッサから見てもメモリ空間が物理的に均質であるもの。複数のプロセッサがバスあるいはネットワーク経由でメモリに接続される形態。この方式の典型的なものにバス結合のマルチプロセッサがある。

(2) NUMA(Non Uniform Memory Access) 型 (図 1(b))
共有メモリ方式で、それぞれのプロセッサから見てメモリが物理的に非均質であるもの。プロセッサとメモリで構成されるノードが複数個互いにネットワークで接続された形態。他ノードのメモリをキャッシュするシステムをハードウェアで実装しているものとしていないものがある。

(3) NORMA(NO Remote Memory Access) 型 (図 1(c))
プロセッサとメモリで構成されるノードが複数個互いにネットワークで接続された形態だが、共有メモリを持たずプロセッサからのアクセスはプロセッサ対応のローカルメモリに対してのみ行われる。他ノードのメモリへのアクセスは、相手ノードのプロセッサの支援が必要である。

3.2 分散共有メモリシステム

分散共有メモリ (Distributed Shared Memory:DSM) システムとは、ノードごとに物理的に分散配置されているメモリをユーザに 1 つの共有メモリに見せるシステムである。DSM の実現方法として、ハードウェアによる方法とソフトウェアによる方法がある。

ハードウェアによる実現方法とは NUMA 型のマルチプロセッサのことで、ハードウェアに

よってDSM機構が実装されている。ソフトウェアによる実現方法とはNORMA型のメッセージパッシング型マルチプロセッサやワークステーションクラスタ上でのソフトウェアによる実現のことで、ユーザからの共有メモリアクセスがソフトウェアによって自動的にノード間通信に変換されて実行されるものである。

Cenju-4[1] の CJrmread/CJrmwrite や AP1000+[2] の put/get などは、ハードウェアで他ノードのローカルメモリをキャッシュする機構はないが、他ノードのメモリへアクセスができるので、一種のDSMであると考えることも可能である。

3.3 DSMでのキャッシュシステム

DSMにおけるキャッシュシステムとは、他ノードのローカルメモリアクセスをキャッシュするシステムのことで、自ノードのローカルメモリアクセスのみをキャッシュすることではない。DSMにおけるキャッシュシステムの実現方法にも、ハードウェアによるものとソフトウェアによるものが考えられる。

まず、NUMA型のマルチプロセッサには、ハードウェアによって他ノードのメモリへのアクセスをキャッシュしコヒーレンス制御をサポートするものがあり、これにはCC-NUMA(cache-coherent NUMA)型マルチプロセッサが対応する。また、他ノードのメモリへのアクセスのみをハードウェアによってサポートしているものもある。

ソフトウェアによって実現されているDSMでハードウェアでキャッシュシステムを実現することはないが、仮想メモリ機構のページフォールトハンドラを用いるなど、ハードウェアが提供している機能をソフトウェアで用いながら実現する方式と、プログラム実行時に共有メモリへのアクセスが起こるとキャッシュシステムのルーチンへ制御を移動するソフトウェアによる実現方式がある。

3.4 DSMの高速化

DSMでは共有メモリが物理的に分散配置されているためメモリアクセスが不均一となる。そのため高速化を実現する上で共有メモリの管理手法が重要となる。

3.4.1 メモリコンシステンシモデル

共有メモリシステムでは複数のプロセッサが共有メモリにアクセスできるため、それらのメモリアクセスの順序に自由度がある。このアクセス順序のモデルをあらかじめ決めておき実行結果を正しくしなければならない。このモデルをメモリコンシステンシモデルという。DSMでは、共有メモリが物理的に分散配置されているため、メモリコンシステンシを維持するためにはノード間通信が必要となる。そのためどのようなモデルを採用するかが性能に大きくかかわる。

CC-NUMAの場合、排他制御の必要のない通常のロードやストアは、同期命令とは無関係に他のプロセッサに伝えられるが、ソフトウェアによって管理されているものでは、同じようにデータの変更のたびに通信を行った場合、通信のオーバーヘッドが大きくなる恐れがある。それを防ぐために、制約を緩くしコンシステンシの維持のための通信を減らすメモリコンシステンシモデルが提案されている。

3.4.2 管理粒度

DSMでは共有メモリをブロックと呼ばれる単位に分割してこの共有メモリブロックを各ノードに分散配置し管理を行う。この共有メモリブロックの大きさをどうするかがDSMの性能にに大きくかかわる。

共有メモリブロックを大きくとると管理対象数が減少し、管理情報を保持する領域は少なくてすむが、大きくすることによりFalse Sharing (図2) が頻発してしまう。

False Sharingとは、同じ共有メモリブロック内の異なったデータに対して複数のノードが同時にアクセスを行い、そのブロックがあたかも共有されているかのように管理される状態のことで、無駄な通信を引き起こし性能低下の原因となる。これらのデータはアクセスパターンが異なるので、別の共有メモリブロックに格納すると効率的に管理できる。

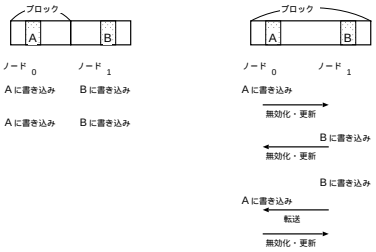

図 2: False Sharing

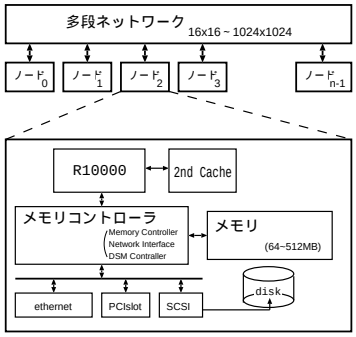

図 3: Cenju-4 の構成

CC-NUMA では共有メモリブロックの単位は、キャッシュメモリのラインサイズなどの特定の値で固定であるシステムが多い。ソフトウェアによる DSM では、共有メモリブロックの大きさをアプリケーションごとに変化させたり、管理方法を変化することにより高速化が可能である。

