スケーラブルな分散サーバ環境の研究
— *SSS–CORE* の実用化に向けて —
A Study of the scalable distributed-computing servers
— Toward practical use of the *SSS–CORE* —

松本　尚 [*1*2]
tm@is.s.u-tokyo.ac.jp

渦原　茂 [*3]
uzu@axe-inc.co.jp

竹岡 尚三 [*3]
take@axe-inc.co.jp

平木 敬 [*1]
hiraki@is.s.u-tokyo.ac.jp

[*1] 東京大学 大学院理学系研究科 情報科学専攻
[*2] 科学技術振興事業団 さきがけ研究 21 「情報と知」領域
[*3] 株式会社 アックス

近年、コンピュータネットワーク（インターネットや企業内ネットワーク）の規模は拡大し、ネットワーク利用者は急速に増大している。また、ネットワークの利用方法も年々高度化して、マルチメディアデータがネットワーク上において流通し始めている。これに伴い、ネットワークを流れるデータ量が大幅に増大している。このため、ネットワークを介した情報処理の中核となるコンピュータ（サーバマシン）への性能要求が増大している。企業内ネットワークを構築する企業やインターネットプロバイダはこの要求増大に、高価なサーバ用計算機（専用並列計算機や大型機）を導入することにより対処している。既に導入したサーバマシンの処理能力が不足した場合には、さらに高価な計算機とリプレースする必要に迫られ、リプレース時にはソフトウェアおよびデータを移し換えるために、多大な作業コストが発生する。

本研究においては、これらの状況を大幅に改善するために、サーバマシンをワークステーションクラスタによって構築し、その能力を構成マシンの台数によってスケーラブルに変更可能にするための基本ソフトウェア群を研究開発する。

1 はじめに

1.1 研究の背景

近年、コンピュータネットワーク（インターネットや企業内ネットワーク）の規模は拡大し、ネットワーク利用者は急速に増大している。また、ネットワークの利用方法も年々高度化して、マルチメディアデータがネットワーク上において流通し始めている。これに伴い、ネットワークを流れるデータ量が大幅に増大している。このため、ネットワークを介した情報処理の中核となるコンピュータ（サーバマシン）への性能要求が増大している。企業内ネットワークを構築する企業やインターネットプロバイダはこの要求増大に、高価なサーバ用計算機（専用並列計算機や大型機）を導入することにより対処している。既に導入したサーバマシンの処理能力が不足した場合には、さらに高価な計算機とリプレースする必要に迫られ、リプレース時にはソフトウェアおよびデータを移し換えるために、多大な作業コストが発生する。

本研究においては、これらの状況を大幅に改善するために、サーバマシンをワークステーションクラスタによって構築し、その能力を構成マシンの台数によってスケーラブルに変更可能にするための基本ソフトウェア群を研究開発する。

† 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。

1.2 期待される効果、成果

本研究開発の大局的目的は前記研究背景で示されるように、サーバ計算機の大幅なコストパフォーマンスの改善と処理能力にスケーラビリティを与えることである。本目標を達成するために下記４項目の研究開発内容を含む基本ソフトウェア群の再構築を行う予定である：

1. マルチプラットホーム間メモリベース通信ファシリティ
   メモリベース通信ファシリティ (MBCF: Memory-Based Communication Facilities) [1] [2]は、独創的先進的情報技術に係わる研究開発事業のテーマの一つとして平成６年度から約４年間研究開発を行った「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」 [3] [4]において、平成８年度に松本が考案開発した新しい通信方式である。MBCF は特殊なハードウェアをまったく必要とせず、通常の LAN に接続可能なコンピュータに保護され仮想化された高速な通信および同期手段を提

供する。現在事実上標準となっている TCP/IP プロトコルと比べて、オーバヘッドコストを二桁、レイテンシ（通信遅れ）を一桁改善することができる。この MBCF 方式をクライアントとして使われる可能性のある様々なマシンやオペレーティングシステムに移植する。MBCF プロトコルによって通信することにより、サーバに掛かる通信のためのオーバヘッドを大幅に削減することができる。

2. スケーラブルサーバ内の仮想化された高性能メモリシステムおよびファイルシステムの実現
   サーバマシンは大量のデータおよびプログラムをファイルシステムに保持管理する必要があり、ファイルシステムの性能がシステム全体の性能のボトルネックとなる可能性がある。このため、メモリ管理システムと統合された高性能ファイルシステムをサーバマシンに提供する。また、本研究において開発するサーバ用オペレーティングシステムが提供する共有メモリ機能と共有ファイルシステムを統合して、メモリ資源の有効活用とファイルシステムの高性能化を目指す。
3. 高速分散 Java 言語実行環境
   本研究において、サーバ用スケーラブルオペレーティングシステムを独自開発するため、サーバ用アプリケーションプログラムを確保する必要がある。並列化が必要な負荷の重いアプリケーションは並列処理可能な形に変更する必要がある。しかし、負荷の軽いアプリケーションまですべて新しいオペレーティングシステムに移植したのでは、移植のコストが大きくなってしまう。このため、プラットフォーム依存性がなく、アプリケーション開発言語として注目されている Java 言語の実行環境を構築する。この Java 言語実行環境自体もシステムの高速通信機構やスケーラビリティの恩恵を享受できるように、分散並列拡張を行う。
4. スケーラブルサーバ用オペレーティングシステムの実現
   独創的先進的情報技術に係わる研究開発事業のテーマ「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」として、平成 6 年度から約 4 年間研究開発を行った *SSS–CORE* オペレーティングシステムをスケーラブルサーバ用オペレーティングシステムとして機能強化と最新鋭ワークステーションへの移植を行う。機能強化の内容としては以下の項目が挙げられる。
   - オペレータから単一イメージに見えるオペレーティングシステム
   - 低オーバヘッドのメモリベース通信によって多数のクライアントに対応
   - UDP/IP，TCP/IP，MBCF/IP を高効率でサポート
   - 分散並列拡張された Java をスケーラブルに処理
   - サーバ OS としての安定性とセキュリティの確保

