分散記憶方式並列計算機のための
効率的時分割スケジューリングの研究

堀 敦史

論文要旨

情報処理ニーズの高まりに伴い，並列処理技術に対する期待も強まっている．最近ではより費用対効果の高いクラスタ技術が注目されていることもあり，並列計算機が急速に身近になりつつある．ところで，並列計算機の実際の運用を見てみると，バッチスケジューリングを中心とした運用が多い．時分割スケジューリングの特徴を活かした対話処理は，プログラム開発効率を高めることが知られている．過去にメインフレームやスーパーコンピュータがバッチスケジューリング主体による運用から時分割スケジューリング主体の運用に移行したように，成熟しつつある並列計算機も時分割スケジューリング主体の運用にと移行するものと考えられる．クラスタ技術による並列処理の底辺の拡大は，この傾向をより強めるであろう．

しかしながら，並列計算機における時分割スケジューリングを実現するには，いくつかの技術的な問題を解決しなければならない．また，並列計算機における時分割スケジューリングのオーバヘッドは対話処理を可能にする程度に小さいことが望まれる．本研究は，並列計算機において対話処理を範疇に含めた効率的な時分割スケジューリングを，ソフトウェアの立場から実現することを目的としている．

本研究ではギャングスケジューリングによる時分割スケジューリングの実現に焦点を絞る．並列計算機が持つ性能ポテンシャルを十分引き出すと同時に，時分割スケジューリングのオーバヘッドを最小限に抑えることが重要である．このために本研究では，ギャングスケジューリングに焦点を絞り，ネットワークプリエンプション方式による実装方式を提案し，時分割スケジューリングオーバヘッドかどの程度かを評価した．その結果，時分割間隔 $100\,msec$，64 ノードにおいて $4\,\%$ 以下というスケジューリングオーバヘッドが得られた．さらに，現在までに提案されている他の時分割スケジューリング手法とギャングスケジューリングとを比較し，それぞれの特性について検証した．その結果，実装されたギャングスケジューリングにおいては，他の方式に比較してより安定した性能を発揮することが判明した．

並列計算機においては時分割スケジューリングだけによる対話処理は効率的にはなり得ない．例えば，並列プログラム実行中にユーザからの入力を待っているようなアイドル状態を検出し，プロセッサ資源の無駄な割り当てを避けることが必要である．本研究ではネットワークプリエンプション方式を応用することで，効率的に並列プログラム実行におけるアイドル状態を検出する機構を提案する．前述したような低い時分割スケジュー

リングオーバヘッドと，アイドル状態検出の機構により，並列計算機においても逐次計算機に近い対話処理が実現可能であることを示す．

さらに，ジョブスケジューリングという立場から，並列計算機の空間分割という特徴を考慮し，時分割スケジューリングを主体とする新たなスケジューリング手法が必要である．これに対しては，時空間分割スケジューリング方式のひとつとして“Distributed Queue Tree”を提案する．その特性は評価され，従来提案されているバッチスケジューリングとの性能が比較された．提案された時空間分割スケジューリングは，バッチスケジューリングと同等の性能を発揮することが判明した．

以上の結果から，並列計算機におけるギャングスケジューリングに基づく時分割スケジューリングは，僅かなスケジューリングオーバヘッドを代償として，対話プログラミング環境を実現できること，また，時分割スケジューリングを主体とする時空間分割スケジューリングにおいてはバッチスケジューリングと同等の性能が得られることが実証された．これにより，近い将来，並列計算機においても時分割スケジューリングが主体の運用になるものと考えられる．

THE SUMMARY OF PH.D DISSERTATION

The higher the demand of information processing, the higher the needs on parallel processing. Recently clustering technologies which enable to develop more cost-effective parallel machines are gathering attentions. Most parallel machines including the clusters, however, are operated with batch scheduling. Interactive programming environment which time sharing scheduling enables is thought to be efficient for program development. In the past, main frames and super computers had been operated with batch scheduling, but today they are operated with time sharing scheduling. Parallel machines are also expected to be operated with time sharing scheduling in the future.

To realize a time sharing scheduling on parallel machines, some technical problems must be answered. And the scheduling overhead of time sharing scheduling must be so low that response time of interactive parallel programs is short enough. The objective of this paper is to develop time sharing scheduling techniques on parallel machines, and to investigate if the developed time sharing scheduling can be applied to an interactive parallel programming environment.

In this dissertation, gang scheduling is focused as a time sharing scheduling technique on parallel machines. To minimize time sharing scheduling overhead and maximize the performance of parallel programs, a gang scheduling implementation technique using network preemption is proposed. And this proposed time sharing scheduling technique is evaluated. As a result, the gang scheduling overhead less than 4 % on 64 processing nodes with a 100 $msec$ time slice is obtained. Further, the implemented gang scheduling is compared with another time sharing scheduling technique. The comparison results show that the gang scheduling exhibits more stable scheduling behavior.

Time sharing scheduling may not provide an effective interactive programming environment. A parallel program under execution may be idle and waiting for user input. This situation must be detected by a scheduler, and processor resources must be reallocated to the other runnable parallel processes. It is proposed that the implemented network preemption technique is applied for detecting idle state of a parallel program

execution. As a result of the time sharing scheduling overhead of less than 4 %with a 100 $msec$ time slice, and this proposed idle detection mechanism, it is shown that an interactive parallel programming environment can be realized on parallel machines.

From the viewpoint of job scheduling, scheduling scheme based on time sharing scheduling is required to take account of parallel machine characteristics. In this paper, a "time space sharing scheduling" technique, called "Distributed Queue Tree," is proposed. The proposed time space sharing scheduling is evaluated and compared with batch scheduling techniques so far proposed. It is found that the proposed time space sharing scheduling technique exhibits the same performance with the best batch scheduling.

It is concluded that a time sharing scheduling based on gang scheduling on a parallel machine can realize an effective interactive parallel programming environment with a small sacrifice of scheduling overhead. And a time space sharing scheduling based on time sharing scheduling exhibits almost the same performance with the best batch scheduling. Consequently, it is expected that time sharing scheduling will also be popular on parallel machines in the future.

目次

図目次

表目次

第 1 章

序論

ある程度の規模を越える機械を設計する際には，その機械を構成する要素の特質を考慮しバランス良く設計することが，高い性能を引き出すためには非常に重要である．計算機もその例外ではない．プロセッサ，バス，メモリ，I/O の性能といったハードウェアの性能だけでなく，ソフトウェアの低レベルからアプリケーションレベルに至るまで，全てにおいて「バランス」が大切な要素である．特に並列計算機においては，通信ハードウェア及び通信ソフトウェアの要素が付加されるため，逐次計算機に比べバランスがより大切となる．

本研究は，ソフトウェアという立場から，分散メモリ型並列計算機を対象とした時分割方式によるジョブスケジューリング方式をテーマとする．本研究は，実装に留まる，あるいは，単にスケジューリングの効率のみを追い求めるものでもない．その設計には，並列計算機ハードウェア技術，通信ソフトウェア技術の最新傾向が反映されている必要がある．並列処理というコンテキストでは，絶対的あるいは価格を意識した，アプリケーションレベルでの性能が一義的な目標である．並列計算機上の時分割スケジューラは，この点を考慮し，ハードウェアからアプリケーションに至るまで，バランス良く設計される必要がある．本研究ではこのような立場から，分散メモリ型並列計算機上に対話処理を目標とし，性能バランスを考慮した，並列計算機上での時分割スケジューリングについて研究するものである．

1.1 並列処理のニーズとジョブスケジューリング

昨今のパーソナルコンピュータ（PC）の急速な普及や，インターネットの爆発的な拡大は，個人レベルでの情報化時代の到来を示すものと考えられる．企業においてもイントラネットの構築によりインターネット技術を積極的に利用しようとする動きが盛んである．PC は個人レベルでの情報処理能力を高め，インターネットは情報の流通を地球規模で急激に拡大しつつある．マルチメディア通信の基盤となった World Wide Web（WWW）の普及は，画像，音声，などというように，これまでとは本質的に異なる通信の質，量が要求されている．Web ページの検索や WWW による放送といった新しい情報処理システムの開発も盛んである．このように情報処理に対する要求は業種職種を問わず拡大しつつある．流通情報量の拡大により，今後も高次元の情報および情報サービスの提供が求められるという傾向に変化はないと考えられる．

情報処理能力を飛躍的に高める技術のひとつとして並列処理が注目されてきた．現在，並列計算機は価格対性能比に優れたスーパーコンピュータの一種であると認識されている [情処 93]．並列計算機のアプリケーションの大半は米国の HPCC などに見られるように科学技術計算である．ワークステーションと高速ネットワークを用いるクラスタ技術 [ACP+95] は，並列計算機の価格対性能比を向上させる．拡大する情報処理のニーズとクラスタ技術のシーズは，並列計算機のアプリケーション分野を，コスト意識の強いより一般的な情報処理，例えば WWW サーバ，Video on Demand，巨大データベースの OLTP など，に拡大させる．例えば，カリフォルニア大学バークレー校（UCB）では NOW [ADV+94] を用いて，Web ページの高速検索システム[1]や，ディスクに格納してある大量データの世界最高速ソーティングを実現している．カルフォルニア大学サンタバーバラ校では，Meiko 社の CS-2 を用いて高性能かつ拡張性の高い WWW サーバの構築を推めている [AYHI96]．また，Pixar Animation Studio 社が製作した映画 “Toy Story” は 300 台規模のワークステーションクラスタにより全編コンピュータグラフィクスで作成されている [HHJK96].

ところで，現在並列計算機を有する計算センターの多くは，基本的にバッチスケジューリングにより運用されている．昼間はプログラム開発ジョブを優先するために数十分から 1 時間の時間打ち切りで運用し，夜間にはプロダクション実行を優先するためにより長い時間打ちきりにより運用がされている [STT95]．バッチスケジューリングはプロ

[1]現在，Inktomi Corp. になっている．

ダクション実行では問題にならなくても，開発環境としては問題がある．大規模な計算ジョブを投入する場合は翌朝のプログラム開発時間帯に食い込むことは避けなければならない．また急ぎのプログラム開発があっても深夜に及ぶデバッグは避けなければならない．このように，バッチスケジューリングによる運用は，弾力的な運用を妨げ，プログラムの生産性を低下させる．文献 [STT95] では，米国 NASA の NAS （Numerical Aerodynamic Simulation）スーパコンピュータ施設におけるシステム管理者の立場からの並列計算機における管理システムに対する不満と将来への要望がまとめられており，この中で並列計算機における時分割スケジューリングに対し大きな期待が寄せられている．

もし，現在のワークステーションと同等の時分割スケジューリングが並列計算機上に実現されれば，並列計算機の柔軟な運用や効率的なプログラム開発を可能にする [STT95]. その結果，ニーズの多様化に対応できる場面も増えることが期待される．例えば大規模な Computer Aided Design などのように，数時間を要した実行時間が並列化により数分に短縮されるならば，それまで対話処理が非現実的であったアプリケーションの対話処理が可能になる．科学技術計算のオンラインによる可視化された結果を見ながら，シミュレーションのパラメータを調整することが可能になる．さらに，時分割スケジューリングが実現する対話環境は並列プログラム開発を効率的にする． Web サーバやデータベース処理のように 24 時間連続したサービス提供が必要な場合でも，トランザクション処理の合間を縫ってプログラム開発が可能になる．

このように，時分割処理スケジューリングとそれによって実現される対話処理は，並列計算機の弾力的な運用を可能にすると同時に利用率の向上が期待される．その結果，多様化する並列計算機に対するニーズへの適応力を増し，並列計算機の運用コストを下げ，新たなアプリケーション分野を切り開く可能性を持っていると考えられる．

1.2 研究の背景

逐次計算機においてバッチスケジューリングと時分割スケジューリングの比較に関する議論において，以下のことが知られている．

バッチスケジューリング： スケジューリングオーバヘッドが少ないが，応答時間は長い．

時分割スケジューリング： スケジューリングオーバヘッドは大きいが，応答時間は短い．

これらから，バッチスケジューリングによる運用はプロダクション実行に向き，時分割スケジューリングはプログラム開発や対話処理に向くとされている．ここで「並列プロセス」とは，あるひとつの並列プログラムから同時に起動された並列計算の実体となる逐次プロセスの集合と定義する．

メモリキャッシュ等の影響を除けば，時分割スケジューリングのオーバヘッドは原理的に時分割間隔に逆比例する．時分割間隔を十分大きくすることで時分割スケジューリングのオーバヘッドを許容可能な範囲に収めることが可能である．しかしながら，対話処理という側面からは非実用的な応答時間となる可能性がある．時分割スケジューリングの大きな特徴である対話処理を実用とする場合には，ユーザとの対話という局面において十分な応答性を示すことができる時分割間隔に対し，並列プロセス切替に要する処理時間が十分小さいことが必要となる．

一方，最近では並列プロセス内のプロセス間通信において， OS をバイパスする「ユーザレベル通信」が主流になりつつある [CMC97, THIS97, WBvE97]． OS を経由することで生じるオーバヘッドを低減し，その分高性能な通信を実現するためである．しかしながら， OS を経由しないことで，多重並列プログラミング環境を実現できない可能性が生じる． OS 経由の通信では多重並列プログラミング環境を実現することは可能であるが，その分並列プロセスにおける通信性能が低下する．従って，並列計算機の時分割スケジューリングのオーバヘッドを論じる際には，並列プロセス切替に要する時間の他に，多重並列プログラミング環境に対応することにより生じる被スケジューリング並列プロセスそのものの実行効率低下をも考慮されなければならない．

さらに，並列プロセスをどのように時分割スケジューリングするか，という問題もある．先に並列プロセスは逐次プロセスの集合と定義した．並列プロセスを時分割スケジューリングする際，並列プロセスの要素である個々の逐次プロセスのプロセス切替を同期させるかさせないかにより，被スケジューリング並列プロセスの実行効率や挙動にどのような影響があるのか，といった問題もある．

対話処理では並列プロセスがユーザからの入力待ちとなる場合がある．ディスクなどの I/O デバイスと異なり，ユーザの応答時間に有効な上限を設定することができない．このため並列プロセスがユーザからの応答だけを待ち，他に有効な計算を行なわないような状況に陥ることが容易に想定される． Unix などの逐次計算機の OS においては，逐次プロセスがこのような状況に陥った場合を検出し，他に走行可能な逐次プロセスがあれば，走行可能なプロセスにプロセッサ資源を譲るというスケジューリングが行なわれ

ている．並列プロセスにおいて有効な計算が行なわれずに，なんらかの事象だけを待っている状態を検出するのは簡単ではない．個々のプロセスの状態が通信メッセージの到着により変化するからである．このように並列プロセスの大域的な状態を検出することは，「分散プロセスの大域的停止状態の検出問題」（global termination detection problem）として 80 年代から知られている問題である．この問題に対し，実践的な解法を与えることで，並列プロセスがユーザからの入力を待っている CPU 時間を他の並列プロセスのスケジューリングに割り振ることが可能になる．その結果，システム全体のスループットが向上し，多くの対話処理ジョブが投入されているような状況では応答性の改善も期待できる．

ところで，逐次計算機におけるジョブスケジューリングは，待ち行列にあるタスクをどの順で処理するかという問題であり，基本的に時間軸での議論である．並列計算機の場合，投入されたジョブを複数のプロセッサで構成されるプロセッサ空間のどこに配置するかという空間軸におけるスケジューリングも可能である．従って，並列計算機におけるジョブスケジューリングでは，待ち行列にあるジョブをどの順で，プロセッサ空間のどこに割り振るか，といった問題になる．並列計算機の時分割スケジューリングにおいては，空間分割スケジューリングとどのように融合させるかが焦点となる．また，ジョブの投入や終了のタイミングでシステムの負荷状況が変化するため，動的に時分割スケジューリングを変化させる必要がある．どうのようなアルゴリズムで動的なスケジューリングを実現するか，その時のスケジューリングの特性はどのようなものか，バッチスケジューリングと比較して性能的な差異がどの程度なのか，といった事項が研究の焦点となっている．

1.3 研究の目的と範囲

本研究のテーマは「並列計算機における効率的時分割スケジューリング」である．時分割スケジューリングの方式そのものに関しては，時分割スケジューリングのオーバヘッドの程度，時分割スケジューリング方式の違いによるスケジューリング特性の違い，という 2 つの問題点を前節で示した．また，効率的な対話処理を並列計算機上に実現するための被スケジューリングプロセスの大域状態の検出が必要となることを述べた．さらに，並列計算機におけるジョブスケジューリングの特徴として，時間軸と空間軸のスケジューリングが必要であることを示した．

これらの背景を基に，本研究における具体的な課題を以下のように設定する．これらの課題を達成することで，並列計算機において時分割スケジューリングがバッチスケジューリングに対し，妥当なオーバヘッドを伴って実用的になり得ることを実証するものと考えられる．

- 並列計算機における低オーバヘッドな時分割スケジューリングの実装方式の提案
- 並列計算機における時分割スケジューリング手法の比較
- 被スケジューリング並列プロセスの大域状態の検出手法
- 時分割スケジューリングと空間分割スケジューリングを融合したスケジューリング方式の提案
- 時分割スケジューリングと空間分割スケジューリングを融合したスケジューリング方式とこれまでに提案されたバッチスケジューリング方式の比較

本研究で対象とする並列計算機は，MIMD あるいは SPMD 型で，空間分割可能な分散記憶方式（メッセージ通信型）とする．原理的にギャングスケジューリングの利点は共有記憶型並列計算機でも同じである [GTU91]．しかしながら，共有記憶型並列計算機の通信は暗黙的であり，分散記憶型並列計算機の通信は陽に記述される．このように通信に対する考え方が共有記憶型と分散記憶型では大きく異なっている．第 2.2.3 節で述べるように，ギャングスケジューリングの実装は通信方式に強く依存する．本研究においてギャングスケジューリングの実装方式を検討するにあたり，分散記憶型の通信モデルを対象とする．

本研究において，分散記憶型並列計算機を構成する要素計算機は全て均しいものとする．これは不均質な並列計算環境でのジョブスケジューリングは問題を徒に複雑にし過ぎ，均質な並列計算機上での効率的な時分割空間分割ジョブスケジューリングおよびギャングスケジューリングの研究が優先する，との理由による．

また，スケジューリングの対象とする並列計算機は十分なメモリを持っているものとし，メモリ資源の不足がスケジューリングに影響を与えないものとする．現時点において，並列計算機上での有効な要求時ページング方式が存在しないこと，メモリの低価格化傾向が今後も続くと考えられており，システムに対するメモリの相対的な価格が引続き低下するという理由による．

並列計算機の効率的な時分割スケジューリング方式を検討するにあたり，基本的な方針として「提供されるシステムに合わせて効率的になるように記述された並列プログラムを効率的に走らせる」ことを優先することとした．非効率に書かれた並列プログラムの性能を底上げすることは考慮しない．ジョブスケジューリングの対象としては並列ジョブのみを扱うものとする．逐次ジョブは並列度 1 の並列ジョブとして扱うものとする．

1.4 研究の方針

ここで先に示した 5 つの課題に対しアプローチを示す．

課題「並列計算機における低オーバヘッドな時分割スケジューリングの実装方式の提案」においては，ギャングスケジューリングに焦点を当て，実装することによりそこでのオーバヘッドを検証する．ギャングスケジューリングとは，並列プロセスを構成するプロセス群を一斉に切替える方式のことである．実際にギャングスケジューラを実装し，そこでの評価を行なう．ギャングスケジューラのオーバヘッドを低減すると同時に，被スケジューリング並列プロセスの並列効率を損なわないようにするために，被スケジューリング並列プロセスではユーザレベル通信を可能とすることを前提とする．前述したように，ギャングスケジューリングのオーバヘッドは時分割間隔に逆比例する．本研究においては，並列計算機における対話処理の可能性について論じることも目標としていることから 100 $msec$ という時分割間隔をひとつの基準にした． 100 $msec$ という時分割間隔は初期の Unix において設定された時分割間隔である [LMKQ89]．

課題「並列計算機における時分割スケジューリング実現手法の比較」においては，並列計算機における別な時分割スケジューリング手法である非同期コスケジューリング（第2.2.3節参照）を比較対象とする．スケジューリングオーバヘッドに関して先に実装されたギャングスケジューリングと非同期コスケジューリングの比較評価を行なう．

課題「被スケジューリング並列プロセスの大域状態の検出手法」では，大域状態検出のための実践的な手法を提案し，先に実装されたギャングスケジューラに組み込み，その動作を検証する．

課題「時分割スケジューリングと空間分割スケジューリングを融合したスケジューリング方式の提案」と課題「時分割スケジューリングと空間分割スケジューリングを融合したスケジューリング方式とこれまでに提案されたバッチスケジューリング方式の比較」では，時分割と空間分割スケジューリングを融合させるようなスケジューリング方式を

提案し，シミュレーションによる評価を通じて提案されたスケジューリング方式の特性を調べる．また，これまでに報告されているバッチスケジューリングと提案されたスケジューリング方式の比較し，スケジューリング性能の違いについて論じる．

1.5 構成

次章では，本研究の研究範囲に特に関連すると思われる技術や研究ついて，クラスタ技術と並列計算機におけるジョブスケジューリング技術を選び，それらの現状についてサーベイを行なう．特に並列計算機で時分割スケジューリングを実現するためのひとつの技術であるギャングスケジューリングに関しては，より詳しく報告する．第 3 章では，本研究における実装や評価のプラットフォームとなった RWC PCC-II と SCore クラスタソフトウェアについて概要を述べる．SCore クラスタソフトウェアとは，クラスタで用いられているハードウェア，通信ライブラリ，並列実行時ライブラリなどで構成されるソフトウェアパッケージの名称であり，本研究の過程で開発されたソフトウェアが含まれている．ここでは，特にソフトウェアについてその構成や機能の概要について述べ，同時にそれらの基本性能を示す．第 4 章では，本研究のひとつの柱であるギャングスケジューラの実装方式を提案し，それを用いた時分割スケジューリングオーバヘッドや，オーバヘッドの分析を行なう．第 5 章では，並列計算機上で時分割スケジューリングを実現するもうひとつの手法として注目されている非同期コスケジューリング（第 2.2.3 節参照）と，第 4 章で提案されたギャングスケジューリング実現手法との比較を試みる．第 6 章において，開発されたギャングスケジューラにより効率的な対話処理を実現するため，並列プロセスの大域状態の検出方法の提案を行なう．第 7 章では，並列計算機の特徴を考慮し，並列計算機上でのギャングスケジューリングの実践的なジョブスケジューリングへの応用として，時分割と空間分割を組み合わせた時分割空間分割スケジューリング手法として Distributed Queue Tree（DQT）を提案し，シミュレーションによりその特性を解析し，従来提案されているバッチスケジューリングとの性能比較を行なう．

第 2 章

関連技術サーベイ

本章では，本研究のプラットフォームとなったクラスタ技術と並列計算機のジョブスケジューリングの研究動向に焦点を当てサーベイを行なう．

クラスタ技術とは，平易に言えば「既存の技術を組み合わせて並列計算機を構築する」ことである．PC の性能や通信ハードウェアの進歩により，従来カスタム部品を用いて構築されてきた並列計算機と同等の性能が，PC クラスタとして実現可能になることが広く知られるようになってきた．クラスタ技術は，ユーザのニーズに合わせ自由にクラスタを構築可能である，価格対性能比が高い，といった利点以外に，ハードウェアからソフトウェアに至る全てにおいて情報が入手可能であり，透明性が高いという特徴がある．これは研究を進める上で重要なポイントである．

並列計算機におけるジョブスケジューリングが逐次計算機のそれと最も大きく異なる点はプロセッサ空間というスケジューリングの対象となる軸が加わることである．この軸を加えたスケジューリングについては，従来の待ち行列理論に基づいた理論構築は未成熟で，研究分野として未開拓な部分が大きい．コスケジューリングは並列計算機上に効率的に時分割スケジューリングを実現するひとつの手法と考えられている．並列プロセスを構成する個々のプロセスのスケジューリングをなんらかの方法で同期させることで，並列プログラムの性能低下を防ぐことが可能とされている．本章では，時分割スケジューリングを対話並列処理に応用した際の問題点についても論じる．

2.1 クラスタ技術

情報処理能力を飛躍的に高める技術のひとつとして並列処理が注目され，最先端の技術とともに商品化されてきた．しかしながら 90 年代の後半に入ってから多くの並列計算機メーカー，例えば Thinking Machines 社や Cray Research 社などは市場から撤退した．スーパーコンピューティング市場は多くのメーカーを受け入れるほど大きくはなかったためである．近年，ワークステーションや PC を高速ネットワークで接続することでクラスタを構成し，価格対性能費に優れた並列計算機を構築しようとする試みが盛んに成りつつある（表 2.1）． CRAY T3-E， IBM SP-2，富士通 AP-3000，日立 SR2201 などの多くの商用並列計算機は，ハードウェア的にはワークステーションクラスタと見做すこともできる．しかしながら，本研究ではカスタム電子部品を用いないで構築されたものをクラスタと呼び区別することにする．カスタム部品を用いないことの利点は，大量生産による部品コストの低減，カスタム部品を開発する期間が不要なことによる開発期間の短縮と，それによる最新の高性能部品が利用可能であること，の 3 点が挙げられる．特に PC 分野におけるプロセッサの性能競争は激しく，毎年倍に近い性能向上を達成している．この結果，開発期間の短縮がそのまま性能に結び付く．また，ニーズに合わせクラスタを自由に構成あるいは拡張できるという利点もある．

表 2.1: 代表的なクラスタ計算システム

開発機関	名称	プロセッサ	ネットワーク	PE 数
ASCI	DAS [DAS]	PentiumPro	Mryinet	64
Hebrew Univ.	- [MOS]	PentiumPro	Mryinet	60
INRIA	- [INR]	PentiumPro	Mryinet	6
NASA/Caltech	Beowulf [Beo]	PentiumPro	FastEther	16
NCSA	HPVM [HPV]	Pentium II	Mryinet	128
RWC	WSC [RWC]	SuperSparc	Mryinet	36
RWC	PCC-I [HT96]	Pentium	Mryinet	32
RWC	PCC-II [手塚 98]	PentiumPro	Mryinet	64
UCB	NOW [ACP+95]	UltraSparc-1	Mryinet	100

表 2.1 に示すように，クラスタの利点は，現在多くの並列処理研究者に認められてお

り，クラスタの構築とその上での並列ソフトウェアの研究開発が世界的に盛んに進められている．本節では，まずクラスタによる並列処理において中核となる通信技術をサーベイし，最後にクラスタ全体をひとつのシステムとして運用することを目的とするソフトウェアに関するサーベイを行なう．

2.1.1 通信技術

並列処理における通信の頻度は分散処理の通信の頻度に比べ桁違いに高いと考えられる．また，最近の通信ハードウェア（表 2.2 に代表的なものを示す）では数 μ 秒から数十 μ 秒程度の通信遅延を実現している．例えば Sun Microsystems 社製の SparcStation20 （クロック周波数 75 MHz）における `getpid()` システムコールは約 5 μsec を要し，通信遅延時間と同等である．このため通信の都度システムコールを発行することのソフトウェアオーバヘッドが問題視されるようになってきた． OS の介在を排除し，システムコールも割込も用いない通信を実現することで，ソフトウェアのオーバヘッドを極限にまで減らし，その分通信性能を向上させようとするユーザレベル通信と呼ばれる研究が注目されている．

表 2.2: 主要な高速ネットワーク製品

ベンダー名	製品名	Bandwidth $[MB/s]$	Host Interface
Myricom.	Myrinet [Myr, BCF+95]	160	S-Bus, PCI
DEC	Memory Channel 2 [Gil96, FG97]	133	PCI
Dolphine	SCI [OP97]	200	S-Bus, PCI

表 2.3 にこれまでに発表された主要なユーザレベル通信システムをまとめる．通信遅延は 10 μsec を切り，バンド幅は 100 MB/s を越えるものがある．これらのユーザレベル通信ライブラリは，いずれもワークステーションや PC （パーソナルコンピュータ）上で動作し，クラスタにおいて高性能な並列処理を実現するのに大きな役割を果たしている．

表 2.3 で用いられている通信ハードウェアは Myrinet [BCF+95] が最も多く，次いで ATM となっている．表 2.3 に示されたいずれの通信ハードウェアもネットワークイン

表 2.3: 代表的なユーザレベル通信ソフトウェア

Name	Network (I/O Bus)	Latency [μsec]	Bandwidth [MB/s]	Virtual Network
AM-II [CMC97]	Myrinet (S-Bus)	21	31	yes
BIP [PT97]	Myrinet (PCI)	4.3	126	no
FM [PKC97]	Myrinet (PCI)	11.5	56.3	no
PM [THIS97]	Myrinet (PCI)	7.2	118	yes
U-Net [WBvE97]	ATM (PCI)	45.5	14.8	yes
VMMC [DBLP97]	Myrinet (PCI)	9.8	108.4	no

ターフェイス上にプロセッサを持ち，そのファームウェアをユーザが開発可能となっている．つまり，高性能なユーザレベル通信を実現するにあたり重要なのは，最下位プロトコル層においてそれぞれ最適化されたことである，という点である．また，表 2.3 のそれぞれのシステムに共通していることは，通信ハードウェア内にプロセッサを持ち，本来は OS が行なうべき通信処理をそのプロセッサに代行させている点である．これにより通信処理は通信プロセッサが対処し， OS への割込を排除し，通信と計算が並行処理されるようになっている．また，多くの実装において，通信ハードウェアのメモリ領域がユーザのプロセス空間にマップされ，通信ハードウェアが送信メッセージをポーリングし，受信メッセージは通信ハードウェアがマップされたメモリ領域に直接書き込み，ユーザプロセスがポーリングする，という設計になっている．こうすることでメッセージのコピーの回数を減らすことが可能であるからである．

ユーザレベル通信を用いてギャングスケジューリングのような多重プログラミング環境を実現するには，ユーザレベル通信においてなんらかの支援機能が必要である（詳細については，第 2.3.1 節を参照のこと）．そのひとつの例として，スケジューラが同期を取るために必要な仮想ネットワーク機能がある．仮想ネットワークとは，物理的にひとつのネットワークを多重化したものである．表 2.3 に示したユーザレベル通信ライブラリにおいて，仮想ネットワーク機能を支援しているものとしては AM-II [CMC97]， PM [THIS97]，そして U-Net [WBvE97] がある．仮想ネットワークにおいては，仮想化された個々の仮想ネットワーク間の独立性が問題となることに注意を要する．例えば，あ

る仮想ネットワークにおいてメッセージが渋滞しているような状況が，他の仮想ネットワークに及ぶことは避けなければならない．特にスケジューラが同期のためにひとつの仮想ネットワークを用い，他の仮想ネットワークをユーザの並列プログラムの計算に用いるような場合には，仮想ネットワークの独立性は重要な要件となる．

2.1.2 クラスタ管理ソフトウェア

ワークステーションや PC を高速ネットワークで接続し，適当なユーザレベル通信ライブラリを実装することで高性能な並列計算機を構築することは可能である．しかしながら，これだけではクラスタは単なるコンピュータの集合であり，単一システムにはならない．例えば，このようなクラスタを用いて並列プログラムを実行しようとすると，ユーザはまずどこかひとつの計算機にログインし，そこから他の計算機に並列プロセスを構成するためのプロセスを生成し，それらが全て走り始めるまで同期を取り，やっと並列計算が可能になる．もし，他のユーザが同時に同じことをしようとしていた場合，排他制御など計算機資源の大域的な管理が必要となる．

このような状況を避けるためには，計算機の集合体であるクラスタを，ひとつのシステムとして見せるためのソフトウェアが必要である．このようなソフトウェア開発のひとつの大きな流れとしては， MPI [GLS94] や PVM [GBD+94] などの上位通信ライブラリや言語の実行時ライブラリなどに，クラスタ管理機能として含める方向がある．もうひとつは通信や言語とは独立したクラスタ管理ソフトウェアである．並列プロセスの生成やスケジューリングなどは基本的に通信や言語とは独立しているため，後者の方式がより自然であると考えられる．

表 2.4 はこのような目的のために開発された主要なシステムを示したものである．特定の通信ライブラリや言語に特化したものは除かれている．これらの大きな特徴として，管理ソフトウェアの実装がカーネルレベルであるのか，ユーザレベルなのかという違いがある．前者の場合，並列プロセス間の保護は完全なものとなる利点があるのに対し，複雑なカーネルを新規に開発しなければならないという欠点がある．表 2.4 中の MOSIX [BGW93] がこれにあたる．一方，後者の方式では，保護は完全ではないが，既存のローカル OS の上に大域的な機能を実装するだけで済むため，開発期間が短くなり，可搬性も高くなる．前述したようにクラスタの大きな利点は，最新のハードウェア技術を利用するために構築期間が短いことであるため，ソフトウェアの可搬性はクラスタの利点を

表 2.4: 多重並列プログラミングを可能とするクラスタ管理ソフトウェア

システム名称	MOSIX	HPVM	SCore-D	Glunix
開発機関	Hebrew Univ.	NCSA	RWC	UCB
実装レベル	kernel	user	user	yser
通信ライブラリ	-	FM	PM	AM
並列プロセス	yes	no†	yes	yes
マルチ並列プログラミング	yes	no	yes	yes
ギャングスケジューリング	no	no	yes	yes

† ひとつの並列プロセスと複数の逐次プロセスのみ可能

活かす意味でも重要である．大域的なカーネルを開発することのもう一つの問題は，どのような大域的な機能が必要であり，どのように実現するのかが良くは分かっていないことである．第 2.2.1 節で述べるように，並列プロセスのスケジューリングだけをとって見ても未解決な問題は山積している．したがって，ユーザレベルで機能面での研究を行ない，その後で大域的なカーネルを開発するといったアプローチが有効と考えられる．

またシステムによっては，敢えてクラスタを複数のワークステーションの集合とし，個々のワークステーション上に逐次プロセス（これは普通のワークステーションの利用法である）と並列プロセスという組み合わせを考慮したものがある（表 2.4 中の HPVM と Glunix）． HPVM では単一の並列プロセスとの組み合わせのみサポートしている [Sob97, PKC97] のに対し， Glunix では複数の並列プロセスとの組み合わせを許している [ACP+95, ADCM98, CMC97]．一方， SCore-D では逐次プロセスを単一プロセッサしか必要としない並列プロセスとして位置付け，並列プロセスのみを扱うものとしている [堀 93a, HTI+96, 堀 96c, HTI97b]．

まとめと問題点の整理

- 市販の通信ハードウェアとワークステーションや PC を組み合わせることで高性能な並列計算機を安価にかつ自由に構築することができる．

- ユーザレベル通信は OS の介在を排除し，その分のオーバヘッドを低減させ，通信性能を向上させようとするものである．

- ユーザレベル通信において多重プログラミング環境を構築するには，仮想ネットワークなどのユーザレベル通信における支援機能が必要となる．

- クラスタを単一のシステムと見せるためには大域的な管理ソフトウェアが必要である．

- 大域的なクラスタ管理ソフトウェアとしては，特定の通信ライブラリや言語に特化しないものが望まれる．

2.2 ジョブスケジューリング

スケジューリングとは，有限の資源を有効に活用するために，複数の資源利用要求の取り扱いのことである．ジョブスケジューリングでは，計算機システムを資源とし，計算機上での計算の要求をスケジューリングする．スケジューリングの目的である資源の活用の有効の度合を示すものとして，以下の指標がある [SPG90]．

利用率（utilization）： 資源の利用率．

スループット（throughput）： 単位時間あたりのジョブ処理量．

ターンアラウンド時間（turnaround time）： ジョブの投入からジョブ終了までの時間．

待ち時間（waiting time）： ターンアラウンド時間内における，他のプロセスが占有している資源の解放を待つのに費やされた時間．

応答時間（response time）： 対話システムにいてジョブが投入されてから最初の応答がユーザの端末に表示されるまでの時間．

多くの場合，利用率とスループットの指標はできるだけ高く，ターンアラウンド時間，待ち時間，応答時間はできるだけ短い方が良いスケジューリングとされる．また，指標ではないが，スケジューリング方式を検討するにあたり，資源要求が無限に待たされるような飢餓状態（starvation）に陥らないようにしなければならない．

2.2.1 並列計算機におけるジョブスケジューリング

逐次計算機におけるジョブスケジューリングでは，プロセスは単一のプロセッサを占有してしまうため，他のジョブはプロセッサ資源の解放を待つために待ち行列に入る．スケジューリングはこの待ち行列の管理方法のことであり，逐次計算機においては時間的なジョブのプロセッサ資源割り当ての順序を規定することである．

並列計算機おいては，ジョブの投入時にそのジョブの実行に必要なプロセッサ資源の量（プロセッサ台数）がユーザにより指定される場合が多い．均質な並列処理システムで，ジョブで指定されたプロセッサ台数がシステムのプロセッサ台数よりも少ない場合，どのプロセッサを割り当てるかという自由度がある．この自由度の多さが並列計算機におけるジョブスケジューリングを複雑なものにしている．

2.2.2 並列計算機用ジョブスケジューリングの分類

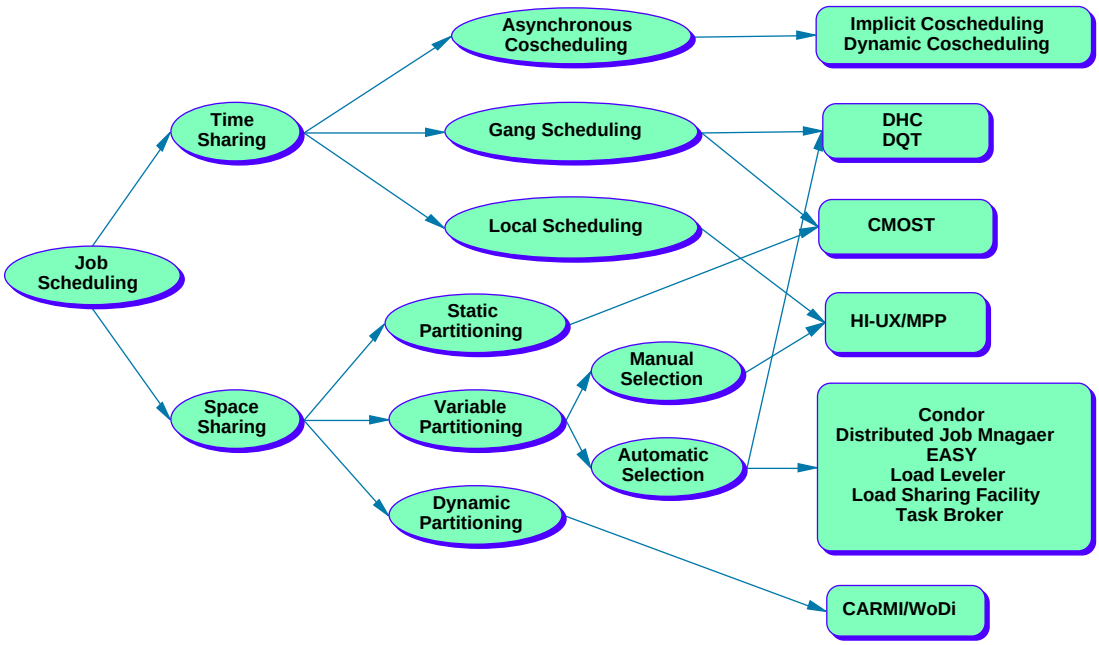


図 2.1: ジョブスケジューリング方式の分類

図 2.1 にこれまでのジョブスケジューリングの研究（商品を含む）の分類の例を示す．分類名称は楕円で示され，具体的なシステムは角が丸い長方形で示されている．文献 [Fei97] にはこの図とは異なった分類が示されている．並列計算機における多重プログラミングは，プロセッサ空間軸で多重化する空間分割（Space Sharing）と，時間軸で多重化する

時分割（Time Sharing）の 2 種類に大別できる．

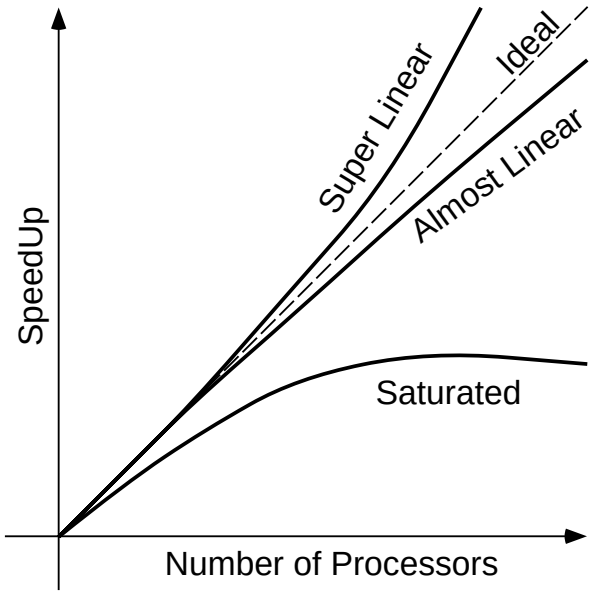


図 2.2: 並列プログラムの台数効果曲線による分類

図 2.2 は，アプリケーションプログラムの並列化による速度向上の度合をモデル化したものである．アプリケーションの並列化による速度向上は，問題を解くための並列アルゴリズム，データおよび並列計算機の特性に依存する．この図ではアプリケーションの並列化による速度向上の度合に応じて，飽和（saturated），線形（almost linear），超線形（super linear）という 3 つの典型的な曲線を示している．速度向上が飽和してしまうようなアプリケーションでは，台数効果曲線のピークを示すプロセッサ台数よりも多くのプロセッサを割り当てるのはプロセッサ資源の浪費となる．超線形のアプリケーションでは，最大のプロセッサを割り当てることがシステム全体のスループットに貢献する．線形に近いアプリケーションではプロセッサ台数がシステム全体のスループットに与える影響は少ないので，ユーザが必要に応じて任意のプロセッサ台数を自由に選択できる．一方，ユーザが必ずしもシステムが提供するプロセッサ台数を必要としないならば，空いているプロセッサの部分集合を他のユーザに割り当てるということは，システム全体のスループットを向上させるという観点から重要である．このようにプロセッサ空間を分割し，分割された部分空間のそれぞれに異なる並列プロセスを走らせる方式を空間分割（Space Sharing）と呼び，どのように空間分割し，どの部分空間に割り当てるかを決定することを「空間分割スケジューリング」と呼ぶ．アプリケーションプログ

ラムに割り振るプロセッサ台数は，アプリケーションの性質，システム全体のスループットあるいはユーザの希望に応じて空間分割が可能であることが望まれる．

図 2.3 には 64 プロセッサから成る空間を 5 つの並列プロセスが共有している例を示す．空間分割は，分割がシステム起動時のパラメタなどにより決定される静的空間分割（Static Partitioning），システム（管理者）が予め定めておいた部分空間やユーザが定めた部分空間を選択する可変空間分割（Variable Partitioning），および，プログラムの実行時に動的に変化する動的空間分割（Dynamic Partitioning）に分類することができる [FR95]．静的空間分割は自由度が他の分割方式に比べ乏しいため，検討の対象から除外する．

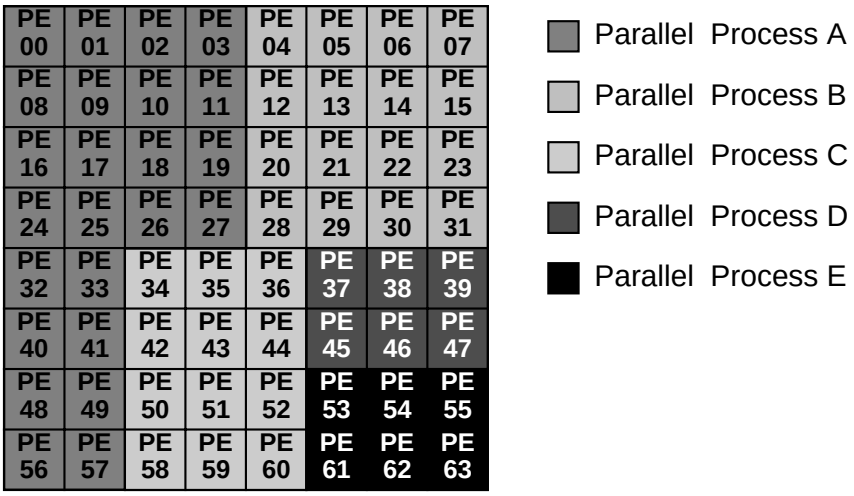


図 2.3: 空間分割の例

可変空間分割においては，投入されたジョブの分割された空間への割り当てを，ユーザが選択する方式と，システムが選択する方式とに分類することができる．前者の場合，空間分割そのものをユーザ自身が行なうものが多い（図 2.1 中 “Manual Selection”）．ユーザは負荷の少ない部分空間を自分で探索する必要があり，ユーザの負担が増える．後者においては，バッチスケジューリングか時分割スケジューリングかにより探索方式が異なる．バッチスケジューリングの場合では，並列プロセスが走行していない部分空間をシステムが探索する．時分割スケジューリングの場合は，システム全体での負荷を均衡させるようにシステムが部分空間を割り当てる場合が多い [FR92, HIK+95, 堀 96d]．

図 2.1 には示されていなが，可変空間分割は分割の方針により，通信ネットワークのトポロジを考慮して部分空間内の通信の影響が他の部分空間に及ぼさないようにする方

式 [CS87, CS90, LC91, Zhu92, CT92, QN95]，トポロジとは独立に空間分割によって生じる断片化の影響を最小にしようとする方式 [FR92, KLDR94, HIK+95, 堀 96d]，トポロジ的に隣接しない部分空間を仮想的にひとつの部分空間にする方式 [LLWN94]，の 3 つに分類できる．部分空間内の通信が互いに干渉する度合は，通信性能やアプリケーションの通信パターンにより異なる [LLWN94, QN95].

空間分割スケジューリングにおいては空いているプロセッサ空間の断片化によるプロセッサ利用率の低下をできるだけ避けるような方式が望まれる．プロセッサ空間断片化の問題は動的な記憶域管理方式の問題に似ている [Ous82]． DHC （Distributed Hierarchical Control） [FR90b, FR90a, FR96] および DQT （Distributed Queue Tree） [HIK+95, 堀 96d] は，記憶域管理方式のひとつとして広く用いられているバイナリバディ（binary buddy）方式 [Knu68] を用いて断片化の影響を抑えている．バイナリバディ方式は，現在実現されている通信ネットワークのトポロジの大半において通信の相互干渉を避けることが可能である，という利点もある．

動的空間分割は，並列プロセスの並列度が動的に変化するのに応じてプロセッサ資源を動的に割り当てようとするものである．動的空間分割は資源管理の立場からは自然な発想であるが，ユーザプログラムはプロセッサ資源の変化に対応する必要がある． Condor ワークステーションクラスタ上の PVM 通信ライブラリとその資源管理システム CARMI/WoDi では，タスクバグ（task bag）のようなプログラミングスタイルのアプリケーションプログラムに対し，動的空間分割によるマルチプログラミング環境を提供する [PL95]．動的空間分割が実際に有利かどうかは空間分割の動的な変更に要するコストに依存する [IPS96].

まとめと問題点の整理

- 空間分割はその方法により，静的，可変，動的の 3 種類に分類できる．
- 静的空間分割ではシステムの運用に柔軟性を欠く．
- 空間分割ではプロセッサ空間の断片化の影響を考慮しなければならない．

2.2.3 並列計算機における時分割スケジューリング手法

図 2.4 は，時分割スケジューリングにおける問題点を示したものである．簡単のため，2 プロセッサ上に A, B, C の 3 つの並列プログラムが時分割により同時に走行しているとする．それぞれのプロセッサ上では独立に Unix に準じたプロセススケジューリングが

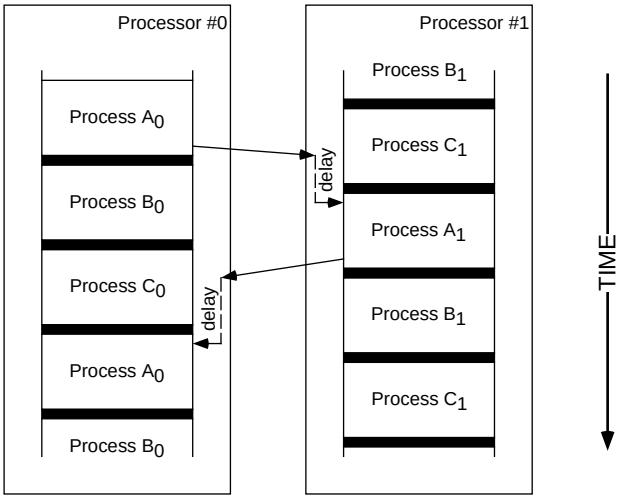


図 2.4: ローカルスケジューリングの問題点

行なわれるものとする．並列プログラム A の第 i プロセッサ上でのプロセスを A_i と表記することにする．ここで並列プロセス A の通信に着目する．プロセス A_0 からプロセス A_1 に対し発せられた通信メッセージの受信は，図に示したようにプロセス A_1 がスケジューリングされるまで待たされる可能性が生じる．その結果，みかけの通信遅延が増大し，並列計算速度の低下を招く．このようなスケジューリング方式を本研究ではローカルスケジューリングと呼ぶことにする．

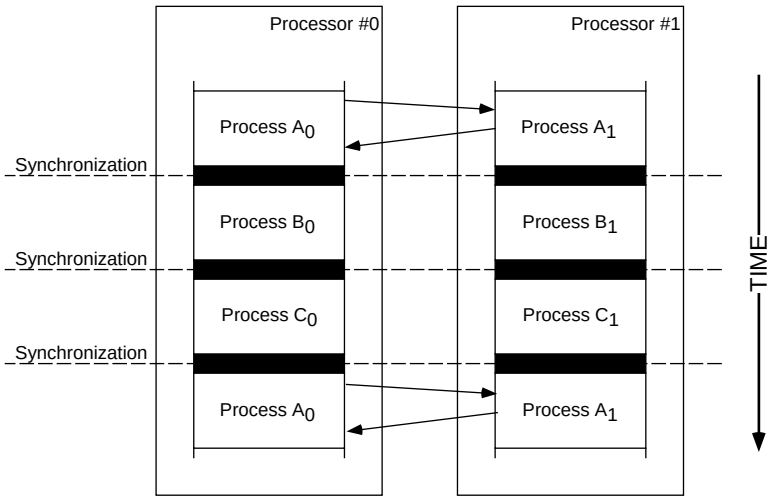


図 2.5: ギャングスケジューリング

Ousterhout は，図 2.4 に示したローカルスケジューリングの問題を要求時ページン

グの問題に例え，プロセススラッシング（process thrashing）と呼び，その回避策としてコスケジューリング（coscheduling）を提案した [Ous82][1]．コスケジューリングはスケジューリング時に計算に関与しているプロセス群のみの切替えを意味する．このプロセス群のことを「プロセスワーキングセット」と呼んでいる [Ous82]．この用語は要求時ページングに対する比喩からすれば自然な発想である．しかしながら，文献 [Ous82] には具体的なプロセスワーキングセットを求める方法については言及していない．また，プロセスワーキングセットがどの程度時間的に安定に存在するか，アプリケーションに依存するかしないか，などといった疑問点は残されたままである．

プロセッサスラッシングの問題は，図 2.5 に示したように並列プロセス A に関与するプロセスを全プロセッサで同期してスケジューリングすれば回避可能である．このスケジューリング方式は現在ギャングスケジューリングと呼ばれている [Fei97]．ギャングスケジューリングはプロセスワーキングセットの存在を無視し，常に全てのプロセッサのスケジューリングを同期させる．

OS が介在する通信（OS レベル通信）の場合，受信メッセージの待ちでプロセスの実行が中断し，メッセージの到着によりプロセスの実行が再開するブロッキング受信が可能となる．通信する頻度が高い並列プロセスでは，その並列プロセスを構成するプロセス群は互いに，ローカルスケジューリングにおいてもスケジューリングが自然と同期する可能性がある．一方，頻繁に通信しない並列プロセスでは他のプロセスとの依存関係が弱いことを意味し，他のプロセスと同期してスケジューリングする必要性が少ない．送信元のプロセスではメッセージの往復時間（これは通信遅延時間の 2 倍に相当）に相手先での処理時間を加味した時間だけ受信メッセージをポーリングし，その後に OS 経由の通信によりブロックして待つ，という方式によりコスケジューリングを実現しようとする方式がある．このように，明示的にスケジューラの同期を取らない方式としては，Dynamic Coscheduling [Sob97] や Implicit Coscheduling [DAC96, ADCM98] が提案されている．本研究ではこれらを総称して「非同期コスケジューリング」と呼ぶことにする．この呼称にならえばギャングスケジューリングは同期コスケジューリングとも呼ぶことができる．

非同期コスケジューリングでは，Unix のような時分割スケジューリングが各プロセッサの OS で実現されていることを仮定している．非同期コスケジューリングにおいて，

[1] これがギャングスケジューリングのアイデアの最初の提案とされている [Fei97].

