汎用超並列オペレーティングシステムカーネル *SSS–CORE*

The general-purpose scalable operating system: *SSS–CORE*

松本 尚 *1
tm@is.s.u-tokyo.ac.jp
渦原 茂 *2
uzu@axe-inc.co.jp
竹岡 尚三 *3
take@axe-inc.co.jp
平木 敬 *4
hiraki@is.s.u-tokyo.ac.jp

*1,*4 東京大学 大学院理学系研究科 情報科学専攻
*2,*3 株式会社 アックス

SSS–CORE はワークステーションクラスタの上で効率の良い並列処理を可能とする汎用スケーラブルオペレーティングシステムである。本オペレーティングシステムはマルチタスクという汎用計算機に不可欠な機能を実現しつつ、この汎用性と従来相容れなかった効率の良い並列計算を実現するための各種機能を搭載する。本稿では、まず SSS–CORE の設計思想について述べ、最大の特徴であるメモリベース高速通信同期機構 MBCF について述べる。次に、分散メモリ計算機上で効率良く分散共有メモリを実現する方式について述べ、その方式をサポートする最適化コンパイラ RCOP について簡単に解説する。その後、市場経済モデルのスケジューラや外部 OS エミュレータと言った SSS–CORE の特徴的な機構を述べ、実装された SSS–CORE Ver.1.1 のスペックを記述する。 SSS–CORE の基本機能および実用規模ベンチマークによる性能評価結果を示し、簡単な開発経緯とまとめで本稿を締めくくる。性能評価の結果、通信同期は従来型 OS よりも一桁以上高速であり、並列アプリケーションベンチマークにおいても最高 83% の性能向上を示した。

1 はじめに

並列計算機も実用化時代に入り、多くの商用マシンが開発され実務に供されている。しかし、多くのマシンは複数の独立したジョブを高速に処理するサーバ機として使用されており、汎用の並列計算環境つまりマルチタスク環境における高効率の並列処理はまだ実用レベルではない。一方、 LAN 用ネットワークの高速化に伴って、高速ネットワークで複数台のマシンを結合したワークステーションクラスタ（WS クラスタ または Network of Workstations: NOW）、サーバ機クラスタ、 PC クラスタが注目を集めるようになってきている。現状ではまだ汎用 LAN の性能不足や OS のオーバヘッドのために性能が十分に発揮できていない面があるが、データベース共有やファイル共有の分散処理程度であれば、並列計算機に取って代わりそうな勢いである。

汎用並列計算機や WS クラスタにおける次なるチャレンジとしては、現在の分散処理環境と同等の汎用環境を維持しつつ、スケーラビリティと並列処理による高速性を安価に実現することである。汎用性と並列処理性能の高さから集中共有メモリやハードウェア分散共有メモリを持つ商用並列計算機の発売が相次いでいる。しかし、これらの専用マシンは量産効果がなく非常に高価である。やはり、 WS クラスタのような量産効果を活かし安価に汎用高性能並列計算機を開発する方向を模索する必要がある。

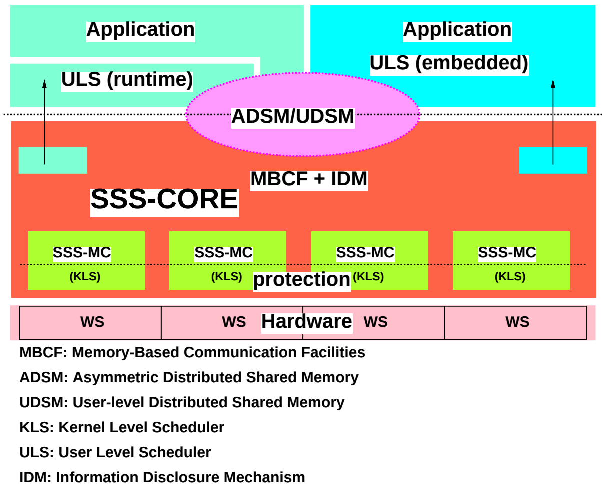

図 1 **SSS–CORE** の機能構成

このような背景の下で、我々は 1994 年より汎用スケーラブルオペレーティングシステム **SSS–CORE** [1]の開発を開始した。 SSS–CORE は並列アプリケーションの自律的最適化を補助する各種機構を持ち、並列処理の効率を極力落とすことなくマルチタスクの汎用環境を実現する汎用オペレーティングシステム（汎用 OS）であり、特殊なハードウェアを仮定しない WS クラスタを対象としている。

† 本研究は情報処理振興事業協会「独創的先進的情報技術に係わる研究開発」の一環として行われたものである。

2 SSS–CORE の設計思想

2.1 ユーザレベル最適化の支援

SSS–CORE は分散システムとしての汎用性を損なわずに、並列アプリケーションプログラムの効率の良い実行環境を構築することが目的である。この実行効率を向上させるための手段としてユーザレベルのコード最適化を最重要視している。こう書くと、プログラム開発が非常に困難になって、開発コストが大きくなることを心配する向きもあるかもしれない。アプリケーションプログラマが最適化されたコードを書かなければならないとすれば、確かにその通りである。もちろん、最適化されたコードを書きたいプログラマには大いに手腕を発揮してもらえば良いが、我々はプログラマが人手によって最適化を行うことを念頭においているわけではない。最適化コンパイラまたは最適化ツールといったシステムソフトウェアによって、自動的にシステムに合わせた最適化を行うことを目指している。プロセッサ内部の命令レベルの並列性（RISC の遅延スロット、スーパスカラ、VLIW）の利用に関しては、すでに最適化コンパイラがかなり効率の良いコードを生成している。プロセッサを越えた並列性に関しても、計算と通信のオーバラップ、通信網の飽和状態の回避、遠隔データのキャッシングといった最適化が施された並列実行コードを、将来的に最適化コンパイラが自動生成可能になると考えている。そして実際に、我々はその第一歩となる最適化コンパイラとして RCOP を SSS–CORE 用に開発した。 RCOP に関しては後述する。 SSS–CORE はそのユーザレベル最適化（静的なものも動的なものも両方含む）に必要な情報や手段（システムコール、資源割り当て）を可能な限り低オーバヘッドで提供する。

2.2 機能構成

図 1に SSS–CORE の機能構成図を示す。各ノードに SSS–CORE の核となり、ノードの資源管理と資源保護を担当する SSS–MC（Micro Core）が常駐している。 SSS–MC は単体で単一ノードの資源保護と資源管理を行うマルチタスクのオペレーティングシステムである。その SSS–MC の機能を使って、複数ノードに跨る通信同期機能である MBCF と資源管理情報開示機構（IDM）が提供されている。コンパイラ支援型分散共有メモリ機構（ADSM/UDSM）[3] [4]は、最適化コンパイラがユーザレベルのキャッシュエミュレーションコードと MBCF およびメモリ管理機構のユーザインタフェースを利用するコードを生成することにより実現され、図 1ではユーザ／カーネルの両境界に跨る機構として図示されている。ユーザアプリケーションは MBCF もしくは ADSM/UDSM を用いることにより WS クラスタ全体に跨るグローバルな共有メモリ空間を低コストで使用することができる。

2.3 SSS–MC (Micro Core)

SSS–MC は、設計方針として「コンパクトなカーネル」を基本としているが、従来より提案されているマイクロカーネル [5] [6] [7]とは設計方針が異なる。つまり、性能を犠牲にしてまでカーネルのコンパクトさやカーネルインタフェースの統一を追求するようなことはしない。例えば、従来のマイクロカーネルでは、ポートやメッセージといったなんらかの抽象化を含んでおり、それらはソフトウェア的に高機能なサービスを提供しているが、その抽象化のためにオーバヘッドが増加する。そして、マイクロカーネルとのインタフェースが非常にプリミティブな操作に限定されているため、カーネル呼び出しの回数が非常に多い。これらのことが性能の問題になっている。それに対し、 SSS–CORE および SSS–MC ではハードウェアが提供する機能を、必要最低限の保護と仮想化を守った上で、できるだけ直接的にサービスとして見せることを基本としている。また、 SSS–CORE のサービスとして重要であり、 SSS–MC と密に協調して実現されるべき高レベル機能は、別タスクとしてカーネル空間内に実装されている。

2.4 スケーラビリティ

また、マシンを増やすことでシステムの処理能力を容易に拡張できるスケーラビリティを持つことも SSS–CORE の目標の一つである。マシンの増減をオペレーティングシステムのコンフィギュレーション設定ファイルの更新程度でシステムに反映可能であり、マシン数の追加によってシステム全体の能力の増強がかなりの台数まで行えることを SSS–CORE では「スケーラビリティ」と呼んでいる。ただし、並列アプリケーションがマシン数増強の恩恵を受けるためにプログラムの書き直しが必要になるようでは「スケーラビリティ」の要件を満たしていない。マシン数増強に伴って何の変更もなしに並列アプリケーションの能力が向上することが望ましいが、最悪でもマシン台数のパラメータを変更して再コンパイルするか実行時に並列実行台数を指定する程度で最適化された並列実行が実現できなければならない。

2.5 メモリベースの OS アーキテクチャ

資源を最低限の保護と仮想化によってユーザに提供しつつも、ユーザに雑駁で多様な知識を要求することにならないように、 SSS–CORE はシステムの統一的な視点を提供する。統一的な視点として SSS–CORE は共有メモリビューを全面的に採用し、ノードを跨る資源やカーネル内の情報を統一的に見せている。プロセッサのメモリ管理機構の恩恵により、メモリマップされた資源はほとんどオーバヘッドなしに保護と仮想化が実現できる。このプロセッサのメモリ管理機構をノード間通信同期やユーザカーネル間通信同期に活用して、付加的な保護や仮想化のオーバヘッドをほとんど導入せずに SSS–CORE はグローバルな共有メモリ環境を実現する。 SSS–CORE のメモリシステムの活用例として以下の項目が挙げられる。

