汎用超並列オペレーティングシステムカーネル SSS–CORE — 基本方式および NOW 版の実装について —

Architecture and implementation of the general-purpose massively-parallel operating system: SSS–CORE

松本 尚 [*1]
tm@is.s.u-tokyo.ac.jp

信国 陽二郎 [*2]
nobukuni@is.s.u-tokyo.ac.jp

駒嵐 丈人 [*3]
koma@axe-inc.co.jp

渦原 茂 [*4]
uzu@axe-inc.co.jp

竹岡 尚三 [*5]
take@axe-inc.co.jp

平木 敬 [*6]
hiraki@is.s.u-tokyo.ac.jp

[*1,*2,*6] 東京大学 大学院理学系研究科 情報科学専攻
[*3,*4,*5] 有限会社 アックス

SSS–CORE はワークステーションクラスタおよび分散メモリ型並列計算機を対象とした汎用超並列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ユーザ / マルチジョブ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並列計算の効率を落さないための各種機構機能を実現する。本稿では SSS–CORE の構成と特徴を解説し、現在開発実装中のワークステーションクラスタ版 SSS–CORE の実装方式ならびに開発状況について述べる。最後に、カーネルレベルスケジューリング方式を決定するために行ったシミュレーション実験ならびに開発中の実機の SSS–CORE 上で行われた簡単なメモリベース通信に関する初期評価実験の結果を示す。

1 はじめに

マイクロエレクトロニクス技術の急速な進展に伴い、ワークステーションを高速ネットワークで結合したワークステーションクラスタ (Network of Workstations: NOW) や多数の CPU を搭載した分散メモリ型並列計算機が容易に実現可能になり、従来メインフレーム、すなわち大型 / 超大型計算機システムだけが実現可能であった応用分野に対しても適用の可能性が高まってきた。特に、近年のコンパイラ等言語処理系による相互結合網の遅延隠蔽、プロセスやデータの最適分割方式、スケジューリング方式、同期最適化方式の導入により、幅広い応用分野に対して分散メモリ型の並列計算環境を効率良く利用することが可能となりつつある。しかしながら、汎用性 (マルチユーザ・マルチジョブ) 導入による保護機能の実現や実資源の仮想化は並列処理性能の大幅な低下の原因となるため、現状の並列計算環境および並列計算機は基本的にバックエンドプロセッサとしてバッチ処理で使用されることが多い。アプリケーションに関して言えば、バッチ処理で支障がない科学技術計算のみが並列処理され、他のアプリケーションは実行単位ごとに個別にプロセッサの割り当てを受けて、ほとんど非同期に分散処理をしているに過ぎない。その原因は、先にも述べたように保護や仮想化を汎用性実現のために導入することが並列処理に大幅な性能低下をもたらすことに起因する。言い替えると、ソフトウェアによる様々な最適化方式の適用および軽い通信同期手法がシステムの独占的使用を要求し、システムの柔軟かつ汎用的な使用形態と相容れなかったからである。

我々は以前よりすべての CPU から等距離でアクセス可能な共有メモリを持つ Uniform Memory Access (UMA) 型並列計算機を対象にして、ユーザによる最適化を支援する汎用オペレーティングシステム核 SS–CORE [1]を研究開発してきた。SS–CORE の研究においては議論を単純化するために、UMA 型並列計算機に対象を限定し、実プロセッサの資源管理に的を絞った研究を行った。しかし、技術の進歩は目覚しく、UMA 型並列計算機よりも多くのプロセッサを接続可能であり、構成のスケーラビリティが高い NUMA (Non-Uniform Memory Access) 型並列計算機[1]が現在実用化されつつある。従って、汎用性を持った使用形態においても NUMA 型並列計算機を高い効率で使用するためのキーとなるオペレーティングシステム核 **SSS–CORE** (**Scalable SS–CORE**) [2]の確立が非常に重要となっている。

2 UMA 型並列計算機用 OS 核 SS–CORE

SSS–CORE に関して述べる前に、SSS–CORE のベースとなった SS–CORE に関する研究について簡単に紹介

† 本研究は情報処理振興事業協会「独創的情報技術育成事業」の一環として行われたものである。

1 ここではワークステーションクラスタ、分散メモリ型並列計算機、分散共有メモリ型並列計算機等の総称

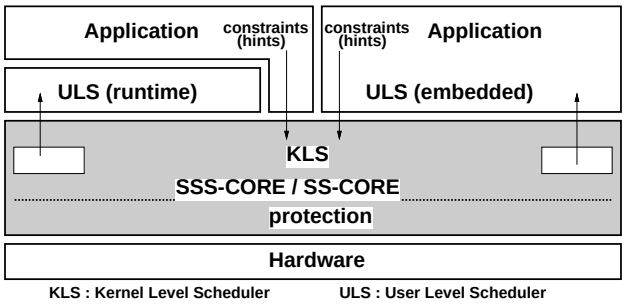

Figure 1 SS–CORE / SSS–CORE の資源管理構造

する。 UMA 型並列計算機用の OS 核である SS–CORE はユーザによる並列実行の最適化のために、

- 「時分割されたパーティショニングによるマルチジョブ」
- 「資源のグループ単位管理」
- 「適切な保護の下でユーザへ資源を最大限に解放」
- 「カーネルレベルスケジューリング情報をユーザへ開示」
- 「カーネルレベルスケジューリングへのプログラムの静的情報に基づいたユーザ制約 / ヒントの採用」

といった基本方針 [3] [4]を持つ (図 1参照) 。この基本方針に基づいた研究の結果、プロセッサ資源管理に関して、世界に先駆けて以下のような方式 / 機構を考案した。

- カーネル / ユーザの資源割当状況や実行状況によるスケジューリングオプションつきのスピンウェイト: Snoopy Spin Wait (SS-wait[2]) [5]
- カーネルにプリエンプトされたプロセッサコンテクストのユーザレベルの復旧 [5]
- 実プロセッサグループをユーザ (プロセス) に割り当て、グループ内のプロセッサスケジューリングはユーザに一任するユーザ / カーネルの二階層スケジューリング方式 [5]
- 並列プログラムの性質による分類とそれを利用したカーネルレベルスケジューリング [1]

SSS–CORE の資源管理方針は SS–CORE の基本方針を受け継ぎ、 SS–CORE の研究において考案された機構 / 方式に関しては対象計算機システムを NUMA 型に変更したことに伴う拡張を行っている。

3 メモリベース通信機能

SS–CORE ではどの CPU からも等距離でアクセス可能な共有メモリを対象計算機側に仮定していたので、ユーザレベルの通信同期はこの共有メモリを介して行うことが可能である。また、メモリにはメモリ保護, 論理 / 物理のアドレス変換, 仮想記憶サポートがあるために、 UMA 型並列計算機における共有メモリを介した通信同期はオペレーティングシステムの直接的な仲介なしに保護と仮想化を実現できる。さらに、ハードウェアで実現された対称型共有メモリのアクセス速度は速いため、ユーザレベルの軽い仮想化され保護された通信同期が共有メモリを用いるだけで実現できる。

SSS–CORE では対称型共有メモリを持つ UMA 型並列計算機の高実現コストとスケーラビリティの欠如を理由に、対象計算機システムをより一般的な NUMA 型並列計算機に設定している。さらに、 SSS–CORE では通信同期用の特別なハードウェアを仮定しない方針を立てており、現在実装中の NOW 版 SSS–CORE では市販の量産ワークステーションを LAN 用のネットワークで接続したシステムを対象としている。このため、ハードウェアで実現された対称型共有メモリや Memory Based Processor (MBP) [6]のような専用ハードウェアを通信同期に対して使うことはできず、軽いユーザレベルの保護され仮想化された通信同期を実現する方法が大きな問題となる。この問題に対して、 SSS–CORE ではソフトウェアによるメモリベース通信 [7]で解決を図っている。全体像の理解を容易にするため、従来のユーザレベルの通信同期を行う際に問題となっていた点と、これらの問題点を解決するためのメモリベース通信の技術的な要点を以下に述べる。メモリベース通信の詳細方式に関しては別稿にて記述する予定である。

