UNIX系オペレーティング・システムにおける応用プログラムおよびシステム・プログラムの並列実行に関する研究

中　上　泰　一

UNIX系オペレーティング・システムにおける応用プログラムおよびシステム・プログラムの並列実行に関する研究

中山 泰一

目 次

第 1 章

序言

1.1 本研究の目的

近年、半導体技術の進歩により、高性能のマイクロプロセッサ、および大容量のメモリが安価に入手できるようになった。マイクロプロセッサを多数結合することにより、著しく高い性能をもつマルチプロセッサ・システムを優れたコストパフォーマンスで実現することが可能である [Tomi86]。現在、ハードウェアおよびソフトウェアの両面から活発な研究が行われている。

マルチプロセッサ・システムを有効に利用するためには、応用プログラムを並列化するのはもちろんのこと、オペレーティング・システム自体についても再検討が必要になってくる。すなわち、マルチプロセッサ・システムに使用するオペレーティング・システムには、

(1) 応用プログラムの並列処理を効率よくサポートする実行管理機構の用意、

(2) オペレーティング・システムが提供するシステム機能自体の並列化、

が要求される。

本研究では、まず、**(1)** の要求について考える。共有メモリ型のマルチプロセッサ・システムを対象として、高並列細粒度の応用プログラムを効率よく実行させることの

できる並列実行機構を提案する。

従来の並列処理の管理方式としては並行プロセスに基づく方式が広く採用されている。これは、並列に実行すべき処理が発生するとプロセスを生成し、その実行が完了するとプロセスを消滅させるものである。しかしながら、並列実行の要求があるごとに UNIX で使用されているようなプロセスを生成してタスク[1]を実行させる方式では、あまりにオーバーヘッドが大きく、多数のタスクを発生させて処理を行う場合には決して効率的ではない。

最近では、より軽量のスレッド、軽量プロセスなどとよばれるものを用いる方式も提案されている。しかし、軽量プロセスを用いる場合でも、その生成と消滅にはかなりのプロセッサ時間を消費する。また、スタック領域を割り当てることによるメモリ資源の消費も無視し得ない問題となる。たとえば、プロセッサの台数が 100 台の場合、並列に実行することのできる軽量プロセスの個数は 100 個であり、全プログラム中で処理しなければならないタスクが 10000 個発生するからといって 10000 個の軽量プロセスを生成するのは無駄である。

そこで、あらかじめプロセッサの台数と同数の軽量プロセスを用意しておき、これらを繰り返し使用することにより、無用の軽量プロセスの生成を防止する方式を提案する [Tago89b]。本方式をアクティビティ方式とよぶ。タスクの実行中にサスペンドがまったく発生しない理想的な場合には、新しい軽量プロセスを生成する必要がないので、高い効率を実現できる [Tago91]。

しかしながら、上記のアクティビティ方式にも問題点がある。並列化した応用プログラムには、ネストした fork-join 形式のものがしばしば用いられる。そこでは、親タスクが再帰的に子タスクを生成していき、しかも、それぞれの親タスクがそのすべての子タスクの実行の完了を待って後処理を実行する形式となることが多い。この場合、親タスクによる子タスクの完了待ち合わせにより多数のサスペンドが発生するので、

[1]本研究では、並列プログラムにおいて並列に実行される処理単位を「タスク」とよぶ。

上記の方式のままでは顕著な効率の向上が得られない。

本研究では、上記形式のプログラムの実行効率をも向上させるため、アクティビティ方式に「遺言」とよぶ新しいコンストラクトを追加する方式を提案する。これは、後処理を子タスクに「遺言」して親タスクは実行を完了し、最後に処理を終える子タスクに割り当てられていた軽量プロセスがその「遺言」を実行する方式である。この方式に基づいた並列実行管理機構を実現し、プロセッサ時間とメモリ消費量が大幅に節減できることを確かめた [Nakay93]。

次に **(2)** の要求、すなわちオペレーティング・システムのシステム機能自体の並列化について考える。オペレーティング・システムをプロセス・ネットワーク方式で実現することにより、システム機能自体の並列処理が可能になる [Tago84]。本研究では、上記の方式を用い、システム・コール時の応答時間を短縮することにより、既存の応用プログラムの処理時間を短縮することを試みた。システム機能自体の並列度はあまり大きくないが、システム機能のボトルネック防止に有用である。

具体的には、対象として低並列のメッセージ通信型マルチプロセッサ・システムを用い、プロセス・ネットワーク方式を用いて実現したオペレーティング・システムを各プロセッサに分散配置することにより、システム機能の並列化を行った。分散配置を適切に行えば、システムの並列度を上げることが可能である。実現したシステムは広く実用されている UNIX と互換性をもつ。このシステムにおいて、システム・コール時の応答時間を向上させるための並列性の抽出について考察し、プロセス・ネットワークを各プロセッサに分散配置するための指針を与え、この指針による分散配置が適切なものであることを実験により検証した [Nakay92a]。

1.2 本研究の背景

1.2.1 応用プログラムの並列実行

応用プログラムの並行処理をサポートするための実行機構としては、従来のUNIXでは、タスクごとに独立したアドレス空間を有する設計としたプロセスを用意した（軽量プロセスの対語として「重量プロセス」とよぶこともある）。このような設計は、タスクの並行処理を行う際にプロセス間の通信を明示することを要求することから、プログラムの誤りを少なくするという利点がある。これは、従来のオペレーティング・システムは、各種の処理を実行する少数の独立したプロセスが、お互いに協調しながら動作するプロセス集合体としてシステムを実現することを想定して設計されているためである。

しかしながら、プロセスは実現するためのオーバーヘッドが大きい。UNIXでは、プロセスを生成するのに`fork`というシステム・コールを用いる。`fork`システム・コールが呼ばれると、プロセスの完全なコピーを作成するために、プログラム領域のうち読み出し専用のコード領域（テキスト領域）を除いたデータ領域およびスタック領域のすべてがコピーすることを行う。これには多大な処理時間およびメモリを必要とする。従来のオペレーティング・システムでは想定していなかった、多数のタスクを発生させて処理を行う場合には決して効率的ではないという問題点がある。

新しいプロセスが生成された直後に、そのプロセスで`exec`システム・コールが呼ばれ新たなプログラムが起動される場合にも、`fork`システム・コールでコピーされたプログラム領域がほとんど使用されずに無駄になってしまうという問題がある。そこで、Berkelay版のUNIXでは、プログラム領域のすべてを共有する`vfork`システム・コールを用意して、プロセス生成の効率化を図った。しかしながら、これはプロセス間でスタックをも共有するので、`vfork`システム・コールが呼ばれた直後に`exec`システム・コールを呼ぶ場面にしか用いることはできず、一般的な並行処理に適用することはできない。

プロセス生成の際のプログラム領域のコピーに多大なコストがかかるという問題に対処するため、最近の Sun-OS などでは、コピー・オン・ライトの機構を導入している [Ginge87]。これは、fork システム・コールによりプロセスを生成する際に、タスクごとに独立したアドレス空間を有するという設計はそのまま用いるが、その実現方法を工夫して効率化を図るものである。コピー・オン・ライトでは、プロセスが生成された時点では実際にコピーは行わない。親プロセスと子プロセスとの間ではアドレス空間の共有が行われるだけである。その後に、どちらかのプロセスが書き込みを行った場合に初めて、実際のコピーが行われる。コピー・オン・ライトにより実際のコピーは最小限におさえることができ、処理時間の短縮、および、メモリ利用の削減が期待できる。UNIX のテキスト領域のような読み出し専用の領域を共有することも、コピー・オン・ライトの機構によりそのまま実現することができる。

上記のように、プロセスの実現のためのオーバーヘッドを軽減させる試みがなされている。これと同時に、Mach[Rashi86] をはじめ、近年のオペレーティング・システムでは、軽量プロセスの機構をサポートするようになった。軽量プロセスは、全体で単一のアドレス空間を構成する。そのため軽量プロセスの生成、切替、消滅、および軽量プロセス間通信のオーバーヘッドが従来のプロセスに比べて小さい。

たとえば、Mach では、UNIX のプロセスに相当するものを「タスク」[2]とよび、「タスク」を単位として仮想記憶管理などの処理を行う。1 つの「タスク」の中では複数のスレッドを走らせることができる。スレッドは処理の制御の流れの単位であり、軽量プロセスである。同一「タスク」内のスレッドの切り替えは、UNIX におけるプロセスの切り替えに比べて非常に軽い。これは、スレッドが走っている資源の管理は「タスク」の側で行っているため、スレッドを切り替える際に必要な処理が、UNIX のプロセスを切り替える際に必要な処理に比べて少ないからである。

また、共有メモリ型マルチプロセッサ・システム上で、Mach のスレッドを各プロ

[2]Mach での「タスク」という用語は、前節において脚注で説明した、本研究で用いている意味とは異なる。区別のために、カギ括弧を付ける表記とした。

セッサに割り当てて並列実行させれば、マルチプロセッサ・システムの性能を有効に利用できる。Amoeba[Mulle90]、V-System[Cheri88] などにおいても、マルチプロセッサ・システムにおいて並列実行させることが可能な軽量プロセスの機構を提供している。

軽量プロセスは、ウィンドウ・サーバやファイル・サーバなど、複数のクライアントを同時に扱う必要があるサーバを実現する際にも、有効であることが知られている [Tanen85]。最近の Sun-OS ではコルーチン方式による軽量プロセス・ライブラリを提供し、協調して作動する複数のタスクからなるサーバなどのプログラムの記述に便宜を図っている [Sun88]。

国内における軽量プロセスの実現方式に関する研究としては、多田 [Tada90] [Tada92]、新城 [Shinj92] などがある。

上記のように、従来の並列処理の管理方式では、並列に実行すべきタスクが発生するごとにプロセス実体を生成し、その実行が完了するとそのプロセス実体を消滅させる方式をとっている。もちろん、プロセス実体をできるだけ効率よく生成・消滅させる工夫が行われているが、これには限界がある。これに対して、本論文では、あらかじめプロセッサの台数と同数の軽量プロセスを用意しておき、これらを繰り返し使用する方式を提案している。軽量プロセスの生成・消滅の際のコストが削減でき、非常に多数の細粒度のタスクを並列に実行させる場合にはとくに効率の高い並列実行環境を実現することができる。

1.2.2 オペレーティング・システムの並列実行

オペレーティング・システムに応用プログラムのための効率のよい並列処理機構が用意されると同時に、システム機能自体もできるだけ並列に処理される構成とする試みがなされている。

田胡 [Tago84] は、オペレーティング・システムのシステム機能を機能ごとに多数の軽量プロセスに分割し、それらをランデブ通信により結合してシステムを実現する方式を提案した。本方式をプロセス・ネットワーク方式という。この方式により実現されたシステムを、マルチプロセッサ・システムの各ユニットに分散配置してやれば、システム機能自体の並列処理が図れる。

このように、オペレーティング・システムのシステム機能を機能ごとに分割してそれを並列処理に適用する試みは、V-System、Sprite[Ouste88]、Mach など、近年の並列／分散環境に対応したオペレーティング・システムにおいても、ほぼ同時期に始められている。システム機能の多重スレッド方式により、システム機能のボトルネックを防ぐことをめざしている。

V-System では、V-Kernel とよばれる分散核がプロセス間通信や記憶管理などプロセスを実現するために必要な機能を実現し、各プロセッサに配置される。ファイル管理、プログラムの実行管理などは核外の利用者プログラムである、ソフトウェア・サーバによって実現されている。

Sprite のカーネルはモニタの集合体として実現され、共通のモニタによって管理される資源を同時にアクセスしない限り、複数のプロセスがカーネルを実行することができる。単一スレッド方式では、1 つのロックを用いて排他制御を行うのに対して、Sprite の多重スレッド方式では、モニタごとにロックが存在し、モニタ単位で排他制御が行われる。

Mach は UNIX 4.3 BSD と完全なバイナリでの互換性をもつ。この UNIX の機能は核外の UNIX サーバにより実現されており、Mach のカーネル自体は非常に小さい

ものとなっている。MachのカーネルがUNIXのシステム・コールを受け取ると、利用者プログラムにリンクされているUNIXエミュレーション・ライブラリに制御を渡して、エミュレーション・ライブラリが要求されたサービスを実行するサーバを呼び出す。現在、オペレーティング・システムの機能を複数のサーバで実現する作業が進められている。

上記のように、オペレーティング・システムのシステム機能自体をもできるだけ並列に処理される構成として、システム機能の処理性能の向上を図る試みが、現在盛んに行われている。国内においても、オペレーティング・システムのシステム機能の並列化の研究が始められている[Tsune92][Imamu92]。最近は、システム機能をできる限り核外に出し、利用者レベルのサーバにより実現する試みが主流となってきている。

本論文においても、オペレーティング・システムをプロセス・ネットワーク方式で実現し、これをマルチプロセッサ・システムに分散配置することにより、システム機能の並列化を行う。本研究では、システム機能を実現するプロセス・ネットワークはカーネル・レベルに置き、システム・コール時の応答時間を向上させるための並列性の抽出について考察し、プロセス・ネットワークを各プロセッサに分散配置するための指針を与え、この指針による分散配置が適切なものであることを実験により検証する。

1.3 本論文の構成

第2章において、高並列の共有メモリ型マルチプロセッサ・システムを対象として、アクティビティ方式の並列実行管理方式について述べる。第3章では、アクティビティ方式において、親タスクが再帰的に子タスクを生成していき、それぞれの親タスクがそのすべての子タスクを待って後処理を実行する場合の効率を向上するための、「遺言」とよぶ新しいコンストラクトについて述べる。第4章では、複数の応用プログラムが同時に動作しており、かつ、各応用プログラムがアクティビティ方式を用いて細粒度の並列処理を行う場合について検討する。第5章では、メッセージ通信型マルチプロセッサ・システムを対象として、プロセス・ネットワーク方式によるオペレーティング・システムの構成法について述べる。第6章では、プロセス・ネットワークのプログラム解析により、オペレーティング・システムのシステム機能自体の並列性の抽出を試みる。第7章はまとめと、今後の課題について述べる。

第 2 章

細粒度高並列プログラムのための並列実行機構

2.1 はじめに

汎用高並列計算機を用いて非常に多数の細粒度のタスクを並列に実行させる場合、効率の高い並列実行管理機構をシステム・ソフトウェアとして提供することが重要な課題となる。

1.2で述べたように、並列処理の管理方式としては並行プロセスに基づく方式が広く採用されている。これは、並列に実行すべき処理が発生するとプロセスを生成し、その実行が完了するとプロセスを消滅させるものである。しかしながら、並列実行の要求があるごとにUNIXで使用されているようなプロセスを生成してタスクを実行させる方式では、あまりにオーバーヘッドが大きく、多数のタスクを発生させて処理を行う場合には決して効率的ではない。

最近では、より軽量のスレッド、軽量プロセスなどとよばれるものを用いる方式も提案されている。しかし、軽量プロセスを用いる場合でも、その生成と消滅にはかなりのプロセッサ時間を消費する。また、スタック領域を割り当てることによるメモリ資源の消費も無視し得ない問題となる。

そこで、共有メモリ型のマルチプロセッサ・システムを対象として、あらかじめプロセッサの台数と同数の軽量プロセスを用意しておき、これらを繰り返し使用するこ

とにより、無用の軽量プロセスの生成を防止することをめざす。このようにして、効率の高い並列実行管理機構を実現するという基本方式を、田胡とともに提案した[Tago89b]。本方式をアクティビティ方式とよぶ。

アクティビティ方式では、並列に実行すべきタスクが発生するごとに軽量プロセスを生成し、タスクの実行が完了するとこれを消滅させるということはしない。並列に実行すべきタスクが発生するとこの要求をキューに接続しておき、一方、軽量プロセスには1個のタスクの実行が完了するごとにキューから要求を1個取ってきては実行させる方式である。

このアクティビティ方式は、どのような形式の並列プログラムにも適用できる汎用の並列実行管理機構を提供するものである。これまでに、本方式の具体的なインプリメントと性能評価も行われている[Tago91]。タスクの実行中にサスペンドがまったく発生しない理想的な場合には、新しい軽量プロセスを生成する必要がないので、高い効率を実現できる。

2.2 アクティビティ方式の基本原理

2.2.1 従来の並行プロセス方式

1.2で述べたように、従来のUNIXではforkシステム・コールを用いて、応用プログラムが並列に実行すべきタスクが発生すると同時にその実行のためのプロセス生成を要求する。生成されたプロセスはレディ・キューに接続されてスケジューリングの対象となり、つぎつぎとプロセッサを割り当てられて実行される。この方式には、

(1) 並列実行の単位となるプロセスを、ハードウェアに依存せずにプロセッサ台数とは独立に、任意に生成できる、

(2) 並列に実行すべきタスクのスケジューリングを利用者プログラムから隠蔽することができる、

などの利点がある。

しかしながら、並列に実行すべきタスクが発生するごとにプロセス実体を生成し、その実行が完了するとそのプロセス実体を消滅させる方式は、あまりにオーバーヘッドが大きい。プロセス実体として、生成・消滅ができるだけ効率よく行われるように工夫された軽量プロセスを用いる方式でも、なお、その消費するプロセッサ時間とメモリ空間の量は大きい。すなわち、

