

分散共有メモリモデルに基づく HPC 環境の高速化実験

了戒 清 手塚 忠則† 末吉 敏則

九州工業大学 情報工学部 知能情報工学科

e-mail: kao@mickey.ai.kyutech.ac.jp

sueyoshi@ai.kyutech.ac.jp

概要

本稿で紹介する分散スーパーコンピューティング環境 (DSE) は、分散システムにおいて既存のオペレーティングシステムに変更を加えずに、分散共有メモリモデルに基づく並列処理機能を提供するハイパフォーマンスコンピューティング環境である。DSE は移植性を考慮してユーザプログラムレベルで実装を行っているため、DSE 内の通信は必ず UNIX カーネルを介して行われ、DSE における並列処理効率に大きな影響を及ぼしている。本稿では、この問題を解決するために採用した入出力関数の削減方法を説明すると共に、これを実現したスレッドおよびシグナルを用いた DSE の実装について述べ、その性能評価を行う。

Experiments to Speed Up Processing in HPC Environment based on Distributed Shared Memory Model

Kiyoshi Ryoukai Tadanori Tezuka† Toshinori Sueyoshi

Department of Artificial Intelligence

Kyushu Institute of Technology

680-4 Kawazu, Iizuka

820 Japan

Abstract

This paper presents a Distributed Supercomputing Environment (DSE) based on distributed shared memory model which can be categorized as a high-performance computing environment. DSE provides facilities of parallel processing on a distributed system without altering the existing operating system. Since all communications in DSE have to go through the UNIX kernel, the communication processing takes most of the DSE processing time and presents a serious threat to the overall performance. In this paper, we describe solutions to this problem and further present a method to reduce the frequency of using I/O system calls in communication processing by utilizing threads and/or signal.

† 現在 松下電器産業株式会社

1 はじめに

近年、ネットワーク技術の発達に伴い、比較的容易にローカルエリアネットワーク (LAN) 環境を構築することが可能となった。特に、ワークステーション環境においては、複数のワークステーションを LAN に接続し、分散した計算機資源の共有管理を行う分散処理システムが企業や研究所等で多く構築されている。このような分散処理システムでは、高速な通信処理機能を利用することにより、複数のワークステーション上で1つの仕事を行う並列処理能力を潜在的に備えている。ワークステーション間の通信処理を扱う機能として、リモートプロシージャコール (RPC) が多くの分散処理システム上に実装されている。RPC を用いて並列処理記述を行うことは可能であるが、元来、RPC は分散処理アプリケーションを対象とした機能であり、並列処理記述には同期やプロセス制御などの点で多くの困難が伴う。また、V-system[1] や Chorus[2] など、並列処理機能を備えた分散オペレーティングシステムが多く研究されているが、そのほとんどがカーネルの変更を強いるものである。しかし、分散処理システムは、複数のユーザによって利用されているため、オペレーティングシステムなどのシステムソフトウェアの変更を簡単に行うことができない。このため、容易に並列処理を取り扱う処理環境として、既存のシステム上にオペレーティングシステムの変更なく並列処理機能環境を実装することが望まれる。

そこで我々は、UNIX オペレーティングシステムを変更することなく分散処理システム上に並列処理環境を実現するハイパフォーマンスコンピューティング環境の研究・開発を行っている [3][4]。将来のマイクロプロセッサの高速化および通信技術の発達に伴い、高並列のアプリケーションが分散処理システム上で実行されることを想定して、本システムを分散スーパーコンピューティング環境 (Distributed Supercomputing Environment, DSE) と呼ぶ。DSE は並列処理機能を実現するためにメッセージ交換の通信処理機構を用い、その並列処理機能として分散共有メモリ型並列計算機と等価な機能を提供する。DSE は、移植性を高めるために、そのすべての処理をユーザプログラムレベルで実装しており、DSE が行うすべての通信は

