特別研究報告書

ワークステーション・クラスタへのソフトウェア分散共有メモリシステムの実装

指導教官　富田 眞治 教授

京都大学工学部情報工学科

宋　栩華

平成9年2月14日

ワークステーション・クラスタへの
ソフトウェア分散共有メモリシステムの実装

宋　栩華

内容梗概

近年，分散メモリ型マルチプロセッサ計算機またはワークステーション・クラスタ上にソフトウェアによってサポートした仮想アドレス空間に関する研究が活発に行われている．分散共有メモリ (Distributed Shared Memory) は，分散メモリシステム上に構成された仮想的アドレス空間である．メッセージ通信方式に比べてユーザにより記述しやすいプログラミングモデルを与える．メッセージ通信方式では，プロセス間データの転送があるたびに，各プロセスがいつ通信するか，誰と通信するか，データの分割を明確に定めなければならない．それに対して，分散共有メモリ方式では，プログラマが直接的なプロセス間通信を避け，容易な記述ができる．したがって，分散共有メモリは共有メモリのやさしいプログラミングモデルと分散メモリ型計算機のスケーラビリティを兼ねる．本研究はネットワーク上のワークステーションクラスタ環境で，メッセージ通信による仮想的な分散共有メモリシステム (DSM) の構築を試みた．

分散共有メモリシステムの実装方式について，4 つのレベルがある：ハードウェアによる実現，オペレーティングシステムによる実現，コンパイラによる実現，ライブラリによる実現である．本研究は分散共有メモリ環境のためのライブラリを開発した．アプリケーションプログラムからランタイムライブラリをリンクすることによって，分散共有メモリ空間を利用できる．

本システムライブラリは，標準的な UNIX オペレーティングシステム上に実装された．UNIX 環境での移植性が高いといえる．またライブラリは C 言語のインタフェースを提供している．アプリケーションプログラマは明示的に共有データに対する READ/WRITE 操作を記述する．

本 DSM システムでは 2 種類のプロセスが存在する．ユーザアプリケーションプロセスとサーバプロセス．後者は共有メモリへのアクセスを一括して管理する．各ワークステーションマシンにおいてはただ一つのサーバを持つ．サーバは複数のユーザからの要求を受付ける．ユーザプロセスが共有メモリへのアクセス要求を出すと，その要求はサーバプロセスに伝えられ，そこで実際にメモリの共有が行われる．ユーザプロセスはライブラリ関数を介して，サーバプ

ロセスに通信する．

共有メモリへのアクセスを高速化するために，本システムではさらにソフトウェア制御のキャッシュメモリを実装した．ユーザプロセスは共有データを共有メモリからプロセスローカルメモリへコピーして高速に利用する．現在のところでは，キャッシュコヒーレンス制御に関しては，システムレベルでは自動的に制御を行わず，ユーザに任せる．すなわち，ユーザがライブラリより提供した関数を利用して，アプリケーションプログラムで，キャッシュを無効化したり，最新データを書き換え入れたりする．ユーザの負担は大きくなるが，自由なコヒーレンス制御が可能になる．

今後の課題としては，いくつかの共有データのパターンに適用するマルチコヒーレンスプロトコルをシステムに実装し，プログラマがアプリケーションの特徴に合わせて，共有データの型を決めるだけで，コヒーレンス制御をシステムに任せることが考えられる．

本文では，DSMシステムの構成及び実装について述べ複数のSun社のSPARCstation20を10MbpsのEthernetならびに80MbpsのMyrinetにより接続したワークステーションクラスタ環境で構築し評価を行なった．ソフトウェア通信オーバヘッドがシステム性能向上のボトルネックとなっている．ユーザレベルでの実装を行うが故に，DSMにおける並列処理は必ずUNIXカーネルを介して行われるため大きなオーバヘッドを伴い，効率良い並列処理を妨げていた．通信処理における入出力システムコールを最適化することが今後のもう一つ重要の課題となっている．

A Software-based Distributed Shared Memory System for Workstation Clusters

SONG XUHUA

Abstract

There has been much recent interest in supporting a shared address space in software across distributed memory multiprocessores or workstation clusters. Distributed shared memory has been recognized as an alternative programming model to exploit the parallelism in distributed memory systems since it porvides a higher level of abstraction than simple message passing. With message passing the programmer has to be aware of the data movements between processes, and each process has to know when to communicate, with whom to communicate and what data should be sent in a message. The distributed shared memory hides the remote communication from the application programmer and provides a simpler abstraction that the programmer understands well.

In short, Distributed shared memory combines the simple programming model of shared-memory with the scalability of distributed memory machines. Our system provides a Distributed Shared Memory (DSM) abstraction runs atop of message passing for a network of workstation cluster.

Distributed shared memory system can be basiclly classified in four main approaches: Hardware implementations, Operating system implementation, Compiler implementations and Library implementations. Our softwared-based DSM system is included in this latter class. The distributed shared memory abstraction is completely implemented by a run-time libaray that is linked with the application program.

The library was implemented on a standard UNIX operating system. It does not require any specail compiler or hardware. This is a key feature to obtain portability on UNIX environment. It also provides a easy-to-use C language programming interface. The READ/WRITE operations of all shared data should be declared explicitly by the application programmer.

In this DSM system there are two kinds of processes: user application processes and server processes. The latter ones are responsible for the management of shared data. There should be only one server in each machine. And Each server is responsible by some requests from one or more user processes to access to shared data practically. All the communication from user application process to server process is called by system library fuctions.

DSM also provides software controlled caches to hide a large amount of the communication latency. Each user application process has access to a local cache that is localed in its own address space where it keeps the copies of replicated data objects. An application programmer should be aware of cache coherence to invalidate or renew a cache replication which is updated in shared memory since thers isn't any automatical cache coherence protocol in the system. This approach gives to the user the freedom to choose the optimum coherence protocol though it impose a burden to the user. We plan to implement multiple coherence protocols tailored for several different types of data specified once in the next version.

This thesis describes the DSM implementation and presents some performance results taken on SunOS using SPARCstation20 connected by a 10Mbps Ethernet and 80Mbps Myrinet. Software communication overhead is the bottleneck in achieving high performance. The network bandwidth and overhead is one significantly important factor. And this implementation at user-level impedes the speed-up gained by parallel processing. This is because the DSM parallel processing is performed via the UNIX kernel which incurs a large overhead. We will put much emphasis on reducing the input/output system calls to improve the DSM performance in the future.

ワークステーション・クラスタへの
ソフトウェア分散共有メモリシステムの実装

目次

1 緒論

近年，LANに接続された多数の計算機の資源を有効に活用し，分散並列処理環境を構築するワークステーションクラスタ技術に関する研究が活発に行われている．

ネットワークの高速汎用化とマイクロプロセッサの高性能化に伴って，ワークステーションクラスタは1台のスーパーコンピュータのように振り舞うことが可能になる．

また，ネットワーク環境は，個々の固有の機能を持った独立性の高い計算機ノードを多数接続してシステムを構成するため，拡張が可能であるという利点を持ち，システム全体のコスト/性能比を良くすることもできる．さらに，複数のノード間で並列に処理を行うことにより，台数に見合った処理能力の向上が期待されている．

並列処理を実現する環境として，**PVI**(Parallel Virtual Machine)と**MPI** (Message Passing Interface)に代表されるメッセージ通信をベースとした研究や，共有メモリをユーザに提供するといった研究が多くの研究者によって進められている．並列処理を行なう上で通信方式の選択は重要な要素である．

メッセージ通信方式では，ユーザがプログラム中でのデータの分割形式、置く場所やプロセス間通信を明確に定める必要があるため，並列プログラムを書くのに熟練を要する．

その欠点を補うために，共有メモリ方式は，メッセージ通信を用いて仮想的に共有メモリを実現し，ユーザのプログラムの記述性を容易とするメモリREAD/WRITEを提供する．ワークステーションクラスタにおいて共有メモリによって記述されたプログラムは，共有メモリ型のマルチプロセッサ計算機に容易に移植できるという利点もある．