- 中粒度メモリ操作としてシステム全領域の通信同期が定義されている（MBCF）
- カーネルの資源管理情報でユーザに有益な情報はノード内外を問わず、ユーザタスクからローカルに読み出し可能（IDM）
- 最適化コンパイラによって実現されるユーザレベル分散共有メモリ機構を支援（ADSM/UDSM）

- 自由にノード内に共有メモリを生成できる

3 SSS–CORE の通信同期方式

3.1 背　景

並列処理と分散処理の差はプロセッサ間の通信同期の頻度に過ぎず本質的な差は存在しない。しかし、通信同期の頻度の多い並列処理では通信および同期のオーバヘッドを可能な限り抑えなくてはならない。1994 年当時、UNIX WS の通信用システムコールは 1msec のオーダのコストがかかっており、専用並列計算機よりも二桁程度オーバヘッドが大きかった。この通信オーバヘッドの主な要因はシステムコールおよび TCP/IP 等の通信プロトコルにクラスタ内通信にとって無駄な処理が多く含まれていることに起因していた。この大きなオーバヘッドのため、既存の UNIX を使って WS クラスタを構築しても効率の良い並列処理の実現は到底おぼつかない。

一方同期方式に関しては、専用並列計算機ではメモリ上のフラグもしくは通信ハードウェア内レジスタ上のフラグをポーリングする（フラグが立つまでループを回る）ことで同期するのが一般的である。しかし、TSS (Time Sharing System) を行っている汎用オペレーティングシステムの下では複数のタスク（プロセス）が時分割で実行されており、通信相手が必ずしもプロセッサにスケジューリングされているとは限らない。このため、ポーリングによる同期ではプロセッサに割り当てられていないタスクの処理が進むことを期待して無駄に同期待ちループを回る可能性がある。そして、この可能性は多くのタスクが同一のマシン上で動いているほど高くなる。UNIX 等の分散処理では、無駄な同期待ちループをプロセッサが実行しないように、オペレーティングシステムに再起動の条件（イベント）を設定してプロセスを一時停止させ、通信イベントの発生を待つようにプログラムされている（シグナル機構と呼ばれている）。しかし、従来のシグナルのオーバヘッドは非常に大きく、並列処理のように同期頻度が多い状況では使用することが難しい。

分散メモリ型並列計算機の多くのものはポーリング時の無駄な同期待ちを少なくするために TSS を行っておらず、並列処理するタスクを一度割り当てたら終了するまで次のタスクを割り当てない（バッチ処理システム）。バッチ処理では大きな並列ジョブがシステムに投入されると、それが終了するまで他のジョブは処理されないため、WS クラスタを通常の作業環境として利用することができない。

3.2 SS-Wait 同期方式と情報開示機構

SSS–CORE では、TSS を維持しつつ、効率の良い同期を実現するために、松本が 1989 年に考案した Snoopy Spin Wait（SS-Wait）[1] [8]と呼ばれる同期方式を採用している。これはポーリング方式の変形で、フラグをチェックすることで同期の確認を行う点ではポーリング方式（スピンウェイト）とまったく同一であり、同期が成立していれば、そのまま処理を続行する。同期が成立していない場合に、フラグのチェックを繰り返すのが単純なスピンウェイト方式であるが、SS-Wait では資源割り当て情報や同期相手の進捗情報を利用してフラグチェックを繰り返した方がよいかどうかの判断が追加される。もし、同期相手がプロセッサの割り当てを受けていなかったり、大幅に進捗状況が遅れている場合には、フラグチェックの繰り返しを放棄して、タスク内の別の処理を行うかプロセッサの実行権を他のタスクに委譲する。

SS-Wait の実現でキーとなるのは自分のノードのみではなく他ノードの資源割り当て情報や資源使用状況といった情報が低コストで参照可能な機構である。このために、SSS–CORE は情報開示機構を持っている。カーネルが管理している資源割り当て情報や資源使用状況といった情報を格納しているメモリ領域がユーザ空間に read-only でマップされており、低コストで参照できる。しかも、情報開示機構には自分のノードの情報のみではなく、他ノードの情報も適宜交換され開示されている。ユーザアプリケーションはこの情報を参照して、SS-Wait を行い、また今後の負荷分散戦略や実行タイミングの検討を行う。なお、同期相手の進捗状況に関しては適当な間隔で、並列処理を行うタスク間でお互いに情報交換すればよいため、特に情報開示機構の開示対象とはなっていない。

3.3 MBCF 通信機構

SS-Wait によって TSS 環境でもフラグベースの同期が効率良く実現できる目処が立ったため、通信相手のローカルメモリ上のデータ領域やフラグが操作可能な通信方式が必要となる。つまり、SS-Wait でオペレーティングシステムの関与なしに同期が実現できても、同期の元となる情報の通信にオペレーティングシステムを介した重い処理が必要となっては元も子もなくなるのである。このため、SSS–CORE では、新しい通信機能として、保護と仮想化の機能を保存したまま他ノードのメモリを直接操作するメモリベース通信を採用することにした。メモリベース通信は筆者が 1992 年に提案した Memory-Based Processor (MBP) [9]が実現する機能として提案した通信同期方式であり、メモリ管理機構を遠隔メモリアクセスにも拡張して通信や同期をすべて論理アドレスに基づくメモリ操作によって実現するという方式である。

MBP はプロセッサの単一の load/store やプロセッサキャッシュの 1 ブロックの転送を遠隔メモリアクセスに変換するハードウェア機構であったが、SSS–CORE においてこれをそのままソフトウェアエミュレーションするには一回当たりのデータサイズが小さ過ぎて（通信の粒度が細かすぎて）オーバヘッドの割合が高くなってしまう。そこで SSS–CORE ではより粒度の大きな（通信データサイズの大きな）メモリベース通信を定義して使用することにした。このメモリベース通信方式を Memory-Based Communication Facilities (MBCF) [10] [11]と呼ぶ。具体的には、データサイズを可変長にして、MBP のようなプロセッサのレジスタを介した遠隔

1 SSS–CORE の「SSS」は Scalable Snoopy Spin の略であり、この同期方式にちなんで命名されている。

メモリアクセスではなく、ローカルメモリとローカルメモリの間のメモリ操作と定義した。データサイズは可変長と言っても、むやみに長いデータの転送は他のジョブの通信や処理に干渉する可能性があるため、適度の長さの上限 (Ethernet パケットの最大長) を導入している。

3.4 MBCF の動作原理

MBCF の基本動作である MBCF_WRITE (遠隔書き込み) を例にして MBCF の動作を以下に解説する。

現状の SSS–CORE では、 Ethernet 系の汎用 NIC 上の新しいプロトコルとして MBCF を実装した。 SSS–CORE では、汎用 NIC を使って保護された通信を実現する制約から、パケットの送信操作をシステムコールとして実装し、ユーザの送信データを DMA 領域にコピーする。しかし、このコピーされたデータ領域は実は Ethernet 系の NIC にとってはもう一つの重要な役目を兼ねることができるため、無駄なコピーを行っていると一概に言うことはできない。 Ethernet 系の通信ではパケットが転送途中で消失する危険性があるため、到着保証を行うためには、パケット送信元にパケットを再送する能力が必要である。この到着保証を行うための再送用コピーとして NIC の DMA 領域の送信用パケットのコピーが流用できる。ユーザレベルですべてを行うゼロコピー方式を使ったとしても、到着保証が必要な場合はアプリケーションプログラム内で結局コピーを作るか、パケット消失時にはいつでも同じパケットを再生成可能なように複雑かつオーバヘッドの大きいプロトコルを採用する必要が生じる。

結局、 SSS–CORE の MBCF の送信ではシステムコールによって、送信先タスク、送信先の対象アドレス、メモリベース通信の種類 (コマンド) といったパラメータと共に、送信すべきデータ領域のポインタ (送信データを伴う場合のみ) をシステムコールとして受け取って、 DMA 領域内に最終的なパケットイメージを構成し、 NIC に外部に転送させている。送信先タスクの指定は仮想化されており、送信元タスク毎に設定される論理的な識別子で指定される。送信先タスクは送信元タスクと同一ノードに存在しても構わない。つまり、送信先タスクの指定と送信先タスク内の操作対象論理アドレスの組が、システム全体にわたる共有論理アドレス空間を構成する。送信時の動作に関しては、操作対象の論理アドレスをパケットごとに指定している点を除けば、既存 OS の通信プロトコル (TCP/IP や UDP/IP) と同等の処理を行っている。しかし、既存 OS の通信プロトコルの実装では無駄なコピー (プログラムを読みやすくするという大義名分はあるらしい) や通信と無関係な処理等を行うために非常にオーバヘッドが大きい。 MBCF の送信時オーバヘッドは既存プロトコルの 100 分の 1 程度である。

受信側では NIC が自分が宛先になっているパケットを DMA 領域内のパケットバッファに受け取り、プロセッサに割り込みによってパケットの到着を通知する。受信用パケットバッファは有限であるため、すみやかにデータを待避する必要があり、従来の通信プロトコルではカーネル内の他の領域にコピーしていた。 MBCF ではパケット内に操作対象タスクと操作対象アドレスが含まれているため、割り込みルーチン内で対象タスク内のメモリを直接操作して処理を完了させる。操作内容が遠隔書き込みであれば、パケット内のデータを対象タスクの対象アドレスに直接書き込む (コピーする) 。送信元タスクはオプションとして操作が終了したことを確認する同期操作のための返信を要求することが可能であり、このオプションが指定されているとメモリ操作終了後に返信パケットが発送される。返信パケットは MBCF 操作要求パケットと同様に送信元タスクが予め指定したアドレスのフラグまたはカウンタを返信パケット受信時に直接操作する。