どのように「コスケジューリング」が実現されるのだろうか．ここでふたつの並列プロセスが同時に走行している状況を想定する．ここでひとつの並列プロセスを構成するあるプロセスに着目する．このプロセスは他のプロセスからのメッセージを待っているが，メッセージ送信側の相手プロセスはスケジューリングされていないとする．この場合，相手プロセスはローカルな時分割スケジューリングの方策により決まるある一定時間だけ処理が中断されている．もし，設定されたポーリング時間が時分割間隔よりも短い場合は，メッセージを待っているプロセスもポーリング時間後にブロックし，その結果プロセスがスケジューリングされる．一方，もし相手プロセスがスケジューリングされていた場合は，メッセージはすぐに到着し，受信プロセスもブロックすることはない．結果的に通信し合うプロセス群は，明示的な同期がなくても，互いにスケジューリングが同期する可能性がある．もし，このふたつのプロセスが局所計算に忙しく互いに通信し合うことがなければ，スケジューリングは同期しないが，そもそもこのような場面では同期する必要がない．しかしながら，同時走行する並列プロセスの数が 3 つ以上になった場合は，状況はより複雑になり，スケジューリングがうまく同期するかどうかは不明である．

文献 [GTU91, FR92] では細粒度の並列プログラムにおいてギャングスケジューリングが有効であることを示しているが，文献 [ADV+94] では粗粒度かつ負荷の不平衡が大きな場合に非同期コスケジューリングがスループットの面で有効であると報告している．本節では，これまで並列計算機上では並列プロセスのみが走行するとしてきたが，クラスタ上では，普通の逐次プロセスも走行する可能性がある． Sobalvarro は， I/O を伴うような複数の逐次プロセスとひとつの並列プロセスのスケジューリングにおいて，非同期コスケジューリングが有効であると主張している [Sob97].

非同期コスケジューリングにおいては，ポーリング時間をどのように決めるかが大きな焦点になる．これは通信遅延時間，並列プログラムの通信パターン，ローカルスケジューラのプロセス切替時間などに依存すると考えられている [ADV+94]．非同期コスケジューリングがギャングスケジューリングに対し明らかに有利なのは，同時走行する並列プロセスの負荷にバラツキがあり，かつ負荷のバラツキが並列プロセス間で相関がない場合に，トータルなスループットの向上が見込める点である．データ並列的に記述された並列プログラムでは，速度向上のために負荷を平衡させるように記述する．このためデータ並列プログラムに対し非同期コスケジューリングの効果は薄いと考えられる．この点に関しては，第 5 章でギャングスケジューリングと非同期コスケジューリングの比較を

試みる．

まとめと問題点の整理

- 独立したローカルな時分割スケジューリングだけでは並列プログラムの実行性能を著しく損なう可能性があり，協調的にスケジューリングする枠組が必要がある．
- 並列プロセス全体を同期させて時分割スケジューリングする方式をギャングスケジューリングと呼び，ギャングスケジューリングでは原理的に並列プログラムの実行性能を損なうことはない．
- 通信メッセージをブロックすることで暗黙的にローカルスケジューリングをある程度同期させることができる．これを非同期コスケジューリングと呼ぶ．

2.2.4 時分割スケジューリングと空間分割スケジューリング

空間分割スケジューリングは，空間軸におけるスケジューリングであるが，実際には First-Come Fisrt-Served（FCFS）などといった従来のバッチスケジューリング方式と組み合わされている．逐次計算機におけるバッチスケジューリング方式では，空間軸がない，時間軸のみにおけるスケジューリングである．並列計算機のジョブスケジューリングは，基本的に空間と時間の直交する 2 種類のスケジューリングから構成される．

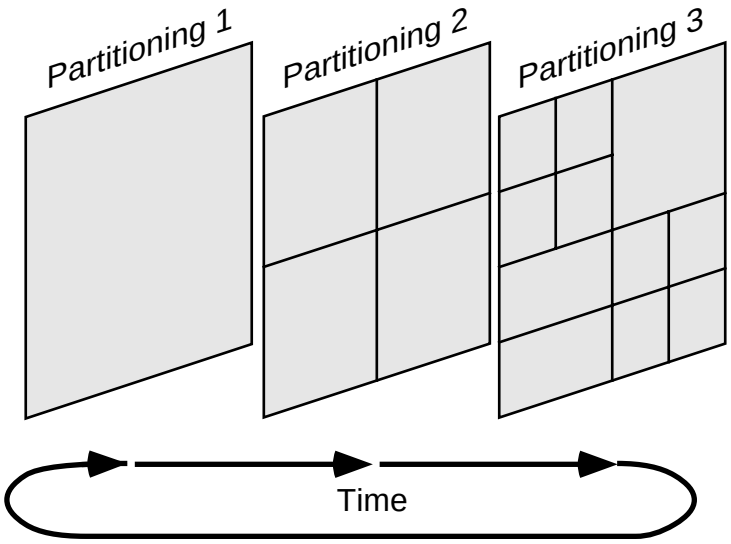


図 2.6: 時空間分割スケジューリングの概念図

時間軸のスケジューリング方式として時分割スケジューリングを用いることも可能である．空間分割と時分割を組み合わせたジョブスケジューリング方式を本研究では「時空間分割スケジューリング（Time-Space Sharing Scheduling:TSSS）」と呼ぶことにする（図 2.6） [堀 93b, 堀 93a, HIK+93][2]．時空間分割スケジューリングの例としては DHC [FR90b, FR90a, FR96] および DQT [堀 94b, 堀 94c, HIK+95, 堀 95, HIN+95, 堀 96d] がある．

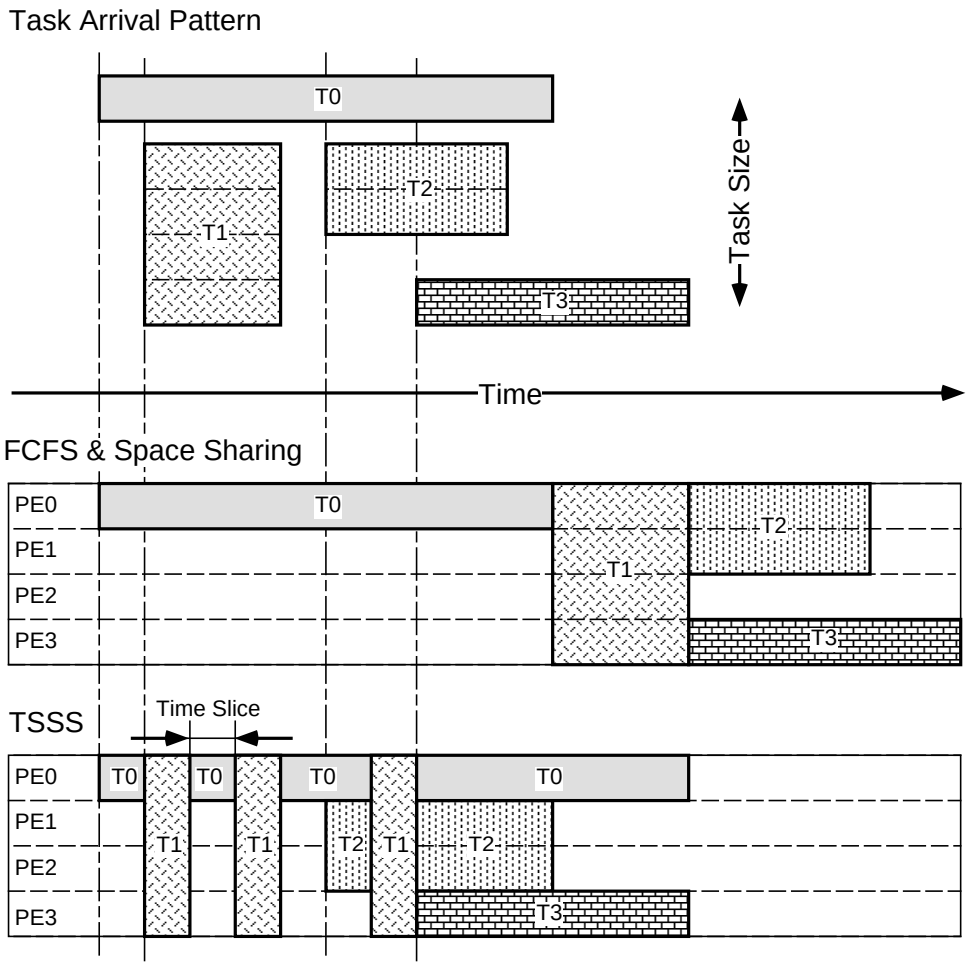


図 2.7: 時空間分割と空間分割のスケジューリング例

時分割と空間分割を組み合わせることで，空間分割により生じる断片化の影響を低減することが可能である [Fei95]．図 2.7 は，あるジョブ投入パターン（図中，$T0, T1, T2, T3$ という 4 つのジョブが図に示された時刻，順序で投入される）におけるスケジューリン

[2]文献 [堀 95, 堀 96d] では「時分割空間分割スケジューリング」としていたが，本研究では「時空間分割スケジューリング」と改めた．

グを，空間分割でかつ FCFS によるスケジューリングの場合と，時空間分割スケジューリングの場合で比較したものである．それぞれのジョブの実行には図の縦方向の長さに相当するプロセッサ数が必要であるものとし，実際のスケジューリングにおける並列計算のプロセッサ数は 4 である．FCFS 空間分割スケジューリングでは，ジョブ $T0$ の投入の後，4 プロセッサを必要とするジョブ $T1$ が投入されるが，ジョブ $T0$ が終了するまでジョブ $T1$ はスケジューリングされない．一方，時空間分割スケジューリングでは，時分割によりジョブ $T1$ の実行が投入と同時に開始されている．この結果，時空間分割スケジューリングでは，FCFS 空間分割スケジューリングよりも全てのタスクの実行が終了するまでの時間を短くすることができ，プロセッサの利用率もその分優れている．

図 2.7 に示した FCFS によるバッチスケジューリングにおける問題は広く認識されており，バッチスケジューリングによる解決方法もいくつか提案されている．ここでは代表的なものとして Backfilling [Lif95] と Scan スケジューリング [KLDR94] を取り上げる．

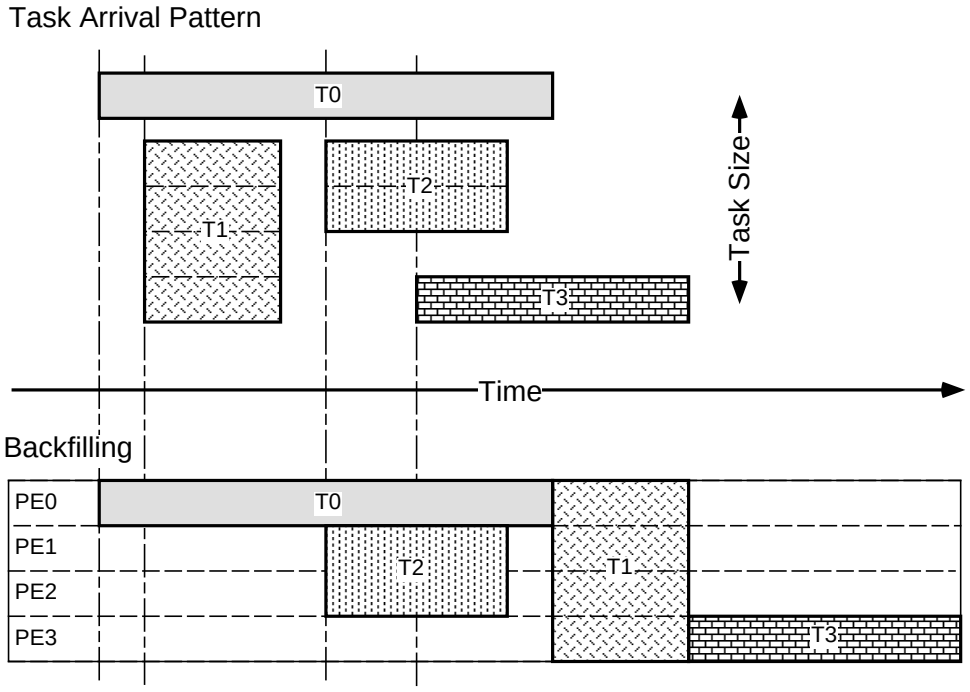


図 2.8: Backfilling によるスケジューリングの例

図 2.7 と同じジョブ投入パターンでの Backfilling によるスケジューリングの例を図

2.8 に示す．Backfilling スケジューリングでは，待ち行列は 1 本であり，基本的に は FCFS で待ち行列内のジョブをスケジュールする．次にジョブ $T1$ が到着するが，これは既にジョブ $T0$ があるためスケジューリングできないで待ち行列に入る．ここまでは，FCFS の場合と全く同じである．ジョブ $T2$ が投入された時に FCFS とは異なる挙動を見せる．ジョブ $T2$ は 2 プロセッサしか必要としないので，待ち行列にあるジョブ $T1$ を追い越してスケジューリング可能である．また，このようにジョブ $T1$ を追い越してもジョブ $T1$ の実行開始時刻は変化がない．Backfilling では，このような条件を満たす時に限り，待ち行列内の順序を崩してジョブをスケジュールする．ジョブ $T3$ に関してはこの条件を満たさないため，ジョブ $T1$ を追い越すことはない．ここで待ち行列にあるジョブの実行開始時刻を遅らせない，という条件は飢餓状態を防ぐために設定されたものであるが，このために Backfilling では事前にジョブの実行時間をスケジューラが知っている必要がある．Backfilling を目的にジョブの処理時間を事前に推測しようとする研究もあるが [Dow97, Gib97]，一般にはジョブ投入時に正確にジョブの処理時間を予測することは難しい問題であり，Backfilling の最大の欠点となっている．

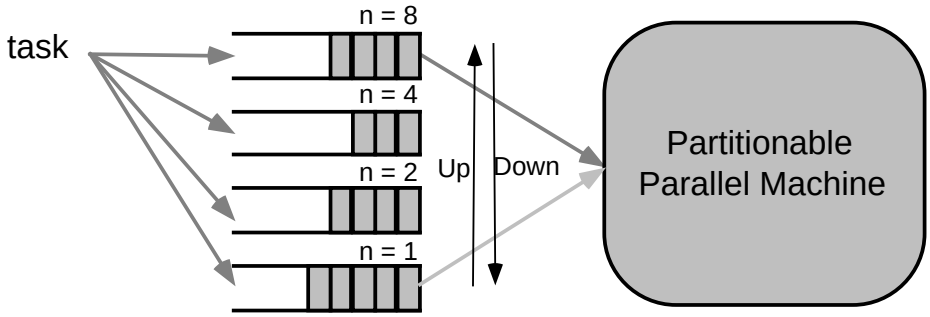


図 2.9: Scan スケジューリングの例

図 2.9 に Scan スケジューリング方式 [KLDR94] の例を示す．ここでは簡単に，8 プロセッサから成る空間分割可能な並列計算機の場合を例として説明する．Scan スケジューリングでは複数の待ち行列を持つ．ジョブが必要なプロセッサの数に応じて入る待ち行列が決まっている．図 2.9 の例では，待ち行列は全部で 4 本あり，それぞれ 1, 2, 4, 8 とプロセッサ数が決まっている．ここでは簡単のため，投入されるジョブが必要とするプロセッサ数は 2 のべき乗とする．待ち行列は大きいもの，あるいは小さいものから走査され，待ち行列が空でなければその待ち行列にあるジョブをスケジュールし，空になったら次の待ち行列を移行する，という方式である．待ち行列を移動する方向に

より ScanUp, ScanDown という 2 種類のスケジューリング方式が提案されている．この方式には飢餓状態の可能性があるが，文献 [KLDR94] では飢餓状態を防ぐスケジューリング方式も提案されており，オリジナルに比べ若干の性能低下があると報告されている．基本的に Scan スケジューリングは，連続して投入するジョブの大きさ（必要とするプロセッサ数）を揃えることで，図 2.7 に示した問題を回避しようとするものである．

まとめと問題点の整理

- 並列計算機のジョブスケジューリングは，空間軸のスケジューリングと時間軸のスケジューリングの組み合わせである．
- 時空間分割スケジューリングは，空間分割による断片化の影響を低減させることができる．
- 空間分割における断片化によるプロセッサ利用率の低下はバッチスケジューリングによってもスケジューリングを工夫することで回避可能である．

2.2.5 時空間分割スケジューリングの研究事例：DHC

DHC はギャングスケジューリングをベースとした時空間分割スケジューリング方式である [FR90b, FR90a, FR96]．図 2.10 は DHC におけるユーザ並列プロセスの制御構造を示す．DHC は分散された 2 分 X ツリー構造であり，ツリー構造の構成するノードは “Controller” と呼ばれ，リーフはプロセッサに対応している．DHC において並列プロセスという概念は明確ではなく，controller はツリーのリーフ方向に向かって制御を伝搬することで並列プロセスを構成する個々のプロセスを制御している．レベル i に位置する controller は，2^i 個のプロセッサ群を管理する．

DHC における分散ツリー構造は，プロセッサ空間のバイナリバディによる分割を表現している．このため，あるプロセッサ数を必要とする並列プロセスは，プロセッサ数で決まるツリー構造のレベルに属するどこかの controller で管理される．X ツリー構造となっているのは，ツリー構造を横断的に負荷バランスを取り易くするためである [FR90a]．一般にバイナリバディ方式では断片化による利用率低下が問題となる [PN77]．このため DHC ではバイナリバディによる分割を拡張し，特殊な管理方式を導入することで断片化による効率低下を防ぐ工夫がされている．これを selective disabling と呼んでいる [FR90a, FR96].

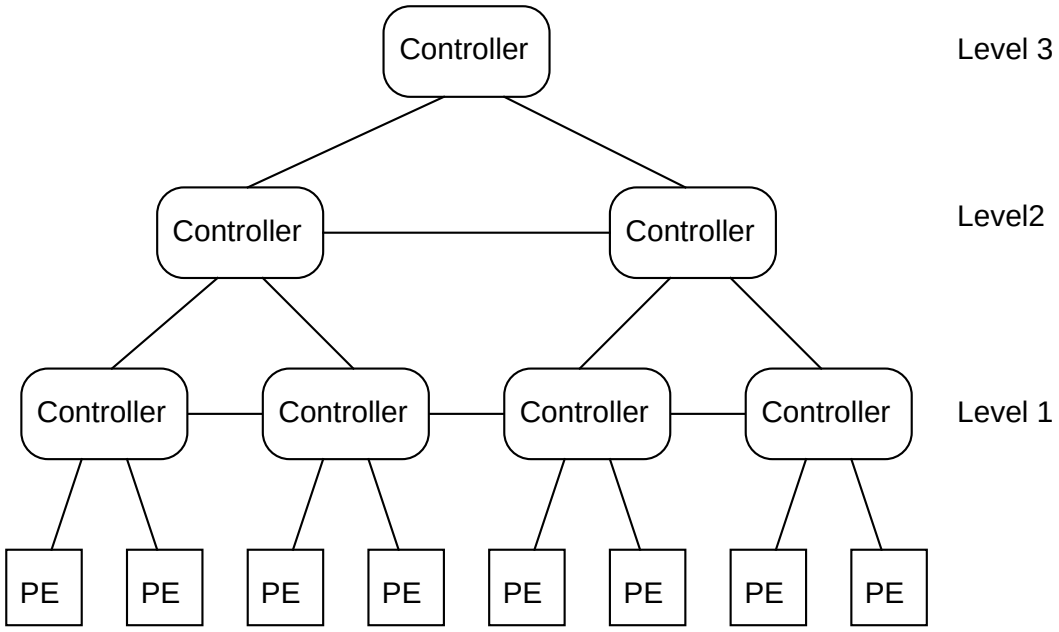


図 2.10: DHC （Distributed Hierarchical Control）の例

文献 [FR96] では DHC における空間分割でのプロセッサ利用率について詳しく解析されている．これによれば，オフラインでの処理，つまり投入されるジョブの集合が事前に判明していると仮定し，バイナリバディによる分割の場合，ジョブが必要とするプロセッサ数に偏りがない一様分布の場合に， 75 % の利用率であることが示されている．また，ジョブが必要とするプロセッサ数に正比例するようなジョブの投入分布の場合，約 72 % の利用率であるとしている．さらに selective disabling 方式を用いた場合，一様分布で 80 %，正比例の分布で 90 % であるとしている．これらの結果はオフラインでの計算なので，投入されるジョブの集合が事前に分からないオンラインの場合は，これより若干利用率が低下するものと考えられる．

時分割スケジューリングは， DHC 全体の時分割間隔が決まっており，それぞれのレベルの controller において自分の持ち分の時分割間隔を自分が管理すべきプロセス群に割り当て，残りは下位の controller に委ねるという方式で実現されている [FR96]．また，断片化が生じているレベルをスケジューリングする際，プロセッサ利用率の低い部分はより短い時分割間隔，利用率の高い部分はより長い時分割間隔とすることで，全体としてのプロセッサ利用率を向上させようという狙いである．

まとめと問題点の整理

- DHC は，バイナリバディによる空間分割スケジューリングと，ギャングスケジューリングを仮定した時分割スケジューリングを同時に行なう時空間分割スケジューリングの一種である．

- DHC において並列プロセスの概念は明確でなく，並列プロセス全体と，それを構成する個々のプロセスは，同じ分散ツリー構造を用いて制御される．

- バイナリバディによる空間分割で，オフラインでスケジューリングを行なった場合，ジョブが必要とするプロセッサ数にジョブ数が一様な頻度分布の場合，理論的なプロセッサ利用率は 75 % であり，正比例の頻度分布では約 72 % の利用率になる．

2.3 ギャングスケジューリング

空間分割を伴ったギャングスケジューリングは，並列計算機をプロセッサ空間および時分割のタイムスロットで分割された仮想並列計算機をユーザに提供する．ユーザプログラムは与えられた仮想並列計算機内で，アルゴリズムや問題に応じて（スレッドなどの）スケジューリングをプログラムすることができる．ユーザ並列プログラムでは，割り振られた仮想並列計算機の中で並列プログラム自身がスケジューリングを陽に記述する必要がある．ジョブスケジューリングとは実並列計算機と仮想並列計算機のマッピングであり，仮想計算機とジョブ（あるいはスレッド）とのマッピングは「タスクスケジューリング」と呼ばれている．本研究はジョブスケジューリングのみを対象とし，タスクスケジューリングは本研究の範囲外とする．

ギャングスケジューリングが提供する仮想並列計算機は，原理的には（ネットワークトポロジーに起因する通信の干渉や，キャッシュなどの影響を無視すれば）他の並列プロセスに与えられる仮想並列計算機と互いに干渉することがない．このため，ユーザは，与えられた仮想並列計算機を用いて性能チューニングをすることが可能である．一方，前節で説明した非同期式コスケジューリングの場合には干渉が避けられない．このため他の並列プロセスの有無が性能に影響を及ぼす可能性がある．

ここで今一度ギャングスケジューリングの意味を考えてみることにする．文献 [GTU91, FR92] においてギャングスケジューリングが有効とされる理由の一つは，あるプロセスがビジーウェイトで他のプロセスの応答あるいはロック解除を待っている場合に，応答

を返すプロセスがスケジューリングされていないとビジーウェイトから長期間抜け出せない，というものである．ユーザレベル通信において受信メッセージはポーリングで待つ．この意味でユーザレベル通信はギャングスケジューリングと相性が良い．

2.3.1 ギャングスケジューリングの実装方式

表 2.5 に今までにメッセージ通信型並列計算機上に実装が報告されたギャングスケジューラの一覧を示す．この表では，ギャングスケジューリングの実装において，*i)* 通信が OS 経由なのかユーザレベル通信なのか，*ii)* 横取り可能（preemptive）かそうでないか，*iii)* ギャングスケジューリングのためのハードウェア支援機構を用いているかいないのか，という 3 点に着目している．

表 2.5: ギャングスケジューラの実装例

System Name	Platform	Comm. Level	Pre-emptive	Hardware Support
(anonymous)[FR92]	Makbilian	OS	Yes	Yes
CMOST[Thi92a]	TMC CM-5	User	Yes	Yes
Medusa[OSS80]	Cm*	OS	Yes	Yes
Meiko CS-2	Meiko CS-2	OS	Yes	N/A
OSF-1 AD[ZRB+93]	Intel Paragon	OS	Yes	No
SCore-D[HTI+96]	Clusters†	User	Yes	No
SHARE[FPR96]	IBM SP-2	User	No	No

† Myrinet で接続されたワークステーションあるいは PC クラスタを指す

ユーザレベル通信とは基本的に OS が提供する保護機構やマルチプログラミングに起因するオーバヘッドを排除しようとするものである．OS レベル通信の場合，ギャングスケジューラは通信を特別に扱う必要がない．OS がプロセスと通信メッセージを関連付けることが可能であり，通信が許されていないプロセスに対しメッセージを送ることを禁止できる．ユーザレベル通信の場合，プロセス切替のタイミングにより，切替える前に走行していたプロセスが受けとるべきメッセージを切替後のプロセスが受けとる可能性がある．また，通信ハードウェアの操作がアトミックでない場合にハードウェアの状態をリセットする，あるいはプロセス切替のタイミングでハードウェア状態を退避

復帰するといった手順が必要になる．任意のタイミングで通信ハードウェアの状態をリセットあるいは退避復帰が不可能であるようなシステムでは，通信手順の一部に危険区域を設定し，その内部を走行している場合はプロセス切替を禁じているものもある [FPR96]. このような危険区域の設定は，プロセスを任意の時点で強制的に終了させることを不可能にするため望ましくない.

別な問題として，ユーザレベル通信を用いたギャングスケジューリングの実装では，どのように個々のプロセッサ上のスケジューラを同期させるか，という別の問題が生じる．もし同期に通信が必要であるとすると，スケジューラはユーザ並列プロセス内の通信とは独立に通信をする必要が生じる．ユーザレベル通信の通信機構は基本的にマルチユーザのための仮想化がなされていないため，スケジューラ間の同期のための通信ができない．SHARE [FPR96] では，NTP（Network Time Protocol）により各プロセッサの時計が同期していることを利用して，スケジューラを時刻により同期させている．この方法では，NTP による時刻合わせの精度が問題になり，また非同期的な事象に対応することが難しくなる.

この問題に対する最もナイーブな対処方法は，スケジューラ用とユーザ並列プロセス用を合わせてふたつのネットワークを持つことである．物理的にふたつのネットワークを持たなくても，ソフトウェアによりひとつのネットワークハードウェアを多重化し，多重化された仮想的なネットワークをスケジューラとユーザ並列プロセスのそれぞれに割り当てる方法が考えられる．ユーザレベル通信ライブラリ PM [THIS97] や AM-II [CMC97] はこのような仮想ネットワークをサポートしている．しかしながら，ユーザ並列プロセスは誤って自分に割り当てられた仮想ネットワークを閉塞させる，つまりメッセージをネットワーク中に詰まらせる，可能性がある．このような状態に陥ってもスケジューラはユーザ並列プロセスを正常に制御できなければならない．従って，仮想ネットワークを用いる場合，ひとつの仮想ネットワークの閉塞状態が他の仮想ネットワークに及ばないように設計されなければならない.

Makbilian 上に実現されたギャングスケジューリングや，CMOST [Thi92a]，Medusa [OSS80] では特殊な通信ハードウェアがギャングスケジューリングを支援している．Makbilian で用いられているネットワークや Medusa が実現された Cm* のネットワークには，割込（あるいはシグナル）をブロードキャストする機能がある．これによりギャングスケジューリングの同期の問題を解決している．CMOST は CM-5 用の OS である．CM-5 では “All-Fall-Down” と呼ばれる機構がネットワークに備わっている．特殊レジ

スタを操作することでネットワークは All-Fall-Down モードになり，その時点でネットワーク中に存在するメッセージは全て，その宛先に関係なく，（ハードウェア的に）最寄りのプロセッサノードに転送される（図 2.11）．プロセス切替時にはネットワークを All-Fall-Down モードにし，各プロセッサ上では All-Fall-Down の結果として到着したメッセージをメモリに退避する．新しいプロセスが起動されると，退避されたメッセージは再びネットワークに注入される．この方法では，メッセージの順序を保存することができないが， CM-5 のネットワークでは動的ルーティングを行なっているので，メッセージ順序は通常の通信においても保存されないため，特に問題になることはない．

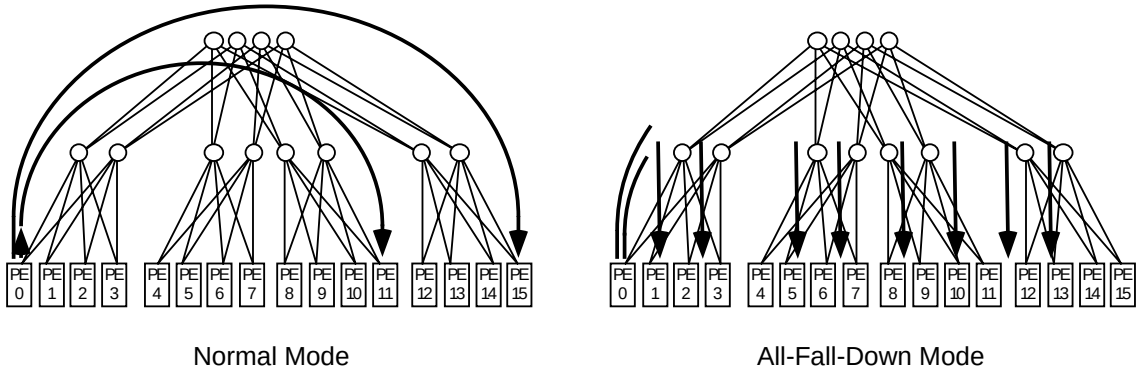


図 2.11: CM-5 の All-Fall-Down

ギャングスケジューリングのためのハードウェア支援機構には，柔軟な空間分割スケジューリングを対象に想定していないものがある．例えば CM-5 では空間分割を変更するのにはシステムを立ち上げ直す必要がある．前節 2.2.2 で述べたように，これまでに様々な空間分割方式が提案されており，どれも一長一短である．従って，現時点で空間分割方式を決めることは難しい．この結果，ギャングスケジューリングを支援するためのハードウェアに対する要求仕様を決めることは容易ではない．一方，ギャングスケジューラを実装する立場からは，特定のハードウェア支援機構を仮定することは可搬性を損なう結果となる．

ちなみに（分散）共有メモリ型計算機では， SiliconGraphics 社 Origin 2000， BBN 社 TC2000 [GW95] などにおいてもギャングスケジューリングが実装されている．メッセージ通信ではない通信，つまりメッセージにアドレスが付随しており，メッセージ本体は付随するアドレスに格納されるタイプの通信，ではメッセージ毎に付随するアドレスにより受信プロセスの識別が可能であるため，通信に OS の介在がなくてもメッセー

ジが誤って別なプロセスに配送される心配がない．ただし，アドレスが論理アドレスの場合は，プロセス ID に対応する識別子が別途必要となる． SiliconGraphics 社の Cray T3-E においてもギャングスケジューラが実装されているが，メッセージにアドレスが付随しているため，表 2.5 からは除いてある．

メッセージ通信においてもメッセージ毎にプロセス識別子を設けることで，上記の問題を回避可能ではあるが，ユーザレベル通信を行なう場合は正しいプロセス識別子を誰が付加するかという問題が生じる．悪意のユーザが間違ったプロセス識別子を付加することで，他のユーザの並列プロセスにメッセージを送りつけることが可能になってしまう．このような状況はマルチユーザのプログラミング環境としては避けなければならない．

仮に，なんらかの方法で全てのメッセージに正しいプロセス識別子を設定できたとし，十分な数の仮想ネットワークが提供できたとする．この場合，同時に時分割スケジューリングする全ての並列プロセスのそれぞれに別な仮想ネットワークを割り当てることができれば，後は並列プロセスの実行を制御するだけでギャングスケジューリングは実現可能である．つまり仮想ネットワーク資源は無限大にあるとの仮定のもとに，仮想ネットワーク資源の管理は不要になり，プロセッサ資源にのみ着目すれば良い，という考え方である．しかしながらこの方式にも欠点がある．ユーザ並列プロセスが死滅した際，その並列プロセスの占有していた仮想ネットワークを再利用できるようになるのはいつか分からない，という問題がある．仮想ネットワークが再利用可能になるのは，死滅した並列プロセスのメッセージが仮想ネットワークに存在しなくなった時である．この状態を検出することができない限り，正しく仮想ネットワークを再利用することはできない．

まとめと問題点の整理

- ビジーウェイトやポーリングによる待ちの実現方式はギャングスケジューリングとの相性が良い．

- ユーザレベル通信用いたギャングスケジューリングの実装では， *i)* スケジューラの同期方式， *ii)* 並列プロセス単位での通信メッセージの分離機構， *iii)* ハードウェア支援機構の有無，に着目する必要がある．

- ギャングスケジューリングのハードウェア支援機構では，スケジューリングソフト

ウェアの可搬性や，プロセッサ空間分割方式と合わせて検討されるべきである．

- 仮想ネットワーク機能をサポートするユーザレベル通信では，スケジューラの同期の問題を解決する手段を与える．

- 仮想ネットワークを実現する際，ひとつの仮想ネットワークの閉塞状態が他の仮想ネットワークの及ばないようにしなければならない．

2.3.2 ギャングスケジューリングと対話並列処理

逐次処理におけるバッチスケジューリングに対する時分割スケジューリングの利点は，*i)* 短い応答時間，*ii)* 対話処理が可能，という 2 点である．長い時分割間隔は時分割スケジューリングのこれらの利点を損なう．一般に時分割スケジューリングの時分割時間は，スケジューリングのオーバヘッドと応答性のトレードオフにより決められる．対話処理においては概ね 1 秒以下の時分割時間が望まれる．典型的な逐次処理 OS としての Unix では，プロセスは I/O などのブロックするシステムコールの発行時，あるいは定期的な優先度計算により，プロセス切替が発生する．システムコールでブロックされたプロセスは，そのシステムコールの完了とともに再びスケジューリングの対象となる．この結果，Unix では対話処理の応答性の良さとプロセッサ利用率の向上を両立させている [Bac86, LMKQ89].

ギャングスケジューリングにおける並列プロセスとは，スケジューリングの単位である．並列プロセスはプロセスの集合である．ここで，プロセッサの利用率に着目し，システムコールのブロックがどのように影響を受けるかについて考えてみる．例えば，128 プロセッサで動いている並列プロセスを構成するひとつのプロセスがシステムコールでブロックしたとしても，他の 127 個のプロセスは動き続けていることになり，全体として 1 % 以下のプロセッサ利用率低下にしかならない．同じことが 2 プロセッサで動く並列プロセスで生じた場合は，半分の利用率低下になる．並列プロセスにおけるブロックしたプロセスの割合に閾値を設け，それを越えた場合に並列プロセス全体をスケジューリングしないという戦略が考えられよう．しかしながら，現在のところ対話並列処理の実体が明らかになっていないので，閾値の具体的な値に関する議論はできない．全てのプロセスがブロックした場合は，明らかにその並列プロセスはアイドル状態であり，他のアイドル状態でない並列プロセスをスケジューリングすべきである．

一方，前節で述べたようにギャングスケジューリングにおいて，ある事象を待つにはビジーウェイトやポーリングが有効であると述べた．これは，並列プロセス全体でみた場合，個々のプロセスの待ちが必ずしもプロセッサ資源を大きく無駄にしないとの理由に依る．このことはスケジューラというユーザ並列プロセスの外部から見た場合，並列プロセスの待ち状態にあることを知ることは簡単ではない事を意味する．仮に個々のプロセスのアイドル状態が検出できたとしても，全てのプロセスがアイドル状態であることを検出する処理は通信を伴い，アトミックに全体の状態を把握することはできない．メッセージを待っているプロセスが次の瞬間にメッセージを受信し，走行状態になる可能性が残る．

ある並列プロセスの全てのプロセスが受信メッセージを待っていて，なおかつネットワークに通信メッセージが存在しない場合，その並列プロセスにおいて有効な計算がこれ以上進まないことを意味する．このような大域的な状況を検出することは「分散プロセスの大域停止検出問題」（global termination detection of distributed processes）として知られている問題である．非同期通信における大域停止問題には既にいくつかの解決方法が提案されている．例えば通信の FIFO 性を仮定してマーカーメッセージを送り，ネットワーク中のメッセージを掃き出す方法 [Mis83, CL85]，送受信メッセージの数を数え上げる方法 [KMY94]，重み付き参照カウントを利用する方法 [六沢 97] などがある．しかしながら，これらのアルゴリズムはユーザ並列プログラムでの処理を必要とする．

並列プロセスを構成する個々のプロセスが待ち状態になるのは，システムコールの完了を待っている場合とユーザレベル通信における受信メッセージを待っている場合だけである．ある並列プロセスが大域的な停止状態にあり，一つ以上のプロセスがシステムコールの完了を待っている場合，その並列プロセスはシステムコールが完了するまで計算が進まないことを意味する．全ての I/O を含むシステムコールが有限時間内に終了すると仮定し．並列プロセスのこのような状態を「並列プロセスのアイドル状態」と定義する．また，未完了のシステムコールが存在せず，ネットワーク中にメッセージも存在せず，かつ，全てのプロセスが受信メッセージ待ちであった場合は「並列プロセスの停止状態」と定義する．この状態には並列プロセスがデッドロック状態に陥った場合も含まれる．並列プロセスの大域的アイドル状態，停止状態，あるいはそのどちらでもない状態を総称して，本研究では「並列プロセスの大域状態」と呼ぶことにする．

ギャングスケジューリングにおいて，プロセッサ利用率と応答性を兼ね備えた対話処

理を実現するためには，並列プロセスの大域状態検出機構が必要である．

まとめと問題点の整理

- 逐次 OS における時分割スケジューリングでは，I/O 待ちの時点でプロセスを切替えることで，プロセッサ利用率と応答性を両立させているが，逐次 OS と同じ手法はギャングスケジューリングによる時分割スケジューリングに適用できない．
- ビジーウェイトあるいはポーリングしているプロセスの状態を外部のスケジューラがどうやって把握するかという問題がある．
- 対話処理において，プロセッサ利用率と応答性を両立させるためには並列プロセスの大域停止問題を解決する必要がある．

2.4 まとめ

以上，本章では，クラスタを構築するためのハードウェア及びソフトウェア技術と，並列計算機におけるジョブスケジューリング手法を概観した．クラスタ技術は価格対性能比の高い並列計算機を構築できるものとして，現在多くの注目を集めている．しかしながら，クラスタを逐次コンピュータの単なる集合としてではなく，単一システムとして統合するクラスタ管理ソフトウェアの研究は多くはない．

クラスタ管理ソフトウェアのひとつの重要な使命としてジョブスケジューリングがある．現時点でジョブスケジューリングを目的としたクラスタ管理ソフトウェアの大半は空間分割とバッチスケジューリングのみを扱うものである．ギャングスケジューリングは並列計算機上に効率的な時分割スケジューリングを実現する手法である．並列計算機に対するニーズから，並列計算機のジョブスケジューリングには空間分割スケジューリングが望まれ，時分割スケジューリングは空間分割により生じた断片化による効率低下を低減させることができる．一方，ギャングスケジューリングを用いて効率的な対話処理を実現する際の問題点についても触れた．

本章では特にユーザ並列プログラムの効率的な実行を可能とするユーザレベル通信に注目し，そこでのギャングスケジューリング実現の問題点を明らかにした．次の第 3 章では，実現するプラットフォームとなる PC クラスタとその上に実現された SCore クラスタソフトウェアについて紹介する．第 4 章では，ユーザレベル通信とギャングスケジューリングを同時に実現する手法の提案を行なう．また，第 4 章で提案するギャング

スケジューリングの実装方式は，効率的な対話処理を実現する際に生じる，いかにユーザ並列プロセスの大域状態を検出するか，という問題点に対し，解決策をも提供する．

第 3 章

SCore クラスタソフトウェア

本章では，本研究のプラットフォームとなるクラスタのハードウェア構成と，本研究に関連する SCore クラスタソフトウェアの概要について述べる．本研究の目的である効率的時分割スケジューリングは，SCore-D という名称で SCore クラスタソフトウェアの一部として実現されている．

本研究のハードウェアプラットフォームとなる RWC PCC-II は，クラスタが商用並列計算機に対抗できるだけの性能を持つことができることを実証するために開発されたものである．SCore クラスタソフトウェアは，クラスタ管理ソフトウェアだけでなく，MPI 通信ライブラリや並列言語のための実行時ライブラリなどを含む．

本章の目的は，本研究のプラットフォームとなった環境を紹介することであり，その基本性能を示すことである．

3.1 RWC PCC-II ハードウェアの概要

図 3.1 にはその概観の写真を，表 3.1 に RWC PCC-II[手塚 98] についての主要な諸元を示す．RWC PCC-II（以下，単に PCC と略記）各ノードは，基本的にシングルプロセッサの IBM-PC 互換機である．PCC 全体では 67 台の PC を用いているが，そのうち 3 台はファイルサーバや X Window サーバの役割を果たすサーバノードであり，残り 64 台が計算ノードである．ノード間通信ネットワークとしては Myricom 社製の Myrinet[BCF+95] を用いている．

図 3.1: RWC PCC-II

Myrinet は Ethernet と異なり，通信パケットの順序を保存し，ハードウェアエラーが起きない限り通信パケットを失うことはない．ネットワークインターフェイス（NIC）及びネットワークスィッチとも流量制御機能が備わっている．このため，信頼性を保証するプロトコルや，受信パケットを送信順序に並び替える必要がなく，その分，高速な通信が可能になる．図 3.2に PCC で採用されたネットワークトポロジを示す． Myrinet

表 3.1: RWC PCC-II の仕様

Cluster System	Number of Compute Nodes	64
	Number of Server Nodes	3
Network Subsystem	Network	Myrinet & Fast Ethernet
Node Subsystem	Node Processor	PentiumPro
	Chip Set	440FX
	Clock [MHz]	200
	Memory [MB]	256
	Disk [GB]	2
	I/O Bus	PCI ($133\,MB/s$)
	Local OS	NetBSD 1.2

スィッチは 8 ポートのクロスバスィッチを 8 個組み合わせ，ひとつの箱に収めたものを 4 つ用いている．この制約からネットワークトポロジは複雑なものとなっているが，基本的には 4 つのスィッチの箱を完全ネットワーク的に結合したものである． Myrinet のスィッチは cut through による転送のため，スィッチを経由することによる転送遅延時間は短い [BCF+95]．計算ノード間のバイセクションバンド幅は $5.12\,GB/s$ である [手塚 98].

