汎用超並列オペレーティングシステム SSS–CORE の資源管理方式

Resource management methods of the general-purpose massively-parallel operating system: SSS–CORE

松本 尚　　古荘 進一　　平木 敬
Takashi MATSUMOTO　　shinichi FURUSO　　Kei HIRAKI
東京大学 大学院理学系研究科 情報科学専攻
Department of Information Science, The University of Tokyo

概 要

Non-Uniform Memory Access（NUMA）型並列計算機上の汎用オペレーティングシステム **SSS–CORE（Scalable SS–CORE）**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機構を実現する。並列計算の効率を維持するために、OS によるハードウェア資源（プロセッサ、メモリ、I/O）のスケジューリングに対しアプリケーション側が制約条件を提示できる。また、OS の資源割り当て情報を低コストで実行時にユーザに開示するインタフェースを用意することで、プログラマ／コンパイラによる実行コードの最適化を支援する。

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、多数の MPU を使用した NUMA 型並列計算機が、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 NUMA 型並列計算機の効率の良い利用が可能となりつつある。しかしながら、汎用性導入による並列処理性能の大幅な低下が発生するため、現状の NUMA 型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は、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プロセスごとにプロセッサを割り当てて分散処理をしているに過ぎない。その原因は、ソフトウェアによる様々な最適化方式の適用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。つまり、高い効率を実現する鍵となる最適化が全てコンパイラ等静的手段で実現されるため、実行環境が独占的なものとなるためである。

我々は 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搭載できる NUMA 型並列計算機（分散メモリ型並列計算機、分散共有メモリ型並列計算機、ワークステーションクラスタ等）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）**の確立が非常に重要となっている。

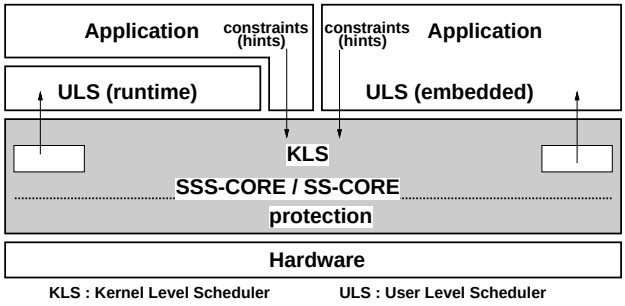

Figure 1　SSS–CORE / SSS–CORE の資源管理構造

2 UMA 型並列計算機用 OS 核 SS–CORE

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

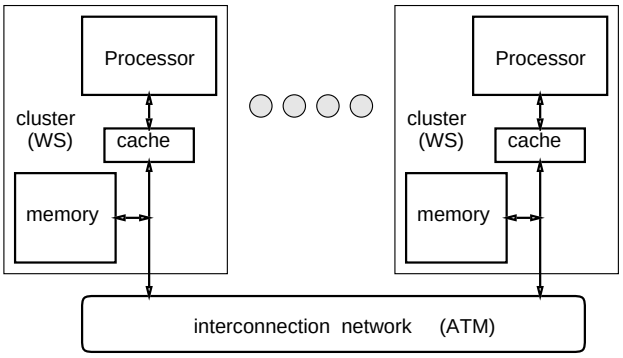

Figure 2 対象システム（ワークステーションクラスタタイプ）

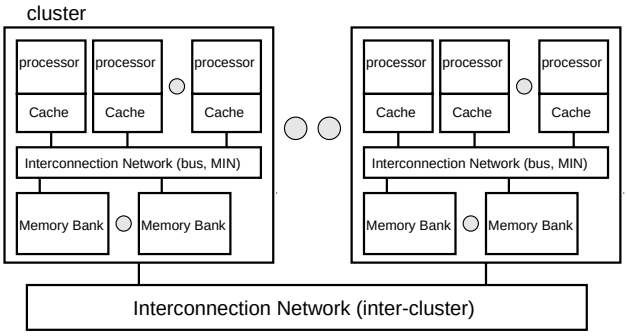

Figure 3 対象システム（並列計算機タイプ）

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 [2]を持つ（図 1参照）。また、SS–CORE ではマルチユーザ／マルチジョブの汎用性を実現するために、プロセッサ台数のみで決まるパーティショニングと時分割スケジューリングを利用していた。これらの基本方針に基づいてオペレーティングシステム核について研究を行ってきた。その結果、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／ユーザのスケジューリング情報を利用して飢餓状態を回避可能な Snoopy Spin Wait [3]
- 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 [3]
- 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 [3]
- 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 は SS–CORE の基本方針を受け継ぎ、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い使用する。