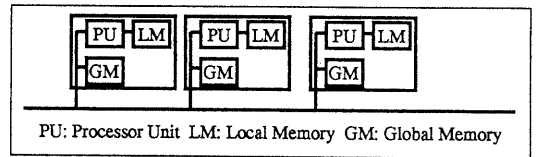


図 1: システムモデル

UNIX カーネルを介して行われる。そのため、システム全体の性能の点において、メッセージ交換における通信処理の重さが、これまでの研究により明かになってきた [5]。この通信処理は、通信のインタフェースに当たるシステムコールの実行とプロトコル処理に分けられる。本稿では、このインタフェースに当たるシステムコールの実行処理に着目し、改良における問題点および解決法を述べる。そして、実際に実装を行った2種類のDSEと従来のDSEの性能の比較を行い、その評価を行う。

2 DSE の概要

DSE は、オペレーティングシステムに手を加えずに並列処理を実行できる環境を分散処理システム上に構築したものである。また、メッセージ交換によって並列処理を実現し、ユーザに分散共有メモリ型並列計算機と等価な機能を提供する。DSE がモデルとする分散共有メモリ型並列計算機を図1に示す。DSE では、1つのプロセッサ要素を1台のワークステーションに対応させている。各々のプロセッサ要素は、すべてのプロセッサが参照可能な共有メモリであるグローバルメモリとローカルプロセッサのみが参照できるローカルメモリを持つ。メモリ管理においては、グローバルメモリをDSEが行い、ローカルメモリをUNIXのメモリ管理に任せている。

DSE の処理空間は2つある。1つは並列アプリケーションを実行するためのDSE processと呼ぶ空間である。もう1つはDSE プリミティブと呼ばれる並列処理機能を処理するDSE kernelのための空間である。これらの空間を、それぞれ別々のUNIX プロセスに割り当てることによって、それぞれの処理を独立に行っている。DSE は、UNIX カーネルに変更を加えることなく、つまり、UNIX

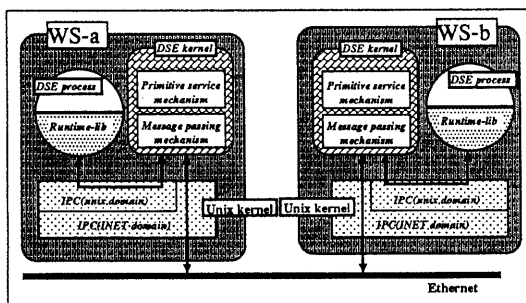


図 2: システム構成

のユーザプログラムレベルで実装されているため、容易に様々な UNIX 環境に移植することが可能である。また、DSE 上で実行される並列アプリケーションの記述は、用意されたランタイムライブラリを利用することにより、C 言語を用いて記述を行うことが可能である。

2.1 システム概要

DSE は、DSE kernel と DSE process の 2 つの UNIX プロセスで構成される。図 2 に DSE のシステム構成図を示す。DSE kernel は、DSE プリミティブを処理するための空間であり、内部を 5 つのモジュールで構成している。このモジュール群は、2 つの機能ブロックにグループ化できる。1 つは、他のワークステーション (DSE kernel) との通信処理を行うためのメッセージ交換機構であり、メッセージ交換モジュールによって構成される。もう一方は、DSE プリミティブを処理するための機構である。この機構に含まれるモジュールは、メッセージを解析し、それに対応した DSE プリミティブの処理を行うメッセージ解析モジュール、グローバルメモリを管理するためのメモリ管理モジュール、DSE process の生成・終了処理を行うプロセス管理モジュールおよび DSE kernel と DSE process 間のメッセージ交換を行うカーネルプロセスメッセージ交換モジュールである。DSE process は、並列処理を行うための空間、つまりアプリケーションを実行するための空間である。アプリケーションは、用意されたランタイムライブラリを用いて並列処理記述を行うことによって並列処理機能をアプリケーションに付加できる。ランタイムライブラリでは、DSE kernel に DSE

表 1: プリミティブ一覧

共有メモリに対する操作	mem_read* mem_write mem_writes*	共有メモリのread 共有メモリのwrite 共有メモリのwrite(同期)
制御に関する操作	pb vpb ext* vext* rfree shutdown	並列分岐 並列分岐 (軽量プロセス版) プロセスの切替え プロセスの切替え (軽量プロセス版) 資源解放 DSEの終了
同期に関する操作	lock* unlock sem_p* sem_v faa*	lock unlock セマフォのP操作 セマフォのV操作 fetch and add
メッセージ表示	message	stdoutの出力