Myrinet のネットワークケーブルは SAN （System Area Network）用と LAN （Local Area Network）用の 2 種類がある． SAN はケーブルの最大長が 3 メートルと短いが，リボン形状のケーブルなので配線の自由度が高く， PCC の計算ノード間は全て SAN ケーブルが用いられている．

Myrinet の NIC には LANai と呼ばれる特殊なプロセッサが載っている [BCF+95]. 図 3.3 は Myrinet NIC のブロック図である．この NIC には LANai と呼ばれる特殊なプロセッサが載っており，ホストインターフェイスを制御する． LANai のプログラム（ファームウェア）の格納と，ワーキングメモリとして SRAM が搭載されている．この SRAM 領域は I/O バスを経由してホストプロセッサからアクセスすることも可能である． LANai をプログラムすることにより NIC において通信プロトコルを処理することが可能である．

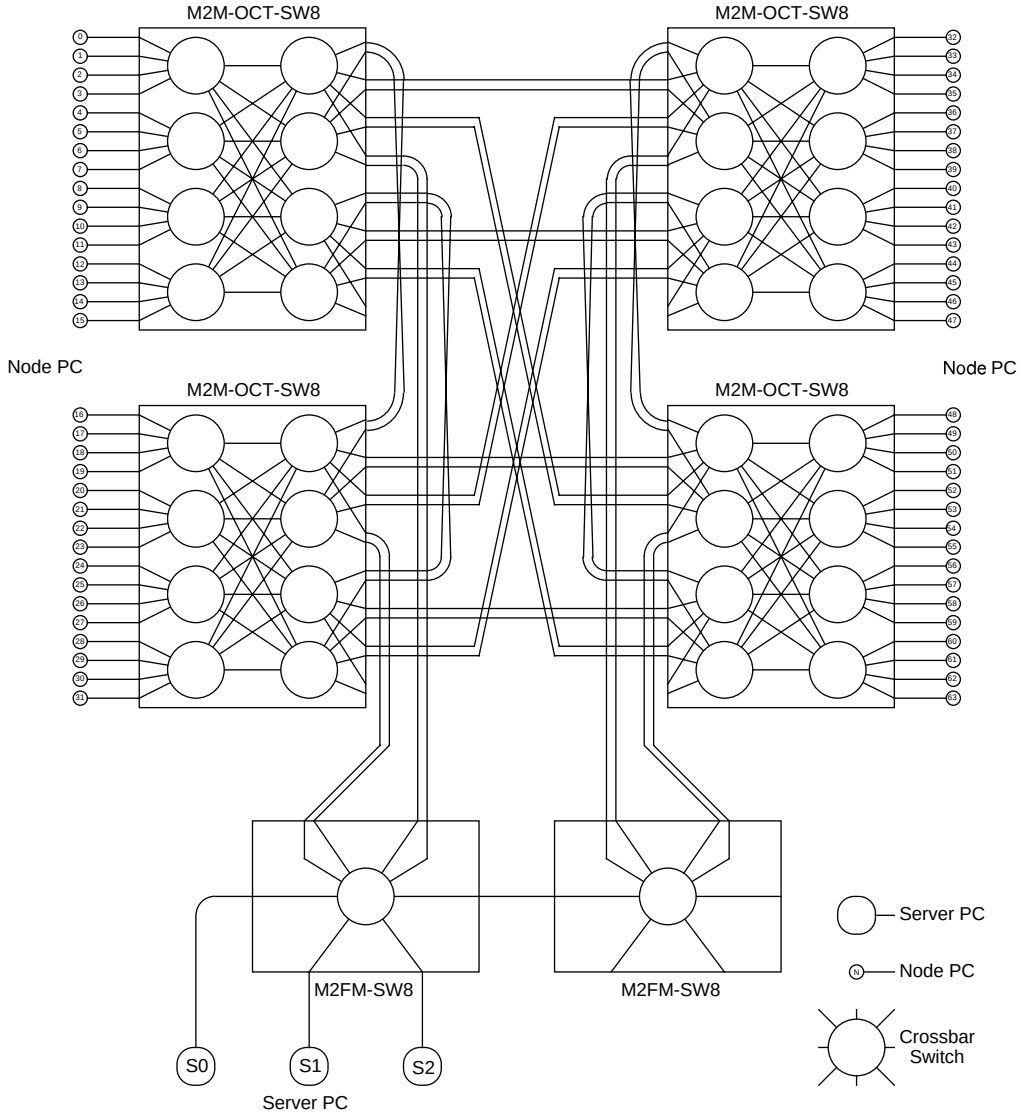


図 3.2: ネットワークトポロジ

3.2 SCore クラスタソフトウェア

SCore クラスタソフトウェアとは，クラスタを管理する管理ソフトウェア，並列言語実行時ライブラリ，および MPI 通信ライブラリ [GLS94] の集合である．基本的に SCore クラスタソフトウェア一式で，クラスタの管理から並列プログラムの開発実行までの全てが可能である．図 3.4 にその構成を示す．通信ハードウェアとしては Myrinet が想定されており． Myrinet の性能を最大限に発揮し，クラスタ上で多重並列プログラミング環境構築を想定した低レベル通信ライブラリ PM[手塚 96, THI96, THIS97, STH+98, TOHI98], と言語や高レベル通信ライブラリに独立した SCore 用の実行時ライブラリがある．さら

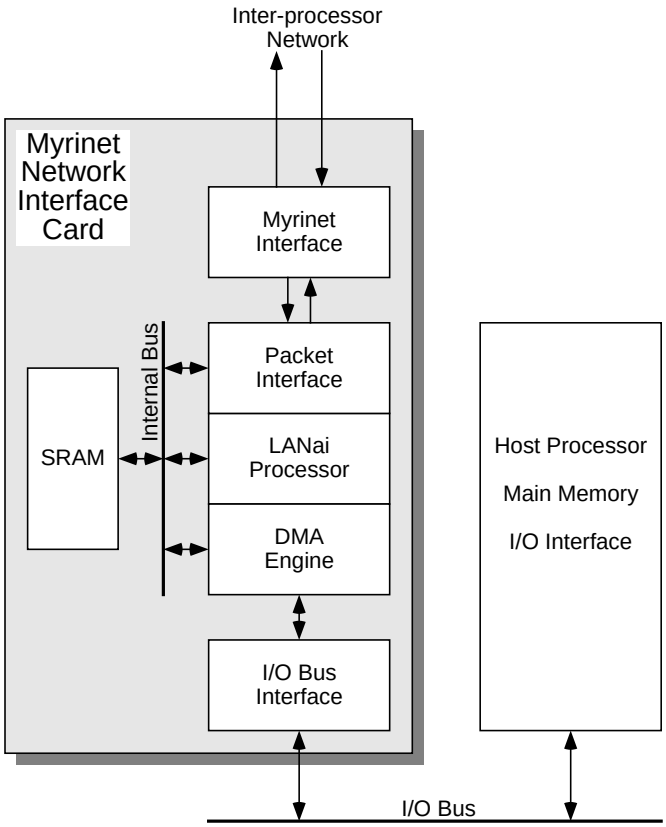


図 3.3: Myrinet NIC のブロック図

に上位には，ユーザの並列プログラミングを可能とする， C++ を並列拡張した MPC++[石川 94, 石川 95, Ish96] や，並列オブジェクト指向言語 OCore[KTM+95, KIT+96, Kon97] 用実行時ライブラリ， MPI 通信ライブラリ [OHT+97, OTHI98b] などが提供されている．SCore クラスタソフトウェアでは，基本的にクラスタを並列処理に特化した並列計算機の一種と考え，逐次プロセスは並列度 1 の並列プロセスとして扱う．

3.2.1 基本通信ライブラリ PM

PM は Myrinet[BCF+95] 用に開発されたソフトウェアの総称である [手塚 96, THI96, THIS97, TOHI98]． PM は LANai ファームウェア，デバイスドライバ，ライブラリから構成されている． PM は可能な限り低レイテンシ，高スループットな高性能な通信を実現すると同時に，ギャングスケジューリングの実装を念頭に置いて設計されている．高性能な通信を実現するために， Myrinet のパケット伝送の信頼性やパケット順序の保証を仮定して設計されている．

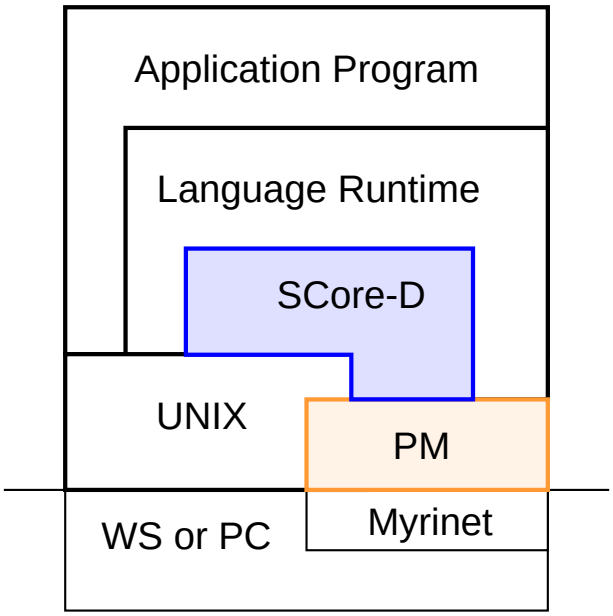


図 3.4: SCore クラスタソフトウェアの構成

PM は 4 つの「チャネル」をサポートしている． PM のチャネルは物理ネットワークを仮想化し，多重化するものである． PM では，チャネル毎に設定されたプロセッサノードの集合に対し，どこにでも送信が可能であり，どのノードからのメッセージも受け付けることができるコネクションレス型の通信をサポートしている．その意味で， PM のチャネルは「仮想ネットワーク」と呼ぶことができよう．送信できるプロセッサノードを限定することで，仮想ネットワークはサブネットワークとすることも可能である．それぞれのチャネルは他のプロセスによるアクセスから保護されている．また，あるチャネルからの送信は，受信側で同じチャネルに配送される．チャネル間の通信は認められていない．チャネルの実体は，送信メッセージ用の FIFO バッファと受信メッセージ用の FIFO バッファ，およびそれらの管理情報から成る．送受信用の FIFO バッファは，PM のチャネルをオープンすることで，オープンしたプロセスにメモリマップされる．これによりソフトウェアによるメッセージコピーの回数を減らしている．

さらに PM では，スループット向上を目的に，ソフトウェアによるメッセージのコピーを無くした「ゼロコピー通信」をサポートしている．これは，送信メッセージ領域と受信メッセージ領域を（メモリページがディスクに swap-out されないように）ピンダウンし， Myrinet インターフェイスから直接 DMA 可能なようにしたものである [TOHI98]. 一般にピンダウン操作は OS 内のページ表の操作を伴い， Myrinet の通信速度に比べコ

ストが大きい．PM では，ピンダウンした領域をキャッシュすることでピンダウンのコストを軽減している [TOHI98]．PM を通じてピンダウンされた領域は，PM 内部のピンダウンキャッシュ表に登録される．ゼロコピー通信が終了し，ピンダウン領域が（ピンダウンを）解放されても，ピンダウンキャッシュ表が溢れない限りピンダウン領域は解放されない．次にピンダウンの要求が来た時，要求のあった領域がピンダウンキャッシュ表に存在するならば，実際のピンダウン操作を省略することができる．

Myrinet はハードウェアレベルで流量制御が支援されており，メッセージの配送が保証されているが，ハードウェアにまかせた流量制御はデッドロックの危険性があるので，PM レベルで別途流量制御を行なっている．このためのプロトコルは Modified Ack/Nack と呼ばれている [手塚 96]．以下，簡単にプロトコルの概要を説明する．受信側では，メッセージを受信し，そのメッセージが受信バッファに収まれば送信側に Ack を返し，そうでない場合はそのメッセージを捨て，送信側に Nack を返す．一方，送信側では，Ack が戻ってくるまで送信バッファの当該メッセージが占有するメモリ領域を解放しない．Nack が返ってきた場合に再送する必要があるからである．Ack はメッセージ領域の解放にのみ用いられ，ユーザから見た場合の送信は送信バッファにメッセージを書き込んだ時点で終了しているように見える．従って，PM におけるメッセージ送信は非同期である．

図 3.5 にゼロコピー通信ではない通常の PM のメッセージ転送のバンド幅とメッセージサイズ（図中 “Message Size”）の関係を示す [TOHI98]．この図で “No Copy” とあるのは，ユーザが送信側ではメッセージの内容を書き込まず，また受信側ではメッセージの内容を読み込まない場合のバンド幅を示し，”Copy” とあるのは，送受信双方でメッセージの内容を読み書きした場合のバンド幅である．”No Copy” の場合において最大バンド幅が 120 MB/s 近くまで出ている．PCC の Myrinet インターフェイスは PCI バスでホストプロセッサと接続されている．PCI バスのハードウェアの理論性能は 133 MB/s であることから，PM はほぼハードウェアの性能上限まで性能を引き出していることが分かる．ちなみに copy 時において，メッセージサイズが 100 KB 辺りからバンド幅が低下する傾向が見られるが，これはキャッシュの影響と考えられている．

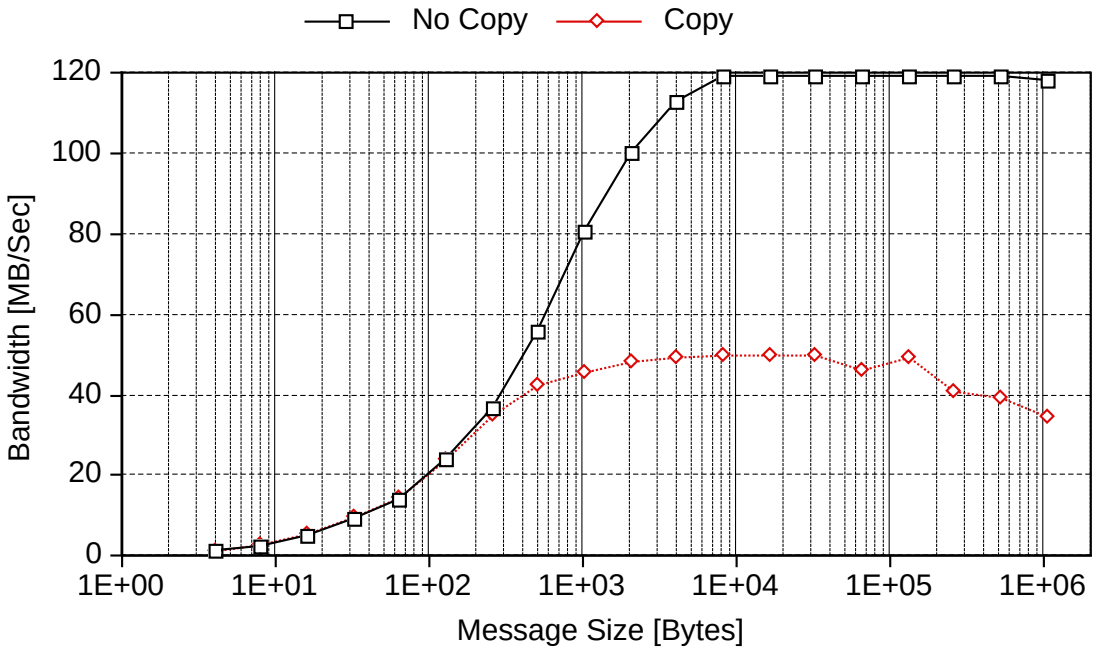


図 3.5: PM のメッセージ通信バンド幅

3.2.2 SCore 実行時ライブラリ

SCore 実行時ライブラリは，Unix をローカル OS とするクラスタ上で，PM を初期化し，並列プロセスを生成し，抽象化された通信インターフェイスと SPMD 並列実行モデルや並列デバッグモデルの提供を目的とする．ここで並列プロセスの生成には 2 通りの方式が SCore 実行時ライブラリにより提供されている．ひとつは，後述する SCore-D が提供する多重並列プログラミング環境の下でユーザ並列プロセスを生成し実行する方法，もうひとつはクラスタの一部あるいは全てを排他的に並列プロセスを生成し実行する方法である．後者の場合では，SCore 実行時ライブラリは Unix の `rsh` コマンドを用いて並列プロセスを生成する．これを「スタンドアローン」という意味で SCore-S と呼び SCore-D 下での並列プロセスと区別する．ちなみに SCore-D は SCore-S 実行時ライブラリを用いて起動される．

SCore 実行時ライブラリは，指定された数のノード（あるいはプロセッサ）上で同じプログラムを起動する SPMD（Single Program, Multiple Data）並列実行モデルを提供する．また，並列環境でのデバッグを容易にするために，例外シグナルを検出するとデバッガを attach するようにしている．

3.2.3 マルチスレッド言語 MPC++

MPC++ は C++ を拡張した並列言語である [石川 94, 石川 95, Ish96]. C++ の言語拡張をユーザが定義可能とするメタレベルアーキテクチャ [Ish95] と, C++ のテンプレート機能を用いたマルチスレッドによる並列拡張 [Ish96] という 2 点を特徴とする. 後述する SCore-D は MPC++ が提供するマルチスレッド機能を用いて記述されているため, ここでは後者の特徴について簡単に紹介する.

MPC++ の並列拡張機能としては, スレッドモデルを基に関数の遠隔起動, 同期機構および遠隔メモリアクセスがサポートされている. MPC++ のスレッドモデルはスタックを保存する. このため C や C++ で記述された逐次プログラムを自然に並列プログラムに移行することが可能である.

MPC++ の実行時ライブラリは ULT （User-Level Thread または Ultra-Light Thread の略）と呼ばれ, SCore および PM ライブラリとともに使用することを前提に設計されている [堀 96a]. ULT はユーザレベルスレッドをサポートする. 表 3.2 には, ULT の null 関数スレッドを起動しそれが終了するまでの時間を計測した結果を示す.

表 3.2: ULT のスレッド起動性能

Local Function Call	2.28 [μsec]
Remote Function Call	24.1 [μsec]

3.2.4 MPI 通信ライブラリ

SCore クラスタソフトウェアでは, 標準通信ライブラリ MPI [GLS94] のポータブルな実装である MPICH （アルゴンヌ国立研究所が中心となって開発）をベースに, PM および SCore 用に移植した MPICH-PM [OTHI98b, TOHI98, OTHI98a] が含まれている.

図 3.6 に MPICH-PM のメッセージサイズとバンド幅の関係を示す. PM のゼロコピー通信では, メモリページのピンダウン処理は DMA の設定など, 実際の通信開始に至るまでの手間がメッセージ通信に較べ大きい. そのため, メッセージサイズが小さい場合のバンド幅はメッセージ通信に劣る. しかしながらメッセージサイズが $8\,KB$ 辺りからゼロコピー通信の方が高いバンド幅を示すようになる. 前述したようにメモリペー

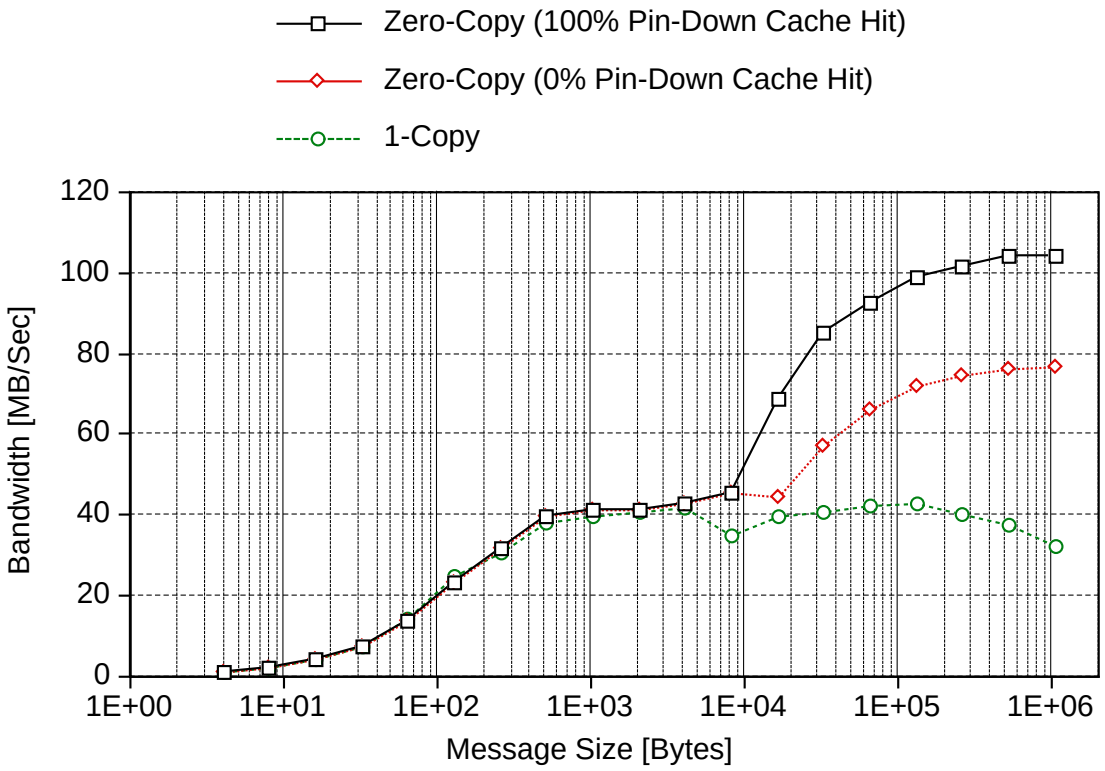


図 3.6: MPICH-PM のバンド幅

ジのピンダウンの手間は，ピンダウンキャッシュにヒットするかしないかで異なる（第 3.2.1 節参照）．当然ながら，キャッシュのヒット率が高いほど高いバンド幅を示す．コピーを伴うメッセージ通信では 8 KB 以上のバンド幅が頭打ちの傾向を示す．このため，8 KB より大きいのメッセージでは，Myrinet の DMA を用いたゼロコピー通信になっている．この図においてメッセージサイズが 8 KB のところで段があるのは，このようにメッセージ送信方式を切替えているからである．DMA によるコピーを効率化するためにピンダウンされた領域をキャッシュしていることから，8 KB を越えるメッセージをバンド幅はピンダウンキャッシュのヒット率の影響を受けている．

3.3 RWC PCC-II の基本性能

3.3.1 NAS 並列ベンチマーク

本研究の多くの箇所で，性能評価を目的に NAS 並列ベンチマーク [BBLS93] が用いられている．本節では NAS 並列ベンチマークについて簡単に紹介する．NAS 並列ベン

チマークは，計算流体力学に必要な並列処理を代表する 8 本のプログラムから構成されている．基本的にこれらのプログラムは全てデータ並列プログラムである．表 3.3 は 8 本のプログラムの処理内容と必要とするプロセッサ数に関する制限を示している．

表 3.3: NAS 並列ベンチマーク

Name	# of Processors	Note
CG	2^N	Conjugate Gradient
EP	2^N	Embarrassingly Parallel
FT	2^N	Fourier Transform
IS	2^N	Integer Sort
LU	2^N	LU Decomposition
MG	2^N	Multi-Grid
BT	M^2	Block Tridiagonal Solver
SP	M^2	Pentadiagonal Solver

N， M は自然数

ベンチマークのそれぞれのプログラムは問題の大きさによりクラス S， W， A， B， C（小さい順）とあるが，本研究では時間的および PCC のメモリの制約からクラス A を用いた． NAS 並列ベンチマークの大きな特徴は，問題の大きさはクラスによってのみ決まることである．このため同じクラスの同じ問題の場合，使用するプロセッサ台数が少ないほど長い計算時間を必要とする．

NAS 並列ベンチマークの 8 本のプログラムはそれぞれ異なる通信パターンを持つ．例えば EP においては，その名が示すようにほとんど通信は発生しない．主な計算は浮動小数点数の演算である． IS では唯一整数演算性能のみが評価結果に影響する． IS や FT においては大量の通信が発生し，多くの並列計算機において通信性能が大きく評価結果に影響する．

性能評価を目的としているため，それぞれのプログラムは計算結果の正当性をチェックし，計算能力を自分で評価し，出力する． NAS 並列ベンチマーク 2.3b は MPI で記述されており，可搬性が高く，多くの並列計算機上での結果が公表されている [NAS].

3.3.2 RWC PC Cluster のベンチマーク性能

PCC の基本性能の高さを示すために，本節では NAS 並列ベンチマーク [BBLS93] の結果を他の並列計算機と比較する．図 3.7 は NAS 並列ベンチマーク（2.3b 版，クラス A）の結果を，PCC の性能を 1 とした場合の相対性能でグラフにしたものであり，表 3.4 はベンチマークプログラムが内部で計算した全体での MOPS 値（絶対性能）を示す．NAS 並列ベンチマーク（2.3b 版）は MPI で記述されているため，前述した MPICH-PM を SCore-S により PCC 上で動かした．プロセッサ数は全て 64 台であり，日立製作所製 の SR2201 は技術研究組合新情報処理開発機構にあるものの上で実測した結果を用い，Cray T3-D，Cray T3E-900，UCB NOW については NAS にある結果 [NAS] を用いた．

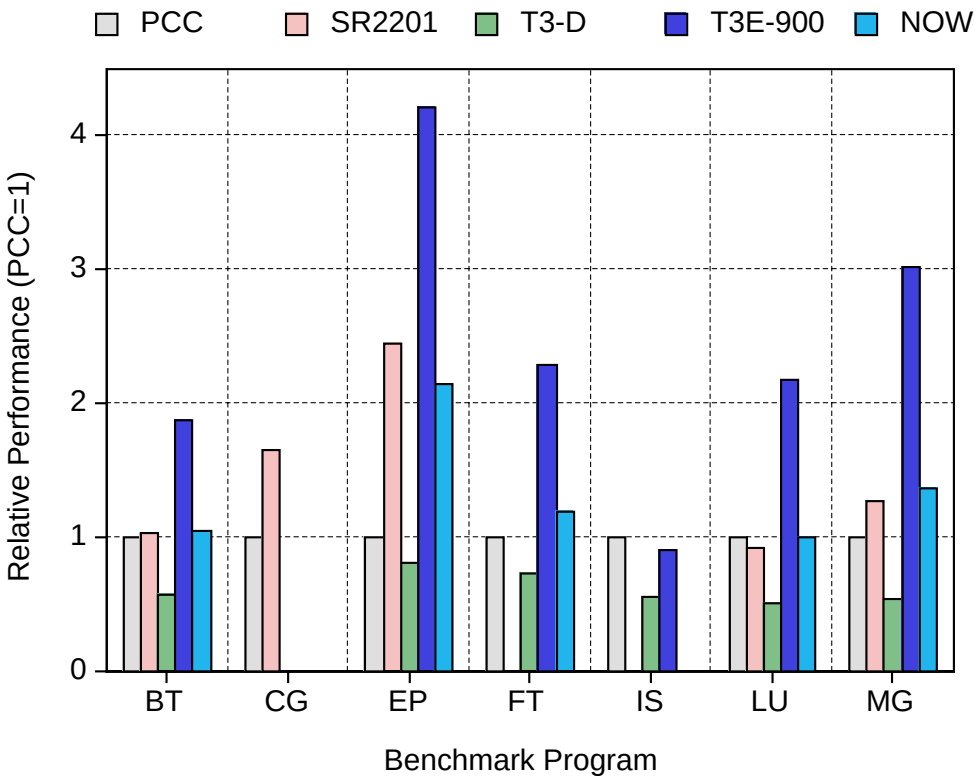


図 3.7: NAS 並列ベンチマークによる性能比較

このグラフから，PCC の相対性能は全てのベンチマークにおいて Cray T3-D を凌いでおり，SR2201 や UltraSparc を用いた UCB NOW の EP を除き，ほぼ匹敵する性能を発揮していることが分かる．EP （Embarrassingly Parallel）ベンチマークでは

表 3.4: NAS 並列ベンチマークの絶対性能（64 プロセッサ）

Program	PCC	SR2201	Cray T3-D	Cray T3-E 900	NOW
BT	1720.4	1777.1	983.6	3225.6	1816.0
CG	459.1	761.1	N/A	N/A	N/A
EP	30.6	75.3	25.2	129.2	65.9
FT	1053.51	N/A	772.1	2420.0	1264.3
IS	84.73	N/A	47.4	76.6	N/A
LU	2071.37	1913.2	1055.3	4529.4	2079.2
MG	1520.2	1948.1	839.0	4597.7	2074.2

単位： MOPS

通信がほとんどないため，プロセッサの浮動小数点演算能力がベンチマークの結果に直接影響する．PCC のプロセッサである PentiumPro の浮動小数点演算能力は他の同じクロック周波数の RISC プロセッサに比べ劣っているため，PCC における EP の性能が見劣りする結果となっている．しかしながら，他のベンチマークでは PCC の性能は，Cray T3E-900 を除いて，それほど見劣りするものではない．

3.4 SCore-D

SCore-D [HTI+96, 堀 96c, HTI97a, HTI97b, 堀 97, HTOI98b, 堀 98a] はクラスタを総体としてひとつのシステムとしてユーザに見せ，多重並列プログラミング環境を提供することを目的とするクラスタ管理ソフトウェアである．効率的な多重プログラミング環境を実現するために SCore-D はギャングスケジューリングによる時分割スケジューリングを実現している．また，ジョブスケジューリングに留まらず，より広範な並列 OS の研究を目的にシステムコールや I/O などの機能を備える．この意味で SCore-D はローカル OS としての Unix に大域的な OS 層を構成している．SCore-D は Unix 的には普通のユーザレベルのプロセスであり，保護という側面からは完全ではない．しかしながら敢えてユーザレベルの並列 OS を開発するのは，並列 OS の機能的および性能的な研究のプラットフォームとなるべく開発効率を優先させた結果である．

SCore-D の設計において，既存の逐次プログラムからのスムーズな移行に関しては

特に考慮されていない．つまり，既存の逐次プログラムの実行イメージは，そのままでは SCore-D で実行はできない．これは SCore-D が基本的に並列プログラムしか対象としないためである．

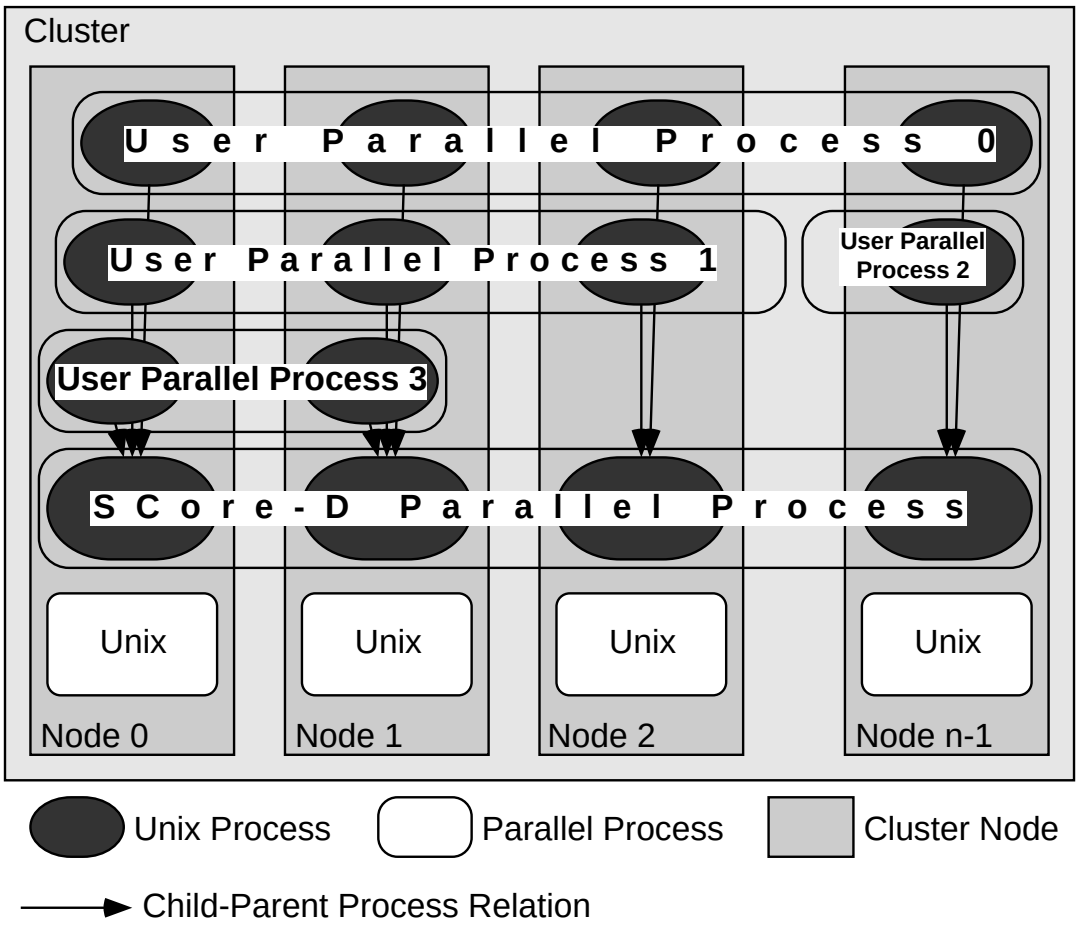


図 3.8: SCore-D のプロセス構造

SCore-D[堀 96c] は MPC++ 言語 [IHT+96, Ish96] で記述されている．近代的な OS カーネル内部の動作は基本的にマルチスレッドである． SCore-D を記述する言語として MPC++ を選択したのは， SCore-D 自体が並列アプリケーションであるということの他に， OS カーネルをマルチスレッド言語で記述することが自然であるとの発想による．SCore-D の設計に際し，スケーラビリティの確保に重点を置いた．

図 3.8 に SCore-D とユーザ並列プロセスのプロセス構造を示す． SCore-D はクラスタ上で動作する並列プロセスである．ユーザ並列プロセスを構成する個々のプロセスは， Unix システムコールの `fork()&exec()` を用いて生成される．スケジューラは時分

割空間分割スケジューラを想定している．従って，スケジューラがユーザの要求に従いプロセッサ資源を自動的に割り当てる．

SCore-D は PM のひとつのチャネルを占有し，残りのチャネルはユーザ並列プロセスが利用可能である． SCore-D では受信メッセージを `select()` システムコールで待つ．ユーザプロセスでは受信メッセージをポーリングで待つ．こうすることで， SCore-D とユーザ並列プロセスはスレッドレベルでインターリーブされて並行動作することが可能になる．ただし，こうすることで SCore-D 並列プロセス内の通信性能がある程度犠牲になっている．

3.4.1 ユーザ並列プロセスの制御

図 3.9 は SCore-D がどのように並列プロセスを生成し，制御するかを示したものである．図中，角が直角の四角は Unix のプロセスを表し，角が丸い四角および丸は SCore-D 内部の概念的なオブジェクト（機能モジュール）を表す．またシェードされた領域はそれぞれのオブジェクトがどのように計算ノード（プロセッサ）に分散されているかを示す．制御のための分散ツリーは，ユーザ並列プロセスの制御が効率的に行なえるよう，またユーザ並列プロセスを構成する個々のプロセスの状態変化を効率的にとりまとまるように分散されたツリー構造になっている．別な言い方をすれば，このツリー型の制御構造は，一種のブロードキャストおよびバリア同期の役目を負っている．図中の矢印は，ユーザ並列プロセスがどのような手順で生成されるかを示している．

ユーザ並列プロセスがどのように生成されるかという手順を以下に示す。

1. ユーザはクラスタの外の計算機上で，クラスタで走らせたい並列プログラムを起動する．ここで起動されたプロセスは FEP （Front-End Process）と呼ばれ，ここでは SCore 実行時ライブラリのみが走り，ユーザのコードは実行されない． FEP はクラスタで走行中の SCore-D に TCP コネクションを確立し，必要なプロセッサ数など，並列プログラムの実行に必要な情報を SCore-D に送る．

2. SCore-D ではユーザからのコネクションを受け付けると， Login Manager が要求内容をチェックし，その結果をスケジューラに渡す．

3. スケジューラは要求されたプロセッサ数やシステムの負荷状況などから，ユーザ並列プロセスをプロセッサ空間のどこに割り当てるかを判断し，そこに並列プロセス

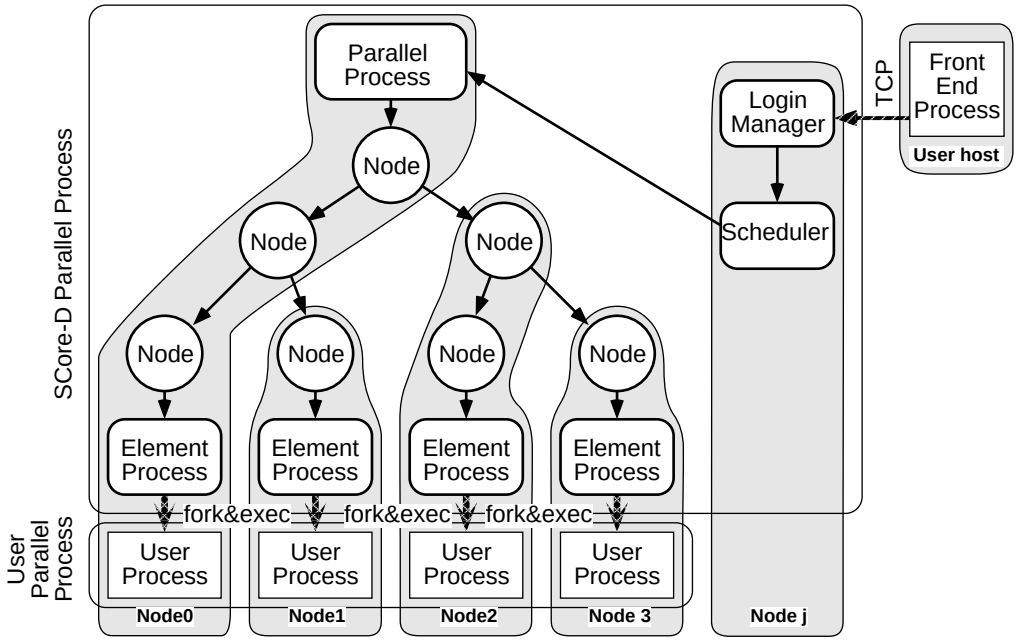


図 3.9: 分散制御構造によるユーザ並列プロセスの生成

全体の管理を目的とする Parallel Process オブジェクトを生成する．

4. Parallel Process オブジェクトはユーザ並列プロセスの制御のための分散ツリー（図中 “Node” で構成される木構造）を生成する．

5. FEP は自分自身の実行ファイルの内容を TCP のストリーム経由で SCore-D に渡す． SCore-D は分散ツリーに沿って実行ファイルの内容を計算ノードのローカルディスクにコピーする．

6. 分散ツリーの葉にあたる部分では，並列プロセスを構成する Unix プロセスの管理を目的とする Element Process オブジェクトを生成し，初期化完了後に Unix のシステムコールである `fork` と `exec` を用いてユーザプロセスを生成する．

SCore-D は全てのユーザプロセスの親プロセスであり，シグナルによりユーザプロセスを制御することが可能である．また，ユーザプロセスで発生した例外シグナルや正常終了に伴うプロセスの状態変化は，対応する Element Process オブジェクトに反映され，最終的に Parallel Process オブジェクトに伝わる．

3.4.2 システムコール

SCore-D は単なるクラスタ管理ソフトウェアではなく，OS のようにユーザ並列プロセスにサービスを提供することができる．これにより，ユーザ並列プロセスは SCore-D を経由して Myrinet/PM による高速な I/O や並列プロセス間通信が実現可能である．本節では，この SCore-D のシステムコールの機構について説明する．

SCore-D は OS と似たシステムコールの機能を提供している [HTI97a, HTI97b]．普通の OS のシステムコールでは OS に対するサービス要求の通知にトラップ命令を用いる．SCore-D においてトラップはユーザプロセスが自分自身に発行するシグナルで模擬することは可能である．しかしながらシステムコールを発行したことを通知する手段として pipe を用いた．これは，SCore-D 並列プロセスがプロセス間通信におけるメッセージの待ちを Unix の `select` システムコールで実現しているため，pipe で実現した方が相性が良いからである．

SCore-D のプロセスとユーザのプロセスとの間で効率的に通信するために，System-V 仕様のプロセス間通信機構のひとつである共有メモリセグメントを双方のプロセスで共有している．この共有メモリセグメントは c_area と呼ばれている．c_area はユーザプロセスの初期化に必要な情報を SCore-D から渡すためだけでなく，システムコールのための領域も確保されている．図 3.10 は c_area を通じてどのようにシステムコールの情報がユーザプロセスと SCore-D のプロセスで受け渡されるかを示している．

ここでユーザ並列プロセスはマルチスレッドで動作しているものとする．Unix と異なり，SCore-D はひとつのプロセスから同時に複数のシステムコールによるサービス要求を受け付けることができるように設計されている．このためにシステムコールの情報を受け渡す領域は，いくつかの「システムコールセル」に分割されている．ひつとにシステムコールはひとつのシステムコールセルを確保し，そのセルを通じて SCore-D とシステムコールに関する情報をやりとりする．

システムコールは以下に示す手順で処理される．

1. ユーザプロセスは，システムコールセルを一つ確保する．

2. 確保したシステムコールセルにシステムコールの引数をコピーする（図中 “Copyin”）．

3. SCore-D にシステムコール用の pipe を通じ，システムコールの要求を発行したことを通知する．システムコールを発行したスレッドはサスペンドする．この後，

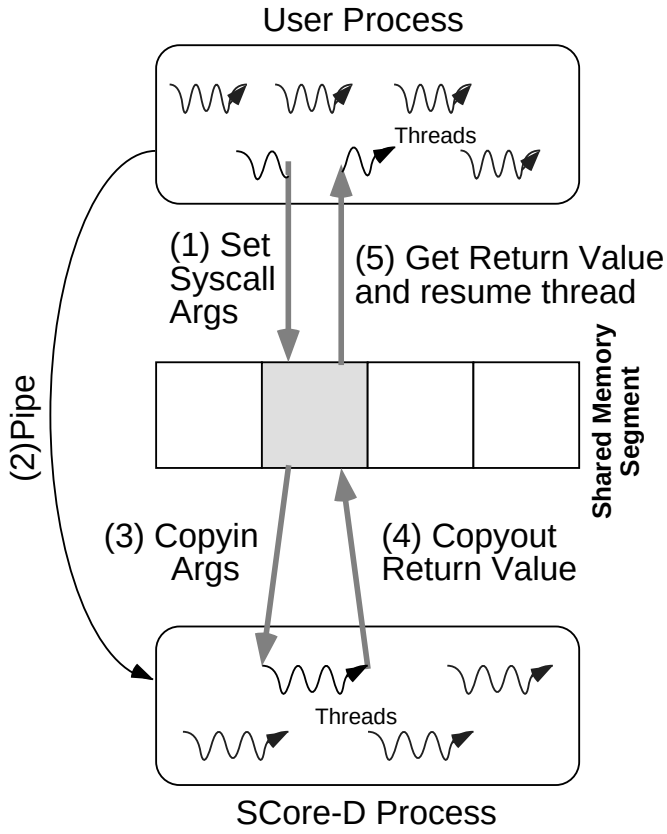


図 3.10: SCore-D のシステムコール機構

ユーザプロセスはシステムコールの完了をポーリングにより待つ.

4. SCore-D は pipe を通じ，システムコールの発行を知り，システムコールセルの内容に応じたサービスを行なう.

5. システムコールのサービスが終了したら，その結果をシステムコールセルに書き戻す（図中 “Copyout”）.

6. システムコールセルをポーリングして待っているユーザプロセスは，システムコールの完了を知るとその結果を取り出し，サスペンドしていたスレッドをスケジュールする.

一般に，ユーザレベルスレッドはカーネルスレッドに比べ，スレッド切替は高速であるが，システムコールがブロックした場合に他のスレッドに制御を移すことができない. 一方，カーネルレベルスレッドは，システムコールがブロックしても他のスレッドに制

御を移すことが可能であるが，スレッド切替のコストが高い．SCore-D が提供しているシステムコールの枠組は，ユーザプロセスのマルチスレッドがカーネルあるいはユーザレベルなのかといった違いには影響されない．もし，ユーザレベルスレッドで実現されていた場合，ユーザレベルスレッドにも関わらずユーザプロセスをブロックすることなく，カーネルスレッドの場合と同じように他のスレッドの実行を継続することができる．

ユーザレベルスレッドの高速なスレッド切替という利点と，カーネルスレッドのシステムコールにおける利点を併せ持つようなスレッドの研究としては，Scheduler Activation [ABLL92] が著名である．Scheduler Activation の枠組と SCore-D が提供するシステムコールの枠組とで最も大きく異なるのは，システムコール終了時の処理方式の違いである．Scheduler Activation ではシステムコールの終了は非同期にユーザプロセスの処理に割り込んで通知される．このためユーザのスレッドが危険区域を走行していた場合に問題が生じる可能性がある．一方，SCore-D ではシステムコールが完了したことを積極的にユーザプロセスに通知しようとはしない．ユーザプロセスのスレッド実行時ライブラリに任せる方式である．もしスレッド実行時ライブラリがスレッド切替のタイミングでシステムコールセルをポーリングするならば，Scheduler Activation における危険区域の問題は回避可能である．

表 3.5: getpid に要する時間

NetBSD 1.2	2.2 [μsec]
SCore-D	122.6 [μsec]

表 3.5 は NetBSD および SCore-D において，それぞれが管理するプロセス ID（SCore-D においては並列プロセス ID）を得るための時間を PCC 上で計測し，比較したものである．SCore-D においては getpid という最も単純なシステムコールのオーバヘッドが NetBSD に較べ 50 倍以上も遅い．プロセス ID を取得する時間の大半はシステムコールのオーバヘッドであると考えることができる．SCore-D におけるシステムコールのオーバヘッドはディスク I/O の遅延時間に較べれば十分に高速であり，このような I/O を目的としたシステムコールとしては実用的な範囲と考えることができる．

3.4.3 I/O

SCore-D では前節で述べたシステムコールを用いて簡単な I/O の枠組を提供している．図 3.11 は I/O システムコールの手順を示している．I/O デバイスの実体は（恐らくは I/O や通信を行なう）Unix プロセスである．このデバイスプロセスは SCore-D にデバイスを登録することで生成され，ユーザからオープンの要求があった場合には，SCore-D は対応するデバイスにオープン要求があったことを伝える．SCore-D プロセスとデバイスプロセスはシステムコールと同様，System-V 仕様の共有メモリセグメントを持ち，この共有メモリセグメントを経由してデータをやりとりする．

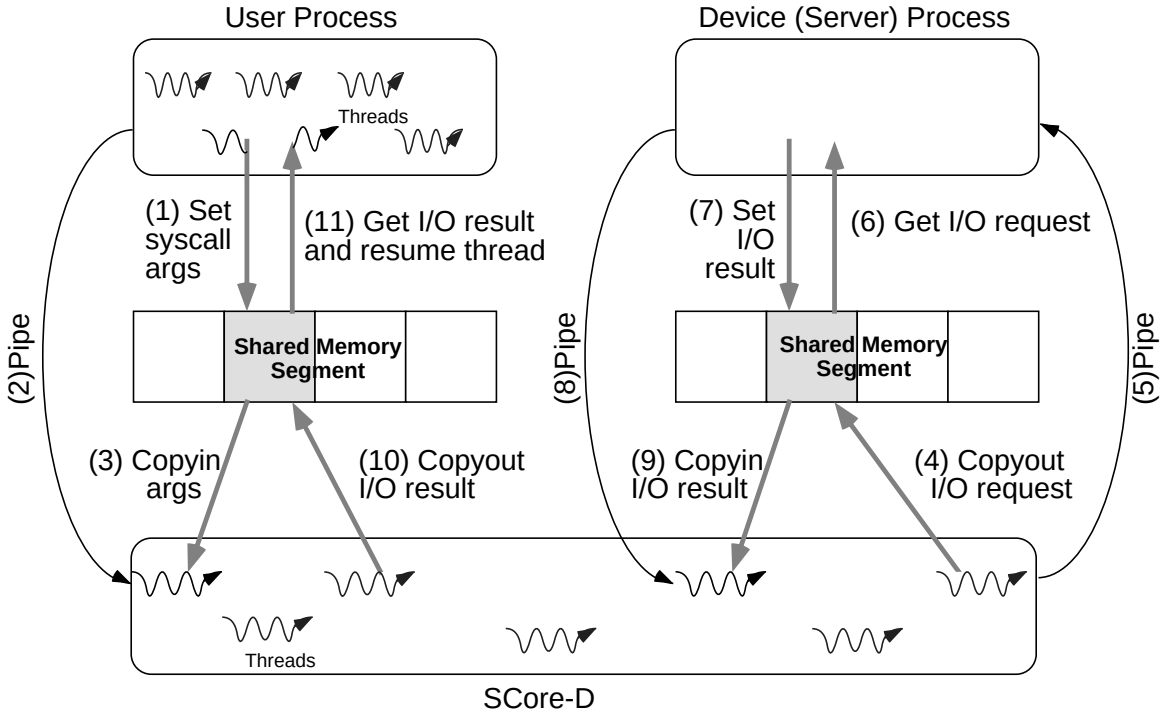


図 3.11: SCore-D の I/O 機構

デバイスプロセスはデバイスの登録者が指定したノードに生成される．デバイスプロセスとユーザ並列プロセスは同じノードにあるとは限らない．SCore-D はユーザ並列プロセスとデバイスプロセスとの間でデータを遠隔コピーする．このコピーは PM を用いるため，Ethernet などよりも高速である．

本節で述べた SCore-D の I/O の枠組は，並列プロセスからの並列なアクセス要求には対応していても，I/O デバイスの実体が単一の逐次の Unix プロセスなので PCC の並列処理性能に見合った I/O 性能を提供することはできない可能性がある．この機能は

主に， SCore-D のデモンストレーションや，第 6 章で述べる並列プロセスの大域状態検出機構を検証するために付加された機能である．

3.5 まとめ

RWC PCC-II は浮動小数点演算では他の並列計算機に若干見劣りするものの，商用並列計算機に匹敵する並列計算能力を発揮することが判明した．クラスタ技術の最大の利点のひとつは最新のプロセッサを利用可能なことである．例えば，プロセッサを 200 MHz の PentiumPro から 400 MHz の Pentium II に変更することで， 2 倍程度の性能向上を見込むことができ，商用並列計算機との差はますます縮まるものと予想される．

SCore クラスタソフトウェアは， PM と呼ばれる低レベルのユーザレベル通信ライブラリから， MPC++ といった並列言語や標準とも言える MPI 通信ライブラリの提供，さらに， SCore-D によるクラスタ管理ソフトウェアを含み，クラスタによる並列計算を様々な角度から支援することを目的としている．

研究のプラットフォームとして RWC PCC-II と SCore クラスタソフトウェアを見た場合，研究用プロトタイプというレベルではなく，実践的な並列処理を可能とするレベルでの評価が可能であるという点に注目すべきと考える．

第 4 章

ギャングスケジューリングの実装

第 2 章において，ユーザレベル通信が高い通信性能を発揮し，ギャングスケジューリングは並列計算機において時分割スケジューリングを実現するひとつの有力な方式であることを示した．一方，ネットワークインターフェイスのユーザからの直接操作を許すユーザレベル通信と，多重並列プログラミング環境を実現するギャングスケジューリングとを両方同時に実現する際に問題があることを示した．

本章では，ユーザレベル通信が実現する高性能な通信と，ギャングスケジューリングを両立させる実装方式として，ネットワークプリエンプションを用いる．ネットワークプリエンプションとは，ギャングスケジューリングの時点でネットワークの状態を退避及び復帰することで，ユーザレベル通信上にギャングスケジューリングを実現する際の問題点を解決するものである．

ネットワークプリエンプション自体は概念的なものであり，ハードウェア支援機能を仮定して実現する方法と，ソフトウェアのみで実現する方法が考えられる．本研究では，実現の容易さ，可搬性および空間分割の高い自由度を考慮し，ソフトウェアによるネットワークプリエンプション方式を実現し，評価を行なう．

4.1 ネットワークプリエンプション

第 2.3.1 節での議論を，PM ユーザレベル通信を用いるという前提でもう一度見直そう．PM がサポートする仮想ネットワーク機能は，スケジューリングの同期を取るために必要である．問題は，並列プロセス間の通信メッセージの分離方式である．PM では送信オーバヘッドを低減するためチャネル数を 4 に制限している．そのうちギャングスケジューラである SCore-D がひとつのチャネルを占有するためユーザ並列プロセス用には 3 つしか残らない．同時走行する並列プロセスに対して 3 という数は少な過ぎる．PM のチャネルを増やすという方法も考えられるが，送信オーバヘッドの増加と，チャネルの再利用の問題（第 2.3.1 節参照）を考えると，何か別な方式が望まれる．

仮想ネットワークの状態を全て退避し，後に復帰することができるのなら，ギャングスケジューリングの都度，ユーザプロセスに割り当てられた仮想ネットワークの状態を退避復帰すれば良い．逐次プロセスのプロセス切替ではプロセスの状態を退避復帰するのに対し，並列プロセスでは全てのプロセスの状態だけでなく，ネットワーク状態をも退避復帰することになる [HTI+96, 堀 96c, 堀 97, HTOI98b, 堀 98a, 堀 98b, HTI98, 堀 99]. これを「ネットワークのプリエンプション」と呼び，退避復帰されるネットワークの状態を「ネットワークコンテキスト」と呼ぶことにする．

CM-5 の All-Fall-Down 機構は，ギャングスケジューリング時にネットワークルータに存在するメッセージを最寄りのプロセッサに退避するものである [Thi92b]．メッセージの順序は保存されないが，そもそも CM-5 のネットワークではメッセージの順序は保証されないので問題にはならない．この意味で CM-5 の All-Fall-Down もネットワークプリエンプションのハードウェアによる実現と考えることができる．直接ネットワーク，つまりネットワークのノードがルータを兼ねているような場合，ルータとプロセッサの物理的かつ電気的な距離が近いのため，ハードウェアによってはプロセッサからルータの状態をアクセスできるものもある．RWC-1 [SHO+94] のネットワークはプロセッサからルータの状態をアクセス可能であり [YMO+95a, 横田 95, YMO+95b, 横田 97]，この機構を用いてネットワークプリエンプションを実現することが可能である [HYI+95].