3.4.3 キャッシュコヒーレンス

CC-NUMA、全ての共有メモリブロックに対して同じプロトコルでコヒーレンスを維持するものが多いが、ソフトウェア DSM では、共有メモリブロックごとにプロトコルを別にしているものや、プログラムごとに切り換えできるようにして高速化を図っているものがある。

このように、CC-NUMA では多くの場合、固定的なメモリ管理が行われているが、ソフトウェア DSM では様々なメモリ管理方法が提案されている。

3.5 Cenju-4 の概要

以下に、ソフトウェア・キャッシュシステムの実装を行った並列計算機 Cenju-4 の概要と Cenju-4 を用いた理由を示す。

3.5.1 Cenju-4 の構成

Cenju-4 は、1024 個までのノードが 4×4 のネットワークスイッチから構成される多段ネットワークによって接続された構成をとる。各ノードは、プロセッサ R10000 と 1MByte の 2 次キャッシュ、512MByte まで拡張可能なメモリ、PCI バス、それらを一手にコントロールするメモリコントローラからなる。本研究で使用した Cenju-4 は、32 ノード構成で、各ノードにメモリを 64MB 搭載している。

3.5.2 Cenju-4 上の並列プログラミングライブラリ

Cenju-4 は、全てのノードで同一のプログラムが動く SPMD(Single Program Multiple Data stream) 方式である。プログラミングライブラリとしては、Paralib/CJ4 という Cenju-4 独自の並列プログラミンブライブラリを利用する方法と、それに加えてハードウェアでサポートされる共有メモリシステムの DSM/CJ4 とよばれるライブラリを利用する方法と、MPI(Message Passing Interface) を利用する方法がある。

- Paralib/CJ4 は高速通信のための通信ライブラリで、機能として、他ノードのメモリを割り込みなしに読み込み・書き込みが可能な遠隔メモリ読み込み・書き込みや、他ノードに割り込みそのノード上で関数を実行させることができる遠隔関数呼び出し、メッセージ通信、バリア同期などの機能を実現している。

- DSM/CJ4 は Cenju-4 上の共有メモリライブラリでこれを用いることにより専用ハードウェアによる共有メモリシステムを利用できる。そのシステムはディレクトリベースの無効化型プロトコルで 128Byte のブロック単位でのキャッシュコヒーレンス制御を行い、そのブロックは固定である。また、

共有メモリ上のデータを自ノードのローカルメモリにキャッシュされることはなく、2nd Cache でのみ キャッシュされる。

DSM/CJ4 と Paralib/CJ4 を併用することは可能であるが、MPI と Paralib/CJ4 や DSM/CJ4 を併用することはできない。

3.5.3 Paralib/CJ4

Paralib/CJ4 での、低レイテンシの遠隔メモリアクセスとメッセージ通信は、これまでＯＳが行っていた情報の保護とイベントのユーザへの通知をハードウェアで行い、ユーザーレベル通信を実現している。図 4に遠隔メモリ読み込みの性能を測定した結果を示す。1B から 2KB まで、1B から 16B までは 1B 間隔で、16B から 64B までは 4B 間隔で、64B から 256B までは 8B 間隔で、256B から 1kB までは 16B 間隔で、それ以降は 32B 間隔で、遠隔メモリ読み込みを 10 万回繰り返し平均をとった。

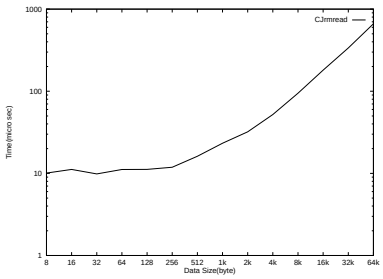

図 4: 一対一通信の性能

256B までは CJrmread の起動のレイテンシため、読み込みにほぼ同じ一体の時間がかかる。それ以降は読み込むサイズに応じて時間も増加している。

3.5.4 Cenju-4 の分類

Cenju-4 の ハードウェア DSM システムは、CC-NUMA 型の DSM システムで、以後、この環境のことを CC-NUMA/CJ4 と呼ぶ。

遠隔メモリアクセス (RMA:Remote Memory Access) は共有メモリとは別で、相手のノード番号とアドレスを用いて、任意サイズのブロック転送を行う関数である。しかしこの機構は、OS を介在せずに他ノードのメモリにアクセスできるので一種共有メモリインターフェイスととらえることもできる。この RMA 機構では、自ノードのローカルメモリのアクセスにはハードウェアのキャッシュが用いられるが、他ノードのローカルメモリへアクセスするには一旦自ノードのローカルメモリにコピーを作成し、そのコピーをアクセスする必要がある。