市販のワークステーション等をノードに使用することを仮定すると、通信用インターフェースへの低レベルアクセスは保護の観点からカーネルモードで行う必要がある。また、受信インタフェースは通常受信バッファメモリに通信内容を受け取る構成となっているが、受信バッファメモリの領域は予め設定しておく必要があり、容量は有限である。このため、受信割り込みハンドラを用意して、割り込みベースで適宜受信内容を主記憶上に退避する必要がある。このような計算機環境において従来型オペレーティングシステムの上でメッセージパッシング方式の通信を実装した際の問題点は以下の通りである。

- 高オーバヘッド
  - ユーザ / カーネル切替コスト
  - コンテクスト切替コスト
  - キュー操作オーバヘッド
  - バッファ操作 (コピー等) オーバヘッド
  - ノード間コネクション管理オーバヘッド
- マルチキャスト通信のオーバヘッド
- Acknowledge(Ack) 回収のオーバヘッド
- 遠隔実行による不公平性
- 遠隔実行によるキャッシュ / TLB ポリューション

問題点が認識されれば解決策は簡単であり、それは共有メモリプログラミングモデルに基づく高機能遠隔メモリアクセス、つまりメモリベース通信 (同期) 機能である。 SSS–CORE におけるメモリベース通信機能とは MBP と共に提案された高機能分散共有メモリシステム (SMS: Strategic Memory System) [6]上の各種メモリベース通信機能やメモリベース同期機能 [8]を、 NOW や分散メモリ型並列計算機に可能な限り低オーバヘッドのソフトウェアにより実装したものに他ならない。前記の問題点を解決する要点を簡単に説明する。

1. 単にユーザ / カーネルのモード切替トラップや外部割込ハンドラへの反応時間だけで見れば、高性能マイクロプロセッサのオーバヘッドは数クロック ~ 数

2 SS–CORE の名前の由来となる同期方式

十クロック程度である。従来型のオペレーティングシステムでは余分なチェックや余分な再スケジューリングでコストを大幅に増大させている。そこで付随するオーバヘッドを最小限に抑えるために以下のような実装を行う。ユーザレベルで使用するノード間通信同期用システムコールや割込ハンドラは、 OS の他の I/O 関係のシステムコールや割込ハンドラと分離して実装し、通信同期にとって余分なチェック等を全廃し、カーネル権限で実行されるコードおよびユーザ / カーネル切替に際して退避復旧するプロセッサコンテクストを最小限にする。この通信同期用システムコールや割込ハンドラではキャッシュフラッシュや TLB フラッシュを行わない。

2. OS のプロテクションの下でバッファのコピー回数やソフトウェアオーバヘッドを抑える最良の方法は、メモリ管理機構と協調動作する遠隔メモリアクセス用のハードウェア (つまり MBP のような物) を使用することである。この遠隔メモリアクセス用ハードウェアを仮定しない場合は、通信パケット内に通信相手先のアドレス空間内の目的アドレスを格納しておき、 1. の実装方針に従って作られたメッセージ受信用の割込ハンドラが直接相手先のユーザ空間内のメモリ領域に通信パケット内のデータを (受信バッファ領域から) 格納する。プロテクションはパケットの発信元と受信先の少なくともどちらか一方でケイパビリティチェックを行うことで実現できる。 SSS–CORE の現在の実装ではパスワード (32bit 数値) を利用したケイパビリティチェックを受信先の割込ハンドラで行っている。
3. 動的な通信はコネクションも動的に接続・解除されるので、多数のコネクションを同時に保持管理するためのコストの問題はあまり生じない。しかし、静的 (コンパイル時) に、通信相手や送信先アドレスやフェッチデータのアドレスが判る場合には、通信相手や転送先アドレスといったコネクションに関する静的情報を利用した方が、動的なコネクションを張るオーバヘッドが回避できるので効率が良い。並列実行に非常に適したデータパラレル (SPMD) の数値計算アプリケーションでは、論理アドレス空間を各プロセッサで共通 (ただしメモリの実体は別) にすることが非常に有効である。論理アドレス空間を共通にすることにより、遠隔メモリアクセスのためのコネクション情報はプログラムコード内に操作対象の論理アドレス (と静的に判明する論理プロセッサ) として格納可能になり、実行時には論理プロセッサと物理プロセッサ (物理ノード) の対応を取るだけで通信同期が可能である。通信同期要求パケットの受信割込ハンドラが対象論理プロセッサの論理アドレスを解決[3]してメモリ操作を行う。
4. MBP による SMS の大きな特徴に階層マルチキャストと Ack コンバイニングによる update ベースのキャッシュの実現がある。この update ベースのキャッシュシステムは上記の SPMD プログラムの例と同様に論理アドレス空間を各ノードで共通に取り、論理アドレスが同じキャッシュ用のページを各ノードに用意して、階層マルチキャストや Ack コンバイニングをパケット受信割込ハンドラでエミュレートすることで実現できる。基本的に大規模なマルチキャストは階層構造を利用して行われ、 Ack のコンバイニングも階層構造を利用しない限り効率化できない。このため、 update キャッシュの実現方式は NOW であれ、分散メモリ並列計算機であれ、分散共有メモリ並列計算機であれ、基本方式に変わりはない。そこで、 NOW においてはネットワーク内に階層的な経路を静的に見い出して、それを利用して実現される。
5. ナイーブなメッセージ駆動や要求駆動などのパケット到着駆動による実行形態を基本実行形態として選択するには現在の高性能マルチプロセッサは処理の空間的時間的局所性を期待しすぎている。また、局所性の利用は、コンピュータの高速化にとって最も重要な事項であるから、不可避である。しかし、処理したいデータセットの大部分が他のノードにあるような場合は逆に通信によってコントロールを移動した方が局所性が抽出できる。この場合においても、現在実行中のジョブから CPU を奪って実行したのでは、現在実行中のジョブの局所性の利用を妨げ、スケジューリングの公平性からも望ましくない。このため SSS–CORE では、 SMS の Memory-Based Signal と同様に、ユーザレベルのランキューにパケットとして運ばれてきた実行スレッドを直接格納することで、キャッシュポリューションや不公平性を回避する。このユーザレベルランキューの所有者であるジョブが実プロセッサにスケジューリングされている時しかこのキューからスレッドが起動されることはない。このため関係のないジョブの CPU 時間の消費はキューへの登録操作 (データ量が多い場合は実際のデータ移動は DMA 転送) のみであり最小限である。さらに、このスレッド起動はアドレス空間がすでに切り替わっており、ユーザレベルで実現されるので、極めて低コストである。

また、遠隔メモリリードや各種メモリベース同期機能は要求元からの遠隔メモリアクセス要求パケットの送信、受信時に割込ハンドラ内で目的クラスタ (ノード) 内の目的タスクの対象アドレスの操作 (同期処理の種類によっては不可分操作) および要求元への返り値 (返り値) の返送、要求元での受信割込ハンドラによる返り値の返り値アドレスへの書き込み (Ack の場合はカウントアップやカウントダウン) で実現される。

ノード間通信同期に関わる低レベルの各種同期情報はすべてユーザレベルのフラグに反映され、このフラグを利用して SS-wait によって同期が行われる。遠隔メモリアクセスの順序モデルには緩和されたメモリアクセスモデルを利用することで性能を向上させる。遠隔メモリアクセスの順序管理は Ack ベースのメモリバリアで

3 解決と言っても通常 TLB を多くとも 1 本セットするだけである。

行われるが、可能な限り Ack を省略して済ませるように実装を行う。また、実装上の問題としては、 Ether や FastEther のようにデバイスレベルでパケットが消失する可能性がある通信ハードウェアに対応するために、パケットに送信ノード毎にシリアルナンバーをつけて管理している。

4 非対称分散共有メモリ

SSS–CORE では、前述のメモリベース通信機能を使って、新しいソフトウェア分散共有メモリ (ソフトウェア DSM) の枠組である非対称分散共有メモリ (Asymmetric Distributed Shared Memory: ADSM) [9]をユーザに提供する。遠隔メモリ書き込みを伴う可能性がある場合に、従来のソフトウェア DSM [10]は単なる store 命令をコンパイラが用意し、ページの書き込みトラップ時にコンシステンシ維持のためのコードを実行していたのに対して、 ADSM のためのコンパイラは書き込み対象のデータの従うべき分散共有メモリプロトコルに適合した一連の命令 (コンシステンシ維持コードを含む) を実行コードとして埋め込む。遠隔メモリ読み出しの可能性に関しては、従来のソフトウェア DSM と同様に、 load 命令を用意しページトラップでキャッシュミスを検出して対処する。遠隔メモリの読み出しと書き込みの取り扱いが異なる点が ADSM の最大の特徴であり、名前の由来となっている。この遠隔メモリ書き込みのための store 命令を一連の命令列で置き換えてしまうことにより、 ADSM の自由度は非常に向上しており、様々なメモリモデルやコンシステンシプロトコル (更新型を含む) をサポート可能であり、通信量および通信回数を削減するための通信の動的コンバイニングもコードとして埋め込むことができる。さまざまなプロトコルが実現可能なことから、 SSS–CORE 上の ADSM ではページ単位のプロトコル切替 [11]による最適化をサポートする。