3.5 メモリ管理機構による MBCF の高速化

パケット到着時の割り込みルーチン内でユーザタスクのアドレス空間を直接操作している点が MBCF の大きな特徴であり、汎用 NIC の使用にもかかわらずオーバヘッドを大幅に低減させている理由の一つである。ユーザのアドレス空間とパケット到着による割り込みルーチンのアドレス空間は一般には異なっている。このため、 MBCF では受信割り込みルーチンのメモリ操作時のみ一時的にアドレス空間を切り替えて、ユーザのアドレス空間を操作しているのである。しかも、最近の高性能プロセッサにとってはこのアドレス空間の切り替えはほとんどコストがかからない。最近の高性能プロセッサは TLB 内に論理アドレスだけではなくアドレス空間の識別子であるコンテクスト ID も同時に保持しており、コンテクスト ID と論理アドレスの上位が共に一致したときのみ TLB 内のエントリを使ってアドレス変換を行う。この機能拡張によって TLB は複数のアドレス空間に対するアドレス変換のための情報を同時に保持できるようになった。このため、 MBCF のパケット受信時に空間を切り替える場合、単にプロセッサ内部のコンテクスト ID を一時的に切り替えるだけであり、アクセスする領域の変換情報が TLB 内にあれば即時にメモリをアクセスできる。もし TLB 内に情報がなかったとしても、 TLB におけるアドレス情報不在 (TLB ミス) に対する処理は最新プロセッサでは高速であり、 MBCF のメモリ操作に必要な TLB のエントリのごく一部を入れ替えるだけで直接ユーザ空間がアクセスできる。受信割り込み終了後も、 TLB のエントリがほとんど保存されているため、割り込み前のタスクが処理速度の低下なしに実行を再開できる。

3.6 MBCF の高機能および高セキュリティ方式

遠隔書き込みや遠隔読み出しといった基本的な遠隔メモリ操作の他に、 MBCF は、マルチキャスト機能、 SWAP や Compare&Swap といった不可分操作、受信側タスクが任意のアドレスに FIFO キューを設定可能なメモリベース FIFO、受信側が設定したプログラムを受信側タスクの権限で起動できるメモリベースシグナルといった高機能の遠隔メモリ操作機能を有している。また、サーバクライアントのような保護やセキュリティが重要となるアプリケーションに対して、メモリ管理機構のエイリアシング (同一ノードの複数のメモリ空間が一

部のメモリを共有する）機能を利用した高セキュリティのアクセス方法が提供されている。高機能メモリベース通信および高セキュリティのアクセス方法に興味のある方は、MBCF に関する文献 [12]を参照されたい。

4 分散共有メモリ実現の新技法

4.1 基本方針

MBCF は歴史的には MBP の機能をソフトウェアエミュレーションするために開発された。そして、MBP はユーザレベルの保護され仮想化された高速通信同期を提供する機能の他に、ハードウェア分散共有メモリを提供する機能を持っていた。MBCF がいかに効率良く実装されていても、MBP のようにプロセッサの load/store の単位でキャッシュの管理とそれに伴う遠隔通信同期を行っていたのではオーバヘッドが大き過ぎる。そこで、何らかの手段でキャッシュ更新および通信を行う粒度（データサイズや通信間隔）を大きくする必要がある。なお、本節でキャッシュと呼ぶのは、プロセッサに付随するハードウェアキャッシュのことではなく、OS ベースのソフトウェア分散共有メモリと同様にノード内に遠隔メモリのキャッシュ用に確保された主記憶の領域のことである。分散共有メモリ機能をもっとも必要とするのは、共有メモリモデルに従って記述された並列アプリケーションである。これまでの分散共有メモリ機構は、ハードウェアによる実装であれ、OS がメモリ管理機構を流用してサポートするソフトウェアの実装であれ、実行コードのリモートメモリアクセスはプロセッサの単純な load/store に変換されていることに固執していた。しかしながら、メッセージパッシング型の通信ライブラリである MPI を使った並列アプリケーションでさえ、異なるマシン間ではソースコードで流通しており、マシンが異なれば再コンパイルすることが当たり前である。ということは、最適化コンパイラがリモートメモリアクセスをキャッシュエミュレーションコードとそれに伴う明示的な通信コードとしてコード生成を行っても構わないことになる。そして、キャッシュエミュレーションコードと通信コードに対して、アプリケーションコードと共に可能な限り最適化を施して、キャッシュメンテナンスや通信の粒度を大きくなるようにコード生成を行えば良い。

4.2 UDSM と ADSM

最適化コンパイラに支援された分散共有メモリ実現法として、松本は User-level Distributed Shared Memory (UDSM) [14] [4]と Asymmetric Distributed Shared Memory (ADSM) [15] [4]を考案した。外部発表された経緯とは逆であるが、基本方針に忠実な UDSM を先に説明する。

UDSM は最適化コンパイラがソースコードを解析して、共有メモリアクセスが必要な共有書き込みと共有読み出しに対して、キャッシュコンシステンシ管理コードを挿入する。共有読み出しに関しては、キャッシュ存在確認コードと言った方が正確かもしれない。また、緩和された同期モデルに基づくプログラムでは同期ポイントにもキャッシュコンシステンシ管理コードを挿入する。キャッシュコンシステンシ管理コードは必要に応じてランタイムの通信ライブラリを呼び出して、明示的にメモリベース通信を行う。メッセージパッシング型の通信機構を用いても通信は可能であるが、受信側のユーザレベルプログラムによって送信側が示した対象アドレスを操作させる必要がでるため、メモリベース通信に比べてオーバヘッドが大きくなる。基本方針にも書いたように、細粒度のメモリアクセスの度にキャッシュコンシステンシ管理コードを挿入したのでは、オーバヘッドが大きくて実行効率は非常に悪くなってしまう。そこで、キャッシュコンシステンシ管理コードを実行する回数と通信回数が少なくなるように最適化を施す。UDSM はユーザレベルで共有書き込みと共有読み出し双方を実現しているため、OS からのサポートは不要に見える。しかし、メモリリプレースの高速化まで考慮に入れると、OS サポートが必要である。リプレースする対象として、dirty でないキャッシュコピーの領域は非常に適しており、メモリの内容を退避することなくリプレース対象にできる。また、新たに割り当てる必要が出た場合でも、システム内にデータが存在するため二次記憶装置をアクセスする必要はない。さらに、リプレース対象に優先的に選ばれリプレースコストも低いため、キャッシュコピー領域として主記憶を大量に消費しても、他のアプリケーションに迷惑をかけない。

ADSM はユーザレベルのキャッシュエミュレーションと OS レベルのソフトウェア分散共有メモリの双方の特徴を受け継いだ方式である。共有読み出しはメモリ管理機構を流用した方式によって単純な load としてコードが生成される。キャッシュミスはページフォールトとして検知され、ページフォルトハンドラからユーザレベルのキャッシュミスハンドラを呼び出すことで処理される。ページ機構を流用するためにキャッシュブロックサイズはページサイズと一致させられる。キャッシュブロックサイズに自由度がない代わりに、共有読み出しに関してはキャッシュコンシステンシ管理コードの挿入が不要である。通常のプログラムでは共有読み出しの方が共有書き込みよりも圧倒的に多いことが普通であるため、UDSM において共有読み出し用の挿入コードを削減する最適化がうまく働かないときには非常に有効である。そして、共有書き込みに関しては、キャッシュコンシステンシ管理コードを明示的に実行コード内に挿入する。もちろん、キャッシュコンシステンシ管理コードを実行する回数と通信回数が少なくなるように最適化を施す。共有書き込みと共有読み出しで取扱が異なることから、非対称分散共有メモリ（ADSM）と命名されている。

4.3 RCOP の概要

我々が開発した最適化コンパイラ RCOP (Remote Communication OPtimizer) [13] [4]は LRC (Lazy Release Consistency) モデル [16]に基づいた共有メモリ並列プログラムを取り扱う。RCOP の入力プログラムは共有メモリ並列プログラム記述用のマクロライブラリ PARCMACS [17]で拡張された C 言語で書かれている。RCOP は共有メモリ並列プログラムを解析し、ADSM

用のキャッシュコンシステンシ管理コードを含んだ C 言語プログラムに変換する。出力された C 言語プログラムは gcc2.7.2（最適化レベル O2）でコンパイルされ、ADSM ランタイムライブラリとリンクされて、最終的な実行コードが生成される。現在、RCOP を UDSM 対応のコード生成も可能なように拡張中である。以下に、現在の RCOP の特徴を列挙する。