そこで、この RMA 機構に他ノードのメモリの内容を一貫性を保ちつつキャッシュするシステムをソフトウェアで実装し、ローカルメモリ上にキャッシングされたデータに対して、ハードウェアキャッシュを用いてアクセスさせることで、他ノードが保持する共有メモリ領域への見かけ上のアクセス速度を高速化する。以後この環境を SWC-RMA/CJ4 と呼ぶ。SWC-RMA/CJ4 はソフトウェアによる実装であるので様々に提案されているキャッシュメモリの実装が可能であり、また、Cenju-4 ではハードウェアで DSM がサポートされているので、そのソフトウェアキャッシュを用いた CC-NUMA 型 DSM との比較も可能である。

本研究では、Cenju-4 の RMA 機構を高速化すべく、ソフトウェアで様々なキャッシュシステムを実装することとした。

4 ソフトウェア ・キャッシュシステム

本ソフトウェア ・キャッシュシステムは、並列計算機 Cenju-4 の DSM の実現レベルとしてはハードウェアとソフトウェアによる実現の中間にある遠隔メモリアクセス機構に付加する形で実現する。キャッシュシステムの実現レベルはソフトウェアレベルで、ユーザプログラムは共有メモリアクセスを本ソフトウェア ・キャッシュが提供する関数呼び出しで行い、その関数がキャッシュの管理を行う。

4.1 ソフトウェア ・キャッシュの構成

Cenju-4 の遠隔メモリアクセス機構では、他ノードのローカルメモリ上のデータを指定した

サイズで自ノードのローカルメモリに読み込むことと、他ノードのローカルメモリ上にデータを指定したサイズで自ノードのローカルメモリから書き込むことができる。そこで、各ノードのローカルメモリに共有メモリ領域とキャッシュ領域をとり、他ノードの共有メモリ領域と自ノードのキャッシュ領域の間で遠隔メモリアクセスを行い、共有メモリへのアクセスをキャッシュする (図 5)。

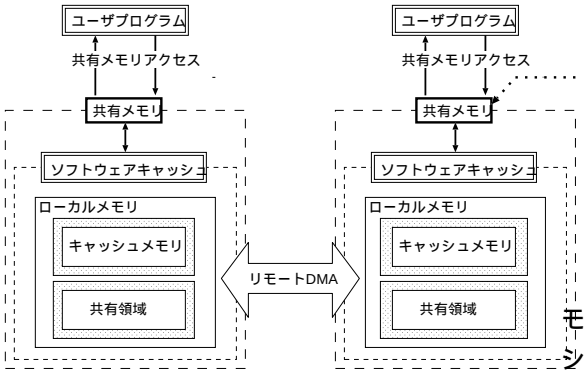

図 5: キャッシュシステムの構成

実際には、各ノードごとに共有メモリ領域とキャッシュ領域をそれぞれ 4MB づつとった。

共有メモリはユーザから直接アクセスされずソフトウェアキャッシュが提供する共有メモリアドレスを用いて関数を呼び出すことによってアクセスされる。ユーザに見えるのはそのアドレスのみで共有メモリ上のデータがどのノードに存在するかやキャッシュのどこに存在するかなどは見えない。

4.2 キャッシュの管理

本キャッシュシステムでは、共有メモリをブロックに分けて管理する。共有メモリブロックの大きさは、大きければ管理対象が少なくてすむため、管理情報のための領域が少なくてすむが、大きすぎると False Sharing が生じ性能低下を招く。小さくすると管理対象が多くなり管理情報のための領域が多く必要となる。この共有メモリブロックの大きさは、実行するプログラムごとに適当な値がそれぞれ存在するので可変とした。

共有メモリブロックには、その共有メモリブロックの状態と、どのノードがキャッシュしているかを管理する情報が必要である。共有メモリの管理情報の管理には、ノード全体の情報を１つのノードでまとめて管理する集中管理方式と、各ノードが自分の情報を管理しノード間で情報をやりとりしながら管理する分散管理方式があるが、ノードごとの負荷分散を考えて、分散管理方式をとった。

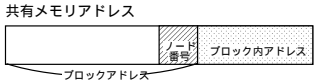

図 6: 共有メモリアドレス

キャッシュシステムは全ノードで１つの共有メモリアドレス (図 6) をユーザに提供する。キャッシュシステムは、共有メモリアドレスを変換することにより、共有メモリブロックを持つノードとそのノードのどこにあるのかを判断する。また、この変換方法を変更することにより、ノードへの共有メモリブロックの分散の方法を変更することができる。現在は、全ノードに共有メモリブロックごとにサイクリックに分散配置されている。

共有メモリブロックをどのノードがキャッシュしているかを管理するのは、その共有メモリブロックをローカルメモリに持つノードのローカルメモリにディレクトリを置くことにより管理する。実装に使用した Cenju-4 の最大ノード数が 32 であるため、それぞれのノードの共有メモリブロックごとに 32bit のディレクトリタグをローカルメモリに置いた。遠隔メモリアクセスによりそのディレクトリタグにはどのノードからもアクセスできる。また、その共有メモリブロックが最新であるかどうかを管理するタグもコヒーレンスの維持のために必要である。

キャッシュディレクトリの管理の仕方は、ソフトウェア DSM の場合、キャッシュのヒット率を上げるためフルアソシアティブ方式をとり、共有メモリアドレスをキーとしたテーブルで管理するものが多く、また、ハードでは実現困難な

高度なリプレース手法を用いつことによりキャッシュのヒット率を高くしている。

本ソフトウェアキャッシュでは、ヒット率の向上よりもキャッシュヒット時のレイテンシを少なくするために、構成が簡単なダイレクトマッピング方式で、キャッシュディレクトリを管理する。ソフトウェアキャッシュは、そのキャッシュブロックがどの共有メモリブロックをキャッシュしているのかを、ブロックアドレスを記憶しておくことにより判断する。また、コヒーレンスの維持のためキャッシュブロックの状態を管理するタグが各キャッシュブロックごとに必要である。