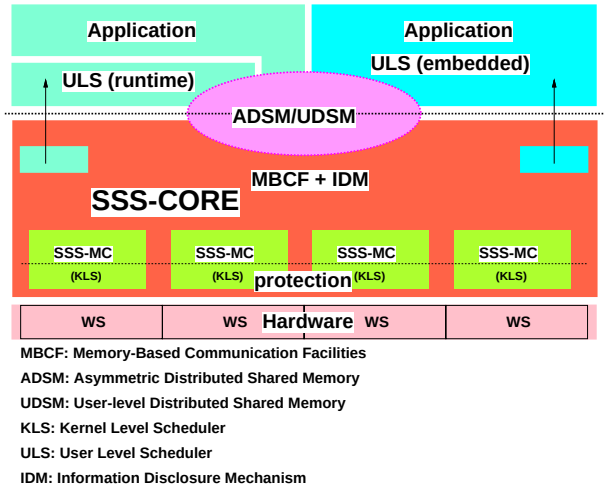

図 1　*SSS–CORE* **の機能構成**

図 2　*SSS–CORE* **Ver.2.2（Ultra 版）の開発実行環境**

   - 共有メモリ・共有ファイルシステムを活用した負荷分散

2　これまでの研究開発成果の概要

本研究開発は平成 10 年度から 3 年間の予定で始まり、平成 11 年度は 2 年目である。本節ではこれまでの研究開発成果の概要について述べる。

2.1 SSS–MC Ver.3.0 の開発

独創的先進的情報技術に係わる研究開発事業のテーマ「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」において開発した *SSS–CORE* Ver.1.1 オペレーティングシステムは、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する。SPARCstation 20 は *SSS–CORE* の開発二年目である平成 7 年度に発売されたワークステーションであり、プロセッサは SuperSPARC である。現在、Sun Microsystems 社の最新鋭ワークステーションに搭載されているプロセッサは UltraSPARC であり、単体性能は SuperSPARC の数倍高速である。*SSS–CORE* をサーバ用オペレーティングシステムとして実用的な物とするためには、UltraSPARC プロセッサへの対応が必須である。UltraSPARC はユーザレベルのコードに関して、SuperSPARC と互換性があるものの、カーネルレベル（Supervisor mode）のアーキテクチャと命令体系は大幅に改良されている。このため、プロセッサ依存のコードを大幅に変更する必要がある。また、UltraSPARC が使用された Ultra シリーズのワークステーションは Boot ROM が Open BOOT ver.3.x になっており、SPARCstation 20 の Open BOOT ver.2.x と大幅に異なっている。これらの事由により移植作業は非常に困難であったが、プロセッサ周りのコードの移植が終了して、*SSS–CORE* のノード常駐核である SSS–MC Ver.3.0 の作成に成功した。SSS–MC Ver.3.0 をベースに SBus 仕様の UltraSPARC ワークステーションに対して、*SSS–CORE* Ver.1.1 の移植版である Ultra 版 *SSS–CORE* Ver.2.1 の作成がほぼ終了している。ただし、Sun Microsystems 社は I/O バスを SBus から PCI バスへ移行しようとしており、一部の製品を除いて最近のワークステーションには PCI バスを採用している。PCI バス版のワークステーションと SBus 版のワークステーションでは、周辺チップセットが異なり、これらのチップに対する制御コードが異なる。現在、PCI バス版ワークステーションに *SSS–CORE* Ver.2.1 を対応させるべく、作業を進めている。

2.2 *SSS–CORE* Ver.1.2 の開発

UltraSPARC プロセッサへの *SSS–CORE* の移植に並行して、*SSS–CORE* 自体の機能を強化した。この機能強化に伴い *SSS–CORE* のバージョンを Ver.1.1 から Ver.1.2 に変更した。以下に機能強化の内容を列挙する。

1. 非対称分散共有メモリ [5] [6]用メモリ管理機構の実装
2. 遠隔シェル機能および並列シェル機能の実装、並列実行機能の充実
3. カーネルレベルスレッドのサポート
4. ユーザレベルメモリ操作関数の充実
5. 標準入力システムの作成
6. カーネル内 malloc への対応
7. 最適化コンパイラ RCOP [7] [8]の安定性強化