(1) プロセスの生成時には、スタック領域の割り当てとその初期化、プロセス制御ブロック（PCB）の初期化とそのレディ・キューへのリンク、

(2) 実行開始時には、プロセスの走行開始のための管理情報の操作とコンテクスト・スイッチ、

(3) 実行完了時には、プロセス消滅の操作、

などが必要であり、さらに、

(4) プロセス管理のためのシステム処理は直列化されるため、多数の要求が発生するとボトルネックとなりやすい、

ので、並列に実行すべきタスクが非常に多数で、細粒度である場合には大きなオーバーヘッドとなり、効率的ではない。

2.2.2 アクティビティ方式

上で述べた利点を活かしつつ、オーバーヘッドが小さくなる並列実行管理機構として、本研究ではアクティビティ方式を提案する。アクティビティ方式では、並列に実行すべきタスクの発生と、タスクの実行機構の用意を別個に取扱う。並列実行可能なタスクが発生すると同時にその実行のための軽量プロセスを生成することはしない。

応用プログラムでは、並列に実行させるタスクが発生すると、

```
make_child(手続き, 引数)
```

というシステム・コールを用いて、管理機構に並列実行の要求を出す。この要求に対して管理機構はすぐに実行のための軽量プロセスを生成することはせず、要求の形のままで保持する。これをアクティビティとよぶ。応用プログラムがつぎつぎとタスクを発生すると、図2.1に示すように、管理機構はアクティビティを1本のキューに結合して保持する。これをアクティビティ・キューとよぶ。

一方、アクティビティの実行機構としてはプロセッサ数と同数の軽量プロセスをあらかじめ用意しておく。軽量プロセスは、未実行のアクティビティをアクティビティ・キューから1個取ってきて実行する。この軽量プロセスがそのアクティビティの実行を完了すると、また次のアクティビティをアクティビティ・キューから取ってくる。以下同様の動作を繰り返す。このように、あらかじめ用意した軽量プロセスを繰り返し利用するので、軽量プロセスの生成・消滅に必要なプロセッサ時間を節減できる。また、スタック領域として割り当てられるメモリ消費量を節減できる。

アクティビティは、実行すべき手続きとその引数から構成されているので、1個あたりのメモリ消費量は小さい。また管理機構が実行しなければならない処理はアクティビティのキューへの結合とキューからの取り出しだけであり、処理量は小さい。システム処理のボトルネックが緩和できる。

図2.2に示すように、軽量プロセスの内部状態はレジスタとスタックとで表現される。1個のアクティビティの実行が終了すると、この内部状態はすべて不要になるの

で、軽量プロセスはスタック・ポインタの初期化だけで新しいアクティビティの実行を開始できる。

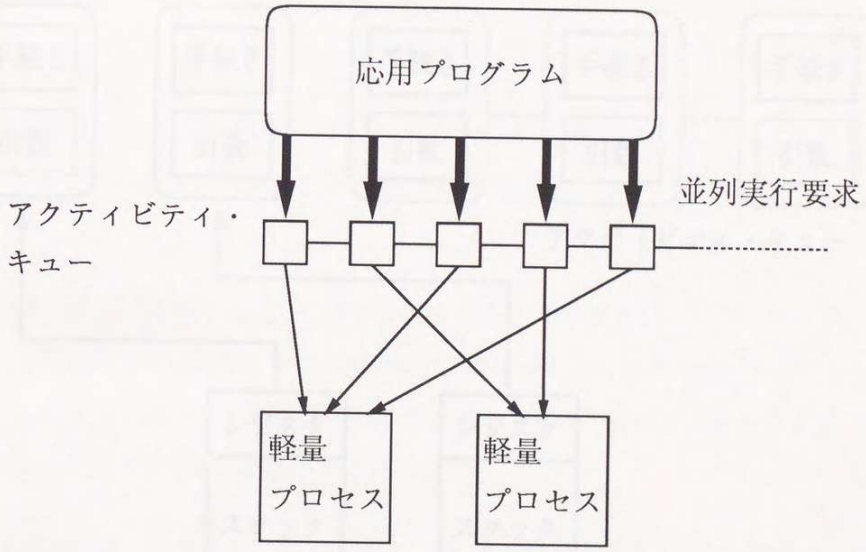


図 2.1: アクティビティ方式による並列処理機構

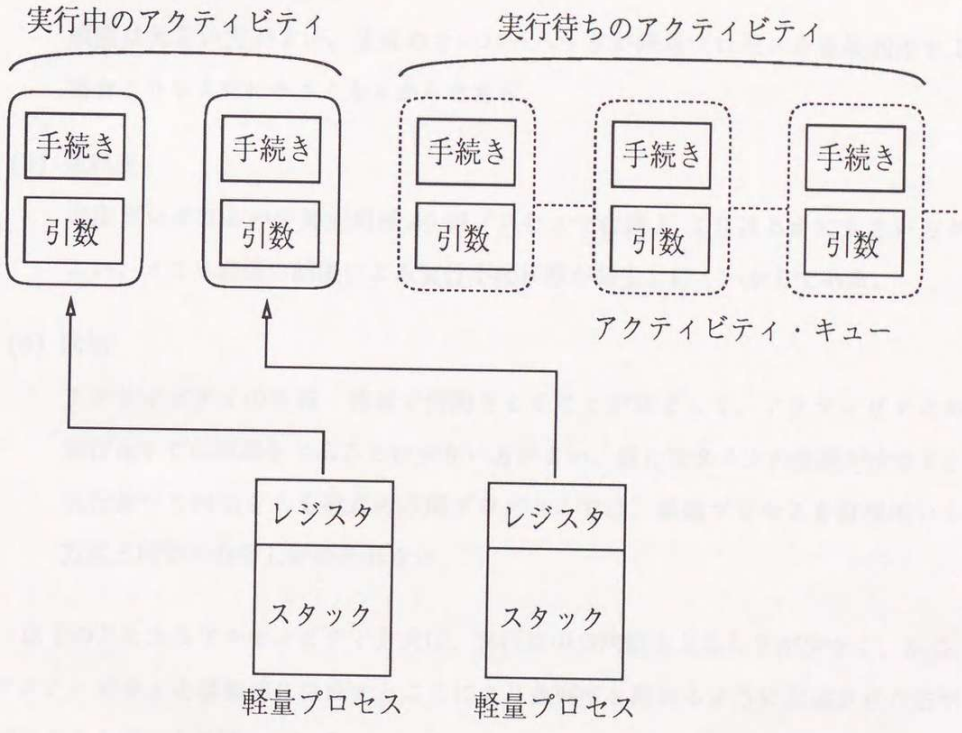


図 2.2: アクティビティとその実行機構

2.2.3 アクティビティ方式の適用対象

アクティビティ方式は、論理的にはプロセス、軽量プロセスを直接利用する方式と同一の環境を実現することができるので汎用性があるが、よりメリットが大きくなるのは次の場合である。

(1) タスクの生成要求の頻度

頻度は大きい方がよい。生成のオーバーヘッドが軽量プロセスを直接利用する場合よりも大幅に小さくなるからである。

(2) 並列度

応用プログラムの平均並列度 M がプロセッサ台数 N よりはるかに大きい方がよい。メモリ資源の制限による実行不能状態が発生しにくいからである。

(3) 同期

アクティビティの生成・消滅で同期をとることが望ましく、アクティビティの実行途中では同期をとることが少ない方がよい。新たなタスクの生成が少なく、実行途中で同期をとる形式の応用プログラムでは、軽量プロセスを直接用いる方式と同等の効率しか得られない。

以上のことからアクティビティ方式は、実行途中で同期をとることが少なく、かつ、アクティビティを積極的に生成することにより並列度を高めるように記述された応用プログラムが、より適している。

2.3 アクティビティ方式による処理速度の向上

前節に述べた基本原理のみからなるアクティビティ方式については、これまでにシステムのインプリメント、および、性能評価が行われている [Higa90] [Tago91]。

具体的には、2台の68020プロセッサからなる共有メモリ型のマルチプロセッサ・システム上において、500,000個のランダム・データをクイック・ソートする応用プログラムにより性能評価を行った。

図2.3に、クイック・ソート・プログラムの一例を示す。qsort_list は、引数 orgp の指すリストのサイズがしきい値 THRESHOLD より大きければ、関数 div_list によって、リストをキー値より大きい部分と小さい部分の2つのエレメントに分割し、make_child を用いてそれぞれのエレメントについて並列に qsort_list を適用する。しきい値より小さければ、単純なバイナリ・ソート bsort_list によってソートを行う。したがって、qsort_list から bsort_list に移行することを決めるしきい値を変えることにより、全体として実行するソートの総数は一定にしつつ、要求されるタスクの数を制御するできる。軽量プロセスを直接利用する場合には、make_child の代わりに、軽量プロセス生成のシステム・コールを用いる。

なお、この性能評価は、純粋にタスクの生成／消滅時のオーバーヘッドを測定することを目的とするので、軽量プロセスが同期のためにサスペンドすることが起こらないように応用プログラムを工夫した[1]。すなわち、親タスクが子タスクを生成した場合においても子タスクの完了待ち合わせをしないようにプログラムを記述してある。

2台のプロセッサを用いて、500,000個のランダム・データをソートするのに要した時間を図2.4に示す。アクティビティ方式が軽量プロセス方式よりも、つねに実行時間が短い。

[1]軽量プロセスが同期のためサスペンドする場合については次節で述べるが、この場合には新しい軽量プロセスを生成するために、その軽量プロセス生成のコストも必要となる。アクティビティの生成／消滅時のオーバーヘッドのみの測定をすることができない。

```
typedef struct
{       int head;
        int tail;
} LIST;
#define listsize(listp) (listp->tail - listp->head)

qsort_list(orgp)
LIST *orgp;
{
        LIST *upp,*lowp;

        if (listsize(orgp) > THRESHOLD)
        {       upp = (LIST *)mem_alloc(sizeof(LIST));
                lowp = (LIST *)mem_alloc(sizeof(LIST));
                div_list(orgp,upp,lowp);

                make_child(qsort_list,upp);
                make_child(qsort_list,lowp);
        }
        else
                bsort_list(orgp);
}
```

図 2.3: 並列クイック・ソート・プログラムの一例

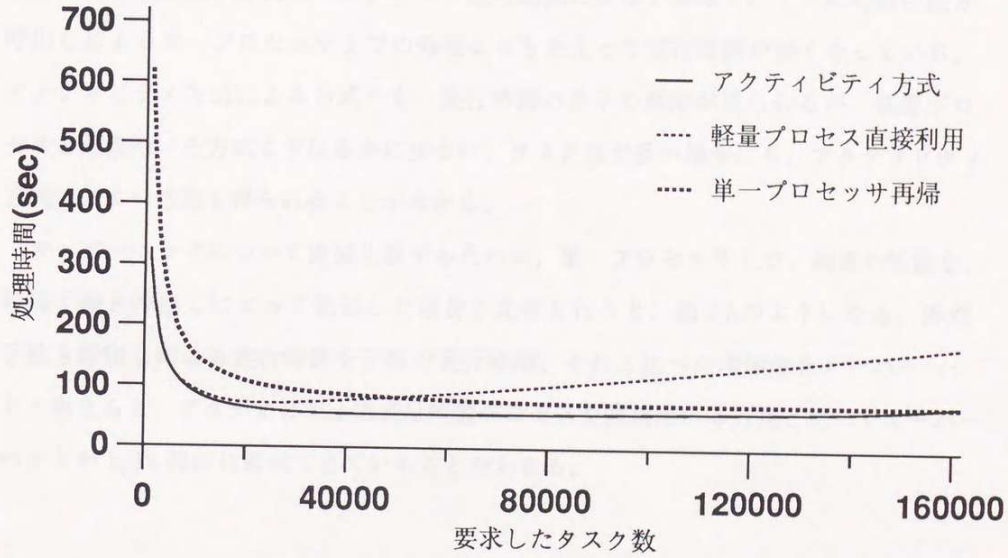


図 2.4: アクティビティ方式による処理速度の向上

(檜垣 [Higa90] より引用)

比較のために、単一のプロセッサ上で再帰手続き呼出しによって記述されたソート・プログラムの性能も示してある。しきい値が同じときに、再帰手続き呼出しの実行総数はタスク数に等しい。２台のプロセッサを用いた場合には、タスク数が少ない領域においては、実行時間が著しく短縮される。これは、並列処理によりソートの実行効率自体が改善されたことによる。これに対して、タスク数が増加するにしたがって実行時間の増加が見られる。これは、オーバーヘッドによるものである。とくに、軽量プロセスを直接用いる方式では、タスク数の比較的少ない領域で、すでに再帰手続き呼出しによる単一プロセッサ上での処理よりもかえって実行時間が長くなっている。アクティビティ方式による方式でも、実行時間の若干の増加が見られるが、軽量プロセスを直接用いる方式よりはるかに少ない。タスク数が多い場合にも、アクティビティ方式ではよい性能を得られることがわかる。

オーバーヘッドについて直接比較するために、単一プロセッサ上で、両者の性能を、再帰手続き呼出しによって記述した場合と比較を行うと、図2.5のようになる。再帰手続き呼出しによる実行時間を正味の実行時間、それと比べた増加分をオーバーヘッドと考えると、アクティビティ方式は軽量プロセスを直接用いる方式と比べてオーバーヘッドが1/15程度に節減できていることがわかる。

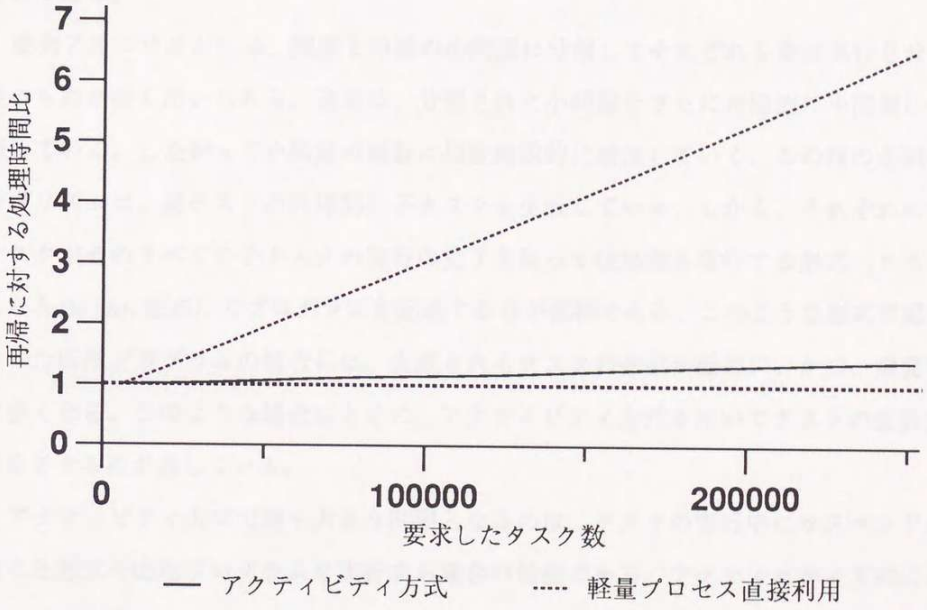


図 2.5: オーバーヘッドの比較

(檜垣 [Higa90] より引用)

2.4 アクティビティ方式の改良点

前節で述べたように、アクティビティ方式を用いることにより軽量プロセスを直接用いるよりも性能の向上が図れることが、実験により明らかになった。アクティビティの生成・消滅のためのオーバーヘッドはきわめて小さいので、**2.2.3**でも述べた通り、非常に多数の細粒度のタスクを並列に実行させる場合に、より高い効率を得ることが可能である。

並列アルゴリズムには、問題を同種の小問題に分割してそれぞれを並列実行させる型のものが多く用いられる。通常は、分割された小問題をさらに再帰的に小問題に分割していく。したがって小問題の総数は指数関数的に増加していく。この種の並列アルゴリズムは、親タスクが再帰的に子タスクを生成していき、しかも、それぞれの親タスクがそのすべての子タスクの実行の完了を待って後処理を実行する形式（ネストしたfork-join形式）でプログラムを記述するのが便利である。このような形式で記述された応用プログラムの場合には、生成されるタスクの総数が動的に、かつ、爆発的に多くなる。このような場合にとくに、アクティビティ方式を用いてタスクの並列実行をさせるのが適している。

アクティビティ方式で最も大きな問題となるのは、タスクの実行中にサスペンドが起こる形式の応用プログラムを実行する場合の性能である。アクティビティ方式においても、軽量プロセスが同期のためにサスペンドすることが起こり得る。この場合、プロセッサを無駄なく利用するためには、新しい軽量プロセスを生成して実行を開始させなければならない。

新たな軽量プロセスを生成・消滅させることに要するプロセッサ時間は小さくない。また、スタック領域割り当てのためのメモリ消費も無視できない。

上でアクティビティ方式に適していると述べた、ネストしたfork-join形式の並列プログラムの場合でも、親タスクによる子タスクの完了待ち合わせにより多数のサスペンドが発生し、多数の軽量プロセスを生成しなければならなくなる。

例えば、7都市巡回セールスマン問題をネストしたfork-join形式の並列プログラムを用いて解く場合、応用プログラムは、図2.6に示すように、まず6個の子タスクを作り、そのそれぞれが5個の子タスクを作り、と繰り返すものとなり、全体で1237個のタスクが作られる。そのうち517個のタスクが実行途中でサスペンドするので、517個の軽量プロセスの生成が必要となる。いま、プロセッサ台数を8台と仮定すると、実行途中でサスペンドするタスクがない場合と比べて約65倍の個数の軽量プロセスを生成することになる。メモリの消費もこれに比例して増大する。

この場合においても、軽量プロセスを直接利用する方式と同程度の性能は保証されるが、アクティビティ方式をより有効に利用するためには、子タスク待ちを行う形式の応用プログラムの実行効率をも向上させることが必要である。上記の効率の低下を防ぐためには、