3 対象計算機システムと実行モデル

用語の混乱を避けるために対象計算機システムの構成と想定するプログラムの実行モデルについて説明する。

SSS–CORE は対象計算機を NUMA 型並列計算機に広げ実用的なオペレーティングシステム核の実現を目指

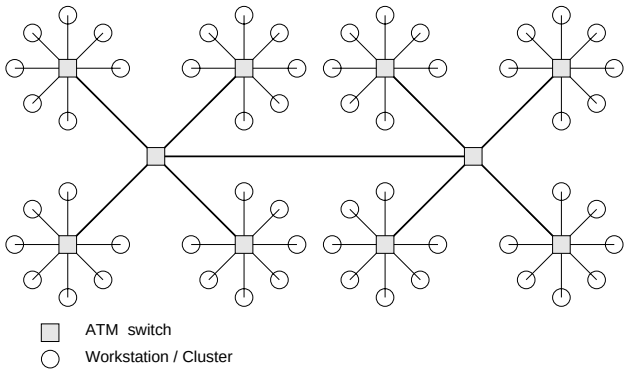

Figure 4 クラスタ間ネットワーク例

す。SSS–CORE で対象とする NUMA 型並列計算機を図 2および図 3に示す（入出力装置は省略した）。SSS–CORE は最終的には非均一な構成の並列計算機／分散計算環境を対象とするが、議論を単純にするために当面は同一プロセッサを使用し、クラスタ間ネットワークに対象性があるものに対象を限定する。図 3のようにシステム内に UMA 型並列計算機と見做せる部分を内包することを許し、UMA 型並列計算機部分を**クラスタ**と呼ぶ。よって、図 2ではプロセッサ 1 台と主記憶 1 バンクがクラスタを構成する。クラスタ間のネットワークは図 4のような階層性を持つ結合方式やメッシュのようなフラットな結合方式またそれらを組み合わせた方式で結合される。他のクラスタのメモリへのアクセスはハードウェアによる共有メモリであろうが、ソフトウェアによる非同期通信であろうが問題としない。これらの差は最適化の時点でのパラメータの違いとして吸収する。ただし、本稿では議論が発散するのを防ぐために、ハードウェアによる分散共有メモリがサポートされないシステムでは、お互いのクラスタのデータを低コストで参照できるように、仮想共有メモリ [4]がカーネルレベルで実装されているものと仮定する。仮想共有メモリのプロトコルはデータの性質によって最適なプロトコル（update 系を含む）が選択されるものとする。

SSS–CORE はメモリ管理および二次記憶管理も研究対象とする。個々のアプリケーションプログラムで実メモリ以上のデータを扱うのはユーザの責任、つまり明示的にユーザが nonblocking の I/O インタフェースを使って、I/O を行うこととする。しかし、マルチジョブであるために実メモリが不足する事態には SSS–CORE が仮想記憶によって対応する。このため、SSS–CORE 対象計算機システムには二次記憶（ディスク）システムが含まれる。seek を含めたディスクのデータアクセスレイテンシはクラスタ内メモリアクセスより二桁程度、クラスタ間メモリアクセスより一桁程度悪いものと仮定する。また、ディスク（実際には複数の RAID）のクラスタとの接続形態によって、各クラスタとディスクの間に距離の差が実際には生じるが、本稿ではこのコスト差は seek 等のオーバヘッドにより無視できるものと仮定する。つまり、ディスク内のデータのアクセスはどのクラスタからも等距離であると仮定する。また、ディスクとクラスタの間のデータ転送はディスクコントロール用のハード

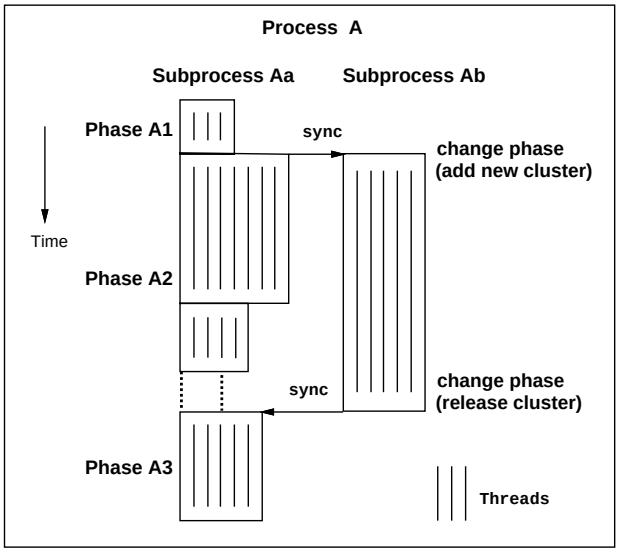

Figure 5 プロセス概念図

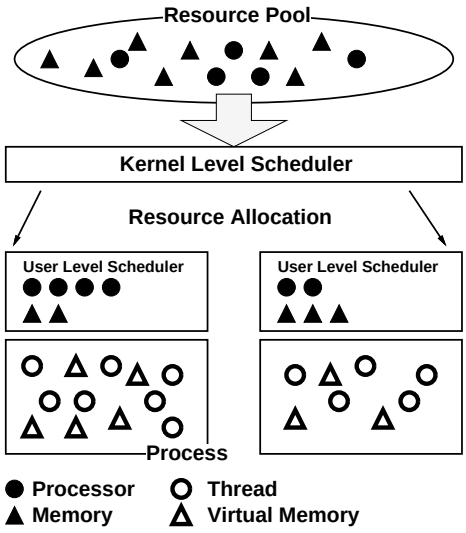

Figure 6 二階層スケジューリング

ウェアが行うため、このデータ転送のためのプロセッサパワーの消費は無視できるものとする。

対象計算機システム上のプログラムの実行モデルとしては、SS–CORE のモデルとほぼ同じものを採用する。差はクラスタ内とクラスタ間のアクセスコスト差を反映するためのものである。図 5に示すように実行モデルはプロセス、サブプロセス、フェーズ、スレッドという階層構造を持つ。 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。 1 つのサブプロセス内のすべてのスレッドは同一のアドレス空間を共有し、異なるサブプロセスに属する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共有メモリが実現される。つまり、同一プロセスであれば基本的に同一アドレス空間に属するが、クラスタが異なるスレッドのプライベート領域までアドレス共有されているわけではない。