図 3 *SSS–CORE* **Ver.1.2（SS20 版）の開発実行環境**

8. MPI Ver.1 の規格のフルサポート [9] [10]
9. オリジナル C 言語ライブラリの充実
10. 外部ファイルシステムの安定性強化

第一項目に挙げた非対称分散共有メモリ（ADSM: Asymmetric Distributed Shared Memory）は「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」において、平成 8 年度に松本が考案した分散共有メモリの実現手法であり、共有メモリサポートハードウェアのない NUMA（Non-Uniform Memory Access）環境上であっても、コンパイラのサポートによって効率良く分散共有メモリを実現する手法である。「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」においては、時間切れで ADSM の実装が不可能であった。ADSM においては、共有メモリへの共有書き込みは丹羽純平らが作成したコンパイラ（RCOP: Remote Communication OPtimizer）で読み切って、最適化された明示的なキャッシュ管理コードと通信コードに変換された実行コードが生成される。共有メモリの読み出し時のキャッシュミスに関しては、オペレーティングシステム（*SSS–CORE*）がページトラップによって検出し、ユーザのページハンドラを呼び出す。*SSS–CORE* の上では、ページ管理機構を使用しないコンパイラサポートによる UDSM（User-level Distributed Shared Memory）が RCOP によって既に実現されていた。このため、UDSM と ADSM の性能比較が可能になり、アプリケーションによる UDSM と ADSM の特性が調べられた。

第二項目に挙げた遠隔シェル機能および並列シェル機能は、*SSS–CORE* の高効率な並列実行環境をより簡単に操作可能にするために実装された。遠隔シェルは

表 1　RPC/MBCF のラウンドトリップ最小遅延時間

data size (byte)	4	256	512	1024
RPC/MBCF_Fifo (μs)	148	194	251	372
RPC/MBCF_Signal (μs)	127	173	221	315
RPC/UDP(SunOS)(μs)	605	661	719	797
RPC/TCP(SunOS)(μs)	712	790	794	850

UNIX の遠隔シェル（remote shell）とは異なり、遠隔シェルコマンドが実行されたノード（マシン）の環境を実行対象となるノードに受渡してコマンドが実行される。これは *SSS–CORE* によって提供される計算環境がどのノードからも同じに見えることを利用している。このため、オペレータは環境変数の設定やホームディレクトリの移動を自分が直接操作しているシェルの上で行うだけで、他のマシンにも同一環境を自動的に引き渡して、操作しているマシンと同一環境下でコマンドをリモート実行することができる。並列シェルは遠隔コマンド実行を同時に複数のマシンに反映することができる機能である。しかも、これらの機能のマシン間の通信同期はすべてセキュアな MBCF 通信によって実現され、非常に高速かつ高セキュリティである。また、並列実行されるアプリケーションにとっては、通信同期を行うタスク間のセキュリティおよび保護の確保は重要ではない。しかし、 MBCF は保護され仮想化された通信機構であるため、並列アプリケーションであっても、通信同期を行うためには保護やセキュリティに関する初期設定や通信相手の登録操作をプログラム内で行う必要があった。プログラム作成を簡略化するために、並列アプリケーションに関して、各ノードでアプリケーションの構成要素タスクを生成すると同時に、 MBCF 通信の初期設定を自動的に行う並列実行機能を作成した。遠隔シェル機能および並列シェル機能の具体的な使用方法および実装に関しては後述する。

第三項目に挙げたカーネルレベルスレッドもプログラマの負担を軽減するために導入された。 *SSS–CORE* において、従来からタスクを新規に生成することも、タスク間で共有メモリを張ることも可能であった。しかし、共有メモリを張ってから、その領域内に変数エリアを割り当てて初期化するコードをアプリケーションプログラマに毎回記述させるのは繁雑である。このため、スタック以外のメモリ空間を完全に共有したタスクを生成する task_fork 機能を付加した。 MBCF の通信相手に関する情報もコピーして受け渡されるため、親タスクが MBCF に関する設定を済ませていれば、生成された子タスクが再度設定を行う必要がない。 UNIX の fork との差はメモリ空間がスタック以外完全に共有されているため、共有メモリを改めて張るまでもなく、タスク間の通信同期をメモリを介して行える。子タスクもタイムスライスによるスケジューリング対象となるため、アプリケーションプログラマが明示的にタスクのスケジューリングを行わなくても並行処理が記述できる。