上記から判るように、 ADSM はコンパイラによる最適化を前提としたソフトウェア DSM である。 Release Consistency (RC) モデル [12]に基づき false sharing 問題を緩和するハードウェア DSM およびソフトウェア DSM も共有書き込みが単一の store で済むと言っても、 RC モデルに基づいたコードを生成する特殊な最適化コンパイラが必要であることに変わりはない。また、更新型および無効化型の双方をサポートするハードウェア DSM [6]は理想ではあるが、多くのコンシステンシ管理用のハードウェアを必要とするため、 NOW 環境では実現不可能である。

5 プログラミングモデル

低レベルのユーザ用通信同期ライブラリとして提供されるメモリベース通信機能を明示的にユーザが直接使ってプログラムしても構わない。しかし、基本的に最適化コンパイラによって生成されたユーザオブジェクトコードが実行されることを念頭に SSS–CORE は設計されている。 SSS–CORE 上の ADSM に基づくユーザのプログラミングモデルを述べる。 SSS–CORE では並列実行コードの生成が比較的容易な以下の二つのプログラミングモデルを推奨モデルとする。

並列性の抽出が容易な (もしくは並列性を明示する) HPF のような SPMD タイプを推奨プログラミングモデルの一つとして採用する。このプログラミングモデルでは大域変数の宣言に特徴があり、大域変数ごとに ADSM の各種プロトコルとホームノード (論理プロセッサ番号で指定) を宣言する[4]。最適化コンパイラは同一ノードの同一プロトコルの変数をページ単位でまとめ、大域変数への書き込みではプロトコルとノードに対応したコードシーケンスを生成する[5]。また、静的に読み切れる場合は冗長なノード間通信の削除やコンバイニングを行う。また、実行時の動的なコンバイニングコードをユーザレベルで挿入する[6]。

SSS–CORE は NUMA 環境 (NOW 版 SSS–CORE では、 ADSM により NOW を NUMA として扱う) に UMA 型のクラスタ (例えば、密結合マルチプロセッサ型のワークステーション) が含まれることを許している。 SSS–CORE が推奨するもう一つのプログラミングモデルはクラスタ単位のマルチスレッドプログラミングとクラスタに跨る大域配列、同期変数、通信変数による共有メモリアクセスを許すプログラミングモデルである。このモデルを SSS–CORE では Multi-Thread Multi-Task (**MTMT**) モデルと呼ぶ。 UMA 部分がない場合は単一プロセッサと付属のメモリをクラスタと見做して、その内部でマルチスレッディングがなされる。 SPMD タイプとは異なり、個々のプロセッサにスケジュールされるべきスレッドごとにプログラムが記述されている。スレッド間のスケジュールは言語によって提供されるランタイムライブラリやマクロ関数によって行われる他、特別なスレッドであるユーザレベルスケジューラをユーザが書くこともできる。大域共有変数は SPMD モデルと同様にプロトコルの種類とホームの論理位置を宣言する。この MTMT モデルは従来の実行モデルに対応しており、クラスタ内はプロセッサ台数に関わらず完全に同一アドレス空間が実現されており、ユーザレベルスケジューリングによってスレッドのスケジューリングがなされる。ただし、ユーザレベルでクラスタを跨いで他のクラスタにスレッドの実行を移動させることはできない。 (OS によってクラスタ単位でマイグレーションされる可能性はある。)

SSS–CORE には以下の属性を持つメモリ領域 (ページ単位) を持つ[7]。ただし、 ADSM としての機能を持つページの書き込みは invalidate ページを除いて、前述のメモリベース通信機能を用いた一連のコードシーケンスで達成される。

4 宣言がないと言語のデフォルトのプロトコルが用いられ、デフォルトの領域分散ルールでホームノードが決まる。

5 間接参照やポインタで静的にホームノードが不明の場合は論理アドレスからホームノードを解析するコードも書き込みシーケンス内に埋め込む。

6 現在のワークステーションでは細粒度通信に耐える通信能力はない。 OS レベルでもコンバイニングのサポートは行う予定。

7 ADSM では書き込み時のユーザコードによって列挙した以外の自由なプロトコルを実現することができる。

local ページ　SPMD モデルにおいてこの属性がデフォルトであり、クラスタ外、クラスタ内を問わず、論理プロセッサごとに同一の論理アドレスに対して別の物理領域が割り当てられ、コンシステンシの維持はなされない。MTMT モデルでは存在しない。

c_local ページ　このページ上の変数は同一クラスタ内でのみ、同じ物理領域を指す。MTMT モデルにおけるローカル変数はすべてこのクラスである。

home_only ページ　読み書きを問わず遠隔メモリアクセスを要求し、コピーを他ノードに生成しない領域。ページの一部しか利用しないことが判っている時には有利。

invalidate ページ　ADSM の一種。ホーム以外に同一論理アドレスのコピーページを生成する。ページ単位の無効化によって Sequencial Consistency (SC) でコンシステンシ管理がなされる。

ulrc ページ　ADSM の一種でユーザコード支援による Lazy Release Consistency (LRC) [13]を実現するページ。ホーム以外に同一論理アドレスのコピーページを生成する。LRC と同様なページ単位無効化によりコンシステンシ管理がなされる。

home_update ページ　ADSM の一種で Automatic Update Release Consistency (AURC) [14]を専用ハードウェア不用で実現するページ。ホーム以外に同一論理アドレスのコピーページを生成する。書き込みはホームページへの遠隔メモリ書き込みとローカルコピーへの書き込みを行い、AURC と同様な手法によりコンシステンシ管理がなされる。

update ページ　ADSM の一種でホーム経由の update プロトコルを専用ハードウェア不用で実現するページ。ホーム以外に同一論理アドレスのコピーページを生成する。メモリベース通信の update 機能が書き込みで使用される。RC ベースの同期がアプリケーションコードに挿入される。

update_drct ページ　ADSM の一種でホームを経由しない update プロトコルを専用ハードウェア不用で実現するページ。ホーム以外に同一論理アドレスのコピーページをプログラムロード時または大域的なメモリ割り付け時に用意する。このコピーページは対応するホームページが消去されない限り消去されない。メモリベース通信の update_drct 機能が書き込みで使用される。RC ベースの同期がアプリケーションコードに挿入される。

6　実行モデルとアドレスマッピング

一台のプロセッサに割り当てられるプログラムの実行の軌跡（命令流）のことをスレッドと呼ぶ。複数のスレッドで 1 つのプロセスを構成し、プロセスの内で同一クラスタに属するスレッドがサブプロセスを構成する。さらに、プロセス内の論理的にも物理的にも完全にメモリ空間を共有するスレッドがタスク [8]を構成する。タスクはサブプロセスと一致する場合もあり、複数のタスクが一つのサブプロセスに包含される場合もある。しかし、タスクが複数のサブプロセス（つまりクラスタ）に跨って存在することはない。プロセスの識別子（プロセス ID）はシステムで一意であり、プロセス ID が一致するサブプロセス / タスク / スレッドは同一プロセスに属している。メモリベース通信は相手の論理空間アドレスで通信がなされるため、タスク ID で通信相手や送信元が記述される。SPMD 型のプログラムのナイーブな実行モデルでは 1 タスクに 1 プロセッサが割り当てられ一時点では 1 タスクにおいて 1 スレッドが実行される。従って、マルチプロセッサ構成のクラスタを持つ場合はサブプロセスに複数のタスクが包含される。（図 2）。MTMT 型のプログラムではタスクはクラスタ内の複数のプロセッサの割当を同時に受けることができ、複数のスレッドが同時にスケジュールされ得る（図 3）[9]。