- ソースコードを自動変換し、プログラマやユーザによる書き換えは不要
- オーバヘッド削減のため、すべてのタスクにおいて同一オブジェクトに対するアドレスは共通している（メモリの論理アドレスと共有オブジェクトの識別子を一体化させている）
- 最適化のため手続き間解析を行う
- ポインタ解析を行って共有書き込み候補を限定する
- 無効化型プロトコルとして、AURC [18]を RCOP による通信コード挿入によってソフトウェアエミュレーションする SAURC [15]プロトコルを採用している。LRC 系の他のプロトコルも使用可能であるが、実験により SAURC が無効化型プロトコルの中では優れていることが確かめられている
- 最適化技法
  - ループ誘導変数と配列引数に関する解析を行い、同一同期区間内の連続領域へのアクセスはなるべく一つにまとめる（アクセス粒度を大きくする）。この最適化をコアレシングと呼ぶ
  - 細かい粒度の同一宛先の通信は動的に通信内容をマージして、通信粒度を大きくする。この最適化をパケットコンバイニングと呼ぶ
  - 同一共有アドレスの内容を同期区間内で複数回更新する場合は最後の書き込みのみをキャッシュ操作の対象とする。この最適化を冗長共有書き込みの除去と呼ぶ
  - 無効化型プロトコルを用いるとキャッシュミスによるレイテンシが大きくなる状況では更新型プロトコルを使用する（要は更新用のコードを挿入）。RCOP は更新型プロトコルの自動挿入にはまだ対応していない。しかし、プログラム中のアノテーションによって更新型プロトコルを使用する共有メモリ領域や変数を宣言することはできる。この最適化をプロトコル切替と呼ぶ

RCOP のプログラム解析手法や最適化技法に関する詳細は文献 [4] [19]を参照されたい。RCOP が生成したコードが実行するキャッシュプロトコルに関する詳細は文献 [20]を参照されたい。

5 スケジューリング

SSS–CORE の目指す実行環境では WS クラスタの各ノードは従来通りの単体 WS としても使用される。SSS–CORE 開発当初はグローバルなカーネルレベルのスケジューリングの作成を考えていた。しかし、各ノードの負荷がユーザの単体 WS としての使用によって大幅に変化するため、カーネルレベルスケジューラが良かれと思って行った負荷調整がかえって個別のアプリケーションの実行時間やシステムの全体性能に悪影響を与えるかもしれない。さらに、従来はプロセッサの割り当てのみに注目してスケジューリングが行われているが、WS クラスタではメモリ配置や通信ネットワークの競合といった要因も考慮する必要がある。プロセッサの使用状況のみしか考えずに、タイムスライスごとにタスクの割り当てノードを移動させていたのではデータやプログラムのノード間の移動のオーバヘッドで性能が大幅に低下してしまう。また、十分にバンド幅の高いネットワークを持っていない場合には、ネットワークが飽和しないようにネットワーク使用頻度の高い複数のタスクを同一タイムスライス内にシステムに割り当てないことが望ましい。これらの様々な要因を考えて資源の割り当てを行うグローバルスケジューラの作成は非常に困難である。

そこで、SSS–CORE の基本方針に立ち戻って、実行コードの最適化はアプリケーションプログラムが自己責任で行うことにして、グローバルなカーネルレベルスケジューラは設置しないことにした。SSS–CORE には情報開示機構があり、他ノードの負荷状況や資源割り当て状況が低コストで確認できるので、各並列アプリケーションプログラムはそれらの情報を元に自分で負荷分散戦略を動的に決定する。もちろん、負荷分散を動的に行うコードを含んでいなくても、起動されたノード（またはノード群）において TSS を実現するローカルなカーネルレベルスケジューラによってアプリケーションの実行は進められて行く。負荷分散を行うコードを含んでいると、実行途中で空いているノードを見つけてタスクの実行場所を移動させて実行時間を短縮できるかもしれない。ただし、移動したノードに突然大きなジョブが投入されるかもしれないので、この自発的なタスクの移動（マイグレーション）は自己責任による一種の博打である。この動的負荷調整機能のコードはアプリケーションプログラマが記述するのではなく、基本的に最適化コンパイラが自動的にコード内に埋め込むことを SSS–CORE では想定している（プログラマが明示的に記述することを妨げはしない）。我々の最適化コンパイラ RCOP には、この方式の最初の一歩であるネットワーク飽和を避けて自ら一時的に短期間休眠するコードを自動挿入する機能がある。

負荷分散戦略を各アプリケーションに自前で決定させた場合でも、ある特定の資源への使用要求が重なってシステム全体の性能もしくは一部のアプリケーションの性能が低下することを防止するルールは必要である。例えば、ネットワークを頻繁に使用するアプリケーション同士は同一タイムスライス内に資源（プロセッサ）割り当てが行われないように調節する必要がある。また、あるアプリケーションがネットワーク飽和を回避するために通信要求の頻度を意図的に下げているのに、他のアプリケーションがその分余計に通信要求の頻度を上げていたのでは、「あつかましい」アプリケーションがシステム全体や一部の「人のいい」アプリケーションに迷惑を掛けてしまう。こういった事態を回避する必要がある。つまり、システム全体の性能低下とアプリケーション間の

公平性を守るための基本ルールを設ける必要がある。この基本ルールに関しては現在研究開発中でまだ実装はなされていない。 SSS–CORE では、競合度の高い（性能が飽和している）資源に対しての使用要求に大きなペナルティを課してタスクの優先度を下げるという基本戦略により、このルールを確立する予定である。

グローバルなカーネルレベルスケジューラというのは中央集権的な意志決定機構（計画経済）に対応し、硬直的で融通が効かなくなる可能性が高く、必ずしも個別の利益を考慮してくれない。これに対して、 SSS–CORE が最終的に採用した自己責任方式は、自由市場（自由主義経済）に似ている。本来の自由市場と同様に、情報開示と基本ルールの確立が重要である。

6 外部オペレーティングシステムエミュレータ

SSS–CORE はスクラッチから開発されたため、開発開始から二年半を過ぎても開発環境や実行環境で整備すべき機能が多すぎて、なかなかまともなアプリケーションを移植できずに困っていた。そこで思い付いたのが、外部オペレーティングシステムエミュレータである。幸いなことに TCP/IP や UDP/IP といった標準の通信機能はかなり早期にプログラムされていたため、実現が難しい機能は通信によって外部に委託してしまうことが可能である。具体的に述べると、 SSS–CORE は SunOS のシステムコール用のソフトウェアトラップを受け付けるようになっている。メモリの新規獲得（sbrk）やプロセス ID （タスク ID）の獲得といったそのマシンで解決せざるを得ない機能は SSS–CORE 内で実現するが、ファイルシステムや I/O に関連したシステムコールはそっくりそのまま SunOS を搭載しているマシンに要求を転送して、そこに常駐する SSS–CORE 用システムコールサーバにその要求を実行してもらい、結果を SSS–CORE マシンに返送してもらう。もちろん、 SunOS ではなく、 Solaris でも Linux でも NT でも、それぞれのオペレーティングシステムが動いているマシンが近くにあれば、この方式によって簡単にエミュレーション可能である。現状では SunOS のシステムコールに対してのみこの機能を実装しているが、このおかげで SunOS のライブラリが利用可能になり、市販 OS のファイルシステムや各種 I/O と SSS–CORE の MBCF や情報開示機構を組み合わせたプログラムが開発可能になった。この外部委託による他オペレーティングシステムのエミュレーションは、スクラッチからオペレーティングシステムを開発する場合には非常に便利な方式であると自負している。

7 SSS–CORE Ver.1.1 のスペック

我々が開発した汎用スケーラブルオペレーティングシステム「SSS–CORE Ver.1.1」 は Sun Microsystems 社の SPARCstation 20 またはこの互換機を Ethernet または Fast Ethernet で接続した環境で動作している。図 2に SSS–CORE の動作環境である WS クラスタの外観を示す。

図 2 SSS–CORE Ver.1.1 の動作環境

図 3 並列アプリケーションのマルチタスク実行

開発開始当初は汎用オペレーティングシステムの中核部分のみ、つまり保護やセキュリティもしくは性能上プロセッサが特権モードで動作せざるを得ない部分のみを開発目標としていた。しかし、シェル、ランタイムライブラリ整備、各種通信プロトコルの整備、ファイルシステムといった周辺を整備しなくては、当然ながらアプリケーション開発および実行がおぼつかなくて、 SSS–CORE の性能や汎用性の検証ができない。そのため、あ

る程度汎用オペレーティングシステムとしての環境整備も行わざるを得なくり、開発コストは大きなものになってしまった。

並列アプリケーションがマルチタスクで実行される様子を図 3に示す。図はあるノードのコンソール画面の映像である。右上のアプリケーションは各面に動画[2]を張り付けた二つの立方体が空中を飛行している三次元 CG になっており、 1 台の表示ノードと動画再生および三次元透視変換および座標変換を行う 6 台のノード（各ノードは立方体の表裏二面ずつを担当）の計 7 台のノードで並列処理されている。左上のアプリケーションはレイトレーシングを実行中で、各計算ノードは 4 ピクセルを単位としてサイクリックに光線追跡計算を担当している。1 台の表示ノードと 8 台の光線追跡ノードの計 9 台で並列処理がなされている。レイトレーシングの計算の進捗が計算ノードごとにばらついているのは、動画付きの立方体の計算時間がノードごとに異なる（二つのノードには立方体の計算は割り当てられてすらいない）ことによる。このように SSS–CORE 上ではさまざまの並列度のアプリケーションを TSS によるマルチタスクで実行することができる。

以下に現バージョンの SSS–CORE Ver.1.1 において実現されている主な機能を列挙する。

SSS–CORE Ver.1.1 に実装された主な機能:

- マルチタスク（時分割 + パーティショニング）
- タスク間保護, メモリ保護
- 二種類のシステムコール
  - 軽いシステムコール（主にメモリベース通信サポート用に使われ、コンテクスト切替を伴わない）
  - 通常のシステムコール（タスク管理、メモリ管理、ノード内タスク間通信、標準通信機能）
- 標準外部通信プロトコル UDP/IP、 TFTP、 TCP/IP、 TELNET、 TELNET による複数端末の接続
- オリジナル C 言語ライブラリ
- ノード内タスク間共有メモリ
- グラフィック表示可能コンソール, キーボード入力, マウス入力
- OS 組み込みシェル機能
  - TFTP によるファイルロードとコマンド実行、バッチファイル実行
  - 環境変数、コマンド行編集、コマンドリトリーブ機能
  - ハードウェアのパラメータ設定・表示機能
  - デバッグサポート（メモリダンプ編集、スタックダンプ編集、レジスタダンプ編集、フォルトタスクのデバッグ）
- SunOS システムコール エミュレーション（ファイルシステムの外注）
- オンメモリファイルシステム（ファイルアクセスの高速化）
- BOOTP による実行環境パラメータの取得
- Ethernet の全二重通信およびオートネゴシエーションのサポート
- ソフトウェアメモリベース通信（MBCF）
  - ノードの仮想化、保護、セキュリティ
  - エラスティックメモリバリア [21]機能オプション
  - ステータス返送オプション
  - WRITE, WRITE_F, READ, SWAP, etc.
  - Memory-Based FIFO (仮想化された高速ノード間 FIFO キュー)
  - Memory-Based Signal (仮想化された高速遠隔呼び出し)
  - 階層マルチキャストと Ack コンバイニング
  - TCP/IP および UDP/IP と HW 共用可能
- 非対称分散共有メモリ（高速ソフトウェア分散共有メモリ）用ページ管理機構
- PARMACS マクロ対応の非対称分散共有メモリ用最適化コンパイラプロトタイプ RCOP（クロスコンパイラ方式） [13] [22]
- 情報開示機構の部分的実装（V1.1 ではネットワーク混雑状況とメモリ使用状況の開示）
- メモリベース通信を使用した高速 MPI ライブラリ（MPI/MBCF） [23] [24]
- メモリベース通信を使用した高速 Remote Procedure Call（RPC/MBCF）

SSS–CORE は性能最重視の汎用オペレーティングシステムであるため、いわゆるマイクロカーネルアプローチの OS とは異なり性能を低下させてまでカーネル部分をコンパクトにするというアプローチは採っていない。多くのシステムコールは、システムコールの回数が大幅に削減可能になるように、データ粒度の大きな操作が可能になっている。例えば、メモリページ無効化であれば、 1 ページの無効化手段のみを提供するのではなく、連続した複数ページを一回のシステムコールによって無効化することが可能となっている。もちろん、 SSS–CORE においても汎用化にも性能向上にも不必要である機能までカーネル内に取り込むようなことは避ける方針である。気づかれたかどうか判らないが、 SSS–CORE のシェル機能がカーネル組み込みとして提供されているのは、この方針に反している。シェル機能をユーザモードで実現するためには、シェル機能のためだけに多くのシステムコールを作成しなくてはならない。このため、開発期間を短縮するために、カーネル内で実現するという妥協を行った。将来のバージョンではカーネルの外に追い出したいと考えている。

8 SSS–CORE の性能評価

SSS–CORE の設計思想と主な新規機能について述べてきたが、性能重視の汎用スケーラブルオペレーティングシステムとして実際に従来のオペレーティングシステムに比べて性能が向上しなくては、いくらもっともらしい理屈を並べてもお話にならない。本節では、 SS20 を

2 立方体表面のアニメーション「勇者警察ジェイデッカー」の著作権は名古屋テレビ株式会社および株式会社サンライズに帰属する。

使った WS クラスタにおける SSS–CORE Ver.1.1 の性能に関する評価結果を示す。

8.1 基本性能評価

まず、もっとも基本的なシステムコール（軽量システムコール）である get_taskid（UNIX の getpid に相当）のコストは 1.12μsec であり、同一 HW 環境における SunOS4.1.4 の getpid のコスト 4.39μsec よりもかなり高速である。

SSS–CORE に設計思想で大きなウェイトを占める、属性を指定して実メモリページを獲得する（物理メモリの獲得と論理アドレスとのマッピングを行う）システムコールとそのメモリを返却するシステムコールの実行時間をメモリサイズ毎に表 1に示す。 SSS–CORE Ver.1.1 は 4K, 256K, 16M, 4G の各ページサイズを扱うことが可能であるが、本測定ではソフトウェア分散共有メモリの便宜を考えて、獲得領域のサイズによらず 4K ページサイズを指定した。参考までに、 SunOS4.1.4 の sbrk（メモリ獲得に相当）のコストを表内に掲載する。

表 1 メモリ獲得 / 返却システムコールのコスト (μsec)

サイズ (byte)	4K	16K	64K	256K	1M
alloc	23.91	28.91	48.77	123.2	431.2
free	19.49	20.36	23.91	36.23	99.06
SunOS sbrk	133.2	375.8	894.3	1828	2020

表 2に MBCF の MBCF_WRITE（遠隔書き込み）の最高通信バンド幅を送信当たりのデータサイズを変えながら測定した結果を示す。ノード間の接続は半二重通信と全二重通信の二通りで行った。半二重通信では 100BASE-TX(Fast Ethernet) をハブによって接続し、全二重通信ではスイッチングハブによって接続した。 SunOS 4.1.4 の TCP/IP では、このデータサイズでは大幅に低いバンド幅しか記録できない。

メモリベース通信の MBCF_WRITE、 MBCF_FIFO (Memory-Based FIFO)、 MBCF_SIGNAL (Memory-Based SIGNAL) の片道の通信遅延時間を表 3に示す。ノード間の接続は最高バンド幅の半二重通信と同じである。 MBCF_FIFO の遅延時間は送信側が MBCF 要求システムコールを発行してから、受信側がポーリングによって到着を検知し、システムコールによって受信データを FIFO キューから別メモリ領域に読み出し終るまでの時間である。実際の測定は、受信側が再度 MBCF_FIFO コマンドで発信元に同一データを転送して、転送元が返送されたデータを読み出した時刻から最初の要求発行時刻を引いてラウンドトリップタイムを求めて半分にした。この MBCF_FIFO のレイテンシは、二回の軽量システムコールと 1 回の FIFO 読み出しに伴うデータコピーが含まれた値である。

表 3 MBCF/100BASE-TX の片道遅延時間 (μsec)

data size (byte) command	4	16	64	256	1024
MBCF_WRITE	**24.5**	27.5	34	60.5	172
MBCF_FIFO	**32**	32	40.5	73	210.5
MBCF_SIGNAL	**49**	52.5	60.5	93	227.5

MBCF_SIGNAL の遅延時間は送信側が MBCF 要求システムコールを発行してから、受信側において設定されたプログラム（DSP）が起動されて、その起動プログラム内で MBCF_SIGNAL コマンドが運んだデータを FIFO キューから読み出すシステムコールを終了するまでの時間である。実際の測定は、 MBCF_FIFO コマンドの場合と同様に、ラウンドトリップタイムを求め半分にした。この MBCF_SIGNAL のレイテンシは、二回の軽量システムコールと 1 回のユーザ権限でのプログラム起動（アップコール）と 1 回の FIFO 読み出しに伴うデータコピーが含まれた値である。

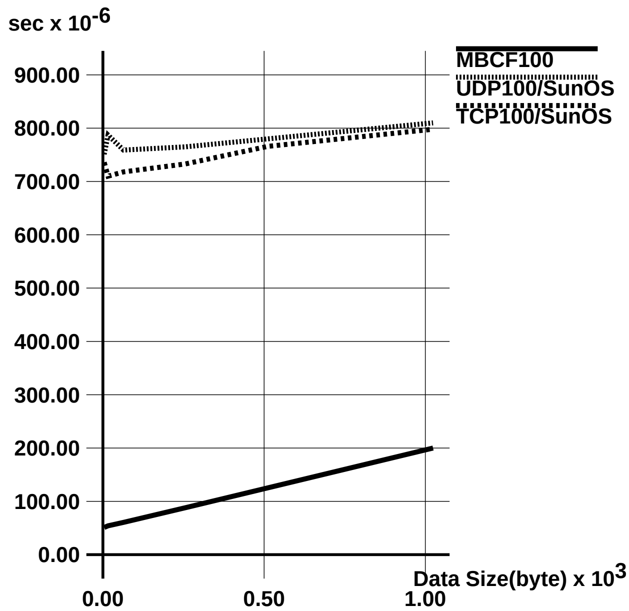

図 4 Round-trip タイムの比較

性能比較として、表 3の MBCF_WITE 内容（正確には往復遅延なので表の値の二倍）と、同一ハードウェアに対して SunOS4.1.4 の TCP/IP (図中 TCP100/SunOS) および UDP/IP （図中 UDP100/SunOS）ソケットライブラリを用いた場合のラウンドトリップ（往復）遅延時間と転送パケットサイズの関係を図 4に示す。 MBCF100 が MBCF_WRITE のラウンドトリップ遅延時間を示す。ただし、 TCP/IP のソケットには TCP_NODELAY オプションを付加することにより、ストリーム通信を基本とする TCP が細粒度通信のために不利にならないようにした。 UDP/IP のケースでは、 UDP/IP のみではパケットの転送保証および順序保証が行なわれないため、 UDP/IP 上に MBCF のパケット転送プロトコルを実装し測定した。 64byte 以下のデータ長のパケットでは、前出の高速実装技法を駆使した SSS-CORE 上の MBCF は、 SunOS のソケットライブラリを使った通信に比べ、 10 分の 1 以下のレイテンシで通信が可能である。送信ノードと受信ノードにおけるプロセッサの処理オーバヘッドにいたっては、 MBCF は TCP/IP の 100