2.3　RPC/MBCF の試作

効率の高いネットワークファイルシステム（NFS）を作る第一歩として、 *SSS–CORE* Ver.1.1 上において RPC （Remote Procedure Call）の試作[1]を行った。 *SSS–CORE* の持つオリジナル TCP/IP および UDP/IP は IP フラグメンテーションと TCP の eager 転送機能をサポートしていない簡易版であるため、 BSD 版の TCP/IP および UDP/IP を *SSS–CORE* に移植した。他のシステムとの NFS にはこの TCP/IP および UDP/IP による RPC を使用する。ただし、 *SSS–CORE* システム内では MBCF を利用した方が有利であるため、 MBCF の MBCF_Siganal および MBCF_Fifo 機能を利用した RPC/MBCF を試作した [11]。 RPC/MBCF と SunOS 上の通常の RPC （TCP/IP および UDP/IP 上）のラウンドトリップレイテンシの比較を表 1に示す。実験条件はワークステーションとして、 SPARCstation 20 (85 MHz SuperSPARC × 1) を使用し、 Sun microsystems Fast Ethernet SBus Adapter 2.0 と Bay Networks BayStack 350T (switching 100BASE-TX HUB) によってファーストイーサネット接続を行った。

表 1の結果から判るように、 *SSS–CORE* の RPC/MBCF の方が既存の SunOS の RPC よりも数倍高速である。特に転送データサイズが小さい時の差が顕著であり、プロセッサへのオーバヘッドコストが RPC/MBCF の方が大幅に小さいことを示している。

3　遠隔・並列実行環境

遠隔シェル機能および並列シェル機能として前述した *SSS–CORE* の遠隔・並列実行環境に関して、具体的なオペレータによる使用方法および実装方式を述べる。統一ビューの観点から UNIX 等の遠隔実行とは異なるセマンテックスを持ち、スケーラビリティを確保するために UNIX 等の遠隔実行とは異なる実装方式を採っており、技術的にも興味深いものとなっている。

SSS–CORE では MBCF によってクラスタ全体に跨る共有メモリ空間が構成され、プログラマはその共有メモリ空間を拡張された論理アドレス（論理タスク番号と論理アドレスの組）で自由にアクセスできる。このことはアプリケーションプログラマに論理的かつ統一的な視点を与えている。オペレータに対しても統一的な操作ビューを与えて、操作効率を上げることが望ましい。 Ver.1.1 以前においては、コンソール切替器を使って物理的にコンソールを切り替えたり、ノード毎に複数の

1　現在の正式版 Ver.1.2 には統合されていない。

telnet セッションを張ることにより、ノード単位でオペレーション（shell コマンド）を指定していた。この状況を改善するために作成されたのが遠隔・並列実行環境である。

```
msh 5 submit start.bat
```

上記は遠隔実行コマンドの一例である。ノード番号 5 番のノードで start.bat というバッチファイルを実行することを意味している。 submit コマンドはバッチファイルを起動するコマンドである。自分のノードでこのバッチファイルを起動する場合には以下の記述になる。

```
submit start.bat
```

この自ノードの実行において、 start.bat というバッチファイルはフルパスで指定されていないため、カレントディレクトリに存在している必要がある。このファイルが /share/tm10 というディレクトリにある場合には

```
cd /share/tm10
```

を前もって実行していることになる。 msh コマンドはオペレータの環境を遠隔実行するノードに輸出する機能があるため、自ノードでのカレントディレクトリは遠隔ノードでもカレントディレクトリとなる。

UNIX との対比で説明すると、 UNIX の rsh に対応するコマンドは msh である。 msh と rsh の機能的な違いは、すでに述べたように msh ではコマンドを入力している端末における環境を、遠隔ノードに輸出して遠隔実行を開始する点にある。カレントディレクトリや環境変数を設定し直すことなしに、手元と同じ環境で遠隔コマンド実行が可能になる。上記例で説明すると、すでにオペレータが /share/tm10 にいる場合は *SSS–CORE* においては、

```
msh 5 submit start.bat
```

で済むが、 UNIX においては

```
rsh node5 sh /share/tm10/start.bat
```

または、

```
rsh node5 "cd /share/tm10; sh start.bat"
```

というようにディレクトリも明示的に指定する必要がある。ここで node5 はノード番号 5 に相当するマシン名とする。

SSS–CORE は並列分散処理をワークステーションクラスタにおいて可能にするための汎用オペレーティングシステムであるため、複数のノードで同一のオペレーションを行いたい場合が多い。このため、同一引数で msh を複数ノードに適用することで並列に shell コマンドを実行する並列実行環境として psh を用意した。

```
psh 1 -- 6 8 10 : submit start.bat
```

このコマンド列はノード番号 1 から 6 と 8 番および 10 番のノードにおいて start.bat ファイルを実行することを意味している。

他にも、 MBCF を利用して効率良くユーザ定義コマンド（ユーザアプリケーション）を load する pload コマンド、およびユーザアプリケーションの並列実行時の MBCF 環境のセットアップを shell が行う pstart コマンド等が遠隔・並列実行環境用のシェルコマンドとして *SSS–CORE* Ver.1.2 から実装された。