*は、REPLYあり

プリミティブの実行要求を送信し、DSE kernel からその応答を受け取るだけである。実際の DSE プリミティブ処理は、DSE kernel 内で行われる。

2.2 DSE の機能

DSE の機能は、並列処理を行うための機能とユーザインタフェースであるシェル機能の二つからなる。

並列処理機能

並列処理を行うための基本機能である DSE プリミティブは、グローバルメモリの操作、DSE process の制御および同期の制御をサポートしており、15 種類の DSE プリミティブが用意されている。DSE プリミティブの一覧を表 1 に示す。

シェル機能

DSE は、並列処理のモニタ機能を持った簡易シェル機能を備えている。シェルの機能を以下に列挙する。

- 通常の UNIX コマンドの実行
- 並列アプリケーションの実行
- 並列アプリケーションのモニタ機能
- 並列アプリケーションの動作情報表示

シェルの機能は、DSE kernel 内部で処理される。

2.3 通信処理の影響

通信処理が DSE に与える影響を調べるために、DSE における機能部分の処理時間の測定を行った [5]。その結果、DSE が行う処理を通信処理と DSE プリミティブのための処理とに分けてその処理時間を比較した場合、その処理時間の比率は 9 : 1 であることがわかった。また、通信処理は read(), write(), select() 等の入出力システムコールの処理と UNIX カーネルが行うプロトコル処理とに分けることができる。この処理時間の比率は 2 : 1 であり、入出力システムコールの処理時間は、全体の 6 割にも及ぶ。このことより、DSE プリミティブのための処理に関する部分を改善しても、全体に与える影響は少なく改善の効果はあまり期待できない。一方、通信処理に関する改善は、その処理が全体の 9 割を占めることから考えて、大幅な処理の高速化が期待できる。しかし、プロトコル処理は処理の大半をカーネルに依存しているため、ユーザレベルでの処理の改善は難しい。一方、処理全体に対して処理の占める割合が大きい入出力システムコールの呼出回数の削減による処理の高速化については、その処理がカーネルに依存していないため、ユーザプログラムレベルで改善を行うことが可能である。

3 高速化技法

3.1 方針

処理全体に対して処理の占める割合の大きい入出力システムコールの呼出回数の削減による処理の高速化の方法について述べる。DSE は、これまでのシステムの改良により、入出力システムコールの呼出回数を極力減らしているため、現実装では内部シーケンスの改良による高速化の効果は望めない。また、現実装においては DSE kernel と DSE process の 2 つの処理の流れを独立に処理するため、2 つの異なった UNIX プロセスを用いて実現している。そのため、入出力システムコールは DSE kernel-DSE process および DSE kernel-

DSE kernel 間のメッセージ交換に使用されている。DSE kernel と DSE process のそれぞれが異なる UNIX プロセスで動作しているため、プロセス間のメッセージ交換のために入出力システムコールが必要である。しかし、DSE kernel と DSE process の 2 つの処理を 1 つの UNIX プロセス上に実装することによって DSE process-DSE kernel 間の入出力システムコールを削減することができる。この結果、DSE 全体として大幅な処理の高速化が期待できる。

1 つの UNIX プロセス上に DSE kernel および DSE process を実装する上で、大きく二つの問題点が上げられる。一つは、並列アプリケーションのロードの手段である。もう一方は、2 つの処理の流れを独立に扱わなければならない点である。

前者の問題は、DSE は様々な並列アプリケーションの実行環境を提供するものであり、DSE 起動後に様々な並列アプリケーションのロードが可能でなくてはならないために生ずる。現在の構成では、並列アプリケーションを DSE kernel とは別のプロセス (DSE process) にロードすることにより実現していたが、DSE kernel と DSE process を 1 つの UNIX プロセスに実装するため、この手法は使えない。今回の構成では、これを実現するための手法としてダイナミックリンク [6] を用いた。ダイナミックリンクとは、現在一般的に用いられているスタティックリンクと異なりプログラムの実行時にオブジェクトを動的にリンクする方法である。この手法によって動的に並列アプリケーションを DSE にロードすることが可能となる。

