システムソフトウェアと
オペレーティング・システム 67－4
(1994. 12. 8)

超並列オペレーティングシステムカーネル SCore における IPC

効率的 IPC を目指したスケジューリング技法

堀 敦史†, 石川 裕†, Jörg Nolte†, 原田 浩‡, 古田 敦‡, 佐藤 忠§
†技術研究組合 新情報処理開発機構 (RWC) つくば研究センタ
‡(株) SRA, § 三菱電機東部コンピュータシステム (株)

†〒 305 茨城県つくば市竹園 1-6-1 つくば三井ビル
e-mail:{hori,ishikawa,jon,h-harada,furuta,csato}@trc.rwcp.or.jp

あらまし
超並列 OS におけるプロセス間通信では, 仮想記憶管理機構の技法を用いたプロセス間共有メモリによるプロセス間通信の高速化技法が必ずしも使えないことや, プロセスのスケジューリングに注意しないと効率的なプロセス間通信が実現できないなどといった, 逐次マシンの OS とは異なる側面を持っている. 本稿では, 超並列 OS 上での効率的なプロセス間通信を実現するために, "buddy" と "alternate" という 2 種類のプロセスグループを提案する. Buddy プロセスグループはパイプライン処理, alternate プロセスグループはプロセス間共有メモリを可能とする. 同時に, alternate プロセスグループを用いたスケーラブルな並列デバッガ実行モデルの提案も併せておこなう.

和文キーワード 超並列, OS, プロセス間通信, スケジューリング, デバッガ実行モデル

IPC on the Massively Parallel Operating System Kernel *SCore*

A Process Scheduling Technique for Effective IPC

Atsushi HORI†, Yutaka ISHIKAWA†, Jörg Nolte†,
Hiroshi HARADA‡, Atsushi FURUTA‡, Tadashi SATO§

†Real World Computing Partnership, Tsukuba Research Center, ‡SRA, §MTC
†1-6-1 Takezono, Tsukuba-shi, Ibaraki 305, JAPAN

Abstract
Inter-process communication on a massively parallel operating system have some different aspects from that on a sequential system. For example, the virtual memory management technique for efficient inter-process communication that can be seen in the Mach micro kernel can not be implemented on a distributed memory parallel machine. Process scheduling is also very important to realize efficient inter-process communication. In this paper, we propose 2 kinds of process groups, "buddy" and "alternate" to realize efficient inter-process communication. The buddy process group is suitable for pipeline processing, while the alternate process group enables inter-process shared memory. Also, we propose a scalable parallel debugger execution model based on the alternate process group.

英文 key words Massively Parallel, Operating System, Inter-Process Communication, Scheduling, Debug Model

1 はじめに

我々は, 超並列オペレーティングシステムカーネル SCore を設計中である. SCore[8, 9] は, RWC-1[7] のような, マルチスレッド, 分散メモリ, 空間（パーティション）分割可能な超並列マシンをターゲットに, 超並列のスケーラビリティを損なうことのない超並列 OS の開発を研究目標としている.

超並列マシンの OS におけるプロセス間通信では, Mach などに見られるプロセス間共有メモリによるプロセス間通信の高速化技法 [6] が必ずしも使えないことや, プロセスのスケジューリングに注意しないと効率的なプロセス間通信が実現できないなどといった問題が生じる. 並列あるいは分散システムにおけるプロセス間通信とスケジューリングの問題については, 例えば文献 [1, 4] で, 既に論じられている.

本稿では, 超並列 OS 上で効率的なプロセス間通信を実現するために, プロセススケジューリングとの関係に焦点を当てる. 分散メモリ, 空間分割可能な超並列 OS におけるプロセス間通信の問題点を明らかにし, 我々が設計／開発を進めている SCore において, それらの問題に対しどのように対処するかについて述べる.

