ソフトウェアからのヒントを利用する
キャッシュ一貫性制御方式の提案と
その予備的性能評価

山本 考伸

平成11年2月12日

1 はじめに

拡張性のある並列計算機を構成するためには、主記憶を物理的に分散配置する必要がある。分散配置された主記憶を持つ並列計算機においては、プロセッサ間通信のコストが並列計算機の性能に大きな影響を及ぼす。

近年の並列計算機においては、プロセッサ間の通信機構がユーザに解放され、通信のためにシステム・コールが不要となり、1回の通信にかかるソフトウェア・オーバヘッドが大きく低減された[1, 2]。また、コンパイル技術の向上や最適アルゴリズムの探索の結果、多くの並列アプリケーションに出現するプロセッサ間通信における1回の通信データサイズは、ソフトウェアオーバヘッドに耐え得るくらい大きくすることが可能であるという報告もなされている[3]。このような流れを受けて、ハードウェアが提供する共有メモリによってプロセッサ間通信を行うシステムの重要性は薄らいできている。

しかし、問題の性質上1回のプロセッサ間通信におけるデータサイズを大きくできないなど、ソフトウェア・オーバーヘッドが無視できないアプリケーションも存在する。そのようなアプリケーションを実行する際には、ハードウェアが提供する高速なデータ検索機能や有効性管理機能等によって、かなりの高速化が期待できる。

しかし、ハードウェアDSMだけでは満足の行く性能が得られないアプリケーションが存在するのも事実である。現在のハードウェアDSMにおいては、一貫性制御の単位及び方式がほぼ固定されているために、共有データの性質に合わせた処理を行うことは困難である。例えば、明らかにデータはその消費者の近傍にあった方がよい。しかし、現在のハードウェアDSMでは、データは消費者よりむしろ生産者のキャッシュの上に残る。アプリケーションによっては、消費者が生産者の所までデータを取りに行くレイテンシが支配的になる。そのようなアプリケーションにおいては、ソフトウェアからの指示によってデータを消費者側にあらかじめ転送して、消費者側のレイテンシを低減することによる性能向上が期待できる。

つまり、多くのアプリケーション中に出現するプロセッサ間通信を効率よく実現するためには、

- ハードウェアDSMの持つ高速性
- ソフトウェアからの指示による柔軟性

の双方とも必要である。

ソフトウェアからの指示によるデータ転送機能とハードウェアDSMによるデータ転送機能の両方を持つCenju-4の様な計算機も実在するが、各メモリオブジェクトに対してどちらか一方の転送しか利用できなくなっている。そのため、ハードウェアDSMで管理しているメモリオブジェクトに対してソフトウェアからの転送指示を行いたい場合には、自プロセッサのメモリ上でのコピーが必要である等の問題がある。

このような背景から、本稿では上述の高速性と柔軟性を両立するために、ソフトウェアからのヒントを利用するキャッシュ一貫性制御方式を提案する。各アプリケーションにおいて共有メモリオブジェクトにアクセスする際には、そのメモリオブジェクトの性質に合わせた一貫性制御方式をア

プリケーションから指定することができる。例えば、生産した共有メモリオブジェクトの消費者を指示して消費者の近傍にデータを保持したり、生産者のキャッシュにデータを保持しない事を指示したりできる。ハードウェアDSMが一貫性制御を行うメモリ空間に対してソフトウェアから指示を出すことができるので、Cenju-4のような問題も生じない。

以下本稿では始めに、ハードウェアDSMによる高速なプロセッサ間通信が有効である問題が存在し、また一方でソフトウェアからの指示による柔軟なプロセッサ間通信が求められる問題も存在することを示し、各問題においてハードウェアDSM及びソフトウェアからの指示の各機能が有効であることを、Cenju-4を用いた実測値によって示す。続いて、本稿で提案する、ソフトウェアからのヒントを利用する一貫性制御方式の実装法として、アドレス空間を利用した方式について説明する。最後に、現在の実装方式の問題点等に関する考察を行う。

2 背景

本章では始めに、並列計算機におけるプロセッサ間通信の管理者がハードウェアであるかソフトウェアであるかという点に注目し、プロセッサ間通信方式を、ハードウェアDSMと、ソフトウェアからの指示によって行う物の2種類に大別する。また、ハードウェアDSMの持つ高速なデータ転送管理機能とソフトウェアからの指示に基づく柔軟なデータ転送管理機能の双方とも、それぞれ有効となる問題が存在することを示す。

まず2.1節において、ハードウェアDSMとソフトウェアから通信指示を行うシステムの定義及び分類を行う。続いて、2.2節においてハードウェアDSMが有効であると思われる問題の一般的性質について述べ、2.3節においてはソフトウェアからの通信に関する指示が有効であると思われる問題の一般的性質について述べる。

2.1 プロセッサ間通信方式の分類

本節では、プロセッサ間通信の管理者がハードウェアであるかソフトウェアであるかという観点から、並列計算機におけるプロセッサ間通信方式を分類する。本稿では大規模に拡張可能な並列計算機を対象としているために、主記憶は物理的に分散配置されていることを前提とする。

2.1.1 ハードウェアDSM

本稿では、一貫性制御をハードウェアによって行う並列計算機を、ハードウェアDSMと呼ぶ。共有空間に対するメモリアクセスに応じて必要な処理をハードウェアが自動で行うために、ソフトウェアからはプロセッサ間通信時の物理的なデータ転送を意識する必要がない。以下にハードウェアDSMの特徴を示す。

- システムの拡張性を増すために、各主記憶領域の一貫性制御に必要となるディレクトリを分散管理する。

- 各要素プロセッサは共有空間である主記憶領域を自プロセッサの近傍にキャッシュできる。キャッシュには一貫性制御に必要なタグが付加されている。

- ソフトウェアからのメモリアクセス時に、システムが提供しているメモリモデルに応じて適当な値を返すために必要となる一連の処理の流れを、本稿では一貫性制御方式と呼ぶ。

  ハードウェアDSMにおける一貫性制御方式は、共有空間に書き込みがあった時に当該領域のキャッシュコピーを全て無効化する方式write-invalidateや、キャッシュコピー全てを新しい値に書き換える方式write-update等の決まりきった処理を行う方式であり、並列計算機毎に方式が固定しているものが多い。

- 一貫性制御は、ブロックと呼ばれる固定長を単位として実行される。

- 共有空間に対する一貫性制御が必要である時に、実際にハードウェアが一貫性制御処理を起動する時期をソフトウェアから指定することはできない。

2.1.2 ソフトウェアからの指示によるプロセッサ間通信

プロセッサ間通信をソフトウェアからの指示によって行う方式としては、ソフトウェアがメモリ管理をすることによって共有メモリプログラミング環境を実現するソフトウェア DSM と、アプリケーションから直接プロセッサ間通信を指示するメッセージ通信が挙げられる。以下にそれぞれの方式について述べる。

- ソフトウェア DSM

  ハードウェアが提供する共有メモリを持たないシステムにおいて、ソフトウェアがメモリ管理することによって共有メモリプログラミング環境を実現する。プロセッサ間で転送するデータオブジェクトを単位として一貫性制御を行う方式と、ページテーブルを利用して一貫性制御を行う方式がある。

  ページテーブルを利用した一貫性制御方式

    アドレス変換の単位であるページを一貫性制御の単位とする。あるページ内のデータに対する一貫性制御が必要となった時には、ページ内のデータを共有するプロセッサのそのページに対する TLB エントリ及びページテーブルを無効化し、ページテーブル中にそのページに対する一貫性制御が必要であることを記録する。後にそのプロセッサ上のソフトウェアがそのページ内のデータにアクセスした時にはページフォールトが発生し、データの転送等の必要な処理が実行される。

    TLB ヒット時には当該ページ内のデータの有効性を、データにアクセスする度にソフトウェアで調べる必要がないので、ソフトウェアオーバヘッドが軽減される。

    共有データオブジェクトのサイズがページサイズより小さい時にはフォールスシェアリングが発生し、無駄な一貫性制御処理が頻発して性能が悪化する。

    ページテーブルを利用したソフトウェア DSM が提供するメモリモデルとしては Release Consistency [4] が有名であり、一貫性制御処理を実行するタイミングによって、Eager Release Consistency と Lazy Release Consistency に分類される。本稿ではこれらの方式の詳細は省略する。

  共有オブジェクト毎に管理する方式

    共有空間内のデータオブジェクト毎に、一貫性制御に必要となる情報を管理する方式である。一貫性制御の単位はメモリオブジェクトのサイズである。共有空間内のメモリオブジェクトにアクセスする際には、ソフトウェアによって当該データを自プロセッサをキャッシングしているかどうか、キャッシングしたデータが有効かどうか、等の判断を行う必要がある。オブジェクトデータのサイズが大きくない場合には特に、このソフトウェアオーバヘッドが問題となる。

- メッセージ通信

  アプリケーション中で共有アドレス空間の存在を意識せず、プロセッサ間での物理的なデータ転送をソフトウェアから指示する。

  ストリーム通信を実現するバッファを用いた通信方法と、送受信プロセッサ間でお互いのアドレス空間に直接データ転送する通信方法に分類できる。以下にそれぞれの特徴を示す。

  システムバッファを用いた通信 システムがストリーム管理を保証する領域を用いてプロセッサ間のデータ転送を実行する。従来の並列計算機においてはストリームの管理をシステムレベルで実現していたため、プロセッサ間通信の度にシステムコールが必要であった。最近の並列計算機においては、ユーザの権限でストリーム通信を実現できるものも出現している [20]。

  リモートメモリ転送 送受信プロセッサ間でお互いのアドレス空間に直接アクセスすることができる。送信プロセッサが受信プロセッサのメモリ上にデータを書き込むリモートメモリライトや、受信プロセッサが送信プロセッサのメモリ上からデー

タを読み出すリモートメモリリードが有名である。

2.2 ハードウェア DSM の必要性

最近の並列計算機においては、プロセッサ間通信機構がユーザに解放され、ソフトウェアからの通信指示コストが大幅に削減された。また、コンパイル技術の向上や最適並列アルゴリズムの開発等の成果によって、多くの並列問題における 1 回の通信データ量はソフトウェアオーバヘッドを隠蔽できるほど十分大きくできるようになったと考えられている。

しかし、問題の性質上 1 回の通信データサイズを大きくできないような問題が存在する。そのような問題においては、ハードウェア DSM の持つ高速な転送管理機構及び有効性管理機構が効率的であると思われる。

ハードウェア DSM が有効であると思われる問題としては、

- 消費時のデータ転送コストが重要である問題
- 消費時のデータ転送を行う問題の中でも特に、消費データの動的な有効性管理が必要となる問題

等が挙げられる。以下、2.2.1節において消費時のデータ転送コストが重要である問題についての詳細を述べ、2.2.2節において消費データの動的な有効性管理を行う問題の詳細を述べる。

2.2.1 消費時のデータ転送

プロセッサ間で転送されるデータは、消費時のレイテンシを削減するために、データ生産時に消費者側に転送した方がよい。しかし、データ生産時に消費者を指定できないアプリケーションや、消費データが大きいアプリケーションにおいては、消費時にデータ転送する必要がある。

前述したように、消費時にデータ転送する必要のある問題は

- 消費データの有効性を、プログラムの実行の流れで保証できる問題
- 消費データの有効性を、消費時に動的に確かめる必要のある問題

に分類できる。以下本節では、プログラムの実行の流れで消費データの有効性を保証できる問題におけるハードウェア DSM の有効性を示す。動的な有効性管理が必要な問題に関しては、2.2.2節において述べる。

データ消費時には、

- 消費データをシステム内で検索する
- 消費データを保持するプロセッサから自プロセッサへのデータ転送を指示する

処理のコストがかかる。また、消費データを再利用する可能性のある問題においては、転送してきたデータをキャッシュすることによって、後続のメモリアクセス時のレイテンシを低減することができる。この場合、キャッシュにどのデータを保持しているのか管理する必要がある。