(1) 子タスク待ちにおいても軽量プロセス生成を行わないですむように工夫をする、

(2) 子タスク待ちにおいての軽量プロセス生成／消滅のコストをできるだけ小さなものにする、

という改良が考えられる。

(2) の、軽量プロセス生成のコストを軽減させる改良について、軽量プロセスの生成時に新たなスタック領域を確保するのではなく、親タスクを実行していた軽量プロセスのスタックの上に積む形でスタックを割り当てる方式を試みた[Koba.K92]。このメカニズムの概略を図2.7に示す。しかしながら、ややプロセッサ時間の短縮が実現できたものの、軽量プロセス自体の再利用が効率的には行われないので、大きく効率の改善を図ることはできないという結果を得た。とくに、メモリ消費を減少させることはできない。

結論として、アクティビティ方式の利点を活かしつつ、子タスク待ちを行う形式の応用プログラムの実行効率をも向上させるためには、上記 **(1)** の、子タスク待ちによ

るサスペンドが発生した場合でも軽量プロセスの生成を行わないですむようにする何らかの工夫が必要である。

次章では、基本のアクティビティ方式に「遺言」とよぶ新しいコンストラクトを追加する方式を提案する。これは、子タスクの完了を待って行われる後処理は自分では実行せず子タスクに「遺言」して、最後に処理を終える子タスクに割り当てられていた軽量プロセスがこの「遺言」を実行する方式である。親タスクは「遺言」を伝達すると実行を完了するので、サスペンドが発生しなくなり、子タスク待ちにおける無用の軽量プロセス生成を防ぐことが可能となる。この改良方式の具体的な実行機構の設計およびインプリメントを行い、シミュレーションによる性能評価を行う。

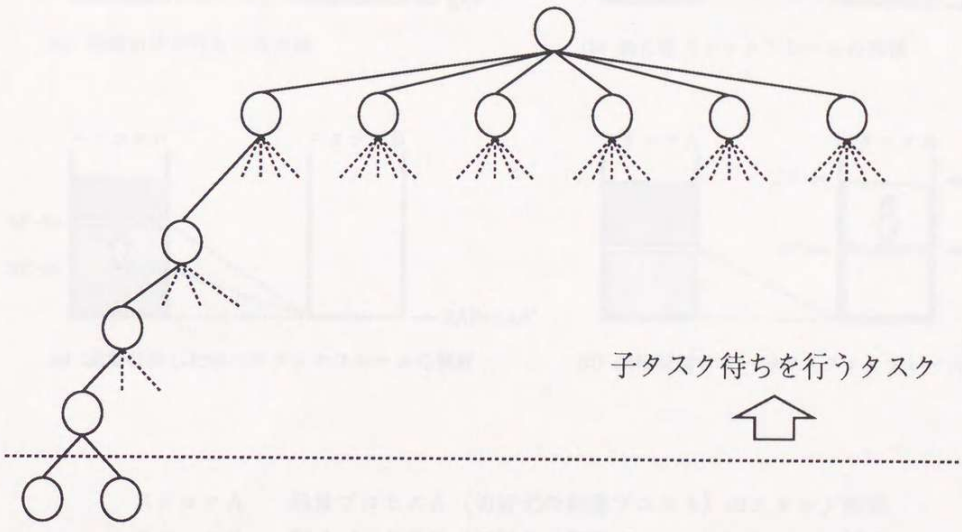


図 2.6: 7 都市巡回セールスマン問題におけるタスクの実行要求

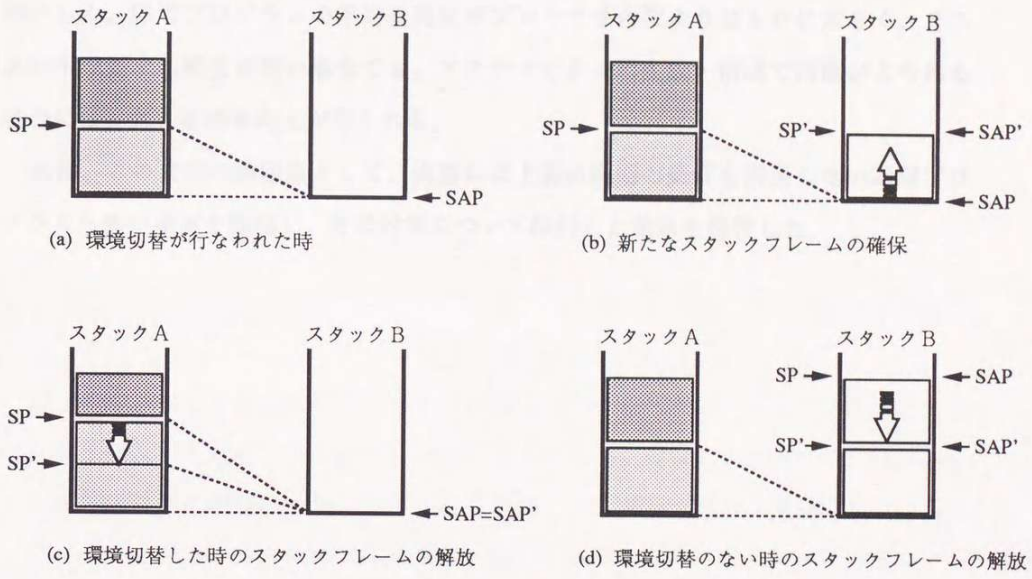


図 2.7: 軽量プロセス生成／消滅のコストを軽減するための試みの概略

2.5 結論

本章では、まず、高並列で細粒度のタスクを効率よく実行するための並列実行機構としてアクティビティ方式を提案し、この提案に基づいて行われた評価実験の結果を紹介した。応用プログラムの平均並列度がプロセッサ台数よりはるかに大きく、タスクの生成要求の頻度が高い場合でも、アクティビティの生成・消滅で同期がとられる場合には、大きな効率向上が得られる。

次に、この方式の問題点として、実際には上記の同期の条件を満足しない応用プログラムも多い事実を指摘し、その対策について検討した結果を報告した。

第 3 章

「遺言」コンストラクトを用いる性能の向上

3.1 はじめに

前章に述べたとおり、並列アルゴリズムには、問題を同種の小問題に分割してそれぞれを並列実行させる型のものがしばしば用いられる。通常は、分割された小問題をさらに再帰的に小問題に分割していく。したがって小問題の総数は指数関数的に増加していく。この種の並列アルゴリズムは、親タスクが再帰的に子タスクを生成していき、しかも、それぞれの親タスクがそのすべての子タスクの実行の完了を待って後処理を実行する形式（ネストした fork-join 形式）でプログラムを記述するのが便利である。この形式のプログラムの場合、親タスクによる子タスクの完了待ち合わせにより多数のサスペンドが発生し、基本のアクティビティ方式のままでは、顕著な効率の向上が得られない。

上記の問題点は、田胡 [Tago91] がすでに指摘しているように、実行途中で同期によるサスペンドを発生しないように応用プログラムを作成すれば解決可能である。しかし、この方法では後処理を実行しないように応用プログラムを作成することが必要で、プログラム記述に大きな制限を加えることになる。

本章では、親タスクは後処理を「遺言」として子タスクに伝達して実行を完了し、最後に実行を完了した子タスクにその「遺言」を実行させる方式を提案する[Nakay92b][Nakay92c][Nakay93]。これは、後処理は自分では実行しないことにして「遺言」として子タスクに伝達し、この「遺言」は最後に処理を終える子タスクに割り当てられていた軽量プロセスに実行させる方式である。親タスクは「遺言」を伝達すると実行を完了するので、サスペンドが発生しなくなる。

この方式では、後処理を含むプログラムの記述が許されるので、プログラマに無用の負担をかける欠点が解消する。すなわち、基本のアクティビティ方式の利点を活かしつつ、ネストしたfork-join形式の並列プログラムをも効率よく実行することが可能となる。上記以外の形式の並列プログラムでも、基本方式とほぼ同一の効率で実行できる。

この「遺言」のコンストラクトを追加するために、新しい並列実行管理機構を設計した。以下、改良方式の具体的な実行機構について述べる。

3.2 改良方式の提案

3.2.1 「遺言」の作成

具体的には、まず、タスクの親子関係が後処理が完了するまで保持されることを積極的に利用し、図3.1に示す家系記述子（Family Tree Descriptor、以下、FTDと略記）とよぶ構造体を新たに導入して親子関係を記述する。親タスクが子タスクの完了を待って後処理を実行するしないにかかわらず、子タスクが作成される時点でFTDを用意して、親のFTDにポインタを用いてリンクする。これにより子タスクは自分の親が誰であるかを知ることが可能となる。また、親のFTDには現在生きている自分の子の数を格納しておく。子タスクが発生するごとにFTDを作る必要があるが、FTDは軽量プロセスに比べはるかに小さいメモリしか必要としない。

次に、親タスクが後処理を実行しなければならない場合には、後処理を手続きとその引数の形で宣言する。具体的には、図3.2に示すように、

```
make_will(後処理の手続き, 引数)
```

というシステム・コールを用いて宣言する。このシステム・コールがよばれると、FTDの「遺言」の領域に後処理の手続きとその引数の情報が格納され、同時に親タスクは「遺言」を遺したまま強制的に終了される。

親タスクを実行していた軽量プロセスはその内部状態を保持しておく必要がないので、そのまま次のタスクの実行を開始することができる。

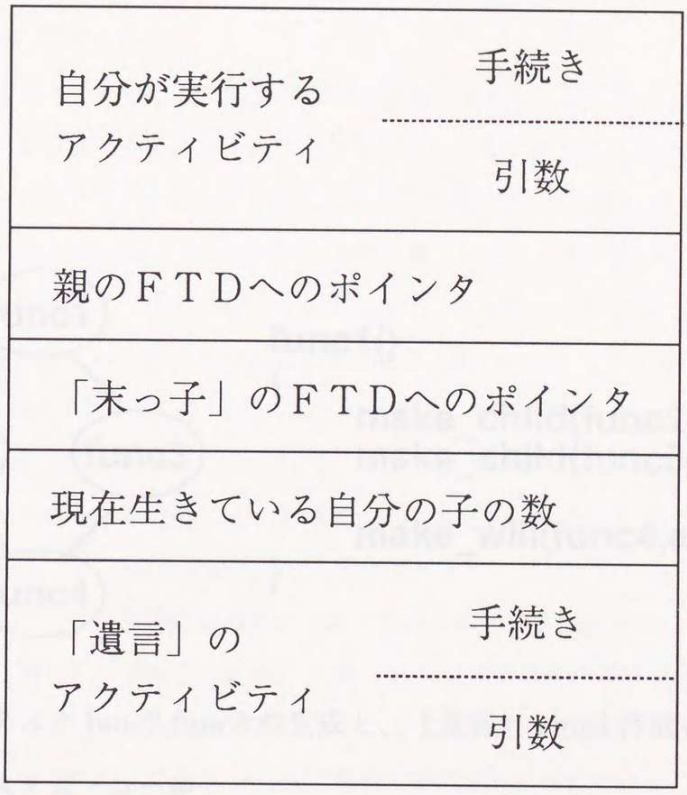
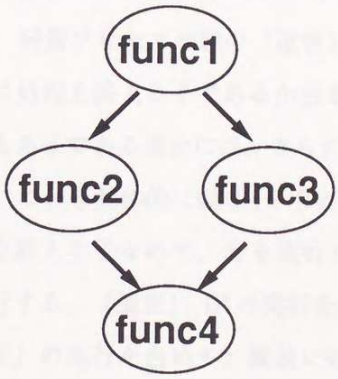


図 3.1: 家系記述子（Family Tree Descriptor）の構造




```
func1()
{
        make_child(func2,arg2);
        make_child(func3,arg3);

        make_will(func4,arg4);
}
```

図 3.2: 子タスク func2,func3 の生成と、「遺言」 func4 作成の
システム・コール

3.2.2 「遺言」の実行

作成した「遺言」は1つのアクティビティであるが、作成と同時に実行可能状態にあるのではなく、子タスクのすべてが実行を完了したあと実行すべきものである（起動条件付きのアクティビティ）。したがって、子タスクのうち最後に処理を終えたものを実行していた軽量プロセスに、終了後ただちに実行させる方式とするのが、全処理時間の短縮に有利である。

そこで、子タスクを実行した軽量プロセスには処理の終了後、親タスクのFTDを参照して、自分が実行していたタスクが親タスクから見て最後に処理を終えた子に当たるか否かを判定させる。自分が実行していたタスクが最後に処理を終えた子に当たる場合には、図3.3に示すように親のタスクの「遺言」をただちに実行する。

軽量プロセスが親の「遺言」の実行を終えると、その親が「親の親」から見て最後に処理を終える子であるか否かを判定する。親が「親の親」から見て最後に処理を終える子である場合には、さらに「親の親」の「遺言」を実行する。

これを具体的に図3.4を用いて説明する。タスクEはタスクBから見て最後に処理を終える子なので、Eを実行していた軽量プロセスがBの「遺言」B'をただちに実行する。「遺言」B'の実行を終えると、その親タスクであるAから見てBが（「遺言」の実行を含めて）最後に処理を終えた子であるか否かを判定する。図3.4の場合では、Cの「遺言」が未完了のため、BはAから見て最後に処理を終えた子ではないので、Aの「遺言」A'はまだ実行しない。そこで、Eを実行していた軽量プロセスは解放される。次に、GはCから見て最後に処理を終える子なので、Gを実行していた軽量プロセスはCの「遺言」C'をただちに実行する。「遺言」C'の実行を終えると、その親タスクであるAから見てCが最後に処理を終えた子であるか否かを判定する。C'が終了した時点でB'の処理がすでに終了しているので、CはAから見て最後に処理を終えた子である。そこでC'の実行に続いてA'もただちに実行する。この時点で、Gを実行していた軽量プロセス以外の軽量プロセスはすべて解放されて

いる。

以上のように、先祖をたどって「遺言」を実行すべきであるかを判定する。ただちに実行すべき「遺言」がないとき、軽量プロセスはアクティビティ・キューからタスクを取ることを行う。

もちろん、「遺言」の中でも子タスクの生成や「遺言」の作成を行うことができる。「遺言」の実行が始まるときにFTDの「遺言」の情報は、図3.1における「自分の実行するアクティビティ」に移され、そのFTDを用いて実行が行われる。すなわち、元の親タスクの延長として「遺言」が実行される。したがって、「遺言」実行中に生成された子タスクの親は、元の親タスクである。

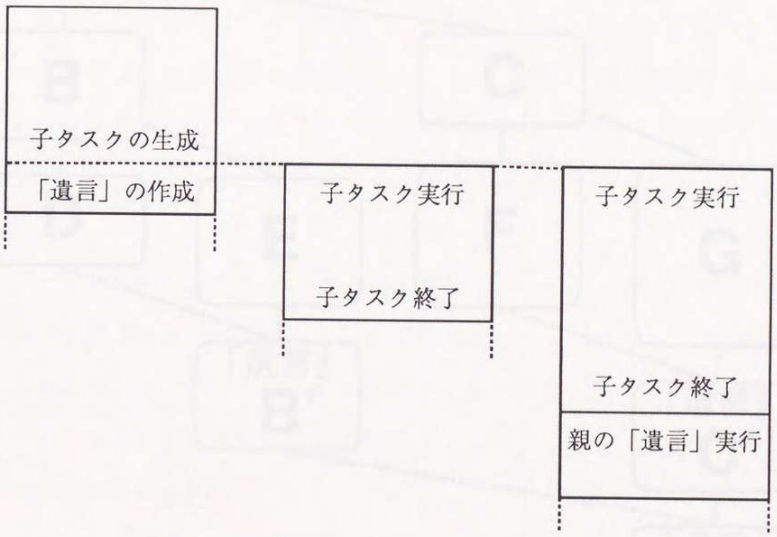


図 3.3: 「遺言」の作成とその実行のタイミング

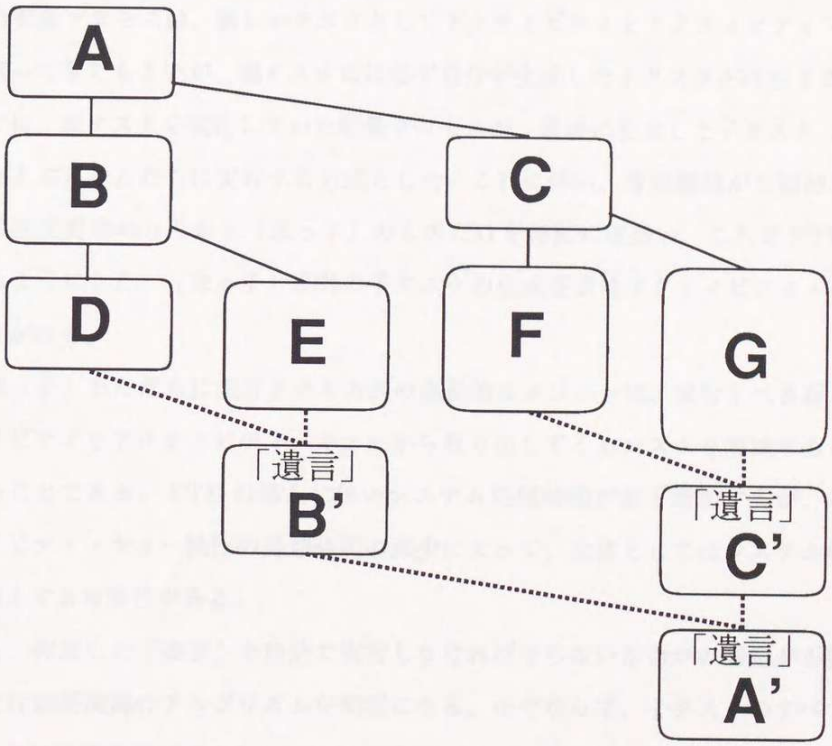


図 3.4: 「親の親」の「遺言」の実行

3.2.3 「末っ子」への軽量プロセスの譲り渡し

前節に述べた機構により、親タスクは後処理を「遺言」の形で残し、最後に処理を終えた子タスクを実行していた軽量プロセスにより最優先で実行してもらうことができる。また、親タスクを実行していた軽量プロセスは後処理を宣言した時点で実行を完了するので、新しいタスクの実行開始が可能になる。