仮想ネットワーク中に存在するメッセージの有無をソフトウェアで判別することができ，仮想ネットワーク中にある並列プロセスに関する通信メッセージがなくなるまで待つことができるならば，これは，即ち，ある並列プロセスに関するネットワークの状態が，ネットワークインターフェイスを含むプロセッサからアクセス可能な領域に移動し

たと考えることができる．そして，全てのプロセッサ上でネットワークインターフェイスやそれに伴う状態を保存することで，ネットワークのプリエンプションがソフトウェアで実現できたことになる．

仮想ネットワーク中のメッセージの有無を検出するために PM の流量制御プロトコルを応用する．ここで，全てのメッセージはネットワークのノードから発せられるものとし，ルータやスィッチなど他のネットワークを構成する要素はメッセージを発しないものとする．また，全てのメッセージは指定したネットワークノードを最終目的地とする．第 3.2.1 節で述べたように，PM のプロトコルでは，あるノードから送信したメッセージを受信したノードは必ず Ack あるいは Nack を返す規則になっている．あるノードから送信した再送を含む全てのメッセージに対応する Ack あるいは Nack を受けとった時点で，そのノードから送信したメッセージはネットワーク中に存在しないことになる．もしこの条件がある仮想ネットワークの全てのノードで成り立つならば，その仮想ネットワーク中にメッセージは存在しないことになる．これは全てのメッセージがノードから送信される，という前提から明らかである．

あるネットワークにおいて，全ての送信されたメッセージは必ず有限時間内に目的のノードに到着するものとする．このようなネットワークにおいて，受信ノードがメッセージを受信する，あるいはネットワークから取り込む，ならば，送信されたメッセージは必ず受信ノードに有限時間内に到着する．第 3.2.1 節で説明したように，PM はいかなる場合でもメッセージをネットワークから取り込む．受信バッファが満杯の場合は，受信したメッセージを破棄し，直ちに Nack メッセージを返送するので，ここでの議論には無関係である．もし，ネットワークがルーティング方式を含め上記のネットワークの条件を満たすなら，PM の Ack および Nack メッセージは有限時間内に送信元に返送されることになる．送信されたメッセージが有限時間内に目的ノードに到着しないようなネットワークとは，具体的には，パケットを損失するような信頼性のないネットワークであった場合，ネットワークスィッチにおいてメッセージが飢餓状態に陥るようなスケジューリングを行なった場合，あるいは，ネットワークトポロジおよびルーティング方式にデッドロックの可能性がある場合，である．

あるノードがメッセージを送信して，まだ対応する Ack あるいは Nack を受けとっていない状態を「遷移状態」と呼び，全ての送信メッセージの Ack あるいは Nack を受けとった状態を「定常状態」と呼ぶことにする．仮想ネットワークに含まれる全てのノードにおいて定常状態が成り立つならば，この仮想ネットワークは定常状態にある，とい

うものとする．ネットワークプリエンプションのために仮想ネットワークが定常状態になるまで待つ必要がある．このためには，もちろん，仮想ネットワークへのメッセージ送信は事前に禁止される必要がある．これは PM が送信バッファからメッセージを取り出して送信することを禁止することで容易に実現可能である． Ack および Nack が有限時間内に到着するということは，このようにノードからのメッセージ送信を禁止すれば，仮想ネットワークが有限時間内に定常状態になることを意味する．

仮想ネットワークの定常状態は，通常のメッセージの送達確認とは異なることに注意しなければならない．メッセージ送達確認とは，送信メッセージが正しく受信されたことを示すものである．一方，定常状態では，受信側のバッファが満杯のため受信したメッセージが破棄されている可能性がある．逆に，仮想ネットワークの定常状態の検出をメッセージの送達確認で代用することはできない．なんらかの理由により受信側のプロセスが受信バッファのメッセージを消費できなくて，受信バッファが満杯になっていた場合，受信プロセスがメッセージを消費し始めるまでそのノードへの送信は失敗することになるからである．受信バッファが既に満杯となっているプロセスが，スケジューリングにより停止していた場合，そのプロセスに対するメッセージの送達確認は有限時間内に終らない可能性がある．

逆に， PM のプロトコルにおいてメッセージの送達確認をするのは簡単である．送信メッセージに対応する全ての Ack を待つだけである [手塚 96]．メッセージの送達確認は遠隔メモリアクセスの完了のタイミングを知るために必要な機能である．このように PM の Ack/Nack プロトコルは単に流量制御のためだけでなく，ソフトウェアによるネットワークプリエンプションの実現や，メッセージの送達確認にも応用可能である．また，ネットワークプリエンプションのための上位プロトコルは不要であり，そのために余分なオーバヘッドは生じない．

RWC-1 で提案されたネットワークプリエンプションのためのハードウェア支援機能 [HYI+95, YMO+95a, 横田 95, YMO+95b, 横田 97] や CM-5 の All-Fall-Down [Thi92b] と， PM の Ack/Nack プロトコルによるソフトウェアによるネットワークプリエンプション機能は性能的にどのような差があるのであろうか． RWC-1 で提案されているネットワークプリエンプション機能においては，直接網というネットワークの特徴を活かし，プロセッサから直接ルータの状態を退避するものであり，ネットワークの状態退避をもっとも直接的な方法で実現しようとしている [HYI+95, YMO+95a, 横田 95, YMO+95b, 横田 97]. CM-5 の All-Fall-Down では，ネットワーク中のメッセージの目的ノードを一時的に変

更し，ネットワーク中の全てのメッセージを最寄りのノードに退避しようとするものである [Thi92b]．PM では特殊なネットワークを想定していないため，ネットワーク中のメッセージの宛先を変更することはできない．

一般に，ネットワーク中の i 番目のメッセージの長さを L_i とし，ネットワークのバンド幅を B とすると，ネットワーク中の全てのメッセージが目的ノードに到着する時間 T_{flush} は，おおまかに次式で求まる．

$$T_{flush} = \frac{\sum_i L_i}{B}$$

つまり，ネットワークの状態が全てのノードに反映されるまでの時間は，ネットワークが保持できるメッセージの量に比例し，バンド幅に反比例する．ネットワークが保持できるメッセージの最大量は，多くの場合，ルータが持つメッセージバッファの容量と，ネットワーク中に存在するルータの個数で決まる．実際にどれだけのメッセージ総量がネットワーク中に存在するかはアプリケーションの通信パターンに依存する．Myrinet の場合，ネットワーク中のメッセージの宛先を変更することはできないので，最悪のケースは，ネットワーク中にネットワークが保持できる最大量のメッセージが存在し，その全てのメッセージが特定のノードを宛先とする場合である．この場合，バンド幅 B は単一リンクのバンド幅となる．CM-5 のネットワークにおけるネットワークプリエンプション機構 [Thi92b] では，メッセージの宛先を一時的に最寄りのノードに変更するために，バンド幅 B がネットワーク全体のバンド幅になり PM に比べ有利である．これまでは理想的な状況，つまりネットワーク中のメッセージを無くすために，どこか適当なノードに送りつけるためのコストにのみを着目してきた．しかしながら，例えば CM-5 ではネットワーク全体を All-Fall-Down 状態に移行するためのコストも考慮されなければならない．CM-5 のギャングスケジューリングに要する時間の詳細は不明であるが，文献 [BHMW94] には，”CM-5 incurrs a minimum overhead of 4 $msec$, with typical times closer to 10 $msec$.” と報告されている．

PM の Ack/Nack プロトコルには，ネットワークの定常状態の検出やメッセージ送達確認が可能になるという以外にも，ギャングスケジューリングを実装する際に重要な性質がある．受信側の受信バッファが満杯になった場合，受信側は常に Nack を送信側に返す．Nack を受けとった送信側はメッセージを再送する．この手順は受信側がメッセージの消費を開始するまで繰り返される．この結果，送信バッファが満杯になり，ユーザはメッセージを送ることができなくなる可能性がある．このような状況は，ユーザから

見た場合，一種の閉塞状態に見えるが，実際にはネットワーク中には Nack と再送メッセージが流れ続けている．このため，あるチャネルがこのような状態に陥っても，他のチャネルのメッセージは，閉塞しているチャネルの Nack と再送メッセージの合間を縫って，ネットワーク中を流れることができる．このように，ある仮想ネットワークの閉塞状態が（通信性能の低下は避けられないが）他の仮想ネットワークに及ぶことがない．これを「仮想ネットワークの独立性」と呼ぶ．仮想ネットワークの独立性は，ひとつの仮想ネットワークを用いてギャングスケジューリングの同期を取る場合に重要である．ユーザ並列プロセスがユーザ並列プロセスに割り当てられた仮想ネットワークを閉塞させても，ギャングスケジューラはユーザプロセスを制御し続けることができるからである．

4.2 ネットワークプリエンプションの実装

ネットワークプリエンプションを用いたギャングスケジューリングの手順を図 4.1 に示す．図では左から右の方向に時間が進む．一番上の水平線は Parallel Process オブジェクトでの処理を示し．その下にある水平線群は Element Process オブジェクト群の処理である（第 3.4 節を参照のこと）． Parallel Process と Element Process は図 3.9 で示した分散ツリー構造で結ばれている． Parallel Process オブジェクトから Element Process オブジェクトへの指示，および Element Process オブジェクトから Parallel Process オブジェクトへの指示（図中では矢印で示される）は，全てこの分散ツリー構造に沿って処理されるため，$log(P)$ の時間オーダー（P は Element Process の数）で処理される．Parallel Process オブジェクトから Element Process オブジェクトへの指示は一種のブロードキャストであり， Element Process から Parallel Process への報告は一種のバリア同期である．

手順は freeze フェーズ， save フェーズ， restore フェーズ，そして run フェーズと 4 つのフェーズから構成されている． Freeze フェーズと restore フェーズは，これから実行を停止する並列プロセスのための処理であり． Restore フェーズおよび run フェーズはこれから実行を再開する並列プロセスのための処理である．以下にそれぞれのフェーズでの処理概要について説明する．

Freeze フェーズ　ユーザ並列プロセスの実行を `SIGSTOP` シグナルで中断すると同時に，ユーザに割り当てられた仮想ネットワーク（PM チャネル）への送信

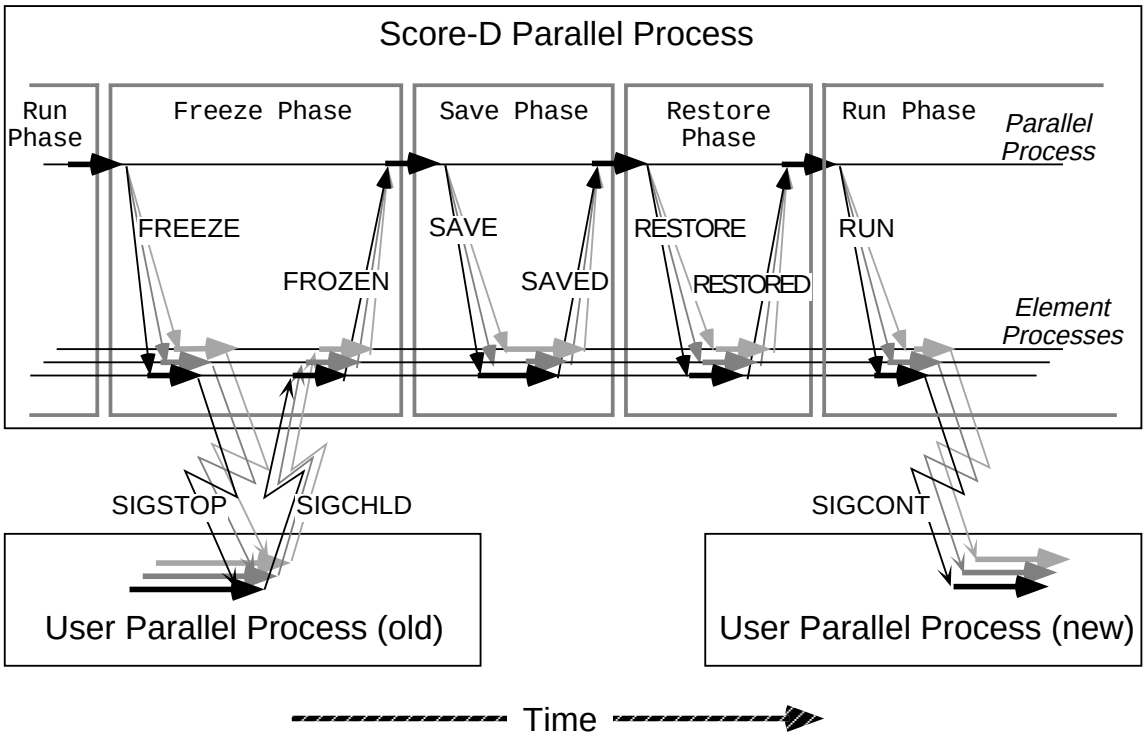


図 4.1: 並列プロセス切替

を停止させ，そしてユーザに割り当てられた PM のチャネルが定常状態になるまで待つ．全てのノードにおいて freeze フェーズが終了した時点で，ユーザに割り当てられた仮想ネットワークの状態はノードに移行している．

Save フェーズ　ノードの通信状態をメモリに退避する．このフェーズの終了をもって，いままで走行していたユーザ並列プロセスの全ての状態は退避が完了したことになる．

Restore フェーズ　これから実行を開始しようとする並列プロセスの退避してあった通信状態をメモリから復帰する．

Run フェーズ　仮想ネットワークへの送信を許可すると同時に，ユーザプロセスの実行を SIGCONT シグナルにより再開する．

この手順においてユーザ並列プロセスのプロセッサの状態保存は明確ではないが，実際には暗黙的に SIGSTOP シグナルにより OS 内部に退避され，SIGCONT シグナルによ

り復帰されている．

4.3 評価

本節では，前節で提案されたネットワークプリエンプションを用いたギャングスケジューリングを SCore-D に組み込み，そこでの性能を評価する．次節では， PM や SCore-D においてギャングスケジューリングの性能に直接関係するいくつかの指標について予備評価を行ない，第 4.3.2 節以降において実際のギャングスケジューリングのオーバヘッドについて計測する．

4.3.1 予備評価

予備評価として，最初に PM 単体でのチャネルコンテキストの退避や復帰に要する時間を計測し，次に SCore-D の分散ツリー制御構造のツリーが何分木が適切かについて評価を行なう．

PM のチャネルコンテキストの退避復帰時間

表 4.1 は， PM 単体でのチャネルコンテキストの退避と復帰に要する時間を測定したものである． PM のバッファの大きさは，送信バッファは最大 512 メッセージ，メッセージ総量の最大 48 KB，受信バッファは最大 4,096 メッセージ，メッセージ総量の最大 64 KB である．表中，”Both Empty” とあるのは送受信双方のバッファとも空， “Send Full” とあるのは送信バッファが満杯の状態，”Recv Full” とあるのは受信バッファが満杯，”Both Full” とあるのは両方のバッファとも満杯の状態を意味する． PC クラスタにおける Myrinet インターフェイスは普通の PCI デバイスである．メッセージバッファ管理情報は， Myrinet の特性から PCI メモリ領域（より具体的には Myrinet インターフェイスに付随する SRAM）に置かれている．チャネルコンテキスト退避の時間の方が復帰に比べ長いのは，プロセッサから見た PCI デバイスからの読み込み速度が書き込み速度よりも遅いというチップセットの特性から来るものと考えられる．

退避／復帰の時間は，コンテキストに含まれるメッセージの数およびメッセージの総量に依存する．メッセージの数が多いほど，また，メッセージの総量が多い程，退避／復帰に要する時間は長くなる．メッセージ総量の最大値は， PM のチャネルに割り当てられたバッファの大きさで規定される．より大きなバッファは通信性能に貢献するが，

表 4.1: チャネルコンテキストの退避／復帰時間

Buffer Status	Save [$msec$]	Restore [$msec$]
Both Empty	0.62	0.18
Send Full	1.96	1.56
Recv Full	2.16	1.59
Both Full	3.70	3.19

ネットワークコンテキストの退避／復帰により時間がかかることになる．この関係はプロセッサのレジスタファイルの大きさとプロセスの切替速度の関係に良く似ている．より大きなレジスタファイルは，個々のプロセスの計算速度に貢献するが，プロセスコンテキストの退避／復帰が遅くなる．

ここで表 4.1 の値から，ギャングスケジューリングのオーバヘッドについての考察を試みる．まず最悪ケースについて考えてみる．表 4.1 から送受信バッファがともに満杯だったとすると，チャネルコンテキストの退避復帰だけにおよそ $7\,msec$ 要することが分かる．一方，最良のケースでは，チャネルコンテキストの退避復帰に要する時間は $1\,msec$ 以下となり，最悪ケースの 1/7 しか要しない．このことから，ネットワークプリエンプション方式によるギャングスケジューリングのオーバヘッドはアプリケーションの通信パターンに大きく依存することになるものと予想される．

分散ツリー制御構造

第 3.4 節で述べたように， SCore-D 内部ではユーザ並列プロセスを分散ツリー構造により制御している．ここでの計測の目的は，ツリー構造として何分木が適切であるか，ということである．

分散ツリー構造単体での評価は実際の挙動と異なる可能性がある． SCore-D はマルチスレッドで記述されている．また， SCore-D はユーザレベル通信 PM を用いてはいるものの，通信の待ちを Unix の `select` システムコールで実現している．これは SCore-D プロセスとユーザプロセスは， SCore-D のスレッドレベルで実行がインターリーブされようにするためである．このため，ユーザ並列プロセスの有無により， `select` システムコールの OS カーネル内での処理が異なる可能性がある．また，ユーザプロセスと SCore-D がインターリーブされて実行されるため，キャッシュへの影響も無視できな

い．この問題を回避するために，ここでは NAS 並列ベンチマークから EP プログラムを選び，これを用いて分散ツリー構造の評価とすることにした． EP という名前は Embarrassingly Parallel に由来しており，その実体は数値計算が主体で，プログラム実行の最初と最後に若干の通信が行なわれるのみである．この EP ような単純な並列プログラムを用いることで，ユーザ並列プロセスにおける通信を無視して， SCore-D のオーバヘッドを計測することが可能となる．

図 4.2 は，分散ツリーにおけるノードからの枝の数を， 2， 4， 8， 16， 32，そして 64 と変化させた時（X 軸），第 4.2節で提案したネットワークプリエンプションによるギャングスケジューリング手順の freeze フェーズ， save フェーズ そして restore フェーズの処理時間の和の変化（Y 軸）を示したものである．時分割間隔は 100 $msec$ である．EP プログラムは， 8， 16， 32 そして 64 プロセッサの 4 通りで走らせた．

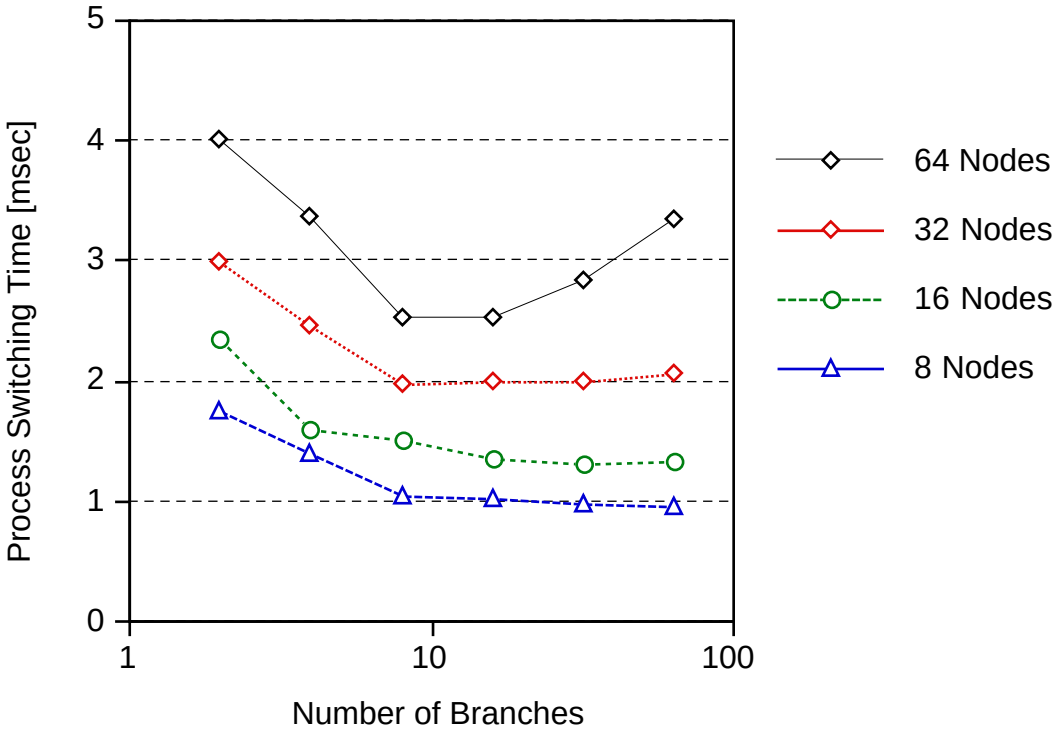


図 4.2: 分散ツリー構造の違いによるギャングスケジューリングオーバヘッドの違い

例えば 8 プロセッサの EP プログラムの実行の場合を見てみよう．枝の数が 8 を越えるとギャングスケジューリングによる並列プロセス切替の時間に変化が見られなくなるのは，枝の数がそれより大きくなっても実際にユーザ並列プロセスの制御に必要なの

は 8 だけであるからである．逆に 64 プロセッサの EP では，枝の数が 8 と 16 の場合に並列プロセス切替時間が最小となっている．以下，本研究における SCore-D による計測では，ここでの結果を基に分散ツリー構造として枝の数を 16 に設定した．

4.3.2 並列プロセス切替時間

本節では第 4.2 節で提案されたネットワークプリエンプションによるギャングスケジューリング実装方式を用いて，単純な時分割スケジューラを構築し，そこでのスケジューリングオーバヘッドを計測する．時分割スケジューリング方式としては単純なラウンドロビン方式とする．ここで実装された時分割スケジューラはスケジューリング対象となる並列プロセスがひとつしかない場合でもナイーブにギャングスケジューリングを繰り返す．このため，ひとつの並列プロセスをスケジューリングするだけでも，その結果生じるスケジューリングオーバヘッドを計測することが可能である．評価プログラムは第 3.3.1 節で説明した NAS 並列ベンチマークの 8 本全てのプログラムを用い，プログラムの起動直後から終了までの全ての並列プロセス切替時における freeze フェーズ， save フェーズ， restore フェーズの処理時間を計測し，その結果を平均する． NAS 並列ベンチマークで用いられている MPICH-PM では，ひとつの PM チャネルしか必要としない．

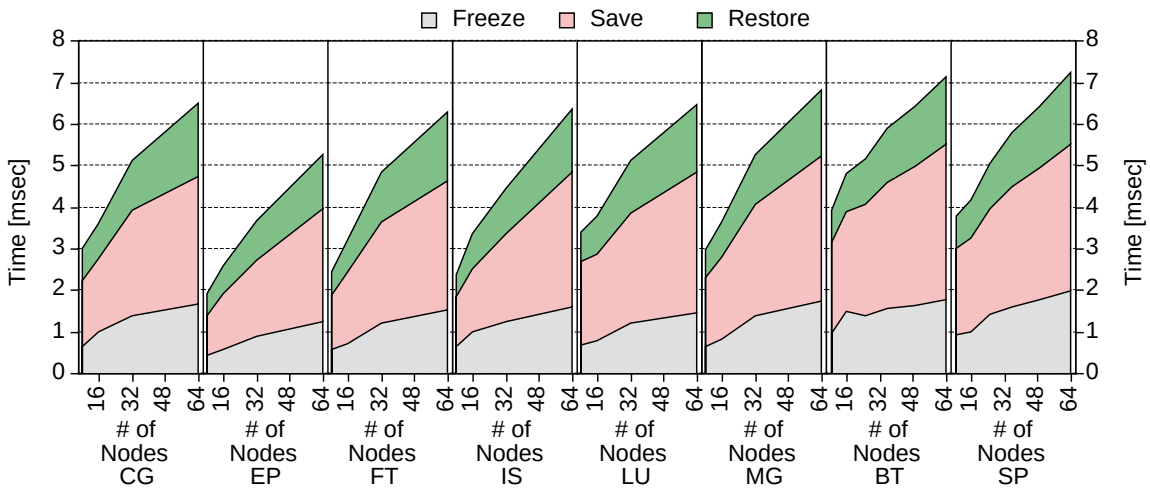


図 4.3: 各フェーズの処理時間

図 4.3 は，このようにして計測された各フェーズの処理時間である．グラフは縦軸が処理時間，横軸がプロセッサ台数である．時分割間隔は 40 $msec$ に設定した．プロセッサ数は， BT と SP を除き 16， 32， 64 と変化させた． BT と SP ではプロセッサ数が

二乗でなければならないという制限から，9，16，25，36，64 と変化させた．この図においてグラフのスケールは比較を容易にするために全て同じにしてある．

Freeze フェーズでは，第 4.2 節で述べたように，ネットワークプリエンプションの前段階として，ネットワーク中のメッセージがなくなるまでの時間，ユーザプロセスを SIGSTOP で中断し，wait システムコールで中断を確認するまでの時間が含まれている．別途簡単な計測プログラムによる計測では，子プロセスの実行を SIGSTOP で中断し，それを wait システムコールで確認するまでの時間は 40 〜 50 μsec であった．一方，save や restore フェーズでは，PM のチャネルの状態をメモリに退避あるいは復帰する時間が含まれる．Freeze フェーズ，save フェーズ，restore フェーズの全ての処理時間には，第 3.4.1 節で説明した分散制御ツリーに沿ってブロードキャストやバリア同期を取るための時間も含まれている．

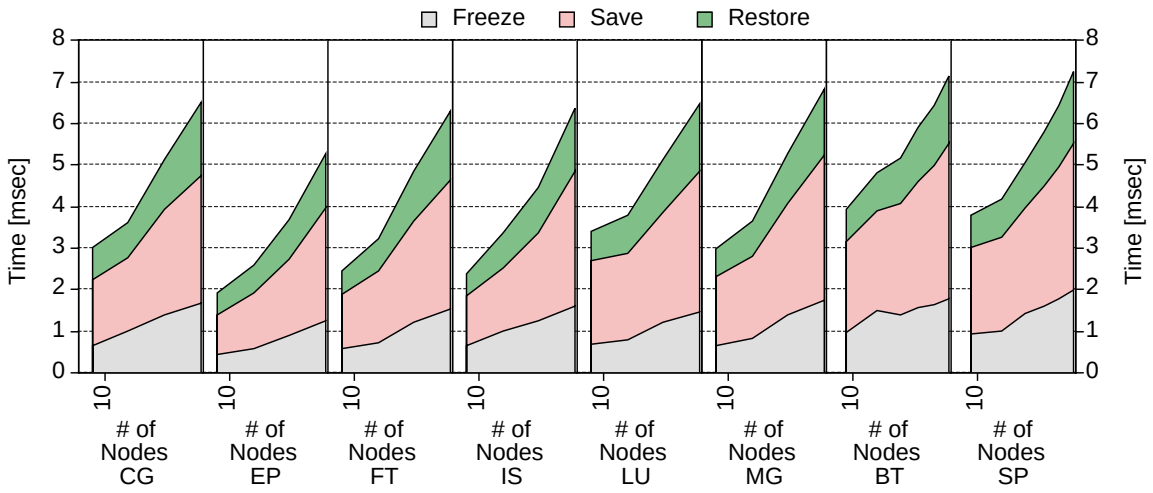


図 4.4: 各フェーズの処理時間（片対数）

分散制御ツリーに沿ってのブロードキャストやバリア同期は，ツリー構造の特性からほぼ $log(P)$ の時間オーダ（ここで P はプロセッサ数）になると考えられる．一方，ネットワーク中のメッセージがなくなるまでの時間や，save フェーズおよび restore フェーズにおける退避復帰時間は，メッセージ数や総量に依存する．スケーラビリティを確認する意味で，図 4.3 と同じデータを X 軸を対数軸としてグラフにしたものを図 4.4 に示す．このグラフから，スケジューリングオーバヘッドが $log(P)$ よりも大きくなっているのが分かる．

PM のチャネルの状態（コンテキスト）は送受信のためのメッセージバッファが大半

を占めている．このため，save フェーズと restore フェーズの処理時間の大半は，そのチャネルのメッセージバッファに含まれている通信メッセージの総量と，メッセージ数に比例するメッセージバッファの管理情報の退避領域への退避および復帰である．このため，その処理時間はメッセージ数とメッセージ総量に依存する（第 4.3.1 節参照）．また，freeze フェーズの時間はその時点でネットワーク中に存在するメッセージ量に依存する．このことから，SCore-D のギャングスケジューリングのオーバヘッドはアプリケーションの通信特性に大きく依存すると考えられる．

図 4.3 及び図 4.4 からは，freeze フェーズの処理時間は並列プロセス切替時間の 1/3 以下程度しかないことが分かる．逆に，save フェーズおよび restore フェーズを合わせた処理時間の方が，並列プロセス切替時間の約 2/3 以上という大きな部分を占めている．

4.4 問題点と改良方式の提案

この図 4.3 及び図 4.4 の結果から，PM チャネルコンテキストの退避復帰時間が，ネットワークプリエンプションによるギャングスケジューリングに要する時間の大半を占めていると推測される．そこで，このチャネルコンテキストの退避復帰時間を短縮する方法を考える [堀 98a, HTI98, 堀 99]．送受信バッファの大きさを小さくすればチャネルコンテキストの退避復帰時間は短縮されるが，通信性能に悪影響を及ぼす可能性がある．送受信バッファの大きさと通信性能の関係は，プロセッサのレジスタファイルの大きさと処理性能の関係に良く似ている．大きなレジスタファイルは，コンテキスト切替時間を大きくするが，プロセッサの処理速度に貢献する．

ここで PM のチャネルとそのコンテキストの関係を見直す．PM のチャネルは物理的にひとつのネットワークを多重化したものである．チャネルのコンテキストとは，チャネルの状態を指し，その実体はメッセージの送受信バッファである．もし，チャネルとそのコンテキストを切り離すことが可能であり，チャネルの数よりも多くのコンテキストを持つことが可能であるならば，わざわざコンテキストをメモリに退避復帰させずに，チャネルに接続されているコンテキストを別のコンテキストに切替えることで，チャネルコンテキストの切替が実現できることになる．ここでは，これを「PM のマルチコンテキスト化」と呼ぶことにする．これに対応して，従来の方式を「シングルコンテキスト」と呼び，区別する．

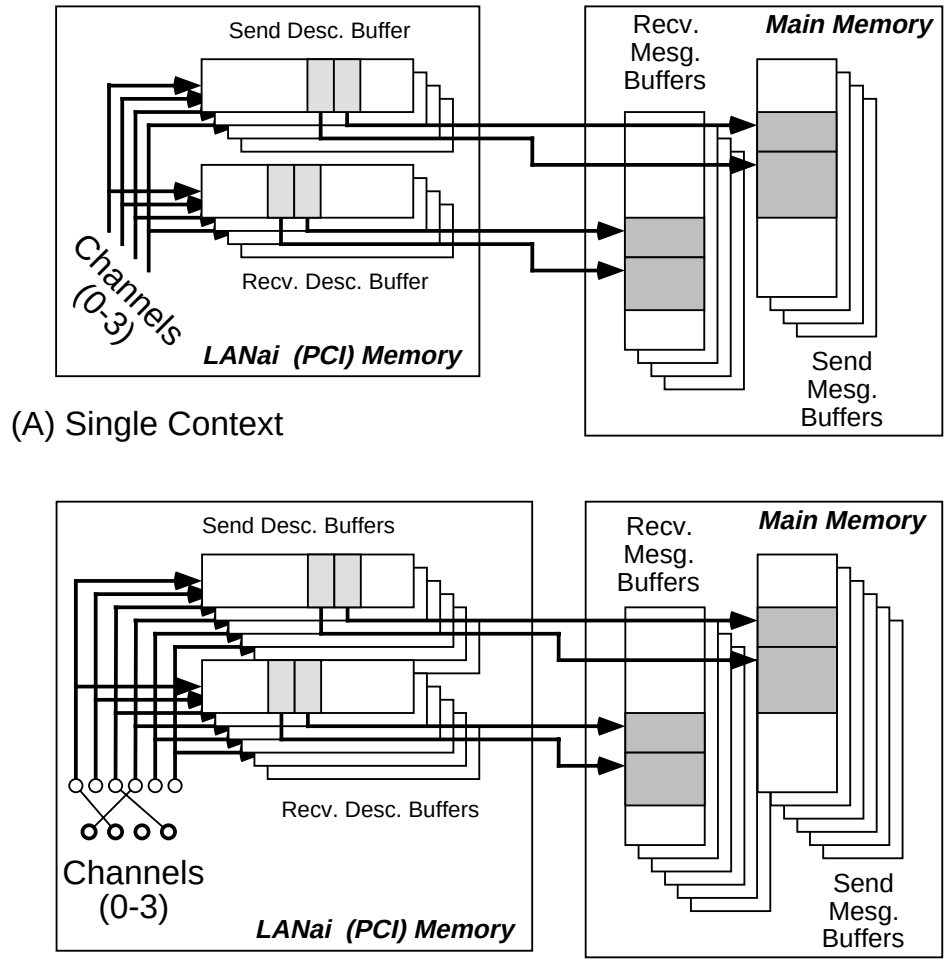


図 4.5: PM のマルチコンテキスト化

図 4.5 に，PM におけるマルチコンテキスト化について示す．あるチャネルのコンテキストは，具体的にはメッセージを格納する FIFO バッファ（図中では，単に “Message Buffer”）と，メッセージのメッセージバッファ上での場所，長さ，状態などの属性を保持するためのメッセージ記述子バッファ（図中では，”Descriptor Buffer”）の組が送信と受信の対で構成されている．メッセージ記述子バッファは，Myrinet インターフェイスに搭載される LANai プロセッサから頻繁にアクセスされるため，インターフェイス上の SRAM 内に置かれる．メッセージバッファ領域は，主記憶上の DMA 可能な領域に置かれ，ユーザプロセスからも直接アクセス可能なようにマッピングされている．

シングルコンテキストの場合では，チャネルとコンテキストは 1 対 1 対応であった．このため，あるチャネルをプロセス切替時に別のプロセスに空け渡す必要があり，これ

がコンテキストの退避復帰処理の必要性になっていた．チャネルコンテキストの退避とは，具体的には記述子バッファとメッセージバッファの退避領域へのコピーであり，復帰とは逆方向へのコピーである．退避復帰に要する時間はメッセージバッファにあるメッセージの総量と，記述子バッファにあるメッセージの個数に依存する．マルチコンテキスト化により，コンテキストの切替はポインタの付け変えだけで済む．この結果，高速なチャネルコンテキストの切替が実現でき，コンテキスト切替処理に要する時間はメッセージの量や個数に依存しないと期待される．

プロセッサアーキテクチャの分野では，コンテキスト切替の高速化の手法のひとつとして，複数のレジスタファイルを持つという手法が実現されている． PM における複数のチャネルコンテキストを切替える方式は，これと基本的に同じアイデアである．

表 4.2: Myrinet メモリ容量と PM チャネルコンテキストの数

Memory Size [KB]	Number of Contexts
256	5
512	13
1,024	28

コンテキストの数の上限は，記述子バッファをいくつ LANai メモリ上に確保できるかによる．表 4.2 に， PM を用いた場合の， Myrinet インターフェイスの SRAM の大きさと，実装可能なチャネルコンテキストの数の関係を示す．ここに示したコンテキストの数を越えて必要になった場合，適当に選んだコンテキストの内容をメモリに退避して再利用すれば良い．実際， 512 KB の SRAM 容量があればコンテキストの数は 13 になり，同時走行するプロセスの数としては必要十分と考えられる．もし，同時走行する並列プロセスの数が PM が持っているコンテキストの数を上回った場合は，シングルコンテキストで行なっていたようにコンテキストをメモリに退避復帰すれば良い．

4.5 改良方式の評価

4.5.1 各フェーズの処理時間

こうして改良された SCore-D の並列プロセス切替における各フェーズの処理時間を第 4.3.2 節と全く同じ方式で計測した結果を図 4.6 に示す．また，X 軸を対数目盛とし，シングルコンテキストのグラフを併せたものを図 4.7 に示す．

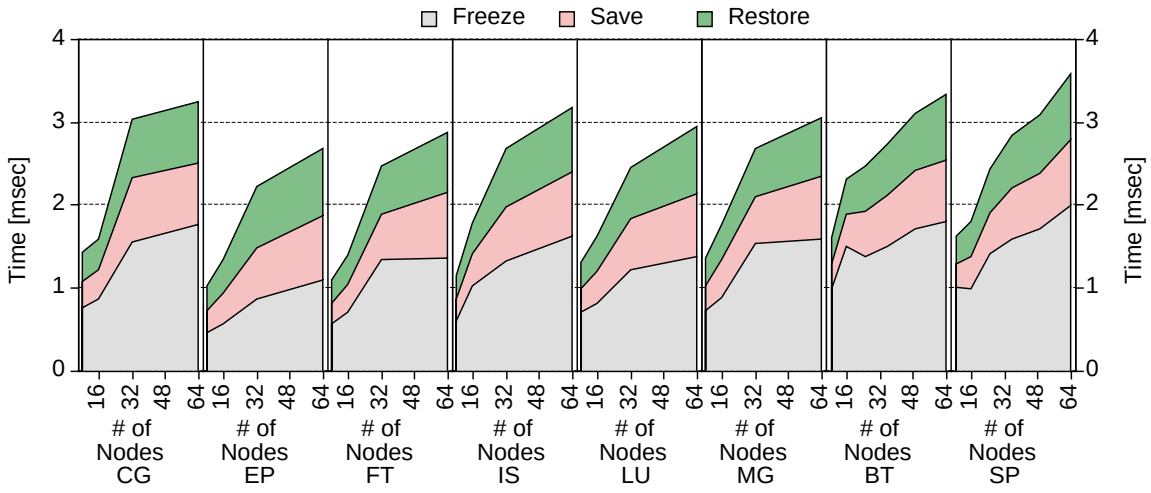


図 4.6: 各フェーズの処理時間（マルチコンテキスト）

マルチコンテキスト化の効果として期待されるように，save フェーズ及び restore フェーズの時間が特に少なくなっていることが分かる．また，これも予想されたことだが，freeze フェーズの処理時間はほとんど変化がみられない．Save フェーズ及び freeze フェーズにおいてチャネルコンテキストの退避および復帰処理がコンテキストの切替だけとなったためである．コンテキストの切替は基本的にポインタの付け変えなので，save フェーズと restore フェーズの処理時間の大半は制御構造に沿ったブロードキャストとバリア同期になる．この結果，save フェーズと restore フェーズの処理時間に差が少なくなる．表 4.1 に示したように，チャネルコンテキストの退避時間が復帰よりも遅いため，マルチコンテキスト化の効果は restore フェーズよりも save フェーズの処理時間により効果が見られる．さらに，save フェーズと restore フェーズの処理時間においてプログラムの違いによる変化が少なくなっている．これらのフェーズの処理時間がコンテキストに含まれるメッセージの数や総量に無関係になったためと考えることができる．以上の結果か

ら，並列プロセス切替の全体の処理時間がシングルコンテキストの場合に比べほぼ半分になったことが確認された．

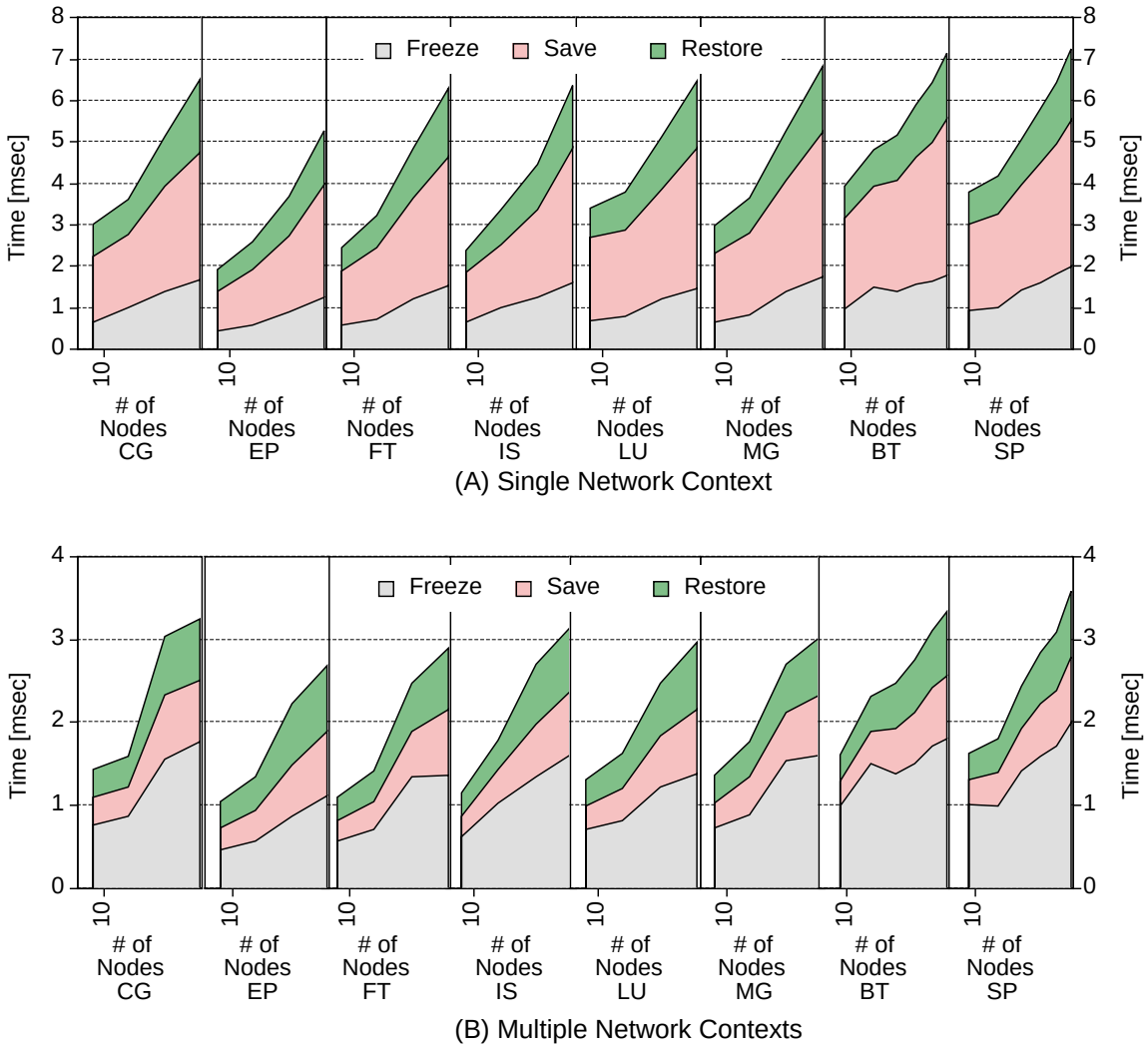


図 4.7: 各フェーズの処理時間（片対数）

図 4.6 において，マルチコンテキストの場合における並列プロセス切替時間のプロセッサ台数に対する変化が，16 プロセッサから 32 プロセッサのところで大きくなっているのが分かる．これは，マルチコンテキストでは，並列プロセス切替のための処理時間においてメッセージの量に依存する部分が少なくなったため，分散ツリー構造の 16 分木の影響がシングルコンテキストの場合に比べ顕著に現れたものと考えられる．

4.5.2 アプリケーションから見たオーバヘッド

図 4.3 や図 4.3 で判明した並列プロセス切替の処理時間からアプリケーションプログラムが被るであろうスケジューリングオーバヘッドを推定することは可能である．しかし，スケジューリングオーバヘッドの要因は単純ではない． SCore-D がスケジューリングの都度走行することによるキャッシュへの影響， freeze 処理がアプリケーションプログラムの通信に与える影響，ギャングスケジューリングの時間的なずれ（co-scheduling skew） [ADV+94] などの影響なども考えられる．そこでギャングスケジューリングの有無や時分割の時間間隔が，アプリケーションプログラムの実行時間に与える影響を別途調べることにする．

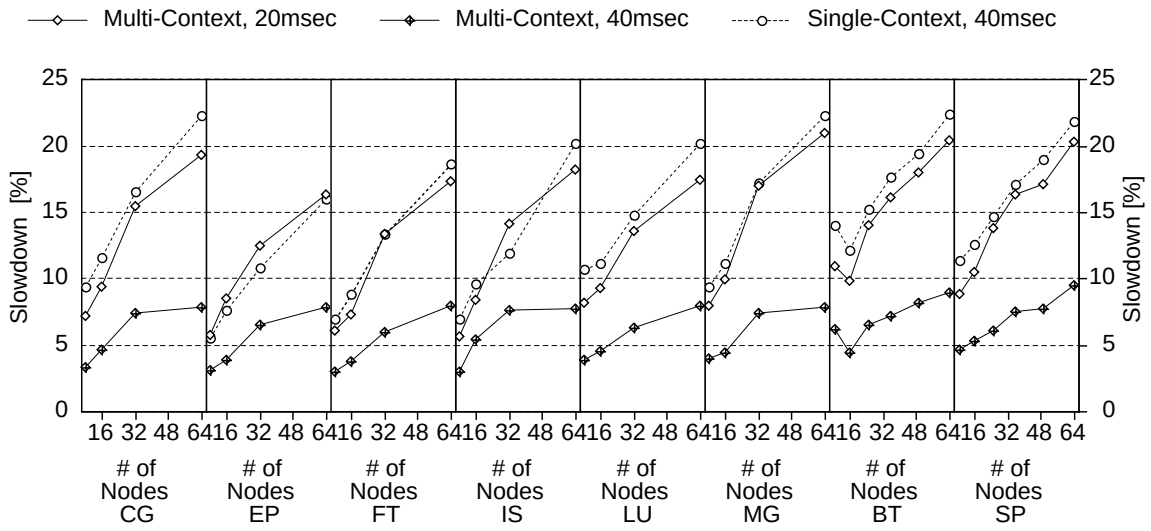


図 4.8: アプリケーションから見たオーバヘッド

アプリケーションから見たギャングスケジューリングのオーバヘッド（O）は，ギャングスケジューリングなしの場合のプログラムの実行時間（T_{NoGang}）とギャングスケジューリング下での実行時間（T_{Gang}）から次式により求めた．

$$O = (T_{Gang} - T_{NoGang})/T_{NoGang}$$

ここで計算された値は T_{NoGang} と T_{Gang} の時間差が少ない場合，計測誤差や偶然的要因の影響を受け易いことに注意を要する． T_{NoGang} の値は時分割の間隔をプログラムの実行時間よりも大きくすることで計測した．図 4.8 に計測結果を示す．プログラム実行時間（経過時間）はプログラム自身が計測したものを用いた．時分割の時間間隔は 20 $msec$,

40 $msec$ の 2 つのケースで計測した．これらの値は，計測誤差の影響を抑えるため，実際にギャングスケジューリングが適用されるであろう値（100 〜 200 $msec$ を想定）よりも小さい．縦軸はオーバヘッド（%），横軸はプロセッサ台数である．この図においていずれのグラフにおいてもシンボルは共通になっており，実線はマルチコンテキストの場合であり，破線はシングルコンテキストの場合を示す．

このグラフにおいて，時分割間隔 40 $msec$ におけるシングルコンテキストのオーバヘッドの線が，時分割間隔 20 $msec$ におけるマルチコンテキストの線とほぼ重なっている．アプリケーションで観測される時分割スケジューリングのオーバヘッドが時分割間隔に反比例すると仮定すると，マルチコンテキスト化によりオーバヘッドが半減したものと考えることができる．このことは，図 4.7 の結果を裏付けるものである．

実際に SCore-D が利用される際の時分割間隔は 100 〜 200 $msec$ 程度を想定している．図 4.8 のグラフからアプリケーションが被るスケジューリングオーバヘッドは時分割間隔 40 $msec$ の時に 10 % 以下であることから，時分割間隔を 100 $msec$ とした時のマルチコンテキストにおけるアプリケーションが被るオーバヘッドを推測すると，4 % 程度と推測される．この 4 % という値は，図 4.6 における並列プロセス切替時間が最大 4 $msec$ という値と合致している．

マルチコンテキスト化，つまり PM のメッセージバッファという大きなメモリ領域のコピーを省くということは，データキャッシュをコピーで潰さないで済み，ユーザプロセスのキャッシュのヒット率低下を抑える効果も期待できる．図 4.9 は，ギャングスケジューリングを行なわない場合と行なった場合において，2 次キャッシュのヒット率にどのような違いが生じるかを計測した結果である．ギャングスケジューリングを行なわない場合とは，時分割間隔をプログラムの実行時間よりも長く設定して計測したものである．この計測では，マルチコンテキストの場合の時分割間隔を 20 $msec$ と 40 $msec$ に，シングルコンテキストの場合は 40 $msec$ に設定して計測した． PentiumPro プロセッサには PMC （Performance Measuring Counter）と呼ばれる内部レジスタが 2 つあり，プロセッサ内部の状態をモニタできる [Int95]． 2 次キャッシュのミスは， PentiumPro の PMC レジスタ [Int95] で計測した各プロセッサの 2 次キャッシュミス回数（`L2_LINES_IN`）を最大と最小を除いて単純平均した．図 4.9 ではギャングスケジューリングを行なわない場合を 1 とした相対的な値を用いた．