ワークステーションクラスタにおいて共有メモリを介して通信を行う場合，計算機のCPU能力に対してネットワークアクセス遅延が著しく大きい．キャッシュメモリを用いてネットワークのトラフィックを緩和することが必要となる．

最近大学における計算機環境では多数のワークステーションが配置されており，並列処理環境としての潜在的か構成を備えているが，多数のユーザで共同利用している環境では, OSなどのシステムソフトウェアを気軽に変更することは難しい．本研究では，UNIX OSのカーネルに手を加えずユーザレベルによる

ソフトウェアのみによって，ネットワーク上のワークステーションクラスタ環境でメッセージ通信による仮想的な分散共有メモリ構築を試みた．共有メモリへのアクセスを高速化するために，ソフトウェア制御のキャッシュメモリを実装した．C 言語のプログラミングインタフェースを支援し，ユーザの使い易さを意識した並列処理環境を提供する．

本 DSM システムプロセス間通信は UNIX の TCP ソケットによる実現されている．普通の Ethernet または高速 Myrinet ネットワークにより接続したワークステーション・クラスタ環境で構築した．性能評価よりネットワークのバンド幅と通信速度が分散共有メモリシステム性能にかかわる重要な要素であることを確認した．分散共有メモリを構築するもう一つ重要な問題は共有メモリのキャッシュ管理．すべてのアプリケーションに合うコヒーレンスプロトコルは存在しないため，システムライブラリからユーザにいくつかプロトコルを提供し，ユーザがアプリケーションの特徴にあわせて，自由なコヒーレンス制御できる．

以下，2 章では分散共有メモリ概要を述べる．また，3 章では DSM システム全体の構成について述べた後，その実現を 4、5、6 章に説明する．さらに，7 章では DSM システムに対する性能評価を行い，最後にまとめを述べる．

2　分散共有メモリ

分散共有メモリ (**DSM**: Distributed Shared Memory) とは，「物理的にメモリが分散配置されたシステム」上で「論理的に実現された共有メモリ」のことである．すなわち，分散共有メモリは，分散システム上に構成された仮想的なアドレス空間である．

DSM システムが分散メモリシステム（分散メモリマルチプロセッサまたはネットワークワークステーション）上に構築され，共有メモリのプログラミングモデルを与える．よって，分散メモリシステムのパフォーマンス性、スケーラビリティと共有メモリ型計算機上でのプログラミングの容易さとを兼ね備えることが DSM システムの最大な魅力点である．

ネットワークに接続されたワークステーションを仮想的な分散メモリ型並列計算機とした扱うワークステーションクラスタ上に，分散メモリシステムを構築することにより，安価に並列計算機を利用することができる．

2.1 メッセージ通信方式と共有メモリ方式の比較

並列処理を行なう上で通信方式の選択は重要な要素である．一般に，並列処理におけるプロセス間通信のプリミティブとしては，従来のメッセージ通信方式と分散仮想メモリを共有メモリとして用いることによる方式がある．**PVI**(Parallel Virtual Machine) と **MPI**(Message Passing Interface) などののライブラリを用いたメッセージ通信が分散並列処理を構築するワークステーションクラスタの通信方式の主流であった．

共有メモリを用いた方法では，メッセージ通信に比べてユーザが並列処理プログラムを記述しやすいという長所がある．DSM プロセス間の通信は，共有メモリによって実現されているために，プログラマにとってはその共有データを実際にどこに置かれているのを心配する必要がない．しかし，メッセージ通信方式では，プロセス間データの伝送があるたびに，各プロセスがいつ通信するか，誰と通信するか，データの分割を明確に定めなければならない．DSM システムがユーザをプロセス間メッセージ通信の煩わしさから解放される．ゆえに，DSM プログラムコードがメッセージ通信コードよりコンパクトで，分かりやすい利点がある．

一方，共有メモリ方式は，ハードウェア上のメッセージ通信を用いて仮想的に共有メモリを実現したため，直接メッセージ通信を利用する分散システムより，DSM システムのアプリケーション性能が落ちると予想されている．しかし，文献 [?] によって，多くのアプリケーションに対して，分散共有メモリシステムがメッセージ通信システムと競争できる性能を得られることを確認された．

よって，DSM は並列処理環境において，効率良いプログラミングモデルを与える．

2.2 ソフトウェア DSM の概念

ワークステーションクラスタにおいては物理的に共有されたメモリは存在しないため，ソフトウェアで共有メモリを実現する．そのためのアプローチとして，各プロセッサの局所メモリを共有メモリの一部として割り当て，これらへのアクセスはプロセッサ間の通信によって実現される．つまり，ハードウェア上のメッセージ通信を利用することによって，プログラマに仮想的な共有メモリを与える．

共有メモリに対するアクセスを分散させるため，システム全体の共有メモリ

を，あるサーバノードが一括して管理するのではなく，各ノードに共有メモリを分散して配置する．これによって，共有データへのアクセス及び管理負荷を分散させる．

共有メモリを介して通信を行う場合，キャッシュメモリを用いてネットワークのアクセス遅延を減少させ，ネットワークのトラフィックを緩和することが必要となる．この時，キャッシュメモリもソフトウェア的に制御される．

2.3 共有メモリの実装レベル

共有メモリモデルを構築する分散共有メモリシステムは，様々の形で研究されている．文献[5]に，種々の分散共有メモリシステムの実装方式についてまとめてある．実装のレベルを分類すると以下のようになる．

- ハードウェアによる実現
  キャッシュコヒーレンス制御とDSMプロトコルがハードウェアにサポートされる．
  DASH, PLUSなど．
- オペレーティングシステムによる実現
  仮想メモリ機構のページフォールトハンドラに手を加えて実装される．
  IVY, Mirage, Cloudsなど．
- コンパイラによる実現
  コンパイラがすべてのデータを分配、管理する．
  Orca, Amber, COOLなど．
- ライブラリによる実現
  OSカーネルに手を加えず，ランタイムなユーザライブラリの形で提供する．
  Munin, TreadMarks, ADSMITHなど．

ユーザの現存並列処理環境を考えると，ハードウェアレベルとOSカーネルレベルの実装は気軽くできない．

コンパイラによる実現する(compile-time)システムは比較的により良いパフォーマンスを得られる．それは，正確な分析から正規プログラムを生成し，ランタイムシステムのようなオーバーヘッドを避けることができるからである．しかし，正確な分析ができるアプリケーションプログラムが限られている．ランタイムシステムならがその制限を悩むこと必要はない．

ランタイムな(run-time)DSMライブラリは，共有メモリへのアクセスを動的に見つけてから，キャッシュコンシステンシプロトコルに応じて，メッセージを送る．コンパイラは静的に共有メモリへのアクセスを分析し，いくつかのパターンによってメッセージ通信プログラムを生成する．

ライブラリモデルを用いると，プログラミングモデルは分りやすいものを取り，ユーザは従来のプログラミングモデルの自然な拡張によって，複数のプロセッサを使用する並列処理環境が可能になる．

本研究のDSMシステムがランタイムなユーザライブラリによって実現されている．UNIXの基本カーネル以外，特別なハードウェア，OS，コンパイラのサポートが要らないため，移植性が高い．標準なワークステーションとオペレーティングシステムのUNIX環境上に，本ソフトウェアDSMシステムが低コストに構築される．

3　システム構成

本DSMは，UNIXオペレーティングシステムに変更を加えず，分散共有メモリモデルの並列処理環境をユーザに提供する．ユーザに共有メモリモデルの環境を提供することによって，プログラムの連続性を保証し，従来のアプリケーションプログラムの移植を容易にしている．

3.1　システムモデル

本DSMシステムモデルを図1に示す．

ネットワーク環境におけるノード要素（**Node**）はそれぞれ，分散システム上のワークステーションに対応している．各ノード要素のメインメモリから一部を割り当て，共有メモリとして確保する．図1のように各ノードはプロセッサユニット（**PU**）以外に，グローバルメモリ（**Mem**）とローカルメモリ（**LM**）を保持する．各グローバルメモリMemは共有メモリを構成し，単一のアドレス空間を形成している．ローカルメモリLMは共有メモリとして割り当てられたシステムグローバルメモリに対して，ローカルノードにあるメモリを示す．