ハードウェア DSM においては高速なデータ検索、転送、管理機能が実現されているため、上記のようなデータ転送を含む問題におけるハードウェア DSM の有効性は非常に高いものと思われる。

このような性質のデータ転送を含む問題の具体例として、Barnes Hut [6] の第 2 フェーズが挙げられる。Barnes Hut の第 2 フェーズ を実行する際にハードウェア DSM が実際どれほど有効であるかを検証するため、Cenju-4 を用いて性能評価を行った。Barnes Hut の詳細及び Cenju-4 を用いた性能評価結果については、3章において述べる。

2.2.2 動的な有効性管理

転送データ生産時に消費者側に転送できない問題においては、消費時にデータ転送を行う必要がある。消費したデータに再びアクセスする可能性が高い場合、データをキャッシュすることによって後続のメモリアクセス時のレイテンシを低減することができる。

消費データの有効性をプログラム実行の流れによって保証できない問題においては、消費データが非同期に更新される可能性がある。非同期に更新される可能性のある共有データをキャッシュしてアクセスする場合、共有データが更新された時には、当該領域のキャッシュコピーの値を更新するか、もしくはキャッシュコピーを無効化する必要がある。

データサイズが大きくない場合にはキャッシュの有効性管理にページテーブルを用いることが効率的ではないので、共有オブジェクト毎にキャッシュの有効性をソフトウェアで管理する必要がある。よって、キャッシュアクセス時には毎回有効性をソフトウェアで確かめる必要がある。そのソフトウェア・オーバヘッドは、ハードウェアDSMを用いた場合の有効性管理コストと比較して、非常に大きいと予想される。

このような性質のデータ転送を含む問題の具体例として、Barnes Hut [6] の第1フェーズが挙げられる。Barnes Hutの第1フェーズを実行する際にハードウェアDSMが実際どれほど有効であるかを検証するため、Cenju-4を用いて性能評価を行った。Barnes Hutの詳細及びCenju-4を用いた性能評価結果については、3章において述べる。

2.2.3 ハードウェア DSM の必要性に関するまとめ

本節では、ハードウェアDSMの高速性が効果的であると思われる問題として

- 消費時のデータ転送が必要な問題
- 消費データの有効性をプログラムの実行の流れで保証できない問題

があることを示した。

そのようなデータ転送を含む具体的な並列アプリケーションの例としては、Barnes Hutを示した。Barnes Hutを、ハードウェアDSMとソフトウェアからの指示によるデータ転送それぞれの機能を用いた場合に実際にどれほどの性能差が出るのか確かめるために、Cenju-4を用いて性能評価を行った。性能評価に用いた問題の詳細、性能評価の方法、性能評価結果等については3章に示す。

2.3 柔軟なプロセッサ間通信指示の必要性

前説では、ハードウェアDSMによる高速な一貫性制御が有効である問題が存在することを示した。しかし、ハードウェアDSMだけでは満足の行く性能が得られないアプリケーションが存在するもの事実である。現在のハードウェアDSMにおいては、一貫性制御処理の高速性の代償として、一貫性制御方式がほぼ固定されてしまっている。このため、各共有データの性質を考慮した処理を行うことが困難になっている。

本節では、ハードウェアDSMの一貫性制御方式が固定的であるために共有データの性質に合わない処理が行われてしまうデータ転送の例として

- 生産者が必要としないデータに対する一貫性制御
- 共有データの消費者へのデータ転送

を示す。以下、2.3.1節において、生産者が必要としないデータの一貫性制御におけるハードウェアDSMの問題点について述べる。続いて2.3.2節において、消費者へのデータ転送におけるハードウェアDSMの問題点について述べる。また、このような共有データの転送においても、アプリケーションからの指示を用いることによって適正な処理が行える場合があることをそれぞれの節において述べる。

2.3.1 生産者が必要としないデータに対する一貫性制御

通常のハードウェアDSMにおいては、共有データはその生産者のキャッシュ上に残る。しかし、共有データの性質上、生産者がそのデータに再びアクセスしないとアプリケーションから判断できる場合もある。このような共有データを生産した際には、アプリケーションからの指示によって、キャッシュ上に共有データを残さずに主記憶に書き戻した方が、一貫性制御の処理が削減され、さらに消費時のレイテンシが小さくなる場合があると思われる。

このような性質のデータ転送を含む具体的なアプリケーションの例としては、Radix Sort [7] における要素交換が挙げられる。Radix Sortを通常のハードウェアDSMを用いて実行した場合にどれほど無駄な通信が頻発するのか、アプリケーションからの指示がどれほど有効であるのか、等の検証を行うために、Cenju-4を用いて性能評価を行った。Radix Sortの詳細及びCenju-4を用いた性能評価結果については、3章において述べる。

2.3.2 消費者へのデータ転送

明らかに、共有データはその消費者の近傍にあった方がよい。しかし、現在のハードウェアDSMでは、生産したデータは消費者より生産者のキャッシュ上の残るものが多い。アプリケーションによっては、消費者が生産者の所までデータを取りに行くレイテンシが支配的になるものもある。このようなアプリケーションにおいては、生産した共有データを生産時に消費者側にあらかじめ転送することによって性能向上が期待できる。

ハードウェアDSMにおける生産時データ転送機能として、write-updateプロトコルによるデータ転送が挙げられる。このプロトコルでは、共有データを消費したプロセッサは再び同一領域の共有データを消費する可能性が高い、と考えて、共有データが生産された場合にはその領域のデータを以前に消費したプロセッサに対してデータを転送する。

共有データの生産者又は消費者が変化するアプリケーションにwrite-updateプロトコルを適用した場合、無駄なデータ転送が増大してしまうという問題がある。この問題を解決する方法として、以下のような改良型write-updateプロトコルを装備したハードウェアDSMも開発された。

Competitive Cache 共有データのキャッシュ領域に対するwrite-updateの回数をカウントし、あらかじめ設定された上限値の越えるまでにキャッシュに対するリードアクセスがなかった場合には当該キャッシュを無効化する。以降、このプロセッサに対してはwrite-updateを送信する必要がなくなるために、無駄なパケット送信の上限値を抑えることができる。[8]

特定の問題においては効果的に作用するが、write-invalidate方式と同様に消費者側には全くデータ転送されないような問題も存在する。

Adaptive Cache Coherence 共有データの生産者が固定であるか変化するかによって、共有データを定住性のものと移住性のものに分類する。定住性の共有データに対してはwrite-updateプロトコルを適用し、移住性のデータにはwrite-invalidateプロトコルを適用する。[10]

上述のCompetitive Cacheと同様に、特定のデータ転送パターンを持つ問題においては有効であるが、消費者が多数である定住性の共有データや、消費者が単数である移住性の共有データに対する制御方式が、共有データの性質とは掛け離れたものになってしまう。

上述のように、現在のハードウェアDSMにおいても共有データを消費者の近傍に置こうとする試みはなされている。しかし、共有データに対するメモリアクセスの履歴から共有データの性質を予想して一貫性制御方式を決定しているため、共有データの性質に一致しない方式を選択してしまう問題も多く存在する。

生産した共有データをどこに保存すべきかを最も正確に判断できるのはアプリケーションであると思われる。よって、生産した共有データの保存方法をアプリケーションから指定する方式が有効であると思われる。

ソフトウェアDSMの中には、アプリケーションからの指示に基づいた一貫性制御を実現しているものもある。例として、AURC [11]を挙げる。

AURC ページテーブルを利用したソフトウェアDSMである。ページテーブル内に各ページ内の共有データの一貫性制御に必要な情報を格納する。アプリケーションから各ページ毎にデータの送信先プロセッサを指定することによって、消費者に的確に転送することが可能になる。アプリケーションからの指定と並行して、ページ内のデータが更新された時には一貫性制御に必要な処理を実行する。

2.3.3 ソフトウェアによる指示の必要性のまとめ

本節ではソフトウェアからの指示に基づいてデータ転送を行った方がデータの性質に合致した転送が可能である可能性のある問題として

- 生産者が必要としない共有データの転送を含む問題

- 共有データ生産時に消費者への転送が可能な問題

を示した。

ハードウェア DSM において共有データに対する一貫性制御方式を決定するために用いている情報は、共有データに対するアクセスの履歴のみであるものが多い。そのために、上記のような問題においては、共有データの性質に合わない方式を選択してしまう場合も多いと思われる。このような問題においては、アプリケーションから共有データの性質に関する指示を与えてやり、その情報を元にハードウェア DSM における一貫性制御方式を決定することが有効である場合があると思われる。

そのような問題の例として、Radix Sort を挙げた。Radix Sort 内の要素交換時のデータ転送においては、ソフトウェアからの指示によって生産者側にデータを残さずに主記憶に書き戻すことの効果が大きいと思われる。Cenju-4 を用いた性能評価結果は 3に示す。

2.4 プロセッサ間通信方式のまとめ

本章では、プロセッサ間通信の管理者がハードウェアであるかソフトウェアであるかという観点から並列計算機を分類し、ハードウェア DSM の持つ高速な一貫性制御機能、及びソフトウェアからの指示による、転送データの性質を考慮した転送管理機能の双方とも有効である場合があると思われることを示した。

次章において、ハードウェア DSM とソフトウェアからの指示によるデータ転送機能を両方持つ並列計算機 Cenju-4 を用いて、上記の 2 種類の性質を持つデータ転送をハードウェア DSM とソフトウェアからの指示の双方を用いて実現した場合の性能を実測する。その実測結果より、ハードウェア DSM の持つ高速性と、ソフトウェアからの指示による柔軟性の両方とも有効であることを示す。

3 Cenju-4 による実測

前章で述べたように、ハードウェア DSM による高速な一貫性制御が有効である問題が存在し、また一方ではソフトウェアからの指示による柔軟なプロセッサ間通信が有効である問題も存在すると思われる。

上記の 2 種類の性質を持つ問題をそれぞれハードウェア DSM とソフトウェアからの指示によるプロセッサ間通信の両方の機能を用いて実現した場合の性能差を定量的に評価するために、ハードウェア DSM とソフトウェアからの指示によるプロセッサ間通信の両方の機能を持つ並列計算機 Cenju-4 を用いた。

本章では、始めに並列計算機 Cenju-4 の概要について述べる。続いて、前述の Barnes Hut を Cenju-4 のハードウェア DSM とソフトウェアからの指示による通信のそれぞれの機能を用いて実行した際の性能評価結果を示し、ハードウェア DSM の有効性を立証する。一方で、前述の Radix Sort の実行においては、共有データの性質に関してのソフトウェアからの指示が有効であることを示す。

以下、3.1節において、並列計算機 Cenju-4 の概要について述べる。続いて 3.2節において、ハードウェア DSM の有効性を示すために性能評価に用いた Barnes Hut の概要と、Cenju-4 における性能評価結果について述べ、3.3節において、ソフトウェアからの指示の有効性を示すために性能評価に用いた Radix Sort の概要と、Cenju-4 における性能評価結果について述べる。

3.1 Cenju-4 の概要

本節では、並列計算機 Cenju-4 の概要について示す。

3.1.1 Cenju-4 の構成

並列計算機 Cenju-4 は、プロセッサ間通信機構としてハードウェア DSM とメッセージ通信方式の両方の機能を装備している。図 1に Cenju-4 の構成を示す。

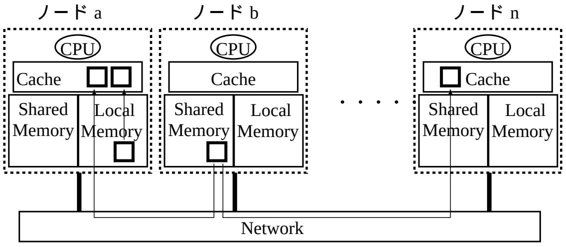


図 1: Cenju-4 の構成

Cenju-4は各要素プロセッサ毎に主記憶を分散配置している。主記憶は、全プロセッサで共有している共有アドレス空間と、各要素プロセッサ毎に独立したローカルメモリとして使用する空間にあらかじめ分割されている。共有空間はハードウェアDSMによって管理されている。また、要素プロセッサ間でローカルメモリを用いた、ソフトウェアからの指示による通信も可能である。