その軽量プロセスは、新しいタスクとしてアクティビティをアクティビティ・キューから取ってきてもよいが、親タスクには必ず自分が生成した子タスクが存在するから、ここでは、親タスクを実行していた軽量プロセスが、最後に生成した子タスク（「末っ子」とよぶ）をただちに実行する方式とした。これに伴い、管理機構が自動的に子タスクの生成要求のうちから「末っ子」のものだけを特別に取扱い、これをFTDに格納するようにした。「末っ子」以外の子タスクの生成要求はアクティビティ・キューにつながれる。

「末っ子」をただちに実行させる方式の直接的なメリットは、実行すべき新たなアクティビティをアクティビティ・キューから取り出してくるコストを削減することができることである。FTDの導入に伴いシステム処理時間が若干増加するが、このアクティビティ・キュー操作の処理時間の減少によって、全体としてはシステム処理性能が向上する可能性がある。

また、作成した「遺言」を自分で実行しなければならないか否かの判定が不要になり、実行管理機構のアルゴリズムが簡潔になる。なぜならば、子タスクのすべてが自分より先に処理を完了するということが絶対に起こらないからである。

タスク実行のスケジューリングという観点から見ると、「末っ子」をただちに実行することにより、自分の子孫の処理が他の処理よりも優先して処理されるという効果が発生する。自分の子孫の枝のうち常に1本は優遇され、待たされることなく処理が実行され、葉に至るまでこれが続く。これにより、応用プログラムによっては処理時間が大きく短縮される可能性がある。

以上に述べた並列実行管理機構の構成を図3.5に示す。基本のアクティビティ方式と比べて make_will という「遺言」作成のシステム・コールが加わり、また、処理要求の保持機構に FTD が加わっている。

管理機構の実現例を図3.6のフローチャートに示す。ここではこのフローチャートの詳しい説明は省略するが、タスクを実行している間に子タスクを生成したか否かは、「末っ子」がいるかいないかで判定している。タスクのツリーにおいて、葉に当たるタスクの場合には「末っ子」はいないことになる。また、親の家系記述子を参照している部分は、先祖の「遺言」を実行すべきか否かを判定するためのものである。したがって、ネストした親子関係のある場合には、先祖にさかのぼる順序でそれぞれの「遺言」が実行される。最後に、このフローチャートは子タスクを生成することなしに「遺言」を作成することが起こった場合も考慮に入れて設計している。この場合には「遺言」は同じ軽量プロセスによってただちに実行される。

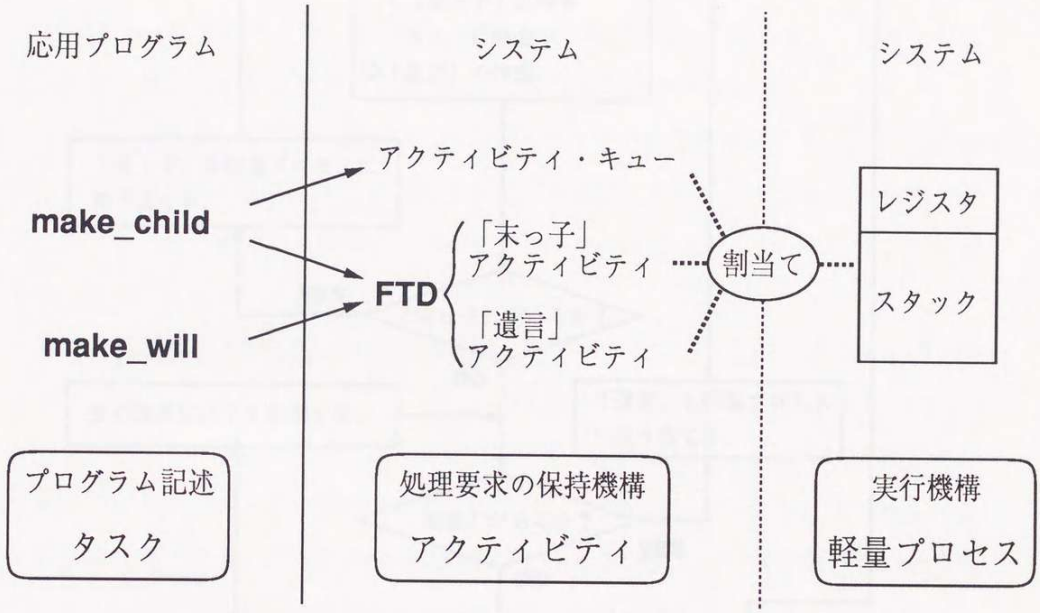

応用プログラム
システム
システム
アクティビティ・キュー
make_child
FTD
「末っ子」
アクティビティ
「遺言」
アクティビティ
make_will
割当て
レジスタ
スタック
プログラム記述
タスク
処理要求の保持機構
アクティビティ
実行機構
軽量プロセス

図 3.5: 改良方式の構成

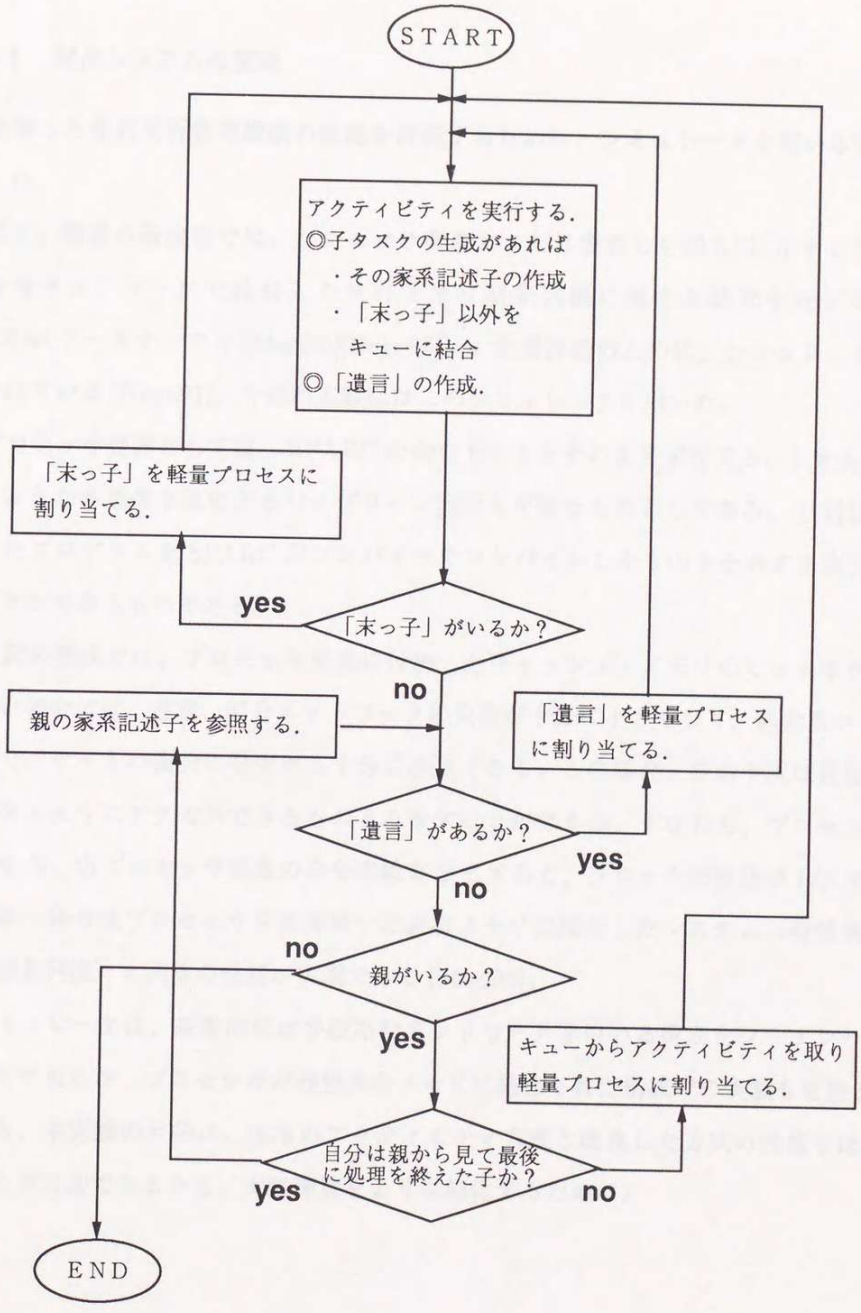


図 3.6: 改良方式における並列実行管理機構のフローチャート

3.3 実験

3.3.1 試作システムの実現

提案した並列実行管理機構の性能を評価するために、シミュレータを用いる実験を行った。

現在、筆者の研究室では、プロセッサ要素とメモリ要素とを図3.7に示すような多段結合ネットワークで結合した共有メモリ型並列機に関する研究を行っている（$\mathcal{PSM}$ アーキテクチャ [Mori90][Degu90]）。性能評価のために、シミュレータが試作されている [Naga91]。今回の実験にはこのシミュレータを用いた。

プロセッサ要素としては、SPARC の命令セットをそのまま実行でき、しかも図3.8に示すような多重命令流によるパイプライン実行も可能なものとしてある。C 言語で記述したプログラムを SPARC 用コンパイラでコンパイルしたものをそのまま実行させることができるものである。

上記の構成では、プロセッサ要素に付加したキャッシュ・メモリのヒット率が十分に高い場合には、通常、結合ネットワークの負荷が十分に小さくなり、結合ネットワーク内でパケットの衝突の発生率も十分に小さくなる。この場合、各命令流は遅延なしで共有メモリにアクセスできるものとみなすことができる。すなわち、プロセッサ要素数を N、各プロセッサ要素の命令流数を S とすると、クロック周波数が $1/S$ の NS 個の単一命令流プロセッサを直接単一の共有メモリに結合したシステム（理想共有メモリ型並列機）と同等の性能が実現できる [Mori90]。

シミュレータは、基本的には多段結合ネットワークを用いる構成をシミュレートするものであるが、プロセッサが理想共有メモリに結合された構成での実験も可能となっている。本実験の目的は、基本のアクティビティ方式と改良した方式の性能を比較することが目的であるから、実験環境をより単純化するために、

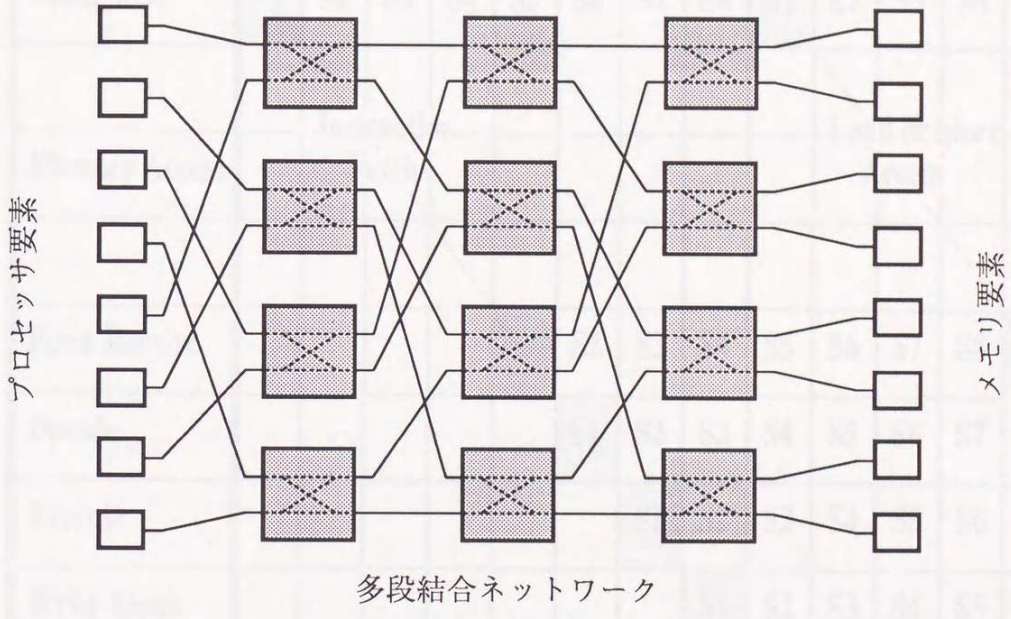


図 3.7: *PSM* アーキテクチャ

(多段結合ネットワークによる共有メモリ)

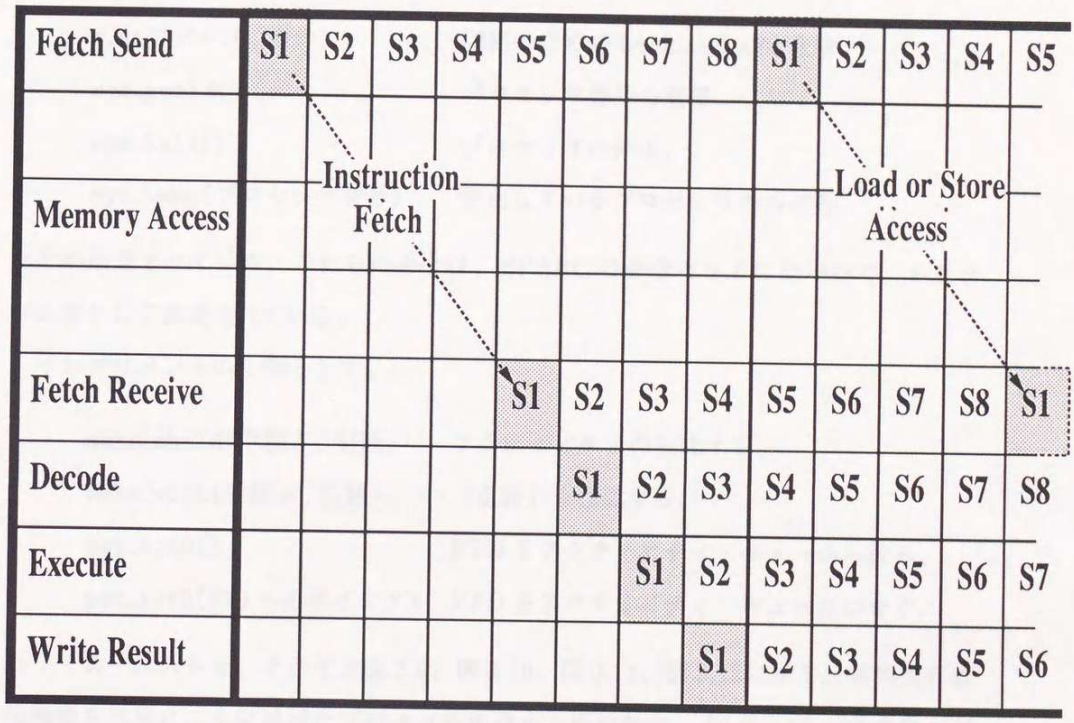


図 3.8: $\mathcal{PSM}$ アーキテクチャ

(多重命令流プロセッサのパイプライン実行)

- 単一命令流プロセッサ、
- 直接理想共有メモリに結合した構成、

について実験することにした。プロセッサの台数は8台とした。

実験に用いたシミュレータでは、並列処理のためのプリミティブとして、

f_a(アドレス, 増分)	同期のための fetch_and_add 命令、
sys_getid()	プロセッサ番号の取得、
sys_halt()	プロセッサの停止、
sys_wake(プロセッサ番号)	停止しているプロセッサの再起動、

などが用意されている。これらの命令は、SPARC の命令セットにないので、トラップ処理として実現されている。

インプリメントの1例として、

make_child(手続き, 引数)	アクティビティの生成する、
make_will(手続き, 引数)	「遺言」を作成する、
get_ActQ()	FTD をアクティビティ・キューから取る、
put_ActQ(FTD へのポインタ)	FTD をアクティビティ・キューにつなぐ、

のソース・リストを、それぞれ図3.9、図3.10、図3.11、図3.12に示す。並列実行管理機構を実現に、上に述べたプロセッサの停止・再起動や、fetch_and_add 命令などの、シミュレータに用意されている機構を利用している。

```
struct FTD                                          /* FTD の構造体の宣言 */
{       int (*func)();                              /*  自分が実行する  手続き */
        int para;                                   /*  アクティビティ   引数 */
        struct FTD *parent;                         /*  親の FTD へのポインタ */
        struct FTD *youngest_child;                 /*  「末っ子」の FTD へのポインタ */
        int nchild;                                 /*  現在生きている自分の子の数 */
        int (*will_func)();                         /*  「遺言」の       手続き */
        int will_para;                              /*  アクティビティ   引数 */
};

struct FTD *new_ftdp;

make_child(func,para)
int (*func)();
int para;
{
        struct FTD *ftdp, *current;

        current = get_current_ftd();                /* 現在実行中の FTD へのポインタ */

        ftdp = (struct FTD *)f_a(&new_ftdp,(sizeof (struct FTD)));
                                                    /* 新しい FTD を 1 つ確保する */
        ftdp->parent = current;
        ftdp->func = func;
        ftdp->para = para;

        f_a(&current->nchild,1);                    /* 現在の FTD の、子の数を 1 増やす */

        if (current->youngest_child != NULL)        /* 現在の FTD にすでに「末っ子」が */
                put_ActQ(current->youngest_child);  /* ある場合、それをアクティビティ */
                                                    /* ・キューにつなぐ */

        current->youngest_child = ftdp;             /* 暫定的に、これを「末っ子」にする */
        ftdp->nchild = -1;
}
```