構成

次章に本研究におけるいくつかの前提について述べ, 3章において超並列 OS に特有のプロセス間通信の特徴についてまとめる. 4章では, それまでに述べた各種の問題点に対するひとつの解決策として, プロセス間通信を考慮したスケジューリング上の技法を提案する. 5章では, 前章の結果を受け, スケーラブルなデバッグ実行モデルの提案をおこなう. 6章では, SCore 上に実装される予定の時分割／空間分割スケジューリング方式である DQT についてその概要を説明し, 時分割／空間分割スケジューリングにおけるプロセス間通信の問題点について触れ, 4章で提案された技法を DQT 上に実装する方法について説明する. 7章では, 本研究と関連研究との関係について論じる.

2 前提

対象とする超並列マシンは, マルチスレッド, 分散メモリ型であり, 時分割／空間分割可能なプロセススケジューリングが可能とする. 図 1 に時分割／空間分割スケジューリングの概念図を示す [8, 9]. 具体的な時分割／空間分割プロセススケジューリングとして, 我々は既に DQT (Distributed Queue Tree) を提案している [10, 11].

本稿においてプロセスとは, あるパーティション（プロセッサ空間のサブセット）を時間的, 空間的に排他的に占有する並列プログラムの実行を指す. OS／カーネルはプロセス単位でスケジューリングするものとする. プロセスは非常に細粒度なマルチスレッドで実行されるものし, OS は基本的にスレッドの管理に関与しないものとする. また, 高速な通信を可能とするため, プロセッサ間通信はカーネル経由でなくユーザが直接通信することを前提とする.

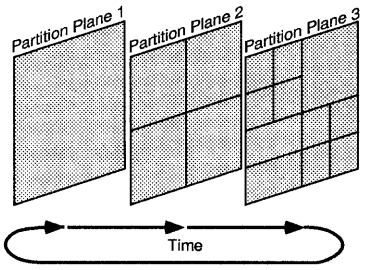

図 1: 時分割／空間分割スケジューリングの概念図

3 超並列 OS におけるプロセス間通信の特徴

以下に, 超並列 OS 固有のプロセス間通信における特徴について論じる.

メモリと通信の非双対性

Mach では, メモリ管理の巧妙な仮想記憶管理テクニックを用い, プロセス間通信におけるデータ（メッセージ）のコピーを避けることで, 効率的なプロセス間通信を実現している [6]. しかしながら, 分散メモリ型の超並列マシンにおいて, Mach のような仮想記憶の技法によりプロセス間通信を実現することは, 一般には不可能である. なぜなら, プロセス間通信をおこなうプロセスが物理的に異なるプロセッサ／メモリに配置される可能性があるからである. したがって, 共有メモリによる効率的なプロセス間通信を実現する場合, プロセスを同じパーティションに配置する必要がある.

プロセス切替を伴わないプロセス間通信

もし, プロセス間通信をおこなうプロセス群が, 空間的に排他的に（あるいは disjoint な関係にあるパーティションで）, 同時にスケジューリングされることが保証されていたならば, それらのプロセス同士のプロセス間通信はプロセス切替なしに可能となる. 例えば, UNIX のようにプロセスがパイプで接続されているような場

合, 上記のような条件が満たされるならば, パイプで接続されたプロセスは, 真の意味で並列にパイプライン処理が可能となることを意味する.

閉パーティション性の保存

RWC-1 では, プロセス内のプロセッサ間通信の影響が, 他のプロセスに及ばないようにプロセッサ間通信ネットワークが「閉パーティション特性」を持っている [8, 9, 12]. 閉パーティション性は, 計算資源であるプロセッサ, およびそのプロセスにおける通信に関与する通信（サブ）ネットワークの割当を保証することを意味する. もし, プロセス間通信でやりとりされるデータが大量であり, その通信経路がプロセス間通信とは関係のないプロセスのパーティションを経由するとなると, その影響は無視できない場合が考えられる. これを回避するには, プロセス間通信に関与するプロセスをグルーピングし, ひとつの「閉パーティション」を構成するようにする必要がある.