異なるサブプロセスに属するタスク / スレッドであっても通信や同期の必要なメモリ領域には少なくとも ADSM の意味の論理的共有メモリが実現される。つまり、同一プロセスであればグローバル共有変数はプログ

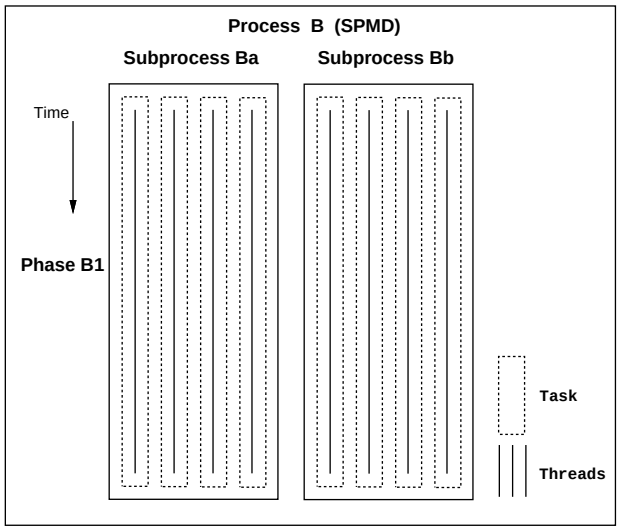

Figure 2　SPMD 型プロセス実行モデル

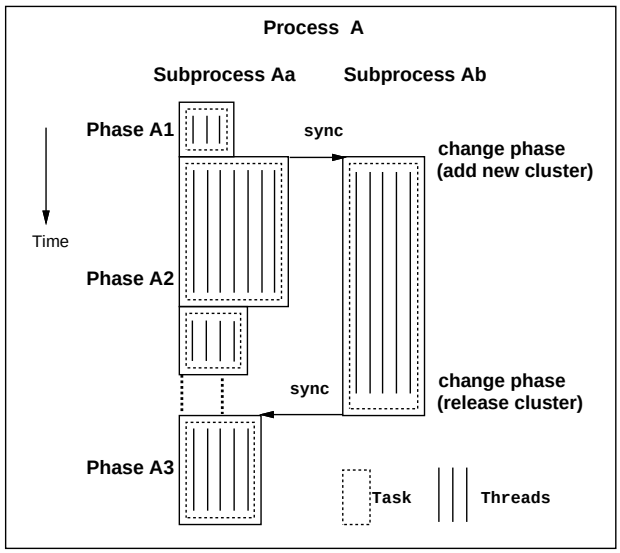

Figure 3　MTMT 型プロセス実行モデル

8　メモリ空間（ページテーブル）の割り当て単位であり、従来の OS ではこのタスクがプロセスと呼ばれることが多い。過去の SSS–CORE の文献ではサブプロセスがメモリ空間の割り当て単位を兼ねていた。

9　従来の SSS–CORE の関連文献では MTMT モデルに相当する実行モデルしか示していなかった。しかし、SPMD 型プログラムのローカル変数を効率良くサポートするためにタスクの概念を導入し、SPMD 型の実行モデルを新設した。

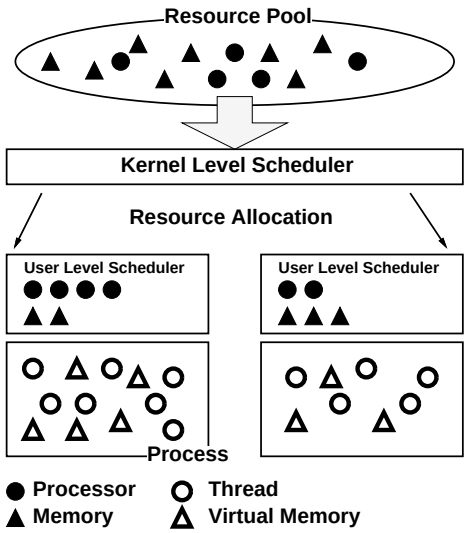

Figure 4　二階層スケジューリング

ラミングモデル上、同一アドレス空間に属し共有されている。ただし、クラスタやタスクが異なるスレッドのプライベート領域はたとえ実行コード上の論理アドレスが同一であったとしても共有されない。

プロセスのプロセッサ希望割り当て台数や実メモリ割り当て量はアプリケーションプログラム内でも時時刻刻と変化しており、特にプログラムの繰り返し構造が変更された時点で大幅に変化することがある。このため、サブプロセスにフェーズという概念を導入し、サブプロセスが複数のフェーズに時分割されると考え、フェーズ変更時にはスレッドの種類やサブプロセスレベルの資源要求数を変更可能にする。プロセス全体として資源要求の大きさが変更される場合はサブプロセス間で同期を取り、プロセス全体の要求としてカーネルに要求が出される。カーネルはプロセス (サブプロセス) のロード時とフェーズ変更時 (実際の変更よりも前に要求は出せる) にユーザからのカーネルレベルスケジューリングのための情報を入手する。プロセスレベルのカーネルレベルスケジューラに対する要請項目については後述する。

7　ユーザ / カーネル二階層スケジューリング

SSS–CORE も SS–CORE 同様にユーザレベルで資源保護制約を破らずに解決可能な資源管理はユーザレベル (コンパイル時の実行コードへの組み込み / 言語ランタイム / サーバ) で実現する。このため、カーネルレベルの機能としてはきめ細かなサービスを実現する必要はない。SS–CORE がプロセスに対して実プロセッサグループを割り当てていたのと同様に、SSS–CORE では実資源群 (プロセッサ、メモリページ、HDD のトラック / セクタ) をカーネルレベルスケジューラ (**KLS**) がプロセスに割り当てる (図 4参照)。基本的にプロセス内の資源の保護は行わず、一旦プロセスに割り当てられた資源は自由にユーザが使用することができる。プロセスにはユーザレベルスケジューラ (**ULS**) が存在し、KLS によって割り当てられた資源をスレッドに対してスケジューリングする。資源管理コスト上の問題から、ULS はクラスタ内 (サブプロセス内) の資源を管理する **CULS** が集まって構成され、CULS が各クラスタ内の資源を管理する。ただし、ULS や CULS は機能の名称であり、実際には独立したユーザプログラムやルーチンが存在する必要はなく、アプリケーションプログラムのコードの中に組み込まれていても構わない (図 1参照)。基本的に、新規に資源をシステムから獲得する場合と資源をシステムに解放する場合のみ KLS に ULS は処理依頼を行う。ただし、プロセスに割り当てられているプロセッサ上のスレッドの切替はユーザレベルで実現可能であるが、実メモリページの用途切替 (違う論理アドレスで使用) や HDD 上のファイル領域の再利用はユーザレベルで行うことは現状のハードウェア構成ではプロセス間の保護制約を守って行えない。このため、メモリやファイル資源の用途切替 / 再利用はオペレーティングシステム核内のカーネルレベルのサービスルーチンを実装上呼び出すことになる。KLS も実装上はクラスタ単位に資源管理を行うプログラム (**CKLS**) の集合体であり、CKLS の協調動作によってシステム全体のカーネルレベルスケジューリングが遂行される。

8　カーネルレベルスケジューラの資源管理

8.1　ユーザによる KLS の制約条件

タイムスライスごとに KLS はプロセスに対して動的な優先度管理を行って、優先度の高いプロセスから制約条件に基づいて、資源割り当てを行う。タイムスライスの間隔は 100msec から 300msec 程度の比較的大きな単位を考えており、スケジューリングのための処理コストは無視できる[10]。ここで使用される制約条件は静的なプログラム解析 / 最適化から得られるものであり、プロセスのロード時およびフェーズの変更時に ULS から KLS に指示される。プロセス側から KLS に指示できる制約条件について説明する。

1. プロセッサ台数制約
   - **1–a.** 台数一意指定:: 実行コードに固定台数による最適化が施されている場合。希望台数の上下限が一致させたもの。
   - **1–b.** 希望台数 (下限、上限) :: KLS は下限以上、上限以下のプロセッサを同時に割り当てる。
   - **1–c.** 初期台数可変 (下限、上限) :: KLS はロード時のスケジューリングにおいて、下限以上、上限以下のプロセッサを同時に割り当てる。二度目以降のスケジューリングでは初回と同じ台数を割り当てる。
2. プロセッサ間通信コスト制約
   - **2–a.** プロセッサ割当領域形状固定:: プロセッサ間トポロジを固定した最適化が実行コードに施されている場合。
   - **2–b.** プロセッサ間通信コスト上限:: プロセッサ間の最大距離の上限を指示する。システムが階層構造を持つ場合は階層に応じた離散値を取る。
3. メモリアクセスコスト制約