図 3.9: make_child のソース・リスト

```
make_will(will_func,will_para)
int (*will_func)();
int will_para;
{
        struct FTD *current;

        current = get_current_ftd();            /* 現在実行中の FTD へのポインタ */

        current->will_func = will_func;         /* 「遺言」の手続き・ */
        current->will_para = will_para;         /* 引数を格納する */

        exit_usr();                             /* 「遺言」を遺して、強制的に終了 */
}
```

図 3.10: make_will のソース・リスト

```
struct ActQ                                     /* アクティビティ・キュー・エントリ
の宣言 */
{       struct FTD *ftdp;                       /*  FTD へのポインタ */
        int wake_me;                            /*  休眠中の軽量プロセスを再起動させ
る */
        int flag;                               /*  ときのために用いるフラグ */
};

struct ActQ *ActQ_for_lwp;

struct FTD *get_ActQ()
{
        struct ActQ *ActQp;

        ActQp = (struct ActQ *)f_a(&ActQ_for_lwp,sizeof (struct ActQ));
                                                /* キューから FTD を 1 つ取ってくる */
        ActQp->wake_me = sys_getid();

        if (f_a(&ActQp->flag,1) == 0)           /* 実行すべきアクティビティがない */
                sys_halt();                     /* 場合は、軽量プロセスは休眠 */
                                                /* put_ActQ により再起動される */
        return(ActQp->ftdp);                    /* FTD へのポインタ */
}
```

図 3.11: get_ActQ のソース・リスト

```
struct ActQ *ActQ_for_ftd;

put_ActQ(ftdp)
struct FTD *ftdp;
{
        struct ActQ *ActQp;

        ActQp = (struct ActQ *)f_a(&ActQ_for_ftd,sizeof (struct ActQ));
                                                /* 新しいキュー・エントリを確保する */
        ActQp->ftdp = ftdp;                     /* FTD へのポインタを格納 */

        if (f_a(&ActQp->flag,1) > 0)            /* 休眠している軽量プロセスに実行 */
                sys_wake(ActQp->wake_me);       /* させる場合には、それを再起動 */
}
```

図 3.12: put_ActQ のソース・リスト

3.3.2 実験内容

並列実行管理機構としては、

(S1) 改良方式のもの、

(S2) 改良方式において「末っ子」への軽量プロセスの譲り渡しを行わないもの、

(S3) 基本方式のもの、

をインプリメントして実験した。

実行させた応用プログラムは、最適探索問題のひとつである巡回セールスマン問題を解く並列プログラムを用意した。巡回セールスマン問題は、複数の都市の間の距離を与えられた場合に、全都市を訪れるための移動距離が最短になるような経路を求める問題である。本実験では都市数を 7 とした。

巡回セールスマン問題のストレートな解法としては、可能性の木を展開して全ての経路を調べる方式がある。全ての枝を展開すると末端では枝が膨大な数となるが、この方式では全処理量が一定となる。

これに対し、ツリーの各段階で現在知られている暫定解をしらべ、見込みのない探索は打ち切る枝刈りを行う方式もある。枝刈りを行う方式の場合、その判定に用いる暫定解は多くのプロセッサで共有され、更新される共有変数となるが、実際に更新の起こる回数は 7 都市巡回セールスマン問題の場合で 10 回以下であり、排他制御やアクセス競合によるオーバーヘッドは小さく問題とならない。

そこで、7 都市巡回セールスマン問題について、

- 全処理量が一定となるように、枝刈りをしない場合、
- 実用プログラムでは当然枝刈りを行うので、枝刈りをする場合、

の、両方の場合のプログラムを試した。

これらのプログラムにおいて、後処理については、

(P1) 後処理を含む自然な形に記述したもの、

(P2) 記述が不自然になるが後処理を含まない形に書き改めたもの、

の両者を用意した。

実験では、

A: S1 — P1 （改良方式 — 後処理あり）

B: S2 — P1 （改良方式「末っ子」なし — 後処理あり）

C: S3 — P1 （基本方式 — 後処理あり）

D: S3 — P2 （基本方式 — 後処理なし）

の 4 個の場合を評価した。

3.3.3 実験結果と考察

実行した処理時間の結果を図3.13、図3.14に示す。数字はシミュレータのクロック時間を示す。

まず、枝刈りをしない場合（図3.13）に、後処理を含む記述をしたものの中では実験A、実験B、実験Cの順によい性能が得られた。基本方式による実験Cに比べ実験Bはアクティビティ・キュー操作の時間には変化がなかったものの、その他のシステム処理時間が大きく減少した。これは子タスク待ちにおける軽量プロセス生成・消滅の時間が削減できたためと考えられる。

さらに、軽量プロセスを「末っ子」に譲り渡す実験Aではこれに加えてアクティビティ・キュー操作の時間も削減されることが確かめられた。改良方式の実験Aは、基本方式の実験Cに比べて処理時間が約10％減少した。

また、応用プログラムを後処理を含まないように書き改めて基本方式の並列実行管理機構を用いて動作させた実験Dとの比較においても、わずかながら実験Aに性能の向上が見られた。FTDの導入に伴うコストの増加の分はアクティビティ・キュー操作のコストの減少で十分に補うことができることが明らかになった。総じて応用プログラムの記述性がよい分、改良方式が優れていると判断できる。

次に枝刈りをする場合（図3.14）において、改良方式の実験Aは、基本方式の実験Cに比べて処理時間が約35％減少した。とくに、ユーザ処理時間が著しく減少した。これは、「末っ子」に軽量プロセスを譲り渡す方式としているため、自分の子孫を優遇して実行する深さ方向優先のスケジューリングが行われる結果、枝刈りの判定のよりよい基準が早く求まるためと考えられる。

最後に、メモリ消費の観点から改良方式と基本方式との比較を行った。その結果を表3.1に示す（割り当てたメモリ量ではなく、実際に消費したメモリ量を示す）。スタック消費量に関して、改良方式の実験Aは、基本方式の実験Cの約1/63の消費ですみ、大きくスタックの消費を節減できた。これは、およそ500個分の軽量プロセス

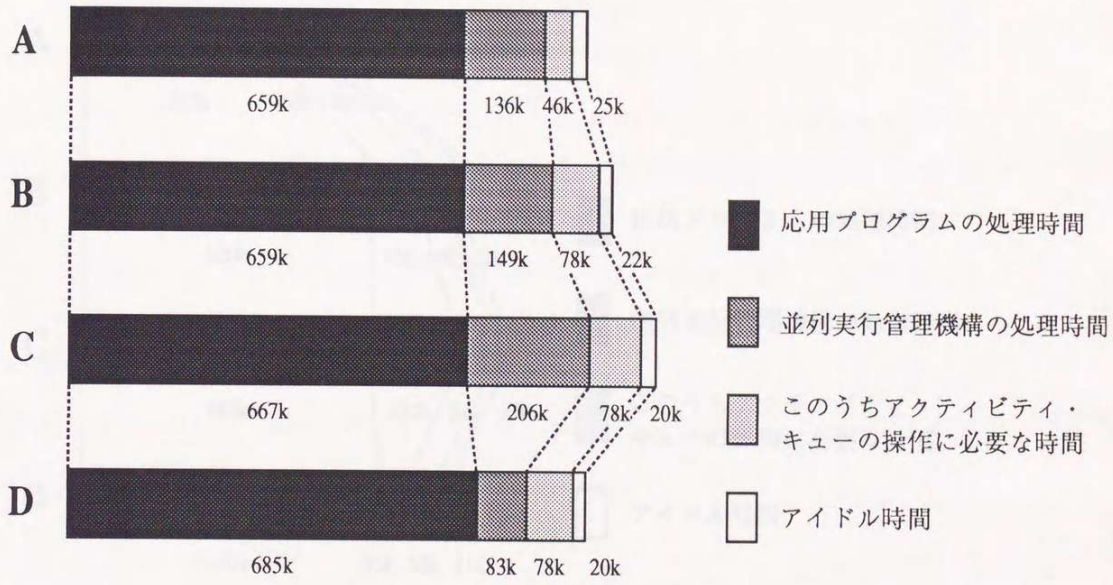


図 3.13: 7都市巡回セールスマン問題のシミュレーション結果

(枝刈りをしない場合)

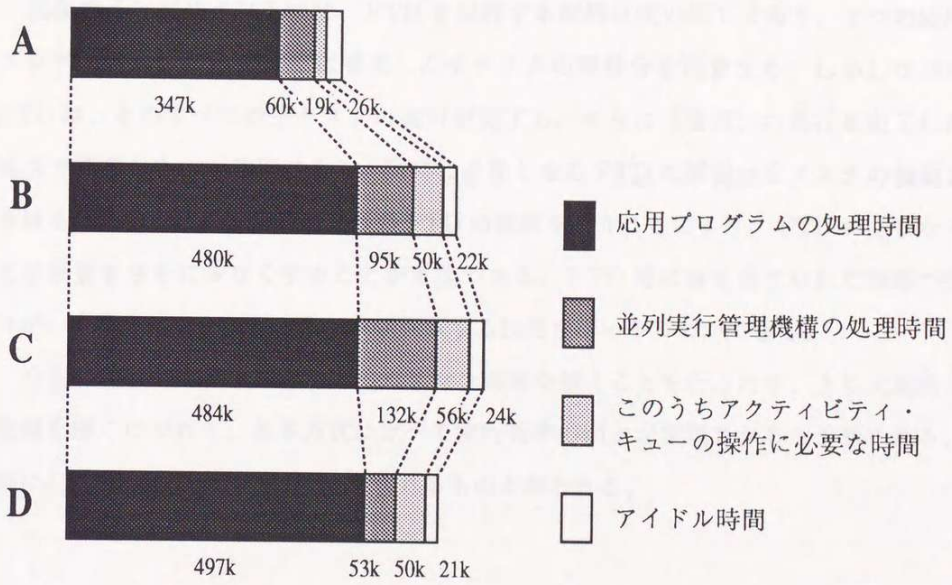


図 3.14: 7 都市巡回セールスマン問題のシミュレーション結果
(枝刈りをする場合)

の生成を節減したことに相当する。**2.4** で議論したように、基本方式では 7 都市巡回セールスマン問題において 517 個のサスペンドによる軽量プロセス生成が行われるが、改良方式ではこの軽量プロセス生成を節減できることが示された。アクティビティ・キューや FTD など、並列実行管理機構の管理のために必要とするメモリ量は、逆に提案方式は FTD を導入することでほぼ倍になったが、このコストに比べスタックの消費を節減できるメリットの方がはるかに大きい。

現在のインプリメントでは、FTD を保持する配列は使い捨てであり、1 つの応用プログラムが完了するまでに要求した全タスクの個数分を消費する。しかしながら FTD は、そのすべての子タスクの実行が完了し、さらに「遺言」の実行も完了した時点で不要となり再利用できる。同時に必要となる FTD の個数は全タスクの個数よりはるかに少ないので、使用済みの FTD の回収を行うことにより、FTD に必要なメモリ消費をさらに少なくすることが可能である。FTD 用に割り当てられた領域の残りが一定量以下となったときに一括回収する技法を用いることができる。

今回の実験では 7 都市巡回セールスマン問題を解くことを行ったが、より大規模な問題を解くにつれて、基本方式に比べて実行効率の向上が実現されると予想される。特に、メモリ削減の効果は顕著に現れるものと思われる。

表 3.1: 7 都市巡回セールスマン問題におけるメモリ消費

(単位は bytes、割り当てたメモリ量でなく実際に消費したメモリ量)

	スタック	アクティビティ・キュー + FTD
実験A（改良方式）	**1664**	**48328**
実験C（基本方式）	**105240**	**24032**

3.4 結論

基本のアクティビティ方式においては、ネストしたfork-join形式の並列プログラムを実行する場合、後処理実行の同期のためにサスペンドが多数発生し、効率がさして向上しないという問題点があった。本章では、後処理を「遺言」として子タスクに伝達して実行させる方式を提案し、この方式に基づく並列実行管理機構を試作し、シミュレーション実験を行った結果について述べた。

アクティビティ方式に「遺言」というコンストラクトを導入することにより、後処理を含む応用プログラムの場合でも、無用の軽量プロセスの生成・消滅が発生しないため、アクティビティ方式の利点を活かしつつ、プロセッサ時間とメモリ消費量が大幅に節減されることが示された。

「遺言」のコンストラクトは、たとえばscheme[Rees86]の`call-with-current-continuation`文による、continuationと似たアイディアである。

schemeをはじめ、関数型のプログラミング言語では通常、処理系は変数などの環境をスタック上ではなく、すべてデータ領域中のヒープ上に取り、continuationにはその環境のすべてが引き継がれる。この場合には「遺言」のコンストラクトを用意しなくても、continuationの処理をする際に軽量プロセスがスタック上の情報を保持したままサスペンドすることが起こらないので、効率が低下しない。

しかしながら、手続き型のプログラミング言語では通常、ローカル変数はスタック上に置かれる。この場合は効率よく後処理実行を行うのに、本章で提案した「遺言」のコンストラクトが効率向上に有効である。「遺言」のコンストラクトでは引数の情報だけを引き継いで、すべての子タスクの完了を待った後で指定した手続きを実行する、という最小限の処理を宣言する。スタック上の情報は引き継がれない。そのために、親タスクを実行していた軽量プロセスを保持しておく必要がなくなり、この軽量プロセスが再利用できるようになる。後処理を実行するのに必要な情報だけを、引数として、もしくはヒープを用いて、引き継いでやればよい。

今回のツリー型の最適探索問題を用いた実験結果において、改良したアクティビティ方式では枝刈りをする場合により大きな効果が現れた。これは、ツリー型の最適探索問題においては、改良方式で採用した深さ方向優先のスケジューリングが適切であったためと考えられる。

現在、2台のSPARCプロセッサが共有バスにより結合された共有メモリ型マルチプロセッサ・システムにアクティビティ方式の並列実行管理機構を実装して性能評価することを試みている [Nakah93]。改良したアクティビティ方式は、より大規模な問題を解くにつれて、軽量プロセスを直接利用する方式、および、基本のアクティビティ方式に比べてよりよい実行効率が得られると予想されるので、このことを実機を用いて確認する予定である。

第 4 章

複数の並列応用プログラムを対象とするスケジューリング方式

4.1 はじめに

前章まで、共有メモリ型の汎用高並列計算機において非常に多数の細粒度のタスクを並列に実行するための並列実行管理機構として、あらかじめプロセッサの台数と同数の軽量プロセスを用意しておき、これらを繰り返し使用するアクティビティ方式を提案した。アクティビティ方式の並列実行機構は、タスクの実行中にサスペンドがまったく発生しなければ高い効率が実現できる。また、ネストした fork-join 形式の並列プログラムにおいて、親タスクによる子タスクの完了待ち合わせにより多数のサスペンドが発生する場合においても、「遺言」というコンストラクトを新たに導入することにより、子タスク待ちにおける軽量プロセスの生成の必要がなくなり、高い効率が実現できることを示した。しかしながら、前章までの議論では、並列計算機上で 1 つの応用プログラムだけが動作している場合を対象としていた。

本章では、複数の応用プログラムが同時に動作しており、かつ、各応用プログラムがアクティビティ方式を用いて細粒度の並列処理を行う場合について検討する。

4.2 複数の応用プログラムのスケジューリングに関する従来の研究

4.2.1 単一プロセッサ上での基本的なスケジューリング方式

単一プロセッサ上で、複数の応用プログラムをスケジュールする方式については、これまでにさまざまな研究がなされている。ここでは基本的なスケジューリング方式について述べる。

従来より、複数の応用プログラムをスケジューリングするときの要求として、

(1) 単位時間当たりの処理量、すなわちスループットを最大にする、

ことに加えて、

(2) すべての応用プログラムを公平に扱う、

ことがある。

最も単純なスケジューリング方式は到着順処理（first come first serve）と呼ばれる方式である。これは、応用プログラムが起動され、それが実行待ちリストに到着した順に従ってプロセッサを割り当てる方式である。ある応用プログラムに1度プロセッサが割り当てられると、そのプロセッサが解放されるまで管理系が横取りを行わない（non-preemptive である）。この方式は、すべての応用プログラムが中断されることがなく、効率的である。しかしながら、処理量の少ない応用プログラムが処理量の多い応用プログラムにより実行開始が妨げられ、応答時間が悪くなることが起こり得る。

これに対し、巡回（round robin）方式では、限られた時間だけ到着順にプロセッサを応用プログラムに割り当てる。この限られた時間（時間刻みとよぶ）を使い果たしても終わらないときは、次の応用プログラムによりプロセッサが横取りされる（preemptive である）。横取りされた応用プログラムは実行待ちリストの最後尾につながれる。この方式は単純に実現できて、応答時間が悪くならず公平である。

上記のように、単一プロセッサにおいてよい応答時間を得るためにはプロセッサの横取りは必要で、多くの管理系において横取りの手法は用いられている。また、応用

プログラムの何らかの性質に基づいて静的にもしくは動的に優先度を定め、優先度の最も高いものを実行待ちリストから次に実行するものとして選ぶという方式がより一般的に用いられているが、この場合にも横取りを行うと優先度の高い応用プログラムを優先して処理をしやすいという利点がある。