ノードプロセッサ間通信によって，グローバルメモリへのアクセスがローカルメモリと同じようにできる．

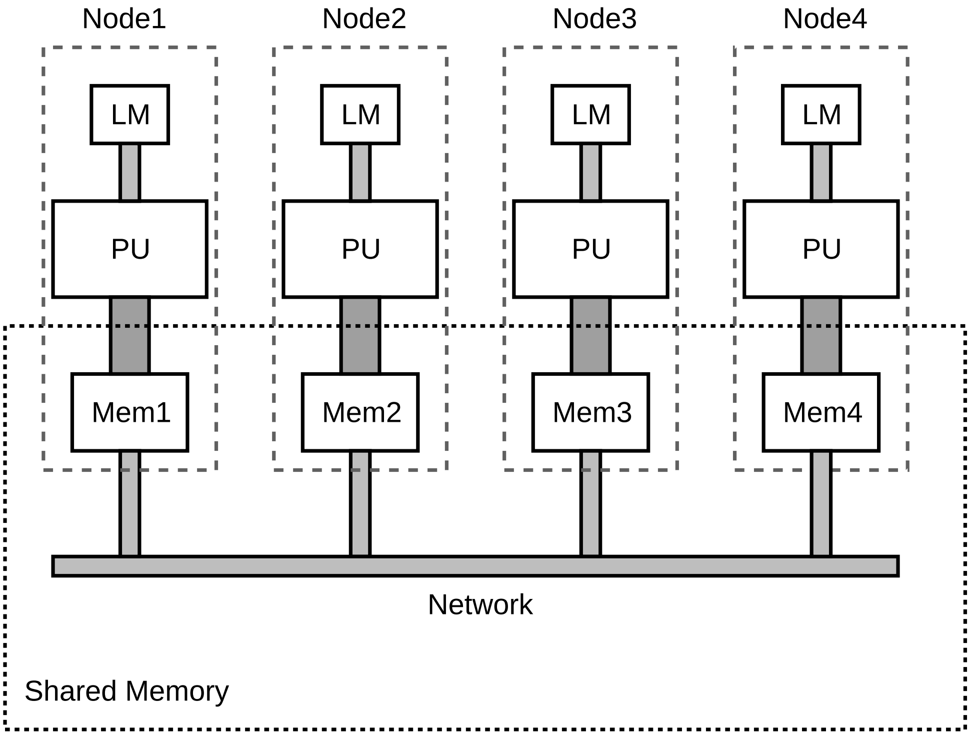


図 1: Distributed Shared Memory

3.2 ソフトウェア構成

次に，本システムのソフトウェア構成を図 2 に示す．

DSM は UNIX 環境にユーザレベルで実現されているため，並列処理の単位となるプロセスは基本的には UNIX プロセスであり，2 種類のプロセスが存在する :ユーザアプリケーションプロセス (**UP**) とサーバプロセス (**SP**)．UP プロセスはユーザのプログラムを実行する処理単位である．UNIX がユーザプロセスを生成するときに確保するメモリ空間をローカルメモリ (**LM**) とみなす．

サーバプロセスは DSM メモリ (データ領域) へのアクセスを一括して管理する．ユーザプロセスからグローバルメモリへアクセス要求に待ち応じて，DSM データを実際に読出し、書込みする処理単位である．

DSM サーバは並列処理のための基本機能であるプリミティブ処理を行い，実質的にユーザアプリケーションに並列処理機能を提供する部分である．もう一方のユーザプロセスは並列アプリケーションを実行するための空間を提供している．

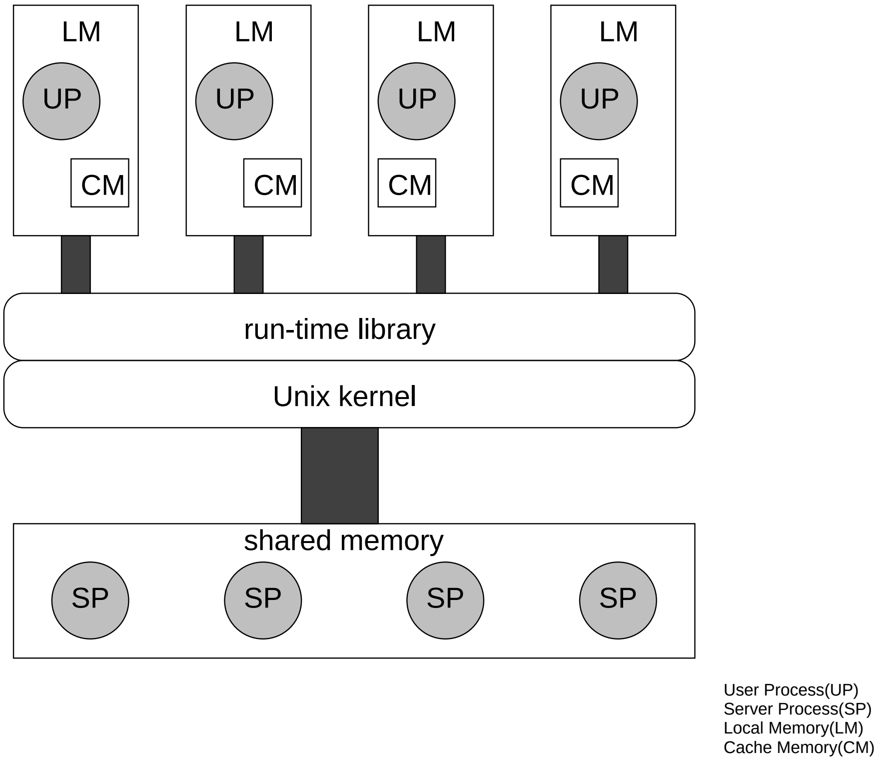


図 2: Distributed Shared Memory

ランタイムな DSM ライブラリはユーザプロセスとサーバプロセスとの仲介であり，ユーザに並列処理プログラミングインタフェースを提供する．ユーザはデータロケーションを考えないで，容易にプログラムを作成することができる．ユーザが DSM ライブラリ関数を呼び出す同時に，オペレーティングシステムコールを呼び出す．プロセス間の通信がメッセージ通信方式によって実現されている．

共有メモリをあるサーバノードが集中管理するのではなく，各ノードのサーバが自分のノードに分散配置されたグローバルメモリを管理する．これによって，共有データへのアクセス及び管理負荷を分散させる．

通常，１つのノード要素に対し，1 台のワークステーションを対応させるが．ノード要素の数に対してワークステーション数が不足する場合は，１つのワークステーションに複数のノード要素を割り付けることもできる．すなわち，1 台のワークステーションに対し，サーバプロセスを１つのみ対応させるが，ユーザプロセスが１つ以上存在することを認める．

ユーザプロセスがグローバル共有メモリのコピーをローカルメモリにある

キャッシュに持ってくる．

4　ソフトウェアキャッシュ

4.1　ソフトウェアキャッシュの定義

プログラムの命令やデータの参照の局所性を利用して，頻繁にアクセスされる命令やデータを実記憶装置より高速なメモリに自動できに移して利用することができる．これがハードウェアキャッシュメモリと呼ばれる方式です．

図2のように，DSMシステムにおいて，グローバルメモリへのアクセスは，ユーザプロセスとサーバプロセス間通信によって実現されているため，ユーザプロセスを生成するときの確保されたローカルメモリへのアクセスとのオーバーヘッドの差が大きい．このスピードギャップを克服するために，共有データをグローバルメモリからより高速なローカルメモリコピーして利用することができる．メモリアクセスのローカリティ（locality）を向上する．これは分散共有メモリにおけるキャッシングの基本的な考えである．

一般に計算機におけるキャッシュメモリシステムは，ハードウェア的に実現されている．これは，各メモリアクセスについて利用されるため，コストを低くする必要があるからである．すなわち，特別に設計されたハードウェアを用いるので，実行速度が速い．しかしその反面，開発及びシミュレーションが比較的困難である．仮に開発途中で仕様が変更されると，基盤の構成を換える必要があるため，仕様の変更に対処しにくい．大量に生産しない限り，実装時コストが高くなる．