msh,psh 等の実装上の特徴は以下のとおりである。

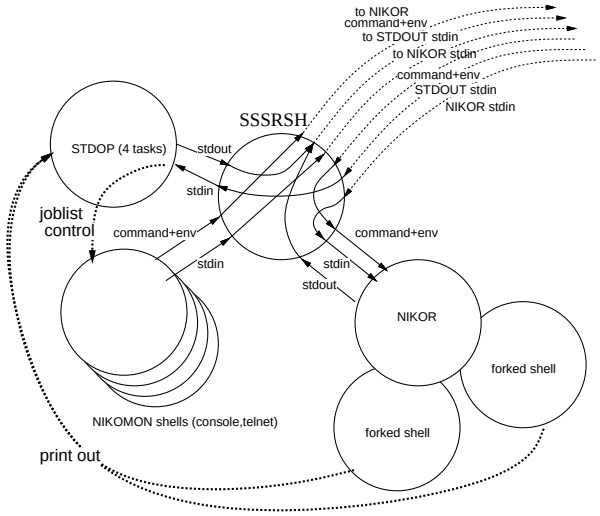

図 4 遠隔・並列実行環境の実装方式

1. msh,psh コマンド入力側のノードには新たなタスクを生成しない
2. background で実行が開始される
3. msh は MBCF 通信を行い、 msh コマンド毎に MBCF 通信セッションを確立する
4. MBCF をセキュアに行うためユーザレベルの Preemptive task である SSSRSH を介してノード間通信はすべて行われる

1 対 1 の遠隔実行である msh コマンドでは特徴の第 1 項目の存在意義はあまり大きくないが、同時に複数のノードでタスクを起動する psh や pstart コマンドではこの第 1 項目の実現は重要である。 UNIX の rsh では外部に実行を依頼したタスク（プロセス）の数だけノード内にもタスクを生成する。この UNIX 流の方式の方がタスク間の標準入出力のコネクション管理や通信のブロッキングに便利である。しかし、 *SSS-CORE* では、 1000 を越えるノードに同時にタスクを生成する並列実行（psh や pstart）を実現可能にしなければならない。 rsh 流の実装ではコマンド入力を行ったノードに多数のタスクが生成され、タスク管理のオーバヘッドが大きくなってしまう。

図 4に遠隔・並列実行環境に関わるタスク群を図示する。 msh コマンドの動作は以下のとおりである。

1. msh コマンドを受け取った shell はノード内の SSSRSH に環境やコマンドの内容を共有メモリを使って通知する（便宜上 MBCF を使っている）。
2. SSSRSH タスクはコマンドを検知すると、ターゲットノードの SSSRSH タスクとの間に MBCF セッションを張る（send_taskque を用いる）。
3. セッションが確立するとコマンドや環境を送りつけるべきアドレスがターゲットノードから返答される。また、セッション確立時にターゲットノードに標準出力の転送先のデータが登録される
4. 遠隔実行すべきコマンドおよび環境や並列実行の内容を SSSRSH タスクを経由して、ターゲットとなる実行ノードの SSSRSH タスクの指定アドレスに

送りつける。
5. ターゲットノードでは、 SSSRSH をスルーして（SSSRSH とノード内共有メモリを張っており、その共有領域にデータは送られてくる）遠隔実行用の親 shell タスクである NIKOR がデータを受け取る
6. NIKOR は子タスクを fork （厳密には shared_fork）して、受け取ったコマンドの実行を行う。この場合に子タスクは通常の shell として振舞い、_niko_ioBuf 構造体（shell 通信用文字列バッファやキー入力バッファを含む構造体）を一つ獲得する
7. この fork された shell (forked shell) が指定された遠隔実行コマンドの内容にしたがって、 shell コマンドまたはユーザアプリケーションの起動を行う。shell として指定されたコマンドを実行後に msh のためのノード間の通信セッションを解放する。新たなタスクを起動していた場合にはそれらのタスクの終了は待たずに、通信セッションを終了させる[2]。
8. forked shell およびそこから起動されたタスクは標準出力および標準入力を共有する。たとえば、printf 等を行うと、遠隔ノードの STDOPR → 遠隔ノードの SSSRSH →（要求ノードの SSSRSH をスルー） → 要求ノードの STDOPT → 要求ノードの STDOPC または STDOPN のシーケンスを辿って、 msh コマンド入力を行った端末画面に出力される
9. forked shell から実行されたタスクがすべてなくなると遠隔ノードで確保しておいた構造体（shell 関連と SSSRSH の接続情報）を解放する。要求元ノードには標準入力およびジョブコントロール関連の情報しか保持されていない。これらのデータ領域はタスクの実行終了時に MBCF によるメッセージが NIKOR から SSSRSH 経由で要求元の STDOPT に飛び、解放される

4 自由市場原理に基づくスケジューリング

本節では、 *SSS–CORE* の新しいスケジューリング方式として研究開発中の「自由市場原理に基づくスケジューリング」方式 [12]について述べる。

4.1 従来方式の問題点

SSS–CORE の目指す実行環境ではワークステーションクラスタの各ノードは従来通りの単体ワークステーションとしても使用される。 *SSS–CORE* 開発当初はノードを跨ったグローバルなカーネルレベルのスケジューラ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ワークステーションとしての使用によって大幅に変化するため、グローバ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可能性がある。

また、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たが、ワークステーション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。メモリ配置を考慮せずに、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によって性能が大幅に低下してしまう。