4.3 コヒーレンスプロトコル

コヒーレンスプロトコルは、

- メモリ更新方式
- キャッシュ間データ転送方式
- 一貫性制御方式

によって分類できる。

1. メモリ更新方式
   共有メモリへの書き込みをキャッシュのみに行うのが、ライトバック方式で、キャッシュのみではなく共有メモリへも書き込みを行うのが、ストアスルー方式である。ストアスルー方式では、常に最新のデータが共有メモリにあるため共有メモリの管理が容易だが、書き込み時に常に共有メモリへもアクセスするため、書き込みに時間がかかる。そのため、書き込みをバッファにためる方式などがある。ライトバック方式では、最新のデータが共有メモリにないため共有メモリの管理は複雑になるが、書き込みはキャッシュのみになされるため書き込みのレイテンシは低く抑えられる。

   本キャッシュシステムでは、プログラムからの呼び出しによって共有メモリアクセスが行われるため、バッファにためてから書き込む動作は実現が困難である。そのためストアスルー 方式では書き込みのたびに他ノードへの遠隔メモリアクセスが起こり、ライトバック 方式に比べ非常に低速であるので、ライトバック 方式とした。

2. **Dirty Shared** 状態の有無
   共有メモリと異なる最新のブロックが複数のノードで共有されている状態を Dirty Shared 状態と呼ぶ。Dirty Shared 状態を認めないプロトコルでは、共有メモリと異なる最新のブロックは 1 つのノードのみが持つことができるため管理は容易だが、最新のブロックに別のノードがアクセスし、そのブロックをノード間で転送するときには、その最新ブロックを共有メモリに書き戻さなければならないため時間がかかる。一方、Dirty Shared 状態を認めると、認めないプロトコルに比べ複雑な管理が必要となるが、高速に実行できる可能性がある。本キャッシュシステムでは、まず管理が容易な Dirty Shared 状態を認めないプロトコルを実装した。

3. 一貫性制御方式
   ある共有メモリブロックを複数のノードでキャッシュしているとき、1 つのノードがそのキャッシュに書き込みを行ったとき、残りのノードのキャッシュを無効化するのが Write Invalidate 方式 (無効化型) で、残りのノードのキャッシュにその新しいデータを送りそのキャッシュを更新するのが Write Update 方式 (更新型) である。更新型には、そのブロック全体を送信する方法とブロック内の変更があった部分のみを送信する方法がある。無効化型は、1 つの共有メモリブロックに 1 つのノードからの書き込みが多く起こる場合に有効で、更新型は、1 つの共有メモリブロックに複数のノードからの読み込み ・書き込みが多く起こる場合に有効である。

   本キャッシュシステムでは、無効化型と更新型の両方の方式を提供している。更新型は共有メモリブロックが大きい場合のことも考慮し、更新があった部分のみを送信する方法をとっている。送信をただちに行いバッファリングは行っていない。

このようなコヒーレンスプロトコルの実装に際して、他ノードのキャッシュブロックの無効化・更新が必要であり、そのために他ノードのローカルメモリの変更が必要となる。遠隔メモリアクセスによる変更も可能であるが、Cenju-4では遠隔メモリアクセスによる他ノードのローカルメモリへの書き込みはいつ他ノード上で終了するかが保証されていない。そこで、本ソフトウェアキャッシュでは、Paralib/CJ4の遠隔関数呼び出し機能を用いて他ノードへ割り込み、無効化・更新を実現する方法を採用した。この遠隔関数呼び出し機能は遠隔メモリアクセスに比べ一回の遠隔関数呼び出しの実行時間は$328\mu sec$かかり時間がかかる(遠隔メモリアクセスは256B読むのに$11\mu sec$)。そのため無効化・更新の実行には非常に時間がかかる。

4.4 排他制御

ある共有メモリブロックに同時に複数のノードが書き込むと、メモリコンシステンシが維持できなくなるため排他制御を行わなければならない。Paralib/CJ4の遠隔関数呼び出し機能では、遠隔関数呼び出しで呼び出された関数が実行されているノードに他のノードが遠隔関数呼び出しで割り込むことはできない。また、自ノードに遠隔関数呼び出しで割り込むことも可能である。この機能を利用してロック機構を実装した。

ロック変数は、通常-1でノード番号のときはそのノードに確保されているとする。あるノードのロック変数を確保したいとき、

```
void lock(int p_num) {
  if(lock==-1) lock=p_num;
}
```

という関数を用意し、

1. まずそのノードから遠隔メモリアクセスでロック変数を読み込む。
2. ロック変数が他ノードに確保されていたら解放される(-1になる)まで読み込む。
3. -1なら、変数のあるノードに遠隔関数呼び出しで割り込み、関数lock(自ノードの番号)を実行させる。
4. それが終了したら、遠隔メモリアクセスでロック変数を読み込む。
5. 自ノードの番号になっていたらロックを確保。そうでなければロックが確保できるまで1.から5.までをくりかえす。