プロセス間通信とプロセススケジューリングの関係

前述したように, Mach などに見られるような仮想記憶管理技法による方法を実現可能とするには, プロセスは同じパーティションに存在する必要がある. 一方, プロセス間通信に関与するプロセスの（時間的）スケジューリングが同じである場合は, 高効率なプロセス間通信が実現可能である. このように, 効率的なプロセス間通信を超並列 OS で実現するには, プロセススケジューリングとの関係を無視することはできない.

4 プロセス間通信に着目したプロセスグループ

我々は, 効率的なプロセス間通信を実現するにはスケジューリング上の工夫が必要と考え, プロセス間通信に着目したプロセスグループの導入を考える. ここでの「プロセスグループ」とはプロセス間通信に関与するプロセスの集合であり, プロセスの集合の時空間的な配置を規定するものである. したがって, UNIX でいうところのプロセスグループとは異なる. また, 各メンバープロセスのスケジューリングは時分割／空間分割スケジューリングとは独立なものとし, 時分割／空間分割スケジューラはプロセスグループ単位で時空間スロットを割り当てる.

ここでは2種類のプロセスグループ buddy と alternate を提案する.

Buddy プロセスグループ

Buddy プロセスグループにおいて, プロセスグループの各メンバープロセスは, スケジューリングパーティション内の部分パーティションに空間排他的にマッピングされる. 図2は4つのプロセスが buddy プロセスグループを構成している例である（図中 "Scheduling Partition" との記述があるが, これについは後述する）. この図の中で太い矢印はプロセス間通信を表している.

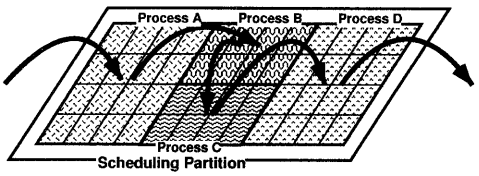

図 2: Buddy プロセスグループの例

Buddy プロセスグループにおける各メンバープロセスは, 常に同時にスケジューリングされることが保証されているものとする. このため, メンバー間のプロセス間通信において, 以下に示す特徴がある.

- 相手プロセスが常に走行状態にあるため, プロセス間通信に起因するプロセス切替の必要がない.
- 複数の通信路を確保することにより, 全体として高いバンド幅を確保できる.
- プロセスによりスループットが異なる場合は, スループットがほぼ等しくなるようにプロセッサの数（パーティションの大きさ）を割り当てることで, 全体として高いスループットが得られる.
- 通信メッセージの送り手と受け手が物理的に異なるプロセッサにあるため, メッセージのコピーは回避できない.
- Buddy プロセスグループはその性質上静的であり, グループ形成後のメンバープロセスの追加は不可能である.

もし, メンバープロセス間の通信速度が完全に同期しているならば, buddy プロセスグループにおけるプロセス間通信においてバッファリングの必要性はなくなる. しかしながら, 現実にはこのような状況を実現することは困難であり, バッファリングは依然必要となる.

Buddy プロセスグループは, 主に UNIX に見られるパイプ接続されたプロセス間通信の高効率化を想定したものである.

Alternate プロセスグループ

Buddy を空間排他的なプロセスグループであるとすると, alternate は時間排他的なプロセスグループである. Alternate プロセスグループの例を図 3に示す（図中 "Scheduling Partition" との記述があるが, これについては後述する）.

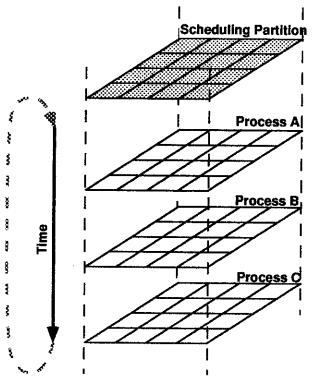

図 3: Alternate プロセスグループの例

Alternate プロセスグループ内のメンバープロセスのスケジューリングは, いくつかの方法が考えられる. 例えば, メンバープロセスでレディ状態にあるものをラウンドロビンで順にスケジューリングする方法が考えられる. Alternate プロセスグループの特徴を以下に列挙する.