この結果から，期待通り，同じ時分割間隔ではシングルコンテキストの場合に比べマルチコンテキストの場合のキャッシュミスが少ないことが分かる．さらにシングルコン

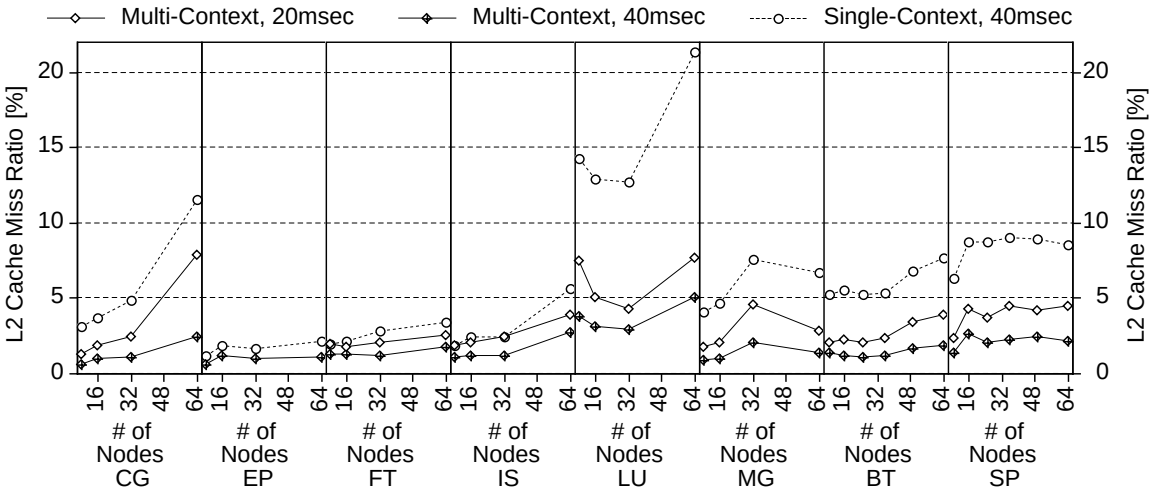


図 4.9: ギャングスケジューリングによる 2 次キャッシュのミス率の違い

テキストの場合の半分の時分割間隔（これはつまり並列プロセス切替の回数が倍になることを意味する）にした場合で，キャッシュミスの割合が時分割間隔 $40\,msec$ におけるシングルコンテキストの場合よりも少なくなっていることが確認された．

4.5.3 複数並列プロセス切替時のオーバヘッド

全節までは，単一の並列プロセスのスケジューリングオーバヘッドに関するものである．ここでは，複数並列プロセスのギャングスケジューリング性能を確認する．表 4.3 は，NAS 並列ベンチマークの 8 本のプログラムから 2 本選び，それらを同時に起動し，早く終了したものの実行時間ついて，単独で実行した場合の実行時間を 1 としたときの比を示したものである．表中，上段の数値は実行時間の比であり，下段の数値は 2 次キャッシュミス回数の比である．プロセッサ数は 64，時分割間隔は $100\,msec$ とした．また，同時に起動されたふたつのプロセスの起動が完了するまで待つようにスケジューラを改造した．これによりプロセス起動の影響を避けることができる．

CG は単独実行の場合に比べ 2.06 〜 2.08 と遅くなり方が大きい．次いで LU が 2.03 〜 2.04 となっている．キャッシュミス回数の増加を見ると CG では 20 % 以上，LU では 10 % 以上増えている．その他のプログラムの遅くなる程度は 2.00 〜 2.03 程度であり，2 つの並列プロセス走行によるスケジューリングの影響はそれほど大きくないことが分かる．これらのことから，CG や LU では他の並列プロセス走行によりキャッシュの影響が大きく，そのためにより遅くなったと考えられる．

表 4.3: 複数並列プロセススケジューリング時の処理時間比

	IS	CG	MG	FT	EP	LU	SP	BT
IS	$\frac{2.01}{1.03}$	$\frac{2.01}{1.03}$	$\frac{2.00}{1.03}$	$\frac{2.01}{1.02}$	$\frac{2.02}{1.03}$	$\frac{2.01}{1.03}$	$\frac{2.03}{1.03}$	$\frac{2.01}{1.03}$
CG	-	$\frac{2.08}{1.25}$	$\frac{2.07}{1.23}$	$\frac{2.08}{1.23}$	$\frac{2.06}{1.25}$	$\frac{2.07}{1.25}$	$\frac{2.07}{1.25}$	$\frac{2.07}{1.24}$
MG	-	-	$\frac{2.03}{1.02}$	$\frac{2.02}{1.02}$	$\frac{2.01}{1.02}$	$\frac{2.01}{1.02}$	$\frac{2.01}{1.02}$	$\frac{2.03}{1,02}$
FT	-	-	-	$\frac{2.01}{1.04}$	$\frac{2.01}{1.05}$	$\frac{2.01}{1.04}$	$\frac{2.01}{1.05}$	$\frac{2.02}{1.05}$
EP	-	-	-	-	$\frac{2.01}{1.00}$	$\frac{2.01}{1.00}$	$\frac{2.02}{1.00}$	$\frac{2.02}{1.00}$
LU	-	-	-	-	-	$\frac{2.03}{1.16}$	$\frac{2.04}{1.18}$	$\frac{2.04}{1.18}$
SP	-	-	-	-	-	-	$\frac{2.02}{1.04}$	$\frac{2.02}{1.04}$
BT	-	-	-	-	-	-	-	$\frac{2.03}{1.03}$

4.6 ギャングスケジューリングオーバヘッド低減の見通し

これまでは，SCore-D のギャングスケジューリングのオーバヘッドは時分割間隔 100 $msec$ の場合に最大 4 % であることを示した．このオーバヘッドを更に根本的に低減する可能性はあるのであろうか，また，低減するとして何をどのようにすればよいのだろうか．

4.6.1 ネットワークコンテキスト退避復帰の影響

これまでは並列プロセス切替の都度，ネットワークプリエンプションを行なう方式について評価を行なってきた．しかし，同時走行するプロセスには別な PM のチャネルを割り振ることで，最大 3 つのユーザ並列をネットワークコンテキストの退避や復帰あるいは切替なしで同時に走らせることが可能である．

図 4.10 は，並列プロセス切替の処理から save フェーズと restore フェーズを省いた場合の，アプリケーションから見たオーバヘッドのグラフである．時分割間隔は 20 $msec$ である．比較のため，時分割間隔 40 $msec$ におけるマルチコンテキストの場合の結果（図 4.8）も載せてある．この図では，およそ倍の時分割間隔でほぼ同じオーバヘッドになっており，save フェーズと restore フェーズを省くことで並列プロセス切替のオーバヘッドがほぼ半減することが分かる．

この結果から，SCore-D が PM のチャネル資源を細かく管理し，出来る限りユーザ並列プロセスに空いているチャネルを割り振り，空きチャネルが無くなった時点で，

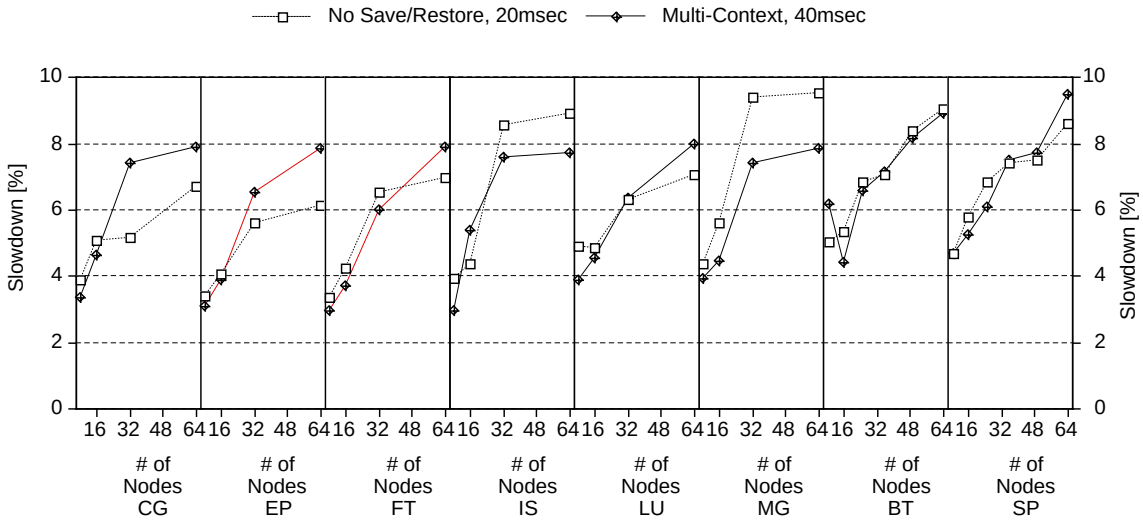


図 4.10: ネットワークコンテキスト退避復帰を行なわない場合のオーバヘッド

別なチャネルコンテキストを割り振り，さらに空いているチャネルコンテキストも無くなった時点で，チャネルコンテキストを退避／復帰する，という方式を採用することで，ギャングスケジューリングオーバヘッドをさらに低減することが可能になることが分かる．

4.6.2 SCore-D 自体の性能

前節までは，ネットワークプリエンプションの方式を改良することでギャングスケジューリングのオーバヘッドを低減する可能性について検討してきた．本節では，SCore-D 自体の性能がギャングスケジューリングのオーバヘッドに与える影響について論じ，今後も予想されるハードウェアの進歩がギャングスケジューリングのオーバヘッドに与える影響について考える．

表 4.4: SCore-D における遠隔スレッド起動性能

	SCore-D	User
Local Sync. Function Call	3.59	2.28
Remote Sync. Function Call	142.2	24.1

単位：μsec

表 4.4 は SCore-D 単体（ユーザ並列プロセスなし）における MPC++の SCore-D 用 ULT 実行時ライブラリの同期スレッド起動の時間を示したものである（第 3.2.3 節参照）． SCore-D 用の ULT 実行時ライブラリでは，PM における通信メッセージの待ちを `select` システムコールを用いている．一方，ユーザ用 ULT 実行時ライブラリではポーリングで受信メッセージを待つ．このため SCore-D 用の ULT 実行時ライブラリの性能がユーザ用に比べかなり劣っている．

表 4.5: 分散制御ツリーに沿った通信に要する時間

# of Processors	Null Op.	Restore Phase (EP)
8	225.3	307.1
16	306.3	402.5
32	514.0	730.7
64	708.0	803.6

単位：μsec

表 4.5 は SCore-D 内における分散制御ツリー（図 4.2）に沿って，ブロードキャストし Element Process オブジェクトにおいては何の処理も行なわずに直ちに結果を返し，それらの結果をバリア同期で待つまでの時間を SCore-D 単体で計測したものである（表中 “Null Op.”）．また，比較のために EP プログラムにおける restore フェーズの時間（時分割間隔 40 $msec$，マルチコンテキスト）も示した．この表から分かるように，restore フェーズに要する時間の内，7 割以上が分散制御ツリーに沿った通信に費やされていることが分かる．

Freese フェーズにおいては save や restore フェーズとは状況が異なる．なぜなら freeze フェーズのブロードキャスト時にはユーザ並列プロセスも走行しているからである． Freeze フェーズにおける本来の目的であるネットワークの定常状態を待つという時間を除いたブロードキャストとバリア同期の時間は，図 4.6 における EP の場合における freeze フェーズの時間に近いと推測される． EP において通信はほとんど発生しないからである．これらの結果から図 4.6 の結果を見直すと，ギャングスケジューリングのオーバヘッドのかなりの部分が分散制御ツリーに沿った通信時間であることが分かる．

これらの結果および考察から， SCore-D におけるギャングスケジューリングのオー

バヘッドを更に根本的に低減させるためには，基本的に `select` システムコールや Unix のプロセス切替といった OS の基本性能の改善が必要であるという結論になる．つまり SCore-D ギャングスケジューリングの性能向上には，ネットワークの高性能化はあまり貢献しないが，プロセッサの性能向上に伴う OS 基本性能の向上は大きく貢献することになる．

4.7 まとめ

本研究において実装されたギャングスケジューリングによる時分割スケジューリング実現方式は，横取り可能（preemptive）であり，ソフトウェアにより実現され，OS の変更を必要としない，という特徴がある．

ネットワークプリエンプションを用いたギャングスケジューリングは，64 プロセッサで，時分割間隔を $100\,msec$ とした場合，最大でも 4 % 程度のオーバヘッドであることが判明した．また，マルチコンテキスト化による改良でアプリケーションが被るオーバヘッドは半減された．ちなみに $100\,msec$ という時間間隔は初期の Unix の時分割間隔の値である [LMKQ89]．

このオーバヘッドは PM のチャネル資源を細かく管理することで半減する可能性を示した．一方，マイクロプロセッサの性能は今後も向上を続けると考えられ，ギャングスケジューリングのオーバヘッドもプロセッサの性能向上とともに低減するであろう．その結果，ギャングスケジューリングのオーバヘッドは現在の逐次計算機における時分割スケジューリングのように，ほとんど問題にされない程度になるかもしれない．

第 5 章

ギャングスケジューリングと非同期コスケジューリングの比較

並列計算機上で時分割スケジューリングを効率的実現する方法として，現時点ではギャングスケジューリングと非同期コスケジューリングが提案されている．前章で，ネットワークプリエンプションを用いたギャングスケジューリングを提案し，評価をおこなった．本章においては，非同期コスケジューリングを実装し， SCore-D によるギャングスケジューリングとのスケジューリング性能の比較を試みる．そして，それぞれの時分割方式の特徴について議論する．

5.1 ギャングスケジューリングと非同期コスケジューリング

ギャングスケジューリングは，細粒度の並列処理において有効とされている [Ous82, GTU91, FR92] のに対し，同期コスケジューリングは粒度が粗く不平衡な負荷の場合において有効 [DAC96, ADCM98] とされている．

これまでに， dynamic coscheduling [Sob97] と implicit coscheduling [DAC96, ADCM98] が非同期コスケジューリング方式として提案されている．これらの方式における基本的なアイデアは，通信メッセージの受信において，ある時間まではビジー（スピン）ウェイトで待ち，それ以降はブロックして待つ，という *two-phase spin-block* [DAC96] 方式に基づいている．ここで，ビジーウェイトの待ち時間が固定なものを *two-phase fixed spin-block* (TPFS) 式コスケジューリング，ビジーウェイト時間が動的に変化するものを *two-phase adaptive spin-block* (TPAS) 式コスケジューリングと呼ばれている [DAC96].

非同期コスケジューリングにおける仮説は，あるビジーウェイト時間 S があって，その時にシステムのスループットが最大になり，そのときの複数ジョブ投入時のスループットは理論的な値を上回る可能性がある，というものである（図 5.1）．一方，理想的なギャングスケジューリングにおいてはシステムのスループットは常に理論的な値と同じである．

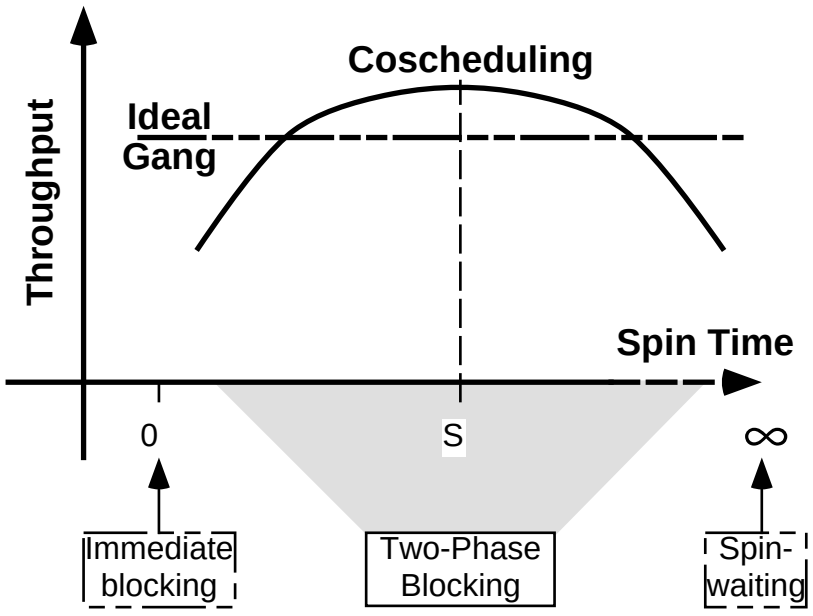


図 5.1: Spin-wait Time and Throughput

ビジーウェイト時間が短過ぎると，プロセス切替の回数が増え，その分オーバヘッドとなる．一方，ビジーウェイト時間が長過ぎると，プロセス切替の回数は減るが，ビジーウェイトしている時間が増え，その分プロセッサ時間を浪費する．このように非同期コスケジューリングでは，ビジーウェイト時間を適切に求める必要がある．文献 [Sob97, DAC96, ADCM98] にはそれぞれのモデルに基づいたビジーウェイト時間の決定方法が示されている．

5.2 比較の方法

本章において，以下に示すアプローチにより，ギャングスケジューリングと非同期コスケジューリングの比較をする．

アプリケーション

一般的な並列処理アプリケーションとして NAS 並列ベンチマーク [BBLS93] から，通信パターンが異なる EP， CG， FT， LU の 4 本のプログラムを選定した．これは，これまでのコスケジューリング方式の評価 [Sob97, ADCM98] がそれぞれ独自に開発された言語やアプリケーションを用いているのと違い，標準通信ライブラリを用いた第 3 者により開発されたプログラムを用いて比較することが公平であると考えたからである．

また，非同期コスケジューリングは BSP （Bulk Synchronization Program）モデルに基づいた並列プログラムにおいて有効とされている [DAC96, ADCM98] が， NAS 並列ベンチマークのようなデータ並列アプリケーションでどの程度の非同期コスケジューリングの効果が出るのかといった興味もある．

非同期コスケジューリング方式

ここでは非同期コスケジューリング方式として最も単純かつ基本的なモデルである two-phase fixed spin-block （以下 TPFS を略記）方式を採用する [HTOI98a]．また，非同期スケジューリングにおけるビジーウェイト時間をモデルに基づいて求めるのではなく，最初にビジーウェイト時間を盲目的かつ網羅的に変化させ，その中から最も効果の高いものをビジーウェイト時間として選び，その時間をもってギャングスケジューリングと比較することとした．

図 5.2 には， MPICH-PM[OTHI98b, TOHI98] における受信メッセージの処理を簡

```
1 while (!(packet = recvNetwork()));
```

図 5.2: ビジーウェイトによる通信メッセージの待ち

略化したコードを示したものである．一方，図 5.3 は， TPFS コスケジューリングを実現するにあたり，図 5.2 の処理を変更した結果である．双方の図において， recvNetwork() はメッセージを受信するための関数であり，未受信の通信メッセージが存在しないときには NULL を返し，そうでない時には，受信バッファ中のメッセージの先頭へのポインタを返す．図 5.2 における関数 sleep_until_recv() は， OS へのシステムコールであり，通信メッセージが受信されるまでブロックして待つという関数である．実際には PM[手塚 96, THI96, THIS97, TOHI98] が提供する関数である．

```
1 spin = SPIN_COUNT;
2 while (!(packet = recvNetwork())) {
3   if (--spin == 0) {
4     sleep_until_recv();
5   }
6 }
```

図 5.3: Two-Phase Fixed Spin-Block による通信メッセージの待ち

図 5.3 におけるビジーウェイトループの一回に要する時間は 0.65 μsec であった．また SPIN_COUNT の値をゼロに設定することで通信メッセージの受信を常にブロックする方式に，また十分長い値に設定することでビジーウェイトを実現することができる．

ギャングスケジューリング方式

第 4 章で提案，実装した SCore-D ギャングスケジューラ（マルチコンテキスト）を用いて比較を行なう．ギャングスケジューリングで動かすプログラムは TPFS コスケジューリングと全く等価であるが，ビジーウェイト時間を 6 sec と十分長く設定した．ギャングスケジューリングの計測に TPFS コスケジューリングのプログラムを用いたのは，ビジーウェイト時間を十分長く設定することで実質的にビジーウェイトと同じ結果を得ることができるという理由と，少しでも異なるプログラム同士を比較することでキャッシュなどの影響が評価結果に反映されることを防ぐという理由による．特に PentiumPro プロセッサの場合，倍精度小数点数が 8 バイトに整列していないことによる性能低下が顕著である．

5.3 評価

以下の評価において TPFS コスケジューリングの計測もギャングスケジューリングの計測も全て SCore-D を用いた．TPFS コスケジューリングの場合，時分割間隔をプログラムの実行時間よりも十分長く（約 6 *sec*）設定して計測した．また，TPFS コスケジューリングで複数の並列プロセスを投入する際，第 4.6.1 章で用いたのと同じ方法，つまり異なる PM のチャネルをそれぞれの並列プロセスに割り振る方法，を用いた．PM は 4 つのチャネルをサポートしているため，最大で 3 つの TPFS コスケジューリングプログラムを走らせることが可能である．また，プログラム起動の影響が計測結果に及ばないよう，投入した全てのプログラムの起動が完了してから計測を開始した．計測を容易にするため，同時に投入するプログラムは全て同じプログラムである．OS によるキャッシュの影響を避けるため，同時に実行されるプログラムの実行ファイルはそれぞれ異なるコピーを起動した．スケジューリングの性能は，ビジーウェイトで通信メッセージを待つ単一プロセスを，スケジューリングなし（実際には十分長い時分割間隔のギャングスケジューリング）で走らせた場合の実行時間を 1 とした時の相対性能で表現した．

5.3.1 TPFS コスケジューリング

図 5.4，図 5.5，図 5.6，図 5.7は，NAS 並列ベンチマークプログラム CG，EP，FT，LU のそれぞれについて X 軸にビジーウェイトのスピン回数（対数目盛），Y 軸を相対性能とした時のグラフである．これらのグラフの凡例で，例えば “64-2” とあるのは 64 プロセッサで 2 つの並列プロセスを同時実行した場合を示す．

EP の TPFS 非同期コスケジューリング（図 5.4）は，ほぼ理想的なスケジューリング性能を示していることが分かる．EP ではほとんど通信が発生しないため，TPFS コスケジューリングの結果というよりは Unix によるローカルスケジューリングのみが効いていると考えられる．10^3 から 10^4 というように多くのスピン回数においてわずかの性能低下を見ることができる．

一方，CG（図 5.5）や LU（図 5.6）においては，スピン回数が多くなるにつれスケジューリング性能の低下する傾向が顕著である．また，いずれのスピン回数においても理論値より実測値が上回ることはなかった．

FT（図 5.7）においてはスピン回数に関わらずかなりの性能低下が見られる．特に

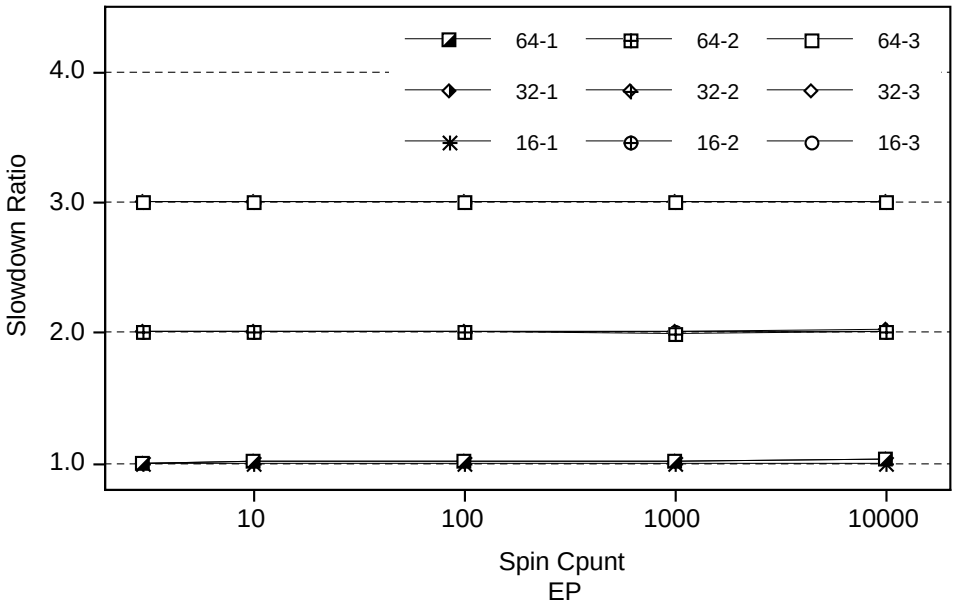


図 5.4: TPFS Coscheduling (EP)

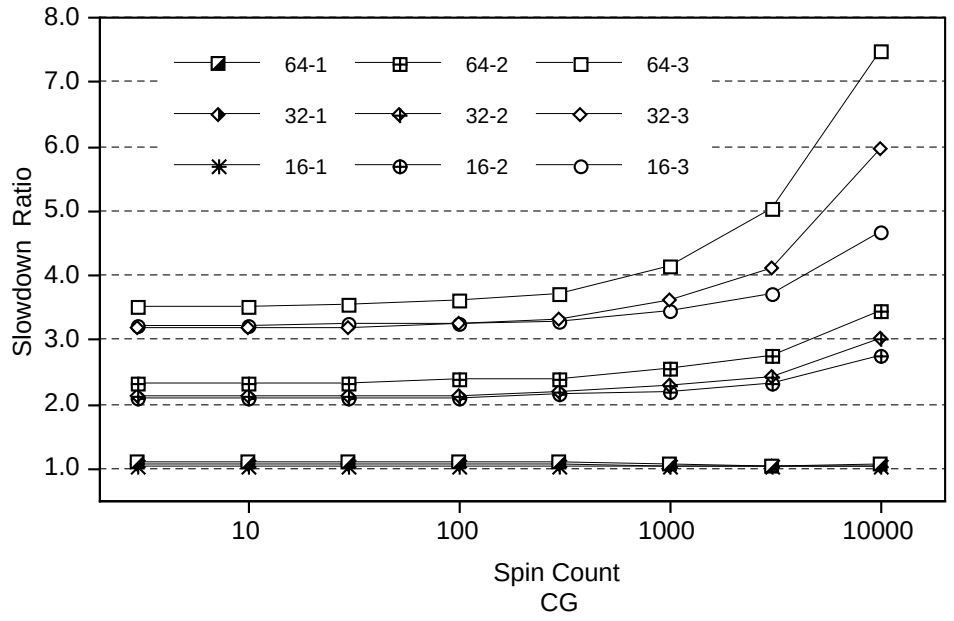


図 5.5: TPFS Coscheduling (CG)

並列プロセス数が多いほど，性能低下が激しい．FT は CG や LU に比べ大量の通信が発生する．FT の基本的な通信パターンは MPI の全対全通信であり，大きな作業領域を全てのプロセッサ間で交換し合う．図 5.8 は並列プロセスの経過時間に対する PM の再

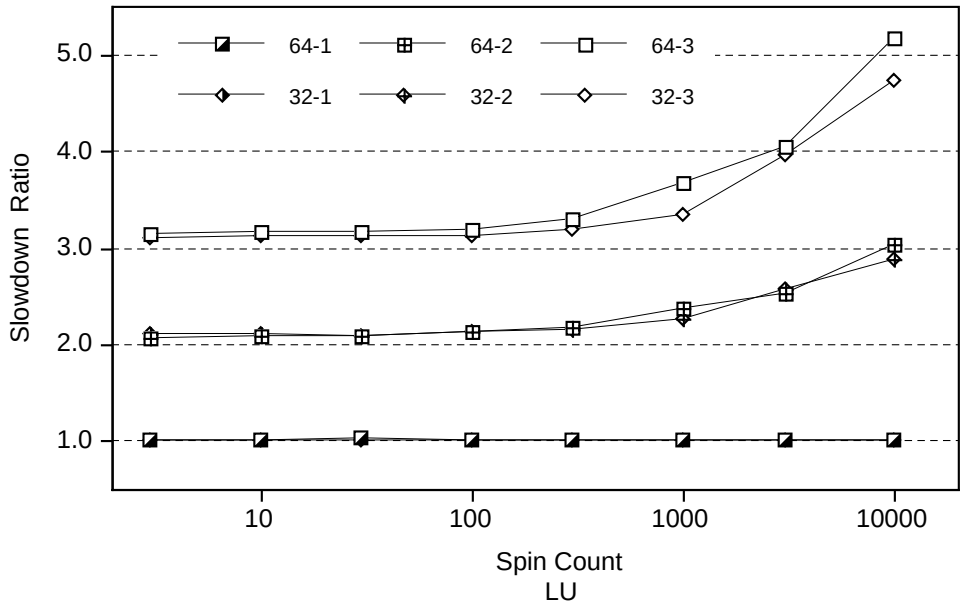


図 5.6: TPFS Coscheduling (LU)

送回数の頻度を示したものである．再送の頻度は単一並列プロセスの時に比べ投入する並列プロセス数が多い程，PM の再送頻度も高くなることが分かる．PM において大量の再送が発生しているということは，恐らくはネットワークが非常に混雑している事を意味し，送信バッファも満杯になる可能性が高いと推測される．

MPICH-PM では，デッドロックを防ぐために，送信バッファが満杯になり，それ以上送信できなかった場合，受信を試みるようになっている（図 5.9）．TPFS コスケジューリングにおいて，送信しようとする相手プロセスがスケジューリングされていない場合，大量をデータを送ろうとしても相手が受けとらず，その結果送信バッファが満杯になり，図 5.9 にあるような送信バッファの空きを長時間待つ可能性がある．図 5.7 に見られる FT の TPFS コスケジューリングの性能の低さはこのような原因によるものと考えられる．この問題は文献 [ADCM98] においても指摘されている．

この状況を避けるためには，図 5.10 に示すように，メッセージの受信だけでなく送信においても TPFS ブロックが必要である．図 5.10 のメッセージ受信関数 `receiveMessage()` においてブロックして通信メッセージを待つことはできない．デッドロックの可能性が生じるためである．あるプロセスがメッセージを送信しようとして送信バッファが取れなかった場合，他の全てのプロセスはその時点でそのプロセスからのメッセージを待っているかもしれないからである．このようなデッドロックを回避するため，図 5.10にお

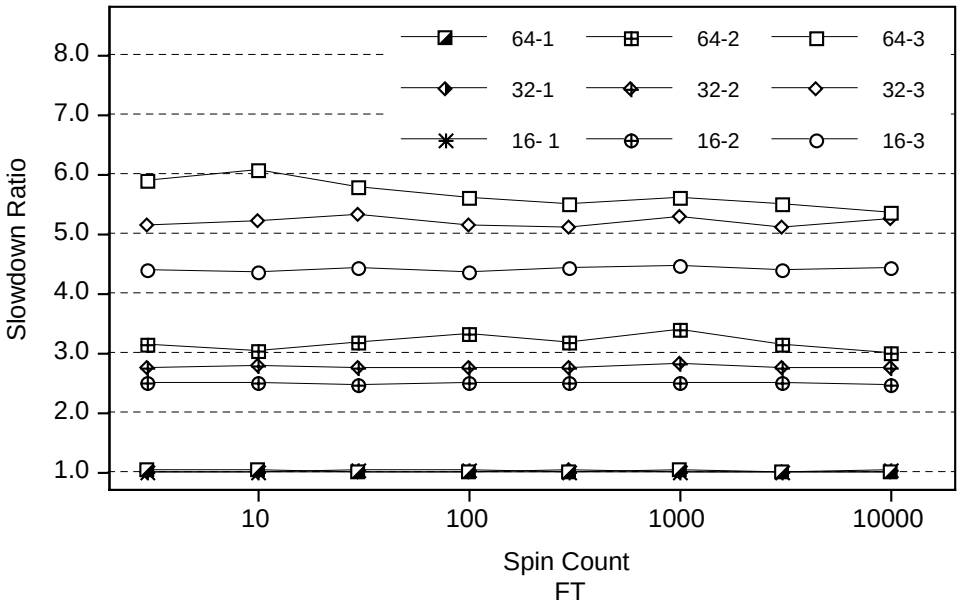


図 5.7: TPFS Coscheduling (FT)

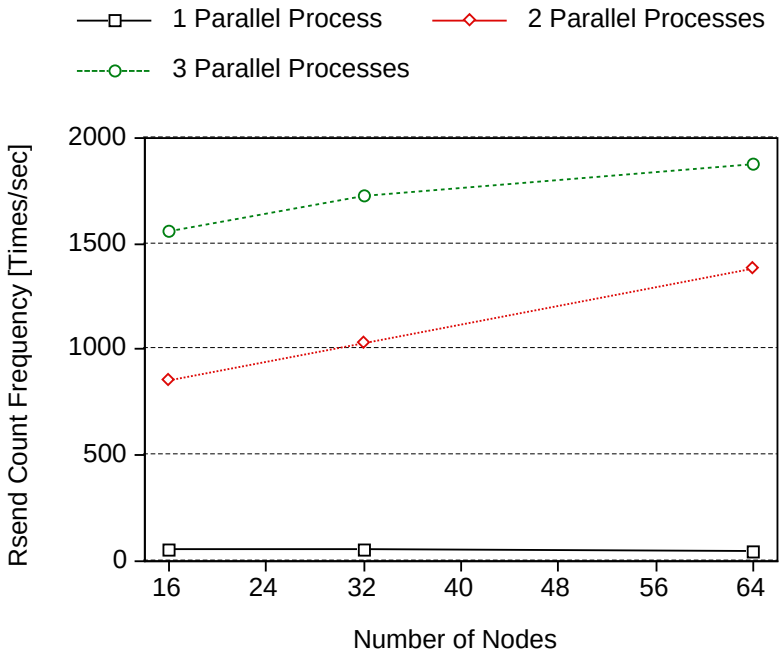


図 5.8: FT における再送の頻度

いては送信バッファが確保できなかった時に，送信バッファの空き待ちと受信メッセージの待ちの両方をポストし，そのどちらかあるいは両方を関数 `sleep_until_recv_or_send()`

```
1 while (!(buf = allocSendBuf(len))) {
2   receiveMessages();
3 }
```

図 5.9: MPICH-PM におけるメッセージ送信

においてブロックして待っている．

```
 1 spin = SPIN_COUNT;
 2 while(!(buf=allocSendBuf(len))) {
 3   if ( --spin ) {
 4     receiveMessages();
 5   } else {
 6     postSend( len );
 7     postRecv();
 8     sleep_until_recv_or_send();
 9     spin = SPIN_COUNT;
10   }
11 }
```

図 5.10: Two-Phase Spin-Block によるメッセージ送信

PM において送信をブロックして待つという機能は提供されていないため，図 5.10 に示したプログラムを実際に検証することはできかった．

5.3.2 ギャングスケジューリングとの比較

図 5.11，図 5.12，図 5.13 は， SCore-D によるギャングスケジューリング（時分割間隔は 200 $msec$）と， TPFS コスケジューリング（スピン待ちの回数は 100 に設定．これは約 65 μsec に相当）のスケジューリング性能を比較したものである．図 5.11 は単一並列プロセス，図 5.12 は 2 つの並列プロセス，図 5.13 は 3 つの並列プロセスを投入した時のグラフである．これらのグラフにおいて Y 軸のスケールが異なっていることに注意を要する．

まず FT に関しては， FT の TPFS コスケジューリング性能は他に比べ顕著に悪い．あまりに極端に悪いので，図 5.12 や図 5.13 では他のプログラムの結果の比較が可能なように意図的に FT の値をグラフの枠からはみ出すようなスケールにした． FT の性能の悪さに関しては既に第 5.3.1 章で述べた通りである．

次に，図 5.11 において顕著なのは単一並列プロセスのスケジューリングにおいて CG の TPFS コスケジューリングがギャングスケジューリングに比べ，目立って悪い結果となっている．どうしてこのような結果となったのであろうか．

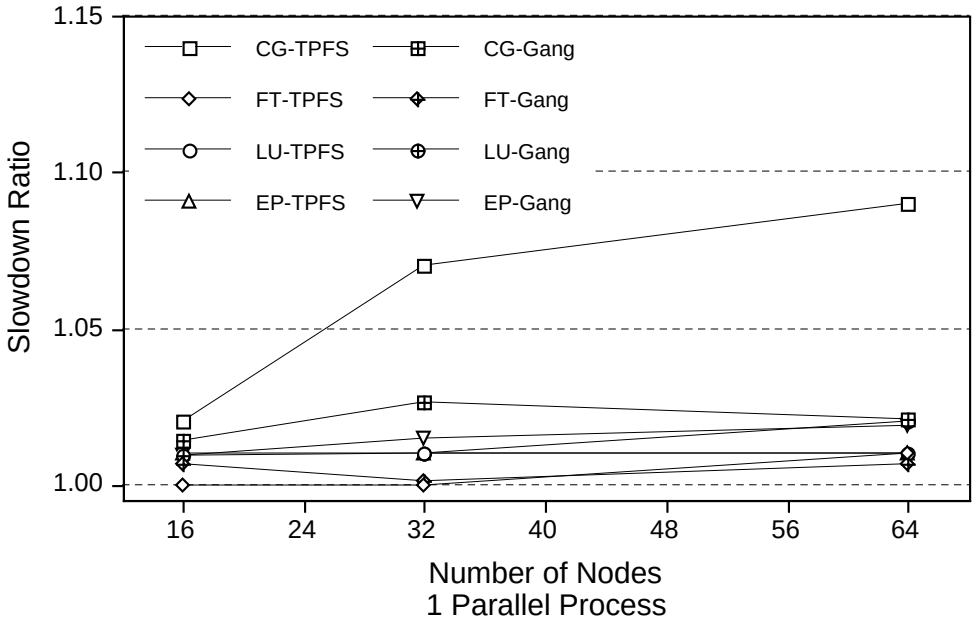


図 5.11: ギャングスケジューリング対 TPFS コスケジューリング（1 並列プロセス）

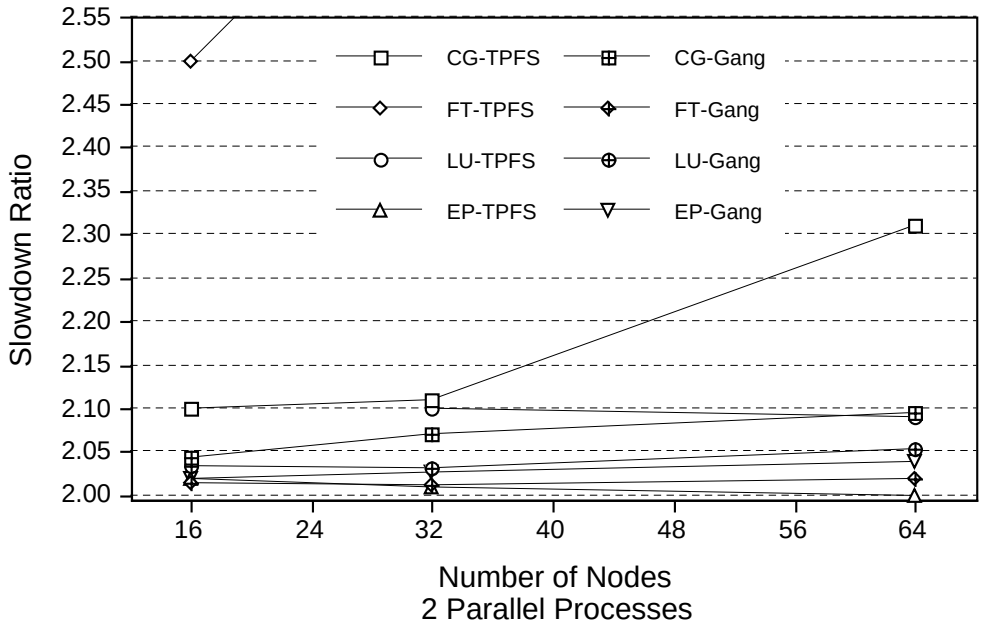


図 5.12: ギャングスケジューリング対 TPFS コスケジューリング（2 並列プロセス）

図 5.14 は CG の TPFS コスケジューリング時における，voluntary なプロセス切替頻度（voluntary プロセス切替回数を CPU 時間で割った値），involuntary なプロ

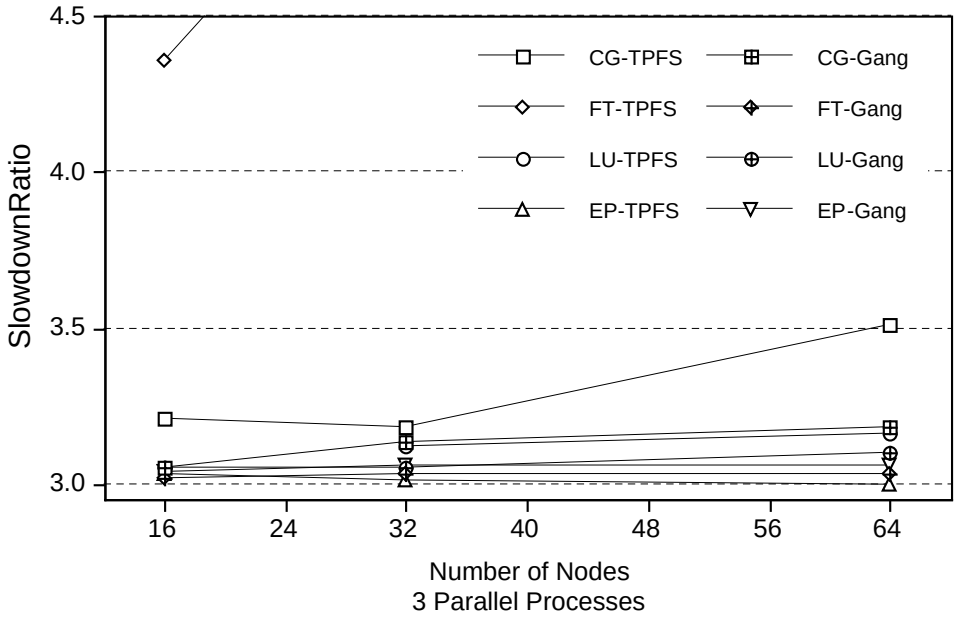


図 5.13: ギャングスケジューリング対 TPFS コスケジューリング（3 並列プロセス）

セス切替頻度（involuntary なプロセス切替回数を CPU 時間で割った値），およびプロセッサ利用率（経過時間に対する CPU 時間の割合）について，スピン回数を変化させて計測したものである．これらの値の収集は SCore-D が行なった．ちなみにプロセッサ数は 64 である． Voluntary なプロセス切替（VCW）とは，この場合，通信メッセージを待つためにブロックすることである．一方， involuntary プロセス切替（IVCW）とは，時分割間隔時間を経過したために生じたプロセス切替である．図 5.14 において，単一並列プロセスの場合， VCW の頻度はスピン回数が多くなると減り， IVCW の頻度はスピン回数が多くなると増える．また，いずれのプロセス切替頻度も，スピン回数の変化に対し，穏やかに変化する．これらの事から， CG においては通信メッセージの受信待ちが多く発生し，また，受信待ちの時間は幅広いスペクトルを持っていると考えることができる．このことは非同期スケジューリングにおいてスピン待ちの時間を動的に変化させる必要性を示すものと考えられる．

図 5.11 の計測において，スピン回数は 100 に設定されており，この時のプロセス切替頻度は毎秒 600 回以上もある．一方 SCore-D によるギャングスケジューリングにおいては， CG のプロセス切替頻度は毎秒 20 〜 30 回程度しかない．以上のことから，図 5.11 における CG の TPFS コスケジューリング性能の悪さは多発するプロセス切替から

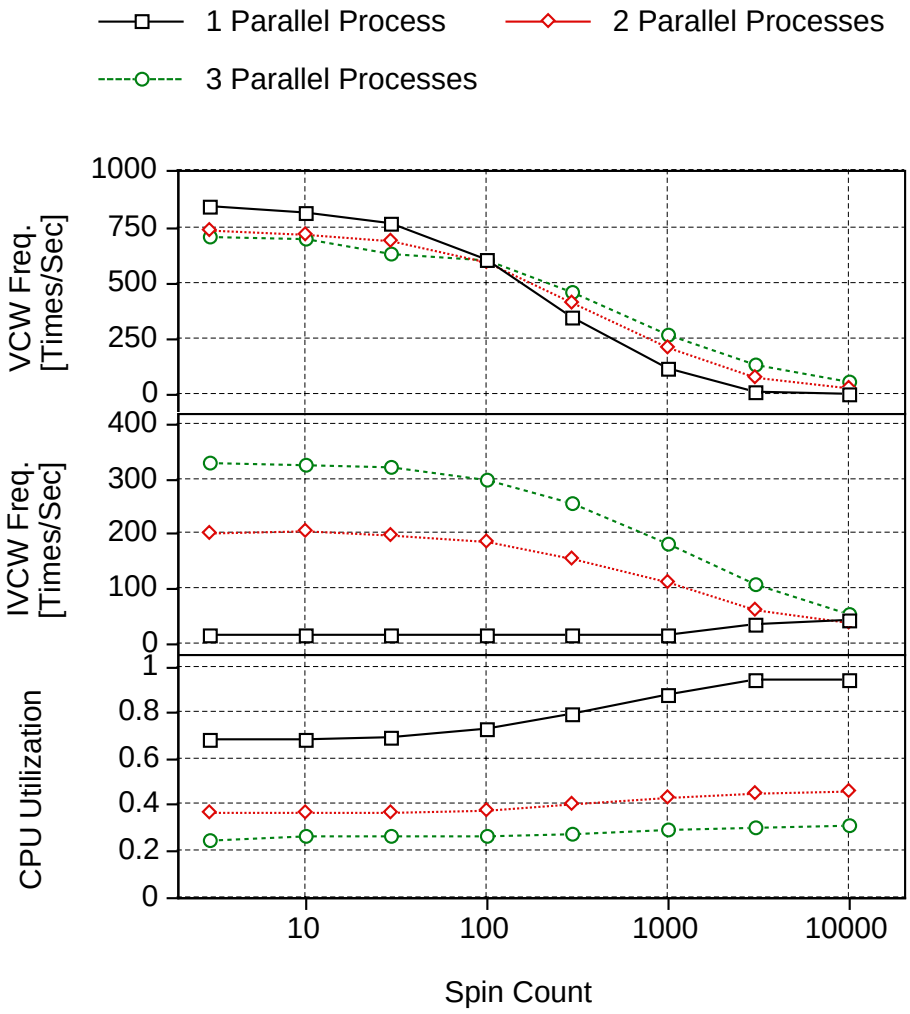


図 5.14: CG におけるコンテキスト切替頻度と CPU 利用率の変化

生じるオーバヘッドと考えることができる.

一方, SCore-D によるギャングスケジューリングでは, CG, FT, LU, EP 全てのプログラムにおいて, TPFS コスケジューリングに比べプログラムの通信パターンやプロセッサ数の影響を受けず, 安定したスケジューリング性能を示している. TPFS コスケジューリングが明らかに SCore-D のギャングスケジューリングを上回っているのは EP だけであった.