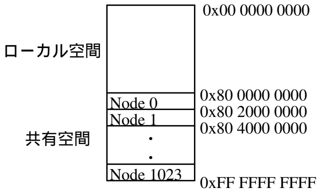


図 2: Cenju-4のアドレス空間

各ノードにおける物理アドレス空間は、図2に示すように自ノードのローカル空間と共有空間に分離されている。各要素プロセッサは内蔵の1次キャッシュと外部2次キャッシュを装備しており、ローカル空間及び共有空間どちらもキャッシュ可能である。Cenju-4における共有メモリは物理アドレス共有によって実現されているために、他プロセッサ上の共有メモリの内容を、自プロセッサ上の共有メモリ又はローカルメモリ上にキャッシュすることはできず、直接1次及び2次キャッシュにキャッシュされる(図 1中矢印2, 3)。1次及び2次キャッシュ上では、共有空間とローカル空間の区別なくキャッシュされる。

3.1.2 Cenju-4 ハードウェアDSM

Cenju-4のハードウェアDSMにおいては、40bitの物理アドレスをグローバルアドレスとしている。共有メモリ空間は1次、2次キャッシュにキャッシュ可能である。物理アドレスをグローバルアドレスとしているために、リモートノード上の共有メモリ空間を自ノードの主記憶上にキャッシュすることはできない。

一貫性制御方式はwrite-invalidate型であり、主記憶領域32バイト毎に、ノード間の共有状態を示すディレクトリを管理している。図3にディレクトリの形式を示す。Cenju-4のハードウェアDSMのディレクトリの形式は、ブロックを共有しているノード数によって異なる。共有ノード数が4ノード以下の時はノード番号をディレクトリに保管し(図3中)、5ノード以上の時はネットワークのマルチキャスト機能を利用するためのビットパターンを保管する(図3下)。

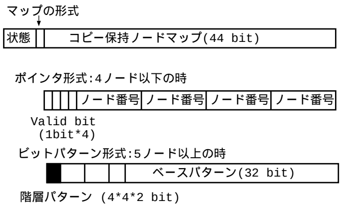


図 3: ディレクトリの形式

3.1.3 Cenju-4 ソフトウェアによるプロセッサ間通信

Cenju-4では、ソフトウェアからプロセッサ間通信を指示可能である。Cenju-4においては他ノード上で実行中のプロセスのアドレス空間が可視であり、各ノードのローカル空間を用いたメモリベース通信が可能である。今回の性能評価においては、メモリベース通信であるリモートメモリ書き込み及びリモートメモリ読み出しと、リモート関数呼び出しを使用している。

メモリベース通信においては、自ノード上のアドレス、相手ノード上のアドレス、転送データサイズを指定する。仮想アドレスを用いて転送を指示するので、相手ノード上のメモリでの保護もユーザレベルで保証できる。

リモートメモリ書き込み、及びリモートメモリ読み出しの動作を、図 4に示す。左図においてはNode 1が、Node 1のアドレスAからLバイトのデータを、Node PのアドレスBからLバイトにコピーすることを指示している。右図においてはNode 1が、Node PのアドレスBからLバイトのデータを、Node 1のアドレスAからLバイトにコピーする事を指定している。

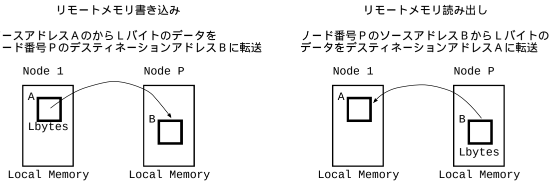


図 4: リモートメモリアクセス

3.2 ハードウェア DSM の有効性の評価

2章において、ハードウェア DSM の高速な一貫性制御機能が有効であると思われるデータ転送の一般的な性質として、消費時のデータ転送と有効性の動的管理が挙げられることを示した。また、消費時のデータ転送のコストが重要である問題の具体例として Barnes Hut のフェーズ 2 があることを示し、動的な有効性管理のコストが重要である問題の具体例として Barnes Hut のフェーズ 1 があることを示した。

本節では、Barnes Hut のフェーズ 1 及びフェーズ 2 を Cenju-4 のハードウェア DSM 及びソフトウェアからの指示による通信のそれぞれの機能を用いて実行した場合の性能評価結果を示し、上記のような性質を持つ問題において実際にどれほどハードウェア DSM が効果的なのか定量的に評価する。

まず始めに、性能評価に用いた Barnes Hut の詳細について述べる。

Barnes Hut 空間内に点在する多数の粒子間の相互動作を計算する問題である。銀河系の星衝突シミュレーション等に利用される。現実では各粒子は他の全粒子の影響を受けるが、この問題においては、距離の遠い空間に存在する粒子の影響は、遠い空間に存在する複数の粒子を 1 つにまとめて、そのまとめた物が及ぼす影響に近似している。図 5 に近似の様子を示す。粒子 b,c,d が粒子 a に及ぼす影響は、b,c,d を 1 つの粒子にまとめたものが粒子 a に及ぼす影響に近似する。

Barnes Hut の実行内容は、始めに問題空間内の全ノードデータを用いて、各空間の重心を示すデータからなるオクトリーを生成する。これをフェーズ 1 と呼ぶ。続いて、フェーズ 1 で作成したツリーを用いて、各ノードに対する作用及び移動を計算する。これをフェーズ 2 と呼ぶ。以下フェーズ 1 とフェーズ 2 の繰り返しを行う。

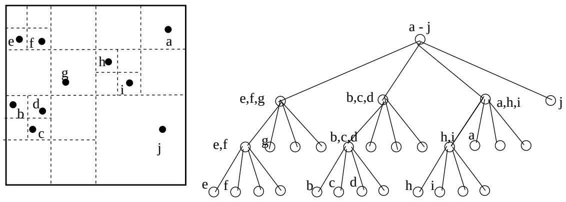


図 5: Barnes Hut

Barnes Hut を複数プロセッサで並列実行する場合、各プロセッサは近隣領域内のノードを担当し、フェーズ 1 で担当ノードをオクトリーにいれる。続いてフェーズ 2 で担当ノードに対する他のノードからの作用を計算する。この計算には担当ノードのデータ及びオクトリーのデータが使用される。各プロセッサが担当するノードは、実行時に各ノードが移動するにつれて変化していく。

ノード数が非常に多いとき、オクトリーのデータ総量は非常に大きいものである。そのため、オクトリー生成時に全プロセッサがオクトリーのコピーを持つことは不可能である。フェーズ 2 においては各プロセッサは共有データであるオクトリーから、必要となるデータを自プロセッサに転送して、計算を行う。つまり、消費時のデータ転送が必要になる。オクトリー中の各重心データの容量は巨大ではないので、細粒度データ転送のコストが重要となる。よって、ハードウェア DSM の持つ、高速なデータ検索、転送指示、転送データ管理の機能が効果的である。

3.2.1 消費時データ転送の評価

消費時のコストが重要である問題の具体例として、Barnes Hut のフェーズ 2 を挙げた。Barnes Hut のフェーズ 2 においては、フェーズ 1 で作成したオクトリーデータの一部を再度使用する。オクトリーデータの総量は非常に大きいものであるため、オクトリー生産時にデータを消費者側に転送することができず、消費者は随時必要なデータを取り寄せる必要がある。一度転送したデータは後に再び参照する可能性が高いために、自プロセッ

サの近傍にキャッシュしておくことによって、後続のメモリアクセスのレイテンシを削減できる。

以下、ハードウェア DSM とソフトウェアからの指示による通信機構を用いた場合の詳細について述べる。

ハードウェア DSM フェーズ1においてオクトリデータを共有メモリ上に作成する。フェーズ2において共有メモリからロードしたデータは、1次及び2次キャッシュにキャッシュされる。オクトリー中のどのデータをキャッシュしているかの管理はハードウェアが行い、ソフトウェアで意識する必要がない。必要なデータがキャッシュになかった時には、ハードウェアがリモートノード上の共有メモリからデータを転送する。

ソフトウェアからの通信指示 オクトリーは各プロセッサのローカルメモリ上に分散して作成される。フェーズ2において必要となったデータが他のプロセッサのローカルメモリ上にある場合には、ソフトウェアから転送を指示する。転送してきたデータを自プロセッサのローカルメモリ上にキャッシュしておくことによって、後続のメモリアクセス時のプロセッサ間データ転送を削減できる。ローカルメモリを用いたキャッシュの管理はソフトウェアで行う必要がある。

ソフトウェアキャッシュの概要を、図 6に示す。他のプロセッサのローカルメモリ上にあるデータにアクセスする際には、そのプロセッサ番号とプロセッサ内ローカルアドレスをキーにして、ソフトウェアキャッシュ内のハッシュ値を求める (図中矢印 1)。ソフトウェアキャッシュのタグを調べて (図中矢印 2) データがキャッシュされていれば、キャッシュへのポインタを返す (図中矢印 4)。データがキャッシュされていなければ、リモートメモリ読み出しを行って (図中矢印 3)、完了後キャッシュへのポインタを返す (図中矢印 4)。

同一のデータに複数回まとめてアクセスする際のコストを低減するために、2度目以降のアクセス時にはグローバルアドレスではなく、ローカルメモリ上のポインタでアクセスする。

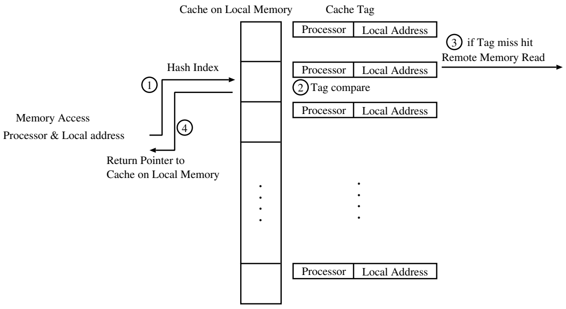


図 6: ソフトウェアキャッシュ

以下、Cenju-4 を用いた性能評価結果を示す。オクトリーの全体量を変化させてキャッシュのヒット率を変化させた時の性能評価も行った。

粒子数を 50000 とした時の性能評価結果を図 7に示し、この時のソフトウェアキャッシュのヒット率を表 1に示す。ソフトウェアキャッシュのヒット率を上げるために粒子数を 10000 とした時の性能評価結果を図 8に示す。この時のキャッシュヒット率を表 2に示す。