次に、後者の問題点は、DSE kernel と DSE process の 2 つの処理を 1 つの UNIX プロセスで取り扱う点である。ただ 2 つの処理を独立に扱うだけであれば、setjmp()/longjmp() 関数を用いた簡単なコルーチンで処理が可能である。しかし、この手法ではシグナル割込みを用いたコンテキスト切替えは難しい。実際に、DSE の処理では任意の時刻に到着したメッセージの処理をサポートしなければならない。このため、シグナル割り込みを用いた横取り可能なスケジューリングをサポートしなければならない。この問題点を解決するものとしてスレッドの概念を実現した、SUN Microsystems 社の Lightweight Process Library (LWP lib) [7] があ

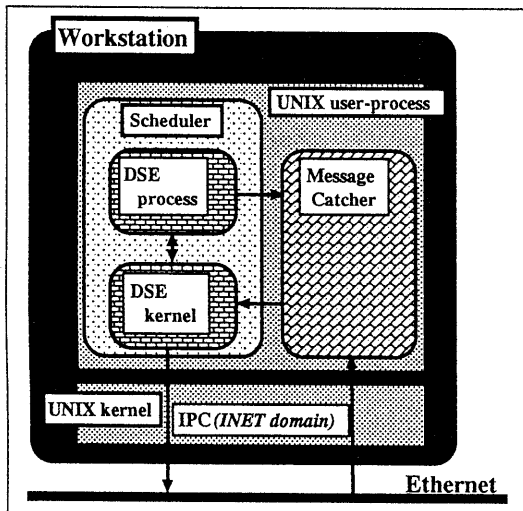


図 3: スレッドを用いた DSE の構成

る。また別の手法としてシグナルハンドラに DSE kernel の処理を実装する方法がある。今回、この二つの方法で DSE を実装した。

3.2 スレッドを用いた DSE

DSE kernel, DSE process を 1 つの UNIX プロセスに実装する場合、2 つの処理を独立にとり扱う必要がある。また、任意の時刻に到着する他のプロセッサの DSE kernel からのメッセージ、つまり入力に対する処理を必要とする。このような要求を満たすものに、ユーザレベルでスレッドを実現している LWP lib がある。これは、UNIX プロセス内部に複数の動作主体と呼ばれるスレッドを作成し、このスレッド上でプログラムを並行して実行させる機能を持つ。プログラム以外の資源はスレッド間で共有されるためコンテキストの切替えは UNIX プロセスの切替えに対して非常に軽いという特徴を持つ。この LWP lib を用いて実装を行った DSE の構成を図 3 に示す。システムは DSE kernel と DSE process の他にメッセージ処理を行う Message Catcher と、DSE kernel と DSE process のラウンドロビンスケジューリングを行う Scheduler が必要となる。Message Catcher の行う処理は、従来の DSE の DSE kernel におけるメッセージ交換機構で行っていた処理であ

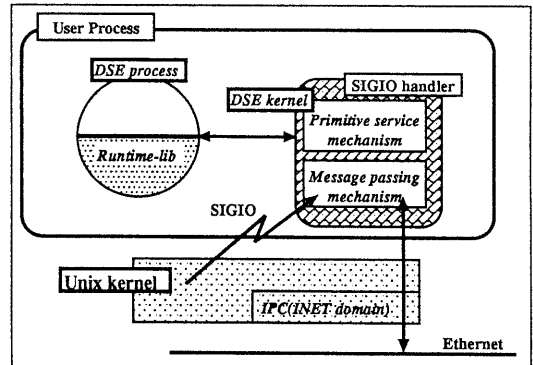


図 4: シグナルを用いた DSE の構成

る。従来の DSE において取り扱われるすべてのメッセージは、ソケットを用いて交換されていたため、select() システムコールで一元的に管理できた。しかし、スレッドを用いた DSE において、DSE kernel-DSE process 間で取り扱われるメッセージはソケットを用いて交換するのではなく、用意されたスレッド間通信を用いて交換を行う。そのため select() システムコールではメッセージの入力を管理することはできない。これを解決するために、Message Catcher で他のワークステーションの DSE kernel からのメッセージの到着を select() システムコールを用いて監視を行い、スレッド間通信を用いて DSE kernel にメッセージの到着を通知する。この結果、DSE kernel で DSE process と他のワークステーションの DSE kernel からのメッセージを一元的に管理することが可能となる。