しかしながら、横取りを行う方式は、状況を切り替えるという代償、すなわちコンテクスト保存・切替・実行再開のコストを必要とする。このためのプロセッサ時間およびメモリ消費はきわめて大きいというオーバーヘッドの問題が生じる。

4.2.2 マルチプロセッサ・システム上でのスケジューリング方式

マルチプロセッサ・システム上において、複数の応用プログラムが、それぞれ軽量プロセスを利用して細粒度の並列処理を行う場合のスケジューリング方式については、これまでにいくつかの研究がなされている。

単一プロセッサ上でのスケジューリング・アルゴリズムをそのままマルチプロセッサ・システムに適用して、よい効率が得られるとは限らない。1例としては、優先度の高いプロセスが、時間刻みを使い果たしてプロセッサを横取りされ実行待ちリストにつながれた直後に、他のプロセッサ上で実行が再開される、といったことが起こり得ることがある。これば全く無駄なプロセッサ切替えであり、このようなことが頻繁に起きると効率の低下を招くことになる。

Machではギャング・スケジューリング（gang scheduling）が採用されている[Black90]。これは、同じ応用プログラムに属する軽量プロセスをできるだけ同時に異なるプロセッサで実行させようとするものである。これは、同じ応用プログラムに属する軽量プロセスどうしが同期をとる場合には、その待ち時間が小さくなるという利点があるが、やはり応用プログラムが頻繁に切り替わるのでそのオーバーヘッドが必要となるという問題を持っている。なおMachでは、プロセッサ数が十分あるにもかかわらず無駄な切替えが生じることはないようにするため、軽量プロセスの数が少ないほど時間刻みを大きくする工夫を行っている。

プロセッサの横取りを伴うスケジューリングを行う場合に、コンテクスト切替えのオーバーヘッドがかかり性能が低下するという問題を根本的にを解決するためには、

- すべての応用プログラムにおける、動作可能状態にある軽量プロセスの総数がプロセッサ台数を越えないように何らかの制御を行う、

ことが必要である。

Tuckerら[Tucke89]は、このことを指摘し、実行プロセスの総数がプロセッサの数をなるべく越えないように制御してコンテクストの切り替えが起こらないようにする

方式を提案した。具体的には、まず、各応用プログラムごとの並列実行を管理するスケジューラを用意するとともに、動作可能状態にある軽量プロセスの総数を管理するサーバを用意する。動作可能状態にある軽量プロセスの総数がプロセッサ台数を越えた場合には、サーバが各応用プログラムごとの並列実行を管理しているスケジューラに指示して軽量プロセスを休眠状態にさせる方式である。

この Tucker らによる方式では、動作可能状態にある軽量プロセスを休眠状態にするためのコスト、すなわちコンテクストの保存などのコストを必要とするが、1度これを行えば動作可能状態にある軽量プロセス数がプロセッサ台数を越えない間、コンテクスト切替えのオーバーヘッドが不要となる。

なお、上記の方式のように、

- 各応用プログラムごとの並列実行を管理するスケジューラ
  （本章ではローカル・スケジューラとよぶ）、
- 各応用プログラムに適切な数のプロセッサを割り当てるスケジューラ
  （同様に、グローバル・スケジューラとよぶ）、

の2段のスケジューラに分け、これらの間で情報を交換しながらスケジュールするという方式は Anderson ら [Ander92] でも試みられており、効率の向上を図られることが示されている。

4.3 アクティビティ方式を用いるアプローチ

前節に述べたことから、マルチプロセッサ上において各応用プログラムが細粒度の並列処理を行う場合、管理系に要求されることとして、

(1) すべてのプロセッサがアイドル状態にならず稼働すること、

(2) 処理量の小さい応用プログラムが処理量の大きい応用プログラムにより必要以上に待たされない、すなわちよい応答時間を保証すること、

(3) コンテクスト切替えのオーバーヘッドがかからないようにする、もしくは小さいものにできること、

が考えられる。

これを実現するための、各応用プログラムにおいてアクティビティ方式が用いられる場合の設計を検討する。まず、図4.1に示すように、前節で説明したローカル・スケジューラとグローバル・スケジューラの2段のスケジューラのモデルを用いる。

ローカル・スケジューラ　ローカル・スケジューラは、前章までに述べてきたアクティビティ方式による並列実行管理機構である。グローバル・スケジューラにより割り当てられたプロセッサ台数分だけ軽量プロセスを生成し、それらを繰り返し再利用する。

グローバル・スケジューラ　グローバル・スケジューラは総プロセッサ台数の範囲内で各応用プログラムに適当な数だけプロセッサを割り当てる。割り当てるプロセッサ数に変更がある場合には、それぞれのローカル・スケジューラにその変更を通知する。

このような構成をとることにより、上記の3つの要求は次のようにして満たすことができる。

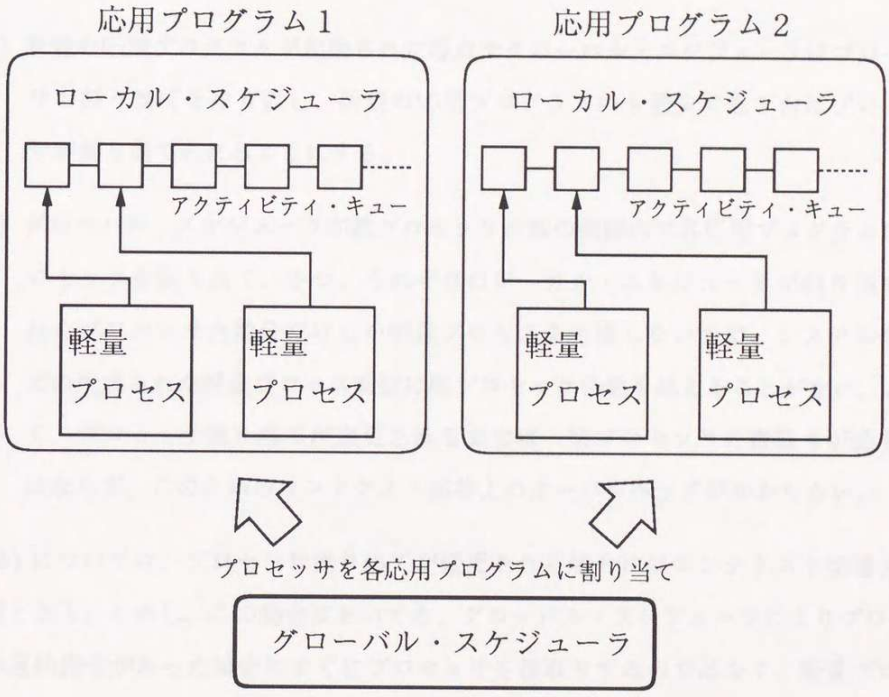


図 4.1: 各応用プログラムへのプロセッサの割り当て

(1) ローカル・スケジューラは基本的にはグローバル・スケジューラにより割り当てられた台数分の軽量プロセスを生成して使用するが、応用プログラムの並列度がある程度以上の時間連続して下がったままとなり、軽量プロセスが休眠した状態が続く時には、グローバル・スケジューラにプロセッサを返納する。このプロセッサをグローバル・スケジューラが他のプロセッサに再割り当てすることにより、プロセッサを有効に利用できる。

(2) 新規の応用プログラムが起動された時点でグローバル・スケジューラはプロセッサの割り当てをやり直し、新規の応用プログラムにも最少でも1台はプロセッサが割り当てられるようにする。

(3) グローバル・スケジューラが総プロセッサ台数の範囲内で各応用プログラムにプロセッサを割り当て、かつ、それぞれのローカル・スケジューラが割り当てられたプロセッサ台数分だけしか軽量プロセスを生成しないので、システム全体での生成された軽量プロセス総数は総プロセッサ台数を越えることがない。よって、プロセッサ割り当てが変更されるまでは一切プロセッサの横取りが必要とはならず、このためのコンテクスト切替えのオーバーヘッドがかからない。

(3) については、プロセッサ割り当てが変更される場合にはコンテクスト切替えが必要となる。しかし、この場合においても、グローバル・スケジューラによりプロセッサの返納指令があった場合にすぐにプロセッサを横取りするのではなく、軽量プロセスが現在実行しているアクティビティの処理を終えてからプロセッサを返納する。Tucker らの方式のように軽量プロセスを休眠させることは行わない。このようにすれば、図4.2に示すように、すでに作られている軽量プロセスをそのまま再利用でき、コンテクスト切替えのオーバーヘッドはきわめて小さなものにすることができる。

なぜならば、アクティビティの処理が完了すればスタック領域の退避は必要なく、元のスタック領域をそのまま使用すればよいからである。とくに、ほとんどの場合応用プログラムのテキスト領域とデータ領域の環境、具体的に MMU のマッピング・テー

ブルはすでに作成されているので、それをそのまま使用してテキスト領域とデータ領域を切り替えればよい。

もちろん、これはアクティビティの処理が細粒度であり、その完了を待ってもその待ち時間が小さいことを仮定している。不幸にしてアクティビティが大きな処理を行うようにプログラム記述されている状況も考慮に入れるならば、あるアクティビティが長時間軽量プロセスを占拠していないかどうかのチェックを、たとえば新規の応用プログラムが起動される時点で行わせる。このチェックに引っかかる場合にのみ軽量プロセスを休眠させる。

本方式は、応用プログラムにまたがったコンテクストの切替えも、アクティビティの完了の時点に行わせることがポイントである。これを行いさえすれば、グローバル・スケジューラの、各応用プログラムへのプロセッサ台数の割り当ては自由である。できれば、一旦割り当てたプロセッサは変更されることが少ないようにしてやれば、メモリ空間の切替えによるキャッシュ・メモリの無効化も防ぐことができ、さらに効率が向上することが期待できる。

現在、複数の並列応用プログラムが動作する環境におけるアクティビティ方式の試作システムの実現をめざしている [Koba.S93]。上記の環境においてもアクティビティ方式による性能の向上が図られることを確認する予定である。

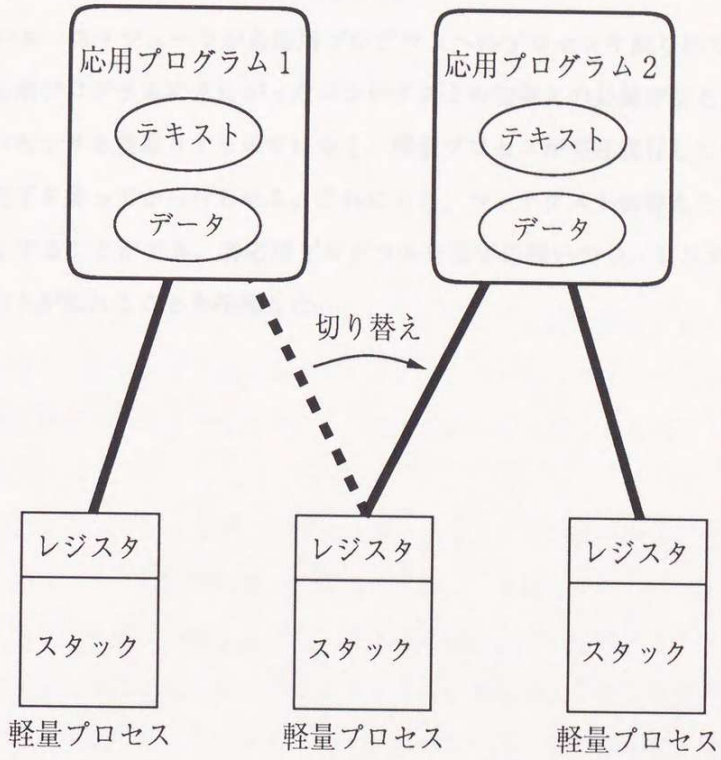


図 4.2: オーバーヘッドの小さなコンテクスト切替え

4.4 結論

本章では、複数の並列応用プログラムを対象とするスケジューリング方式を検討した。ローカル・スケジューラとグローバル・スケジューラの2段のスケジューラのモデルを適用し、各応用プログラムごとのローカル・スケジューラにはアクティビティ方式の並列実行機構を用いる方式を提案した。

グローバル・スケジューラが各応用プログラムへのプロセッサ割り当てを変更した場合に、応用プログラムにまたがったコンテクストの切替えの必要が生じたときでも、すぐにプロセッサを横取りするのではなく、軽量プロセスが現在実行しているアクティビティの完了を待ってから行わせる。これにより、コンテクスト切替えのオーバーヘッドを小さくすることができ、各応用プログラムを公平に扱いつつ、システム全体の処理性能の向上が図れることを指摘した。

第 5 章

システム・プログラムの並列実行

5.1 はじめに

1章で述べたように、マルチプロセッサ・システムを有効に利用するためには、利用者プログラムを並列化するのはもちろんのこと、オペレーティング・システムのシステム機能自体についてもマルチプロセッサ・システムに適合した構成とすることが要求される。

本研究ではオペレーティング・システムのシステム機能自体をマルチプロセッサ・システムの各ユニットに分散配置し、並列化することをめざす。これによりシステム機能がボトルネックとなることを防ぐ。具体的には、オペレーティング・システムのシステム機能を機能ごとに多数の軽量なプロセスに分割し、それらをランデブ通信により結合する。すなわち、プロセス・ネットワークを用いる方法によりシステムを実現する。本方式をプロセス・ネットワーク方式という。

プロセス・ネットワークを用いるオペレーティング・システムの構成法には、

(1) 設計、保守が容易である、

(2) 異機種への移植が容易である、

(3) メッセージ通信型のマルチプロセッサ・システムに自然な形で適用できる、

などの利点のほか、

(4) オペレーティング・システムに内在する並列性を自然な形で抽出できる、

という利点がある。すなわち、オペレーティング・システムは一群のシステム・プロセスのネットワークとして構成されているので、これらのシステム・プロセスをマルチプロセッサ・システムの各プロセッサに適切に配置することにより、

(1) オペレーティング・システムのシステム機能自体の並列実行、

(2) オペレーティング・システムのシステム機能と利用者プログラムとの並列実行、

が実現できる可能性がある。これにより、システム・コールの応答時間が短縮され、利用者プログラムの実行時間が短縮される可能性がある。

本研究では、まず、プロセス・ネットワークによるオペレーティング・システム内部の並列性の抽出について考察する。ここでは、単一のシステム・コールのレスポンスを向上させることを目的として、プロセス・ネットワークの動作を考える。まず、プロセス・ネットワークの構造の位相的な情報、すなわちプログラム時にわかるプロセス間の通信参照関係だけから、マルチプロセッサ・システムの各ユニットへの配置法の指針を求める。そして、試作システム上において種々の配置について実験を行い、どの程度の時間短縮が可能であるかを明らかにする [Nakay91][Nakay92a]。

インプリメントに使用したマルチプロセッサ・システムは、少数のコンピュータ・ユニットが共有メモリで結合された方式のもので、ユニット間のメッセージ通信にはこの共有メモリを使用した。また、インプリメントに使用したオペレーティング・システムは、田胡・益田 [Tago84] が報告した、プロセス・ネットワーク方式による UNIX である。

実験の結果、オペレーティング・システムのシステム機能自体の並列処理により処理性能が向上することが確認された。並列を意識せず設計されたオペレーティング・

システムをそのままプロセス・ネットワークによりインプリメントし、分散配置してやることにより性能の向上が可能であることが明らかになった。

5.2 プロセス・ネットワーク方式

並列処理によって処理性能の改善を図るためには、プログラムをできるだけ多数の並列実行単位に分割すること、および、並列処理の実現コストを軽減することが必要である。プロセス・ネットワークは、この目的に適合する性質をもつ。

プロセス・ネットワークを用いたオペレーティング・システムの構成を図5.1に示す。プロセス・ネットワーク方式では、次にあげる方針でオペレーティング・システムを設計する。

(1) 相互排除アクセスされる計算機資源の管理機能を単位としてモジュール分割を行う。

(2) 軽量なプロセスによりモジュールを実現する。これらのプロセスを、システム・プロセスとよぶ。システム・プロセスの配置は固定している。

(3) ランデブ通信により、システム・プロセスを結合する。システム・コール、および周辺機器からの割込みも通信に変換する。

システム・プロセス間の通信は、OS核が提供する核プリミティブ（call、acc、endr）により実現される。これらの核プリミティブについては**6.1.1**で詳しく述べる。

プロセス・ネットワーク方式によりUNIXシステムを再構成した。システム・プロセスを配置する資源管理機能の分割単位は、UNIXにおける資源アクセスの相互排除単位を踏襲し、UNIXシステムの資源管理アルゴリズムをそのまま適用した。これまでに図5.2に示すような構造によるオペレーティング・システムが設計されている[Tago84]。