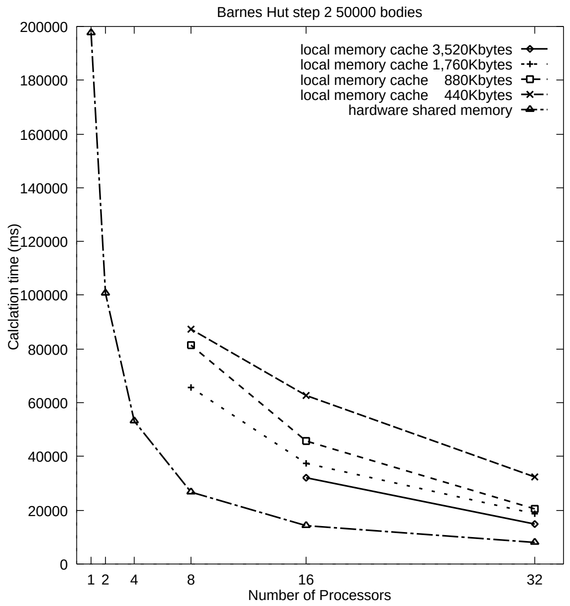


図 7: 消費時データ転送性能評価結果 50000 ノード

表 1: キャッシュヒット率 50000 ノード

	8	16	32
440K	0.797	0.793	0.783
880K	0.881	0.885	0.915
1760K	0.895	0.945	0.938
3520K		0.960	0.969

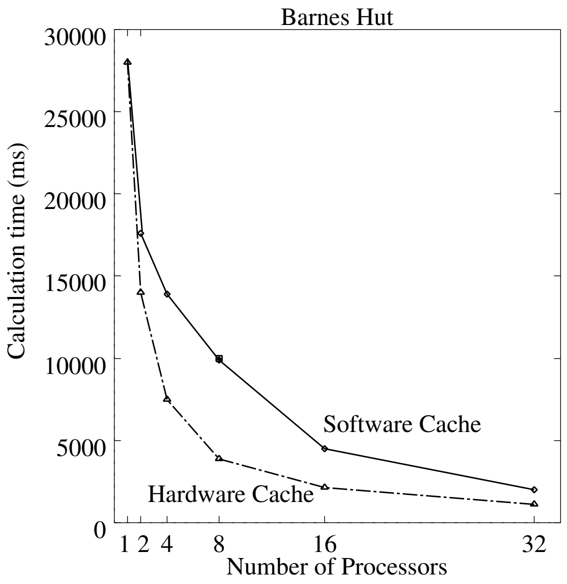


図 8: 消費時データ転送性能評価結果 10000 ノード

表 2: キャッシュヒット率 10000 ノード

	2	4	8	16	32
440K	0.921	0.838	0.831	0.803	0.880
880K	0.978	0.961	0.911	0.888	0.966
1760K	0.992	0.975	0.903	0.955	0.980
3520K	0.996	0.985	0.913	0.982	0.980

表 1及び表 2におけるキャッシュヒット率とは、リモートメモリ上のオクトリーデータを参照する際に、自プロセッサのローカルメモリ上のキャッシュにヒットする確率である。ハードウェア DSM における 2 次キャッシュヒット率を正確に求めることはできないが、そのサイズやコンフリクトが起こる可能性から、ソフトウェアキャッシュより著しく悪いと考えられる。

評価結果において注目すべき点を以下に示す。

プロセッサ間データ転送コスト ソフトウェアで管理するキャッシュサイズが小さくキャッシュヒット率が低い時に、ソフトウェアキャッシュの性能が著しく低下する。これは、ハードウェア DSM におけるリモートメモリアクセスの

レイテンシが約 $2000ns$ であるのに対し、ソフトウェアからの指示によるリモートメモリリードのレイテンシが約 $11\mu s$ であるためである。

キャッシュヒット判定コスト 問題サイズに対してローカルメモリ上のキャッシュサイズを大きく、ソフトウェアキャッシュのヒット率が 100% に近い場合でも、ハードウェア DSM と大きな性能差が現れている。これは、ソフトウェアキャッシュにおいては、ローカルメモリ上のキャッシュヒット判定に約 $880ns$ もかかり、またローカルメモリからのリードのレイテンシが約 $500ns$ ほどかかるのに対し、ハードウェア DSM においては、キャッシュの平均レイテンシが約 $100ns$ 程度であり、この時間内に有効性判定及びリードのレイテンシが含まれているためと思われる。

3.2.2 動的な有効性管理の評価

前述の Barnes Hut の第 1 フェーズにおいては、消費データの有効性をプログラムの流れで保証することができず、動的な有効性管理が必要となる。Barnes Hut の第 1 フェーズにおいては、各プロセッサは担当するノードデータをオクトリーに追加していく。この際、自プロセッサがアクセスしたオクトリーをキャッシュしておくことによって、後続のノードデータを追加する時のメモリアクセスコストが低減される。しかし、他のプロセッサもオクトリーにデータを追加するために、キャッシュしているオクトリーの有効性を動的に管理する必要がある。

ハードウェア DSM においては、共有データは 32 バイト毎に共有状態を示すディレクトリで管理され、データ更新時には他のプロセッサのキャッシュを無効化する。ソフトウェアでデータの有効性を管理する方法としては、オクトリーのノードデータを単位として共有状態を示すディレクトリを管理する手法を採用している。図 9に動的有効性管理機能付きのソフトウェアキャッシュの概要を示す。

リモートプロセッサ上のデータに対するリード時の処理の流れを以下に示す。

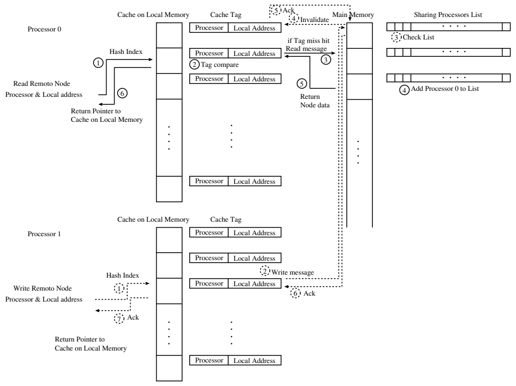


図 9: 動的に有効性を管理するソフトウェアキャッシュ

プロセッサ番号とそのプロセッサにおけるローカルアドレスをキーとしてソフトウェアキャッシュにおけるハッシュ値を求め (図中実線矢印 1)、キャッシュタグを調べて (図中実線矢印 2) キャッシュコピーが存在すれば、キャッシュへのポインタを返す (図中実線矢印 6)。キャッシュミスヒット時にはリモートプロセッサにリードメッセージを送り (図中実線矢印 3) 待機する。リモートのプロセッサはメッセージを受け取ると、ディレクトリを更新して (図中実線矢印 4) データを転送する (図中実線矢印 5)。データの転送が完了したら、キャッシュタグを更新してキャッシュへのポインタを返す (図中実線矢印 6)。

リモートプロセッサ上のデータのキャッシュコピーを更新したときの処理の流れを以下に示す。

ソフトウェアキャッシュにおけるハッシュ値を求め (図中点線矢印 1)、リモートプロセッサにライトメッセージを送る (図中点線矢印 2)。リモートプロセッサがメッセージを受け取ると、ディレクトリを検索して (図中点線矢印 3) 無効化のメッセージを送信する (図中点線矢印 4)。キャッシュコピーを共有していたプロセッサ上で無効化が完了すると ack メッセージを返す (図中点線矢印 5)。主記憶を持つプロセッサが ack を収集して書き込みを行ったプロセッサに ack メッセージを返した時点で一貫性制御の処理が終了する。

リモートプロセッサ上のデータにアクセスする際には、データの管理情報を参照及び更新する必要があるため、データを管理するプロセッサに処理を依頼する必要がある。Cenju-4 においては、リ

モート関数呼び出し機能を使用して、リモートメモリアクセス時に必要となる処理を依頼している。

ソフトウェアで動的に有効性を管理するコストが高いと予想されるので、更新される可能性のあるデータはキャッシュしないというプログラムも同時に性能評価した。更新される可能性のある、リモートプロセッサ上のデータにアクセスする際には、毎回ソフトウェアから転送指示をして、転送されたデータはキャッシュせずに破棄する。

図10に、Barnes Hut のフェーズ1をCenju-4で実行した際の実行時間を示す。ソフトウェアキャッシュのサイズは640Kバイトである。

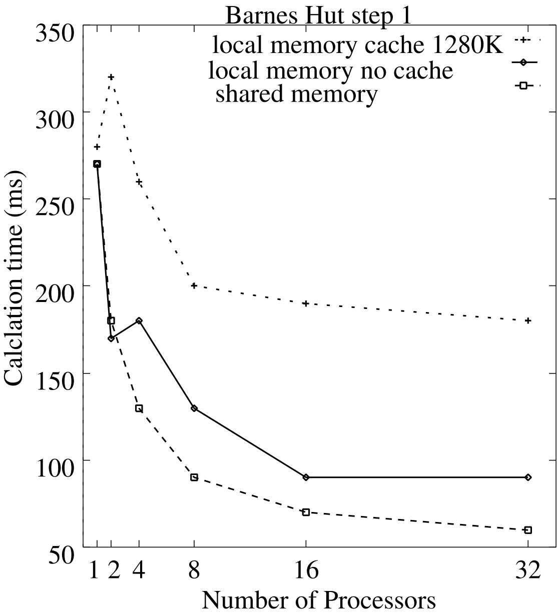


図 10: 動的有効性管理の性能評価結果

ソフトウェアでキャッシュの有効性を管理するコストが非常に高いため、リモートプロセッサのデータをキャッシュしない方が高性能が得られた。プロセッサ数を変化させた時のキャッシュヒット率を、表3に示す。キャッシュサイズを1280Kバイトに大きくした場合もキャッシュヒット率はそれほど高くならず、実行時間も大差なかった。問題の性質上、リモートプロセッサ上のデータへのアクセスの局所性が低く、キャッシュしたデータが無効化される確率が高いためと思われる。

表 3: キャッシュヒット率

	2	4	8	16	32
640K	0.931	0.888	0.913	0.884	0.876
1280K	0.933	0.911	0.911	0.888	0.867

3.2.3 ハードウェア DSM の性能評価のまとめ

2章において、

- 消費時のデータ転送コストが重要である問題
- 消費時のデータ転送を行う問題の中でも特に、消費データの動的な有効性管理が必要となる問題

等の問題においては、ハードウェアDSMが有効であると思われることを示した。

本節では、上記の様な性質を持つ問題においてハードウェアDSMが実際どれほど有効であるのかを調べるために、Cenju-4を用いた性能評価を行った。性能評価には上記のようなデータ転送を含むBarnes Hutを用いた。結果、ハードウェアDSMを用いることによって約2倍の性能をだせることがわかった。

3.3 ソフトウェアからの指示の有効性の評価

前説においてハードウェアDSMが有効である問題が存在することを示したが、2章において述べたように、ハードウェアDSMだけでは十分な性能が得られにくい問題が存在する。ハードウェアDSMがその固定的な一貫性制御方式のために十分な性能が得られていない問題としては、生産者が必要としないデータが生産者のキャッシュ上に残る問題や、生産時に消費者側にデータを転送しにくい問題等が挙げられる。

以下に、そのような問題においてソフトウェアからの、生産した共有データの性質に関する指示が有効であることを示す。

3.3.1 生産者が必要としないデータの転送

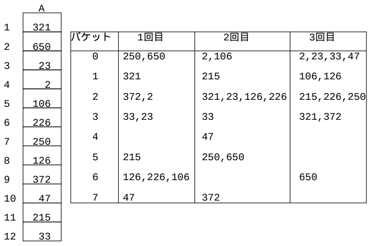
本節では、生産者がアクセスしないデータがキャッシュ上に残ってしまうために一貫性制御に必要な処理が増大し、また消費時のレイテンシが大きくなる問題の例として、Radix Sort を挙げる。

まず始めに、Radix Sort の詳細について述べる。

Radix Sort 各要素が 0 以上 $m^k - 1$ 以下の整数値、つまり k桁の m 進数で表される時、各桁毎に、合計 k回のバケットソートを繰り返して適用することによって全要素を整列できる。図 11に概要を示す。始めに、1 桁目の数値によってバケットソートを行う。その結果を元に 2 桁目の数値によってバケットソートを行う。この操作を、最大桁 (図では 3 桁目) まで繰り返すと、全要素が整列される。

Radix Sort を並列実行する際には、各プロセッサは担当する要素を Input Array から読み出し、バケットソートを行った結果に基づいて Output Array に要素を並べ替えて書き込む。図 12に概要を示す。図中に示すように、共有空間である Output Array に書き込んだ要素を参照するのは書き込んだプロセッサ自身でない可能性が高い。よって、生産者のキャッシュ上にデータを残す必要がないと思われる。

共有空間の分割を行う際に Output Array を担当プロセッサの主記憶空間に割り当てるという最適化を行うとすると、生産データを主記憶に書き戻すことによって、一貫性制御に必要な処理を低減できるだけでなく、消費時のレイテンシを小さくする効果もある。よって、アプリケーションから生産データを主記憶に書き戻すように指示することによる効果が大きいと予想される。