- メンバープロセスが同じパーティションに存在することが保証されているので, プロセス間の共有メモリが実現可能である.
- メッセージ通信においては, 送り手と受け手が同じプロセッサにあった場合, 仮想記憶管理の技法によりメッセージのコピーを回避可能である.
- プロセスグループのメンバーが動的に変更可能である.
- 閉パーティション特性は, メンバープロセス間のプロセス間通信においても保存される.
- メンバープロセス間のプロセス間通信において, 必ずプロセス切替を伴うため, buddy の場合程高いスループットは期待できない.
- メンバープロセスは全て同じパーティションを共有するため, メンバー毎にパーティションの大きさを変えることはできない.

5 並列デバッガ実行モデル

本章では, UNIX におけるデバッガ実行モデルである ptrace もプロセス間通信（あるいはプロセス間共有メモリ）の一種と考え, alternate プロセスグループに基づく並列デバッガ実行モデルを提案する.

ここでは gdb のようなデバッガを分散メモリ型超並列マシンに実装することを想定して話を進める. デバッガプロセスは端末からの入力に従って, 被デバッグプロセス全体の実行を制御したり, レジスタやメモリの内容をダンプしたりするものとする. ここで, デバッガプロセス（debugger）と被デバッグ（debugee）プロセスは同じ alternate プロセスグループにあるものとする.

例えば, 並列デバッガにおけるステップ実行コマンドの処理は, 個々のプロセッサにおいて以下の手順を踏む [3].

1. 被デバッグプロセスの対応するプロセッサのプログラムカウンタを調べ, 次に止める場所を特定し, trap 命令を（ptrace により）埋め込む.
2. 被デバッグプロセスの実行を再開する.
3. 先の手順 1 で埋め込んだ例外が発生するのを待つ. 例外が発生したら, trap 命令で置き換えた命令を元に戻す.
4. 全てのプロセッサで例外が発生したら, ユーザにステップ実行の完了を通知する.

5.1 超並列プロセスデバッグの問題点

先のステップ実行の例において, 例外の発生回数はプロセッサの数に比例する. もし,（逐次の gdb と同様に）例外発生のたびにプロセスを切替えたのでは, プロセス切替の回数はプロセッサの数に比例することになり, スケーラビリティが失われてしまう. したがって, このような場合にはカーネルが被デバッグプロセスの状態を監視し, 全てのプロセッサでの例外の発生を待つ必要がある.

より顕著な例としては, gdb の条件ブレークコマンドが挙げられる. gdb には被デバッグプロセスの内部状態によりブレークする／しないを決めることができるデバッグコマンドがある. これを「条件ブレークコマンド」と呼ぶ. 以下に関数 `foo` のエントリで, 変数 `x` の内容がゼロであった場合に止まる, という意味の条件ブレークコマンドの例を示す.

```
(gdb) break foo if x == 0
```

もし, このコマンドが全てのプロセッサに対し有効だとすると, 条件が成立するまでの間に発生する例外の総数はプロセッサ数に比例する. これは, 例外のたびにい

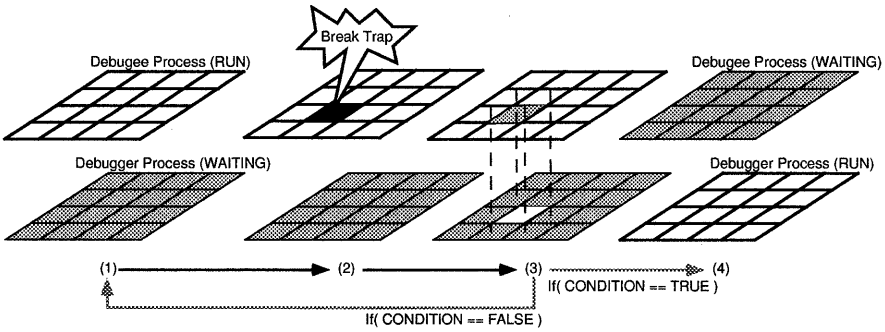

図 4: 条件ブレークの実行例

ちいちプロセス全体を切替えたのでは, スケーラブルな条件ブレークコマンドの実装が不可能であることを意味する.