さらに、近年では入出力や通信に対するユーザレベルの裁量権が拡大し、入出力や通信をユーザレベルで最適化して高効率で行えるようになってきている。具体的に述べると、ノンブロッキングの入出力手段やノンブロッキングの通信手段がユーザに提供されるようになってきている。 MBCF はまさにそのような通信手段の一例である。これらのノンブロッキングの入出力操作手段を過剰に使用すると、周辺装置やネットワークの性能を飽和させて、システム全体の性能を大幅に低下させかねない。従来の周辺装置の入出力要求やネットワークへのデータ送受信要求はブロッキング操作であったため、一つのアプリケーション（タスク）が同時に複数の要求を出すことにより、システムの能力を飽和させてしまうことはなかった。最近になってノンブロッキング操作が提供されるようになってきているが、多くのオペレーティングシステムにおいて資源のスケジューリング自体は昔ながらのプロセッサ使用時間に基づくプロセッサの時分割による割り当てである。このため、ネットワークの性能が飽和している場合には、通信を必要としないタスクを優先的にスケジューリングする等の考慮がなされるべきであるにもかかわらず、なされていない。つまり、ノンブロッキングの入出力操作や通信操作が提供されている環境（オペレーティングシステム）では、従来型の時分割タスクスケジューリング方式では力不足である。

4.2 中央集権か自由主義か

本節では、分散並列処理環境におけるスケジューリングを議論している。負荷分散をタスクの実行単位よりも細かく実現するために、タスクのマイグレーションが可能であると仮定する。

どのノードでタスクを実行するかの判断を中央集権的に行う方式とユーザもしくはアプリケーションタスク自体が走行するノードを選択する方式が考えられる。さらに、ノードの中のスケジューリングを他のノードと同期して（連携して）行う方式と独立して行う方式が考えられる。言い替えると、実行場所の制御を中央制御するか、自由放任で行うかの区別と、実行時刻の制御を中央制御するか、ノード毎に任せる（地方分権）かの区別である。この類別によると 4 種類のスケジューリング方式が考えられる。ここでは、両極端にある二つのケースについて議論する。より中央制御の徹底した実行場所も実行時刻も中央制御する方式を**中央集権方式**と呼び、よりユーザの自由度が高い実行場所はユーザが選択し、実行時刻はノードのスケジューリングに任せる方式を**自由主義方式**と呼ぶ。なお、実行時刻をノード内の独立したスケジューリングに任せたとしても、自分の都合に合わない時刻にプロセッサを割り当てられた場合に、プロ

2 現実装ではこの遠隔コマンド起動セッションが終了しないと要求元ノードもターゲットノードも次の起動セッションを実行できない。

セッサの使用権を放棄する自由／手段があるものと仮定する。プロセッサ割り当て後すぐに使用権を放棄した場合に、実行優先度が下げられないで ready キューの先頭（もしくは先頭付近）に戻されるような仕組みがあれば、実行時刻をタスク自体が制御することが可能である。中央集権方式はグローバルなスケジューラを持ち、時分割のギャングスケジューリングが可能である。自由主義方式はノード毎のスケジューラのみを持ち、ノードの選択はユーザまたはアプリケーション任せである。したがって、中央集権方式ではグローバルスケジューラがマイグレーションの実行の決断およびマイグレーション先の選定を行い、自由主義方式ではマイグレーションの実行の決断およびマイグレーション先の選定をアプリケーションタスク自体が行う。後者の場合、タスクの移送行為自体はシステムコール等でオペレーティングシステムに委託される可能性がある。

この二つの方式の特徴をまとめると以下のようになる。

- 中央集権方式の特徴
  1. グローバルかつ詳細な負荷分散が可能
  2. グローバルな最適化スケジューラが必要
  3. ギャングスケジューリングの実現が容易
  4. グローバルスケジューラが公平性を考慮
  5. プロセッサプール方式に適合
- 自由主義方式の特徴
  1. 自己責任による自律的負荷分散
  2. 自律最適化、ノードスケジューラの構成が単純
  3. ギャングスケジューリングは疑似的な実現
  4. ノードスケジューラが公平性を考慮する必要
  5. 個別ワークステーション的使用と両立

分散並列処理環境が均質かつ平坦な構成でない場合には、協調処理もしくは並列処理ではタスクの環境内の配置やタスクのスケジューリング順序が実行効率に影響する。中央集権方式の場合は、グローバルスケジューラが細かい資源割り当てまで決定するため、ユーザアプリケーションの都合を反映するためには何らかのインタフェースが必要である。つまり、グローバルスケジューラにユーザ側から制約やヒントを提示して、それらを満たすようにスケジューリングしてもらう必要がある。制約やヒントが複雑になると、複数の協調並列タスクの制約やヒントを準最適に満足するスケジューリングを求めることさえ困難になる。簡単なスケジューリング制約しか指定できないと協調並列タスクの実行最適化の機会を減らしてしまう。これに対して、自由主義方式ではアプリケーションタスク自体が資源割り当ての方向性（どのノードで実行するか、割り当てられたタイムスロットを受け入れるか）を決断するため、プログラム次第で自由な実行時最適化が可能である。しかし、逆に言えば「最適化」のつもりで行った行動が実行時間を大きくする方向に働いたとしても他人（スケジューラ）の責任にはできない。システムを作る立場からは、複雑な資源制約を解決するグローバルな最適化スケジューラを実装する必要のない自由主義方式の方が圧倒的に簡単である。