	A
1	321
2	650
3	23
4	2
5	106
6	226
7	250
8	126
9	372
10	47
11	215
12	33

バケット	1回目	2回目	3回目
0	250,650	2,106	2,23,33,47
1	321	215	106,126
2	372,2	321,23,126,226	215,226,250
3	33,23	33	321,372
4		47	
5	215	250,650	
6	126,226,106		650
7	47	372	

図 11: Radix ソート

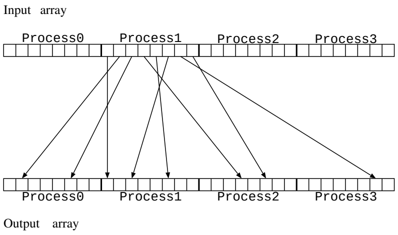


図 12: Radix Sort の並列化

Radix Sort においてプロセッサ間で転送されるデータの単位は 4 バイトであるのに対し、Cenju-4 のハードウェア DSM における一貫性制御の単位は 32 バイトである。同一ブロック内に転送データを生産するプロセッサは複数であり、そのブロック内のデータを消費するプロセッサは固定である。図 13に、転送データ生産時のデータ転送の様子を示す。

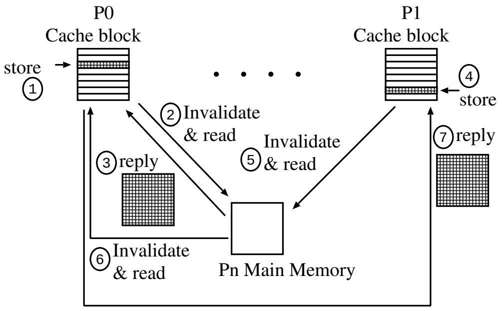


図 13: Radix Sort におけるデータ転送

図 13において P0 が 4 バイトの転送データを生産した時、そのデータを含むブロックをキャッシュ上に残すために、ブロック内の残りのデータを取り寄せる (図中 1 ~ 3)。その後、P3 が同一ブロック内の異なる 4 バイトの転送データを生産したとき、P3 にデータを残すために、ブロック全体が P3 に転送される (図中 4 ~ 8)。ブロック内のデータを消費するプロセッサは固定であり、主記憶を持っているプロセッサである。転送データの生産者がブロック内のデータにアクセスする可能性は

低く、生産者側にデータを残す意味は全くない。

160000 要素の整数データを Cenju-4 のハードウェア DSM を用いて整列した場合の転送時間を、表 4に示す。プロセッサ数を増やしても、一貫性制御のコストが増し、また消費時にデータが主記憶上にないために必要な転送のレイテンシも大きいために、それほど性能が向上しないことがわかる。プロセッサ数が 8 を越してからは、ブロック内の要素数が 8 であるために一貫性制御に必要なコストがこれ以上増えないために、性能向上率は安定する。

表 4: Radix Sort における要素転送時間

Processors	1	2	4	8	16	32
Time (ms)	4320	2960	2700	3340	2010	1040

このように、アプリケーションによっては生産したデータの性質を把握できるにもかかわらず、ハードウェア DSM においては全データに対して固定的な一貫性制御処理を施してしまう場合がある。

Radix Sort において、転送データ生産時に当該データを含むブロックを生産者側にアロケートせず、主記憶に生産データを書き戻すように指示可能であれば、図 13におけるデータ転送は図 14 のように簡素化される。図 14においては、アプリケーションからの指示によって生産したデータを含むブロックをキャッシュにアロケートしていない。プロセッサからの書き込みがあった時には、ストア対象データが含まれるブロック中の残りのデータを取り寄せずに、ストアしたデータのみを主記憶に転送する (図 14中 2)。この指示によって、ブロック内のデータの一貫性制御に必要な処理が軽減されるばかりでなく、転送データが消費者の主記憶上に残るので消費時のレイテンシも軽減される。

Cenju-4 のソフトウェアからの通信指示機能であるリモートメモリライトを使用して Radix Sort における要素転送を実現した。ソフトウェアでデータの転送を管理しているためにハードウェア DSM を用いた時のような無駄な転送は発生せず、生産したデータは消費者に対して必要なデータ量だけ転送される。プロセッサ間のデータ転送の様子は図 14と同様になる。ソフトウェアからの指示を用いた場合の転送時間を表 5に示す。

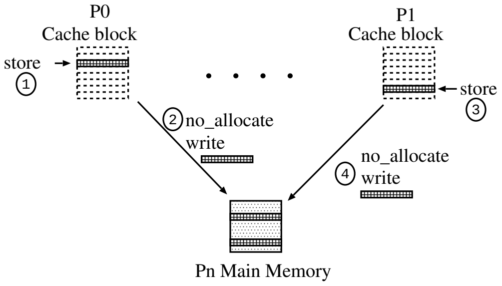


図 14: 生産者がブロックをアロケートしない場合の Radix Sort

表 5: ソフトウェアからの転送指示を用いた Radix Sort

Processors	1	2	4	8	16	32
Time (ms)	2950	12530	9070	5210	2830	1480

ハードウェア DSM を用いた場合と比較して、無駄なデータ転送が行われずプロセッサ間のデータ転送量が大幅に削減され、また消費者の主記憶上に生産データが転送されて消費時のレイテンシが削減されるために、転送時間が短くなるはずではあるが、Cenju-4 のリモートメモリライト指示のソフトウェア・オーバヘッドが高いために、ハードウェア DSM 程の性能も出せていない。

現在の並列計算機におけるソフトウェアからの転送指示機能は、ハードウエア DSM における転送起動に比べると、まだまだ低速である。ハードウェア DSM における転送起動と同等のコストで、ソフトウェアからデータの転送に関する指示を行える機構が必要であると思われる。

3.3.2 消費者へのデータ転送

プロセッサ間で転送される共有データは、明らかにそのデータの消費者の近傍に保存されるべきである。しかし、現在のハードウェア DSM ではソフトウェアからデータの転送を意識する必要が

ない、むしろ意識できないために、転送データはその生産者のキャッシュに残る場合が多く、消費時に生産者側からデータを転送するコストが支配的になる問題も存在する。

ハードウェア DSM において生産時にデータ転送を行う方法として、write-update プロトコルがある。write-update プロトコルにおいては、共有データが繰り返しアクセスされる可能性が高い事を利用して、あるデータが更新された時には、同じ領域の前のデータを消費したプロセッサに対して最新のデータを転送する。プロセッサ間のデータ転送パターンが固定的な問題においては生産時のデータ転送が実現できる。

これに対し、write-update によるデータ転送には、

- 計算途中のデータを共有空間にストアした場合、意味のないデータの転送が実行される。
- 共有データの消費者が変化する問題においては、データの転送量が爆発してしまう。

という問題がある。このような問題の多くが、共有データの更新時にソフトウェアから一貫性制御に関する指示を与えることによって防げると思われる。

write-update プロトコルを用いた場合に上記のような理由によって膨大なデータ転送が起こる問題の具体例としては、MP3D、Water [12] 等が挙げられる。そのような問題を write-update プロトコルで実行した場合に起こる無駄なデータ転送を、ソフトウェアからの指示によってどれほど削減できるのか評価する必要があるが、本稿では今後の課題とする。

3.3.3 ソフトウェアからの指示の有効性のまとめ

上述したように、ハードウェア DSM による固定的な一貫性制御では無駄なデータ転送が乱発したり、生産時にデータが消費者側に転送されない等の問題が起こり得る。そのような問題においても、ソフトウェアから共有データの性質に関する指示を与えることによって、より的確な一貫性制御を実現できるものが多い。しかし、現在の並列計算機におけるソフトウェアからの通信指示コストは、ハードウェア DSM における通信起動コストに比べてまだまだ大きいものである。ハードウェア DSM における通信起動のコストと同等に低いコストで、ソフトウェアからデータの転送に関する指示を出せる機能が望まれる。

3.4 Cenju-4 による実測のまとめ

本章では、Cenju-4 を用いた性能評価結果より、ハードウェア DSM の持つ高速なデータ転送機能及び動的な有効性管理機構の有効性を立証した。一方で、通常のハードウェア DSM では十分な性能が得られない問題も存在することを示した。そのような問題を解くためにはソフトウェアからの指示が有効であるが、Cenju-4 におけるソフトウェアからの通信指示機能は、ハードウェア DSM における転送起動に比べてまだまだ低速であるために、ソフトウェアからの指示を用いてもそれほど性能が向上しない。転送データ生産時に、メモリに対するストアと同等のコストで転送に関する指示をソフトウェアから出せる機能が必要であると思われる。

4 ソフトウェアからのヒントを利用するキャッシュ一貫性制御方式

これまでに述べたように、ハードウェア DSM の高速なデータ転送及び動的な有効性管理機能が有効である問題が存在する。また一方では、ハードウェア DSM における固定的な一貫性制御方式では十分な性能を得られない問題も存在し、そのような問題においてはソフトウェアからデータ転送に関するヒントを与えることが有効であることを示した。つまり、様々な問題に出現するデータ転送に対して十分な性能を得るためには、ハードウェア DSM によるデータ転送機能と、ソフトウェアからの通信指示機能の両方が必要であることがわかる。

ハードウェア DSM によるデータ転送機能とソフトウェアからの通信指示機能の両方の機能を持つ並列計算機として、Cenju-4 が実在する。しかし、Cenju-4 においては、3章で示したように、ハードウェア DSM が管理するメモリ空間と、ソフト

ウェアからの通信指示が可能なメモリ空間が静的に分割されているため、ハードウェアDSMで管理しているデータオブジェクトに対してソフトウェアから通信指示を出すためには、自プロセッサのメモリ上で共有メモリ空間からローカルメモリ空間へのコピーが必要である。逆に、ソフトウェアで転送を管理しているデータに対してハードウェアDSMが制御するためには、ローカルメモリ領域から共有メモリ領域へのコピーが必要である。

このような問題を解決するために本稿では、一貫性制御に関するソフトウェアからの指示を受け付けるハードウェアDSMを提案する。基本的にはディレクトリベースで一貫性制御を行うハードウエアDSMであり、ソフトウェアから通信に関する指示があった時には、指示内容とディレクトリの内容の両方を考慮して必要な処理を行う。システム内の制約によってソフトウェアからの指示に従うのが困難である場合は、共有データの一貫性制御が保たれている範囲でソフトウェアからの指示を無視することができる。そう言う意味で、我々はこの指示を”ヒント”として位置づけている。

以下、ハードウェアDSMに対するソフトウェアからの指示の概要、汎用プロセッサを用いた並列計算機における実装方式の案、現実装方式における問題点等に関して述べ、最後に並列計算機JUMP-1への実装に関して述べる。

4.1 概要

本稿では、固定長のブロック毎にプロセッサ間の共有状態を示すディレクトリを保持して、データ更新時には他のプロセッサのキャッシュを無効化する、write-invalidate型のハードウェアDSMを基本アーキテクチャとする。このハードウェアDSMに対して、以下に示すようなプロセッサ間のデータ転送に関する指示をソフトウェアから行う。プロセッサからの指示はハードウェアDSMにとって一貫性制御を行う上でのヒントとして扱われ、メモリモデル等の制約を破らない範囲でソフトウェアからの指示を実行する。

以下に、ハードウェアDSMに対するソフトウェアからの指示内容と、それらの指示を受けた時の処理を説明する。

データ転送対象スレッド指定 ソフトウェアから、生産した共有データの転送先のスレッドIDを指定する。共有データの一貫性を保つため、データの転送は一貫性制御対象のブロックを単位とする。ソフトウェアからの転送指示のあった対象スレッド以外に当該ブロックを共有するプロセッサが存在する場合には、それらのプロセッサのキャッシュをInvalidateする。

ソフトウェアで転送データの消費者を求められる問題においては、消費時のレイテンシを大幅に削減できる。また、write-updateと違って、データ転送量が増大することもない。