このようにしてロック機構を実現している。

4.5 ユーザインターフェイス

ユーザは本ソフトウェアキャッシュで提供する共有メモリへのアクセスを行うために、以下の関数を使用する。ユーザからは直接共有メモリへのアクセスはできない。

- void *sdsm_malloc(size_t n);
  共有メモリ領域にnバイト確保しその先頭へのポインタを返す。
- void store_sdsm(int to,int *from,size_t size);
  共有メモリ領域のアドレスtoにローカルメモリのアドレスfromからsizeバイトを書きこむ。
- void load_sdsm(int from,int *to,size_t size);
  共有メモリ領域のアドレスfromからローカルメモリのアドレスtoにsizeバイトを読み込む。
- void sdsm_lock(int *lock_num);
  共有メモリ上のロック変数lock_numに関してロックを確保する。
- void sdsm_unlock(int *lock_num);
  共有メモリ上のロック変数lock_numのロックを解放する。
- void sdsm_barrier();
  全てのノード間で同期をとる。

通常DSMシステムでは、共有メモリへのアクセスとローカルメモリへのアクセスは、ユーザは区別することなく可能であるが、本キャッシュシステムでは、ソフトウェアで実行時にルーチン呼び出しでキャッシュシステムを実現しているのでこのような形になっている。そのため、プログラムには多少の書き換えが必要となる。

4.6 ソフトウェア ・キャッシュの動作

本キャッシュシステムは、ユーザが共有メモリへのアクセスを行う場合に関数を呼び出すことにより起動される。起動されたキャッシュシステムは、引数として与えられる共有メモリアドレスから、その共有メモリブロックを持つノード番号、共有メモリブロックの番号を調べる。それにより自ノードのキャッシュディレクトリを検索し、キャッシュにそのブロックが存在するかどうかを調べる。キャッシュに存在した場合そのブロックの状態を調べる。共有されていない場合と複数のノードで共有されていても読み込みの場合は、ライトバック方式なのでキャッシュにアクセスし終了する。複数のノードで共有されている共有メモリブロックへの書き込みは、そのブロックに関するロックを確保しコヒーレンスプロトコルが無効化ならキャッシュしているノードのキャッシュを無効化し、コヒーレンスプロトコルが更新ならキャッシュしているノードのキャッシュに読み込ませ、ロックを解放する。キャッシュに存在しなかった場合、そのブロックに関するロックを確保し、共有メモリに最新のデータがない場合は、最新をキャッシュに持つノードから読み込ませ、それを自ノードのキャッシュに読み込み、アクセスしロックを解放し終了する。アクセスが書き込みの場合でその共有メモリブロックをキャッシュしているノードが他にある場合は、ロックを解放する前に無効化又は更新をする。このようにして共有メモリへのアクセスとキャッシュの管理を実現している。

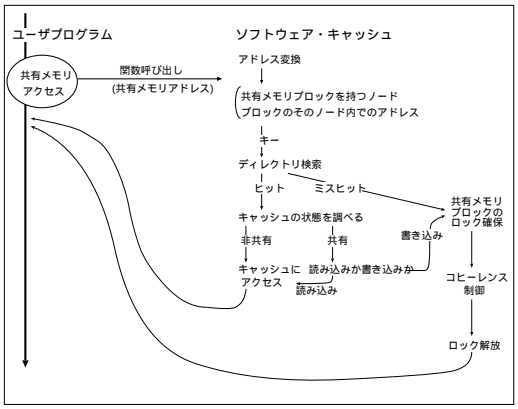

図 7: ソフトウェア ・キャッシュの動作

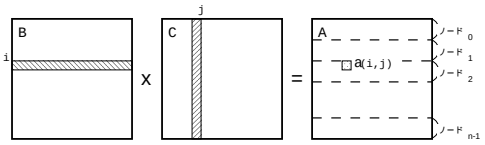

図 8: 行列積の並列実行

5 ソフトウェア ・キャッシュ評価

5.1 行列積を用いた評価

$n \times n$ の正方行列の積を計算する。正方行列 B と Cの積 A を計算するものとする。行列積の並列実行の方法を、図 8に示す。答えの行列 A の n 個の行を最大ノード数で割った数ごとに分割し、各ノードの担当範囲とし並列に実行する。

5.1.1 キャッシュブロックサイズによる比較

行列積は、答えの行列 A の要素 a_{ij} の計算には、行列 Bの i 行と行列 Cの j列が必要となる。従って、通信の単位であるキャッシュのブロックサイズが大きければ大きいほど、キャッシュミスヒットが起こらず、高速に実行できると考えられる。しかし、ブロックサイズが大きすぎると、不必要なデータを読み込むことになり性能低下が考えられる。

ブロックサイズは、128B, 256B, 512B, 1kB, 2kB, 4kB の場合、行列の大きさは 128×128 256×256 512×512 の正方行列で要素は double 型、プロトコルは無効化型についてノード数 16 で計測した。結果を図 9 に示す。

それぞれのノードの計算する要素の数は、128×128 の場合 1024 個で 8kB、256×256 の場合 4096 個で 32kB、512×512 の場合 16384 個で 128kB となり、どの場合もブロックサイズの整数倍となり、同じブロックに複数のノードが書き込むことはない。よって、False Sharing が生じることはなく、ブロックのサイズを大きくすればするほど高速に実行できた。

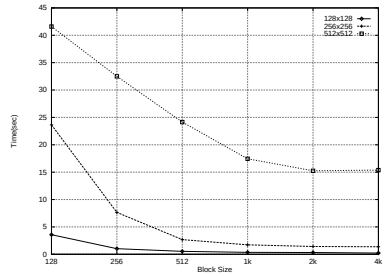

図 9: ブロックサイズによる実行時間の比較 (秒)