ソフトウェアキャッシュはグローバルメモリからローカルメモリキャッシュへのコピーに通信を用いている．すなわち，分散メモリ型並列計算機のレベルで見た場合，ソフトウェアキャッシュへのコピー動作とは，ローカルメモリへの大量通信による通信集団（communication aggregation）として実現される．ソフトウェアキャッシュメモリ方式では，キャッシュリストの管理やコヒーレンス保持には，ハードウェアによる実現で実装したハードウェアキャッシュより要するコストが高い．しかし，ソフトウェアキャッシュが仕様の変更に対処しやすい，開発、実装コストが低下し，さらに拡張性に富むなどの利点がある．キャッシュの動作をソフトウェアによって制御することは可能である．

本システムは移植性を考慮した方針のしたに，特別なハードウェアのいらないソフトウェアキャッシュ方式を採用して，標準ワークステーションとUNIX

オペレーティング環境上に実装した．キャッシュリスト及びコヒーレンス管理はランタイムライブラリによって行われる．

ローカルキャッシュメモリに共有データが存在する確率（ヒット率）が高いほどキャッシュメモリの効果が大きい．キャッシュメモリに共有データが存在しない時（ミスヒット時）グローバルメモリより読出しを行う必要があり，命令の実行時間は長くなる．

4.2　ソフトウェアキャッシュリストの管理

共有メモリアクセスの時間的局所性と空間的局所性を利用し，キャッシュを活用することで，処理性能を向上する．DSMシステムグローバルメモリとキャッシュメモリの間は，ブロック（またはラインと呼ぶ）を単位に対応関係が管理され，データの転送が行われる．

アクセス要求されたデータを含むブロックがキャッシュメモリ内にあるかどうかの検索がキャッシュタグアレイ（cache tag array）を用いて高速に行われる．ソフトウェアキャッシュタグアレイの実装には高効率格納，検索できるハッシュ法より実現されている．

4.2.1　キャッシュハッシュテーブル

ハッシュ法(hashing)は，対象データを非常に効率よく格納し，検査する方法である．対象データをレコードと呼ぶことにし，レコードは互いに異なるキーを含むものとする．キーは格納されたレコードを検索する手段として使われる．それぞれのキーに対して，いわゆるハッシュ関数（hashing function）を適用し，格納または探索する．

ハッシュ法では，すべてのレコードを大きさの固定されたテーブルに格納する方法はクローズド（closed）ハッシュと呼ばれるものである．これに対して，レコード自身は別の記憶場所に格納し，それを指すポインタだけをテーブルに格納するオープン（open）ハッシュと呼ばれる方法を使うこともできる．われわれはオープンハッシュのほうを用いて，キャッシュハッシュテーブル（cache hash table）を実装した．

図3のようにキャッシュハッシュテーブルのキーは仮想共有メモリ空間の仮想アドレス**dsmAddr**である．ハッシュレコードにはキャッシュブロックへのポインタとキャッシュコヒーレンス制御ためのタグを含む構造体**CacheTableEntry**

を格納している．

- **Valid/Invalid** タグ
  そのキャッシュブロックがアクセス可能である時 Valid, アクセスできないとき Invalid となる．
- **Clean/Dirty** タグ
  グローバルデータの書き込み命令に際して，ライトスルー方式とライトバック方式がある．ライトバックは，キャッシュメモリにのみデータを書き，グローバルメモリへはそのブロックが削除されるときまたは置き換え対象となったときに書き込む．キャッシュブロックの初期値が Clean であるにたいして，一度でもライトバックされたら，タグが Dirty となる．

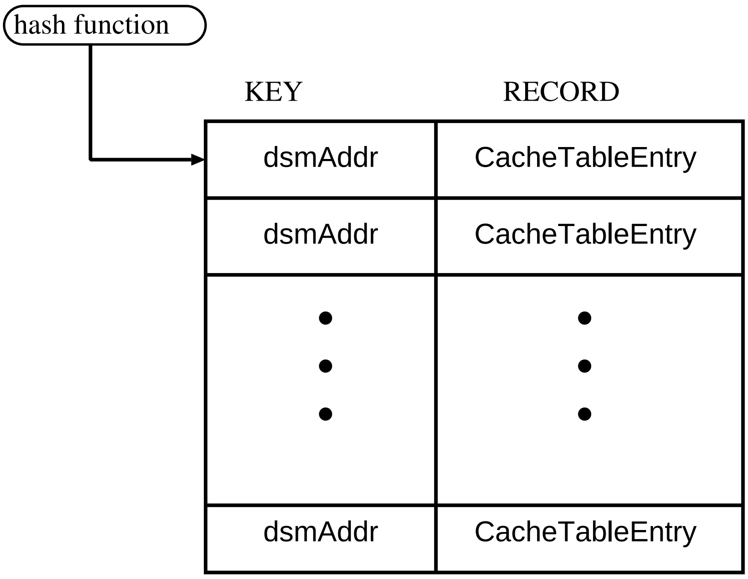


図 3: Cache Hash Table

ハッシュテーブルがほぼいっぱいになると，明らかに対象データに占有されていない空きの場所の探索時間は遅くなる．したがって，テーブルの大きさ（以下はエントリ（entry）数と呼ぶ）を実際に格納するレコード数より充分大きく取っておくことが推奨される．キャッシュに対して，どれだけのレコードを格納すること，つまりキャッシュサイズ（cache size）があらかじめに決めている．したがって，キャッシュハッシュテーブルのエントリ数をキャッシュメモリに収納可能なキャッシュブロック数（キャッシュレベル数）と呼ぶ．より十分おおきくすることによって，テーブル空間のほんの少しの部分だけを実際に使うこと

になる．

4.2.2 キャッシュリプレース方式

キャッシュメモリにデータが存在する確率（ヒット率）が高いほどキャッシュメモリの効果は大きい．したがって，キャッシュメモリ容量が大きいほど，キャッシュのヒット率が高いが，キャッシュの容量をどんどん大きくしてしまうと，探索時間が遅れるので，キャッシュする意味がなくなる．適当なキャッシュサイズを決める必要がある．本 DSM システムでは，キャッシュのレベル数によって，キャッシュメモリ容量を制限する．

キャッシュメモリの容量がすべて利用されている状態で，さらにキャッシュメモリにないメモリブロックへのアクセス要求があると，キャッシュメモリのあるブロックを置き換えねばならない．いろんな置き換えアルゴリズムによる実現方式がある．LRU（Least Recently Used）のキャッシュリプレース方式がよく使用される．この方式では最も遠い過去に参照されたブロックが置き換えられる．しかし，キャッシュレベル数が大きいと，LRU リストを管理するために探索時間をかかってしまうので、効率が悪くなることもある．現在の実装では，キャッシュメモリが溢れた場合には，システムがランダムに一つのキャッシュブロックを選び，置き換える．

4.3 キャッシュコヒーレンス管理の方針

- structure-based DSM system
- page-based DSM system
- ユーザ指定による方式

仮想空間を共有する場合，システムグローバルメモリとキャッシュメモリの間は，ブロックを単位に対応関係がつけられ，データの転送が行われる．同一ブロックのコピーが，複数のユーザプロセスローカル上のキャッシュ内に存在する可能性があり，キャッシュへの書込動作によって，内容の統一性が損なわれる可能性がある．ブロードキャストによってすべてのユーザプロセスへ知らせは，実現しやすいけれども，ネットワークに多大な負荷を与える．

本研究のソフトウェアキャッシュシステムには次の二つのプロトコルを実装し，ユーザが選択できるようにしている．

- **write invalidate**

  DSM グローバルメモリへの書き込み時に全コピーを無効にする．ただその

グローバルメモリを書き換えたユーザプロセスのみ，ローカルキャッシュメモリに更新したデータをもつ．ほかのユーザがそのグローバルメモリの対応するキャッシュブロックに対して，読み出しまた書き込みがあるときは，キャッシュヒットミスによって，実際にグローバルメモリにアクセスする．