主記憶書き戻し 自プロセッサのキャッシュの内容を主記憶に書き戻す。生産したデータの消費スレッドを指定できない時などに使用することによって、消費時のレイテンシを多少削減できる。

no-allocate 指示 生産したデータに再びアクセスする可能性が非常に少ない時には、自プロセッサのキャッシュに残さないことを指示する。上記の2種類の指示と合わせて用いることによって、無駄な一貫性制御処理を低減するとともに、消費時のレイテンシを下げる効果もある。

コヒーレンスプロトコル指示 生産した共有データの転送が単調である場合等には、従来のwrite-updateプロトコルや、write-invalidateプロトコルによって一貫性制御を行うことをソフトウェアから指示する。

上記の指示は全て、共有データを生産するときに行うものであり、消費時にはソフトウェアからは対象アドレス以外の指示は行わないものとする。これは、ハードウェアDSMにおけるデータ検索及び転送の機能は非常に優れており、ソフトウェアからの指示が必要ないと思われるためである。

前述のRadix Sortにおいては、転送データ生産時に主記憶書き戻し及びno-allocateを指示することによって、無駄な一貫性制御処理を軽減できると共に消費時のレイテンシを削減できる。Radix Sortにおいては転送データの主記憶を持つプロセッサが転送データの消費者であるために、ソフトウェアから転送対象スレッドを指定する必要がない。

4.2 ソフトウェアからの指示のタイミング

前節で述べたように、共有空間にデータを生産した時にそのデータの性質に関する指示をソフトウェアから出すことができる。ソフトウェアからの指示を行うタイミングとしては

- 共有メモリ空間に対するストアと分離して、ソフトウェアからのデータ転送に関する指示を行う。
- 共有メモリに対する書き込み時に同時にソフトウェアからの指示を行う。

の 2 通りの方法が考えられる。
それぞれ以下に示すような特徴がある。

- 共有メモリ書き込みとソフトウェアからの指示を分離する

  共有データの生産と転送に関する指示を分離した場合、前述したように一貫性制御処理の開始時期はソフトウェアから指示できないために、 共有空間にデータを生産した時点でハードウェア DSM は必要な一貫性制御処理を行う必要がある。続いてソフトウェアから指示があった場合には、指示に基づく処理を行う。よって、指示の内容によっては始めのハードウェア DSM による一貫性制御処理が無駄になってしまう。

- 共有メモリ書き込み時にソフトウェアから転送に関する指示を行う

  データの生産と一貫性制御に関する指示を同時に行った場合、ハードウェア DSM は指示内容と一貫性制御のためのディレクトリの内容の両方を考慮した処理を実行できる。反面、ソフトウェアからの指示に基づくデータ転送の単位がブロック単位になってしまうという欠点がある。これにより、ブロックよりはるかに大きいデータ構造に対してソフトウェアから転送を指示する場合等には、転送指示に必要なソフトウェア・オーバヘッドが大きくなってしまう。

本稿では、ソフトウェアからハードウェア DSM に対する指示のコストを通常のストアと同程度まで小さくすることを念頭においているために、共有メモリに対するストアと同時に、ストアしたデータの一貫性制御に関する指示をソフトウェアから行う方式を採用する。

4.3 汎用プロセッサを用いた並列計算機への実装法

前節で述べたように、共有メモリ上にデータを生産する際には、ソフトウェアから生産したデータの一貫性制御方法等を指定する。理想的には、store 命令時にその対象アドレスとデータ以外に、データの制御方法に関する指定ができる並列計算機専用プロセッサを開発することが望まれる。しかし、専用プロセッサ開発のコストは非常に高いため、汎用プロセッサを用いて実装することが要求される。

本稿では、汎用プロセッサを用いた実装法として、アドレス空間を用いてソフトウェアからの指示を行う方式を提案する。プロセッサからの store 時には、store 対象アドレスをソフトウェアからの指示で拡張したものをアドレスとして利用する。例えば、図 15において write-update と示された領域に対して srote を行った時には、図右のプロセス内仮想アドレス空間が指すメモリオブジェクトに対する store において、write-update プロトコルで一貫性制御を行うことをソフトウェアから指定している。

アドレス空間のモデルとしては、タスク・スレッドモデルを仮定している。図 15中で示すように、同一のオブジェクトに対して複数の仮想アドレスが割り付けられる。

前述したように、ソフトウェアからの指定内容としては、生産データの転送先スレッド ID、主記憶書き戻し指定、コヒーレンスプロトコル指定、及び生産データを自プロセッサのキャッシュにアロケートするかどうかの指定等が有効であると思われる。

以下に、上記のような指定をソフトウェアから行った時に、TLB 及びページテーブルによってどのようにアドレス変換がなされるのかを示す。

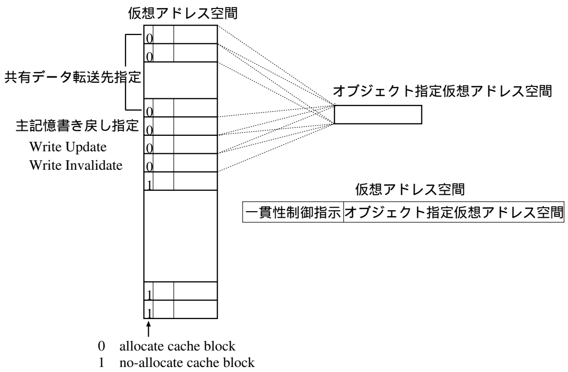


図 15: 仮想アドレス空間

4.3.1 転送先スレッド ID 指定

ソフトウェアから生産データの消費者を指定することによって、生産時に的確に消費者側にデータ転送することが可能となる。このデータ転送を本稿では INJECTION [16] と呼ぶ。

ソフトウェアから生産データの転送先を指定する方法として、対象となる仮想スレッド ID を指定する方法と、複数のスレッド ID のセットを指定する方法を用意する。

以下に、単独スレッド指定及びスレッドセット指定時のアドレス変換について述べる。

単独スレッド指定 仮想アドレスの上位で、消費者の仮想スレッド ID を指定する。図 16にアドレス変換の様子を示す。仮想スレッド ID は、そのスレッドが割り当てられている物理ノード ID に、仮想共有アドレスはノード内の物理アドレスに変換される。それらを結合したものが物理アドレスとして出力される。このアドレス変換は TLB 及びページテーブルでなされる。同一のオブジェクトに対して複数のアドレスを割り付けているため、対象データオブジェクトはノード内で物理ページに割り付けられているのに、対象スレッド ID が異なるためにページフォールトが起こることがある。このようなページフォールト時には、OS の管理するスレッド割り付け表を用いて仮想スレッド ID を物理ノード ID に変換し、さらに対象オブジェクトのノード内物理アドレスをページテーブルを用いて求め、それらを結合することによって、物理アドレスを得ることができる。

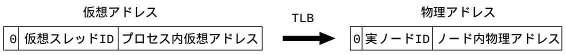


図 16: 単独スレッド指定 INJECTION

スレッドセット指定 複数のスレッドに対してデータを転送したい場合には、あらかじめ複数のスレッド ID からなるセットを作成しておき、そのセットに対してデータ転送を指定する。複数のスレッドからなるセットの構成が固定的な問題においては、このスレッドセットの内容を、OS がスレッドを物理ノードに割り付けるときに有効な情報として利用できる。

システムがハードウェアで提供するマルチキャスト (ブロードキャスト) 機能を持つ場合、スレッドセットの内容によっては、スレッド ID のリストに対してデータ転送するよりもマルチキャスト用のデータを使用する方が効率的である。指定されたスレッドセット No から対応するマルチキャストデータを高速に求めるために、プロセッサの外部にスレッドセット No と実行中のスレッド ID からマルチキャストデータを求める TLB を構築する。この TLB がミスした時には、スレッドセットの表を検索して、マルチキャストデータを作成する必要がある。

実行中にスレッドセットの登録内容を変更する際には、そのセットが外部 TLB に登録されている可能性があるので、システムコールが必要となる。図 17にアドレス変換及びマルチキャストデータ作成の方法を示す。

4.3.2 主記憶書き戻し指定

プロセッサ間転送データを生産した時に消費者を指定できない問題や、消費者側のメモリ空間の容量上の制約から、生産したデータをすぐに消費者側に転送しない場合がある。そのような問題においては生産したデータを主記憶上に書き戻すことによって消費時のレイテンシを多少削減できる。

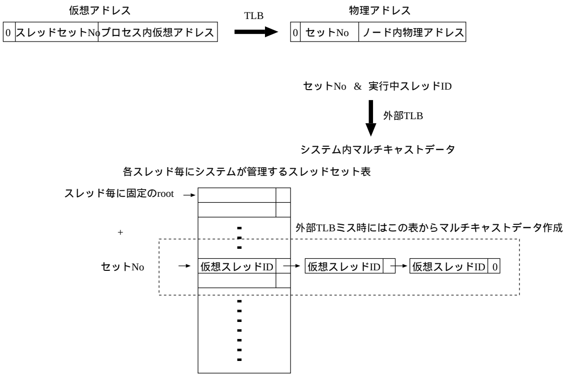


図 17: スレッドセット指定 INJECTION

物理アドレス空間においては、特定の一貫性制御方式用の領域を用意する。

4.3.3 no-allocate 指定

2章で例に挙げた Radix Sort においては、消費者側に転送するデータを生産者は参照しない。このようなデータ転送パターンにおいて生産者側にデータを残すと、コヒーレンス制御のコストが増すだけでなく、消費時のレイテンシも増加する。

前述したソフトウェアからの指定内容と組み合わせて用いるために、物理アドレス上で 1 ビットの情報を付加する。

4.3.4 write-invalidate 指定

本稿では、write-invalidate 型のハードウェア DSM を基本構成としているために、特にソフトウェアからの指示を必要としない、もしくは適切な指示をソフトウェアから与えられない書き込みは、write-invalidate 指定領域になされる。

また、本稿ではデータ消費時においてはハードウェア DSM に対するソフトウェアからの指示は必要ないと考えているため、共有メモリに対する全ての読み出しが、書き込み時の基本領域である write-invalidate 領域に対して行われるものとする。

4.4 アドレスのエイリアスに対する一貫性制御

本稿では、ソフトウェアから一貫性制御方式に関する指示を与える方法として、アドレス空間を用いた実装法を提案している。結果、4.3節において述べたように、同一メモリオブジェクトに対して複数の仮想アドレス、及び物理アドレスが割り付けられる。よって、異なる物理アドレス間での一貫性制御処理が必要となる。

プロセッサの外部においては、図 16中におけるノード内物理アドレスが同一で、ソフトウェアからの一貫性制御に関する指示が異なる場合には、同一のメモリオブジェクトを指定していると認識可能である。

しかしプロセッサ内部においては、物理アドレスが異なるものは異なる領域のデータとして扱われる。よって、プロセッサの外部から、異なる物理アドレスを持つ同一メモリオブジェクトに対する一貫性制御を行う必要がある。この一貫性制御の概要を、図 18に示す。

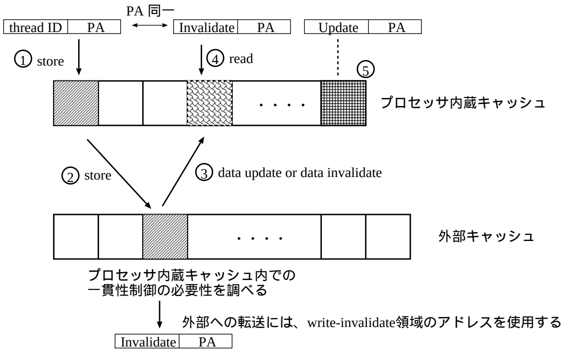


図 18: 異なる物理アドレスに対する一貫性制御

プロセッサ内蔵キャッシュ中には、同一メモリオブジェクトに対する複数のデータがキャッシュされている可能性がある。しかし、4.3節に示したように、全ての共有メモリ読み出しは write-invalidate 指定領域に対してなされるため、外部からの一貫性制御処理が必要なのは write-invalidate 指定領域に対してのみである。

write-invalidate 指定以外の領域に対する書き込み (図中 1) が発生し、その書き込みがプロセッサ外部に伝わった時 (図中 2)、write-invalidate 領域に対する読み出し (図中 4) が行われる前に、プロセッサ外部から write-invalidate 指定領域を更新又は無効化する必要がある。この選択は、共有メモリ書き込み時のソフトウェアからの指定内容に