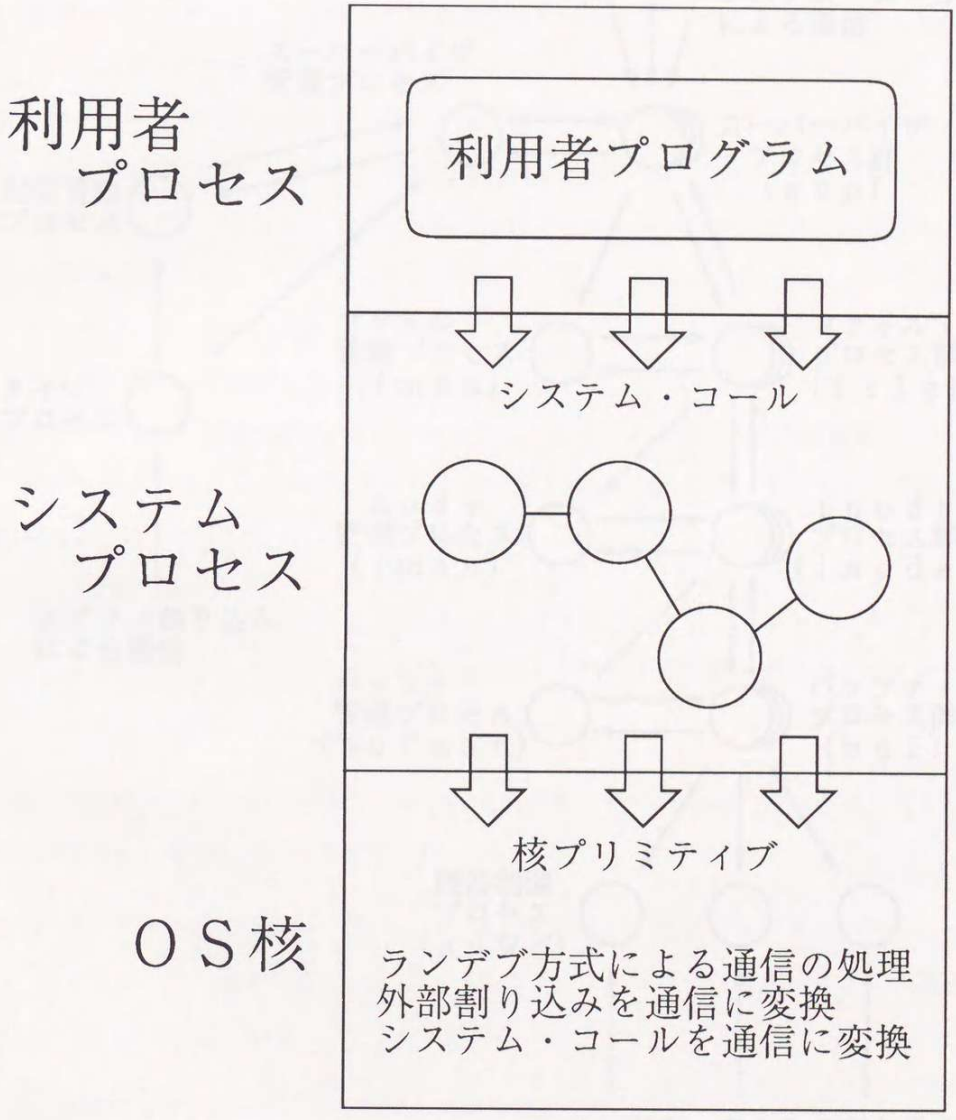


図 5.1: プロセス・ネットワーク方式によるオペレーティング・システム

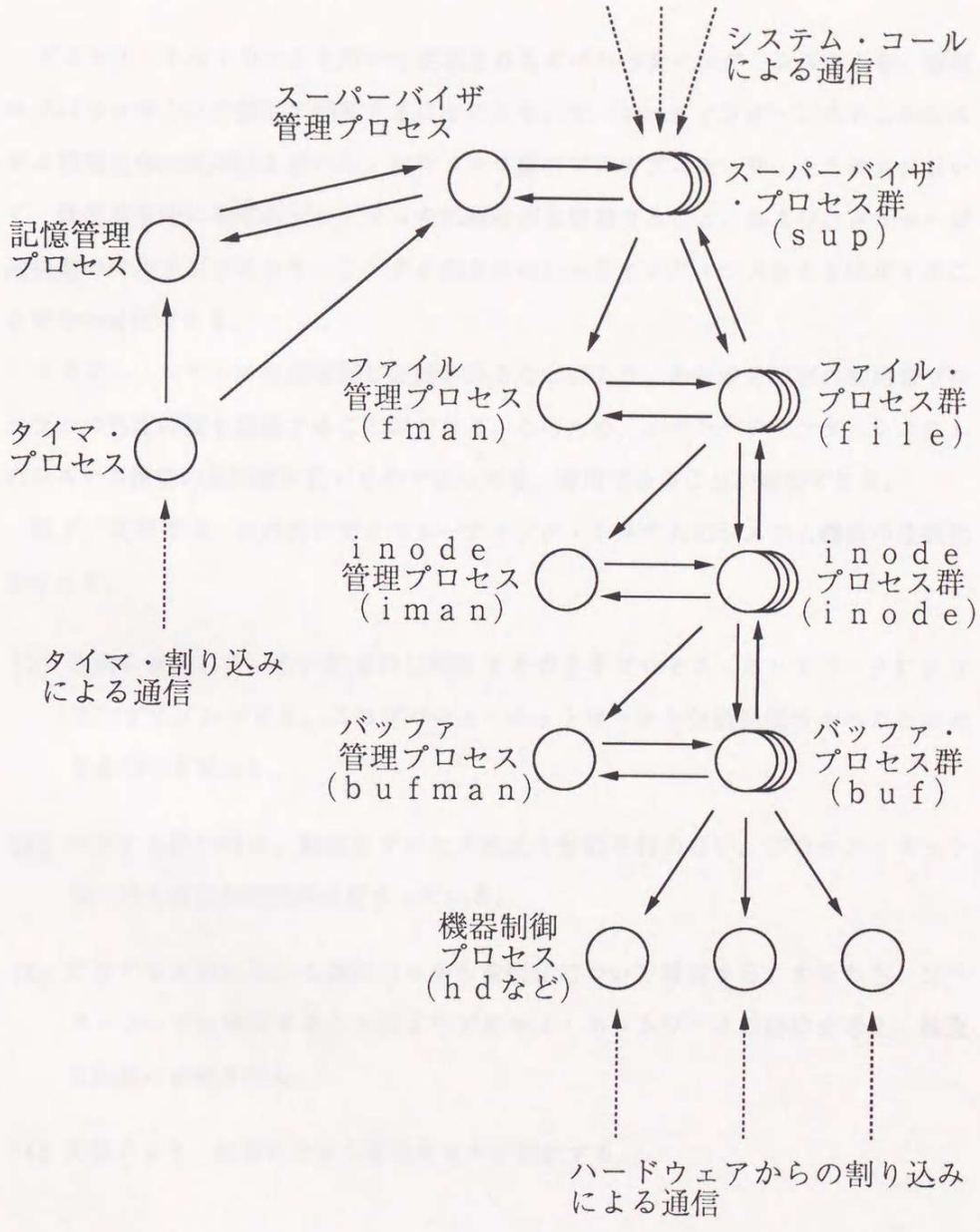


図 5.2: プロセス・ネットワーク方式による UNIX の再構成

5.3 プロセス・ネットワーク方式によるシステム機能の並列化

プロセス・ネットワークを用いて実現されたオペレーティング・システムを、複数のプロセッサ上に分散して配置することにより、オペレーティング・システムのシステム機能自体の並列化を試みる。共有メモリ型のマルチプロセッサ・システムにおいて、低多重度時に利用者プログラムの処理時間を短縮すること、および、メッセージ通信型のマルチプロセッサ・システム向きオペレーティング・システムを実現することがその目的である。

システム・コールの処理時間が短縮されることにより、あらゆる既存の利用者プログラムの処理時間を短縮することができる。このため、オペレーティング・システムのシステム機能の並列度が低いものであっても、有用であることが期待できる。

以下、次章では、次の方針でオペレーティング・システムのシステム機能の並列化を考える。

(1) 並列を意識していない従来の UNIX をそのままプロセス・ネットワークによりインプリメントする。このプロセス・ネットワークを分散配置させることにより並列化を試みる。

(2) システム動作時に、動的なプロセス生成や移動を行わない。プロセス・ネットワークの通信参照関係は定まっている。

(3) プログラム時にわかる情報のみから並列化について検討する。すなわち、ソース・コードを解析することによりプロセス・ネットワークの動作を考え、最適な配置の候補を得る。

(4) 実験により、候補の中から最適なものを選択する。

5.4 結論

本章では、オペレーティング・システムのシステム機能を機能ごとに多数の軽量プロセスに分割し、それらをランデブ通信により結合してシステムを実現するプロセス・ネットワーク方式について紹介した。

プロセス・ネットワーク方式により実現されたシステムは、設計、保守が容易で、異機種への移植が容易である、といった利点に加えて、マルチプロセッサ・システムの各ユニットに分散配置してやることにより、システム機能自体の並列処理が図れることを指摘し、システム機能の並列化についての方針を述べた。

第 6 章

プロセス・ネットワークとして実現されたシステム機能の並列性の抽出

6.1 プロセス・ネットワークのプログラム解析による並列性の抽出

6.1.1 プロセス・ネットワークの動作

プロセス・ネットワーク方式によるオペレーティング・システムでは図5.1に示したように、OS核が通信によって結合されたプロセス集合体としてオペレーティング・システムを実現するための環境を実現する。その機能の1つであるランデブ通信は、図6.1(a)に示す、call、acc、endrの核プリミティブにより実現される。これらの核プリミティブは、通信に関する同期のみを実現し、データ転送は手続き呼出しと同一の機構により実現される。

callを、相手プロセス名、エントリ名、および、通信データを引数として呼び出すことにより、ランデブ呼出しが実現される。また、エントリ名を引数としてaccを呼び出すことにより、ランデブの受付けが実現される。accは、ランデブが成立すると、相手プロセスのcallの引数列へのポインタを値として返すことにより、通信データの番地を知ることができる。データの返送、大量のデータ転送は番地呼出しによって実現する。ランデブはendrにより終了する。

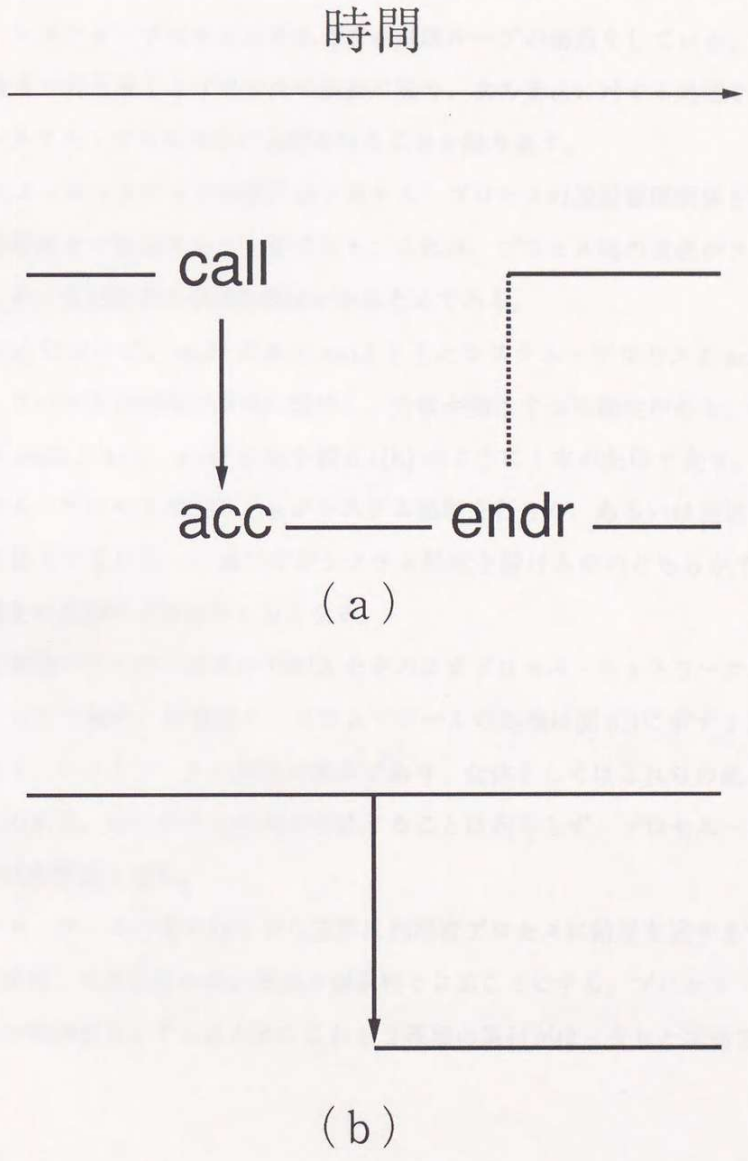


図 6.1: (a) 通信の核プリミティブ

(b) (a) の略記

各システム・プロセスには、図6.2のように必ず通信のプリミティブが埋め込まれている。システム・プロセスはそれぞれが無限ループの構造をしている。プロセス処理の最後まで行き着くとプロセスの最初に戻り、次の要求に対する処理を始める。再び他のシステム・プロセスとの通信を行うことを繰り返す。

プロセス・ネットワークの動作はシステム・プロセスの通信参照関係を追うことによりある程度まで推定することができる。これは、プロセス間の通信がランデブ方式なので、その実行順序に半順序関係があるためである。

図6.1(a)において、endrのあとcallをしたシステム・プロセスとaccをしたシステム・プロセスの両者が並列に動作し、分岐が発生する可能性がある。簡単化のため、以下call、acc、endrの組を図6.1(b)のように1本の矢印で表す。endrのあとシステム・プロセスの両者ともがシステム処理を行うか、あるいは通信を送った側は処理を終えて受け取った側だけがシステム処理を続けるかのどちらかで、図6.3に示す2通りの状態のどちらか1方となる。

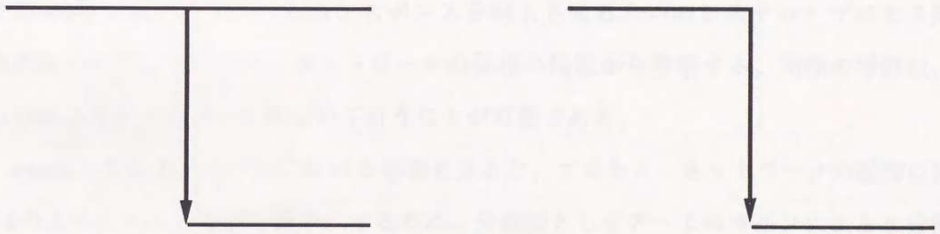
並列を意識していない従来のUNIXをそのままプロセス・ネットワークによりインプリメントした場合、同期的なシステム・コールの処理は図6.3に示す2通りの状態がプロセス・ネットワークの動作の基本であり、全体としてはこれらの組み合わせとなる。この結果、分岐のあと処理が合流することは起こらず、プロセス・ネットワークの動作は木構造となる。

システム・コールの受け付けから実際に利用者プロセスに結果を返すまでの処理の系列を主系列、それ以外の枝の系列を副系列とよぶことにする。プロセス・ネットワークの動作が木構造をしているためにこれら2種類の系列がはっきりと区別される。

```
process()
{
        for(;;){                    /* for 文による無限ループ */
                ..............
                call(put,c);        /* ランデブ通信の呼出し   */
        }
}

process()
        for(;;){                    /* for 文による無限ループ */
                b = acc(get);       /* ランデブ通信の受付け   */
                ..............
                endr();             /* 通信の終了 */
        }
}
```

図 6.2: システム・プロセスの内部記述の例



両方のプロセスで処理が
行われる場合

送った側は処理を終えて
受け取った側だけが処理
を続ける場合

図 6.3: 通信が行われた時のプロセス・ネットワークの動作

6.1.2 プロセス・ネットワークのプログラム解析による並列化

1つのシステム・コールの処理時間、すなわちレスポンスを早くすることをめざすならば、プロセス・ネットワークを複数のプロセッサ上に分散して配置するときに副系列ができるだけ主系列と別のプロセッサで処理されることが好ましい。そのような場合にシステム内部の並列度がより抽出されるものと考えられる。システム・プロセス間の通信参照関係から、並列化についての検討を行うことができる。今後、システム・プロセス間の通信参照関係のことを、プロセス・ネットワークの位相の情報とよぶ。

システム処理の大半はファイル入出力によって費やされていることが知られている[Hori91]。その中でもreadシステム・コールの頻度がもっとも高い。そこで、本研究ではreadシステム・コールのレスポンスを向上させるためのシステム・プロセス配置法について、プロセス・ネットワークの位相の情報から考察する。同様の考察は、writeシステム・コールについて行うことが可能である。

readシステム・コールにおける処理を見ると、プロセス・ネットワークの動作は図6.4のようになる。問題を簡単にするため、分割法としてア～エのラインによる2分割を考え、これを2台のプロセッサに配置するものとする。

前節で主系列、副系列について定義したが、具体的にreadシステム・コールにおける主系列は、システム・コールを受け付け、データをファイルから読み込み、データをシステム・コールの引数により指定されたユーザ領域に格納し、読み込みバイト数を返り値としてユーザにわたす処理を行う系列である。これに対し、現在読み込んでいるブロックの次のブロックに対する先読み要求を出す系列や、次のシステム・コールの受け付けの準備をする系列などが副系列である。

本節の初めに述べた、副系列をできるだけ主系列と別のプロセッサで処理することを考える。同期的なシステム・コールでは、副系列は非同期に動作可能なのに対し、主系列は終了するまで利用者プロセスの実行が待たされるので、できるだけ主系列に