3.3 シグナルを用いた DSE

DSE は、他のプロセッサからのメッセージの到着を知る必要があるが、メッセージの入力、すなわちソケットへの入力を知る方法は 3 つ考えられる。つまり、(1) read() システムコールによるもの。(2) select() システムコールによるもの。(3) 非同期入出力モードにおける SIGIO の割込みを利用したもの、である。(1), (2) については、ソケットへの入力があったことは分かるが、入出力処理を実行している間 UNIX プロセス自体がサスペンド状態になり DSE kernel, DSE process の

処理双方が止まってしまうため望ましくない。スレッドを用いた **DSE** においても `read()` や `write()` 等の入出力システムコールを実行することにより、**DSE kernel**, **DSE process** の処理が停止する恐れがある。これを防ぐために、SUN OS が提供しているライブラリである NonBlocking I/O ライブラリを用いてこの問題を回避した。(3) の手法は、ソケットへ入力が生じた場合、SIGIO の割込みが生じ入力があったことを知らせるものである。この割込み中に **DSE kernel** 処理を行うことにより、任意の時刻に到着したメッセージにも対応できる。また、**DSE kernel** の処理を割込み処理、**DSE process** の処理を通常の処理とすることで **DSE kernel** と **DSE process** の2つの処理を独立に取り扱うことを保証している。シグナルハンドラに **DSE kernel** を実装したシステムの構成図を図4に示す。シグナルハンドラに **DSE kernel** を実装したため、処理のスケジューリングは、横取り可能な優先度スケジューリングであり、**DSE kernel** の処理が優位になる。

4 性能評価

各々の実装についての性能評価を行った。ここで各々の実装に基づく **DSE** を明確に区別するために名称をつける。従来の **DSE** を UNIX process based **DSE**、高速化を行った **DSE** のうちスレッドによるものを Thread based **DSE**、またシグナルによるものを SIGNAL based **DSE** と呼ぶことにする。性能評価は二通りの方法で行った。一つは、プリミティブの実行時間を測定するもので、これにより、改良によるシステムの処理に関する処理向上が得られる。また、二つ目の評価として並列アプリケーションの実行速度についても測定を行った。この測定により、これらの改良がアプリケーションに与える影響を見ることができる。測定環境として SparcStation2 を4台使用した。

4.1 プリミティブ実行における性能評価

グローバルメモリにアクセスするプリミティブである READ プリミティブの処理時間について測定を行った。測定結果を表2および表3に示す。同一プロセッサ内のグローバルメモリにアクセスし

表 2: READ プリミティブ (ローカル) の実行時間

Read size (in bytes)	Local		
	UNIX	thread	signal
4	1.258	0.312	0.266
8	1.352	0.300	0.275
16	1.345	0.300	0.275
32	1.362	0.300	0.278
64	1.374	0.306	0.285
128	1.436	0.315	0.295
256	1.531	0.335	0.314
512	1.676	0.382	0.365
1024	2.008	0.460	0.426

[msec.]

表 3: READ プリミティブ (リモート) の実行時間

data size (in bytes)	Remote		
	UNIX	thread	signal
4	2.823	4.221	2.514
8	2.857	4.291	2.606
16	2.923	4.280	2.618
32	2.935	4.351	2.618
64	3.037	4.446	2.802
128	3.159	4.569	2.902
256	3.459	4.785	3.112
512	3.863	5.063	3.381
1024	4.854	5.770	4.085

[msec.]