- **write update**

  DSM グローバルメモリへの書き込み時に全コピーを更新する．この場合は無効化プロトコルのようにキャッシュヒットミスが起こらない．データの転送が実際に行われる．書き込み更新化プロトコルは，プロセス間の衝突が多い場合，例えば1回の書き込みに対して複数のユーザプロセスから参照が起こる場合に有効である．

二つのプロトコルはそれぞれ利点と欠点がある．無効化プロトコルはキャッシュデータを無効化するだけで，実際にデータを更新する必要がなく，更新化プロトコルより，コヒーレンス関係の小さなメッセージ通信ですませる．書き込み無効化プロトコルは，ほかのユーザプロセスに参照される前に，1つプロセスによる書込が多く行われる場合に有効である．

更新化プロトコルがキャッシュヒットミスの可能性を避免する．書き込み更新化プロトコルは，プロセス間の衝突が多い場合，例えば1回の書き込みに対して複数のユーザプロセスから参照が起こる場合に有効である．どのプロトコルを選択するのがグローバルデータのサイズと READ / WRITE アクセスの比率にかかわる．

また，1つのグローバルデータに対して，複数のユーザプロセスからの書き込みがよく行われる場合，コヒーレンス制御するにはより大量の通信を用いてしまうため，キャッシュメモリにもて来る価値がない．ユーザプロセスがサーバプロセスを経て毎回直接 DSM グローバルメモリへアクセスするほうが逆に効率が良い．

本 DSM では，ユーザ指定による方式をとり，ユーザがグローバルデータの書き込みを指定するインタフェース（DSM ライブラリ）を用意している．システムレベルでは自動的にコヒーレンス制御を行わず，ユーザが明示的にグローバルデータの書込を指示することで，自由なコヒーレンス制御の記述を可能にしている．自由な制御が可能になるかわり，当然ユーザの負担は大きくなる．しかし，すべてのアプリケーションに対して，1つ適応している DSM アルゴリズムはない．各アプリケーションはそれぞれの特徴をもち，そのアプリケーショ

ンの特徴に相応しいコヒーレンスプロトコルを選ぶの最も重要だと思われる．

5　DSM システムサーバ

5.1　共有メモリの実現

サーバプロセスではまず，分散システム上のノードマシンのメインメモリから一部をファイルに割り当て，共有メモリとして確保する．共有メモリへのアクセスはファイルへのアクセスに置き換えられる．これはUNIX 提供する仮想記憶システム関数 mmap() を利用して実現する．mmap() はプロセスが持つ仮想アドレス空間を仮想共有メモリとしてのファイルにマップする．マッピングが成立すれば，プロセスからの共有メモリアクセスはUNIX システムによって，自動的にファイルへのアクセスに置き換えられる．

各ノードにファイルとして割り当てられ，確保された仮想共有メモリの大きさはDSM システムヘッダファイルに **SHARED_MEMORY** という記号定数として定義されている．ユーザのアプリケーション並列プログラムにあわせて自由に変更できる．図 4 に共有メモリ空間の構成を示す．

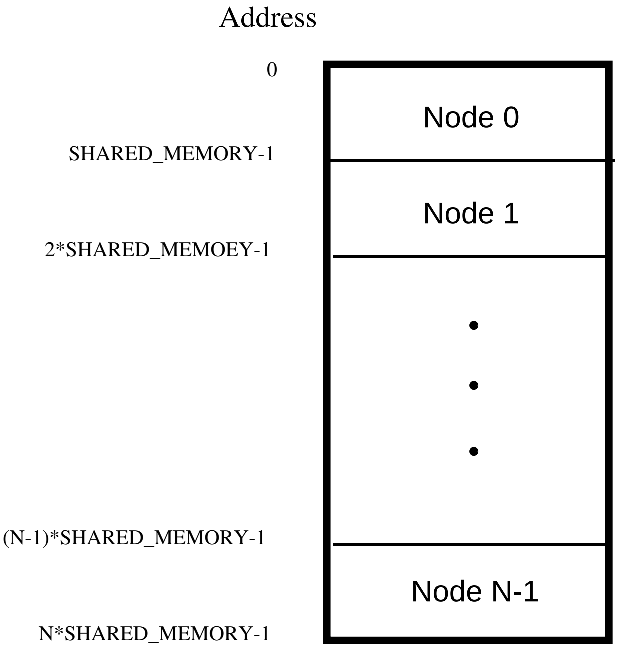


図 4: 共有メモリ空間

共有メモリのグローバルデータへの管理負荷を分散させるため，あるサーバ

ノードが一括して管理するのではなく，各ノードにサーバプロセスを設置する．

ユーザプロセスはサーバプロセスを経由することで，実際にデータが放置されている場所に関係なく，DSMグローバルデータ領域にアクセス可能である．サーバプロセスがユーザプロセスからグローバルメモリのアクセス要求を待ち応じて，データを実際に読出し、書き込みする．

5.2 間接アクセスモデルと直接アクセスモデル

DSMサーバは，ユーザプロセスのキャッシュに対するアクセスミス時に必要なデータ転送や，ほかのユーザプロセスキャッシュと共有しているメモリブロックに対する書込時に，そのユーザプロセスにキャッシュブロックの無効化/更新を依頼するメッセージなどの作成を行い，その要求メッセージへの応答に対する処理を行う．

- 基本アクセス手順

  図5のように間接アクセスモデルシステムサーバによってグローバルデータ領域へのアクセスは以下の手順で実行される．

  1. ユーザプロセスで呼び出されるDSMライブラリのグローバルデータアクセス関数はノード内のDSMサーバプロセスに要求を送る．
  2. アクセス・データが自己の局所グローバルメモリにあれば，DSMサーバはそれにアクセスして，結果をユーザに返す．
  3. それ以外の場合，アクセス・データを保持する局所メモリを管理しているノードのDSMサーバに要求を送る．
  4. の要求を受け取ったDSMサーバは局所メモリにアクセスして，ユーザに返す．

  それに対して，図6のような直接アクセスモデルシステムサーバによってグローバルデータ領域へのアクセスは以下の手順で実行される．

  1. ユーザプロセスで呼び出されるDSMライブラリのグローバルデータアクセス関数はそのデータを保持する局所メモリを管理しているノードのサーバに直接に要求メッセージを送る．
  2. サーバがそれにアクセスして，結果をユーザに返す．

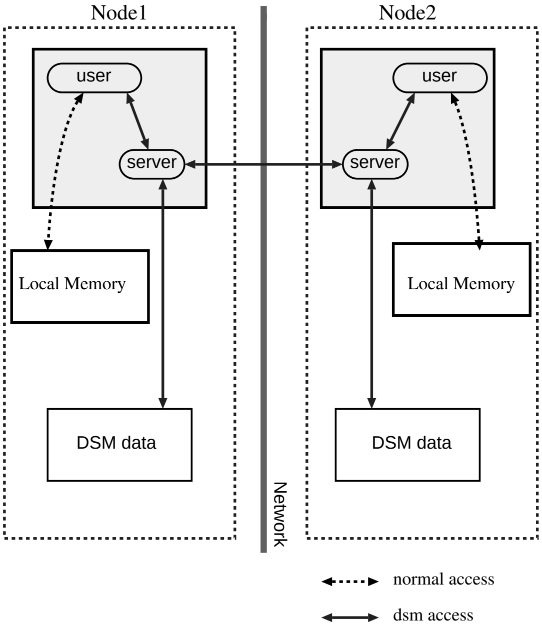


図 5: 間接アクセスモデル

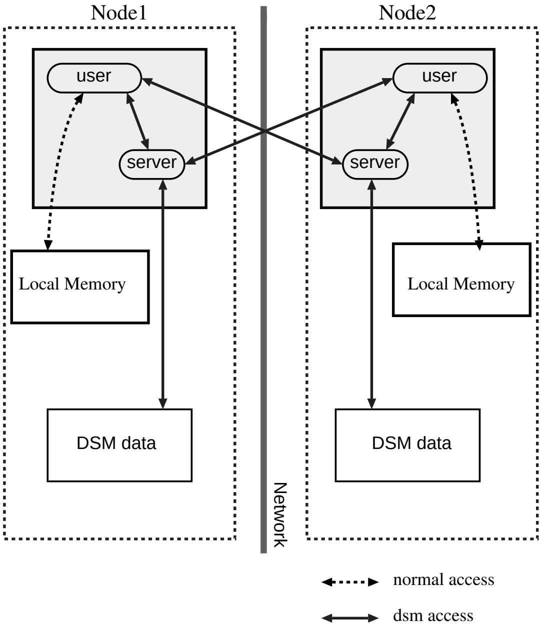


図 6: 直接アクセスモデル

- アドレス変換

  共有メモリの各バイトに仮想アドレス 0 番地，1 番地，2 番地，... N 番地

を仮定し，仮想アドレスがint型の4バイト（32ビット）で構成されており、ユーザはこの仮想アドレス空間を利用することができる．仮想アドレスから物理アドレスへの変換は2段階に分られる．
間接アクセスサーバモデルでは，サーバプロセスが同一マシンのユーザプロセスから依頼メッセージを受け取る．サーバプロセスはその仮想アドレスがどのサーバの局所グローバルメモリに当るのかを計算して，自ノードでなければ，そのサーバへアクセス要求メッセージが送られる．要求に受けたDSMサーバは，UNIXマッピングシステム機構によって，自動的に仮想アドレスから物理アドレスに変換を行うことでアドレス変換を終える．