5.2 Alternate を用いたデバッガ実行モデル

そこで, 例外が発生した時点でプロセス全体をデバッガプロセスに切替えるのではなく, 例外が発生したプロセッサのみをデバッグプロセスに切替えることを考える (図 4) . この手順は以下の通り.

1. 条件ブレークコマンドで指定されたアドレスに trap 命令を埋め込み, 被デバッグプロセスを走らせ, デバッガプロセスは例外の発生を待って休眠する (図 4 の (1)) .
2. trap 命令による例外を検知したカーネルは, そのプロセッサのみをデバッガプロセスに切替える. これをプロセス全体 (パーティション全体) のプロセス切替と対比して「局所プロセス切替」と呼ぶことにする (図 4 の (2) → (3)) .
3. 局所プロセス切替されたデバッガプロセスでは, 被デバッグプロセスの状態を ptrace により調べ, 条件成立の有無を判定する. もし, 条件が成立しなかった場合は, そのまま被デバッグプロセスの実行を再開する (図 4 の (3) → (1)) . 条件が成立した場合は, 被デバッグプロセス全体を休眠させ, 全プロセッサをデバッグプロセスに切替える (図 4 の (3) → (4)) .

この方式は,

- 例外発生プロセッサ以外のプロセッサでは通常の (被デバッグプロセスの) 実行が継続するため, スケーラビリティを損なうことなく条件ブレークコマンドが実現できる.
- 条件ブレークの条件が成立した場合のみ, デバッガプロセスにプロセス全体を切替えれば良い.

といった長所があるが, 以下に示すような欠点もある.

- 例外が発生したプロセッサでは, デバッガプロセスが走行するが, ここではデバッガプロセス全体と通信することはできない. なぜなら, 他のプロセッサ上では被デバッグプロセスが走行しているからである.
- 例外発生プロセッサ上でのみデバッガプロセスが長時間走行すると, そのプロセッサ上で被デバッグプロセスの処理が進行しないため, 通信のためのメッセージバッファが溢れる危険性や, 被デバッグプロセスのタイミングのずれによる挙動の変化が大きくなる可能性がある.
- 条件判定に大域変数の参照を含むことはできない. なぜなら, 大域変数の参照は (一般に) 通信を伴うからである.

これらの欠点は, 必要に応じてデバッガプロセスにプロセス全体をプロセス切替することで回避可能である.

6 DQT における実装方式の検討

6.1 DQT スケジューリングの概要

本章では DQT スケジューリングの概略について簡単に述べる. 詳細については文献 [10, 11] を参照されたい. 図 5に DQT の例を, 表 1 に図 5 の DQT における時分割スケジューリングの例を示す. 表中, $Q_i(j)$ とあるのは i 番目の DQT のノードの待ち行列中の j 番目のプロセスが走行していることを示す.

DQT は動的パーティションの入れ子構造を反映した木構造を成す. DQT ノードはパーティションに 1 対 1

で対応している. DQT ノードにはそのパーティションに割り当てられたプロセスの走行待ち行列を持つ（図5において, DQT ノードを示す丸の右側にあるのが待ち行列を表す）.