た場合、プロセス間通信は行われない。そのため Thread based **DSE**, SIGNAL based **DSE** とともに大幅な改善が見られた(表2)。この結果より、通信処理が **DSE** の処理速度向上におけるボトルネックになっていたことがわかる。また、読み込むデータ量が増えても、READ プリミティブの処理時間の増加はあまり見られない。これは、UNIX process based **DSE** がソケットを介して通信しているためデータのコピー処理が多いのに対し Thread based **DSE**, SIGNAL based **DSE** とともに **DSE kernel-DSE process** 間の通信はメッセージのポインタの交換で行われているため、データのコピー処理がほとんどないためである。

一方、ネットワークを介して他のプロセッサ内のグローバルメモリにアクセスする場合、完全に通信処理をなくしたわけではないので、大幅な処理の改善につながらなかった(表3)。特に、Thread

表 4: 入出力関数の実行時間の比較

	標準 I/O	LWP lib's
read()	230	250
write()	220	500
select()[polling]	120	250
context switch	—	750

[usec.]

based DSE の場合においては、大幅な処理の高速化を招いた。Thread based DSE を実装する際に用いた LWP lib では、スレッド内で行われた I/O が他のスレッドの処理をブロックしないための Nonblocking I/O ライブラリが用意されており、Thread based DSE では、これを用いて DSE で行われるメッセージの入出力ルーチンを記述した。しかしながら Nonblocking I/O ライブラリの処理時間が標準の I/O ライブラリに比べ遅く、また I/O イベント待ちでのスレッドのコンテキストスイッチに非常に大きな時間を要する。表 4 に、標準の I/O 関数の処理時間と Nonblocking I/O 関数の処理時間を示す。select() の際のコンテキストスイッチには 750 マイクロ秒を要している。これは、UNIX process based DSE、SIGNAL based DSE のプリミティブ処理時間がほぼ 3 ミリ秒であることから非常に大きな時間を要していることが分かる。つまり、コンテキストスイッチやさらに遅くなった LWP lib の I/O 関数が、リモートのプリミティブ処理において非常に大きな処理の遅滞の原因であることが判明した。また、SIGNAL based DSE では、わずかながら処理速度に改善が見られた。また、メッセージサイズが大きくなるに従い、処理速度の向上が見られた。これは、read()/write() システムコールの処理時間が、データ量に比例するためである。つまり、メッセージサイズが大きくなるに従い、read()/write() システムコールの処理時間が大きくなる。このため、read()/write() システムコールの実行回数の少ない SIGNAL based DSE は、メッセージサイズが大きいほど処理の効率が上がる。

4.2 並列アプリケーションにおける性能評価

表 5: PDE の実行時間

Mesh size	UNIX (in sec.)	signal (in sec.)	差分 (in sec.)	速度 向上率
16 × 16	4.458	2.727	1.731	1.635
32 × 32	8.974	5.641	3.513	1.586
48 × 48	14.273	9.196	5.077	1.552
64 × 64	21.606	14.203	7.403	1.521
96 × 96	39.204	30.588	8.616	1.282
128 × 128	69.368	54.134	15.234	1.281
256 × 256	323.805	303.515	20.290	1.067

実際に処理速度の改善が見られた SIGNAL based DSE と UNIX process based DSE との比較評価を SOR 法による偏微分方程式の解を求める並列アプリケーションを用いて行った。このアプリケーションは、計算領域の大きさを変更することにより、並列処理の粒度を変更できる。このため、粒度の大きさにおけるシステムの性能の比較を行うことが可能である。測定結果を表 5 に示す。並列処理の粒度が小さくなるにつれ (Mesh size が小さくなるにつれ)、その性能比が向上している。このことは、より細粒度のアプリケーションの並列処理が可能になったことを示す。しかし、逆に計算領域の大きさが大きくなった場合 (Mesh size が大きくなった場合) には、性能比は 1 に近づいてくる。これは、計算領域の大きさが大きくなるに従い、アプリケーションの計算部分が並列処理のための処理に対し大きくなるため、DSE の改善の効果が薄らぐためである。

一方、アプリケーションの処理速度の面においても、改善の効果が見られた。計算領域のサイズが 16 × 16 の iteration の回数の変化における処理時間の変化のグラフを図 5 に示す。iteration の回数が増えるに従い、処理時間の差は大きくなることからわかる。これは、アプリケーションが大きくなり、並列処理記述が増加するに従い、処理時間が大幅に短縮されることを示している。