- 排他制御
  ユーザ - サーバ間でSEND/RECEIVEによるプロセス間通信機構を用いる場合，最も簡単な方法は，サーバを一つのプロセスで実現するというものである．

```
while (TRUE)  {
          ReceiveAny(pid, message ) ;
          do_it ( ) ;
}
```

この方式では，要求されたサービスを実行する(do_it)のに要する時間が長いと，応答性の面で問題があるが，実現が簡単なので，要求の発生がそれほど頻繁でなくサービスの実行に要する時間が短いと，また応答性がそれほど問題でないとき有用である．応答性の問題を解決する一つの方法は，サービスごとにプロセスを割り当てる方法である．仕事の切分け（スケジューリング）はカーネルに任せておくことができ、実現が容易である．あらかじめいくつかのプロセスを生成しておく方法もあるが，本DSMシステムでは，図7のような形でプログラムする．
この場合，時間のかかる処理を子プロセスに任せながら，サーバは次の処理を処理することができる．共有する資源を同時に二つ以上の子プロセスが使用しないように相互排除を制御するする必要ある．本DSMシステムは，共有メモリの危険区域に対して，UNIX同期機構セマフォを用いる．

```
while (TRUE)  {
          ReceiveAny (pid, message) ;
          if ((pid = fork () ) == 0 )  {
            do_it ( ) ;
          }
}
```

図 7: サービスごとにプロセスの割り当て

```
int acquire_lock(int semid)
{
    /* the array of semaphore-operation structures */
    static struct sembuf acquire={0, -1, SEM_UNDO};

    acquire.sem_num = 0;     /* semaphore number */
    if (semop(semid, &acquire, 1) == -1) {
        perror("Server: semaphore acquire error.\n");
        return(1);
    }

    return(0);
}

int release_lock(int semid)
{
    static struct sembuf release={0, 1, SEM_UNDO};

    release.sem_num = 0;
    if (semop(semid, &release, 1) == -1) {
        perror("Server: semaphore release error.\n");
        return(1);
    }
    return(0);
}
```

図 8: サーバプロセスの排他制御

6　DSM システムライブラリ

DSM システムライブラリはユーザに C 言語のアプリケーションインタフェースを提供する．基本命令は図 9 に示す．ライブラリでは複数のコヒーレンス制御プロトコルをサポートし，用途に応じた関数をプログラマが選択できるようにしている．ユーザは DSM ライブラリとして提供する並列処理のための関数群　(共有メモリアクセス，キャッシュ管理，排他制御）を利用し並列プログラムを記述する．

ライブラリ関数

- dsm_init(int argc, char *argv[ ]);
- dsm_quit(int argc);
- dsm_TS(int lock, int address);
- dsm_read(char *local, int address, int length);
- dsm_write(char *local, int address, int length);
- cache_read(char *local, int address, int length);
- cache_write_wb(char *local, int address, int length);
- cache_write_wt(char *local, int address, int length);
- cache_invalidate(int address, int length);
- cache_update(int address, int length)
- cache_wb_dsm(int address, int length);
- cache_flush(void);

図 9: DSM primitives

関数の引数 address と length に，それぞれグローバルデータの仮想アドレスとデータサイズを指定する．引数 local が送られたグローバルデータの値へのポインタを返す．関数が実行成功すれば 1 を，失敗すれば 0 を関数に返す．

6.1　初期化と終了命令

ユーザの並列アプリケーションプログラムまず初期化命令 **dsm_init()** を呼び出さなければならない．初期化命令では，実行ファイルの引数として渡された

サーバの情報を受け取り，ユーザプロセスとサーバプロセスとの通信路を確保する．プログラムの最後に終了命令 **dsm_quit()** を呼び出すことにより，サーバプロセスとの通信路を切断する．しかしフルサーババージョンでは，ユーザプロセスが同一ノードのサーバしか通信しないため，サーバ情報を渡さなくても，初期化命令が勝手に同一ノードのサーバプロセスとの通信路を設立、切断する．

6.2　読出しと書込み操作

アプリケーションプログラマはすべての共有グローバルデータおよび読出し、書込み操作を明示的に記述する必要がある．プロセス間の通信が読出し/書込み操作により実現されている．プログラマが READ と WRITE 関数を呼び出す，そのデータの実体がどこにあるのかを心配する必要がない．

- **dsm_read()**
  DSM システムの基本リードアクセス命令である．
  シンプルサーババージョンでは，ライブラリ関数で仮想アドレスを計算し，その仮想アドレスに当る仮想空間を管理するノードのサーバプロセスに直接リード要求メッセージを送る．
  フルサーババージョンでは，ライブラリ関数がまず自己ノードのサーバプロセスにリード要求を依頼し，サーバプロセスがその仮想アドレスを計算する．自分の管理範囲内にあると，グローバルメモリにアクセスして，ユーザプロセスにデータを返す．そうでなければさら，その仮想アドレスに当る仮想空間を管理するノードのサーバプロセスに要求メッセージを送り出す．
- **dsm_write()**
  DSM システムの基本ライトアクセス命令である．
  基本手順は dsm_read() 関数と同じようにサーバプロセスを経由して，グローバルメモリにライトアクセスする．

前に述べたように，グローバルメモリへのアクセスは，ユーザプロセスとサーバプロセス間（ノード内またはノード間）通信によって実現されているため，ユーザプロセスを生成するときに確保されたローカルメモリへのアクセスとのオーバーヘッドの差が大きい．グローバルデータを共有メモリからより高速なローカルメモリにキャッシュして，システム性能を向上させる．ライブラリは以下

のようなローカルアクセス関数を提供する．

- **cache_read()** まずリードアクセス要求されたデータを含むブロックがキャッシュメモリにあるかどうかをローカルメモリにあるキャッシュタグアレイに調べる．キャッシュヒットの場合にデータをユーザプロセスに返す．
  キャッシュヒットミスの場合（キャッシュに存在しないまたは無効タグ Invalid がついている）は，dsm_read() 関数を呼び出し，該当するサーバに依頼し，グローバルメモリから実際にデータを読出しユーザに返すと同時に，キャッシュタグアレイに挿入する．

共有メモリ書き込み命令の実行に際して，グローバルメモリとローカルのキャッシュメモリの内容を一致させるための方式として，ライトスルー (write through) とライトバック (write back) の 2 つの方式がある．

- **cache_write_wb()**
  ライトバック方式を実現している．キャッシュメモリにのみデータを書き，グローバルメモリへはそのブロックが削除されるときまたは置き換え対象となったときに書き込む．キャッシュブロックのみの読み書きとなるので、高速実行が可能である．
  この方式はグローバルデータに対して，ほかのユーザプロセスに参照される前に，1 つのプロセスによる書込が多く行われる場合に有効である．更新されたデータは，後で説明する cache_wb_dsm() 関数または cache_flush() 関数によって，グローバルメモリに書き戻す．
- **cache_write_wt()**
  ライトスルー方式を実現している．キャッシュメモリにデータが書き込まれるとき，グローバルメモリにも同時に書き込む．毎回の書き込み時に共有メモリアクセスを必要とし、アクセス時間が長くなる．
  この方式は最新データがグローバルメモリにあるので，1 回の書き込みに対して多数のユーザプロセスから参照が起こる場合に有効である．