よって決定される。例えば、ソフトウェアからの指定が no-allocate であった時には、明らかに内蔵キャッシュのデータを無効化する必要がある。同一オブジェクトに対する、write-invalidate 指定以外の領域がキャッシュされていても (図中 5)、外部から無効化する必要はない。

4.5 性能予測

本節では、アドレス空間を用いたソフトウェアからのデータ転送指示機構の予備的な性能予測を行う。

4.5.1 共有データ消費時のコスト

共有データを消費する時には、ソフトウェアからの指示を行わない。よって、ハードウェア DSM における消費時転送機能と同等の性能があると思われる。

4.5.2 ソフトウェアからの指示を用いたデータ転送コスト

生産したデータの転送先スレッド ID をソフトウェアから指定する場合には、TLB ヒット時には通常のストアと同程度のコストで、消費者への転送が実現できる。しかし、転送先のスレッド ID が動的に変化する場合等には TLB ミスヒットの確率が高く、また生産するオブジェクトがメモリ上にマッピングされているにもかかわらず、対象スレッド ID が異なるために、ページフォールトまで起こる可能性がある。このような場合にはソフトウェアからの転送指示コストは無視できるものではない。

一方、主記憶書き戻し指定や no-allocate 指定においては、一貫性制御に関するソフトウェアからの指示のパターン数が少ないために、同一アドレスを複数回利用する事が予想され、TLB のヒット率も十分に高いと思われる。よって、メモリアクセスと同等のコストでソフトウェアからの指示が可能であると思われる。

前述の Radix Sort において、転送データ生産と同等のコストで主記憶に対する書き戻し、及び no-allocate を指示できると仮定すると、転送時間は表 6程度であると予想される。この評価結果は、Cenju-4 を用いて、ローカルメモリに対するストア時にほとんど余分なコストを必要とせずに他のプロセッサのローカルメモリに生産したデータを転送できる機構があると仮定して評価している。仮想アドレスを用いた実装においても、TLB ミスヒットはソフトウェアからの指示を行わない場合と同程度しか起こらないと考えられ、表 6に示すような性能が得られると思われる。

表 6: 低コストな転送指示を用いた Radix Sort

Processors	1	2	4	8	16	32
Time (ms)	2950	1530	780	380	200	90

4.6 現在の実装の問題点

共有空間に対して書き込みを行う際に、アドレス空間を用いてソフトウェアからの指示を与えることによって、通常の書き込みと同等のコストでソフトウェアからの指示を与えられる。

しかし、同一のメモリオブジェクトに対して異なる物理アドレスでアクセスするために、前述のようにプロセッサ内蔵キャッシュの一貫性制御が必要となった。また、同一のデータに対して複数の種類の指示をソフトウェアから行う場合等には、同一オブジェクトによって複数の TLB 及び内蔵キャッシュを消費してしまい、資源の有効利用率が著しく低下してしまう。オブジェクトがノード内の物理メモリにマッピングされているにも関わらず、新しい指示をソフトウェアから出した時にはページフォールトが起こるという問題もある。

これらの問題を解決するためには、メモリアクセス時にアドレスとは独立したソフトウェアからの指示を出力する機構をプロセッサが装備することが望まれる。ソフトウェアからの指示は仮想化されているため、指示を物理空間に変換する専門の TLB を装備することも望まれる。この TLB においては、例えば仮想スレッド ID から物理ノード ID への変換等が行われ、共有メモリアクセス時にアドレスと一緒に出力される。

4.7 超並列計算機 JUMP-1 への実装例

本節では、本稿で提案するアドレス空間を用いたソフトウエアからのヒントを利用するキャッシュ一貫性制御方式の、並列計算機 JUMP-1 への実装について述べる。

4.7.1 JUMP-1 の構成

JUMP-1 は分散共有メモリ型の並列計算機であり、要素プロセッサ間の接続形態を決定するにあたって、物理的な実装条件を自然な形で論理的な接続形態に反映できるように、クラスタ構成をとっている。要素プロセッサ 4 つ毎に 1 つのクラスタを形成している。

JUMP-1 においては、最大 256 のクラスタが、RDT ネットワーク [21] によって接続される。各クラスタ内の構成を図 19に示す。

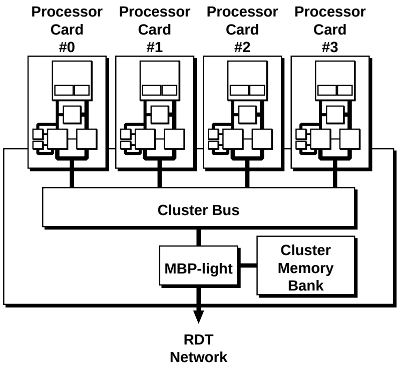


図 19: JUMP-1 のクラスタの構成

以下に、クラスタ内の各要素の詳細を述べる。

MBP-light MBP-light は RDT ネットワークを介したクラスタ間のパケット送受信を管理するプロセッサで、クラスタ間にまたがる共有メモリ管理を担当している。JUMP-1 においては、クラスタ内とクラスタ間の 2 階層に階層化された構成となっているので、共有メモリ管理もクラスタ間とクラスタ内の 2 階層に分かれて実現されている。クラスタ内の共有メモリ管理は、後述のキャッシュコントローラ (CC) によって行われる。

アドレス空間を用いたソフトウエアからのデータ転送に関する指示の中で、生産したデータの消費スレッド指定の機能は、MBP-light 上のソフトウェアによって実現される。no-allocate 指示及びコヒーレンスプロトコルの指示機能は、後述の CC によって実現される。

Cluster Bus Cluster Bus(CBus) は 4 つの要素プロセッサが共有しているメモリバスである。スプリットトランザクション方式を用いている。

Processor Card クラスタ内の 4 つの要素プロセッサは、それぞれ 1 枚のプリント基盤上に実装されている。これを Processor Card と呼ぶ。図 20に Processor Card 内の構成を示す。

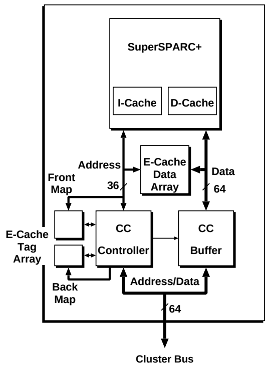


図 20: Processor Card の構成

以下に Processor Card 上の要素について詳細を述べる。

SuperSPARC+ JUMP-1 の要素プロセッサとなる汎用の RISC プロセッサであり、1 次キャッシュとして 16Kbyte のデータキャッシュ(D-cache)、20Kbyte の命令キャッシュ(I-cache) を内蔵している。

CC クラスタ内の共有メモリ管理を行う。MBP-light がプロセッサであるのに対し、クラスタ内の共有メモリ管理にかかる速度がシステムの性能を大きく左右すると思われるので、CC は完全にハードウェアで構成されている。主に 1Mbyte の 2 次キャッシュ(E-cache) の管理を行っている。アドレス空間を用いたコヒーレンスプロトコルの決定、no-allocate の

指示等の機能もCCによって実現されている。

4.7.2 JUMP-1におけるアドレス体系

各クラスタに分散配置されたクラスタメモリは、システム全体における主記憶としてだけではなく、他のクラスタメモリのコピーとしても利用される。すなわち、プロセッサから見ると3次キャッシュとして利用できる。

クラスタメモリを3次キャッシュとして利用する際のブロックサイズは、4Kである。これは、SuperSPARC+が提供する最小のページサイズである。他のクラスタメモリのコピーであるページはコピーページと呼ばれ、元々自クラスタに主記憶として割り当てられたページはオリジナルページと呼ばれる。各クラスタのクラスタメモリ内には、コピーページとオリジナルページが混在することとなる。

要素プロセッサからのコピーページに対するアクセスは、オリジナルページと同様にクラスタ内における物理アドレスによって行われる。このアドレスをクラスタアドレスと呼ぶ。ユーザプロセス上の論理アドレスとクラスタアドレス間のマッピングは、通常のMMUが使用するページテーブルによって管理される。

一方、ページがクラスタ間で共有されている(オリジナルページが他のクラスタにコピーされている)場合、各クラスタにおけるクラスタアドレスは異なるので、それらのページが同一ページであることを示すアドレスが必要となる。このアドレスをネットワークアドレスと呼び、各メモリオブジェクトを一意に決定する。簡単な方法として、ネットワークアドレスはユーザプロセス内の論理アドレスを、システム全体におけるプロセスIDで拡張したものを用いることができるが、ネットワークアドレスは完全にクラスタアドレスから仮想化されているため、MBP-lightによって自由にアドレス付けすることが可能である。

4.7.3 メモリコマンド

JUMP-1においては、アドレスの一部を用いて共有メモリに対するアクセス方法をソフトウェアから指示できる。ソフトウェアからの指示に用いられるアドレスの一部分を、メモリコマンドと呼ぶ。JUMP-1のメモリシステム、すなわちCCとMBP-lightは、アドレスのうちのメモリコマンドに応じた一貫性制御を実現する。例えは、キャッシュコヒーレンスプロトコルや、要素プロセッサ間の通信操作等を、メモリコマンドと呼ばれる、アドレスの一部によって動的に指定できる。

メモリコマンドはクラスタアドレスの一部として特定される。SuperSPARC+の物理アドレス(クラスタアドレスPA[35:0])は36ビットであり、クラスタメモリの最大容量は128Mbyteであるため、クラスタメモリ上の物理アドレスとしてはPA[27:0]しか使用されない。PA[35:32]は、プロセッサがCCに対して、PA[27:0]で指定されるデータに対するアクセス方法を指定するための付加情報として使用され、メモリコマンドと呼ばれる。この4ビットによって、16種類のアクセスパターンが提供される。PA[31:28]は各メモリオブジェクトの静的な属性を示す他、MBP-lightが提供するソフトウェアからの指示機能を指定するために用いられる。

図21にアドレス空間とメモリコマンドの関係を示す。ユーザプロセスが論理アドレスVmで指定されるメモリオブジェクトに対して、異なるメモリコマンドnによってアクセスしたい場合、OSに対して

- 論理ページVm
- 新しいメモリコマンドn (n = 0,1, ⋯,15)
- 新しい論理ページVn

を指示する。OSはVmの物理ページPmの物理アドレスPA[27:0]を、新しいメモリコマンドnで拡張した物理アドレスを持つ物理ページPnを用意し、論理ページVnを物理ページPnにマッピングする。ユーザプロセスは同一メモリオブジェクトに対して異なるメモリコマンドを用いてアクセスすることができる。

同一のメモリオブジェクトに対して異なる物理アドレスを与えているため、前述のようにプロセッサ内蔵キャッシュに対する一貫性制御が必要となる。SuperSPARC+の内蔵キャッシュは4-wayセットアソシアティブであるため、同一オブジェクトが最大4つまでキャッシングされる可能性がある

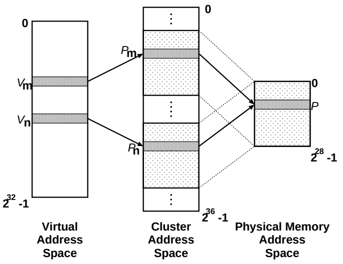


図 21: アドレス空間とメモリコマンドの関係

が、CC は前述の一貫性制御を実現すると共に、内蔵キャッシュの有効利用率を高めるために、最大 2 つまでしかキャッシングされないように管理している。

4.7.4 ソフトウェアからのコヒーレンスプロトコル指示

現在 JUMP-1 はクラスタ内の実装のみ完了しており、複数クラスタを用いたプログラム実行はできない。よって、MBP-light が提供する機能は利用できず、現在ソフトウェアからの指定で実行可能なものは、write-update と write-invalidate の指定等限られたものである。