5 まとめ

我々は、DSE の処理の高速化をさまざまな面 (プロセス交換、内部シーケンス、通信処理など) から行ってきた。特に本稿では、処理の 9 割を占め

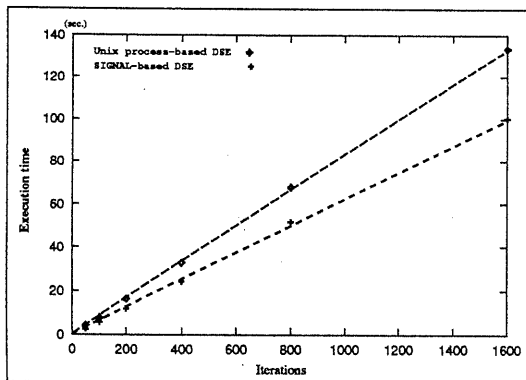


図 5: PDE の実行時間 [mesh size = 16×16]

る通信処理について着目し、その高速化の手法とその効果を示した。DSE kernel と DSE process を 1 つの UNIX プロセス内で処理することにより、ローカルのプリミティブ処理においては大幅な処理の高速化が得られた。一方、リモートの処理においては、スレッドを用いて実装した場合には効果はなく、逆に大幅な処理の遅滞を招いた。またシグナルハンドラに DSE kernel を実装した場合においては、わずかながら処理速度の改善の影響が見られた。さらに、アプリケーション側からこの改良を行った DSE を見た場合、従来の DSE に比べ細粒度の並列処理記述が実用レベルで可能となり、アプリケーション全体の処理速度の向上にも結び付いている。

今回は、`read()/write()` 等の入出力システムコールの使用頻度を減らすため、DSE kernel-DSE process 間の `read()/write()` システムコールの削減方法に着目し改善を行ったが、一方ワークステーション間の通信である、DSE kernel-DSE kernel 間で用いられる `read()/write()` システムコールを削減する方法も考えられる。DSE kernel-DSE process 間の通信に用いたスレッド間通信のように、この通信処理を他の方法に置き換えることはできないが、グローバルメモリへのアクセス処理にキャッシュシステムを実装することによって、`read()/write()` システムコールの使用頻度を下げることが可能となる。また、マルチキャストなどの通信効率の良い機能をサポートした通信プロトコルを実装することによってプロトコル処理

を軽減することが考えられる。今後は、キャッシュシステムの有効性の調査および効率の良い通信プロトコルの実装を行う予定である。

謝辞

本研究を遂行するにあたり、御支援頂く本学有田五次郎教授に感謝します。また、日頃御検討頂く本学マイクロ化総合技術センター久我守弘講師ならびに有田・末吉研究室の諸氏に感謝します。

参考文献

- [1] D.R. Cheriton, "The V Distributed System", *Communication of the ACM*, volume 31, number 3, pp. 314-333, March 1988.
- [2] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guilemont, F. Hermann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser, "Chorus Distributed Operating System", *Computing Systems*, volume 1, number 4, Fall 1988.
- [3] B.O. Apduhan, T. Sueyoshi, Y. Namiuchi, T. Tezuka, T. Fujiki and I. Arita, "Experiments and Analysis Toward Distributed Supercomputing on a Distributed Workstation Environment", in *Proc. of 1991 International Symposium on Supercomputing*, pp. 182-190, Japan, Nov. 1991; or *Supercomputer, Special Issue for ISS '91 (Revised Version)*, Strichting Academisch Rekencentrum Amsterdam (SARA), volume VIII, number 6, pp. 90-100, November 1991.
- [4] T. Tezuka, K. Ryoukai, B.O. Apduhan, and T. Sueyoshi, "Implementation and Evaluation of Distributed Supercomputing Environment on a Cluster of Workstations", in *Proc. 1992 International Conference on Parallel and Distributed Systems*, pp. 58-65, Taiwan, R.O.C., December 1992.
- [5] 手塚 忠則, 了戒 清, 末吉 敏則: 分散システムを利用した並列処理環境における通信処理の影響, 情報処理「マルチメディア通信と分散処理」ワークショップ論文集, pp. 249-256, March 1993
- [6] *SUN Programming Utilities and Libraries*, Chapter 1: Shared Libraries, 1990.
- [7] *SUN Programming Utilities and Libraries*, Chapter 2: Lightweight Processes, 1990.