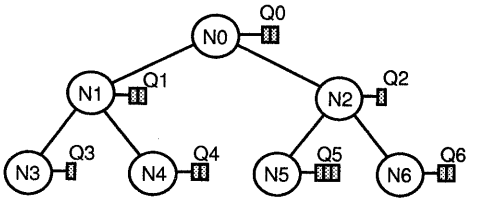

図 5: DQT の例

表 1: DQT スケジューリングの例

Time Slot	PE0	PE1	PE2	PE3
0	$Q_0(0)$			
1	$Q_0(1)$			
2	$Q_1(0)$		$Q_2(0)$	
3	$Q_1(1)$		$Q_5(0)$	$Q_6(0)$
4	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(1)$
5	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(0)$
6	$Q_0(0)$			
7	$Q_0(1)$			
8	$Q_1(0)$		$Q_2(0)$	
9	$Q_1(1)$		$Q_5(0)$	$Q_6(1)$
10	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(0)$
11	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(1)$
12	$Q_0(0)$			
:	:			

プロセス生成時には, そのプロセスの実行に必要な大きさのパーティションを DQT スケジューラに問い合わせる. DQT スケジューラは, システムの負荷状況から判断して, スケジューリング上の公平さ, 負荷のバランスなどを考慮して必要とされるパーティションを確保する.

プロセス生成時にパーティションを割り当てる場合の例を図 6に示す. 図中 add_task というメッセージが DQT のルートノードに投げられている. 各 DQT ノードを示す○の中の数字は, あるポリシーにより決まる負荷を代表する数値を示している. このポリシーにおいては, 各 DQT ノードでは, 子ノードの中から最も数値の低いノードに対し, add_task メッセージを転送している. add_task メッセージ引数の数値は, ここで投入するタスクが必要とするパーティションの大きさを示しており（この場合は１）, add_task メッセージの転送は, タスクが必要とするパーティションサイズを担当する DQT ノードに到達するまで続けられる.

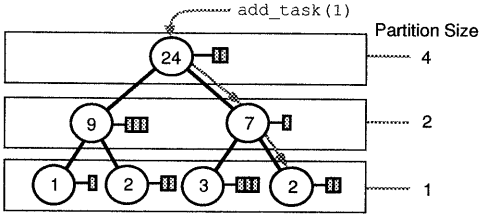

図 6: パーティション割当の例

6.2 プロセス間通信の問題点

超並列 OS におけるプロセス間通信とスケジューリングの関係は逐次のそれと比べ, i) ハードウェアがスレッド実行をサポートするような場合（例えば, RWC-1[7]）, シングルスレッドの場合に較べ, プロセスにおけるプロセス間通信の待ちという状態が明確でないこと, ii) 時分割／空間分割スケジューリングをおこなう場合, プロセス間通信の依存関係を考慮したスケジューリングが困難であること, という問題点が挙げられる.

前者の問題点はマルチスレッドアーキテクチャ固有の問題点である. マルチスレッドアーキテクチャの利点を活かすため, カーネルは複数のユーザスレッドの待ちを内部に抱えている. このため, プロセスの待ち状態が何に起因したものなのかは一般には判らないのである.

後者の問題点は, プロセスが生成されるパーティションの位置は DQT の負荷状況によってのみ決まるため, プロセス間通信をおこなうプロセスの位置関係を規定することはできないことに起因する. また, DQT の枝に負荷のアンバランスが生じた場合, 負荷の軽い方の枝にあるプロセスは, プロセッサ利用率を向上させる目的から, スケジューリングの機会がより多く与えられる. このためプロセス間通信においては, 送信側プロセスと受信側プロセスのスケジューリングの機会のバランスと, プロセス間通信の需要供給のバランスが一致しない可能性がある.

6.3 スケジューリングパーティション

Buddy パーティションにおいては, 最初からプロセスグループに必要なパーティションを確保する必要がある. これは, i) 既に DQT にスケジューリングされている任意のプロセス同士を buddy とすることができないこと, ii) 例え buddy の構成が可能だとしても, 閉パーティション性を保存できない可能性が残る, といった理由による.