本節では、共有メモリ書き込み時にデータの性質に合わせてソフトウェアから一貫性制御に関する指示を行う機能のうちで、write-update か write-invalidate かを、データに合わせて決定する機能の効果を評価する。

性能評価に用いた問題を以下に示す。

内積形式 LU 分解 1 ブロック単位 (以下 IP8) 内積形式 LU 分解において、要素を 8 ワード (1 ブロック size) ずつタイリングしている。問題サイズを N とした時、ロードの回数が$\frac{2}{3}N^3$、ストアの回数が N^2、フローティング演算の回数が$\frac{2}{3}N^3$となる。データの参照及び更新が同一ブロックに対して連続して行われるためキャッシュヒット率が向上する。並列化においては、列方向に 8 列毎にサイクリックに分割している。各ブロックに対する計算責任を持つプロセッサが固定されているため、コヒーレンス制御のための通信は各ブロックにつき 1 回しか行われない。プロセッサ間の同期は、8 ワード (1 ブロック) 毎にとっている。

外積形式 LU 分解 (以下 OP1) 外積形式 LU 分解において、k、i、j の順に外側からループのイタレーションに用いている。ロードの回数は$\frac{2}{3}N^3$、ストアの回数は$\frac{1}{3}N^3$、フローティング演算の回数は$\frac{2}{3}N^3$となる。IP8 に比べてストアのオーダーが増している。各ストアは行方向に連続して起こる。並列化においては、行方向 (イタレーション j) で 1 行ずつサイクリックに分割している。同一ブロックに対する計算責任を持つプロセッサは固定している。プロセッサ間の同期は k (最外側ループ) における各プロセッサの計算が終了する度にとっている。

アプリケーションの性質としては、IP8 においては転送データの再利用性が高く、write-update によるデータ転送が効率的であると思われる。一方 OP1 においては、計算途中のデータを共有メモリ上に保管するために、write-update プロトコルでは無駄なデータ転送が増大してしまうと思われる。よって、IP8 においてはソフトウェアから write-update を指定し、OP1 においてはソフトウェアから write-invalidate を指定するのが良いと予想される。

上記の問題を、write-invalidate、write-update の両プロトコルを指定して実行した場合の性能評価結果を、図 22及び図 23 に示す。IP8 においては write-update プロトコルの方が性能が良く、OP1 においては write-invalidate プロトコルの方が性能が良い。よって、ソフトウェアからデータの性質に合わせてコヒーレンスプロトコルを指示する機能が有効であることがわかる。

4.7.5 JUMP-1 への実装のまとめ

本稿では、JUMP-1 に実装されているアドレス空間を用いたソフトウエアからの指示機能の中で、write-update と write-invalidate を共有データの性質に合わせて切り替える機能の有効性を示した。

write-invalidate と write-update の切り替え指示は、本稿で主張するソフトウェアからの通信指示の中では最も簡単な指定内容の 1 つであり、より高度な、対象スレッドを指定したデータ転送や、

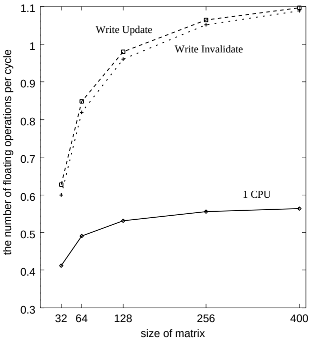


図 22: IP8

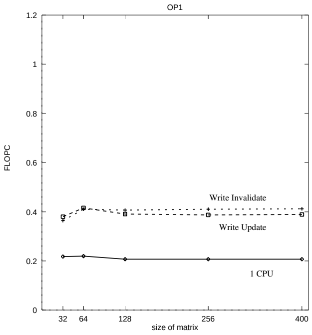


図 23: OP1

主記憶への書き戻し指定等の効果を評価する必要がある。MBP-light の完成後に複数クラスタでの性能評価が可能になれば、write-update と比較してソフトウェアからの指示によるデータ転送の正確性が高いことを立証したい。今後の研究の課題とする。

5 おわりに

本稿では、ハードウェア DSM における高速なデータ転送機能及び有効性管理機構が非常に有効である問題が存在することを示し、Cenju-4 を用いた性能評価結果からもハードウェア DSM が有効であることを示した。

一方では、ハードウェア DSM の固定的な一貫性制御方式では十分な性能が得られない問題があることを示した。

以上のような背景を元に、本稿ではソフトウェアからのヒントを利用するキャッシュ一貫性制御方式を提案した。本方式においては、データの性質を最も理解できるのはアプリケーション自体であるという信念の元に、共有データの一貫性制御に関する指示をアプリケーションから受け付けるようにした。この機能によって、生産者がアクセスしないデータをキャッシュに残さないように指示したり、消費者が分かるデータを生産時に消費者の近傍に転送することを指示したりできる。

本稿では、ソフトウェアからのヒントの実現法として、アドレス中で一貫性制御方式に関するヒントを与える方式を提案した。この方式では、TLB ヒット等の条件がそろった時には通常の書き込みと同等に低いコストでソフトウェアからの指示が可能である反面、様々な指示を織り混ぜて使用した場合にはページフォールトが発生したり、プロセッサ内の資源の有効利用率が下がるといった問題があった。このような問題を根本的に解決するためには、メモリアクセス時に対象アドレスと独立した、アクセスに関するソフトウェアからの指示を出力できる機構を持ったプロセッサの登場が待たれる。

本稿で提案した実装方式は、並列計算機 JUMP-1 に一部実装されており、また MBP-light 上のプログラムによって全機能を実現可能である。現在実装

が完了しているJUMP-1のクラスタ内で使用可能な機能としては、write-updateとwrite-invalidateのソフトウェアからの指示による切り替え等がある。LU分解の複数のアルゴリズムをそれぞれのプロトコルを用いて実行した結果から、write-updateとwrite-invalidateの切り替えが有効であることが示された。この切り替え機能は、本稿の主旨であるソフトウェアからの指示内容の中では最も単純なものの1つであり、MBP-lightが完成し次第、対象スレッド指定等の機能の効果を検証する必要がある。

最後に、本稿で提案するソフトウェアからのヒントを利用するキャッシュ一貫性制御方式の他に、ストリーム通信を管理する機構が必要となる問題が存在することを示し、ストリーム通信機構との融合方法に関する考察を行う。

5.1 ストリーム通信の実現

非同期的にプロセス間のデータ転送要求が発生する問題等においては、メッセージ(ストリームデータ)の管理コストが支配的になる場合がある。

メッセージを共有メモリ上で実現する場合、メッセージ用のバッファに対する排他制御のために生じるオーバヘッドを本質的に回避することができない。共有メモリ上でメッセージ通信を行う場合には一般に、遠隔メモリ上のメッセージバッファに対してロック操作を行う必要があり、ロック操作のレイテンシがメッセージ送受信の度にオーバヘッドとしてあらわれる。

システムが管理するメッセージバッファを利用する場合、メッセージ処理用のハードウェアがバッファを集中管理するため、共有メモリを用いた時のようなロックに伴うオーバヘッドは生じない。しかし、システムが管理するメッセージ通信用のハードウェアに対する保護のために、メッセージ通信の度にシステムコールが必要となる。

上記のような背景から、ユーザ空間上でストリーム管理を実現する、Virtual Queue [20] 等の機構が研究されている。

ストリーム通信が求められるデータ構造は様々である。よって、あらゆるサイズのストリーム管理を行う機構が理想ではあるが、動的にサイズが変化するデータのストリームをハードウェアで管理するのは困難である。よって、本稿では、ストリームデータの転送においては、共有メモリ上でデータ本体を転送し、固定サイズのストリーム通信機構を用いてアドレスを転送する方式を採用する。これによって、任意のサイズのデータに対するストリーム管理が実現できる。ストリーム管理が必要となる転送データ本体に関する排他制御は、転送データ生産者側で独自に管理する。図24において、VQと共有メモリを用いたストリーム通信の様子を示す。

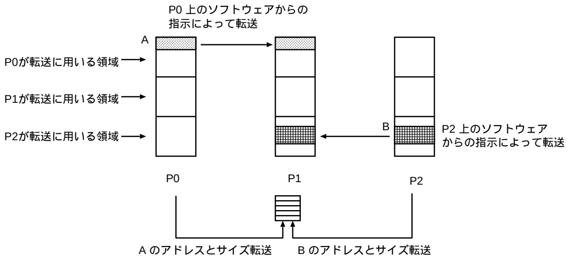


図24: 共有メモリとVQを利用したストリーム通信の実現

ストリームデータの消費者は、ハードウェアによってストリームが管理されている領域でアドレスを受け取り、そのアドレスが示すデータを順次消費する。ストリームデータの転送で使用される共有空間はそれぞれのプロセッサに書き込み権が与えられた空間であるため、ロック等の操作は必要ないが、逆依存等に関する管理はソフトウエアで行う必要がある。

参考文献

[1] 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 '95 論文集

[2] 広瀬 哲也 他, "並列コンピュータCenju-3のプロセッサ間通信方式とその評価" 情報処理学会論文誌 '96 Vol.37 No.7

[3] 松本 尚 他, "中粒度メモリベース通信を支援するMemory-Based Processor 2" 情報処理学会 98-ARC-130-18

[4] K.Gharachorloo 他, " Memory consistency

and event ordering in scalable shared-memory multiprocessors.” ICSA May 1990

[5] Pete Keleher 他, ” Lazy Release Consistency for Software Distributed Shared Memory ” ICSA March 1992

[6] Steven Cameron Woo 他, ”The SPLASH-2 Programs: Characterization and Methodological Considerations” ISCA June 1995

[7] S.C.Singh 他, ” The Performance Advantages of integrating Message Passing in Cache-Coherent Multiprocessors.” CSL-TR-93-593

[8] 細見 岳生 他, ” アップデート型コヒーレンス制御方式の評価 ” JSPP ’94 論文集

[9] 松本 尚 他, ” Memory-Based Processor による分散共有メモリ ” JSPP ’93 論文集

[10] Cristiana Amza 他, ”Adaptive Protocols for Software Distributed Shared Memory” IEEE Computer 1998

[11] Liviu Iftode 他, ” Improving Release-Consistent Shared Virtual Memory using Automatic Update ” IEEE Computer 1996

[12] Alain Raynaud 他, ” Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocessors ” HPCA 1996

[13] David A. Koufaty 他, ”Data Forwarding in Scalable Shared-Memory Multiprocessors” IEEE Computer 1996

[14] Matthias A. Blumrich 他, ”Virtual Memory Mapped Network Interface for the SHURIMP Multicomputer” ISCA 1994

[15] 堀江 健志 他, ” メッセージ通信の分散メモリ型並列計算機性能への影響 ” JSPP ’93 論文集

[16] 松本 尚 他, ”Memory-Based Processor による分散共有メモリ”
JSPP ’93 論文集

[17] 林 憲一 他, ” PUT/GET インターフェースのハードウェアサポートによる並列プログラムの効率的実行 ” JSPP ’94 論文集

[18] Cezary Dubnicki 他, ” Software Support for Virtual Memory-Mapped Communicaction ” International Parallel Processing Symposium 1996

[19] Liviu Iftode 他, ” Understanding Application Performance on Shared Virtual Memory Systems ” International Parallel Processing Symposium 1996

[20] 五島 正裕 他, ”Virtual Queue: 超並列計算機向きメッセージ通信機構” 情報処理学会論文誌 1996

[21] Hiroaki Nishi 他, ”The RDT Network Router Chip”
Asp-Dac Jan. 1997