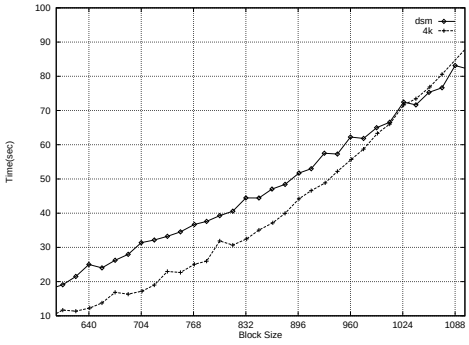

図 10: CC-NUMA/CJ4 との比較

5.1.2 プロトコルの違いによる評価

無効化型と更新型の 2 つについて比較する。ブロックサイズは 128B, 256B, 512B, 1kB, 2kB, 4kB 行列の大きさは 128×128, 256×256, 512×512 の正方行列で、ノード数 16 で計測した。その結果を表 1に示す。

表 1: プロトコルによる比較 (秒)

		block size					
size	プロトコル	128B	256B	512B	1k	2k	4k
128×128	無効化	3.23	1.02	0.54	0.35	0.23	0.22
128×128	更新	3.29	1.07	0.63	0.41	0.25	0.22
256×256	無効化	23.64	7.65	2.68	1.72	1.42	1.37
256×256	更新	26.24	8.21	2.82	1.93	1.63	1.44
512×512	無効化	41.60	32.50	24.15	17.44	15.25	15.38
512×512	更新	49.07	45.28	35.19	18.83	15.86	15.40

プロトコルによる差はほとんどない。ブロックサイズによる比較と同じく同じブロックに複数のノードが書き込むことはないため、無効化・更新は共に起こらないためと考えられる。

5.1.3 CC-NUMA/CJ4 との比較

CC-NUMA/CJ4 と SWC/RMA を比較する。SWC/RMA のブロックのサイズは大きければ大きいほど行列積では高速に実行できるので 4kB とし、行列サイズ $n \times n$ の n を、64 から 1152 まで変化させ、ノード数 16 で計測した。

ハードウェアとソフトウェアキャッシュの速度が逆転する、1000×1000 付近の実行速度を図 10に示す。CC-NUMA/CJ4 による実行時間が dsm で、4kB のブロックサイズの SWC/RMA による実行時間が 4k である。

CC-NUMA/CJ4 ではキャッシュコンフリクトが起こり異常に遅くなってしまう場合があった。そのため、キャッシュコンフリクトが起きた場合は、行列サイズを少し変更し測定した。キャッシュコンフリクトが起きた場合との実行時間の差を図 11に示す。本来、SWC/RMA は 4kB と大きな単位で通信を行っているため、128B で行う CC-NUMA/CJ4 より通信回数が減少するため、高速に実行できるはずである。しかし、共有メモリアクセスが関数呼び出しで実現されており、また共有メモリアドレスの変換、キャッシュディレクトリアクセスなどの様々なオーバーヘッドがあるため、行列サイズが小さいときは高速に実行できない。しかし、行列サイズが大きくなると、それらのソフトウェアのオーバーヘッドよりも CC-NUMA/CJ4 の通信回数の増加による性能低下が大きくなるため、SWC/RMA の方が高速に実行されている。

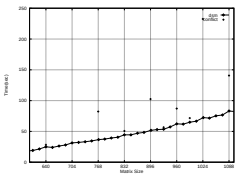

図 11: キャッシュのコンフリクト状態

5.1.4 考察

行列積ではロードの回数に比べストアの回数が絶対的に少ない。また分割の方法により１つ

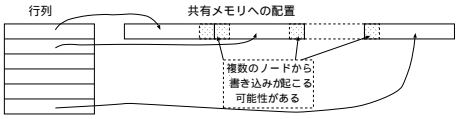

図 12: 行列積の特徴

のブロックに同時に複数のノードが書き込むことはない。このような場合、ブロックサイズを大きくしても無効化 ･更新のための通信は少なくてすむため、False Sharing は起こりにくい。つまり、更新するための意味はあまりないが無効化 ･更新自体が起きないためそれほどプロトコルによる差が現われなかった。

ブロックサイズを大きくすると、一度に読み込んでくるデータも増えるため、キャッシュミスヒットが減り、また通信回数も減少するため、高速に実行できる。

CC-NUMA/CJ4 と比較すると、CC-NUMA/CJ4 はコヒーレンスの単位が 128B と小さいため SWC/RMA の大きな単位と較べると通信の回数が多くなる。そのため問題サイズが大きくなり通信回数が増えると SWC/RMA の方が高速に実行できるようになる。

5.2 LU 分解を用いた評価

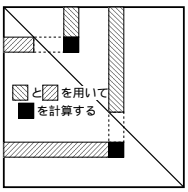

図 13: データ参照

LU 分解 [3] には、内積ガウス法、外積ガウス法、クラウト法があるが、今回は、内積ガウス法を用いて評価した。内積ガウス法は、図 13のように、a_{ij}の計算には、a_{ik}と$a_{kj}(k = 1, 2, ..., \min(j, j))$が必要である。そこで、図 14 のように、現在 a_{ij}までの計算が終了しているものとすると、$j+1$ 列は、a_{i-1j}まで計算が可能となる。行列のメモリへの配置は今回実装に用いた C 言語では、行方向に順番に配置されるので行方向にまとめて計算した方がキャッシュヒット率が向上すると考えられる。ある程度行方向に列をまとめて１つのブロックとし計算することとした。