そこで, DQT スケジューリングに「スケジューリングパーティション」を導入する. スケジューリングパー

ティションとは, DQT においてプロセスグループのためのパーティションを確保するための単位である. DQT はスケジューリングパーティションに対し時空間スロットを割り当てる.

Buddy プロセスグループにおいては, 各メンバープロセスは disjoint なサブパーティションが割り当てられる. Alternate プロセスグループの場合は, ユーザが自由にメンバープロセスの追加／削除を指定できるものとする.

7 関連研究

7.1 Co-scheduling

Ousterhout はプロセス間通信頻度が高いと思われるプロセスのグループ（task force）を常に同じタイムスロットを割り当てることで, スケジューリングのずれによる同期遅延を回避する co-scheduling を提案している [1].

超並列マシンにおける co-scheduling は以下に示すような２つの意味を持つと考えられる.

プロセス内の co-scheduling

並列プログラムにおいては, 他のプロセッサとの通信は基本的な操作のひとつであり, 通信における遅延は並列プログラムの実行効率を低下させる. プロセス内の co-scheduling は, プロセッサ間通信のスケジューリングのずれによる遅延をなくし, 効率的な通信を可能とする.

プロセス間の co-scheduling

Buddy プロセスグループのように, プロセス間通信に関与するプロセスの同時スケジューリングが保証されているような場合には, そうでない場合に較べ, 効率的なプロセス間通信が実現可能でなる.

一般に, 時分割／空間分割スケジューリングにおいてプロセス間の co-scheduling を実現するには, i) スケジューラがプロセス間通信の状況を考慮し, 動的に最適なプロセスのスケジューリングを行なう方法と, ii) 先に説明した buddy プロセスグループのように, ユーザ（あるいはシェル）がプロセス間通信を考慮してプロセスをグルーピングし, スケジューラはプロセスグループ単位でのスケジューリングを行なう方法, の２つの方法が考えられる.

パーティション分割可能な並列マシンの時分割／空間分割スケジューリングでは, 時間的／空間的要因が絡み複雑な処理を必要とする. DQT スケジューリングのようにボトルネック解消の意味から, スケジューリング処理を分散させている場合は, スケジューリング処理の複雑さの度合が増大する. 場合によっては, プロセスマイグレーションの必要性も生じることになる. ここでは, 前者の方法では必要以上に処理の複雑さを招くと考え, 後者の方法を採用することとした.

7.2 Buddy プロセスグループ

Buddy のアイデアは Cm* 上の Medusa オペレーティングシステムで提案されている [2]. Medusa で提案されている buddy (activity) は, あるプロセスで発生した例外事象の処理を例外が発生したプロセッサと異なるプロセッサで処理することを目的としている.

本稿では, buddy プロセスグループを効率的なプロセス間通信を実現するためのスケジューリング手法として提案した. デバッグモデルとして buddy を用いなかった理由を以下に示す.

- デバッガプロセスを被デバッグプロセス生成の後から attach することが困難である.
- 最大パーティションを必要とするプロセスのデバッグが不可能である.
- 被デバッグプロセスがマルチスレッドで走行している場合, デバッガプロセスに例外を通知している間に, 他のスレッドが走行してしまう可能性がある. この結果, 被デバッグプロセスのデバッグ時の実行と通常の実行が乖離する可能性がある.
- 基本的にデバッグプロセスと被デバッグプロセスは排他的に走行する. このため buddy をデバッグモデルとしたのではプロセッサの利用効率が半分程度になってしまう.

7.3 スケーラブルなデバッガ実行モデル

CM-5 には統合された並列デバッグ環境として Prism[3] が提供されており, そこでは条件ブレークコマンドも用意されている. しかしながら, マニュアルにも条件ブレーク設定時の実行は著しく遅くなることが明記されている [5]. これは, 例外発生の都度プロセス切替が生じているためと推測される.