10　カーネルレベルスケジューリングの並列実行方式や軽量化は今後の研究課題である。

3–a. 自クラスタページアロケーションのみ:: 実メモリを要求した場合に自クラスタ以外に領域を取ることを許さない。自クラスタ内に空き領域がなければ、他のプロセス (実行待機中のプロセスを優先) のメモリを他クラスタまたは二次記憶に swap out する。

3–b. クラスタ間距離上限:: メモリ要求時に自クラスタのメモリがすべて使用されている場合、指定距離内 (アクセスコストがある範囲に抑えられる) に空き領域があれば、自プロセスのページを swap out して領域を確保する。指定距離内にも空き領域がない場合は、他プロセス (実行待機中のプロセスを優先) のメモリを他クラスタまたは二次記憶に swap out する。

4. マイグレーションコスト制約

4–a. クラスタ間距離上限:: 再スケジューリング時に、指定距離内のクラスタへのプロセスのマイグレーションを許容する。ただし、システム負荷分散が著しく均衡を欠く場合、この上限を無視して強制マイグレーションが行われる場合がある。システムが非常に大規模な場合には、強制マイグレーションに対する距離の上限も設ける必要がある。

4–b. ページ残存率下限:: 指定以下の割合の仮想ページしか実メモリ上に残っていない場合に、マイグレーションを行う。

資源割り当てに対してユーザが強い制約を課すると、割当が満たされた場合のエイジングの度合が大きくすることで、弱い制約を付けたプロセスを優遇する。

8.2 KLS の資源割り当て方式

資源 (プロセッサ、メモリ) 割り当て方式の概略を示す。クラスタ間ネットワークが階層構造を持つシステムではその階層構造に応じたクラスタ管理 tree を、フラットなメッシュ構造 (通常二次元メッシュ) の場合は 2 の次元数冪乗の tree 構造のクラスタ管理 tree を仮想的に構成する。この管理 tree の各ノードにおいて、その下層に位置するクラスタ内の資源情報を管理する。管理される資源項目は以下のとおりである。

- 階層内で走行中の各プロセス ID
- プロセス毎の占有プロセッサ数
- プロセス (待機中プロセスを含む) 毎の実ページ数
- 階層内フリーページの総数

この他に管理 tree の root では以下の情報を保持する。

- プロセス毎の KLS 制約条件
- プロセス毎の使用途中の仮想ページ総数
- 階層毎の資源総数
- 階層内フリープロセッサの総数

タイムスライスごとに KLS はプロセスに対して動的な優先度管理を行って、優先度の高いプロセスから管理 tree の情報 (制約条件を含む) に基づいて、資源割当を行う。タイムスライス時点でスケジューリングすべき対象を固定化するため、このスケジューリングは優先度および制約条件付きの静的な最適化問題である。静的な最適化問題であるため、評価関数を定めアニーリングや GA で解くこともできるが、 SSS–CORE では以下のヒューリスティックを使用する予定である。

1. 優先度の高いプロセスからプロセッサ資源の割当を管理 tree の subtree 単位で決める。 subtree 単位の選択に任意性がある場合は、実メモリの現在の割当状況を利用して、実メモリ残留数が多い subtree を選択する。ただし、強制マイグレーション中のプロセスはこの限りではない。
2. 上記のように subtree 単位でプロセッサ資源の割当を決定するが、割当対象プロセスの実メモリの残留数がページ残存率の下限を下回る場合もしくはプロセスが新規割り当ての場合は、領域の割当 (位置決定) は実メモリ割当状況と無関係に決定し、その割当が仮割当であることが判るように割当情報にマークを行う。
3. 優先度の関係で実メモリの残留数が多い領域に割り当てられなかったプロセスに関して、
   (a) 実メモリ割当状況から望ましい領域が仮割当のプロセスに占領されている場合は、その仮割当のプロセスが他の subtree に移動できないか調べ、移動可能であれば、望ましい領域にプロセスを割り当て、仮割当のプロセスは新しい領域で仮割当とされる。他の subtree に移動できない場合は、望ましい領域への割当は見送る。
   (b) 実メモリ割当状況から望ましい領域が優先度が高く移動できないプロセスに占領されている場合は、マイグレーション制約をチェックして、制約が満たされる範囲内で割当可能な空領域を探す。制約内に領域が見つからない場合は、このタイムスライスにおける資源割当を見送る。
4. すべてのプロセスの割当のチェックが終了しても未割当領域が依然存在し、領域割当を見送ったプロセス中にマイグレーション制約のみ適合しないものが存在する場合、強制的にこのプロセスをその未割当領域 (の一部) に割り当てる。ただし、これはシステムの負荷分散のための強制マイグレーションであり、強制マイグレーションの発生を優先度の再計算や再スケジューリングで考慮する。

スケジューリングの結果、同一領域で実行が継続されるプロセスに関して、 KLS の実行と無関係なプロセッサはプリエンプトされずにプロセスの実行を継続する。

8.3 優先度再計算方式

KLS に対してユーザによる制約条件が課せられるため、資源の fair share なスケジューリングを行われるような、優先度再計算方式を使用する必要がある。つまり、資源を多く占有したプロセス、厳しい条件を KLS に課したプロセスには相応のペナルティを課して、ユーザがむやみに条件を付加することが不利に働くようにする必要がある。このため、基本方針としては資源の占有量と占有時間の積に制約条件の厳しさや条件の満足度を加味した優先度のエイジングを行う。実際には、タイムスライス時に以下のパラメータを使用して優先度の再計算を行う。

- 独占プロセッサ数と使用時間の積 ($C_p n_p t,\ 0 < t < 1$)
- 割り当て領域内に獲得していた実ページ数とその領域への実行割り当て時間の積 ($C_m n_m t$)
- 制約条件の厳しさ ($0 < r_c < 2$)、制約条件の満足度 ($0 < s_c \leq 1$)
- 無駄になったプロセッサ資源と無駄になった時間の積を有効に利用された領域で加重配分 ($C_{wp} n_{wp} t \frac{n_p}{\sum n_p}$)
- 割り当て待ちの他のプロセスが存在するか ($f_{wait} = 0$ or 1)
- 強制マイグレーション割り引き率 ($f_{mg} = C_{mg}$ or 1, $0 < C_{mg} < 1$)
- 若返り係数と実行待ち時間の積 ($C_r(1-t)$)

単位時間間隔を 1 とし、間隔途中の新規割当 / 割当解除を考慮して、時間間隔内のプロセス割り当て時間を $t(0 < t \leq 1)$ とする。簡単のためメモリ占有領域の時間間隔内の増減は無視する。優先度のタイムスライスにおける更新加算量 (エイジング) 計算式は以下のような形式になる。なお、値が小さいほど優先度が高い。

$$((C_p n_p + C_m n_m) r_c s_c + C_{wp} n_{wp} \frac{n_p}{\sum n_p}) f_{wait} f_{mg} t - C_r(1-t)$$

9 ページ置換方式及びカーネルスケジューリング方式のシミュレーション評価

本節では *SSS–CORE* における分散共有メモリを想定したページ置換方式と、カーネルスケジューリング方式のシミュレーションによる評価について述べる。

9.1 シミュレーションモデル

これまでの研究で利用したモデル [15]を拡張し、 OS レベルのシミュレーションのため可能な限り簡略化されたモデル [16]を構築した。モデルの詳細については [17]を参照されたい。以下簡単に説明する。

対象とするのは分散メモリ環境で、メモリとプロセッサの対をクラスタとし、相互結合網は木構造である。相互結合網は二重化されたスイッチングネットワークで、スイッチングノードにおいてメッセージをバッファリングするワンホップ通信を行なう。

各プロセスには共有空間と非共有空間を、それぞれ別々のページ単位のメモリ参照頻度表により与える。一クラスタにおける並列プロセスの実行コンテキストをスレッドと呼び、各スレッドは異なった空間を与えられる。スケジューリングにより割り当て位置が変化した場合には、ローカルページはネットワークを介して on-demand に移動する。各プロセスは要求プロセッサ台数と同数のスレッドにより構成される。並列度は一定で、生成時から終了時まで変化しない。ページがリモートにも存在しない場合には、ディスクアクセスを生じる。共有空間に対しては分散共有メモリシステムを構築し、プロセスの各スレッドは参照頻度表を共有する。コヒーレンシ管理はアップデート方式で、ページは置換されない限りメモリに残る。また、メモリ参照コストはクラスタ内 $\ll$ クラスタ間 $\ll$ 2 次記憶であると仮定する。