次に、中央集権方式のメリットと考えられるギャングスケジューリング実現の容易さについて議論する。ノンブロッキング通信操作によって複数のタスクが通信している場合は、タスク間の厳密な同期は不要である。もちろん、タイムスライス（tick）オーダの実行のずれが許容できるかどうかは協調タスクの性質による。しかし、前述のように自由主義方式と言えども、実行時刻を調整することが不可能であるわけではない。ギャングスケジューリングが必要なタスクがノードに複数割り当てられた場合は、自然にタスクがノードに割り当てられるタイミングが揃うようにノードスケジューラを構成することは可能である[3]。自由主義方式の方がギャングスケジューリングと同等の動作を行うためには少し余計にコストがかかる可能性があるが、ギャングスケジューリング自体の有効性はまだ明らかではなく、ギャングスケジューリング実現のためのコスト差による効果は今後の検討課題である。

プロセッサプールは無名のプロセッサ資源の集合を指し、個々のプロセッサ資源を特定してタスクの実行が行われることはない。グローバルスケジューラがプロセッサプールから負荷や各種制約を考慮して適したプロセッサ資源を選んでタスクに割り当てる。プロセッサプールというのはバックエンドの計算資源という位置付けである。これに対して、分散並列計算機環境がワークステーションクラスタや PC クラスタである場合は、個々のマシンをワークベンチとして使用したいと言うケースも多い。この場合、ディスプレイカードや音声合成デバイスといったマシン依存のデバイスが大きな役割を果たす。このため、特定のマシンで動く必要のあるタスクや特定のマシンで動いた方が圧倒的に有利なタスクが存在する。また、従来オペレーティングシステムにおける使い方では、単一実行アプリケーションはユーザがマシンを指定して起動することが一般的である。このことから、ワークステーションクラスタや PC クラスタではプロセッサプールによるタスクスケジューリングのみを行うことは望ましくない。各ノードに対して個別ワークステーションとしての使用を認める場合には、そのノードを特定して投入したタスクによって、ノードの負荷が大幅に変動してしまう。グローバルスケジューラがこの変動を事前に予期することは不可能であるため、事後的に負荷調整を行うと調整のためのオーバヘッドが発生する。また、この負荷変動のために、それ以前に行ったマイグレーション等の負荷均衡のための動作が裏目になるかもしれない。もちろん、自由主義方式でも同じことが言えるが、この場合は元々自己責任に基づいているので、予期しない負荷変動による不利益も自己責任の範囲内である。

中央集権方式は全知全能のグローバルスケジューラがスケジューリングの公平性に関する考慮を行う。これに対して、自由主義方式ではノードレベルのスケジューラしか存在しないため、ノードレベルのスケジューラにおいて公平性を実現する必要がある。プロセッサの割り当て頻度を調整することにより公平性を実現するため、

3 すべてのノードで同じ大きさの tick を採用し、自律的なタイムシフトが動的に可能な機構（システムコール等）を用意する。

ノードレベルでも十分に実現可能である。自由主義方式に関する公平性の議論は次節で行う。

結局、中央集権方式を採ろうが自由主義方式を採ろうが実現できる機能には差がない。しかし、自由主義方式はスケジューラの実装が容易であり、アプリケーションから見てインフラストラクチャであるスケジューラを変更することなしにユーザレベルの最適化を強化できる可能性がある。もちろん、逆にユーザレベルの実行時最適化の戦略が幼稚であれば、タスクの実行効率を低下させる危険もある。しかし、ランタイムライブラリやコンパイラ生成コードとして実行時最適化ルーチンが提供されるのであれば、この危険は幼稚な戦略しか採れないグローバルスケジューラを実装してしまう危険と変わらない。このため、自由度が高く、システムの実装が楽な自由主義方式を我々は採用する。

4.3 自由市場原理に基づくスケジューリング方式

前節において望ましいスケジューリング方式に対して「自由主義」方式という言葉を使用したが、理想の市場主義経済体制がそうであるように、決して無政府状態の自由放任方式を意味しているわけではない。ある種の公平性は保証しなくてはならず、自己責任でスケジューリングの判断を下すためには判断を下すのに十分な情報が低コストで入手可能でなくてはならない。

必要性が明白な低コストの情報入手について先に補足する。実行時に動的に負荷分散を自らの判断で行うためには、自分のノードの負荷情報のみではなく、他のノードの負荷情報も必要である。また、システムが非均質であれば、マイグレーション可能なノードを知るために各ノードのプロセッサの種類といった情報も必要になる。これらの情報を獲得するために、各タスクが毎回他ノードに通信を行うようではコストが大き過ぎる。他ノードも含めた資源の使用状況や割り当て情報がユーザアプリケーションから低コストでアクセスできる情報開示機構が必要である。この情報開示機構には *SSS–CORE* で提案した情報開示機構 [3]がそのまま使用可能である。