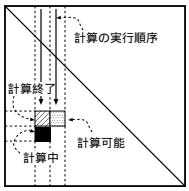

図 14: 実行可能

各ノードに、自分が現在計算を終えた列の値を持たせる。各ノードは、自分の計算を終えた列が隣のノードの計算を終えた列より小さければ、次の列が計算可能になる。そこで、次の列の計算を始める前に、隣の列が進むのを待つ。結局、図 15のように並列に実行する。

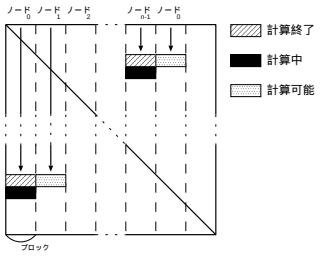

図 15: 並列実行

5.2.1 キャッシュブロックサイズによる比較

ブロックサイズ 128B, 256B, 512B, 1kB, 2kB, 4kB の場合について、512×512 の LU 分解を 4 ノードで並列実行の単位を 8, 16, 32, 64, 128 で実行し、並列実行の単位とキャッシュブロックサイズとの関係を調べる。

行列の各要素は double 型なので 8B である。並列実行単位が 8, 16, 32, 64, 128 ということ

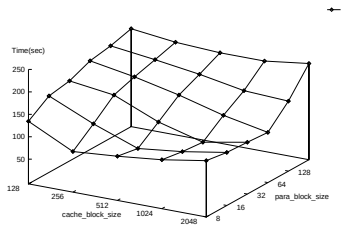

図 16: キャッシュブロックサイズと並列実行単位

は、その大きさは 64B, 128B, 256B, 512B, 1kB ということである。結果より、キャッシュブロックサイズ 128B 並列実行単位 16 とキャッシュブロックサイズ 256B 並列実行単位 32 の場合が高速に実行されている。このとき並列実行単位の大きさは行列の要素の大きさが double 型のため、ブロックサイズと同じになっている。つまり、ブロックサイズと同じ大きさで並列実行することにより不必要なコヒーレンス制御のための通信が必要なく、高速に実行できると考えられる。しかし、キャッシュブロックサイズ 1kB 並列実行単位 128 の時などは遅い。これは、並列化の方法により並列実行の単位が大きくなると並列実行されない部分が増加するためであると考えられる。並列実行の単位が 128 のときは 1kB のブロックが最後の 1 ノードで並列でなく計算されるが、並列実行の単位が 64 のときはそれが 512B ですむ。また、ブロックサイズが大きくなることにより、そのブロックが複数のノードに共有される。並列化の単位が 32 ブロックサイズが 256B のときは複数のノードが同じ 1 つのブロックに書き込むことはないが、ブロックサイズを 512B とすると 1 つのブロックに並列実行の単位が 2 つ入ることになり、2 つのノードが 1 つのブロックに書き込むことになる。そのため、無効化 ・更新のための通信が生じ速度が低下すると考えられる。128B より 256B の大きさを合わせた場合の方が高速に実行されているのは、Cenju-4 の通信性能 (図 4) より、256B までの通信は時間が同じでその分 256B の方が高速になったのだと考えられる。

結果から考えて、ある程度ブロックサイズと並列実行の単位は小さい方が良い。並列実行の単位は 32、ブロックサイズは 256B が最適と考える。

5.2.2 プロトコルの違いによる評価

ブロックサイズでの比較で最も高速に実行できたブロックサイズ 256B 並列実行の単位は 32 の場合、1 つの共有メモリブロックが複数のブロックから書き込まれるということはないため、プロトコルの違いによる性能差を計測することができない。そこで、ブロックサイズ 256B 並列実行の単位は 16 の場合について、ノード数 4、行列のサイズが 128×128, 256×256, 384×384, 512×512 の LU 分解を無効化と更新それぞれ実行し、LU 分解におけるプロトコルの違いによる評価を行った。結果を表 2に示す。

表 2: プロトコルによる比較 (秒)

	プロトコル	
問題 size	無効化	更新
128×128	3.64	5.88
256×128	19.63	28.01
384×384	55.55	72.67
512×512	121.61	149.37

いすれの場合も無効化プロトコルを用いた方が高速に実行された。本来内積ガウス方を用いた LU 分解の並列化方法では、対象となる要素の列方向と行方向の前の要素を使って計算するため、データの再利用性が高い。このため更新プロトコルの方が高速に実行できるはずであった。

しかし、SWC/RMA では更新のコストが高いため、更新が頻発する場合はそこがオーバーヘッドとなり高速に実行できかったものと考えられる。

5.2.3 CC-NUMA/CJ4 との比較

これまでの結果より最も高速に実行できる組み合わせのブロックサイズ 256B 並列実行の単位は 32 について、128×128, 256×256, 384×384, 512×512 の LU 分解を無効化型の SWC/RMA

と CC-NUMA/CJ4 で実行した。結果を表 3に示す。

表 3: ハードとソフトの比較 (秒)

	方式	
問題 size	ハード	無効化
128×128	0.27	1.82
256×256	2.05	11.25
384×384	6.84	35.30
512×512	19.72	85.07

いずれの場合も CC-NUMA/CJ4 を用いた方が高速に実行された。SWC/RMA は、図 10のように通信の単位が大きくてもコヒーレンスの維持のための制御が起きない場合は CC-NUMA/CJ4 に比べ高速に実行できるが、コヒーレンスの維持のための制御が起き、通信の単位が小さい LU 分解のようなアプリケーションでは、CC-NUMA/CJ4 よりも遅くなる。