プロセスの実行はクロックベースの確率モデルで、可能であれば毎クロックリード又はライトのメモリ参照を行なう。またプロセスのスレッドは、パラメータで与えられた時間間隔でランダムな組合せで同期を起こす。ここで有効実行 (時間 / クロック) とは、ページ待ち / 同期待ち以外の動作 (をした時間 / クロック数) のことである。表 1に今回のシミュレーション条件を示す。

Table 1 各種コスト及びパラメータ

項目	値
ディスクアクセス	10000 clk
リモートアクセス	100 clk
通信	10 clk
プロセッサ数	16 台
1 メモリのページ数	400 pages
1 ページのサイズ	4096 Byte
総メモリ量	25.6 Mbyte
1 quantum	100000 clk

9.2 スケジューリング法

シミュレータ上に 5 つのスケジューリング法を実装した。タイムスライス毎に資源の占有 / 消費状況に応じた優先度計算 [15]を行ない、優先度の高いプロセスから順に、それぞれ以下に示す方法で資源管理木構造を利用してスケジューリングを行なう。以下でホームノードとは管理木構造上の、プロセスの要求台数以上のプロセッサを含む最下位レベルのノードで、前回の割り当て領域を代表するものである。

algo4 ページのあるクラスタのみを割り当てる方式

algo3 ホームノード以下の領域でページのあるところから割り当てる方式

algo2 初めにホームノード以下の領域に割当を試み、失敗した場合にはルートノード以下の領域で、自分のページが存在する領域から割り当てる方式

algo1 資源管理木上の端から順に連続したプロセッサを必要なだけ割り当てる方式

algo0 ランダムに必要な数だけクラスタを選択し割り当てる方式

5 つの方式のうち algo2, algo3, algo4 が資源管理木構造を利用する方式である。この 3 方式の違いはプロセスへのプロセッサの割り当て領域が、使用中の実メモリページの位置に固執する度合にある。

9.3 ページ置換方式

分散メモリ環境上では分散共有メモリを仮定すると、参照頻度及び再アクセス時のアクセスコストの違いによりページを分類することが可能である。この分類を基に、参照頻度が少なく再アクセス時のコストが小さいページから置換対象とする効率的な方式が可能となる。

例えば、明らかに実行中のプロセスが利用するページよりは、それ以外のプロセスが所有するページの方がアクセスされにくい。また一つのプロセスのページの中でも、クラスタにローカルなページよりは共有ページへのアクセスの方が一般的に少ない。また共有ページはそれが複数存在するコピーのうちの一つであれば、再アクセス時には遠隔地からネットワークを経由して、低コス

Table 2　プロセス・セット (セット **J2**)

#DP	#PS	非共有空間サイズ * プロセス数
4	4	$30pages*2, 80pages*2$
8	4	$30pages*4, 80pages*1$
16	2	$30pages*2$
総並列度 80		総非共有空間 3200pages

#DP: 要求プロセッサ数, #PS: プロセス数

Table 3　プロセス・セット (セット **K2**)

#DP	#PS	非共有空間サイズ * プロセス数
3	5	$50pages*5, 80pages*2$
6	5	$50pages*5, 80pages*1$
13	1	$30pages*1$
16	1	$30pages*1$
総並列度 74		総非共有空間 3120pages

#DP: 要求プロセッサ数, #PS: プロセス数

トでアクセスすることが可能である。それに対して、再アクセス時にディスクアクセスを生じるようなページ置換は極力抑えた方が得である。そこでページの種類により、次のように置換対象とする順序を与えることができる。以下ではコピーを持たない共有ページを、便宜的にラストワン・ページと呼ぶ。

1. 他のプロセスの共有コピーページ
2. 実行中プロセスの共有コピーページ
3. 他のプロセスのラストワン・ページ
4. 他のプロセスのローカルページ
5. 実行中プロセスのラストワン・ページ
6. 実行中プロセスのローカルページ

置換方式としては、置換対象とするプロセス順をどのように選択するかにより、複数の戦略が可能である。今回のシミュレーションでは、次の二つの置換方式を実装し比較を行なった。

方式 **1**　ページの種類 / プロセスの区別なしに、単純な LRU 順

方式 **3**　スケジューリング優先度の低いプロセスから、上記のページの種類順に選択。各プロセスの共有 / 非共有空間毎に LRU 順を管理する

プロセスのスケジューリングは優先度 [17]の高い順に行なわれるため、優先度の低いプロセスのページほど参照されにくい。方式 3 はこの性質を利用する。ただしどちらの方式も、一貫性処理中の共有ページの場合は置換対象としない。

9.4　結果及び考察

前述のモデルを用いて表 1のようなパラメータによるシミュレーションを表 2, 3にあげたプロセスの組合せに対して行なった。各プロセスはどれも 80 ページの共有空間を持ち、 1000 有効実行クロック毎に同期する。図 5がその実行結果である。各スケジューリング方式 (横軸) に対して、システム全体の有効実行率 (縦軸) を示した。なおデータの取得はシミュレーション開始後、一つ目のプロセスが終了 (20 タイムスライス分の有効実行) した直後の時分割の時点で行なった。各メモリ置換方式を採用した場合の結果を示してある。 16 台構成の小規模システムでの固定台数制約を利用した状況では、プロセッサの利用効率が大きく評価に影響する。この環境で実際のシステムにおいて十分なプロセッサの利用効率がある場合の評価を行なうため、割当から洩れたプロセッサの実行は有効実行にカウントした。

まずメモリの置換方式についてみると、どの場合にも置換方式 3 が方式 1 に勝っている。最も効率の良かったセット J2 の 4 進 2 段の場合で見ると、置換の様子は表 4のようになっている。方式 3 ではプロセスの共有ページが置換されるのに対して、方式 1 ではプロセスのローカル空間のページやラストワン共有ページが置換されることが分かる。

Table 4　ページ種類毎の置換回数 (セット **J2, 4** 進 **2** 段)

ページ種類	0	1	2	3	4	5
方式 1	2815	661	60	1707	46	281
方式 3	7820	0	0	0	0	0

ページ種類は 9.3節での分類上の番号

また、スケジューリング方式についてみてみると、どの場合にもスケジューリング方式 algo4 が最良の結果を示している。 algo4、 3、 2 を比較すると、プロセス

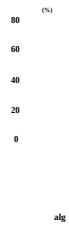

Figure 5　シミュレーション結果

を実ページを所有する領域に割り当てる傾向が高い方式ほど、実行効率が高いことがわかる。セット J2 の 4 進 2 段の場合での実行時間の内訳 (表 5) を例に見ると、通信 (主にページ要求やページ移動) や同期に要した時間に差があることが分かる。

これは、参照頻度の高いローカル空間のページの移動が少なくなるようにスケジューリングする効果が影響したと推定される。さらにプロセスの割り当て領域を頻繁

Figure 6　NOW 版 SSS–CORE の動作環境

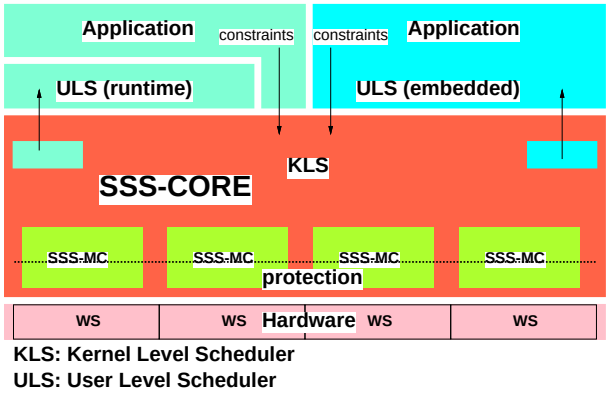

Figure 7　NOW 版 SSS–CORE の構成

に変更すると、各クラスタに各プロセスのメモリが集中してしまい、互いに実ページの奪い合いを起こす量が増加するため、 algo4 に比べて algo3、 algo2 が効率が悪くなっている。