全体の利益を無視してひたすら利己的に振舞うタスク（結果的にはシステム全体の性能を低下させ自分にも不利益をもたらしている）がシステム全体の利益を考慮して自律的に外部入出力やメモリの使用量を抑制しているタスクに迷惑をかけるようでは良いシステムとは言えない。具体例を挙げると、 ethernet 等の通信網では競合がある程度以上になると通信の成功率が大幅に低下して、意識的に通信を控えた方が通信の全体性能が向上する。ユーザレベルの高性能通信手段を利用するような性能重視のアプリケーションでは、常に最高性能が出せるようにプログラムを構成したい。そこで、ネットワークの使用状況を調べて、ネットワークがある程度以上の競合状態にある場合には、通信頻度を自分で抑える（TCP/IP 実装などで見られる輻輳回避をユーザレベルで行う）という最適化が考えられる。この自律的最適化を施したタスクとそうでないタスク（こちらもネットワークを高頻度で使用する）が同時に同じノードにスケジューリングされた場合に、ネットワークの競合状況を見て自律的最適化を行うタスクのみが通信（実行）を遠慮して、最適化を行わない「厚かましい」タスクが優遇されて実行される結果になってしまうようでは「公平な」システムとは呼べない。つまり、システム全体の性能低下の防止ならびにアプリケーション間の公平性堅持のための基本ルールを設ける必要がある。

「低コストでアクセスできる情報開示機構」を持ち、「公平性を守るための基本ルールを確立」した自由主義スケジューリング方式を**自由市場原理に基づくスケジューリング方式**（FMM 方式: Free Market Mechanism 方式）と呼ぶ。

FMM 方式を汎用スケーラブルオペレーティングシステム *SSS–CORE* の次世代スケジューリング方式として採用し、現在実装中である。並列もしくは協調実行中のタスクグループの一部もしくは全体が自由にマイグレーションできるシステムを構築する予定である。

5 おわりに

分散サーバ環境の核となるスケーラブルオペレーティングシステムの研究開発は「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」に引き続いて行われており、オペレータやアプリケーションプログラマを支援する機能の拡充を中心に順調な進展を見せた。また、「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」で積み残した非対称分散共有メモリ (ADSM) 機構を実装し、分散共有メモリ実現機能が大幅に強化された。最新鋭プロセッサを使ったワークステーションへ *SSS–CORE* を移植する作業は、プロセッサのカーネルアーキテクチャが大幅に変更されているため、当初からの予想通り困難な作業であった。しかし、渦原の努力により、SBus 仕様の Ultra ワークステーションに関しては、 *SSS–CORE* のノード常駐部分である SSS–MC の移植を終了しており、 *SSS–CORE* Ver.2.1 が動作を開始している。さらに、 *SSS–CORE* のマンマシンインタフェースを改善するために、 UNIX システムで標準の GUI システムである X ウィンドウシステムを現在移植中である。研究開発とは直接的な関係はないが、 *SSS–CORE* の知名度アップのためマスコットキャラクタ「マネキッコ」（図 5参照）を作成した。今後 *SSS–CORE* 関連プロジェクトの宣伝に活躍してもらう予定である。

MBCF の *SSS–CORE* 以外のシステムへの移植は、MBCF がメモリ管理と密に協調しているため、メモリ管理関連の詳細な資料と開発環境が入手できないオペレーティングシステムでは移植が困難である。このため、ソースコードが入手可能で広がりを見せつつある Linux の UltraSPARC 版を対象に移植作業を開始した。作業が困難であるため、現在まだ移植作業中である。この移植が終了後には、 PC 互換機上の Linux を対象にして MBCF を移植の可能性を調査し、移植作業を行う予定である。

図 5　*SSS–CORE* **マスコットキャラクタ「マネキッコ」**

謝　辞

ワークステーションに使用されている 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。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。

References

[1] 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, 第 1 分冊, pp.37–38 (September 1996).

[2] 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).

[3] 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).

[4] 松本 尚, 渦原 茂, 竹岡 尚三, 平木 敬: 汎用超並列オペレーティングシステムカーネル *SSS–CORE*. 第 17 回技術発表会論文集, 情報処理振興事業協会, pp.175–188 (October 1998).

[5] 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).

[6] 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).

[7] 丹羽 純平, 稲垣 達氏, 松本 尚, 平木 敬: 非対称分散共有メモリ上におけるコンパイル技法. ハイパフォーマンスコンピューティング研究会報告 No.67–21, 情報処理学会. pp.121–126 (August 1997).

[8] 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).

[9] 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).

[10] 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).

[11] 亀沢 寛之, 松本 尚, 平木 敬: メモリベース通信を用いた RPC の実装, システムソフトウェアとオペレーティング・システム研究会報告, 情報処理学会, Vol.98 No.71 OS–79–2, pp.9–16 (August 1998).

[12] 松本 尚, 平木 敬: 自由市場原理に基づくスケジューリング方式. 信技報, Vol.99, No.251, CPSY 99–55, pp.63–70 (August 1999).