5.4 まとめ

SCore-D のギャングスケジューリングと TPFS コスケジューリングの性能を NAS 並列ベンチマークというデータ並列アプリケーションを用いて比較した. その結果, EP

のような BSP モデルに近いアプリケーションでは TPFS コスケジューリングの性能が SCore-D を上回った．LU での性能はほぼ同等であった．しかしながら，TPFS コスケジューリングでは CG に関してはスケーラビリティに問題が発見され，FT に関しては極端に悪い結果となった．同じ PCC と SCore-D を用いてはいるが，全く独立した別の評価においても同様の結果 [福地 98] が得られている．FT に関してはブロックするメッセージ送信を用いることで性能改善が期待できるが，CG に関してはスピンによる待ち時間を動的に変化させる必要性があることが判明した．一方，SCore-D によるギャングスケジューリングでは，プロセッサ台数やアプリケーションの通信パターンの影響を比較的受け難い．

Dusseau らによれば，implicit coscheduling の評価結果はスループットの理論値に対し約 +20 % 〜 -20 % と報告されている [ADCM98]．本章で得られた結果は彼らの結果と矛盾するものであろうか．まず計測に用いたプラットフォームの違いが考えられる．Dusseau らの計測は 16 台の UltraSparc-I を Myrinet で接続した構成のワークステーションクラスタを用いている．図 5.11，図 5.12，図 5.13 に示したように，例えば CG を用いた TPFS コスケジューリングのスケーラビリティの問題はプロセッサ数が 32 台を越えたあたりから顕著になっている．また，彼らが用いた並列言語 Split-C で記述された計測アプリケーショは，MPI で書かれた NAS 並列ベンチマークと通信や同期のパターンが異なっている．

本章での比較結果から，ギャングスケジューリングの性能が非同期コスケジューリングに比べ同等あるいは上回る，と結論することは正しくない．計測のために設定したスピン待ち時間に「スィートスポット」が存在しサンプリングに埋もれてしまった可能性や，NAS 並列ベンチマークとは異なる通信や同期のパターンを持つアプリケーションにおける可能性には言及していないからである．しかしながら，本章における比較結果は，少なくとも TPFS コスケジューリングが万能ではないことを示すものと考えることができる．

本章では，非同期コスケジューリングによる並列プロセスのスケジューリング性能にのみ着目した．Dynamic coscheduling[Sob97] で主張されている逐次プロセスと並列プロセスの混在する環境でのスケジューリング性能については，本研究が対象とする範囲を越えている．

第 6 章

大域状態の検出

第 4 章では，ネットワークプリエンプションによるギャングスケジューリングが実用的なオーバヘッドで実現可能であることを示した．ギャングスケジューリングは並列計算機において時分割スケジューリングを可能とする．しかしながら，時分割スケジューリングだけでは，効率的な対話処理を実現するには至らない．

対話処理，つまりユーザである人間の応答時間は非常に幅がある．並列プロセスがユーザからの応答を待っている間に，有効な計算が進まないのであれば，その並列プロセスのスケジューリングは無駄になる．本研究における「効率的な対話処理」とは，このような無駄なスケジューリングをしないために，並列プロセスの計算が進んでいるのか，それとも何らかの応答を待っているのか，という状態を検知し，その結果をスケジューリングに反映させようとするものである．

6.1 効率的な対話並列プログラミング環境

時分割スケジューリングはバッチスケジューリングに比べ短い応答時間が期待できることから対話処理を可能とする．しかしながら，ディスクなどの I/O 機器と異なり，ユーザの応答時間は変化に富んでいる．並列プログラムは場合によっては数時間もユーザからの応答を待ち続けるかもしれない．このような場合，並列プログラムがユーザからの応答を待つ以外に，意味のある計算をしているのでなければ，そのような並列プロセスをスケジューリングする，つまりプロセッサ資源を割り当てる，ことは明らかに無駄である．もし，この時，他に意味のある計算をしている並列プロセスが存在するならば，そのプロセスをスケジューリングすべきであり，そうすることでシステムのスループットも向上する．

この状況は基本的に逐次計算機においても同様である．例えば Unix では，ユーザからの入力待ちは，最終的には I/O というシステムコールが OS 内でブロックするという事象になり，これにより OS はシステムコールを発行したプロセスが「待ち」状態であることを知り，そのプロセスをスケジューリングの待ち行列から外す [Bac86, LMKQ89]. マルチスレッドに対応する Mach OS はスレッド単位でシステムコールやスケジューリングを管理している [Bla90]．この結果，スレッドをプロセスと読み変えれば基本的な枠組は Unix と同じである．

ギャングスケジューリングにおいては，並列プロセスはスケジューリングの単位である．並列プロセスはプロセスの集合であり，個々のプロセスの状態をどのように全体としての並列プロセスの状態として反映させるかが問題となる．

例えば，どれかひとつでもプロセスがアイドル状態になったら，並列プロセスもアイドル状態になるものとする．こうした場合，個々のプロセッサが単位時間当り I 回アイドルになったとすると，並列プロセスは単位時間当りおよそ $I \times P$ 回アイドルになることになる．一方，ひとつのプロセスがアイドルになったことによるプロセッサ利用率の低下は $1/P$ である． P が大きい場合，並列プロセスの単位時間あたりのアイドル回数は増えるが，プロセッサ利用率の低下は小さくなる．結局，僅かなプロセッサ利用率低下を抑えようとして，頻繁に並列プロセスの状態を変化させなければならないことになる．第 4.1 節に示したようにギャングスケジューリングによる並列プロセス切替は数 $msec$ 必要であり，頻繁な並列プロセス切替は大きなオーバヘッドとなることが懸念される．また，並列プロセス切替している間にもプロセスはアイドル状態ではなくなってし

まう可能性もあり，このような場合には並列プロセス切替が無駄になってしまう．

一方，全てのプロセスがアイドル状態になった場合のみ，並列プロセスをアイドル状態とする方式が考えられる．全てのプロセスがアイドルであるということは，その並列プロセスでは有効な計算は行なわれていないことを意味する．このため，このような状態に陥った並列プロセスに割り当てられたプロセッサ資源を，他の並列プロセスに譲り渡すことは有効と考えられる．以下本章では，このモデルに基づいて並列プロセスの状態を遷移させることを考える．

では，どうやって全てのプロセスがアイドルになったことを検知すればいいのだろうか？第 2.3.2 節で述べたように，この問題は「分散プロセスの大域停止検出問題」（global termination detection of distributed processes）として知られている．同期通信の場合は大域停止の問題は容易であるが，PM（第 3.2.1 節）のような非同期通信においては大域停止状態を検出することは単純ではない．一方，これまでに提案されている方法（例えば，[Mis83, CL85, KMY94, 六沢 97]）をここで議論している問題に当てはめようとすると，これらの方法ではいずれもユーザ並列プロセス側で検知しようとしていることが問題となる．つまり，ユーザ並列プロセスの大域アイドル状態はスケジューラ側で検知し，その結果としてユーザプロセスの状態を遷移させなければならない．

大域停止問題の難しさは，基本的にネットワーク中に存在するメッセージは観測できないという仮定に起因している．これまでに提案されたいくつかの方法 [Mis83, CL85, KMY94] では，例えば，特殊なメッセージを送ることでネットワーク中にメッセージが存在しないことを確認している．第 4.1 節で提案したネットワークプリエンプション方式によるギャングスケジューリングでは，ネットワークに存在する全てのメッセージを退避する．従ってネットワーク中のメッセージの有無は，退避されたネットワークコンテキストを検査することで知ることが可能である．この方式はネットワークプリエンプションを行なう主体がスケジューラであることから，ネットワーク中のメッセージの有無をスケジューラが知ることができ，その結果としてユーザ並列プロセスの大域状態をスケジューラが知ることが出来るという利点がある．

並列プロセスにおけるプロセスの待ちは，実際のところ通信メッセージを待つ場合の方が圧倒的に多いが，システムコールの結果を待つという局面も考えられる．一般に知られる分散プロセスの大域停止問題では，通信メッセージのみに着目している．システムコールの待ちと受信メッセージの待ちとでは，どのような違いがあるのであろうか．

ここで，並列プロセスを構成する個々のプロセスは走行状態あるいはアイドル状態と

いう 2 つの状態を持つとする．初期状態は走行状態である．走行状態からアイドル状態への遷移は，任意のタイミングで生じる．逆に，アイドル状態から走行状態への状態遷移は，システムコールが完了した，あるいは通信メッセージを受信したことで遷移するものとする．また，システムコールの完了待ちの前には必ず対応するシステムコールが発行されているものとする．ある並列プロセスにおいて，ネットワーク中にメッセージが存在せず，ひとつ以上のプロセスがシステムコールの完了待ちであった場合，この並列プロセスはシステムコールの完了とともに走行状態になる．また，ネットワーク中にメッセージが存在せず，全てのプロセスがアイドル状態あった場合，この並列プロセスは全体として永遠にアイドル状態であり続ける．このような場合としては，具体的には並列プロセスがデッドロックに陥った場合が考えられる．

ある並列プロセスが大域的な停止状態にあり，一つ以上のプロセスがシステムコールの完了を待っている状態を「並列プロセスの（大域的）アイドル状態」と定義する．また，未完了のシステムコールが存在せず，ネットワーク中にメッセージも存在せず，かつ，全てのプロセスが受信メッセージ待ちであった場合は「並列プロセスの（大域的）停止状態」と定義する． OS という立場で見てみると，アイドル状態にある並列プロセスをシステムコールの完了以前にスケジューリングすることは明らかにプロセッサ資源の無駄である．また，停止状態にある並列プロセスはこれ以上計算が進まないことを意味するため，資源の有効利用という立場から削除される必要がある．

以下，ネットワークプリエンプションを応用した並列プロセスの大域状態の検出方法について述べる．

6.2 大域状態の検出の実装

SCore-D はネットワークプリエンプションによりユーザ並列プロセスに起因するメッセージがユーザに割り当てられた仮想ネットワーク中に存在するか否かを検出することができる．また，第 3.4.2 節で示したように SCore-D はシステムコールの枠組を持っているため，ユーザ並列プロセスから SCore-D に対するシステムコールの待ちの有無を検出することができる．

次の問題は，ユーザプロセスのアイドルをどうやって検出するかということである．第 2.3.2 節で述べたように，ユーザレベル通信においてメッセージはポーリングやビジーウェイトで待つ．このようなプロセスに対し外部から待ち状態であるかどうかを一般に

知ることは単純ではない．この問題に対して SCore-D では c_area （第 3.4.2 節参照）にユーザプロセスがアイドルであるかどうかを示すフラグを設け，ユーザプロセスが自分自身の状態に応じてこのフラグを設定するようにした．

実際には，このフラグは MPI のような通信ライブラリや言語の実行時ライブラリが設定するものであり，ユーザが直接操作する必要はほとんどない． MPICH-PM （第 3.2.4 節）や MPC++ の実行時ライブラリである ULT （第 3.2.3 節）においてこのフラグは設定されるようになっている．

```
 1 void idle_loop( void ) {
 2   char *messagep;
 3
 4   while( 1 ) {
 5     if( !pmReceive( channel, &messagep ) ) {
 6       c_area->idle_flag = FALSE;
 7       process_message( messagep );
 8       pmPutReceiveBuf( channel );
 9       break;
10     }
11     if( c_area->syscall_cell->status == DONE ) {
12       c_area->idle_flag = FALSE;
13       c_area->syscall_cell->status = FREE;
14       enqueue_thread( c_area->syscall_cell->thread );
15       break;
16     }
17     c_area->idle_flag = TRUE;
18   }
19   return;
20 }
```

図 6.1: アイドルループのプログラム例

図 6.1 は， MPICH-PM や MPC++ の ULT 実行時ライブラリにおいて，実際に行なわれている処理を簡略化したプログラムである． pmReceive() は PM が提供するメッセージ受信のための関数であり，メッセージが到着していればメッセージへのポインタを返す． PM 関数 pmPutReceiveBuf() は受信バッファにおけるメッセージ領域の解放を指示している．変数 c_area は第 3.4.2 節で説明した SCore-D プロセスとユーザプロセスで共有される領域へのポインタであり，ここにある変数 idle_flag がユーザプロセスがアイドル状態であることを示すフラグに相当する．また syscall_cell は SCore-D に対するシステムコールの情報を受け渡すためのシステムコールセル（第 3.4.2 節参照のこと）である．

ここで注意しなければならないのは， 6 行目でアイドルフラグを設定している部分で

ある．このフラグの設定は次の行で呼ばれている pmPutReceiveBuf() よりも前でなければならない．さもなければ，pmPutReceiveBuf() の直後からアイドルフラグの設定までの間に，SCore-D がそのフラグを見る可能性がある．pmPutReceiveBuf() 関数が呼ばれることで PM は該当する受信メッセージの処理を完了したものとみなす．その結果として，ユーザ並列プロセスは本当はアイドルでないにも関わらず，SCore-D がアイドル状態とみなす可能性があるからである．12 行目の部分についても同様である．

最後に残された問題は，いつユーザ並列プロセスの状態を検出するか，ということである．ネットワークプリエンプションによるユーザ並列プロセスの大域状態は，当然のことであるが，ネットワークプリエンプションを行なわないと検出できない．大域状態検出のためのネットワークプリエンプションの時間間隔が長いと，アイドルである並列プロセスをスケジューリングしている可能性が増える．一方，頻繁にネットワークプリエンプションを行なうと，そのためのオーバヘッドが増える．

しかしながら，ギャングスケジューリングのためのネットワークプリエンプションと，アイドル検出のためのネットワークプリエンプションはほとんど違いはない．図 6.2 において楕円形のシェードで示した save フェーズにアイドル検出の機構を組み込み，並列プロセス切替の都度，ユーザ並列プロセスの大域状態を検出することした．

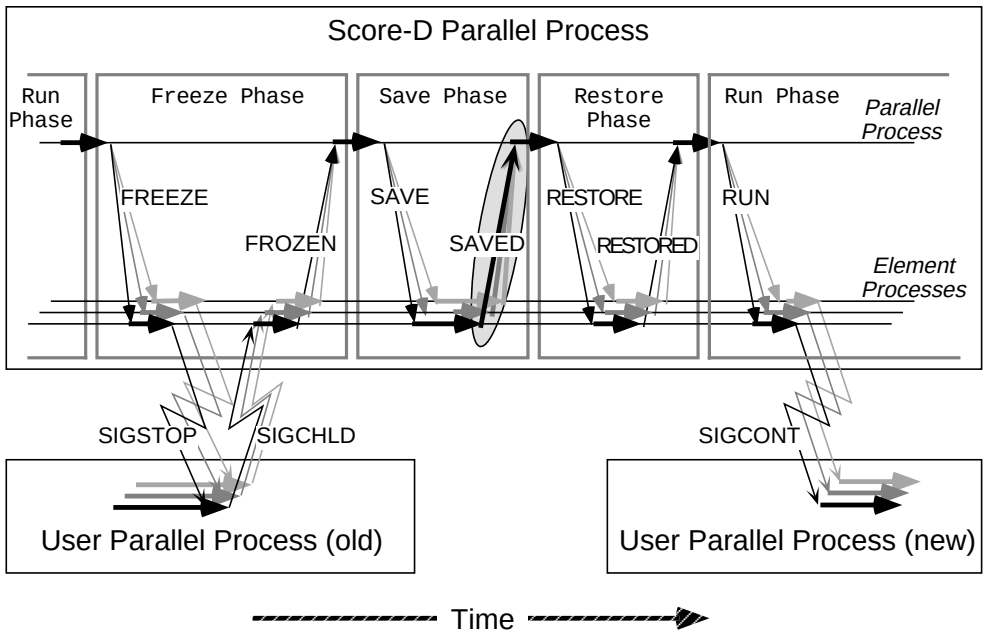


図 6.2: 並列プロセス切替における大域状態検出

理論的には，freeze フェーズ，save フェーズ，restore フェーズの終了を示すどのバリア同期にも，大域状態検出機構を組み込むことが可能である．Save フェーズの終了に組み込んだ理由は，このタイミングが並列プロセスの状態を変化させる時であるからである．大域状態検出のために付加した機構とは，Element Process オブジェクト（第3.4.1節）がサンプリングしたアイドルフラグの結果と，チャネルコンテキストに含まれるメッセージ数を save フェーズ終了時に返し，それらの値を適切なリダクション操作の結果をもって Parallel Process オブジェクトに通知するようにすることである．スケジューラはユーザ並列プロセスの大域状態に応じて次回のスケジューリングを決定する．

これらの変更がギャングスケジューリングのオーバヘッドに与える影響は無視できる程小さい．第 4 章で用いた SCore-D の評価において，既にこれらの機能が組み込まれており，大域状態検出のためのオーバヘッドは含まれている．

6.3 評価

ここでは，SCore-D における並列プロセスの大域状態検出を評価する．図 6.3 は，評価のためのモデルを示す．この図は，ユーザと並列プロセスが互いに応答を待っている様子である．それぞれの応答時間は指数分布とする．実際の SCore-D においては，ユーザはひとつのデバイスとし，決められた指数分布に従って並列プロセスからの呼掛け（デバイスに対する read 要求）に応答する．一方，ユーザ並列プロセスは，やはり決められた指数分布に従ってデバイスに要求を出す．指数分布は 10 $msec$ で離散化した．

図 6.4 に，このようなモデルに従った場合の評価結果を示す．並列プロセスが出す要求の平均頻度は毎秒 4 回，ユーザの応答時間は平均で毎秒 1 回に設定した．この時，要求を出す間隔は平均 1.0 sec であり，並列プロセスは平均で 0.25 sec 間走行するため，全体のプロセッサ利用率は 20 % となる．実際には有限個の試行しか行なわないため，また，並列プロセスの走行時間は正確ではない（ほぼその時間になると思われるだけのループを回る）ため，正確に 20 % の利用率になるとは限らない．プログラムは 300 回ユーザからの応答を受けとると終了する．その時のプログラム実行の経過時間とプログラムがユーザへ応答を返すまでの総時間の比を Y 軸にとる．全く同じプログラムを最大 4 つまで投入し，プログラム投入の数を X 軸とした．個々のプログラムについて仕事量（時間）を経過時間で割った値を利用率（複数投入した場合は，それらの平均）とし，Y 軸とした．

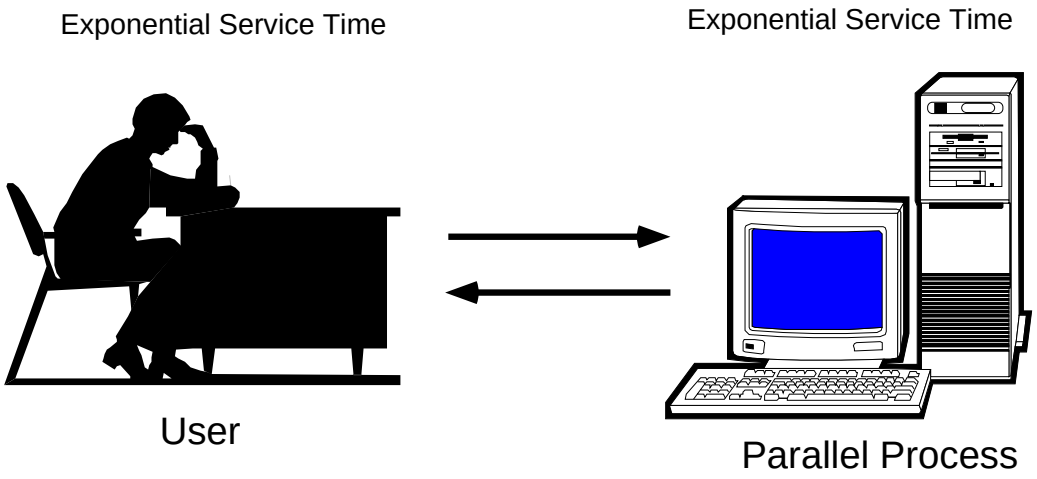


図 6.3: 大域状態検出の評価モデル

並列プロセスのプロセッサ数は 64 である．投入する並列プロセスの集合内においては互いに別な乱数系列となるようにした．比較のため，大域的にアイドル状態にはならない並列プロセスに関しても同じ条件で評価を行なった．図 6.4 には，理想的な状況での値（図中 “Ideal”）と，時分割間隔を無限大にした（実質的にバッチスケジューリングと同じ）場合に得られるであろう値（図中 ”Batch”）も併せて示されている． SCore-D の時分割間隔も 100 $msec$， 500 $msec$， 2500 $msec$ と変化させた．図 6.4 の凡例において，例えば “Idle, 0.1 sec” とあるのは，大域的にアイドルになる並列プロセスを 0.1 sec（100 $msec$）の時分割間隔でスケジューリングしたことを意味する．

利用率が 20 % なので， 5 つの並列プロセスを投入するまで個々の並列プロセスの利用率は変化しないのが理想である．バッチスケジューリングでは，並列プロセスの数に逆比例して利用率が低下する．図 6.4 から，大域的にアイドル状態になる並列プロセスでは，時分割間隔が短い程，複数並列プロセスを投入した時の利用率の低下が少ない．ネットワークプリエンプションを用いた大域状態検出では，並列プロセス切替時にしか大域状態を知ることはできない．このため，並列プロセスの状態が変化しても，すぐにはスケジューリングに反映されないため，理想的な値よりも悪い利用率となっている．そして時分割間隔が長い程，利用率は低下する．

大域的にアイドルにならない場合において，利用率がバッチスケジューリングと同じ程度に低下しないのは，ユーザからの待ち時間とスケジューリングされない時間が重なる場合があるからと考えられる．

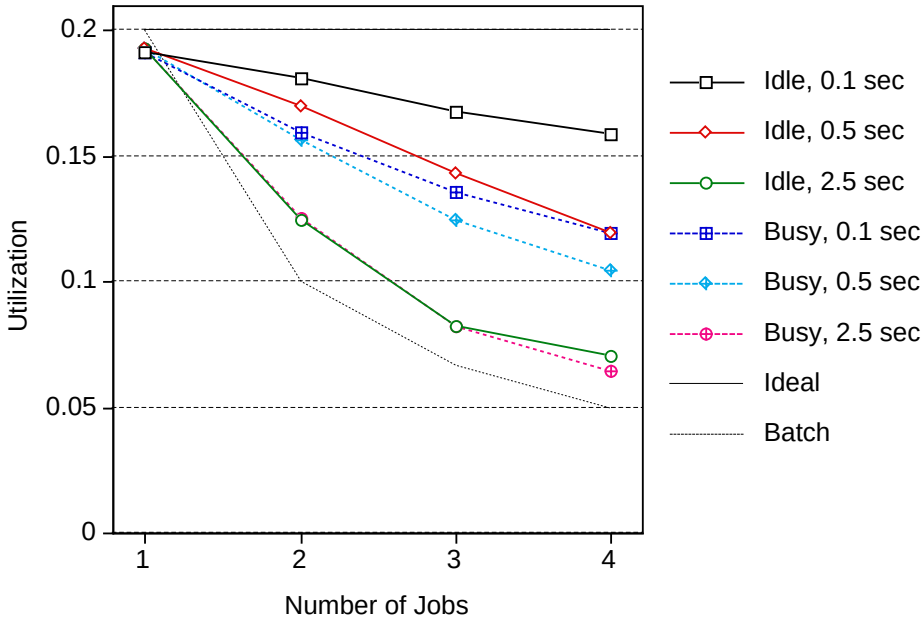


図 6.4: 時分割間隔とプロセッサ利用率の関係（64 プロセッサ）

図 6.4 の結果から，同じ時分割間隔においては大域状態検出をおこなった方がより高いプロセッサ利用率（仕事率の総和）が得られ，大域状態検出を行なった場合については時分割間隔が短い程より高いプロセッサ利用率を示すことが判明した．より高いプロセッサ利用率が得られるということは，ユーザにとってみれば（複数の対話並列プロセスが投入された）高負荷時においても応答時間の低下が少ないことを意味し，望ましい結果ということができる．

6.4 考察

6.4.1 対話並列処理について

本章では，効率的な対話並列処理をする方法について提案した．対話並列処理のニーズは存在するのであろうか．重要な対話並列アプリケーションとしては並列オンラインデバッガがある．クラスタ環境では，コアダンプによるオフラインデバッグでさえも難しい局面が存在する．そのような中でオンライン，それもデバッグ中にクラスタを占有しないデバッグの機構は，並列プログラム開発を目的とした場合，ユーザにとっては切実な要望である．

ここで提案している並列プロセスの大域状態検出と，それに基づいた対話並列処理は，Unix で言うところの対話処理とはかなり趣が異なっている．例えば Unix において，逐次プロセスの状態変化はただちに OS が検知しスケジューリングに反映させる．さらに効率的な対話処理を実現するためにプロセスの優先度を制御している [Bac86, LMKQ89]．一方， SCore-D における対話処理のための枠組では，並列プロセスの状態変化は一定時間間隔でしか検知できないし，今のところ SCore-D の並列プロセスには優先度という概念がない．

Unix の対話処理アプリケーションとして典型的なものは Emacs に代表される対話テキストエディタであろう．はたして並列 Emacs テキストエディタなるものは必要であろうか．多くの対話テキストエディタでは，端末から入力された文字のエコーバックをエディタ自身が行なう．エディタの処理としてプロセッサパワーを必要とする場合もないとは言えない．しかしながら，一文字をエコーバックするのに数十〜数百といったプロセッサ数が必要とは考え難い．もし高い計算能力を必要とするなら，バックエンドとして並列処理を利用するのが適切と考えられる．

並列処理における対話処理の粒度は，逐次処理のそれに比べ粗いと仮定することは自然と思われる．その仮定の下に SCore-D が提供する対話並列処理の枠組は，適切であると考えることができる． SCore-D には SCDB と呼ばれる GDB をベースとした並列デバッガが提供されている [堀 96b]．試験的な実装において， SCore-D の対話並列処理の枠組が並列オンラインデバッガの実装において有効であることが確認されている．

6.4.2 （分散）共有メモリへの応用可能性について

本研究で提案されたユーザ並列プロセスの大域状態の検出方法は，メッセージ通信による並列計算についてのみ考えてきた．はたしてこの方法は（分散）共有メモリ型の並列計算に適用可能なのであろうか．

図 6.1 にメッセージ通信型並列計算における大域アイドル検出のための処理を示した．また，ネットワークからメッセージを取り出すタイミングと，アイドルフラグの設定のタイミングに気をつける必要があることは既に述べた通りである．つまり，ネットワーク，より正確にはネットワークコンテキスト，からメッセージを取り出す前に，アイドルフラグの設定が必要条件である．

共有メモリにおけるメッセージの送信とは，基本的に相手ノードのメモリへの書き込

みである．メッセージの受信という特別な操作はなく，送信ノードによって書き込まれるであろうメモリ領域の内容を見て受信したか否かを判断する．

ここで問題になるのは，ネットワークコンテキストからメッセージを取り出すタイミングと，受信したことによるプロセスの待ち状態から実行状態への遷移のタイミングにおいて，メッセージが到着していてかつメッセージを取り出してしまう直前というタイミングが明確にできないことである．もうひとつの問題は，メッセージの待ちが普通のメモリアクセスであるため，プログラムのあちらこちらにアイドルフラグ設定の処理を埋め込む必要が生じる可能性があることである．

6.5 実時間負荷モニタリング

アイドルフラグはユーザ並列プロセスの大域状態の検出に役立つことを示したが，ここでは，その副作用として，アイドルフラグを利用したユーザ並列プロセスの実時間負荷モニタリング機能を提案する．

ユーザ並列プロセスによるアイドルフラグの設定は，ユーザ並列プロセスの大域状態検出だけでなく，別な応用が考えられる． SCore-D はこのフラグをサンプリングすることで，ユーザ並列プロセスの個々のプロセスがアイドルであるか否かを知ることができる．この情報はまさしく個々のプロセスの状態を示しており，この情報をユーザに表示することでユーザは自分の並列プロセスの挙動を知ることが可能となる．

このようにアイドルフラグを用いたユーザ並列プロセスの負荷状況モニタリングは，PM のようなユーザレベル通信，つまりポーリングやビジーウェイトで通信メッセージを待つ方式，では他に負荷状況を外部から知る術がないため有効である．また， SCore-D が並列プロセス切替のタイミングでアイドルフラグをサンプリングすれば，負荷状況のサンプリングがユーザ並列プロセスの実行にほとんど影響を及ぼさないと期待される．

いくつかの商用並列計算機においては，ハードウェアの稼働状況をモニタリングするハードウェア支援機構が備わっているものがある．例えば Intel 社の Paragon では，ハードウェアによるモニタリングの結果を SPV と呼ばれるソフトウェアにより実時間でユーザに表示することができる [Int96]． Paragon の OS は多重プログラミング環境を提供し，複数ユーザが同時に利用することが可能である [Int93]．しかしながら SPV はシステム全体の状況しか表示することができない．このため，複数ユーザが同時に Paragon を利用している時に SPV を用いてユーザ並列プロセスの状況を把握することは難しい．

多くの並列計算機において，ユーザ並列プロセス単独の挙動を知るには，トレースデータをロギングし，実行終了後にログされたデータを表示する，というオフラインによる処理になる．

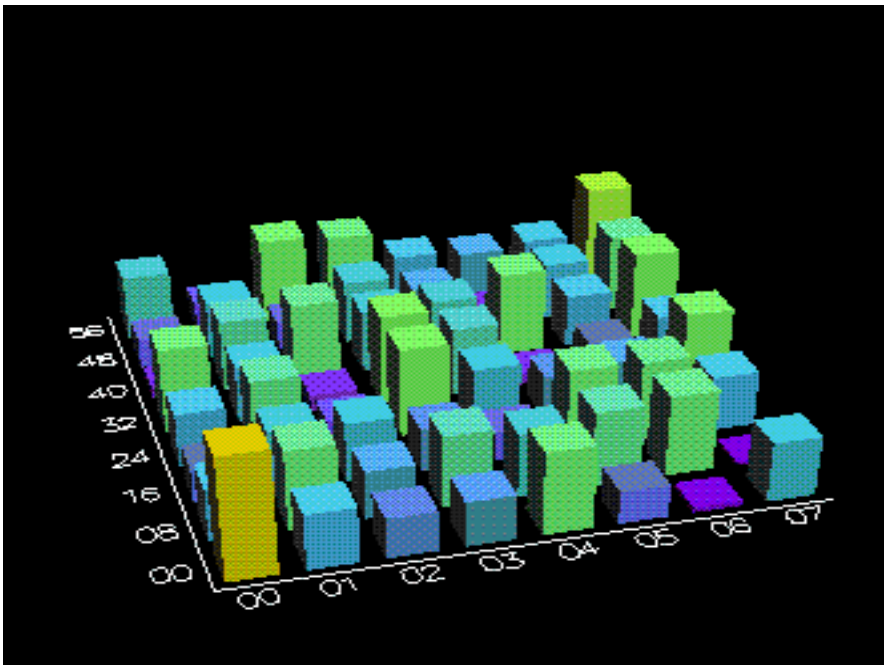


図 6.5: ユーザ並列プロセスの負荷状況モニタリングの例

アイドルフラグを用いた SCore-D のユーザ並列プロセスの負荷モニタリング機構は，ユーザ並列プロセスの稼働状況を実時間で表示することができ，しばしば膨大になるログファイルの必要がないという利点がある．図 6.5 は SCore-D によるユーザ並列プロセスの稼働状況をモニタリングするウィンドウの表示例である．この図において直方体のそれぞれがユーザ並列プロセスの個々のプロセスに対応しており，高さと色により負荷の状況を表している．直方体の高さが高いほど，また（図 6.5 からは判別できないが）色が赤い程，そのプロセスの稼働率が高いことを表している．

6.6 まとめ

本章では，ネットワークプリエンプションを応用して，ギャングスケジューラがユーザ並列プロセスの大域状態を知る方法について提案した．提案された大域状態検出機構は SCore-D に実際に組み込まれ，評価された．その結果，大域状態検出によりアイドル状態にある並列プロセスに対するスケジューリングを避けることができ，システム全体

としてもプロセッサ利用率を向上させると同時に，応答時間をも改善することが示された．

SCore-D が管理するクラスタにおいて走行する並列プロセスの一部が対話並列処理のアプリケーションであったとし，そのアプリケーションが大域的なアイドルになる時間がシステム全体の走行時間の数 % であったとしても，大域的なアイドル時間を他の並列プロセスに割り当てることによる利得は，ネットワークコンテキストの退避／復帰によるオーバヘッドによる損失と同等以上になる可能性がある．

秋山らは WWW を通じて PC クラスタ上でのタンパク質情報解析処理サービスを提供している [秋山 98]．ここでは SCore-D が使われており，サービスの 24 時間提供と，並列プログラムの開発がひとつの PC クラスタ上で提供されている．このような利用形態が可能になったのは， SCore-D が時分割スケジューリングと，大域状態検出によるアイドル検出を提供しているからである．

第 7 章

時空間分割スケジューリング

前章までで，並列計算機における時分割スケジューリングを実現する手法について提案および評価し，さらに他の時分割スケジューリング手法との比較を行なって来た．つまりは，逐次計算機で用いられている時分割スケジューリングを並列計算機上にどのように効率良く実現するかが焦点であった．本章では，並列計算機特有の，時分割スケジューリングを用いたジョブスケジューリング方式を提案する．

本章では，並列計算機上に効率的な時分割スケジューリング方式を実現するため，まず時分割空間分割スケジューリングアルゴリズムである Distributed Queue Tree[堀 94b, 堀 94c, HIK+95, 堀 95, HIN+95, 堀 96d] を提案する．時空間分割スケジューリングアルゴリズムの設計において，以下の問題をどのように解決するかがポイントとなる．

- 時空間分割スケジューラは，システムの状況に応じて適切なパーティションを選定するものとする．また，プロセッサ資源の有効活用という視点から，空間分割により生じる断片化 (fragmentation) を最小限に抑える必要がある．

- ひとつのプロセッサに集中された大域的な待ち行列の実装は，ボトルネックとなる可能性がある．このため，スケジューリングの処理，およびスケジューリングに必要なデータ構造は適切に分散させ，スケーラビリティを損なわないようにする必要がある．

- スケジューリングのオーバヘッドは必要最低限に抑え，同一条件のタスクは公平にスケジューリングされることが望まれる．これら 2 つの要件は，逐次マシンにおけるスケジューリングと全く同じである．

7.1 Distributed Queue Tree

図 7.1に DQT の構造の例を，表 7.2 に図 7.1 の DQT における時空間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 番目の並列プロセスが走行していることを示す．またタイムスロットの切れ目における並列プロセス切替は，ギャングスケジューリングを想定している．

DQT は動的パーティションの入れ子構造を反映したツリー構造を成す．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並列プロセスの走行待ち行列を持つ（図 7.1 で，DQT ノードを示す丸の右側にあるのが待ち行列を表す）．負荷を分散させる意味から，DQT ノードは対応するパーティション内の適当なプロセッサに分散配置される．しかしながら，実際に何をどのように分散させるかは具体的なシステムのパラメータにより決定されるべきである．ここでは，スケーラビリティを損なわないことを優先させ，出来るだけ処理および情報を分散させる方向で考える．

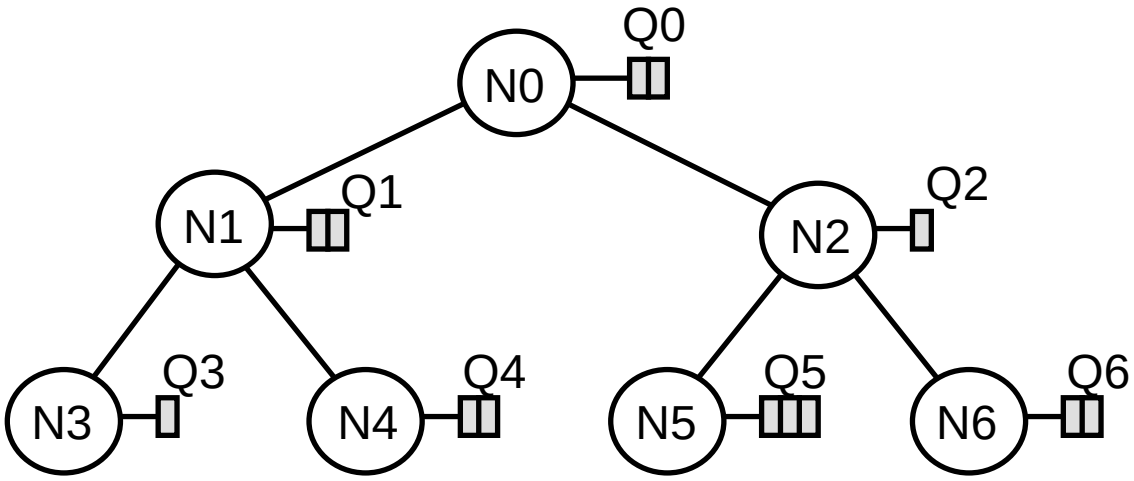
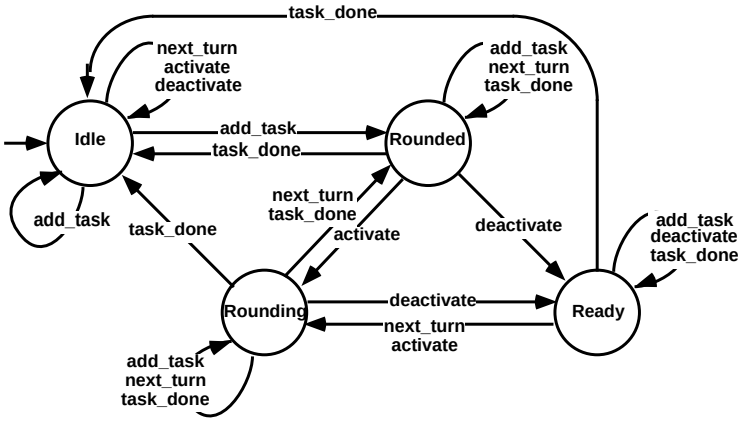


図 7.1: DQT の例

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	$Q_0(0)$			
:	:			

図 7.2: DQT スケジューリングの例

DQT のツリー構造は 2 分木に限らない．一般に n 分木構造を持つことができる．以



状態の説明

Idle：自ノードの待ち行列が空である状態。
Rounded：タスクが投入されてまだスケジューリングされていない状態、または自ノードが管理する待ち行列のラウンドロビンスケジューリングが１周した状態を示す。この状態における next_turnメッセージはそのまま下位ノードに転送される。
Rounding：ラウンドロビンの最中である状態。
Ready：スケジューリング待ちの状態。

図 7.3: DQT ノードの状態遷移図

下 DQT の構造を表すのに w^h と表記する．ここで w はノードから派生する子ノードの数，h は DQT のツリーの高さである．多くの場合，w はネットワークトポロジーとパーティショニングの次数から決定される．h はシステムが提供するパーティションサイズの範囲を指定するパラメータとなる．一般に w が大きくなると内部断片化（internal fragmentation）[PN77] が増大するため，注意が必要である．$w = 2$ の場合，パーティションの割当はバイナリバディ [Knu68] と同等となる．DQT では，ネットワークのトポロジー的に不連続なパーティション（「飛び地」の集合）は考慮していない．これは，i) 並列プロセス内での平均通信距離（ホップ数）が増大する，ii) 並列プロセス間で通信性能に干渉が生じる可能性がある [LLWN94, QN95]，iii) 並列プロセス切替のためのハードウェア支援機能 [堀 93b, 堀 93a, HYI+95] の実現が困難になる，といった理由による．

7.1.1 DQT スケジューリング

DQT の各ノードは，図 7.3 に示した状態遷移と，これから説明するノード間のメッセージにより，DQT 全体のラウンドロビンスケジューリングを分散協調的に実現する．

この図において，並列プロセスの終了を示す `task_done` メッセージが，並列プロセスを走らせていないはずの状態である *Rounded* や *Ready* 状態でも発生していることに注意を要する．これは，ノードの状態遷移は並列プロセスの状態遷移と非同期であるからである．

以下のメッセージの説明において，↓は根ノードから葉ノード（下位）に向かって流

れるメッセージを示し，↑ は葉ノードから根ノード（上位）に向かって流れるメッセージを示す．矢印のないものはそのノード内で発生したメッセージである．また，子ノードを持たない葉ノードは，永久に並列プロセスが投入されない空の子ノードを持つと考える．

add_task ↓ 新規並列プロセスの投入．並列プロセスサイズ（並列プロセスが動作するのに必要なプロセッサ数）が自ノードのパーティションの大きさと等しい場合は，自ノードの待ち行列に接続し，そうでない場合は，タスクのパーティションへの割当ポリシー（後述）に従って下位ノードに転送する．

activate ↓ ノードを活性化する．その結果，待ち行列が空でなければ Rounding 状態になる．そうでなければ Rounded 状態になり activate メッセージを全ての下位ノードに転送する．このメッセージは根ノードが直下の全てのノードより rounded メッセージを受けとった場合（ラウンドロビンでいうところの新しい周回に入ることを意味する），および，下位の子ノードの rounded メッセージの同期待ちの時に，より早く rounded を報告した子ノードを再度周回させる場合に発生する．

deactivate ↓ ノードを不活性化する．その結果，待ち行列が空でなければ Ready 状態になり，そうでない場合は Idle 状態になる．このメッセージは全ての下位ノードに転送される． activate メッセージを受けて活性化したノードが，全ての下位ノードを不活性化する際に発生する．

next_turn ↓ ラウンドロビンにおける並列プロセス切替を指示する．このメッセージを受け，自ノードの待ち行列の次の並列プロセスをスケジューリングする．もし，待ち行列が空であったり，そのノードにおける最後の並列プロセスを実行中であった場合は，自ノードの状態を *Rounded* とし， next_turn メッセージを下位ノードに転送する．このメッセージ送出の結果，以下のメッセージが返される．

rounding ↑ 下位の部分 DQT がラウンドロビンでいうところの周回中であることを示す．

rounded ↑ 下位の部分 DQT がラウンドロビンでいうところの周回が完了したことを示す．

全ての子ノードから１回以上 rounded メッセージを受け取った場合，全ての子ノードに deactivate メッセージを送出し，親ノードに対し rounded メッセージを返す．そうでない場合は，先に rounded メッセージを返したノードに対して再度 activate メッセージを送出し，親ノードに対し rounding メッセージを返す．

task_done 並列プロセスの終了あるいは並列プロセスが中断された場合に発生する．この結果，そのノードの全ての並列プロセスがスケジューリングされた場合，自ノードの状態を *Rounded* にし，全ての子ノードに対し next_turn メッセージを送出する．

DQT のツリー構造において，アクティブなノード（状態遷移で *Rounding* 状態にあるノードのこと）を結んだ曲線を「前線（front）」と呼ぶことにする．図 7.4（a）に上記のスケジューリング方式に基づいた前線移動の例を示す．図中，ノードを表す円の中の長方形は待ち行列のエントリを示し，前線はツリーを横断し左右に伸びる曲線で示した．前線より上のノードの状態は全て *Rounded* であり，前線より下のノードの状態は全て *Ready* である．前線は常に根から葉に向かう．

DQT の同じレベルの枝が全て同じ負荷であるような場合，前線は水平線となる（図中の t0， t1 の前線）．しかし，図 7.4（a）に示すように，同じレベルの負荷が不揃いの場合は，負荷が軽い方の枝上の前線が先に進む．したがって，負荷の軽い枝の前線が先に葉に到着することになる．この場合には先に到着した前線の部分は，負荷の不均衡が生じたノードまでさかのぼり，そこから再度，前線が葉に向かって進む．こうして，全ての枝が少なくともラウンドロビンにおける「一周」するまで，前線は部分的に上下に波を打つ挙動を見せる（図中の t3 〜 t6 の前線）．

結果的に，負荷の低い DQT の部分ツリーは負荷の高い部分ツリーよりも多くスケジューリングされる．この方針はプロセッサ利用効率を高める一方，スケジューリングの不公平さを招く要因になる．この点に関するスケジューリング性能の及ぼす影響について明らかにするために，別な “Fair-DQT” と呼ばれるスケジューリング方式を提案する． Fair-DQT のスケジューリングでは，子ノードから rounded を受けとった ルートノード以外のノードは，いかなる場合もその子ノードに対し activate メッセージを投げない，という変更だけで実現できる．

図 7.4（b）は Fair-DQT における前線移動の様子を示したものである． Fair-DQT において，スケジューリングが一周する間にその Fair-DQT に含まれる全てのプロセス

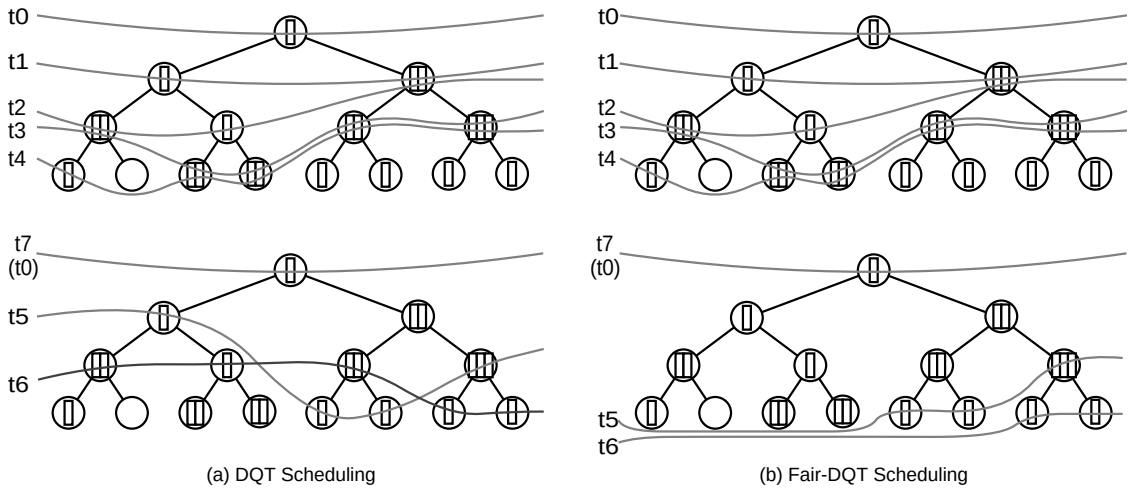


図 7.4: 前線移動の様子の例

は一回しかスケジューリングされない．こうすることでスケジューリングの公平さを保つことができる一方，プロセッサ利用率の低下が懸念される．この点に関しては第 7.2.5 節におてシミュレーションによる評価を試みる．

7.1.2 DQT の性質

Total Queue Length of a Branch (TQLB)

TQLB とは DQT のある葉ノードから根ノードに至る全てのノードの待ち行列長の総和である． TQLB は以下の式で再帰的に計算される値である．

$$B_i = \begin{cases} L_0 & \text{for } i = 0 \\ L_i + B_{\lfloor (i-1) \div w \rfloor} & \text{otherwise} \end{cases}$$

ここで， B_i は根ノードから i 番目の葉ノードに至る TQLB であり， L_i はそのノードにおける待ち行列の長さである．各 DQT ノードは根ノードをゼロとし，幅優先の順序で番号付けられているものとする．図 7.5 に例を示す．

DQT スケジューリングの性質から， DQT 内の全ての並列プロセスは TQLB の最大値分のタイムスロット数の時間内に，少なくとも 1 回はスケジューリングされることが保証されている． TQLB の最大値は DQT の負荷状況を示すひとつの指標と考えることができる．

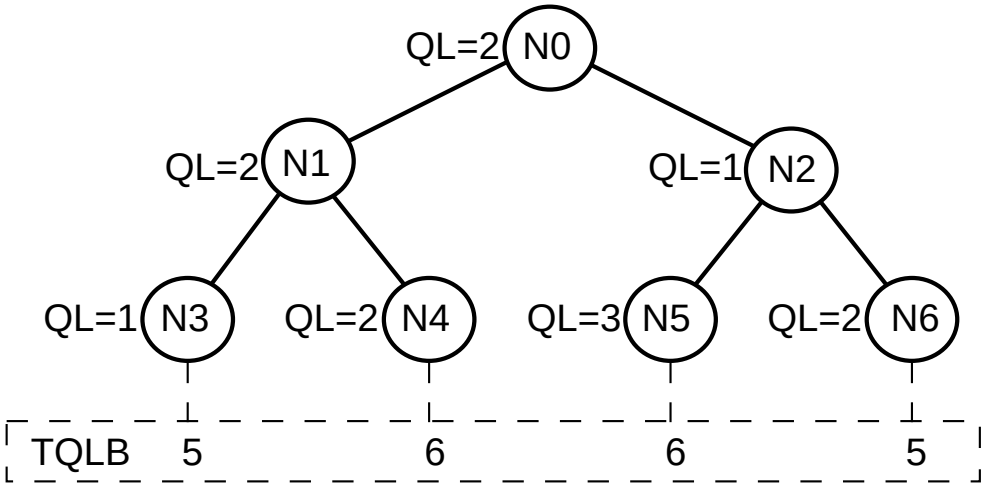


図 7.5: TQLB の例

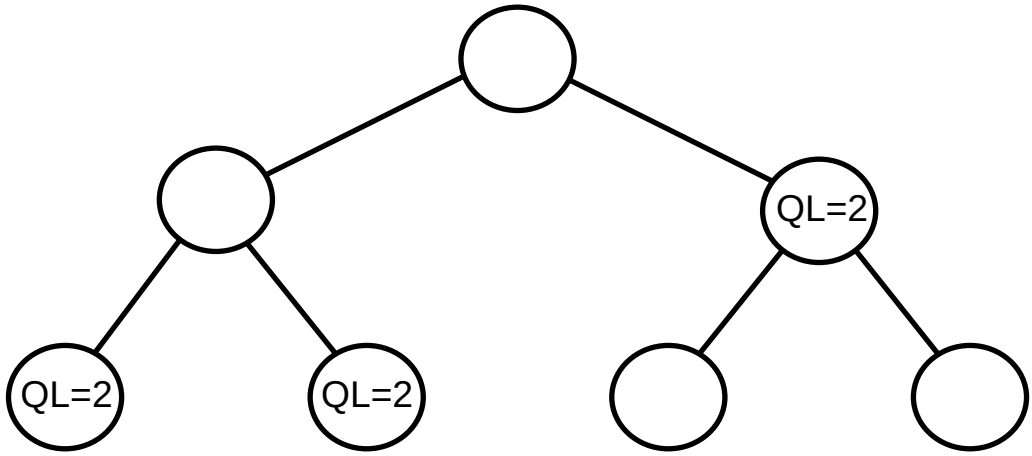


図 7.6: Balanced DQT の例

Balanced DQT

空でない DQT において，全ての TQLB が同じ長さであった場合，その DQT は “Balanced DQT” と呼ばれる（図 7.6）． Balanced DQT においては 100 % のプロセッサ利用率が達成されると同時に，全ての並列プロセスが同等のスケジューリング機会を与えられる（これは DQT のスケジューリング方式と TQLB の定義から明らかである）．

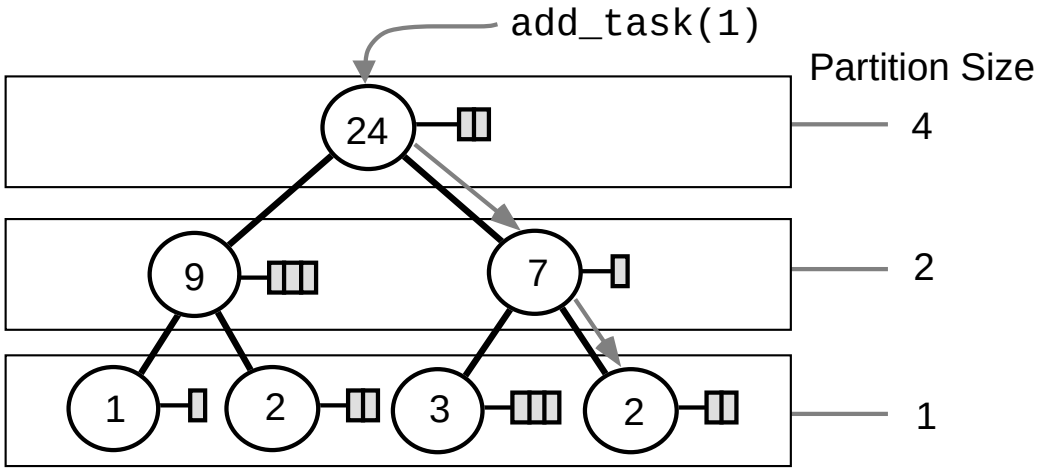


図 7.7: ポリシーの例（APA）

7.1.3 Task Allocation Policy

投入されたジョブにパーティションを割り当てる方式の例を図 7.7に示す．図中 add_task メッセージが DQT の根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．このポリシーでは，各 DQT ノードにおいて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，ここで投入された並列プロセス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，並列プロセスが必要とするプロセッサ数に十分な大きさのパーティションを担当する DQT ノードに到達するまで続けられる．この並列プロセスをパーティションに割り当てるポリシーを我々は “Task Allocation Policy （TAP）” と呼んでいる [HIK+95].