スレッドにはさまざまな粒度のものが考えられ、静的に判断可能な粒度情報によってスレッドを分類し、その分類に応じたスレッド管理を行うことで実行の最適化を行う。粒度によるスレッドの分類は文献 [1]を参照のこと。サブプロセス内には同時に一種類のみのスレッドしか存在しないものとする。しかし、粒度や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。

4 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 6参照）。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。

5 カーネルレベルスケジューラの資源管理

5.1 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 から数秒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。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
   (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。
   (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。
   (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。タイムスライスによ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
   (a) 領域形状固定
   計算時のプロセッサ間の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。
   (b) 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
   (a) 自クラスタのみ
   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。
   (b) アクセスコスト上限
   実メモリを要求した場合に自クラスタがメモリがすべて使用されている場合、指定アクセスコスト内に空き領域があれば、自プロセスのページを swap out することで領域を確保する。指定アクセスコスト内にも空き領域がない場合は、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。
4. マイグレーションコスト制約
   (a) アクセスコスト上限
   タイムスライスによる再スケジューリング時に、指定アクセスコスト内のクラスタであればプロセスのマイグレーションを許容する。ただし、システムのロードバランスが著しく均衡を欠く場合は、この上限を無視してマイグレーションが行われる場合がある。
   (b) ページ残存率下限
   指定された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。

5.2 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て、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。これらの情報はユーザによって参照可能である。

- 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）毎の占有実ページ数
- 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している。

- 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。タイムスライスの時点でスケジューリングすべき対象を固定化するため、この問題は優先度および制約条件付きのスタティックな最適化問題となる。スタティック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 では以下のようなヒューリスティックを使用する予定である。

1. 優先度の高い方からプロセッサ資源の割当を管理 tree の階層単位で決める
2. 階層単位でプロセッサ資源がすべて埋まったら、優先度順に実メモリ残留数が多い位置の領域を割り当てる。ただし、その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の新規割り当ての場合は、領域の割当を後回しにす

る。

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は、マイグレーション制約を満たさない場合は、このタイムスライスにおける資源割当を見送る。
4. 未割当の階層や階層内プロセッサ群があるので、それらを制約条件に照らしながら、優先度順に埋めていく。
5. まだ埋まらないプロセッサ群で、未割当のプロセス中にマイグレーション制約にのみ合わないものが存在する場合、強制的にこの領域にプロセスを割り当てる。ただし、強制マイグレーション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ることなくプロセスの実行を継続する。

この KLS のスケジューリング処理は、システムの規模が非常に大きい場合には、管理 tree の特定の階層以上のノードに対応して、KLS 専用のプロセッサを固定して分散処理する必要がある。しかし、システムの規模が数百クラスタ程度で同時に処理されるプロセス数が数十程度以下であれば、タイムスライス時のみプロセッサをプリエンプトして KLS 処理に使用すればよい。

5.3 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。

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

5.4 先行スケジューリングと二次記憶プリフェッチ

対象システムにおいてはディスクとメモリ間のデータ転送によるプロセッサパワーの消耗は無視できる想定になっている。このため、時分割の計算中に次の間隔で割り当てられるプロセスのためのディスクアクセスを可能な限り行ってしまう。ユーザ／コンパイラがフェーズ毎に必要なデータ領域のヒントを KLS に出すようにする。また、プリエンプト時にはプロセッサの TLB 情報を保存し、その情報を再開時のワーキングセット情報として利用する。前もって二次記憶プリフェッチするため、KLS によるスケジューリングは 1 タイムスライス分先行して行われる。プリフェッチのために犠牲になる実ページは、実行中のプロセスに影響を与えないため、待機中のプロセスの占有ページから選択する。

6 おわりに

汎用超並列オペレーティングシステム SSS–CORE は NUMA 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。ユーザからのカーネルレベルスケジューリングへの制約条件によって、静的なプロセッサおよびデータのパーティショニングとスケジューリングを支援する。また、スケジューリング情報をユーザレベルの実行コードから低コストでアクセスするインタフェースが存在し、アプリケーションコードの実行時最適化と軽い同期を支援する。本稿では、SSS–CORE の基本方針、ユーザ／カーネルの二階層スケジューリング方式、カーネルレベルの資源管理およびスケジューリング方式について述べた。

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 大会論文集, pp.133-136 (October 1990).

[2] 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).

[3] 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).

[4] 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).