さらに，プログラマが次のライブラリ関数を使って，各ユーザプロセスのローカルキャッシュコヒーレンス制御をする．

- **cache_wb_dsm()**
  ローカルメモリにライトバック関数により更新されたキャッシュデータブ

ロックをグローバルメモリに書き戻す関数である．

- **cache_invalidate()**

  あるキャッシュブロックを無効化する関数である．そのブロックに対して，読出しまた書込みがあるときは，キャッシュヒットミスによって，実際にグローバルメモリにアクセスする．

  DSM グローバルメモリへの書き込み時に，プログラマがこのキャッシュ無効化関数を利用して，書込無効化プロトコルを実現する．

- **cache_update()**

  あるキャッシュブロックがもつデータを更新する関数である．DSM グローバルメモリへの書き込み時に，プログラマがこのキャッシュ更新化関数を利用して，書込更新化プロトコルを実現する．

- **cache_flush()**

  ユーザプロセスが持つキャッシュメモリをすべてクリアする関数である．

6.3 排他制御

複数のユーザプロセスがほぼ同時に同じグローバルメモリ領域へアクセスを行った場合，スラッシングが生じてシステムの性能を低下させる危険性がある．そこで，ロックすることによりアクセスの途中で無効化されることを防止する．

DSM ライブラリは基本同期命令であるテストアンドセット命令

- **dsm_TS()**

を提供し，排他制御を行う．TS 命令はグローバルメモリにおかれた制御変数 (ロック変数）L に対して

```
dsm_TS ( lock , address )  {
        status <== load (address) ;
        store (address ) <== lock ;
        return (status) ;
}
```

を不可分に実行する．すなわち，グローバルメモリでは，L に対するリードとライトがあるが，これを切り離した操作ではなく，1 つの操作として実行し，その実行中にほかのプロセスからのグローバルアクセスを禁止する．プログラマが次のように TS 関数をりようすることによって，排他的に並列処理を行う．

```
WHILE(dsm_TS(1,address)==1) /* ロックする */
access to DSM global_data;
dsm_TS(0,address); /* アンロックする */
```

7　評価

7.1　実験環境

本システムを，複数のSun社のSPARCstation20をMyricom社のMyrinet(80Mbps)また標準Ethernetにより接続したワークステーションクラスタ環境で構築し評価を行なった．

7.2　基本アクセス時間の評価

ワークステーション・クラスタにおいては物理的な共有メモリは存在しないため，DSMシステムではメッセージ通信を用いた仮想的な共有メモリを実現している．そのため，分散並列処理環境を構築するに当っては通信速度が重要になってくる．

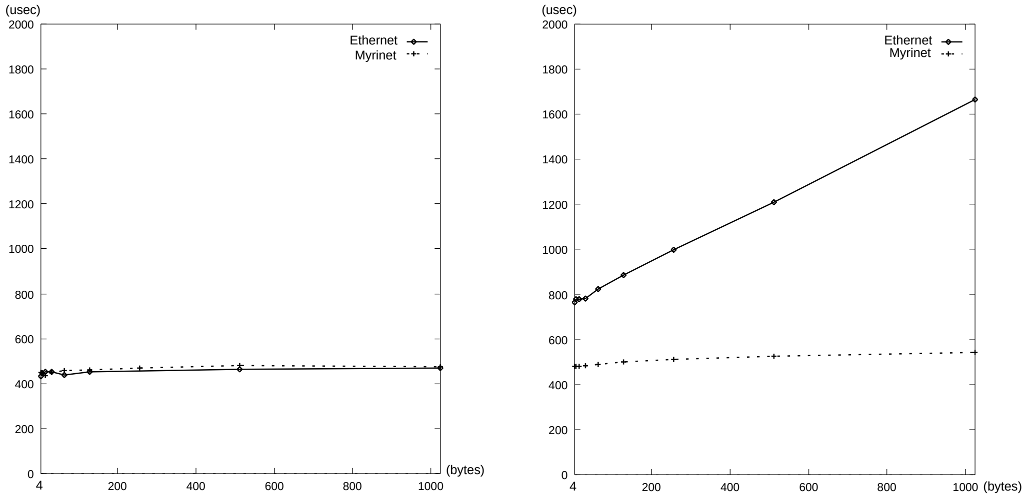
Ethernetはバス型の接続形態であるため，接続されるマシンが増えるにしたがって，通信量が増大し，パケットの衝突（collision）も発生しやすくなる．衝突が発生すると適当な時間待ちを行って再送するため，一定時間内にデータが到着するという保証がない．その通信速度遅いため，性能の向上が難しい．ワークステーション・クラスタにおいては，高速なネットワークの導入が不可欠である．MyrinetをEthernetと比較すると，1) バンド幅が80Mバイト/秒と非常に高い，2) ハードウェアのフロー制御によってメッセージの配送が保証される，3) 同じバスを通るメッセージの順序は保証される，という点に特徴がある．

まず，EthernetとMyrinet上にそれぞれの共有メモリへの基本アクセス時間を測定した結果を図9に示す．

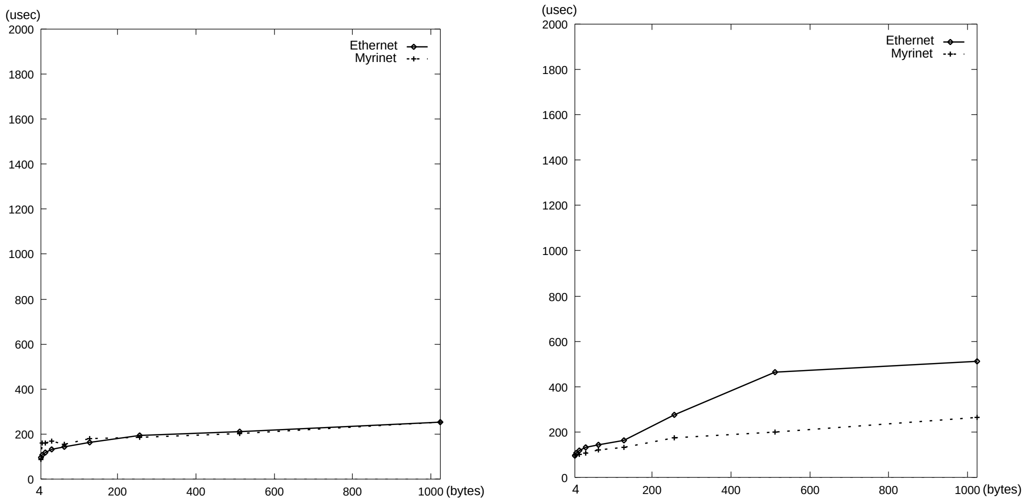
ネットワークを介して他ノードの共有メモリにアクセスする場合，データの量が増えると，Ethernetの処理時間も大きく増え続ける．それに対して，Myrinetの処理時間はそれほど増加していない．ここで，ネットワークのバンド幅と通信速度が分散共有メモリシステム性能にかかわる重要な要素であることを確認した．

READ操作に対して，WRITE操作のオーバヘッドが格段的に速い原因は，

ユーザはサーバに共有メモリへの書き込み要求を出すだけ，ACK信号を待たずに次の命令に進む．書き込みが失敗するとサーバプロセスからエラーメッセージを出しますので，書き込みの信頼性を確保し，処理速度も速くなる．



(a)dsm_read from local node　(b)dsm_read from remote node



(c)dsm_write to local node　(d)dsm_write to remote node

図9: 共有メモリアクセス基本性能

最小のメッセージサイズをリードするには，400 マイクロ秒以上かかります．システムとしては，良い性能を得られてない．本 DSM システム通信機構をソケットを用いて実現しているので，データのコピー処理が重く，プリミティブ処理における通信処理の割合が高くなる原因となっていることが考えられる．また DSM が行う通信が必ず UNIX カーネルを介する必要があり，通信処理に大きなコストを要するからである．つまり，入出力システムコールの処理時間は DSM システムの処理効率に重大な影響を与えていることも考えられる．