並列プロセスのマイグレーションは考えていないため，並列プロセスの割当は負荷を平衡させると同時にプロセッサ空間の断片を埋める唯一の機会である．負荷の偏りは，不公平なスケジューリングの原因になる．

Max-TQLB (MAX) ポリシー

DQT の各ノードは，そのノードを根とする部分 DQT の最大の TQLB （Max-TQLB）の値を持つ．ここで i 番目の子ノードの Max-TQLB （M_i）は再帰的に次のように定義

される．

$$M_i = \begin{cases} 0 & \text{for } i \geq w^h \\ L_i + \max\limits_{0 \leq k < w} M_{i \times w + k + 1} & \text{for } 0 \leq i < w^h \end{cases}$$

ノードの待ち行列長が変化した場合は `info` メッセージにより親ノードにその旨通知する．親ノードでは Max-TQLB を再計算し，さらに親ノードに `info` メッセージを送る．この処理は DQT の根ノードに到達するまで再帰的に繰り返される．パーティションの割当時には，最も小さい Max-TQLB を持つ子ノードに対し `add_task` メッセージを転送する．

このポリシーは負荷が軽い時には有効であるが，高負荷時にはあまり有効ではないと予想される． TQLB の最大値を求めることがより負荷の軽いノードを隠してしまうからである．

Min-TQLB (MIN) ポリシー

このポリシーは Max TQLB に良く似ている．違いは子ノードの TQLB の最小値を用いることである． i 番目の子ノードの Min-TQLB （M_i）は再帰的に次のように定義される．

$$N_i = \begin{cases} 0 & \text{for } i \geq w^h \\ L_i + \min\limits_{0 \leq k < w} N_{i \times w + k + 1} & \text{for } 0 \leq i < w^h \end{cases}$$

MAX ポリシーとは対称的に MIN ポリシーは，負荷の高い時に有効であり，負荷の低い時に有効ではないと予想される．低い負荷の時の TQLB の値の大半はゼロになるからである．

APA ポリシー

i 番目の DQT ノードの Assigned Processor Amount (APA) とは以下の式で再帰的に定義される値（A_i）である．

$$A_i = \begin{cases} 0 & \text{for } i \geq w^h \\ L_i S_i + \sum\limits_{k=1}^{w} A_{i \times w + k} & \text{for } 0 \leq i < w^h \end{cases}$$

ここで， S_i は i 番目の DQT ノードが担当するパーティションの大きさ（プロセッサ数）である．あるノードの APA の値は，そのノードを根とする部分 DQT に含まれる

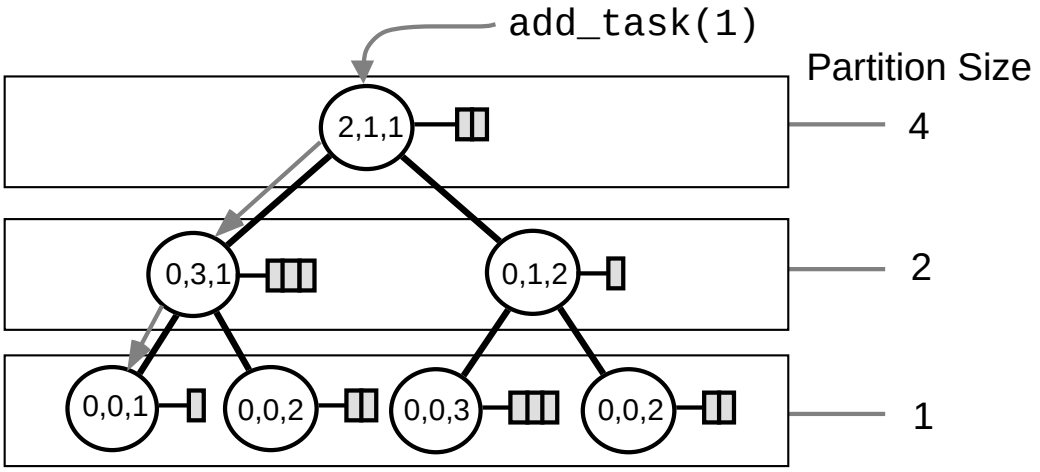


図 7.8: ポリシーの例（FF）

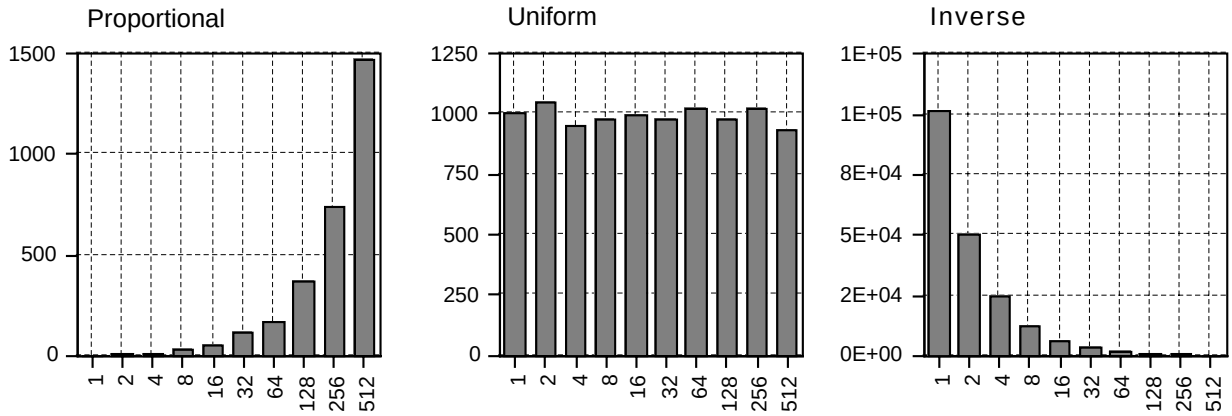
並列プロセスが必要とする仮想的なプロセッサの量を示す．このポリシーに基づく TAP では，子ノードのそれぞれの APA の値を保持しておき，add_task メッセージは最も APA の値が小さいノードに転送する．先ほどの図 7.7 は APA ポリシーによる並列プロセス割当の例である． DQT ノードを示す○の中の数字は APA の値を示す．○の右にはそれぞれのノードの待ち行列の長さを表している．この例では図中最も下位の右端のノードに割り当てられている．

FF ポリシー

各 DQT ノードは，そのノードを根とする部分 DQT のそれぞれのパーティションサイズにおける待ち行列の最小値を保持する．図 7.8 は FF ポリシーによる並列プロセス割当の例である． DQT ノードを示す○の中にある数列は，左から，パーティションサイズ 4， 2， 1 に相当する待ち行列の最小値を示している． add_task メッセージは対象となるパーティションサイズのうち，最も短い待ち行列のノードを含む方に転送される．この例では図中もっとも左端のノードに割り当てられている． FF ポリシーでは TQLB のバランスに対する配慮が全く欠如している．

FF-APA ポリシー

いくつかの TAP を組み合わせることも考えられる．例えば，上記の APA ポリシーと FF ポリシーの組合せである．この場合，最初 FF ポリシーにより子ノードの選択を



目標負荷率： 0.99，プロセッサ数： 1024

図 7.9: 並列プロセスサイズの頻度分布の例

試み，判断できなかった場合に APA ポリシーで判定する．これをここでは，FF-APA ポリシーと呼ぶことにする．現時点でこの FF-APA ポリシーが最良と考えられており [HIK+95]，本稿におけるシミュレーションでは全てこのポリシーを用いた．

7.2 シミュレーションによる評価

シミュレーションは全てバイナリ DQT を対象とした．特に明記されていない場合，シミュレーション時間は 10^6 単位時間，全ての並列プロセスはポアソン到着とし，並列プロセスジョブのサービス時間は平均 10^3 単位時間の指数分布とした．投入する並列プロセスのサイズは，一様分布，並列プロセスサイズに比例した分布，および，並列プロセスサイズに逆比例した分布の 3 種類おこなった．正規分布あるいはそれに類似した分布を用いたシミュレーションを用いなかった理由は，実際の並列マシンの運用を考えた時，中間的な大きさの並列プロセスが集中するような場合は考え難いためである．文献 [FR95] では，実測の結果，並列プロセスサイズの分布はほぼ一様であったとの報告がある．図 7.9 に，シミュレーションで投入した並列プロセスサイズ分布の例を示す．

全ての場合のおいて，サービス時間の分布は並列プロセスサイズの分布と独立である．ただし，最大パーティションを要求する並列プロセスは，単一待ち行列の時分割スケジューリングと同じことを意味するため，全ての場合から除いてある．並列プロセスは他の並列プロセスと独立にスケジューリングが可能であり，並列プロセスの実行の途中でサス

ペンドすることなく最後まで走り切るものとした．また時分割の量子時間は 1 単位時間とした．スケジューリングのオーバヘッドは無視されている．並列プロセスが要求するプロセッサの数は全て 2 のべき乗とした．従って，シミュレーション結果に内部断片化 [PN77] は反映されていないため，プロセッサ利用率の数値は楽観的な値となっている．

ここで平均並列プロセス到着時間間隔（$\widetilde{T_{interval}}$）は，システムの目標負荷率（W_{target}，$0 < W_{target} < 1$）に応じて以下の式で求めた値を用いた．

$$\widetilde{T_{interval}} = \frac{\widetilde{process\_size} \times \widetilde{process\_length}}{P \times W_{target}}$$

ここで，$\widetilde{process\_size}$ は並列プロセスが要求するプロセッサの数の平均（並列プロセスサイズの頻度分布から求めた値），$\widetilde{process\_length}$ は並列プロセスサービス時間の平均，P はシステムのプロセッサ数である．

並列プロセスサイズ，並列プロセス到着時間およびサービス時間は，それぞれ分布をもった乱数となっているため，必ずしも W_{target} で設定した通りの負荷になるとは限らない．そこで，シミュレーションの結果から得られる実績負荷率（W_{actual}）を以下のように定義する．

$$W_{actual} = \frac{\sum_{l=0}^{L-1}(process\_size_l \times process\_length_l)}{P \times T}$$

ここで，$process\_size_l$ は l 番目の並列プロセスが要求するプロセッサの数，$process\_length_l$ は同じく l 番目の並列プロセスのサービス時間，L は投入した並列プロセスの総数，T はシミュレーション時間である．

一方，時刻 t におけるプロセッサ利用率（U_t）は，P_t^* を走行している可能性のあるプロセッサの数（あるいは，アクティブなパーティションにおけるプロセッサ数の総和）とすると $U_t = P_t^*/P$ で定義され，時間区間 t_1 から t_2 における平均プロセッサ利用率（$\tilde{U}$）は以下の式で定義される．

$$\tilde{U} = \frac{\sum_{t=t_1}^{t_2} P_t^*}{P \times (t_2 - t_1 + 1)}$$

さらに，実実行時間比[1]（Real Execution Time Ratio:RETR）を定義する．RETR

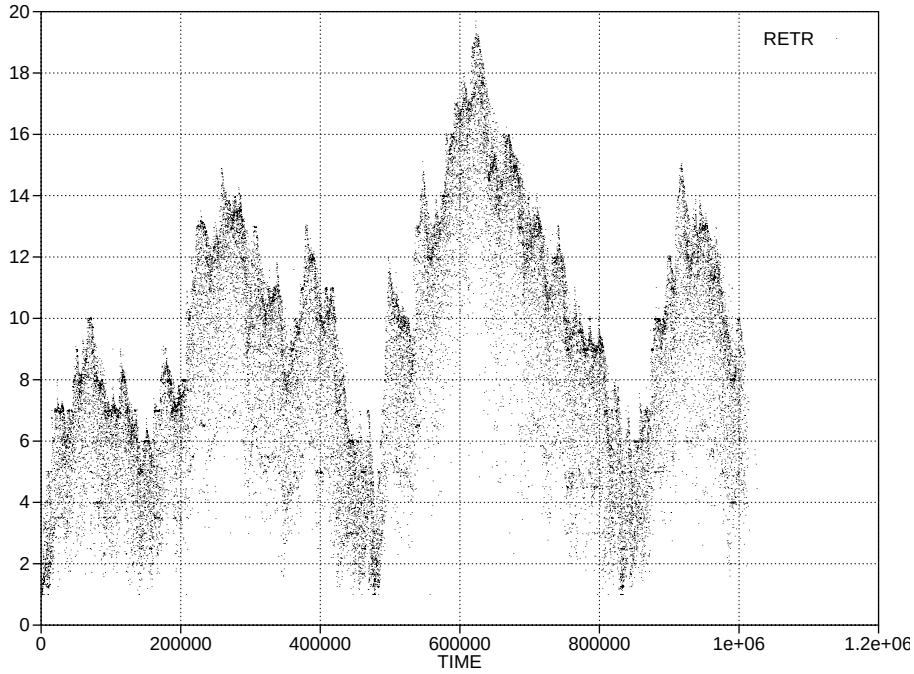
[1]「実実行時間」（Real Execution Time）とは，ある並列プロセスが走行キューに滞在する時間を指す．したがって，I/O 待ちなど並列プロセスがサスペンドしている時間は含まれない．この意味で「経過時間（Elapsed Time）」とは異なる．しかしながら，本章のシミュレーションにおいて並列プロセスはサスペンドしないため，結果的には同じ意味となっている．

は個々の並列プロセスのサービス時間に対する実実行時間の比を表すもので，並列プロセス l の実実行時間比を R_l^{RET} とすると，

$$R_l^{RET} = \frac{t_l^{process\_terminate} - t_l^{process\_invocation}}{process\_length_l}$$

と定義される．ここで，$t_l^{process\_invocation}$ は並列プロセス l の投入時刻，$t_l^{process\_terminate}$ は終了時刻である．

7.2.1 DQT の基本動作

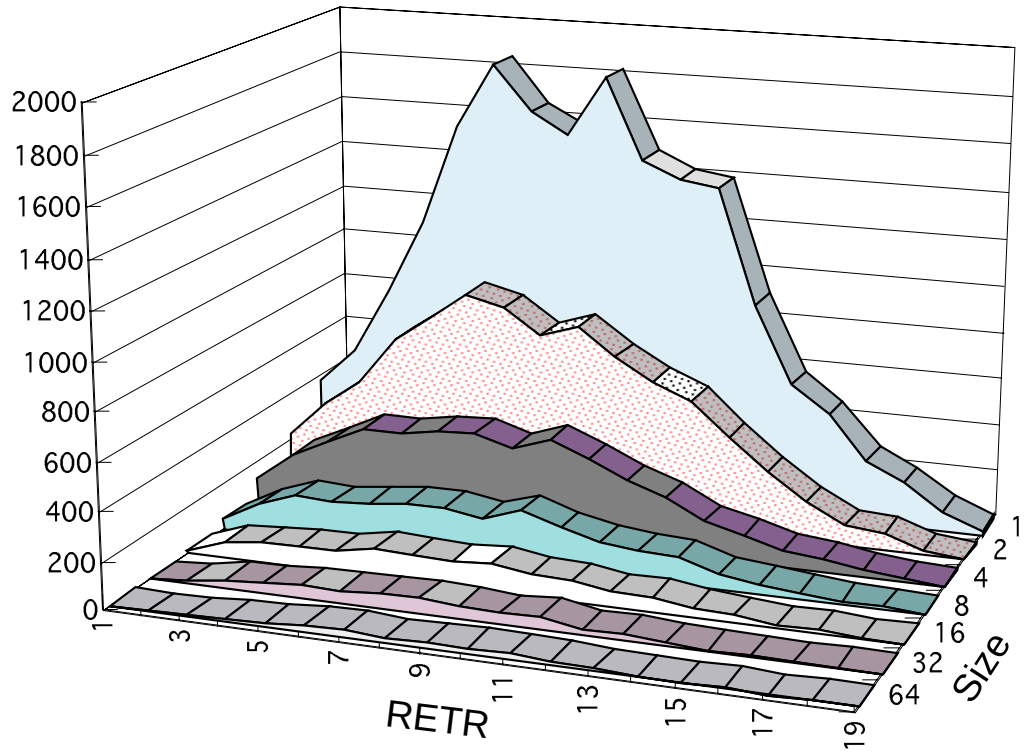


プロセッサ数： 128，並列プロセスサイズの分布：逆比例，目標負荷率： 99 %

図 7.10: 実実行時間比

図 7.10 はプロセッサ数が 128 の DQT に，逆比例の並列プロセスサイズの分布で負荷を与えた場合の，各並列プロセスの終了時刻におけるその RETR を点でプロットしたものである（X 軸：シミュレーション時間，Y 軸： RETR）．シミュレーション時間は 1.2×10^6 であり，10^6 単位時間の時点で並列プロセス投入を打ち切っている．RETR はその並列プロセス実行時間中の最大 TQLB の平均になる．図には判読が難しくなるため示さなかったが，実際の最大 TQLB の線はこのグラフにおける RETR の包絡線に近い．並列プロセスサイズと並列プロセス到着間隔を互いに関連のない乱数として生成していることから，シミュレーション期間中の負荷が大きく変動していることが分かる．

RETR の各点は，最大 TQLB の近くに多く分布しているのが分かる．しかしながら分布の最大幅（ある時間間隔における最大の RETR と最小の RETR）は，特に負荷が高い場合において，かなり広がって分布している．これは， DQT の負荷の低い部分により多くのスケジューリング機会を与えるというスケジューリング方式の結果と考えられる．



プロセッサ数： 128，並列プロセスサイズの分布：逆比例，目標負荷率： 99 %

図 7.11: 実実行時間比のヒストグラム

図 7.11 は図 7.10 の結果を並列プロセスサイズ毎に RETR の頻度分布として示したものである（Y 軸：頻度）．いずれの並列プロセスサイズにおいても RETR の分布はなだらかである．これは，前述したようにシステムの負荷がシミュレーションの期間中に大きく変動することと， DQT のスケジューリング方式の特性からくるものと考えられる．

7.2.2 TAP の比較

勾配負荷時の各 TAP の挙動

以下に TAP の比較を目的としたシミュレーション結果を示す． TAP の挙動は低負荷時と高負荷時で異なる場合が多いので，勾配負荷とした．プロセッサ数は 128，量子時間は 1 単位時間であり，投入した並列プロセスの大きさは全て 2 のべき乗である．並列プロセスの大きさの分布は大きさに逆比例である．並列プロセスの実行時間の分布は 500 から 19,999 単位時間の一様分布となっている．

シミュレーションの負荷率（F_W）は以下のように定義される

$$F_W = \frac{\sum_{q=0}^{Q-1} size_q \times length_q}{P \times T}$$

ここで， q 番目に投入された並列プロセスは， $length_q$ 単位時間， $size_q$ で示されたプロセッサ数を使用するジョブとする． Q は投入する並列プロセスの総数であり， T はシミュレーション総時間である．

ここで，勾配負荷（時間に比例して負荷率が上昇）パターンを生成するために，シミュレーション総時間の 5 % 経過した時点で負荷率を 5 % 上昇させた．シミュレーション時間は，特に断らない限り， 2,200,000 単位時間であり，並列プロセスの投入は 2,000,000 単位時間で打ち切っている．打ち切り時の負荷率はおよそシステムの計算力に対し 133 % である．並列プロセスの実行時間や大きさは乱数により発生しているため必ずしも目標とする負荷率と一致しない．シミュレーションにおいて，通信遅延，並列プロセス切替のオーバヘッドは考慮されていない．並列プロセスの投入パターンは同じシミュレーション条件では全く同一である．あるポリシーにおいてどの子ノードに割り振るかを選択できなかった場合は左端の子ノードを選択するようにした．

図 7.12，図 7.13 はそれぞれ MAX および MIN ポリシーのシミュレーション結果のグラフである．このグラフでは並列プロセスが終了した時点でその並列プロセスの RETR を点で示し， TQLB の最大と最小の値をそれぞれ点線で表したものである． MIN ポリシーは高負荷で TQLB の最大が 6 なのに対し MAX ポリシーは 19 である．また MIN ポリシーでは低負荷時から TQLB が比較的高い値を示す。予想された通り， MAX ポリシーは低負荷時に， MIN ポリシーは高負荷時に比較的良い性能を示すことが確認された．

図 7.16 は FF ポリシーのシミュレーション結果である． FF ポリシーでは最大 TQLB

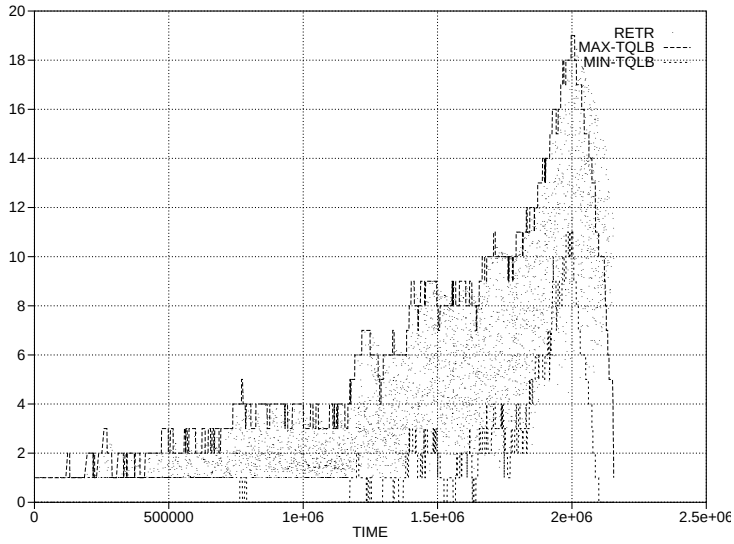


図 7.12: Max ポリシー（128 プロセッサ）

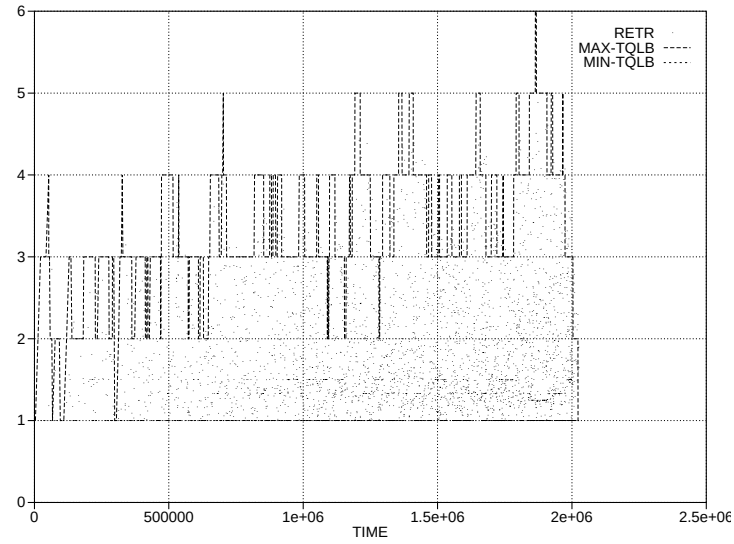


図 7.13: Min ポリシー（128 プロセッサ）

の値は DQT の高さ（この場合は 7）の倍数に近い値になる傾向が見られる．これは同じレベル（深さ）のノードの待ち行列の長さの差は多くの場合 1 であることから生じる現象と考えられる．この現象はより大きな規模の DQT においてより明確に見ることができる．図 7.15 は プロセッサ数を 2,048 とし，シミュレーション時間を 10^6 単位時間とした時の結果のグラフである．最大 TQLB の値はおよそ 11， 22， 33 そして 44 の値で階段状になっているのが判る．また，最大 TQLB の値と最小 TQLB の値の差も約 11 に

なっている．

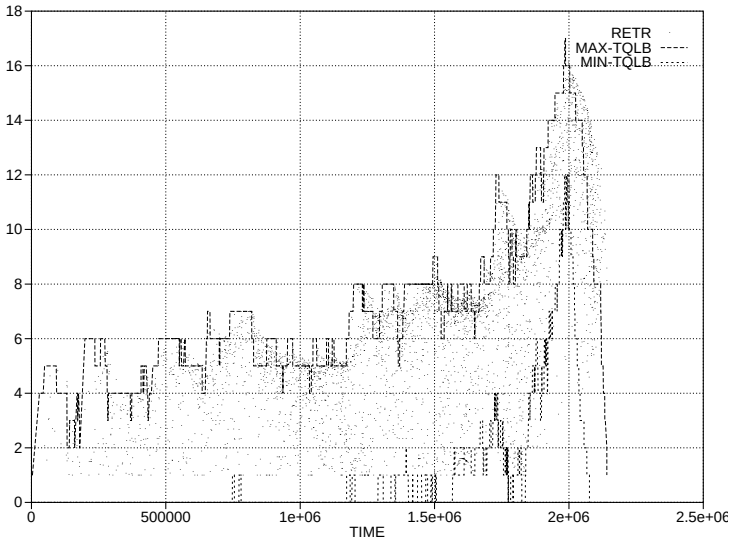


図 7.14: FF ポリシー（128 プロセッサ）

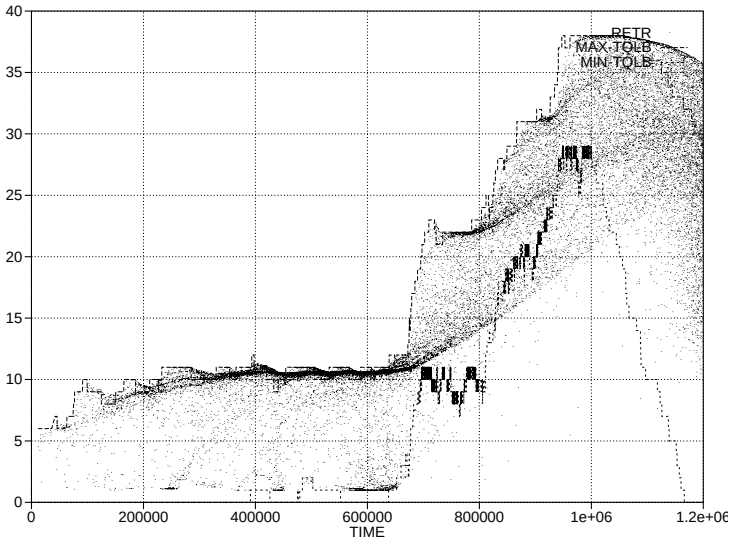


図 7.15: FF ポリシー（2048 プロセッサ）

図 7.14 は APA ポリシーのシミュレーション結果である． APA ポリシーは高負荷時でも低負荷時でも TQLB の最大を良く抑えている．図 7.17 は FF と APA ポリシーを組み合わせた場合のシミュレーション結果である．グラフから，低負荷時には APA ポリシーの振舞いを見せ，高負荷時には APA や FF 単独のポリシーよりも良い結果を示して

いる．

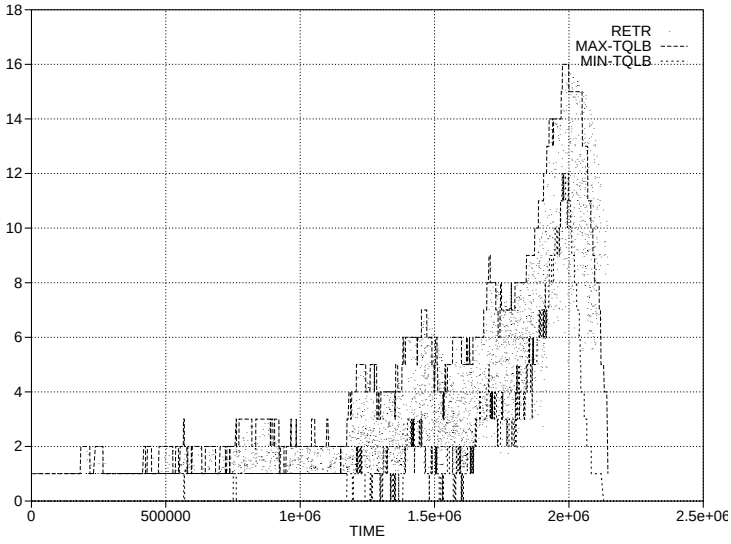


図 7.16: APA ポリシー（128 プロセッサ）

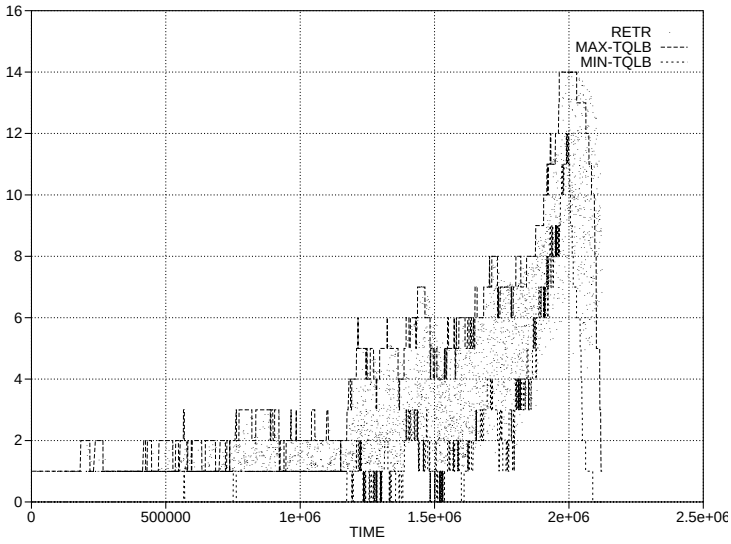


図 7.17: FF & APA ポリシー（128 プロセッサ）

一定負荷時の各 TAP の挙動

表 7.1は，一定の負荷率におけるプロセッサ利用率の平均（表中 “Util.”）とシミュレーションにおいて観測された最大 TQLB のうち最も大きい値（表中 “Max. TQLB”）を，

表 7.1: TAP の比較

Policy	WF = 0.368		WF = 0.793	
	Util.	Max. TQLB	Util.	Max. TQLB
MAX	0.366	4	0.768	10
MIN	0.366	5	0.775	8
APA	0.366	3	0.777	8
FF	0.363	7	0.768	8
FF & APA	0.366	3	0.776	7

Constant workload, 2^7 DQT

それぞれのポリシーで比較したものである．シミュレーション時間は 10^6 単位時間，高負荷時の負荷率（Workload Factor，表中の “WF”）は 0.793 であり，低負荷時の負荷率は 0.368 である．低負荷時において， MAX， MIN そして APA ポリシーにおけるプロセッサ利用率はほぼ同じである．この 0.366 というプロセッサ利用率の値は負荷率から考えてほぼ理想的な状況と考えられる． FF-APA ポリシーは低負荷時および高負荷時において最良の結果を示した．

7.2.3 並列プロセスサイズの分布の違いによる影響

図 7.18にシミュレーション結果を示す．左列の３つのグラフは，負荷率（X 軸）を変化させた時の，シミュレーション全体を通した平均プロセッサ利用率をプロットしたものである． TAP は FF-APA を用いた．上段のグラフは並列プロセスの大きさ（その並列プロセスの実行に必要なプロセッサ台数）に正比例した分布，中段が一様分布，下段のグラフは並列プロセスの大きさに逆比例した分布の場合の結果である．高負荷時の違いを強調するため，低負荷時の結果はクリップしてある．また，参考のため，理想的な値を示す直線（”Ideal”）もプロットした．右列のグラフは，同じ列の左側のグラフと同じ条件でシミュレーションをおこなった場合の， RETR の平均を Y 軸にプロットしたものである．これらのシミュレーションにおいて，システムの規模（システムが保持するプロセッサ数）を 128， 256， 512， 1024， 2048， 4096 としたときのそれぞれについて，目標負荷率を 0.2， 0.4， 0.6， 0.8， 0.9， 0.95， 0.97， 0.99 と変化させた．プロットした負荷率の値は実績負荷率を用いた．乱数の偏りにより実績負荷率が 1 を越えた

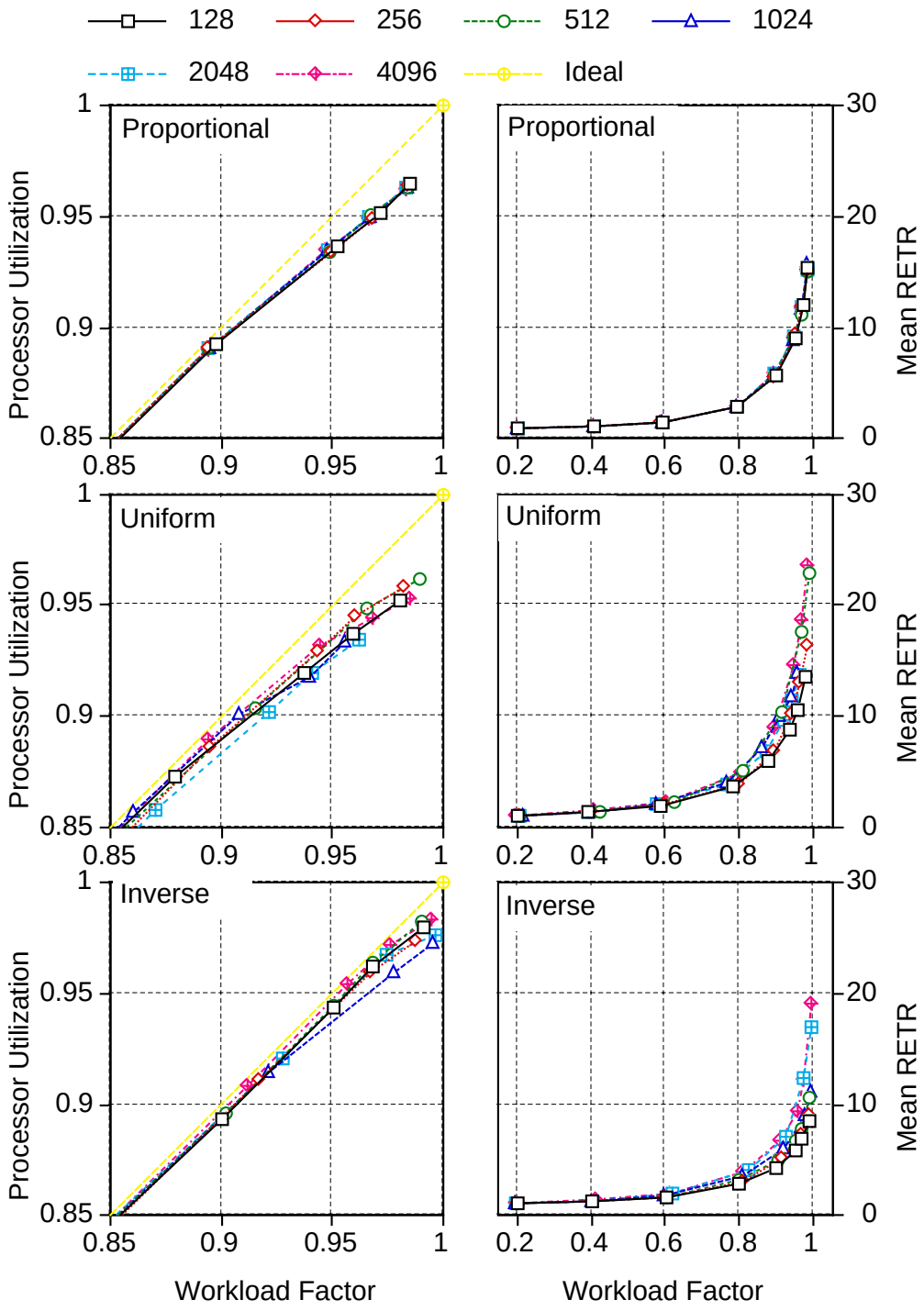


図 7.18: DQT のシミュレーション結果（FF-APA ポリシー）

ケースはグラフより除いてある．

図 7.18の左列のグラフにおいて，理想状態（プロセッサ利用率が設定された負荷率と同じ）に近いほど良いスケジューリングであると言うことができる．これらのグラフから， DQT はシステムの規模の影響を受けずに，低負荷から高負荷に至るまで，理想に

近い挙動を示した.

プロセッサ利用率のグラフ（図 7.18 の左列のグラフ）からは，３種類の並列プロセスサイズの分布の中では，大きさに逆比例する分布がベストであり，次いで正比例の分布，最悪が一様分布であると考えられる．DQT の性質から，あるレベルで生じた負荷のアンバランスはより小さいサイズのタスクでのみ解消することができる．逆比例分布の場合，小さな並列プロセスが十分に投入されるため良好なプロセッサ利用率を示すものと考えられる． 逆に正比例分布では， i) 大きな並列プロセスが多数を占めているためにプロセッサ利用率が下がらなかった， ii) 負荷の不均衡が DQT のスケジューリングにより解消された，と推測される.

7.2.4 Fair-DQT との比較

図 7.19 は，目標負荷率を変化させた時のプロセッサ利用率と平均 RETR について，DQT と Fair-DQT のシミュレーション結果を示したものである．プロセッサは 1,024，APA ポリシーとした.

興味深いことに，正比例および一様分布の場合における Fair-DQT のプロセッサ利用率の低下が DQT に比べ僅かであることが分かる．逆に逆比例分布の場合にプロセッサ利用率の低下が顕著になっている．これは， Fair-DQT が負荷の不平衡時におけるプロセッサ利用率の低下を避けることが不可能であり，多数の小さい並列プロセスにより生じた負荷の不平衡を解消できなかったため，と考えることができる.

一方，平均 RETR のグラフにおいては， Fair-DQT における平均 RETR の高負荷時の増大傾向が DQT よりも強い． DQT においてはプロセッサ利用率を向上させるために，軽い負荷の部分を余計にスケジューリングする．この結果，軽い負荷の部分に属した並列プロセスの RETR は，重い負荷の部分に属した並列プロセスよりも RETR が小さくなる． Fair-DQT ではこのようなことは生じないため，平均 RETR が DQT よりも悪い結果となったものと考えることができる.

7.2.5 バッチスケジューリング方式との比較

図 7.20 は， DQT， Fair-DQT， First-Come-First-Serve スケジューリングとバイナリバディによる空間分割の組み合わせ（これを FCFS-BB と呼ぶ）および， ScanUp [KLDR94] のそれぞれについて，図 7.20 と同じ条件でシミュレーションした時のプロセッサ利用率

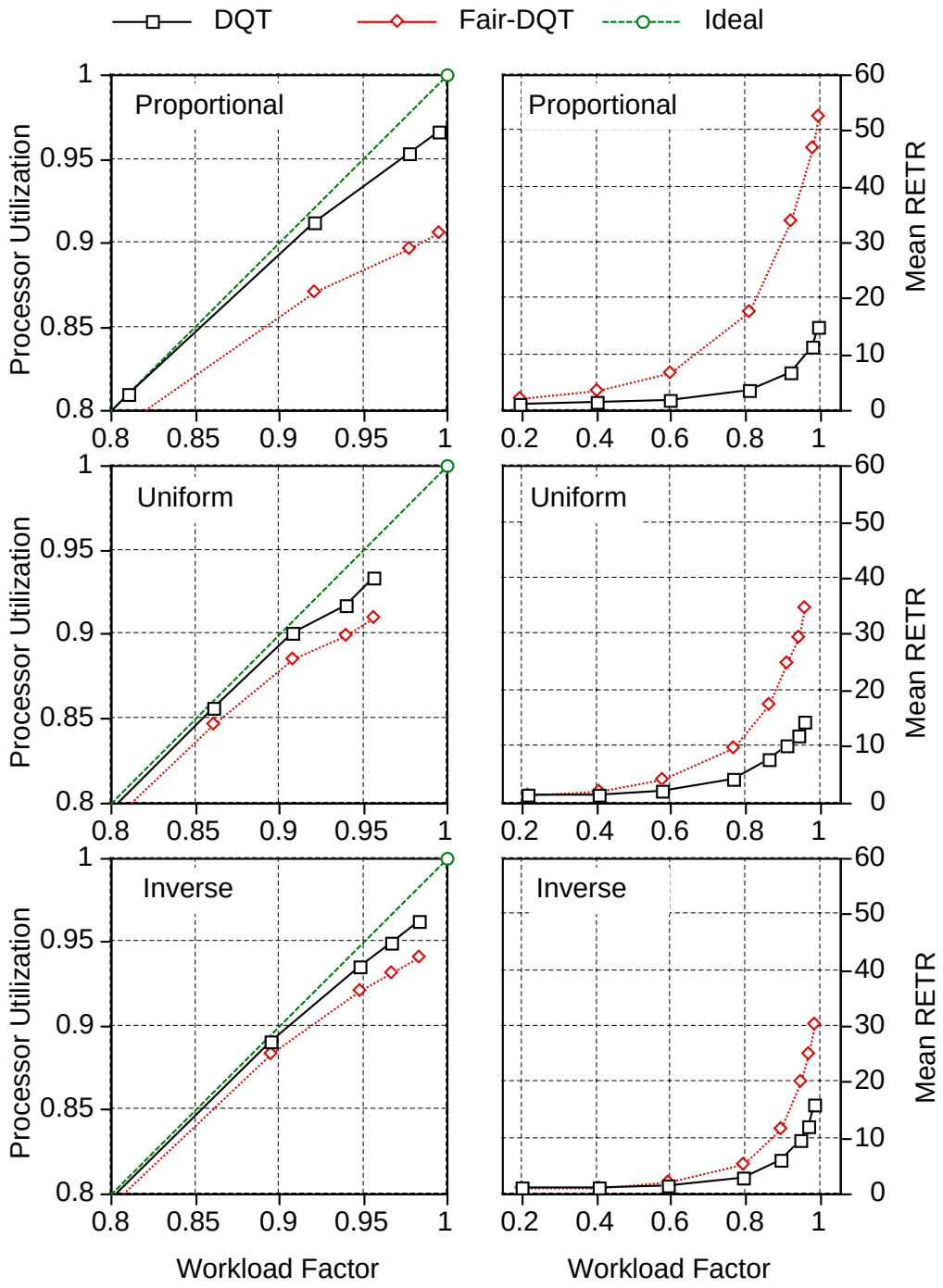


図 7.19: DQT と Fair-DQT の比較（APA ポリシー，1024 プロセッサ）

に関する結果である．この図から，ScanUp は DQT にほぼ匹敵するプロセッサ利用率を示すことが分かる．一方，FCFS-BB は，予想された通り，他に比べかなり悪い結果となっている．これは第 2.2.4 節で述べたように，小さい並列プロセスが大きい並列プロセスの実行を阻害することが原因と考えられる．

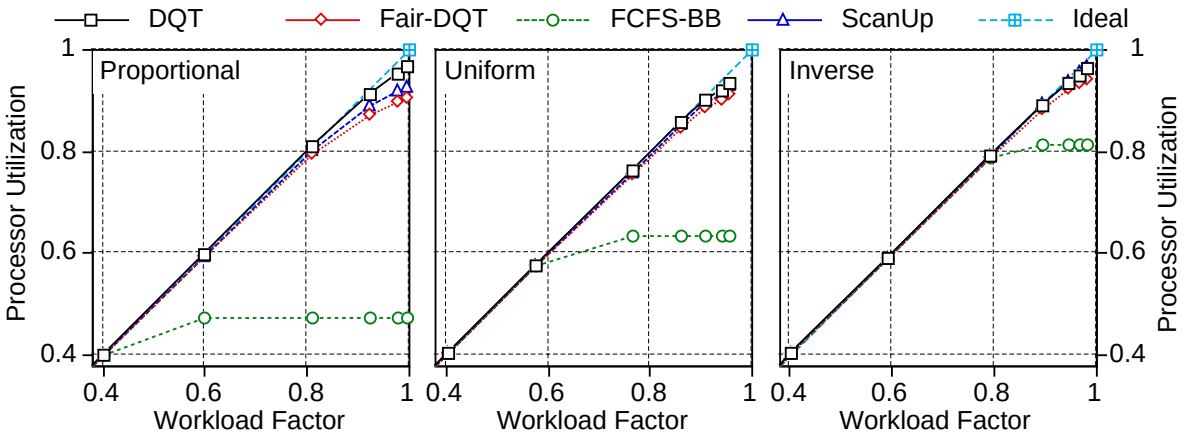


図 7.20: DQT とバッチスケジューリングの比較（APA ポリシー， 1024 プロセッサ）

7.2.6 公平さの評価

今までの評価においてはプロセッサ利用率と RETR についてのみ着目してきたが，ここではスケジューリングの公平さについて評価する．図 7.21 は，図 7.20 のシミュレーションにおいて，並列プロセスサイズ（$log_2(process\_size)$）と RETR の総関係数を Y 軸としたグラフである．

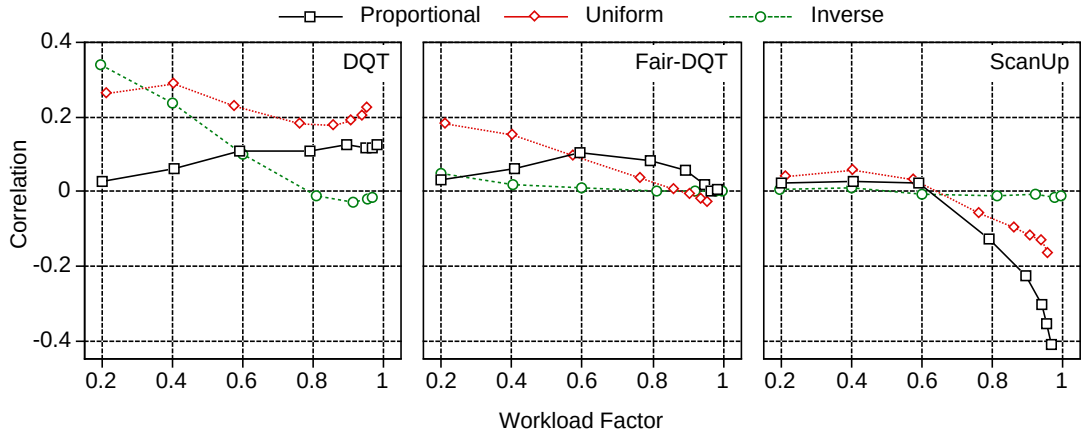


図 7.21: 並列プロセスサイズと RETR の相関係数（APA ポリシー， 1024 プロセッサ）

図 7.21 から，ほとんどの場合 DQT の相関係数は正であることが分かる．これは，大きいサイズのもの程，より少なくスケジューリングされることを意味する．これは DQT のスケジューリング特性が，負荷の不平衡により生じるプロセッサ利用率の低下を，より小さいサイズの並列プロセスをスケジューリングすることで解消しているために生じ

た結果と考えることができる．一方，Fair-DQT においては，期待されるように DQT よりも公平なスケジューリングが実現されている．

一様および逆比例の分布において，DQT の相関係数は低負荷時に高く，高負荷時に低くなる傾向を示している．これは DQT の空間分割の特性から来るものと考えることができる．DQT のように入れ子構造を持つ空間分割の場合，小さい並列プロセスに対しては，複数の，それも小さい程多くのプロセス走行待ち行列があることになる．このため，小さい並列プロセスは，平均的に待ち行列の長さが短くなり，その結果 RETR も小さくなったものと考えられる．

Scan スケジューリングでは公平にスケジューリングされると報告 [KLDR94] されている．しかしながら，図 7.21 からは必ずしもそうではないことが示されている．文献 [KLDR94] において，逆比例の分布しか評価されていない．図 7.21 においても逆比例分布においては理想的な値を示しているが，その他の分布においては公平とは言い切れない結果となっている．

7.3 SCore-D による評価

これまで DQT の評価は全てシミュレーションベースで行なってきたが，ここでは SCore-D の時空間分割スケジューラとして DQT を実装し，そこでの評価結果をシミュレーションと比較する．

図 7.22 に本節での比較方法を示す．シミュレーションでの実行状況を SCore-D での実行状況に合わせるために，まず最初にシミュレーションの結果をファイルに出し，その結果に従って SCore-D 下で実行される並列プロセスを生成する方式とした．実際に SCore-D 下で走らせるプログラムは指定された時間だけループで回る単純なものである．実行時間（仕事量）を規定するため並列プロセス内でタイマを用いることはできない．このためループ回数は指定時間にできだけ近くなるように調整したが，キャッシュなどの影響により，ループ回数の大小によって 1 割程度の誤差が生じる．

投入した並列プロセスの総数は 1524 であり 10,000 単位時間で並列プロセス投入を打ち切っている．SCore-D における単位時間は 1 秒に設定した．第 7.2.2節で用いたのと同じ勾配負荷とし，投入する並列プロセスの大きさの分布は一様である．基本的な設定は第 7.2 節で説明した通りであるが，今回に限り並列プロセスの最大は 64 プロセッサとした．

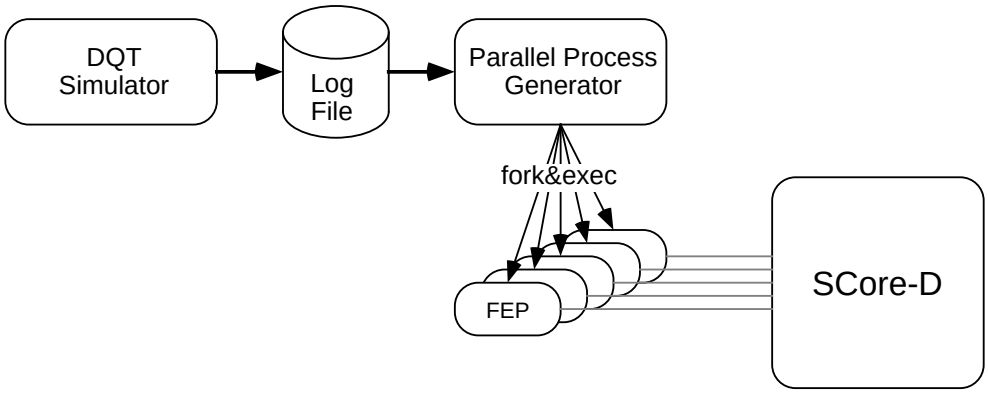


図 7.22: SCore-D による DQT の評価方法

シミュレーションによる結果を図 7.23 に，また SCore-D による結果を図 7.24 に示す．これらの図では RETR が並列プロセスの終了時点でプロットされている．図 7.23 と図 7.24 を比較すると，全体の傾向は良く似ていることが分かる．表 7.2 には，図 7.23 の結果を並列プロセスの大きさ毎に RETR の平均を示す．この表から明らかなように SCore-D における実行結果はシミュレーションに比べ劣っている．

表 7.2: シミュレーションと SCore-D による DQT の性能比較

Process Size	Number of Processes	Mean RETR	
		Simulation	SCore-D
1	221	1.88	2.18
2	223	1.88	2.03
4	229	1.89	2.08
8	199	1.86	1.95
16	218	2.06	2.40
32	204	2.27	2.58
64	230	2.57	3.17
	(Total) 1524	(Ave.) 2.06	(Ave.) 2.35