表 2 MBCF/100BASE-TX のピークバンド幅 (Mbyte/sec)

data size (byte)	4	16	64	256	1024	1408
MBCF/100BASE-TX (半二重)	0.31	1.15	4.31	8.56	11.13	**11.48**
MBCF/100BASE-TX (全二重)	0.34	1.27	4.82	9.63	11.64	**11.93**

表 4 100BASE-TX における MPI/MBCF の round-trip 遅延時間 (μsec)

message size (byte)	0	4	16	64	256	1024	4096
MPI/MBCF on SSS–CORE	**71**	85	85	106	168	438	1026
MPICH/TCP on SunOS	968	**962**	980	1020	1080	1255	2195

分の 1 程度(データコピーを除く)で済んでいる。

SSS–CORE は MPI で書かれたプログラム資産を利用するために、MBCF を使った高速 MPI ライブラリ MPI/MBCF [23] [25]を用意している。MPI/MBCF によるラウンドトリップ遅延時間を測定した結果を表 4 に示す。ノード間の接続方法は MBCF の半二重通信の測定と同じである。比較のため、同一ハードウェア環境において SunOS 4.1.4 上の MPICH Ver. 1.1 [26]を用いた場合の実行時間を測定した。MPICH は Argonne National Laboratory および Mississippi State University において開発された MPI の標準的かつ代表的な実装で、WS クラスタに対する実装では TCP ソケットによる通信を行っている。表から判るように、SunOS から SSS–CORE に基本ソフトウェアを置き換えることにより、MPI 通信の遅延を大幅に短縮できる。

半二重通信ネットワーク(ハブ接続)および全二重通信ネットワーク(スイッチングハブ接続)上で測定した MPI/MBCF の最大バンド幅を表 5 に示す。遅延時間の測定と同様に、参考として SunOS 4.1.4 上の MPICH によるピークバンド幅も同時に示す。ただし、SunOS 4.1.4 が全二重通信に対応していないため、MPICH の測定値の表示は半二重通信のみである。表から判るように、SunOS から SSS–CORE に基本ソフトウェアを置き換えることにより、MPI 通信のバンド幅を大幅に改善できる。特に、データサイズが細かい通信に関する改善は目覚しいものがある。

8.2 NPB による性能評価

前小節ではシステムコールや通信同期といった個別の機能に関する性能を示した。本小節では、NAS Parallel Benchmarks を使用した総合的な性能評価結果 [27]を示す。幸いにも、SSS–CORE Ver.1.1 の HW 環境では他の OS(今回は SunOS を使用)上で同一のベンチマークが実行可能であり、基本ソフトウェア間の性能比較が可能である。

NAS Parallel Benchmarks (NPB) は、NASA Ames Research Center において開発された航空力学数値シミュレーションプログラムを基にした、並列計算機向けのベンチマークである。問題とそれを解くアルゴリズム、問題サイズのクラス、プログラミングモデルを定めた NPB 1.0 [28]と、MPI を用いた実際のプログラムを提供する NPB 2.x [29]とがある。今回使用したのは NPB 2.3 である。以下の 5 つのカーネルプログラムおよび 3 つの計算流体力学 (CFD) アプリケーションからなっている。

- カーネル
  - **EP** 乗算合同法による正規乱数の生成
  - **MG** 簡略化されたマルチグリッド法による 3 次元ポアソン方程式の解法
  - **CG** 共役勾配法による正値対称疎行列の最小固有値問題の解法
  - **FT** 高速フーリエ変換による 3 次元偏微分方程式の解法
  - **IS** 大規模整数ソート
- CFD
  - **LU** Symmetric SOR による LU 分解
  - **SP** 非優位対角なスカラ五重対角方程式の解法
  - **BT** 非優位対角な 5×5 ブロックサイズの三重対角方程式の解法

NPB 2.x では IS のみ C + MPI で、IS 以外は Fortran90 + MPI で記述されている。8 つの問題それぞれに関して問題サイズが小さい方から順に class S (サンプル)、class W (小規模ワークステーションクラスタ向け)、class A (中規模ワークステーションクラスタ向け)、class B (中規模並列計算機向け)、class C (大規模並列計算機向け) というクラス分けがなされている。

コンパイラとして gcc-2.7.2.3 および g77-0.5.21 を用いた。8 つの問題のうち FT のプログラムは g77 ではコンパイルできないため今回の評価の対象から省いた。

問題サイズとして class W を用いた。これは、class A のプログラムが、SSS–CORE 上では動作したものの SunOS 4.1.4 上ではメモリ不足により実行できなかったためである。

表 6 に、7 つのベンチマークプログラムの結果を示す。各測定値は、各プログラムをノード数 8 (SP,BT は 9) の条件の下で実行し、測定したものである。表の項目を順に説明する。

最初の 7 つの項目は SSS–CORE 上で MPI/MBCF を利用して、ベンチマークプログラムを実行した結果に関するものである。実行時間は 8 台(ないしは 9 台)による並列実行の実行時間を示し、対 1 台スピードアップ率は 1 台によるシーケンシャル実行に対する高速化率を示している。続く 5 つの項目はプログラムの性格を示すために測定した項目である。これらは MPI/MBCF のソース中に計数コードを挿入して測定したデータから算

表 5 100BASE-TX における MPI/MBCF の ピークバンド幅 (Mbyte/sec)

message size (byte)	4	16	64	256	1024	4096	16384	65536
MPI/MBCF (半二重)	0.14	0.53	1.82	4.72	8.08	9.72	**10.15**	9.78
MPI/MBCF (全二重)	0.14	0.57	1.90	5.33	10.22	11.68	11.77	**11.85**
MPICH/SunOS (半二重)	0.02	0.09	0.35	1.21	3.37	5.76	5.33	**6.04**

表 6 NPB ベンチマークプログラムによる性能評価

プログラム [実行台数]	EP [8]	MG [8]	CG [8]	IS [8]	LU [8]	SP [9]	BT [9]
MPI/MBCF on SSS–CORE Ver.1.1							
実行時間 (sec)	15.14	7.48	11.02	3.02	160.36	154.91	67.30
対 1 台スピードアップ率 (倍)	7.99	5.24	6.27	3.33	6.26	8.11	9.16
通信データレート (MByte/s)	0.00	9.68	12.69	13.58	1.89	7.83	5.32
通信メッセージレート (個 /s)	4	4670	2138	466	1199	421	488
平均メッセージサイズ (KByte)	0.00	2.07	5.93	29.14	1.57	18.60	10.90
MBCF_WRITE 利用率 (%)	51.10	0.01	53.33	99.22	13.37	49.01	47.24
集団通信関数使用の有無	あり	なし	なし	あり	なし	なし	なし
MPICH on SunOS 4.1.4 + TCP/IP							
実行時間 (sec)	16.25	13.72	14.59	4.81	185.04	231.66	96.02
対 1 台スピードアップ率 (倍)	7.73	2.83	4.71	2.13	5.84	6.01	6.53
MPI/MBCF vs. MPICH/SunOS							
性能向上率 (倍)	1.07	1.83	1.32	1.59	1.15	1.50	1.42

出した。通信データレートは MPI 関数によって送信されたデータのバイト数 (ヘッダ部分は除く) を全ノードに関して合計し、実行時間で割ったものである。通信メッセージレートは送信されたメッセージの個数を全ノードに関して合計し、実行時間で割ったものである。平均メッセージサイズは一回のメッセージ通信に含まれる平均データサイズである。 MBCF_WRITE 利用率は、全ノードから送信されたデータのうち、対応する受信が先行していたために MBCF の遠隔メモリ書き込みにより転送されたもののデータ量 (バイト数) の割合である。そして、集団通信関数使用の有無はプログラムが MPI の集団通信関数を使用しているかどうかを示す。

次の 2 つの項目は SunOS 4.1.4 上で MPICH を利用して、ベンチマークプログラムを実行した結果に関するものである。そして、最後の項目は MPICH による実行時間を MPI/MBCF による実行時間で割ったものであり、 SunOS の代わりに SSS–CORE を採用した場合の性能向上率に当たる。

通信データレートや通信メッセージレートが多様であることから、 NPB が多くの種類の並列プログラムを代表していると考えることができる。そして、どのプログラムに関しても、 SSS–CORE における実行が SunOS における実行を上回っており、同一 HW 環境にもかかわらず 5 割以上も性能が向上するプログラムが 7 本中 3 本も存在した。また、 MG においては、 SSS–CORE の OS としての採用が 83% の性能向上を可能にする。なお、 NPB は MPI で書かれた並列アプリケーションであり、 SSS–CORE が得意とするタイプのアプリケーションではないことに留意しておいていただきたい。共有メモリモデルに従った並列アプリケーションであれば、 SSS–CORE はもっと高性能を発揮できたと予想される。もっとも、 SSS–CORE 以外の OS では、同一の HW 環境で共有メモリモデルの並列アプリケーションを動作させること自体が不可能であるため、今回のような基本ソフトウェア間の比較が不可能である。

8.3 SPLASH-2 による性能評価

共有メモリ並列プログラムのベンチマーク集である SPLASH-2 [30] のほとんどのプログラムを最適化コンパイラ RCOP によって、 SSS–CORE の上で効率良く動かすことに成功している。紙面の都合があるため、ベンチマークプログラムのうちの LU-Contig と Radix を取り上げて、共有メモリ並列プログラムに対する性能評価を示す。