Table 5　同期 / 通信時間 (セット J2, 4 進 2 段, 置換方式 3)

type	algo0	algo1	algo2	algo3	algo4
有効実行	57.48	65.81	71.99	82.88	82.88
同期	9.70	7.56	6.71	5.42	4.21
通信	13.47	11.48	7.96	4.87	3.32
ディスク	0.04	0.05	0.06	0.05	0.06

10　NOW 版 SSS–CORE の開発現状

現在、 NOW 版 SSS–CORE の Sun Microsystems 社の SPARCstation 20 またはこの互換機を EtherNet で接続した環境で動作している (図 6)。 SSS–CORE の WS 単体担当部分 (SSS-MC: Micro Core 図 7参照) では、メモリ保護、タイムシェアリング、プロセス管理、 UDP 通信、 TCP/IP 通信、キー入力、画面出力、外部 UNIX ワークステーションからのファイルの読み書き / プログラムロード / 実行等が実装されている。メモリベース通信機能は最もベースとなる遠隔メモリ書き込み

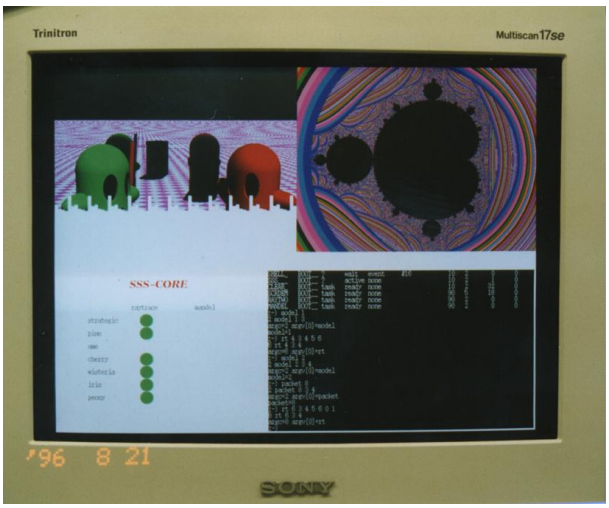

Figure 8　NOW 版 OS 上でのアプリケーション実行 1

がすでに実装されており、現在他のプリミティブを鋭意作成中である。遠隔メモリ書き込みを利用した並列レイトレーシングや並列マンデルブロー集合表示プログラムが現時点の SSS–CORE 上で並列実行可能である。図 8および図 9に NOW 版 SSS–CORE 上で、レイトテーシングならびにマンデルブロー集合表示プログラムを実行中の出力画面を掲載する。図 8はレイトレーシングを 6 台のプロセッサで並列実行中の画面であり、図 9はレイトレーシングを 3 台、マンデルブロープログラムを 6 台で実行中 (ただし、 3 台は二つのアプリケーションを時分割で実行) の画面である。

メモリベース通信機能の実装が済み次第、 ADSM の実装に着手する。ユーザのスケジューリング制約を守って資源をプロセスにスケジューリングするカーネルレベルスケジューラ (KLS) は、基本的な考え方は文献 [2] に示されており (本稿の第 8 節にも再掲されている)、本稿第 9 節にも簡単に掲載した確率モデルの精密なシミュレーションでスケジューリングポリシの詰めがなされている [17]。このシミュレーション結果を参考に、 KLS の最初のバージョンが実装される予定である。

11　メモリベース通信の初期評価

メモリベース通信の基本となる遠隔メモリ書き込みを使った細粒度の通信能力に関して初期評価を行った。評価に使用した NOW 環境は Axil 320 model8.1.2 (Sun SS20 互換機, 85MHz SuperSPARC CPU × 2) 6 台を 10BaseT のハブで EtherNet 接続したものである。

まず、基本性能値として、送信時 / 受信時のオーバヘッドは表 6に示す通りである。時間は 0.5μsec 単位の時計で計測した。 表 6に示されるオーバヘッドはすでにかなり低い値であるが、送受信ルーチン完成から日が浅いため、まだ最適化の余地はあると考えられる。なお、この計測に使った受信ではアクセスコストの高いフレームバッファに直接書き込みを行っている。 EtherNet では最小パケットサイズが 76byte 相当であるため、 10Mbps の 10BaseT ではどのケースでも明らかに

Table 6　遠隔メモリ書き込みの送受信オーバヘッド

転送サイズ (byte)	64	32	16	8	4	2	1
送信コスト (μsec)	11	10	9	9	8	6.5	6.5
受信コスト (μsec)	15	10.5	8.5	7.5	7	7.5	7

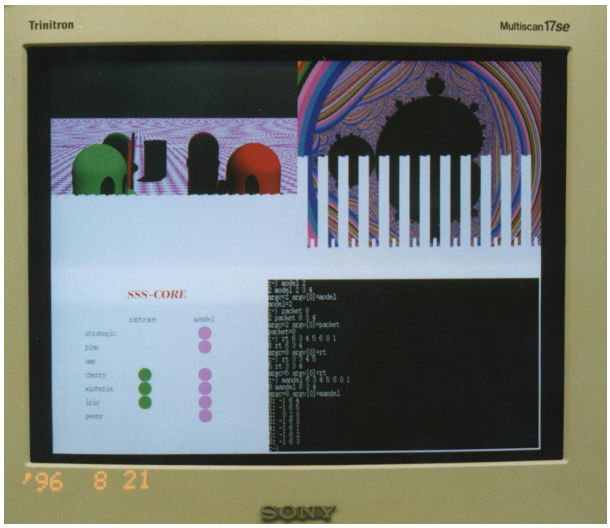

Figure 9　NOW 版 OS 上でのアプリケーション実行 2

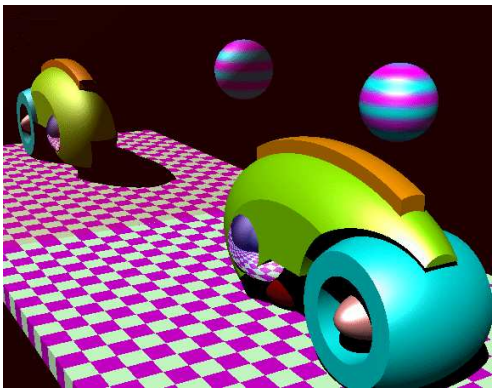

Figure 10　レイトレ画像 (実際は 256 色カラー)

転送がボトルネックになっている。

次に、並列レイトレーシングプログラム (以下レイトレ)[11]を使って、SSS–CORE 上で遠隔メモリ書き込みでフレームバッファ表示を行う実験を行った。実験に使用した 3D ソリッドモデルの完成時の絵を図 10に示す。この絵を 576 × 450 の解像度で並列計算を行った。並列計算の方法はサイクリックに N ピクセルずつを割り当て、N ピクセルのカラー値 (R,G,B 各 8bit) を計算した後、dither 変換を掛けた N ピクセル分の Nbyte データを 1 パケットとして表示ノードのフレームバッファに遠隔メモリ書き込みで直接書き込む (6 台の内 1 台は表示専用とした)。ただし、各クラスタにつき 1 台のプロセッサを使用し、各プロセッサはすべて 576 × 450 回のループを回り、ピクセルが自分の担当領域かどうか実行時に判断し、担当外であればスキップしている。時間は 100msec 単位で計測した。実験結果を表 7に示す。本実験における 1 ピクセル当たりの計算時間は平均約 70μsec(ピクセル上の絵の複雑さによって大幅に変化) である。表 7でアスタリスクが付いている項目は EtherNet の通信競合により大幅にプロセッサごとの実行時間が変動し処理全体の実行時間も安定しない (項目にもよるが結果に± 0.3sec 程度の幅が存在する) 実験項目である。表内には 3 回測定した平均が記入されている。転送サイズが 32byte 以下のパケットではヘッダーやダミー等も含めて 76 バイト分のパケットが毎回 Ether で通信されている。例えば、4byte パケットの通信では 4.92Mbyte ものパケットが送られる計算となる。これから 4byte/5 台の EtherNet の転送レートは 648 Kbyte/sec となり、ほぼ飽和状態である。他のアスタリスクの項目も同様である。なお、現在の実装の SSS–CORE は 16 回再送して送れないパケットは送信を諦めているため、アスタリスクの項目ではピクセルが何点か欠ける場合がある。また、実行時のパケットのコンバイニングをこの実験では行っていない。1 台の計算時間が 4byte 以下になるとパケット数増加による送信コストの増加を大幅に上回る実行時間の増加が発生する。これは計算時間で通信が隠せなくなったためである。比較的大きな粒度の通信でも、台数に関してリニアスピードアップに達していないのは、各プロセッサが担当以外のピクセルに対して若干の処理を行っていることと Ether 通信の衝突に起因するものである。