シミュレーションは勾配負荷，つまりシミュレーション時間が経過するにつれ負荷が高くなるように設定されたいるため，評価の前半では負荷が低く，後半は負荷が高い状

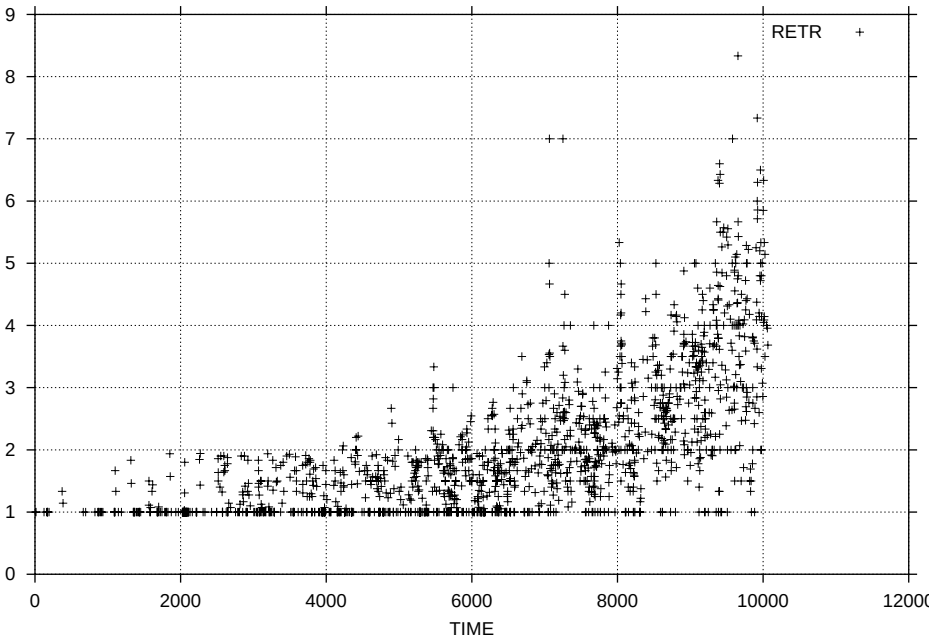


図 7.23: DQT のシミュレーション結果（64 プロセッサ）

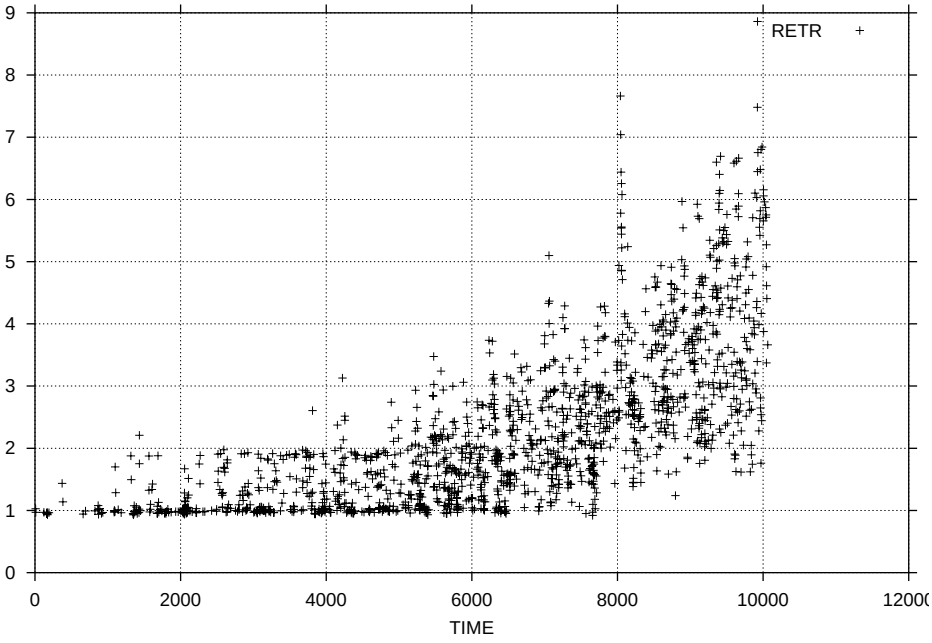


図 7.24: SCore-D における結果（64 プロセッサ）

況になっている．負荷の程度による SCore-D の挙動の違いを明確にするため，表 7.2 と同じ結果について， 5,000 単位時間（SCore-D では 5,000 秒）未満に終了した並列プロ

セスについて集計した結果を表 7.3 に，5,000 単位時間（秒）以降に終了した並列プロセスについて集計した結果を表 7.4 に示す．

表 7.3: シミュレーションと SCore-D による DQT の性能比較（低負荷時）

Process Size	Number of Processes	Mean RETR	
		Simulation	SCore-D
1	60	1.23	1.31
2	60	1.25	1.29
4	63	1.19	1.23
8	52	1.16	1.18
16	42	1.08	1.18
32	61	1.21	1.19
64	65	1.47	1.60
	(Total) 403	(Ave.) 1.23	(Ave.) 1.30

表 7.4: シミュレーションと SCore-D による DQT の性能比較（高負荷時）

Process Size	Number of Processes	Mean RETR	
		Simulation	SCore-D
1	161	2.12	2.49
2	163	2.11	2.29
4	166	2.15	2.41
8	147	2.10	2.22
16	176	2.30	2.69
32	143	2.72	3.16
64	165	3.00	3.77
	(Total) 1121	(Ave.) 2.54	(Ave.) 2.72

これらの表から，SCore-D による実行結果は負荷が高い時に悪くなる傾向が見られることが分かる．このような結果となった理由としては以下に示す要因が考えられる．

- 並列プロセスを生成するためのオーバヘッド
- 並列プロセスが終了した際の SCore-D 内の処理により生じるオーバヘッド
- 第 4 章で示したギャングスケジューリングのオーバヘッドおよび DQT スケジューリングのオーバヘッド
- 投入した並列プロセスの時間誤差

実際の使用状況では，短い時間間隔で並列プロセスの投入要求がやって来ることも考えられる．このような状況では， TAP のための負荷情報の更新が間に合わず，より悪い方向にスケジューリングされる場合もあり得る．従って，ここでの SCore-D によるスケジューリング性能よりも悪化する可能性がある．

7.4 DHC との比較

DQT は基本的に DHC [FR90b, FR90a, FR96] のアイデアをベースにしている．DQT と DHC では，例えば，スケジューリングアルゴリズムが大きく異なっている．本節では， DQT と DHC とのより本質的な違いについて明らかにする．

第 2.2.5 節において DHC について紹介したが， DHC においては並列プロセスという概念が明確になっていない．このため， DHC における分散制御ツリーは，並列プロセスのスケジューリングと並列プロセスを構成するプロセスの制御というふたつの制御構造を反映している．一方， DQT においては並列プロセスのスケジューリングは別なツリー構造としている．

図 7.25 は DQT におけるスケジューリングと並列プロセスのそれぞれの制御構造を一緒に示したものである．この図に示されたように， DQT においてスケジューリングのための分散ツリー構造は，並列プロセス制御のためのツリー構造と直交している．

第 6 章で述べたように，並列プロセスには大域的な状態が存在し，その意味でも並列プロセスという概念は重要である．また，第 4.3.1 節に示したように，並列プロセスの制御構造は 2 分木が最適とは限らない．一方，スケジューリングとしての木構造は，パーティションの構造を反映しており，多くの場合， 2 分木が有利である．また DHC で提案されているような selective disabling や時分割間隔を動的に変化させるといった方式は， DQT にもそのまま適用可能であると考えられる．

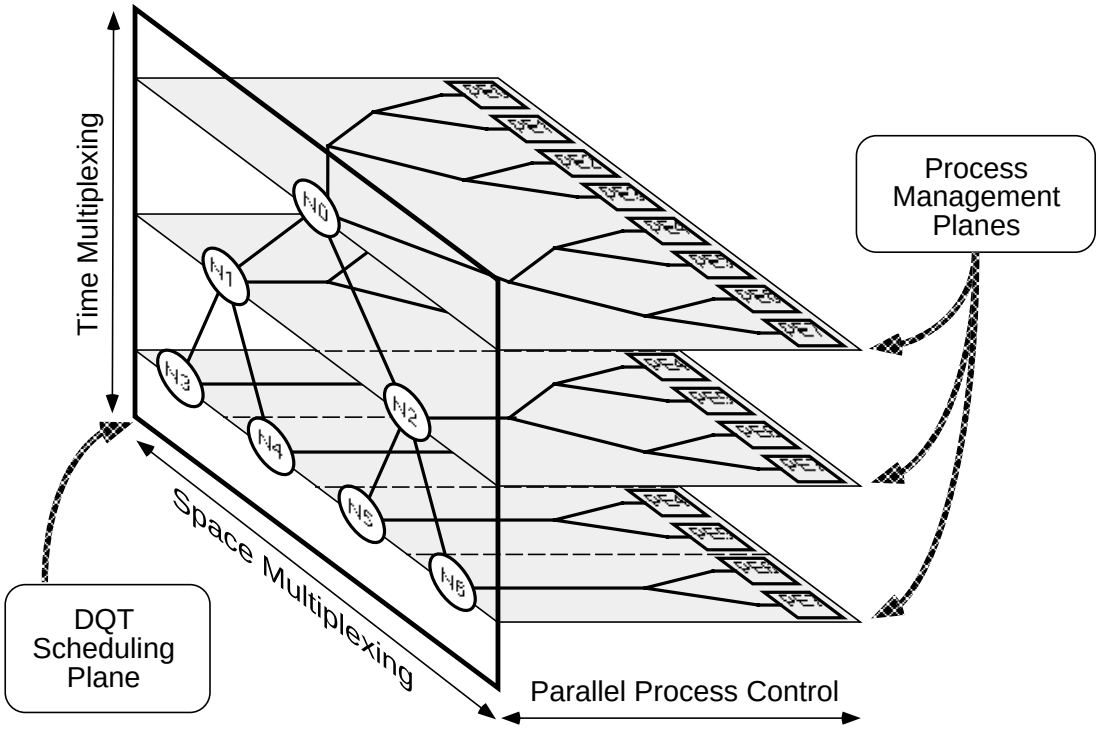


図 7.25: DQT におけるスケジューリングと並列プロセス制御の関係

7.5 まとめ

本稿では，時空間分割スケジューリングの方式のひとつである DQT を提案し，そのいくつかの特性の解析とシミュレーションによる評価を試みた．その結果，適当な TAP の下に低負荷から高負荷に至るまで良好な性能を示すことが確認された．また，並列プロセスサイズの分布による影響も比較的わずかなものであることが判明した．並列プロセスサイズの分布の違いにおける傾向としては，並列プロセスサイズに逆比例した場合が最も良い結果を示し，一様分布の場合が最も悪い結果を示した．スケジューリング上の公平さという面では，並列プロセスサイズと実実行時間比に正の相関が見られ，大きい並列プロセス程不利なスケジューリングとなることが判明した．

DQT は SCore-D 上に実装され，そこでの評価も行なった．同じ条件のシミュレーション結果との比較では，平均 RETR が 1 ～ 2 割程度大きくなることが判明した。 この原因はシミュレーションでは無視されているいくつかの要因が影響していると思われる．

第 8 章

まとめ

本研究は分散記憶方式並列計算機上に効率的かつ実用的な時分割スケジューリングが構築可能であることを実証するために，以下の課題に取り組んできた．

- 並列計算機における低オーバヘッドな時分割スケジューリングの実装方式の提案
- 並列計算機における時分割スケジューリング実現手法の比較
- 被スケジューリング並列プロセスの大域状態の検出手法
- 時分割スケジューリングと空間分割スケジューリングを融合したスケジューリング方式の提案
- 時分割スケジューリングと空間分割スケジューリングを融合したスケジューリング方式とこれまでに提案されたバッチスケジューリング方式の比較

ユーザレベル通信と多重並列プログラミング環境を両立させるためにネットワークプリエンプション方式を提案した．同時に，ネットワークプリエンプション方式を検証，評価するために， RWC PCC-II 上に SCore-D と名付けられた時空間分割スケジューラを構築した．その結果， SCore-D においては 64 ノード，時分割間隔 100 $msec$ におけるギャングスケジューリングオーバヘッドが 4 % 以下で実現されることを示した．この 100 $msec$ で 4 % という値は，スケジューリングオーバヘッドとして妥協可能な値であると同時に，対話処理における十分な応答性を確保できる値と考えることができる．

一方，並列計算機におけるもうひとつの時分割スケジューリングの実現手法として非同期コスケジューリングを実装し， SCore-D が提供するギャングスケジューリングとの比較を行なった．その結果，非同期コスケジューリングでは，アプリケーションの通信

パターンによっては不安定な挙動を示すのに対し，ギャングスケジューリングは安定した結果を示すことが判明し，ギャングスケジューリングの有意性が実証された．

ユーザ並列プロセスの大域状態の検出手法として，ギャングスケジューリングをユーザレベル通信において実現するために提案されたネットワークプリエンプション方式を応用することを提案し，ほとんどギャングスケジューリングのオーバヘッドのみで大域状態の検出が可能になることを示した．64 ノード，時分割間隔 100 $msec$ にいて 4 % 以下というスケジューリングオーバヘッドと，並列プロセスの大域状態検出機構により，効率的な対話処理環境が並列計算機上に実用的な範囲で実現可能であることを示した．

時分割スケジューリングと空間分割スケジューリングを融合したスケジューリング方式として DQT を提案し，シミュレーションによりその特性を解析した．また，空間分割バッチスケジューリングとして効率的であるとされている Scan スケジューリングと比較し， DQT が Scan に比べ遜色ないスケジューリング性能が示されることを確認した．

以上の結果から，本研究の目標である，分散記憶方式並列計算機上に効率的かつ実用的な時分割スケジューリングが構築可能であることが証明されたと考える．

8.1 本研究では検証できなかったいくつかのアイデア

以下，本章では，研究の途中で生じたが，検証できなかったいくつかのアイデアと，問題について認識しているが解決の目処が立たなかった問題について述べる．

8.1.1 ネットワークプリエンプションのその他の応用

これまで，ユーザレベル通信とギャングスケジューリングを同時に実現する手段としてネットワークプリエンプションを提案し，また，それのひとつの応用としてユーザ並列プロセスの大域状態の検出方法を提案してきた．本章では，ネットワークプリエンプションの別な応用のアイデアについて述べる [HTI+96].

Consistent Checkpointing

走行している並列プロセスの状態を退避し，ハードウェア等の障害時には退避された状態から並列プロセスの実行を再開しようとするのが consistent checkpointing である．逐次プロセスの checkpointing とは，*i)* 通信メッセージの退避と復帰，*ii)* 並列プロセス全体での一貫性の取れた状態の退避と復帰，の 2 点で大きく異なり，これらが consistent checkpointing 実装の重要なポイントとなっている [Pla93, PL96, CLP97].

これらの 2 点に関してはネットワークプリエンプションを用いた並列プロセス切替で実現可能である．通信メッセージはネットワークコンテキストとしてメモリに退避され，並列プロセスの実行中断時には並列プロセス全体での一貫性は維持されている．従って，ネットワークプリエンプションによる consistent checkpointing は，並列プロセスの実行を中断し，メモリに退避されたネットワークコンテキストをファイルに保存すれば，後は個々のプロセスの状態をディスクに退避すれば良いことになる [西岡 99].

大域ガーベジコレクション

大域ガーベジコレクション（Garbage Collection:GC）においてもネットワーク中のメッセージの取り扱いが問題となる．局所的な GC では「活きているオブジェクト」は GC ルートから辿ることで検出される．しかしながら，分散環境における大域 GC では，ネットワーク中のメッセージに含まれるポインタから指されるオブジェクトも「活きている」と見做されなければならない．

ネットワークプリエンプションによる並列プロセス切替時には，ネットワーク中のメッ

セージはメモリに退避される．もし，プロセス切替のタイミングで大域 GC を起動するならば，通常の GC ルートに退避されたネットワークコンテキストを含めることで，メッセージに含まれるポインタを辿ることが可能になる．

8.1.2 並列プロセスのマイグレーション

第 7 章で提案した DQT 時分割空間分割スケジューリングでは，並列プロセスのマイグレーション（移送）は考慮されていない． DQT では一度生じた負荷の不均衡は，新たに投入される並列プロセスによってのみに解消される．負荷の高い部分の並列プロセスが終了することで負荷が平衡する場合も考えられなくはない．しかし，多くの場合，低負荷下での並列プロセスの平均実行時間は，高負荷下での並列プロセスの平均実行時間より短くなる傾向にある．このため，新たに並列プロセスが投入されない状況では，負荷の不平衡は拡大する傾向にある．

並列プロセスのマイグレーションが可能であるならば， DQT における負荷の不平衡を積極的に解消することが可能になる．並列プロセスのマイグレーションは，基本的には第 8.1.1 節で示した consistent checkpointing が応用できる． Consistent checkpointing を用いた並列プロセスのマイグレーションは次のような手順になる．

1. マイグレーションの対象となる並列プロセスの状態を consistent checkpoint により退避する．

2. 対象並列プロセスを強制終了する．

3. Consistent checkpointing で退避された並列プロセスの状態を復帰し，再スタートする．

ここで DQT から見た場合，再スタートを新たな並列プロセスの投入と見做すことで結果的に負荷の不均衡を解消するような並列プロセスのマイグレーションが実現できたことになる．

実際には， consistent checkpointing で退避された並列プロセスの状態に含まれるネットワークコンテキストが，ネットワーク中のパーティションの絶対的な位置に依存しないことが条件となる．残念ながら現在の PM のネットワークコンテキストはネットワークの絶対的な位置に依存しているため，そのままでは並列プロセスのマイグレーションは実現できない．

8.1.3 メモリ資源とジョブスケジューリングの関連

Unix においてメモリ資源とジョブスケジューリングは密接な関係を保ち，メモリ資源の有効活用を目指している [Bac86, LMKQ89]．要求時ページングやプロセススワップにより緊急性の低いと思われるメモリ領域をディスクに退避することでメモリ資源を見かけ上より多く見せることに成功している．

本研究で示してきたギャングスケジューリングによる時分割スケジューリングでは，メモリ資源が無限にあると仮定し，要求時ページングやプロセススワップの必要性を無視してきた．しかしながら，ギャングスケジューリングが実現する効率的な多重並列プログラミング環境は，結果的により多くのメモリ資源を必要とし，なんらかのメモリ資源の制約を緩和する機構が必要になる可能性を意味している．はたして逐次 OS で用いられているこれらのテクニックは，そのまま並列計算機において応用可能であろうか．

Burger らは，ワークステーションクラスタ上での実験を通じ，要求時ページングが並列アプリケーションの性能に与える影響について報告している [BHMW94]．その結果，いくつかの並列アプリケーションにおいてページフォールトの発生により 30 倍以上も実行が遅くなることが示されている．ギャングスケジューリングは，通信の相手となるプロセスが走行状態であることを保証し，通信の待ち時間を最小限にすることを目的としている．要求時ページングは通信の相手となるプロセスが走行状態でなくなる可能性を意味し，ギャングスケジューリングの目的に反することになる．その結果，非常に非効率的な並列プログラムの実行となることが容易に予想される．

ギャングスケジューリングの目的に沿うために，並列プロセスを構成するあるプロセスでページフォールトが発生した時に，そのメモリページがディスクから復帰されるまで並列プロセス全体をスケジューリングから外す，という方策が考えられる．しかしながらこの方策では，並列プロセスがスケジューリングから外される頻度は，個々のプロセスがページフォールトを起こす頻度の総和になり（同時にページフォールトが発生しないと仮定），結果的に大きな（多くのプロセッサを必要とする）並列プロセスほど，ページフォールトで並列プロセスの実行が中断される頻度が高くなる．もし，全ての並列プロセスで一斉に要求時ページングが発生するならば，ページフォールトによる並列プロセス実行の中断の頻度は，逐次の場合と同じとなるが，SPMD 型の並列実行モデルではそのような仮定が成り立つアプリケーションは稀であると考えなければならない．以上のように逐次 OS で用いられている要求時ページングをそのまま並列 OS に応用す

るのは簡単ではない.

では，プロセススワップはどうであろうか．並列プロセスのスワップは，第 8.1.1 節で述べたような consistent checkpointing で実現可能である．複数の並列プロセスが走行している状態で，メモリ資源がある閾値を越えて足りなくなりそうになった時点で，適当な並列プロセスを選び， consistent checkpointing によりディスクに退避すれば良い．この方式では，複数の並列プロセス走行時の場合にのみ有効であり，単一の並列プロセスでメモリ資源が不足した場合には無効である，という欠点がある．ここでスケジューリングという観点からは，並列プロセスを consistent checkpointing によりスワップする場合，どのようにスワップの対象となる並列プロセスを選択し，スワップをどのように制御するのが効率的か，などという点に関して研究の余地が残されている．

8.1.4 並列プロセス間通信を意識したスケジューリング

Cm* の OS である Medusa において互いに通信し合う複数の（逐次）プロセスを同時にスケジューリングするという “Buddy Scheduling” 方式が提案されている [OSS80]. ここではこのアイデアを拡張し，複数の互いに通信し合う並列プロセスについて考える．

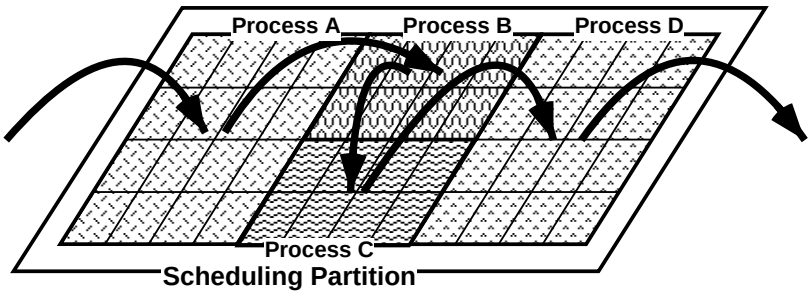


図 8.1: 並列プロセスの Buddy スケジューリングの例

図 8.1 にこのような buddy scheduling の例を示す．ここで，並列プロセス群（図中 “Process A” – “Process D”）は矢印で示されたような並列プロセス間通信を行なっているとする．これは 4 つの並列プロセスが Unix のシェルにおけるパイプで接続されているような関係である．このような並列プロセス間通信は Unix におけるパイプ接続された場合に比べ，以下の示すような利点がある．

- より単純な機能の並列プログラムを組み合わせることで，高機能な並列プログラム

と同等な処理が可能になる．

- それぞれの並列プロセスは，Unix の場合と異なり，本当に並列で動く可能性があるため，高いスループットが期待できる．
- それぞれの並列プロセスの処理スループットが均等になるようなプロセッサ数を割り当てることで，全体のスループットの向上を見込むことができる．

ギャングスケジューリングの利点とほぼ同じ理由から，これら 4 つの並列プロセスは同期してスケジューリングされた方が効率的であることは明らかである．そこで，これら全ての並列プロセスを包含するようなパーティション（図中 “Scheduling Partition”）を考え，そのパーティションに含まれる並列プロセスは同期してスケジューリングされるようにすれば良い [堀 94a].

8.2 残された研究課題

時分割空間分割スケジューリングと並列プロセスの優先度

Unix においては対話処理プログラムのみかけの応答時間を短くするということと，高い頻度のプロセス切替により生じるスケジューリングのオーバヘッドを低減する，という矛盾する要求に対し，multi-level feedback queue という機構を用いている [LMKQ89]. Multi-level feedback queue においては，プロセスが消費した CPU 時間や待ち時間などによりスケジューリングの優先度が決まり，その優先度に応じてスケジューリングの頻度が決定される．また，多くのマルチタスク OS においてもプロセス（タスク）毎に優先度が指定でき，プロセスの重要度に応じてスケジューリングの頻度や順序を規定することができるようになっている．このように優先度は多くの OS において備わっているスケジューリングの基本的な枠組と言うことができる．

しかしながら，時分割空間分割スケジューリングにおいてスケジューリングの優先度を設定することは簡単ではない．図 8.2 は，ある DQT において優先度がその目的通りに機能しない場合を示したものである．この図における丸は DQT のノードを示し，丸の中の数字はその DQT のノードにある並列プロセスの優先度を示している．ここでは数字が大きいほど優先的にスケジューリングされることを意図しているものとする．また，この DQT においてシェードされている部分は，ここでの説明において注目されていない部分を示す．

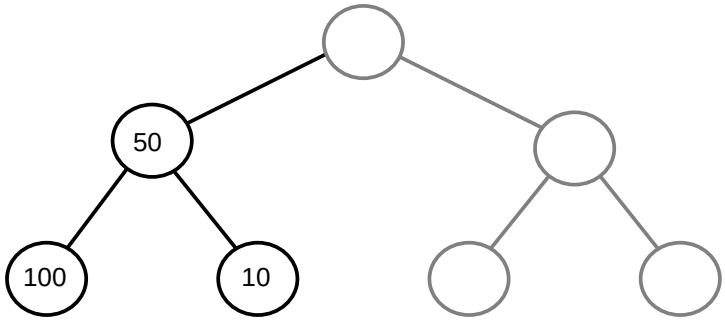


図 8.2: DQT においてスケジューリング優先度に矛盾が生じる例

優先度 100 の並列プロセスはそれより低い優先度の並列プロセスよりも優先的にスケジューリングされなければならない．しかしながら，この図において優先度 100 の並列プロセスをスケジューリングしようとすると，優先度 10 のプロセスも一緒にスケジューリングされてしまう可能性がある．しかし，優先度 10 のプロセスをスケジューリングするよりも，優先度 50 を持つ並列プロセスをスケジューリングすべきである．ところが今度は，優先度 50 の並列プロセスをスケジューリングしようとするとその子ノードにはより優先度の高い並列プロセスが存在する．ここで，敢えて優先度の低い並列プロセスをスケジューリングしないという方策も考えられるが，そうするとプロセッサ資源が使われないで無駄になってしまう．

DQT スケジューリングに厳密に優先度を導入するということとプロセッサ利用率を最大限にしようとすることは，ここに示した例のように矛盾する可能性がある．ここでの可能な解のひとつは，スケジューリング優先度の指定とはその下限を指定するものであり，スケジューリングの状況によっては優先度は他の並列プロセスにより引き上げられる場合も許す，というものである．

謝辞

東京大学工学部田中英彦教授，井上博允教授，武市正人教授，坂井修一助教授，東京大学生産技術研究所喜連川優教授からはもったいなくも貴重な助言を賜わりました．東京大学工学部近山隆教授は本研究をまとめるにあたりお忙しい中，多くの時間を割いていただきました．著者が ICOT の仕事を通じて得た貴重な体験は本研究の基礎となっています．

本研究は著者が技術研究組合新情報処理開発機構に出向中に行なわれました．所長島田潤一博士は著者の長年に渡る研究活動の機会を与えていただきました．研究企画部長尾内理紀夫博士からは学位取得に関する暖かい助言と励ましの言葉を頂きました．並列分散システムソフトウェア研究室室長石川裕博士は，著者が新情報処理開発機構に出向してきた頭初から 6 年間の長期に渡り，学界に不慣れであった著者を親切に導いてくれました．並列分散システムソフトウェア研究室の研究員である手塚宏史氏，住元真司氏，原田浩氏，高橋俊行氏，並びに並列分散システムソフトウェア研究室の前身であります超並列ソフトウェア研究室の小中裕喜博士（現三菱電機株式会社），前田宗則氏（現株式会社富士通研究所），友清孝志氏（現カーネギーメロン大学），Jörg Nolte 博士（現 GMD First 研究所）との研究討論から多くのアイデアを生み出すことができました．PM 通信ライブラリなしには SCore-D の開発はあり得ませんでした．手塚宏史氏は PM の仕様に対する著者の我儘な要求を全て実現してくれました．本研究の基本的なアイデアは，旧超並列アーキテクチャ研究室室長坂井修一博士（現東京大学助教授），横田隆史博士（現三菱電機株式会社），松岡浩氏（現日本電気株式会社）との研究討論から多くの影響を受けています．並列分散システムパフォーマンス研究室室長佐藤三久博士からは並列処理の考え方を教えていただきました．

株式会社 SRA の曽田哲之氏からは豊富な Unix プログラミング経験に基づく多くの助言を頂きました．同亀山豊久氏，西中芳幸氏には快適なプログラム開発環境を整備，運営していただきました．同山田務氏は実時間負荷モニタの GUI を開発してくれまし

た．株式会社エム・アール・アイ システムズの Francis O'Carroll 氏は MPICH-PM の実装により NAS 並列ベンチマークの実行を可能にしてくれました．同清水友晴氏は，SCore-D の SR2201 への移植を，同飯尾淳氏は SCore-D の Java 言語によるモニタ GUI を作成してくれました．また，同西岡利博氏は SCore-D に consistent checkpointing 機能を追加してくれました．東清物産株式会社土屋尚久部長は RWC PC Cluster の設計，組立に非常に大きな役割を果たしてくれました．

株式会社三菱総合研究所情報技術研究センター長岡本吉晴氏および情報技術開発部部長小林慎一氏には著者の長期に渡る出向を暖かく見守って頂きました．著者の元上司であります芝浦工業大学システム工学部川面恵司教授および東京大学教養学部玉井哲雄教授は著者の研究生活を案じて下さいました．

父堀四郎は 1998 年 2 月に他界しました．父の生前に本研究を完成することができなかったことを非常に残念に思います．妻の夫佐は苦労を分かち合ってくれました．長女美沙希は一緒に遊ぶ時間を割いてくれました．ありがとう．

参考文献

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53–79, February 1992.

[ACP+95] Thomas E. Anderson, David E. Culler, David A. Patterson, et al. A Case for NOW (Networks of Workstations). *IEEE Micro*, Vol. 15, No. 1, pp. 54–64, February 1995.

[ADCM98] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *SIGMETRICS'98/PERFORMANCE'98 Joint Conference on Measurement and Modeling of Computer Systems*. ACM, June 1998.

[ADV+94] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. UC Berkeley Technical Report CS-94-838, Computer Science Division, University of California, Berkeley, 1994.

[AYHI96] Daniel Andresen, Tao Yang, Vegard Holmedahl, and Oscar H. Ibarra. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. In *10th International Parallel Processing Sympojium*, pp. 850–856, April 1996.

[Bac86] Maurice J. Bach. *The Design of The $UNIX^{TM}$ Operating System*. Prentice-Hall, 1986. 日本語訳, 坂本文, 多田好克, 村井純訳, 共立出版.

[BBLS93] D. H. Bailey, J. T. Barton, T. A. Lasinski, and H. D. Simon. The NAS Parallel Benchmarks. NASA Technical Memorandum 103863, NASA Ames Research Center, 1993.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, Vol. 15, No. 1, pp. 29–36, February 1995.

[Beo] http://cesdis.gsfc.nasa.gov/linux-web/beowulf.html.

[BGW93] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System*, Vol. 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[BHMW94] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors. In *Supercomputing'94*, pp. 590–599, November 1994.

[Bla90] David L. Black. *Scheduling and Resource Management Techniques for Multiprocessors*. PhD thesis, Carnegie Mellon University, 1990.

[CL85] Mani Chandy and Leslie Lamport. Distributed snapshot: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63–75, February 1985.

[CLP97] Yuqun Chen, Kai Li, and James S. Plank. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. In *Supercomputing'97*, 1997.

[CMC97] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnect'97*, 1997.

[CS87] Ming-Syan Chen and Kang G. Shin. Processor Allocation in an N-Cube Multiprocessor Using Gray Codes. *IEEE Transactions on Computers*, Vol. 36, No. 12, pp. 1396–1407, 1987.

[CS90] Ming-Syan Chen and Kang G. Shin. Subcube Allocation and Task Migration in Hypercube Multiprocessors. *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1146–1155, 1990.

[CT92] Po-Jen Chuang and Nian-Feng Tzeng. A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers. *IEEE Transactions on Computers*, Vol. 41, No. 4, pp. 467–479, 1992.

[DAC96] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *SIGMETRICS 96-5/96 Philadelphia*. ACM, May 1996.

[DAS] http://www.cs.vu.nl/~bal/das.html.

[DBLP97] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *11th International Parallel Processing Symposium*, pp. 388–396, April 1997.

[Dow97] Allen B. Downey. Using Queue Time Predictions for Processor Allocation. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 1291 of *Lecture Notes in Computer Science*, pp. 35–57. Springer-Verlag, April 1997.

[Fei95] Dror G. Feitelson. Packing Scheme for Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*, pp. 89–110. Springer-Verlag, April 1995.

[Fei97] Dror G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems (extended version). Research Report RC 19790 (87657), IBM T.J. Watson Research Center, August 1997.

[FG97] Marco Fillo and Richard B. Gillert. Architecture and Implementation of MEMORY CHANNEL 2. *Digital Technical Journal*, Vol. 9, No. 2, 1997.

[FPR96] Hubertus Franke, Pratap Pattnaik, and Larry Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems. In *Frontier'96*, pp. 1–9, October 1996.

[FR90a] Dror G. Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *COMPUTER*, Vol. 23, No. 5, pp. 65–77, May 1990.

[FR90b] Dror G. Feitelson and Larry Rudolph. Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control. In *International Conference on Parallel Processing, Vol. I*, pp. 1–8, 1990.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, Vol. 16, No. 4, pp. 306–318, 1992.

[FR95] Dror G. Feitelson and Larry Rudolph. Parallel Job Scheduling: Issues and Approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*, pp. 1–18. Springer-Verlag, April 1995.

[FR96] Dror G. Feitelson and Larry Rudolph. Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control. *Journal of Parallel and Distributed Computing*, Vol. 35, No. 1, pp. 18–34, May 1996.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, editors. *PVM*. The MIT Press, 1994.

[Gib97] Richard Gibbons. A Historical Application Profiler for Use by Parallel Schedulers. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 1291 of *Lecture Notes in Computer Science*, pp. 58–77. Springer-Verlag, April 1997.

[Gil96] Richard B. Gillert. Memory Channel Network for PCI. *IEEE Micro*, Vol. 16, No. 1, pp. 12–18, February 1996.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum, editors. *USING MPI*. The MIT Press, 1994.

[GTU91] A. Gupta, A. Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *ACM SIGMETRICS*, pp. 120–132, 1991.

[GW95] Brent Gorda and Rich Wolski. Time Sharing Massively Parallel Machines. In *1995 International Conference on Parallel Processing*, Vol. II, pp. 214–217, August 1995.

[HHJK96] Mark Henne, Hal Hickel, Ewan Johnson, and Sonoko Konishi. The Making of Toy Story. In *COMPCON'96*, pp. 463–468, February 1996.

[HIK+93] Atsushi Hori, Yutaka Ishikawa, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. Overview of Massively Parallel Operating System Kernel SCore. Technical Report TR-93003, Real World Computing Partnership, 1993.

[HIK+95] Atsushi Hori, Yutaka Ishikawa, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. A Scalable Time-Sharing Scheduling for Partitionable, Distributed Memory Parallel Machines. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, Vol. II*, pp. 173–182. IEEE Computer Society Press, January 1995.

[HIN+95] Atsushi Hori, Yutaka Ishikawa, Jörg Nolte, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. Time Space Sharing Scheduling: A Simulation Analysis. In S. Haridi, K. Ali, and P. Magnusson, editors, *Euro-Par'95 Parallel Processing*, Vol. 966 of *Lecture Notes in Computer Science*, pp. 623–634. Springer-Verlag, August 1995.

[HPV] http://www-csag.cs.uiuc.edu.

[HT96] Atsushi Hori and Hiroshi Tezuka. Hardware Design and Implementation of PC Cluster. Technical Report TR-96017, RWC, December 1996.

[HTI+96] Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, Noriyuki Soda, Hiroki Konaka, and Munenori Maeda. Implementation of Gang-Scheduling on Workstation Cluster. In D. G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, Vol. 1162 of *Lecture Notes in Computer Science*, pp. 76–83. Springer-Verlag, April 1996.

[HTI97a] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Global State Detection using Network Preemption. In D. G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, Vol. 1291 of *Lecture Notes in Computer Science*, pp. 176–183. Springer-Verlag, April 1997.

[HTI97b] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. User-level Parallel Operating System for Clustered Commodity Computers. In *Proceedings of Cluster Computing Conference '97*, March 1997.

[HTI98] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly Efficient Gang Scheduling Implementation. In *SC98*, November 1998.

[HTOI98a] Atsushi Hori, Hiroshi Tezuka, Francis O'Carroll, and Yutaka Ishikawa. Gang Scheduling vs. Coscheduling: A Comparison with Data-Parallel Workload. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Vol. II, pp. 1050–1057, July 1998.

[HTOI98b] Atsushi Hori, Hiroshi Tezuka, Francis O'Carroll, and Yutaka Ishikawa. Overhead Analysis of Preemptive Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, Vol. 1459 of *Lecture Notes in Computer Science*, pp. 217–230. Springer-Verlag, April 1998.

[HYI+95] Atsushi Hori, Takashi Yokota, Yutaka Ishikawa, Shuichi Sakai, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Jörg Nolte, Hiroshi Mat-

suoka, Kazuaki Okamoto, and Hideo Hirono. Time Space Sharing Scheduling and Architectural Support. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*, pp. 92–105. Springer-Verlag, April 1995.

[IHT+96] Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Motohiko Matsuda, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, and Jörg Nolte. MPC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, pp. 429–464. MIT Press, 1996.

[INR] http://lhpca.univ-lyon1.fr.

[Int93] Intel Corporation. *PARAGON OSF/1 USER'S GUIDE*, April 1993.

[Int95] Intel Corporation. *Pentium Pro Family Developer's Manual Volume 3 Operating System Writer's Guide*, December 1995.

[Int96] Intel Corporation. *PARAGON Ssystem Performance Visualization Tools User's Guide*, January 1996.

[IPS96] Nayeem Islam, Andreas Prodromidis, and Mark S.Squillante. Dynamic Partitioning in Different Distributed-Memory Environments. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 1162 of *Lecture Notes in Computer Science*, pp. 244–270. Springer-Verlag, April 1996.

[Ish95] Yutaka Ishikawa. Meta-Level Architecture for Extendable C++. Technical Report TR-94024, RWC, January 1995.

[Ish96] Yutaka Ishikawa. Multi Thread Template Library - MPC++ Version 2.0 Level 0 Document -. Technical Report TR-96012, RWC, September 1996.

[KIT+96] Hiroki Konaka, Yoshiaki Itoh, Takashi Tomokiyo, Munenori Maeda, Yutaka Ishikawa, and Atsushi Hori. Adaptive Data Parallel Computation in the Parallel Object-Oriented Language *OCore*. In *Proc. of the International Conference Euro-Par'96, Vol.I*, pp. 587–596. Springer-Verlag, 1996.

[KLDR94] Phillip Krueger, Ten-Hwang Lai, and Vibha A. Dixit-Radiya. Job Scheduling Is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 5, pp. 488–497, 1994.

[KMY94] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient Parallel Global Garbage Collection on Massively Parallel Computers. In *Supercomputing'94*, pp. 79–88, 1994.

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968.

[Kon97] Hiroki Konaka. *An Efficient Cooperative Framework for Concurrent Objects on Multicomputers*. PhD thesis, Tokyo University, 1997.

[KTM+95] Hiroki Konaka, Takashi Tomokiyo, Munenori Maeda, Yutaka Ishikawa, and Atsushi Hori. A Parallel Object-Oriented Language OCore. In T. Ito and Yonezawa A., editors, *Lecture Notes in Computer Science*, Vol. 907, pp. 167–186. Springer-Verlag, 1995. (Proc. of Intl. Workshop TPPP'94).

[LC91] Keqin Li and Kam-Hoi Cheng. A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. *Journal of Parallel and Distributed Computing*, Vol. 12, No. 5, pp. 79–83, May 1991.

[Lif95] David A. Lifka. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*, pp. 295–303. Springer-Verlag, April 1995.

[LLWN94] Wanqian Liu, Virginia Lo, Kurt Windisch, and Bill Nitzberg. Noncontiguous Processor Allocation Algorithms for Distributed Memory Multicomputers. In *Supercomputing'94*, pp. 227–236, November 1994.

[LMKQ89] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[Mis83] J. Misra. Detecting termination of distributed computations using markers. In *Second ACM Symposium on Principles Distributed Computing*, pp. 290–294, August 1983.

[MOS] http://www.cs.huji.ac.il/mosix.

[Myr] http://www.myri.com.

[NAS] http://science.nas.nasa.gov/Software/NPB/.

[OHT+97] Francis O'Carroll, Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, and Mitsuhisa Sato. Performance of MPI on Workstation/PC Clusters using Myrinet. In *Proceedings of Cluster Computing Conference '97*, March 1997.

[OP97] Knut Omang and Bodo Parady. Scalability of SCI Workstation Clusters, a Preliminary Study. In *11th Internation Parallel Processing Symposium*, pp. 750–755, April 1997.

[OSS80] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, Vol. 23, No. 2, pp. 92–105, February 1980.

[OTHI98a] Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *International Conference on Supercomputing '98*, pp. 243–250, July 1998.

[OTHI98b] Francis O'Carroll, Hiroshi Tezukua, Atsushi Hori, and Yutaka Ishikawa. MPICH-PM: Design and Implementation of Zero Copy MPI for PM. Technical Report TR-97011, RWC, March 1998.

[Ous82] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of Third International Conference on Distributed Computing Systems*, pp. 22–30, 1982.

[PKC97] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient Portable Communication for Workstation Clusters and Massively-Parallel Processors. In *IEEE Concurrency'97*, 1997.

[PL95] Jim Pruyne and Miron Livny. Parallel Processing on Dynamic Resources with CARMI. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*, pp. 259–278. Springer-Verlag, April 1995.

[PL96] Jim Pruyne and Miron Livny. Managing Checkpoints for Parallel Programs. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 1162 of *Lecture Notes in Computer Science*, pp. 140–154. Springer-Verlag, April 1996.

[Pla93] James Plank. *EFFICIENT CHECKPOINTING ON MIMD ARCHITECTURES*. PhD thesis, Printceton University, 1993.

[PN77] James L. Peterson and Theodore A. Norman. Buddy System. *Communication of the ACM*, Vol. 20, No. 6, pp. 421–431, June 1977.

[PT97] Loic Prylli and Bernard Tourancheau. Protocol design for high performance networking: a Myrinet experience. research Report 97-22, Laboratoire de l'Informatique du Parallélisme, July 1997.

[QN95] Wenjian Qiao and Lionel M. Ni. Efficient Processor Allocation for 3D Tori. In *9th International Parallel Processing Sympojium*, pp. 466–471, April 1995.

[RWC] http://www.rwcp.or.jp/lab/pdslab/clusters.

[SHO+94] Sakai Shuichi, Matsuoka Hiroshi, Kazuaki Okamoto, Takashi Yokota, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. RWC-1 Massively Parallel

Architecture. In *Proc. High Performance Computing Conference '94*, pp. 33–38, 1994.

[Sob97] Patrick G. Sobalvarro. *Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, MIT, 1997.

[SPG90] Abraham Silberschatz, James L. Peterson, and Pere B. Galvin. *Operating System Concepts*. Addison-Welsley, 3 edition, 1990.

[STH+98] Mitsuhisa Sato, Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, Satoshi Sekiguchi, Hidemoto Nakada, Satoshi Matsuoka, and Umpei Nagashima. Ninf and PM: Communication libraries for global computing and high-performance cluster computing. *Future Generation Computer Systems*, Vol. 13, pp. 349–359, 1997/98.

[STT95] William Saphir, Leigh A. Tanner, and Bernard Traversat. Job Management R. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*, pp. 106–126. Springer-Verlag, April 1995.

[Thi92a] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, November 1992.

[Thi92b] Thinking Machines Corporation. *NI Systems Programming*, October 1992. Version 7.1.

[THI96] Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. PM: A High-Performance Communicatin Library for Multi-user Parallel Environments. Technical Report TR-96015, RWC, November 1996.

[THIS97] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: A Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking '97*, pp. 708–717, April 1997.

[TOHI98] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *IPPS/SPDP'98*, pp. 308–314. IEEE, April 1998.

[WBvE97] Matt Welsh, Anindya Basu, and Thorston von Eicken. ATM and Fast Ethernet Interfaces for User-level Communication. In *Third International Symposium on High Performance Computer Architecture*, February 1997.

[YMO+95a] Takashi Yokota, Hiroshi Matsuoka, Kazuaki Okamoto, Hideo Hirono, Atsushi Hori, and Shuichi Sakai. A Prototype Router for the Massively Parallel Computer RWC-1. In *Proc. Int. Conf. on Computer Design (ICCD'95)*, pp. 279–284, Austin, Texas, 1995.

[YMO+95b] Takashi Yokota, Hiroshi Matsuoka, Kazuaki Okamoto, Hideo Hirono, Atsushi Hori, and Shuichi Sakai. The Multidimensional Directed Cycles Ensemble Networks for a Multithreaded Architecture. In *Proc. Int. Conf. on High Performance Computing (HiPC'95)*, pp. 355–360, New Delhi, India, 1995.

[Zhu92] Yahui Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, Vol. 16, pp. 328–337, 1992.

[ZRB+93] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John Lo Verso, Michael Leibensperger, Michael Branett, Faramarz Rabii, and Durriya Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *San Diego Conference Proceedings of 1993 Winter USENIX*, pp. 449–468, January 1993.

[横田 95] 横田隆史, 松岡浩司, 岡本一晃, 廣野英雄, 堀敦史, 坂井修一. RWC-1 相互結合網用プロトタイプ・ルータの設計. 信学技報, CPSY95-37, June 1995.

[横田 97] 横田隆史. 単方向性環状結合網 MDCE, hMDCE の提案と細粒度超並列計算機への適用. PhD thesis, 慶應大学, 1997.

[手塚 96] 手塚宏史, 堀敦史, 石川裕. ワークステーションクラスタ用通信ライブラリ PM の設計と実装. 並列処理シンポジウム JSPP'96, pp. 41–48. 情報処理学会, June 1996.

[手塚 98] 手塚宏史, 堀敦史, Francis O'Carroll, 石川裕. RWC PC Cluster II の構築と性能評価. In *HOKKE'98*, pp. 25–30. 情報処理学会, March 1998.

[秋山 98] 秋山泰, 鬼塚健太郎, 野口保, 安藤誠, 斎藤稔. 並列タンパク質情報解析（PAPIA）システムの PC クラスタ上での実現. In *HOKKE'98*, pp. 31–36. 情報処理学会, March 1998.

[情処 93] パネル討論会 並列計算機の実用化・商用化を逡順させる諸要因とは - その徹底分析と克服 - 並列処理シンポジウム JSPP92 報告, April 1993.

[西岡 99] 西岡利博, 堀敦史, 手塚宏史, 石川裕. クラスタにおけるコンシステントチェックポイントの実現. 並列処理シンポジウム JSPP'99. 情報処理学会, June 1999. (to appear).

[石川 94] 石川裕, 堀敦史, 小中裕喜, 前田宗則, 友清孝志. 並列プログラミング言語 MPC++ の実現. 並列処理シンポジウム JSPP'94, pp. 105–112, 1994.

[石川 95] 石川裕, 堀敦史, 手塚宏史, 佐藤三久, 松田元彦, 小中裕喜, 前田宗則, 友清孝志. 並列プログラミング言語 MPC++ のワークステーションクラスタ上での実現. コンピュータシステムシンポジウム, pp. 33–38, November 1995.

[福地 98] 福地健太郎, 松岡聡, 堀敦史, 石川裕. クラスタ型並列計算機における Implicit Co-scheduling の性能評価. In *HOKKE'98*, pp. 43–48. 情報処理学会, March 1998.

[堀 93a] 堀敦史, 石川裕, 坂井修一, 小中裕喜, 前田宗則, 友清孝志, 松岡浩司, 岡本一晃, 廣野英雄, 横田隆史. 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.

[堀 93b] 堀敦史, 石川裕, 小中裕喜, 前田宗則, 友清孝志. 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.

[堀 94a] 堀敦史, 石川裕, Jörg Nolte, 原田浩, 古田敦, 佐藤忠. 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. 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-67, pp. 25–32. 情報処理学会, December 1994.

[堀 94b] 堀敦史, 石川裕, 小中裕喜, 前田宗則, 友清孝志. 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.

[堀 94c] 堀敦史, 石川裕, 小中裕喜, 前田宗則, 友清孝志. 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.

[堀 95] 堀敦史, 石川裕, Jörg Nolte, 小中裕喜, 前田宗則, 友清孝志. Distributed Queue Tree のシミュレーションによる解析. 並列処理シンポジウム JSPP'95, pp. 313–320, May 1995.

[堀 96a] 堀敦史, 手塚宏史, 石川裕, 高橋俊行, 曽田哲之, 堀川勉, 小中裕喜, 前田宗則. マルチスレッド言語のための実行時ライブラリの実装. 計算機アーキテクチャ研究会資料, 96-ARC-117, pp. 37–42. 情報処理学会, March 1996.

[堀 96b] 堀敦史, 手塚宏史, 石川裕, 曽田哲之, 原田浩, 古田敦, 山田努, 岡靖裕. ワークステーションクラスタにおける並列プログラミング環境の実現. 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, 96-OS-73, pp. 121–126. 情報処理学会, August 1996.

[堀 96c] 堀敦史, 手塚宏史, 石川裕, 曽田哲之, 小中裕喜, 前田宗則. 並列プログラム実行環境のワークステーションクラスタ上で の実装. 並列処理シンポジウム JSPP'96, pp. 49–56. 情報処理学会, June 1996.

[堀 96d] 堀敦史, 石川裕, 小中裕喜, 前田宗則, 友清孝志. 時分割空間分割スケジューリング. 情報処理学会論文誌, Vol. 37, No. 7, pp. 1320–1331, July 1996.

[堀 97] 堀敦史, 手塚宏史, 石川裕. ギャングスケジューリングの PC クラスタ上での実装. ハイパフォーマンスコンピューティング研究会資料, 97-HPC-67, pp. 79–84. 情報処理学会, August 1997.

[堀 98a] 堀敦史, 手塚宏史, 石川裕. ギャングスケジューリングの高速化技法の提案. 並列処理シンポジウム JSPP'98, pp. 207–214. 情報処理学会, June 1998.

[堀 98b] 堀敦史, 手塚宏史, 石川裕. ネットワークプリエンプションによるギャングスケジューリングの実現. 情報処理学会論文誌, Vol. 39, No. 9, pp. 2705–2717, September 1998.

[堀 99] 堀敦史, 手塚宏史, 石川裕. ギャングスケジューリングの高速化技法の提案. 情報処理学会論文誌, Vol. 40, No. 5, May 1999. (to appear).

[六沢 97] 六沢一昭. 非同期分散環境における大域状態決定方式. 並列処理シンポジウム JSPP'97, pp. 61–68. 情報処理学会, June 1997.