RCOP による最適化の効果

表 7と表 8に、それぞれ LU-Contig と Radix に対して、 8 台で並列実行する場合の RCOP の最適化の効果を示す。 Opt の欄は適用した最適化の種類を示し、 NO は最適化なし（つまりすべての共有書き込みにキャッシュコンシステンシ管理コードを挿入した）を示し、 MB は動的なパケットコンバイニングの適用、 IA はコアレシングの適用を示し、 MB & IA はコンバイニングとコアレシングを共に適用したことを示す。 #CM の欄は RCOP によって挿入されたキャッシュコンシステンシ管理コードを何回実行したかを示し、パケット数と通信量はそれぞれ実行終了までに 1 台のプロセッサが送信した総パケット数と総データ量を示す。なお、コンバイニングされたパケットはペイロード内のメモリベース通信のパラメータをデータとしてカウントしてしまうため、 MB 最適化を施した方が転送量が増大する場合がある。

表 7から判るように、 LU-Contig はコアレシング（IA）

表 7 **LU-Contig (n=512,b=16)** に対する最適化の効果

Opt	実行時間 (sec)	#CM	パケット数	通信量 (MByte)
NO	28.20	5592K	5206K	47.73
MB	14.35	5592K	83.5K	113.00
IA	2.17	1.43K	7.73K	9.42
MB & IA	2.16	1.43K	7.60K	9.27

表 8 **Radix (#key = 1M)** に対する最適化の効果

Opt	実行時間 (sec)	#CM	パケット数	通信量 (MByte)
NO	21.90	793K	3220K	76.72
MB	12.13	793K	75.8K	101.08
IA	1.57	2.08K	19.5K	13.47
MB & IA	1.24	2.08K	10.1K	13.63

によって劇的にキャッシュコンシステンシ管理コードの実行回数の削減が可能であり、パケット数や通信量もそれに伴って激減させることに成功している。コンバイニング（MB）のみの適用によって LU-Contig の総パケット数は大幅に減少するが、その効果はコアレシングに含まれており、コアレシング最適化にコンバイニング最適化を加えても性能の改善は僅かである。

Radix に関しても、表 8から判るようにコアレシング（IA）最適化の効果が非常に大きい。ただし、 Radix の場合にはコアレシング最適化後にコンバイニング最適化を加える余地がある。両最適化を適用することでコアレシング最適化のみの場合に比べて、総パケット数が約半分になり、実行時間が約 21% 改善される。

最適化の効果の程度には差があるが、 RCOP の最適化が分散メモリ環境で共有メモリプログラムを動作させるのに非常に有効であることがこれらの表から判る。

最適化コードの並列実行時間

次に、 RCOP による出力コードの並列実行台数による実行時間の短縮効果を示す。他の性能評価のように SunOS 上で比較実験をするために、 TCP/IP によってメモリベース通信をエミュレートするのは、オーバヘッドが大き過ぎてあまりに現実的ではない。このため、ここでは比較対象として、専用ハードウェアによって連続領域の遠隔書き込みと遠隔読み出しが可能な AP1000+ [31]を使用する。 AP1000+ は各ノードに 50MHz の SuperSPARC プロセッサを持ち、外づけの二次キャッシュはない。ノード間は二次元トーラスの専用ネットワークで結合されており、リンクごとに 25Mbyte/sec の通信能力がある。それに加えて、遠隔書き込みと遠隔読み出しを行う専用ハードウェアを持っており、これらのオペレーションはそのハードウェアにパラメータを設定するだけで実現され、プロセッサのオーバヘッドは非常に小さい。 AP1000+ はマルチタスクのオペレーティングシステムが実装されておらず、通信は仮想化されていない。 AP1000+ ではページトラップをサポートすることができないため、本評価では UDSM に基づくキャッシュエミュレーションを行い、 AP1000+ では遠隔ノードへのサービス要求をポーリング（ループのバックエッジと関数呼び出し時にチェック）で検出している。 UDSM のコードは RCOP が出力したコードに対して、共有読み出し用のコード挿入と人手による共有読み出し用コードの最適化を施して作成した。なお、 SSS–CORE における遠隔ノードへのサービス要求は MBCF_SIGNAL によって実現されている。

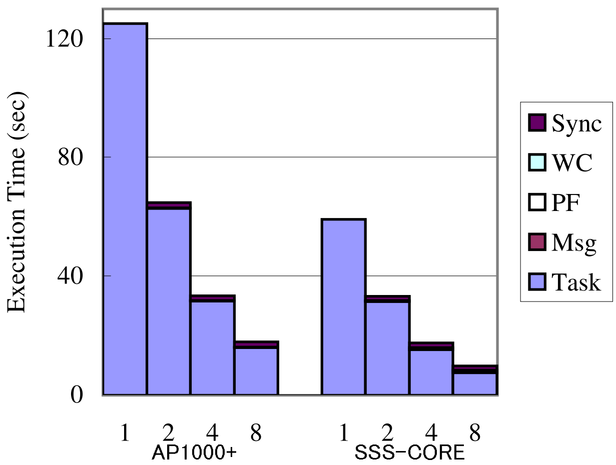

図 5 **SPLASH2 LU** の実行時間と台数効果

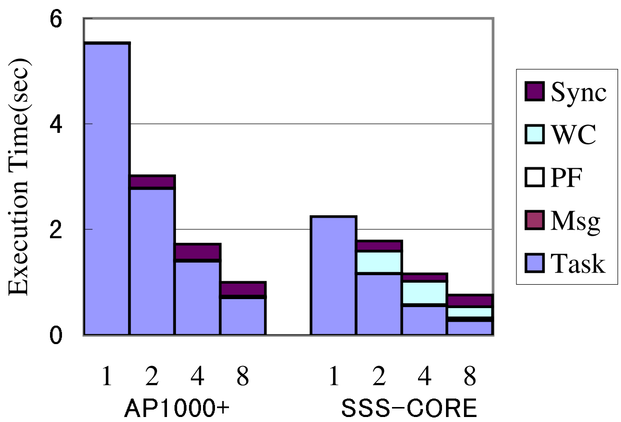

図 6 **SPLASH2 Radix** の実行時間と台数効果

図 5に LU-Contig (n=1024) に関する台数効果、図 6に Radix (#key=1M) に関する台数効果を示す。

プロセッサは同一タイプであるが、 HW や実行環境が大幅に異なるために、 SSS–CORE に関する正確な評価をこれらの図から引き出すことは困難である。しかし、図 5の LU-Contig においては専用並列計算機である AP1000+ に劣らない台数効果を示し、クロック比以上の高速性を示している。図 6の Radix に関しては、 AP1000+ よりもオーバヘッドが目立っている。しかし、これは Fast Ethernet の通信能力が不足して、ネットワークが飽和しているのが主な理由である。このため、 AP1000+ 並に早い通信 HW を使って比較しな

ければ、一概に結論は下せない。ただし、SPLASH-2 の Radix 自身が非常に並列効果を得難いアプリケーションであり、分散メモリ並列計算機 (AP1000+) や WS クラスタにおいて台数効果によるスピードアップを達成していること自体が驚くべきことである。最適化コンパイラによってサポートされた分散共有メモリという UDSM や ADSM のアプローチの有望性を物語っている。

9 研究開発作業について

SSS–CORE の前身となる SMP(Symmetric Multi-Processor) 用の汎用オペレーティングシステム SS–CORE [32]の研究開発（1990 年）を某研究所において途中で断念させられた松本が平木の理解と協力を得て、新たに平木研プロジェクトとして 1994 年に情報処理振興事業協会（IPA）に予算申請し、開発予算を認めてもらったことに SSS–CORE の研究開発は端を発している。 SSS–CORE の全体設計と各種機構の設計を松本が担当し、ブートストラップコード、デバイスドライバ、タスク管理、メモリ管理、システムコール、 MBCF、 IDM、 C 言語ライブラリ、 TCP/IP、 TELNET、シェル、コンソール等のコア部分のプログラミングは松本と株式会社アックスの駒嵐丈人 (1994,1995)、渦原 (1996,1997) が担当した。開発初期の単体 WS 用オペレーティングシステム（SSS–MC）としての開発は株式会社アックスの組み込み用オペレーティングシステム Crystal をモデルにして行われたため、株式会社アックス代表の竹岡の設計思想やプログラミングスタイル（開発コストを極力減らすための各種工夫）も速度が問題とならない機能に反映され残っている。保護と仮想化を満たして安定動作する SSS–CORE Ver.1.0 の開発には四年の歳月がかかった。この開発期間のうちハードウェア情報の収集や ROM 内ルーチンの解析等に多くの労力が割かれており、プログラミング作業自体は 2 人×1.5 年程度である [3]。 WS に使用されている LSI のデータシートならびにユーザマニュアルの手配には、日本サンマイクロシステムズ株式会社の方々に協力していただいた。しかし、 LSI のマニュアルが手に入っても、マニュアルのバグや LSI のバグを避けて正常に動くパラメータの組み合わせを解明するのに時間を費やすことも多かった。また、サンマイクロシステムズ社以外の LSI のプログラミングデータ収集にはインターネットが活躍し、 WS の基板上に見つけた LSI の型番をサーチエンジンに問い合わせてデータシートを獲得したものも少なくなかった[4]。もちろん、 RFC や各種標準規格に関する情報もインターネットを介して獲得した。汎用オペレーティングシステムの作成には膨大な資料が必要であり、小人数のスタッフで曲がりなりにも実用に耐える汎用オペレーティングシステムが完成できたのはインターネットを通じて多くの情報が利用できたおかげである。逆に小人数であったために小回りがきき、ほとんど妥協なしに高性能なオペレーティングシステムを作ることができたと考えている。 SSS–CORE の売り物の一つである共有メモリベースの並列プログラムに対する最適化コンパイラ（RCOP）は平木研究室の丹羽純平と稲垣達氏[5]の力作である。 MBCF を活用した大規模アプリケーション第一号である動画再生プログラム xanim の移植ならびに xanim を使った三次元 CG デモプログラムと MPI/MBCF の開発は平木研究室の森本健司が行い、 RPC/MBCF の作成および BSD4.3 の TCP/IP の移植[6]は平木研究室の亀澤寛之によってなされた。初期のスケジューリング方式に関する精緻なシュミレーションは平木研究室の信国陽二郎[7]によってなされ、この結果に関して多くの外部発表がなされた。スケジューリング方式の転換によって、彼の実験結果と考察は直接コードに活かされることはないかもしれないが、開発初期段階における SSS–CORE の広報活動に非常に貢献してくれた。また、 SSS–CORE の開発環境の整備には多くの平木研究室メンバの支援を受けた。