表 8は表 7と同一条件で動的コンバイニングと遠隔多重書き込み (1 パケットで複数の対象アドレスに書き込む) をサポートした場合の実行時間である。コンバイニングは送信バッファ (8 パケット分) が満杯で書き込めなかった場合にのみ行われ、一時的に送信ハードウェアと無関係な他のバッファにこのパケットを保管しておき、1 ピクセル分の計算を行い次のパケットを求め、新しいパケットと退避しておいたパケットをマージして送信を試みる。このときまだ送信バッファが空いてなければ次のパケットを計算しコンバイニングするといった手順を繰り返していく。このレイトレーシングアプリケーションでは送信粒度が 4byte 以下の領域でコンバイニングの効果が見られ、コンバイニングがない状況では飽和のため並列実行によるスピードアップが得られない領域でも、ほぼリニアなスピードアップが得られている。

最後に、SSS–CORE の軽いメモリベース通信と UNIX において軽いと言われている UDP を使ったソケット通信による性能を参考のために示す。このために UDP のソケットで通信するプログラムに前記レイトレを書き換

11　北海道大学の山本先生のプログラム [18]を C 言語で書き直し並列化した物を使用している。

Table 7 転送サイズと台数による並列レイトレ計算時間

転送サイズ (byte)	64	32	16	8	4	2	1
1 台計算時間 (sec)	18.1	18.2	18.4	19.0	20.5	27.0	37.7
2 台計算時間 (sec)	9.3	9.4	9.5	9.7	10.5	14.1	28.2
3 台計算時間 (sec)	6.8	6.5	6.5	6.7	8.0	*14.3	*26.9
4 台計算時間 (sec)	4.9	4.9	5.0	5.2	*7.8	*13.8	*25.3
5 台計算時間 (sec)	4.0	4.0	4.1	*4.3	*7.6	*13.7	*25.5

Table 8 動的コンパイニングを使った並列レイトレ計算時間

サイズ (byte)	64	32	16	8	4	2	1
1 台計算時間 (sec)	17.0	17.0	17.1	17.3	17.6	18.4	19.7
2 台計算時間 (sec)	8.8	8.9	9.0	9.0	9.1	9.5	10.2
3 台計算時間 (sec)	6.5	6.1	6.2	6.2	6.3	6.6	7.0
4 台計算時間 (sec)	4.6	4.7	4.7	4.7	4.8	5.0	5.3
5 台計算時間 (sec)	3.8	3.8	3.8	3.9	3.9	4.1	4.3

Table 9 UDP 通信による並列レイトレ計算時間

転送サイズ (byte)	64	32	16	8	4	2	1
1 台計算時間 (sec)	17.6	18.5	19.5	22.2	29.7	41.5	68.4

えたものを用意して、表示プログラムは SunOS 4.1.3 上の X11R6 で表示し (ただし XFlush はしない) 、計算プログラムは Solaris 2.4 上で動かした。使用マシン環境は SSS–CORE 用の環境の OS を交換してまったく同じにした。アプリケーションのコンパイル条件も同一のコンパイラを用い、最適化オプション (O4) も同一にした。表 9に実験結果を示す。測定は実時間で行い、 UDP 通信の送信側は送信バッファに書き込めない場合は書き込めるまで待つようにプログラムしたが、オーバヘッドを必要以上に増やさないため、それ以上の転送の保証を行わなかった。このことにより、 2 台以上の実行では通信の衝突により大幅にピクセルを取りこぼす。このため 1 台でレイトレを実行し、 1 台で表示した場合のみを表 9に示す。 64byte 転送においては UNIX 上の UDP が勝っているが、これは浮動小数点ルーチンが異なる等のベース性能の違いである。転送の粒度が細かくなるにつれて、 SSS–CORE の実行時間よりも長くかかるようになる。

12 おわりに

SSS–CORE はワークステーションクラスタ (NOW) および分散メモリ型並列計算機上でパーティショニングと時分割スケジューリングによるマルチユーザ / マルチジョブを実現する汎用超並列オペレーティングシステムである。 SSS–CORE はソフトウェアによるメモリベース通信機能、非対称分散共有メモリ、カーネル / ユーザの協調動作インタフェース、カーネルレベルスケジューラへのユーザ制約指定、資源使用量に応じた実行優先度の公平性の確保 といったスケーラブルな並列オペレーティングシステムのための新しい機能を有し、仮想化され保護された環境下で並列アプリケーションの効率の良い並列実行を可能とする。

本稿ではまず SSS–CORE の基本方式ならびに SSS–CORE のために開発され提案している各種機能について解説した。新機能のうちカーネルレベルスケジューリング方式とページリプレース方式に関しては確率ベースのシミュレーション実験の結果を示した。その結果、我々の提案方式が従来の方式に比べ良好なプロセッサ有効使用率を得られることが示せた。次に、現在研究開発中の NOW 版 SSS–CORE の開発状況を述べ、その上でメモリベース通信機能の基本機能に関する実測を行った結果を示した。実測によると 32byte 以下の細粒度遠隔書き込みであれば、送受信共に数 μsec 程度のオーバヘッドで 1 回の通信が可能である。このことは 100BaseT や高速光リンク等の速い通信ネットワークを使用した場合に、ユーザレベル通信同期専用ハードウェアがなくても高速かつ保護され仮想化されたユーザレベル通信同期が実現できる可能性を示唆している。

今後はカーネルレベルスケジューラの実機への実装等を行い NOW 版 SSS–CORE の完成度を上げるとともに、実機上での各種実験データを収集する予定である。また、仮想記憶をサポートするための DISK I/O ならびに DISK I/O のレイテンシを隠蔽するためのスケジューリング方式に関して研究開発を行う予定である。

References

[1] 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).

[2] 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).

[3] 松本 尚: 細粒度並列実行支援マルチプロセッサの検討. 信技報, CPSY89-37, pp.37-42 (August 1989).

[4] 松本 尚: 細粒度並列実行支援マルチプロセッサの検討. 情報処理学会論文誌 Vol.31 No.12, pp.1840–1851 (December 1990).

[5] 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).

[6] 松本 尚, 平木 敬: Memory-Based Processor による分散共有メモリ. 並列処理シンポジウム JSPP '93 論文集, pp.245–252 (May 1993).

[7] 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).

[8] 松本 尚, 平木 敬: キャッシュインジェクションとメモリベース同期機構の高速化. 計算機アーキテクチャ研究会報告 No.101–15, 情報処理学会, pp.113–120 (August 1993).

[9] 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 — 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).

[10] K. Li: IVY: A Shared Virtual Memory System for Parallel Computing. *Proc. 1988 Int. Conf. on Parallel Processing*, St. Charls, IL, pp.94-101 (August 1988).

[11] 松本 尚: 細粒度並列実行支援機構, 計算機アーキテクチャ研究会報告 No.77-12, 情報処理学会, pp.91-98 (July 1989).

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy: Memory Consistency and Event Ordering in Scalable Shared-Memory multiprocessors. *Proc. the 17th Int. Symp. on Computer Architecture*, pp.15-26 (May 1990).

[13] 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).

[14] 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).

[15] 信国 陽二郎, 松本 尚, 平木 敬: 汎用並列 OS のための資源情報を利用したスケジューリング方式の検討. 信技報, CPSY 95–64, pp.111–118 (August 1995).

[16] 信国陽二郎, 松本尚, and 平木敬: 並列 OS の性能予測を可能にするシミュレーションモデル. 情処研報 96-ARC-117, 96(23):19–24, (March 1996).

[17] 信国 陽二郎, 松本 尚, 平木 敬: 汎用並列 OS SSS–CORE におけるカーネルスケジューリング方式 — 詳細確率モデルによる性能評価 —. 情報処理学会研究報告 96–OS–73, 情報処理学会, Vol.96, No.79, pp.79–84 (August 1996).

[18] 山本 強: The 3 Dimensional Computer・Graphics. CQ 出版 (1983).