本稿で提案したデバッグ実行モデルは, 大域的なプロセス切替をできるだけ回避しようとするものである. 局所的なプロセス切替の回数と大域的なプロセス切替の回数は同じである. 局所的なプロセス切替を用いた場合, 例外が発生していないプロセッサ上での被デバッグプロセスの実行が継続される点が大きく異なる. このため, ここで提案したデバッグモデルは, 条件ブレーク設定時においても, スケーラビリティを損なわないことが期待される.

8 まとめ

本稿では, 超並列OSにおけるプロセス間通信とプロセススケジューリングの関係に焦点をあて, 効率的なプロセス間通信を実現するためには, スケジューリング上の工夫が必要であることを示した. また, 時分割／空間分割スケジューリングにおいて, プロセス間通信を意識した2種類のプロセスグループ buddy と alternate をスケジューリングの単位とすることを提案した.

概念的には, buddy プロセスグループは連続的なパイプライン処理に向いており, alternate プロセスグループはバッチ的なロット単位の処理に向いているものと思われる. プロセス間通信の特性に合わせ, これらのプロセスグループをプロセス間通信の形態により適切に選択することで, 効率的なプロセス間通信が実現できるものと期待される. また, alternate プロセスグループの応用の1例としてスケーラブルなデバッガ実行モデルをも提案した.

なお, プロセスグループやプロセス間通信の具体的な抽象化, シグナルなどの詳細については, 検討が進み次第, 稿を改めて報告したい. 本稿で提案されたプロセスグループおよびデバッグモデルは, 今後さらに詳細な検討を加え, RWC-1 上に実装されるオペレーティングシステムカーネル SCore に組入れる予定である.

謝辞

RWC 超並列ソフトウェアワークショップに参加の各位からは数々の貴重なアドバイスを頂いた. また, RWC つくば研究所の超並列ソフトウェア研究室の OCore 研究グループ, 超並列アーキテクチャ研究室, および RWC 超並列 MRI 研究室の方々には多くの議論に参加して頂いた. ここに感謝の意を表す.

参考文献

[1] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of Third International Conference on Distributed Computing Systems*, pp. 22–30, 1982.

[2] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[3] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. A Scalable Debugger for Massively Parallel Message-Passing Program. In *Scalable High-Performance Computing Conference*, pp. 825–832, May 1994.

[4] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[5] Thinking Machines Corporation. *Prism User's Guide*, April 1994. Version 2.0.

[6] M. Young, et al. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. *Operating Systems Review*, 21(5):63–76, 1987.

[7] 坂井, 岡本, 松岡, 広野, 児玉, 佐藤, 横田. 超並列計算機 RWC-1 の基本構想. In *JSPP'93*, pp. 87–94, 1993.

[8] 堀, 石川, 坂井, 小中, 前田, 友清, 松岡, 岡本, 廣野, 横田. 並列計算機オペレーティングシステムカーネル SCore におけるプロセス管理とハードウェア支援機能. コンピュータシステム・シンポジウム論文集, pp. 59–66. 情報処理学会, October 1993.

[9] 堀, 石川, 小中, 前田, 友清. 超並列システムカーネル SCore の構想. システムソフトウェアとオペレーティング・システム研究会資料, pp. 57–64. 情報処理学会, August 1993.

[10] 堀, 石川, 小中, 前田, 友清. 超並列オペレーティングシステムにおけるスケジューリング 方式の提案. システムソフトウェアとオペレーティング・システム研究会資料, 94-OS-63, pp. 25–32. 情報処理学会, March 1994.

[11] 堀, 石川, 小中, 前田, 友清. 超並列マシンにおける時分割スケジューリング方式. システムソフトウェアとオペレーティング・システム研究会資料, 94-OS-65, pp. 33–40. 情報処理学会, July 1994.

[12] 横田, 松岡, 岡本, 廣野, 堀, 児玉, 佐藤, 坂井. 超並列計算機RWC-1の相互結合網. pp. 25–32, August 1993.