10 おわりに

汎用超並列オペレーティングシステムカーネル SSS–CORE の研究開発は、 1994 年度 1 年間の実現可能性調査の予備研究、 1995 年度から 3 年間の本研究の計 4 年間の所定の研究期間を 1998 年 2 月をもって終了した。研究開発開始当初の計画では、カーネル部分のみのパイロットモデルの完成を目指す予定であった。しかし、1996 年度の研究開発における今後の汎用並列分散処理のキーテクノロジーとなりうるメモリベース通信機能 (MBCF) の考案に伴って、完成度の高いアプリケーション開発実行環境を含む総合オペレーティングシステムとしての完成を目指すことに目標を上方修正した。この目標修正により新規開発項目が大幅に増加したにもかかわらず、開発予算や開発部隊を拡大することが諸般の事情により不可能であったため、当初目標の情報開示機構の実装の完成等は期間内に間に合わなかった。しかし、画期的な高速ユーザレベル通信同期機構である MBCF を含む汎用総合オペレーティングシステム SSS–CORE Ver.1.1 をスクラッチから完成させることに成功した。また、メモリベース通信機能を利用する MPI インタフェースおよびメモリベース通信機能を活用するコードを生成する最適化コンパイラの開発によって、大規模のアプリケーションが SSS–CORE 上で効率良く実行されている。また、長期間に及ぶ連続運用に対して SSS–CORE は安定して動作している。究極的な目標である「NUMA 型アーキテクチャ上で専用ハードウェアを仮定せずに効率良く並列アプリケーションを動作させる汎用オペレーティングシステムの基盤技術を開発する」という点に関しては、実運用レベルの汎用総合オペレーティングシステムが開発できたことは当初予定された成果を大幅に上回っていると考えている。

3 実は、開発予算の確保と強く関わる広報活動や論文発表に最も多くの時間が割かれている。本原稿の執筆もその一つである。

4 現在では、サンマイクロシステムズ社の LSI のプログラミング情報も大半は Web から獲得可能である。

5 現日本アイ・ビー・エム株式会社

6 SSS–CORE には元々アックス製のオリジナル TCP/IP が搭載されているが、一部機能が未実装であるため最近になって BSD の TCP/IP も移植搭載された。

7 現株式会社アーツテック

References

[1] 松本 尚, 平木 敬: 汎用並列オペレーティングシステム SSS–CORE の資源管理方式. 日本ソフトウェア科学会第 11 回大会論文集, pp.13–16 (October 1994).

[2] T. von Eicken, A. Basu, and V. Buch: Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, pp.46–53 (February 1995).

[3] 松本, 駒嵐, 渦原, 平木: メモリベース通信による非対称分散共有メモリ. コンピュータシステムシンポジウム論文集, 情報処理学会 pp.37–44 (November 1996).

[4] Matsumoto, T., Niwa, J., and Hiraki, K.: Compiler-Assisted Distributed Shared Memory Schemes Using Memory-Based Communication Facilities. In *Proc. of The International Confference on Parallel and Distributed Processing Techniques and Applications* (PDPTA-98), Vol.2, pp.875–882 (July 1998).

[5] M. Accetta, et al.: Mach: A New Kernel Foundation for UNIX Development. *Proc. Summer 1986 UNIX Conf.*, USENIX, pp.93–112(1986).

[6] S. J. Mullender, et al.: Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, vol.23, pp.44–53 (May 1990).

[7] M. Berver, et al.: Distributed Systems, OSF DCE and Beyond. in *DCE—The OSF Distributed Computing Environment*, Berlin:Springer-Verlag, pp.1–20 (1993).

[8] 松本 尚: マルチプロセッサ上の同期機構とプロセッサスケジューリングに関する考察. 計算機アーキテクチャ研究会報告 No.79-1, 情報処理学会, pp.1-8 (November 1989).

[9] 松本 尚, 平木 敬: 超並列計算機上の共有メモリアーキテクチャ. 信技報, CPSY 92-26, pp.47-55 (August 1992).

[10] 松本 尚, 平木 敬: 汎用超並列オペレーティングシステム SSS–CORE のメモリベース通信機能. 第 53 回情報処理学会全国大会講演論文集 (1), pp.37–38 (September 1996).

[11] 松本 尚, 平木 敬: 汎用超並列オペレーティングシステム SSS–CORE のユーザレベル通信同期機構. コンピュータソフトウェア, Vol.15, No.3, pp.59–63 (May 1998).

[12] Matsumoto, T. and Hiraki, K.: MBCF:A Protected and Virtualized High-Speed User-Level Memory-Based Communication Facility. In *Proc. of the 1998 ACM Int. Conf. on Supercomputing*, pp.259–266 (July 1998).

[13] 丹羽, 稲垣, 松本, 平木: 非対称分散共有メモリ上におけるコンパイル技法. 情報処理学会研究報告 97–HPC–67, 情報処理学会, Vol.97, No.75, pp.121–126 (August 1997).

[14] Matsumoto, T. and Hiraki, K.: Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory. *Innovative Architecture for Future Generation High Performance Processors and Systems (IWIA'97)*, Edited by Veidenbaum, A. and Joe, K., IEEE Computer Society, pp.30–39 (April 1998).

[15] 松本, 駒嵐, 渦原, 竹岡, 平木: 汎用超並列オペレーティングシステム: SSS–CORE — ワークステーションクラスタにおける実現 —. 情報処理学会研究報告 96–OS–73, 情報処理学会, Vol.96, No.79, pp.115–120 (August 1996).

[16] P. Keleher, A. L. Cox, and W. Zwaenepoel: Lazy Consistency for Software Distributed Shared Memory. *Proc. the 19th Int. Symp. on Computer Architecture*, pp.13-21 (May 1992).

[17] J. Boyle et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[18] L. Iftode, C. Dubnicki, E. W. Felton and K. Li: Improving Release-Consistent Shared Virtual Memory using Automatic Update. *Proc. the 2nd Int. Symp. on High-Performance Computer Architecture*, pp.14-25 (February 1996).

[19] Inagaki, T., Niwa, J., Matsumoto, T., and Hiraki, K.: Supporting Software Distributed Shared Memory with Optimizing Compiler. In *Proc. of the 1998 Int. Conf. on Parallel Processing*, pp.225–234 (August 1998).

[20] 丹羽 純平, 稲垣 達氏, 松本 尚, 平木 敬: 汎用超並列オペレーティングシステム SSS–CORE 上の非対称分散共有メモリにおける最適化コンパイル技法. コンピュータソフトウェア, Vol.15, No.3, pp.54–58 (May 1998).

[21] 松本 尚, 平木 敬: Elastic Memory Consistency Models. 第 49 回情報処理学会全国大会講演論文集 (6), pp.5–6 (September 1994).

[22] 丹羽 純平, 稲垣 達氏 松本 尚, 平木 敬: 汎用超並列オペレーティングシステム SSS-CORE — コンパイラによる通信最適化技法 —. 第 56 回情報処理学会全国大会講演論文集 (1), pp.15–16 (March 1998).

[23] 森本 健司, 松本 尚, 平木 敬: 汎用超並列オペレーティングシステム SSS-CORE — 高速 MPI の実装と評価 —. 第 56 回情報処理学会全国大会講演論文集 (1), pp.13–14 (March 1998).

[24] 森本 健司, 松本 尚, 平木 敬: メモリベース通信を用いた高速 MPI の実装方式. 並列処理シンポジウム JSPP'98 論文集, pp.191–198 (June 1998).

[25] Morimoto, K., Matsumoto, T. and Hiraki, K.: Implementing MPI with the Memory-Based Communication Facilities on the SSS–CORE Operating System. *Proc. of 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'98)*, Springer-Verlag LNCS 1497, pp.223–230 (September 1998).

[26] Gropp, W., Lusk, E., Doss, N., and Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard. *Parallel Computing*, Vol.22, No.6, pp.789–828 (September 1996).

[27] 森本 健司, 松本 尚, 平木 敬: 並列アプリケーションによる MPI/MBCF の評価. ハイパフォーマンスコンピューティング研究会報告 No.72–18, 情報処理学会, pp.103–108 (August 1998).

[28] Bailey, D. and et al.: THE NAS PARALLEL BENCHMARKS. NASA Ames Research Center, Technical Report *RNR-94-007* (March 1994).

[29] Bailey, D. and et al.: The NAS Parallel Benchmarks 2.0. NASA Ames Research Center, Technical Report *NAS-95-020* (December 1995).

[30] S. C. Woo, et al.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd ISCA*, pages 24–36, June 1995.

[31] 林憲一 他: PUT/GET インタ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. 並列処理シンポジウム JSPP '94 論文集, pp.233–240 (May 1994).

[32] 松本, 根岸, 渦原, 森山: 粒度に基づいた並列計算の分類法とマルチプロセッサの資源管理法について. 日本ソフトウェア科学会第 7 回大会論文集, pp.133-136 (October 1990).