7.3　並列プログラムによる評価

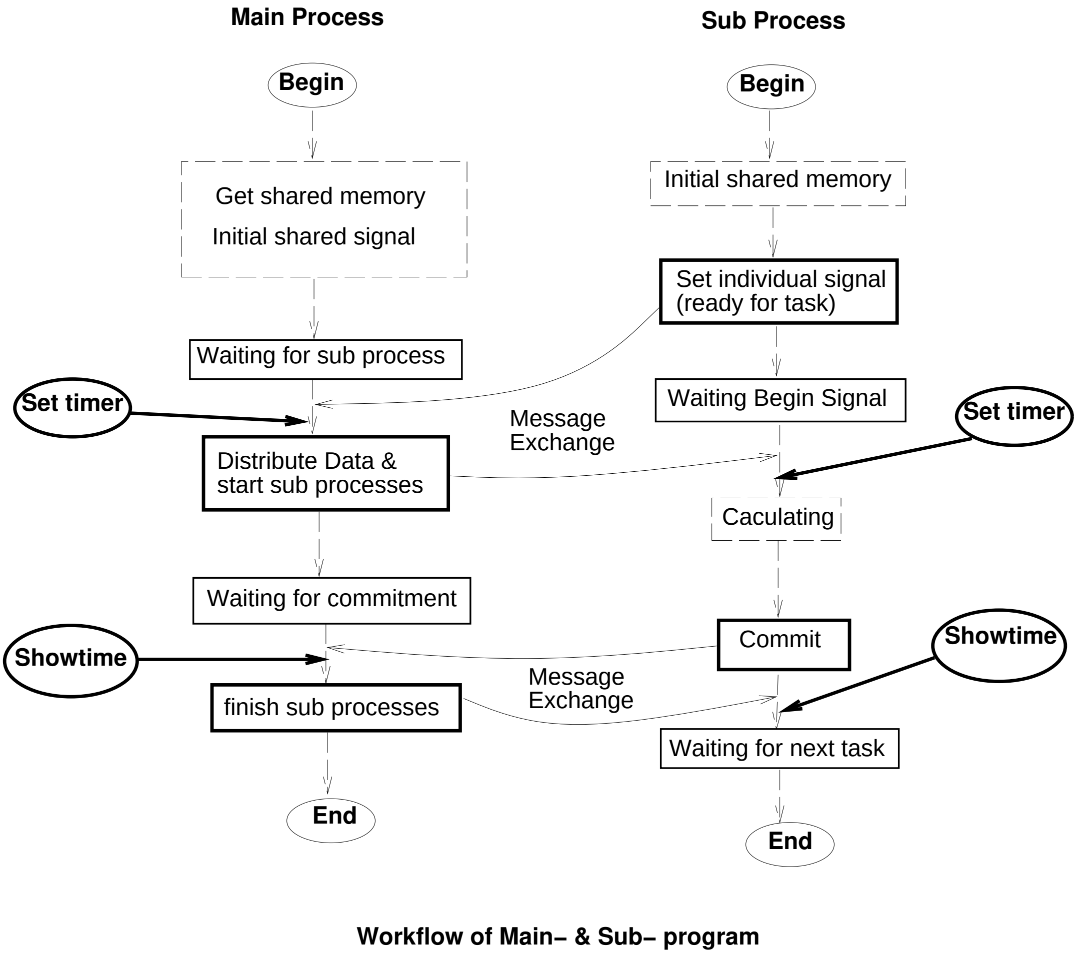


図 10: 並列プログラムの流れ図

行列∗行列並列プログラムで，Ethernet と Myrinet 上に本システムの性能評

価も行った．現在の実装レベルでは，ネットワークの性能より，ソフトウェア通信オーバヘッドがシステム性能向上のボトルネックとなっていることを判明した．

8　まとめ

本DSMシステムは標準的なUNIX上に実装した．UNIX環境での移植性が高いといえる．アプリケーションユーザにC言語のシステムライブラリインタフェースとシステムサーバプログラムを提供する．システムプロセス間の通信は共有メモリを介して実現されている．サーバプロセスは共有メモリへのアクセスを一括して管理する．

共有メモリへのアクセスを高速化するためにシステムにソフトウェア制御のキャッシュメモリを実装した．プログラマはアプリケーションの特徴に合わせて，ライブラリのキャッシュ関数を用いて，自由なコヒーレンス制御ができる．

本DSMシステムは研究室現存のワークステーションクラスタ環境上低コストに構築された．ワークステーションを10MbpsのEthernetならびに80MbpsのMyrinetネットワークにより接続した．性能評価よりネットワークのバンド幅と通信速度が分散共有メモリシステムの性能にかかわる重要な要素であることを確認した．さらに，ソフトウェア通信オーバヘッドが現在のシステム性能向上のボトルネックとなっていることを分った．

今後の課題としては，通信処理における入出力UNIXシステムコールを最適化し，通信オーバヘッドを減少することを検証する．また，DSMシステムレベルで複数のキャッシュコヒーレンス制御のプロトコルを実装し，ユーザの負担を減少する．ユーザにより使いやすく効率の良い分散並列処理環境を提供する．

謝辞

本研究の機会を与えてくださった、富田眞治教授に深く感謝の意を表します．

数々の御指導、御助言を賜った中島浩助教授、森眞一郎助教授、五島正裕助手に心から感謝いたします．

OSグループの共同研究者である、青木秀貴氏、山添博史氏をはじめ，計算機工学講座計算機構成分野の諸氏に感謝いたします．

また，この場をお借りして，筆者を支えてくださいました友人朱凱氏、村上友紀恵氏にお礼申し上げます．

参考文献

[1] Masao Yakushijin, Hironori Nakajo and Yukio Kaneda. *Construction of DSM system using Distributed Function Network no WS Clusters.* 計算機アーキテクチャ, 119-24, 28 Aug 1996, pp.137-142.

[2] Hironori Nakajo, Newton K. Miura, Koichi Wada and Yukio Kaneda. ネットワーク結合型並列計算機上の仮想共有メモリシステムにおける無矛盾化プロトコルの性能評価とハードウェアによる実現. 並列処理シンポジウム JSPP '91, May 1991, pp.45-52.

[3] Takeshi Nanri, Hiroyuki Sato and Masaaki Shimasaki. *Using Software Cache Controlling Compiler for Improving Memory Performance on Distributed Shared Memory.* 並列処理シンポジウム JSPP '96, Jun 1996, pp.331-338.

[4] Richard B.Gillett. *Memory Channel Network for PCI.* IEEE, vol.16, No.1, Feb 1996, pp.12-18.

[5] Matthias A.B., Kai Li, Richard A., Cezary D., and Edward W.F. *Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer.* Proceedings of the 21st Annual International Symposium on Computer Architecture, Apr 1994, pp.142-153.

[6] 手塚宏史, 堀敦史, 石川裕. ワークステーションクラスタ用通信ライブラリ PM の設計と実装. 並列処理シンポジウム JSPP'96, 情報処理学会, Jun 1996.

[7] 石川裕, 堀敦史, 小中裕喜, 前田宗則, 友清孝志. 並列プログラミング言語 MPC ＋＋の実現. 並列処理シンポジウム JSPP'96, 情報処理学会, Jun 1996.

[8] Luis M.Silva, Joao Gabriel Silva, Simon Chapple. *Implementing Distributed Shared Memory on top of MPI: the DSMPI Library.* Proceedings of PDP '96, IEEE, pp.50-57.

[9] Yoshimasa Ohnishi, Kiyoshi Ryoukai, Toshinori Sueyoshi. 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，Vol. 36 No.7, July 1995, pp.1729-1733.

[10] 南里豪志，佐藤周行，島崎眞昭ソフトウェアキャッシュの制御を行なう最適化コンパイラによる分散共有メモリシステムのメモリ性能改善．並列処理シンポジウム JSPP '96, 情報処理学会, Aug 1996, pp.331-339.