5.2.4 考察

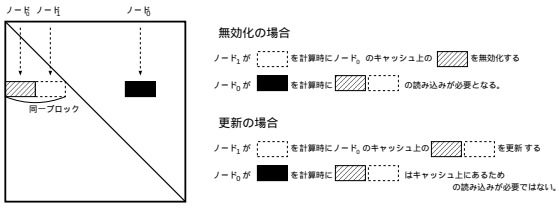

図 17: LU 分解にデータ参照

内積ガウス法による LU 分解では、要素の列方向と行方向の前の要素を使って、計算を実行する。そこで、図 17のように、ノード 0 が計算し、キャッシュに残っているデータを無効化の場合ノード 1 がその隣を計算する時に無効化してしまい、後でノード 0 がそれを用いて計算する場合共有メモリから読み込まなければならない。しかし、更新する場合ノード 1 が計算した結果もキャッシュにあるため共有メモリアクセスが減少する。このように、内積ガウス法による LU 分解ではデータの再利用性が高いため、更新型のプロトコルによる実行が有効であり、そのような結果がでるはずだが、更新プロトコルがキャッシュが書き変わるごとに更新する方法で、遠隔関数呼び出しを用いて更新するため非常に遅く、その有効性を生かすことができなかったと考えられる。

また、この並列化の方法ではブロックサイズと並列実行単位の組み合わせが、128B で 16、256B で 32、512B で 64 の時のように、ブロックサイズを並列実行単位に合わせることにより、必要でないデータをロードしてしまう無駄な通信をなくすことができ、さらに無効化や更新が発生しないため高速に実行できる。

6 おわりに

本研究では様々なキャッシュシステムを評価するという目的のために、並列計算機 Cenju-4 上のキャッシュシステムのない共有メモリアクセス機構に、ソフトウェアでキャッシュシステムを実装しそれを評価した。

試したキャッシュシステムは、ブロックサイズが可変で無効化型と更新型のコヒーレンスプロトコル機構を備えたものである。

まず、キャッシュシステムを実装することにより、ほぼ完全な共有メモリ型のプログラミング環境を提供することができ、キャッシュシステムのない共有メモリアクセス機構であるリモートメモリアクセス機構でのプログラミングを容易にすることができた。

また、このキャッシュシステム上でアプリケーションを動かし性能評価し、ブロックサイズを大きくし通信のサイズを大きくすることにより、通信のサイズが一定で小さいハードウェアよりも高速に実行できる可能性があることが行列積による性能評価で示せた。一方、LU 分解による性能評価によりアプリケーションごとのコヒーレンスプロトコルの切り換えが有効であることが示すつもりだったが、Cenju-4 における他ノードへの割り込みのコストが高いため示すことができなかった。

ハードウェアによる DSM と比較すると、ソフトウェアは全般的に性能が低く特殊な条件でしか上まわることができなかった。それは、本

研究のソフトウェアキャッシュでは、キャッシュコヒーレンスとメモリコンシステンシの維持のために他ノードへ割り込んでおり、そのために使用している他ノードへの割り込み機構のコストが高いためであると考えられる。従って、割り込みを使わないソフトウェアキャッシュの実装を行なうか、もしくはより高速な割り込み手段を実現することができれば、コヒーレンスプロトコルの切り換えの有効性を示し、もう少しハードウェアに迫る性能が出せるはずである。また、本キャッシュシステムでは、ロックを解放した時点でのメモリコンシステンシを保つモデルを採っていて、他ノードへの割り込みを行なわなければならなかったが、ロックを確保した時点でのメモリコンシステンシを保つモデルを採用すると、開放時にコンシステンシを保っている必要がないため、遠隔メモリアクセスによる書き込みで無効化・更新などを行なうことができ、他ノードへの割り込みが必要なくなる。この手法をとることにより更なる高速化が可能である。

更に、評価したキャッシュシステムは少なく、様々なキャッシュシステムとは言えない結果となってしまった。また、評価に用いたアプリケーションも行列積と内積ガウス法による LU 分解のみであった。

今後は、更なる性能向上とより多くのアプリケーションでの評価を実現したい。

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚な謝意を表します。

また、本研究に関して適切な御指導を賜った森眞一郎助教授、五島正裕助手に心から感謝致します。

最後に、日頃から御助力頂いた京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝致します。

参考文献

[1] 加藤 研他: 並列コンピュータ Cenju-4 のユーザーレベルメッセージ通信機構, *JSPP*, Vol. 99, No. 6, pp. 7–14 (1999).

[2] 白木 長武他: 高並列計算機 AP1000+のメッセージハンドリング機構, *JSPP*, Vol. 95, No. 5, pp. 233–240 (1995).

[3] 津田孝夫: 数値計算プログラミング, 岩波書店 (1988).

[4] Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, Vol. 29(2, pp. 18–28 (1996).

[5] 細見 岳生他: 並列計算機 Cenju-4 の分散共有メモリ機構, *JSPP*, Vol. 99, No. 6, pp. 15–22 (1999).

[6] NEC: Cenju-4 システム概要書/設置計画説明書 (1997).

[7] 大西 淑雅他: 分散共有メモリモデルに基づく HPC 環境の高性能実装と性能評価, 情報処理学会論文誌 (1995).

[8] 中條 拓伯他: 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 (1995).

[9] 富田眞治: 並列コンピュータ工学, 昭晃堂 (1996).

[10] 富田眞治: 計算機アーキテクチャ1, 丸善 (1994).

[11] 湯浅太一, 安村通晃, 中田登志之: はじめての並列プログラミング, 共立出版 (1998).