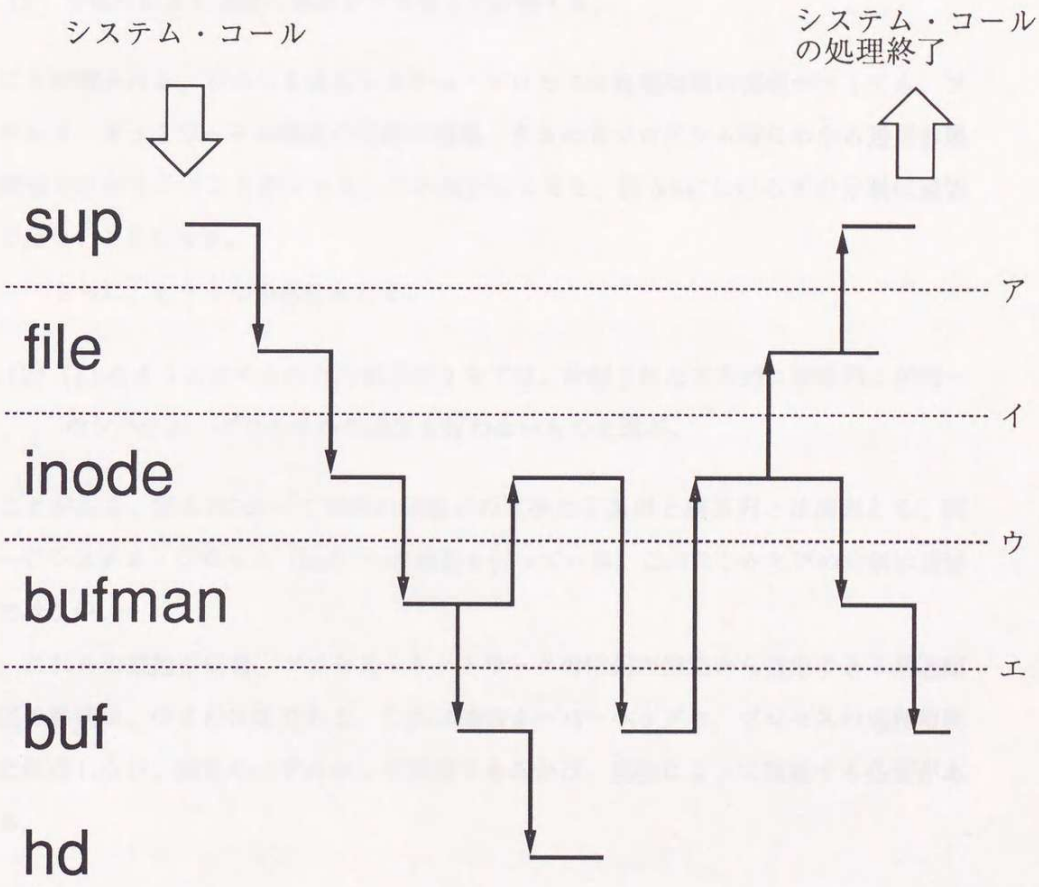


図 6.4: read システム・コールにおけるプロセス・ネットワークの動作

は通信による遅延を入れたくない。そこで、プロセッサ分割により副系列を他のプロセッサに切り分けることを考える。

このとき、プロセッサ分割の指針として、

(1) 分岐の起きた通信の場所でプロセッサ分割する、

ことが導かれる。このことは各システム・プロセスの処理時間の情報がなくても、プロセス・ネットワークの構造の位相の情報、すなわちプログラム時にわかる通信参照関係だけからいうことができる。この指針によると、図6.4におけるイの分割は適切ではないことになる。

さらに、もう1つの指針として、

(2) **(1)** のように求められた分割点のうちでは、分割された主系列と副系列とが同一のシステム・プロセスへの通信を行わないものを選ぶ、

ことがある。図6.4において初めの分岐点の直後の主系列と副系列とは両方とも、同一のシステム・プロセス（buf）への通信を行っている。このことからアの分割は適切ではない。

これらの議論の結果、プロセス・ネットワークの位相の情報から推定できる最適配置の候補は、ウまたはエである。これは通信オーバーヘッドや、プロセスの処理時間に依存しない。両者のいずれがより適切であるかは、実験によって確認する必要がある。

6.2 試作システムによる実験

6.2.1 試作システムの実現

プロセス・ネットワーク方式を用いて、システム機能自体を並列化した並列型オペレーティング・システムを、沖電気製のITCシステム上に実現した[Nakay90a][Nakay90b]。ITCシステムは図6.5に示すように、68020マイクロプロセッサ、68851 MMU、4 MBの局所メモリなどから構成されるコンピュータ・ユニットを、共有バスを用いて最大7台まで実装することができる。これに2 MBの共有メモリが装備されている。

ユニット間の通信用として、各ユニットにパイプ・レジスタが用意されている。送信先のユニット用のパイプ・レジスタにデータを書き込むことにより、送信先のプロセッサに割込みをかけることが可能である。この機能と共有メモリとを使って高速度のユニット間メッセージ通信機構を実現した[Tago89a]。共有メモリは通信路としてのみの利用であり、全体としてITCシステムをメッセージ通信型のマルチプロセッサ・システムのモデルとして用いた。

オペレーティング・システムを構成するプロセス・ネットワークを複数のプロセッサ上に分散して配置することによりシステムを実現した。これは、標準のUNIXシステムのアプリケーション・プログラムをそのまま実行することができる。

ここでは、固定ディスクのかわりにRAMディスクを用意した。システムの構成は、例として、図6.6のようになる。システム・プロセスのプロセッサへの配置は、ディスク制御プロセスなどデバイス・ドライバを除いて自由である。

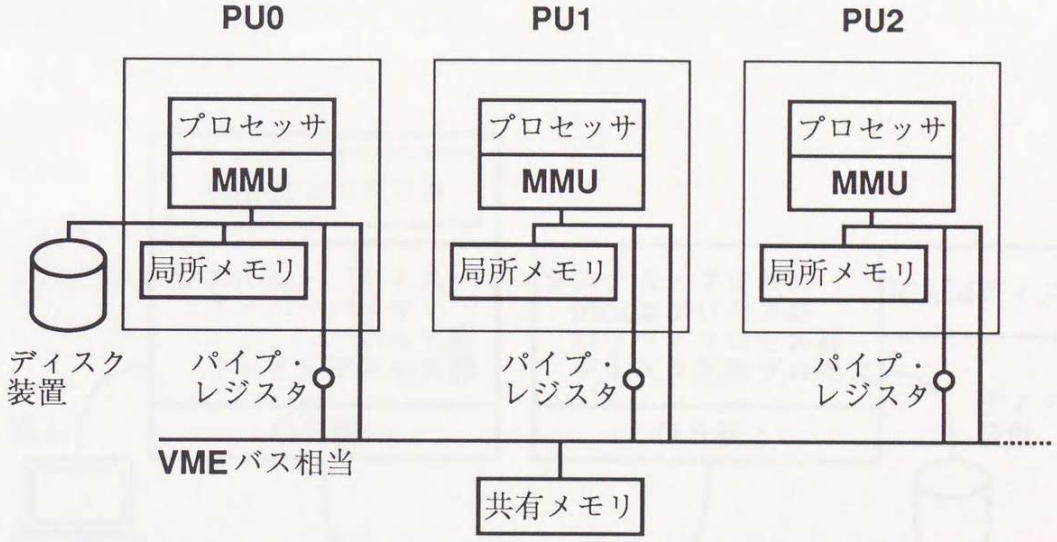


図 6.5: ITC システムのハードウェア構成

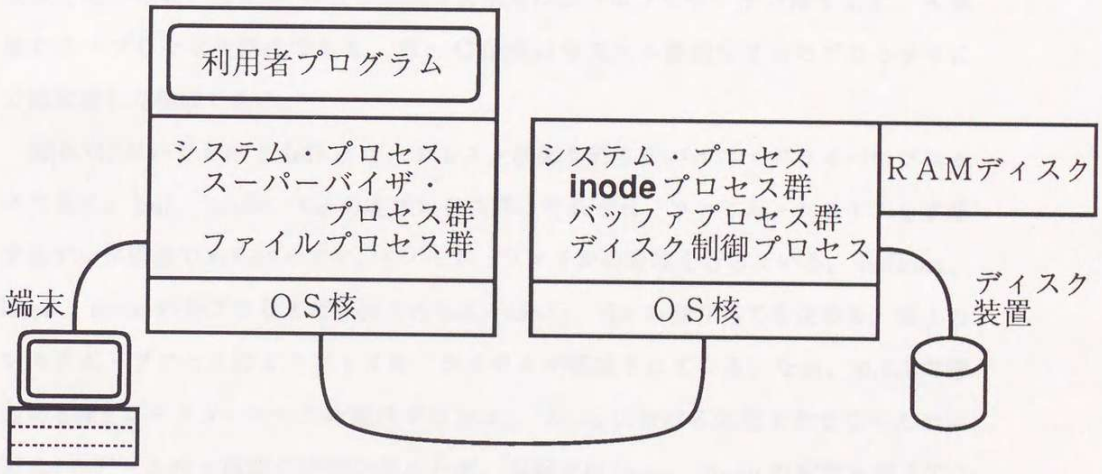


図 6.6: 実現したシステムの構成の例

6.2.2 実験

プロセス・ネットワークの2台のプロセッサへの分散配置の方法を変えながら、アプリケーション・プログラムを実行して、実行時間を測定した。これには、1つの利用者プログラムの開始から終了までに実際にかかる応答時間を測定した。

ここでは、システム処理の大半はファイル入出力によって費やされているので、ファイル・システムに関するシステム・プロセスの配置を変え、それ以外のシステム・プロセスは固定して実験を行った。オペレーティング・システム内部での並列処理の検討が目的であるので、利用者プロセスは1つのプロセッサ上にのみ配置した。

図6.7に示す7種類のシステムを実現して実験を行った。それぞれの構成をA～G構成と名づける。図6.7における破線は各構成におけるプロセッサ分割を示す。A構成が単一プロセッサ構成であり、B～G構成はシステム機能を2台のプロセッサに分散配置した構成である。

図6.7においてhd0とhd1とは、ディスク制御を行うデバイス・ドライバのプロセスである。buf、inode、fileの各プロセス群はそれぞれ、ファイル・システムを実現するデータ構造であるバッファ、iノード、ファイルの管理を行っている。bufman、iman、fmanの各プロセスはそれぞれbuf、inode、fileの割り当てを決める。以上のシステム・プロセスによりファイル・システムが構成されている。なお、**6.1.2**で考えたreadシステム・コールの処理ではiman、fmanにおける処理を含まないために図6.4のア～エの4種類の分割を考えたが、実験ではiman、fmanの配置も変えて7種類のシステムを考えた。

利用者プログラムとしては、5種類のものを取り上げ、各利用者プログラムの実行が完了するまでの時間を測定した。

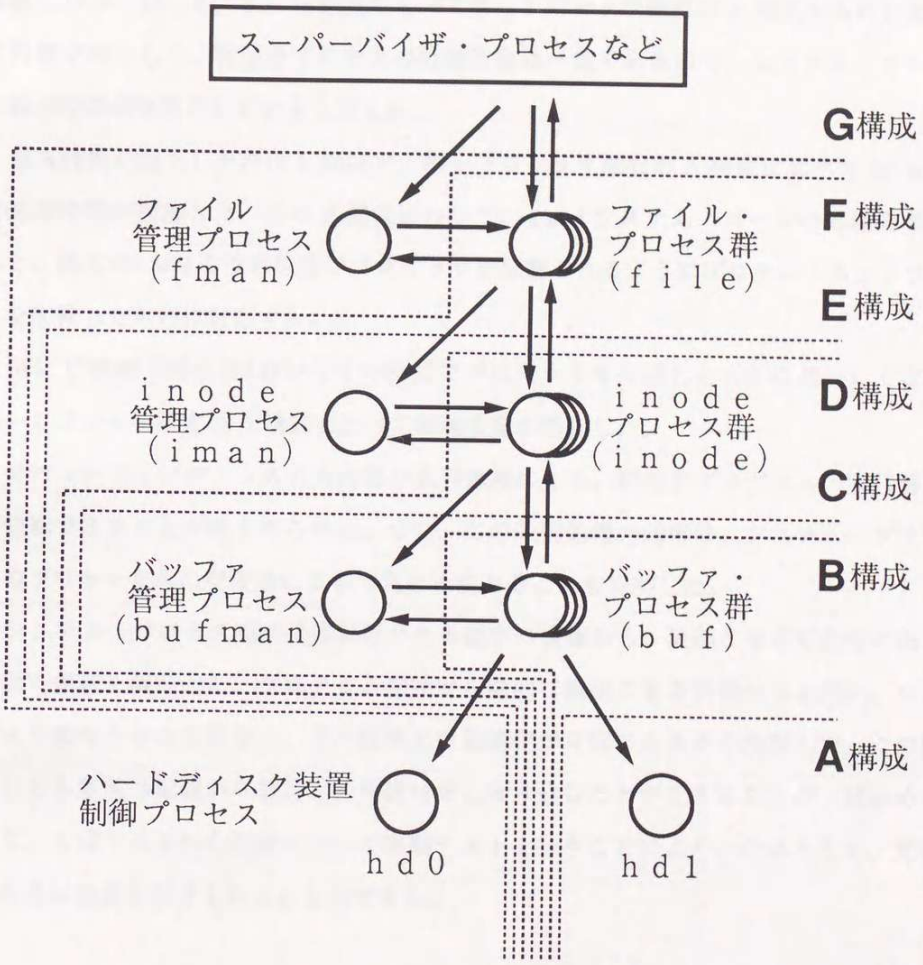


図 6.7: 実験に用いた 7 種類のシステム構成

(破線はそれぞれの構成でのプロセッサ分割を示す)

6.2.3 実験結果と考察

測定結果を表6.1に示す。これによると、システム機能を2台のプロセッサに分散配置したB、D、E、F、G構成のもので単一プロセッサ構成のA構成のものに比べて性能が向上した。利用者プロセスの処理時間は一定であるので、システム・コールの処理時間が改善されていると言える。

最も性能が向上したのはE構成で、単一プロセッサ構成のA構成に比べて30％前後処理時間が短縮した。このE構成について、readシステム・コールの処理を考えると、図6.4におけるウの位置でプロセッサが分割されるようにプロセス・ネットワークを配置したものに対応する。

逆にC構成（図6.4においてイの位置でプロセッサを分割したものに相当）では、単一プロセッサ構成のA構成に比べて処理性能が悪化した。

オペレーティング・システム内部の並列処理により、利用者プログラムの実行時間が短縮できることが確かめられた。また、この並列処理の効率は、システム・プロセスのプロセッサへの配置法によって大きく変わることも判明した。

システム・プロセス間で通信が行われる順序の情報から、最適となる可能性のある配置の候補を限定した。プログラム記述から静的に抽出できる情報のみを用い、システムを動作させることなく、どの程度まで最適化が可能であるかを検討した。この情報からも多数の配置から最適配置の候補をしぼり込むことができることが、確かめられた。しぼり込まれた候補について実動テストを行うことにより、全体として、比較的容易に最適な配置を得ることができる。

表 6.1: 実行結果（単位は sec）

	A	B	C	D	E	F	G
1つのファイルを読みこむ (250Kbytes)							
i)hd0上のとき	3.2	3.0	3.6	2.4	2.3	2.4	2.4
ii)hd1上のとき	3.4	2.6	3.4	2.1	2.0	2.5	2.5
複数のファイルを読みこむ (7files,260Kbytes)							
i)hd0上のとき	3.6	3.4	3.9	2.8	2.6	2.8	2.8
ii)hd1上のとき	3.6	3.0	3.6	2.3	2.2	2.7	2.7
(29files,450Kbytes)							
i)hd0上のとき	6.5	6.0	7.2	5.1	4.9	5.2	5.2
ii)hd1上のとき	6.7	6.2	6.6	4.4	4.1	5.1	5.1
hd0からhd1へのコピー (260Kbytes)	7.8	6.6	8.4	5.6	5.2	6.1	6.1
Cプログラムのコンパイル							
プログラムa	13.9	12.8	15.1	11.6	11.1	12.4	12.5
プログラムb	14.8	13.8	16.1	12.3	11.8	13.3	13.4

6.3 結論

システム内部で並列処理を行う並列型オペレーティング・システムを、通信により結合された軽量なプロセス集合体であるプロセス・ネットワークを用いて設計し、メッセージ通信型のマルチプロセッサ・システム上で実現した。

プロセス・ネットワークのプロセッサへの分散配置を適切に行えば、システム機能の並列度を上げることが可能である。本研究では、システム・コールのレスポンスを向上させるための並列性の抽出について研究した。プロセス・ネットワークの構造の位相の情報から、プロセス・ネットワークの各プロセッサへの分散配置法の指針を求めた。そして、この指針による分散配置が適切なものであることを実験により検証した。

プロセス・ネットワーク方式を採用した場合、原理的には、各システム・プロセスが並列に動作することが可能であり、単一システム・コールの実行時間の短縮だけでなく、多数のシステム・コールの並列実行の可能性が残されている。今後の検討課題である。

第 7 章

結言

本論文では、マルチプロセッサ・システムに使用するオペレーティング・システムについて、応用プログラムの並列処理を効率よくサポートするための実行管理と、オペレーティング・システムが提供するシステム機能自体の並列実行との、2 つの面から議論した。

応用プログラムの並列処理を効率よくサポートするための実行管理機構については、共有メモリ型のマルチプロセッサ・システムを対象として、高並列細粒度の応用プログラムを効率よく実行させることのできるアクティビティ方式の並列実行機構を提案した。この方式では、あらかじめプロセッサの台数と同数の軽量プロセスを用意しておき、これらを繰り返し使用することにより、無用の軽量プロセスの生成を防止する。この基本のアクティビティ方式は、タスクの実行中にサスペンドがまったく発生しない理想的な場合には、新しい軽量プロセスを生成する必要がないので、高い効率を実現できる。

また、ネストした fork-join 形式の並列プログラムおいて、親タスクが再帰的に子タスクを生成していき、しかも、それぞれの親タスクがそのすべての子タスクの実行の完了を待って後処理を実行する場合においても、実行効率を向上させるために、アクティビティ方式に「遺言」とよぶ新しいコンストラクトを追加する改良方式を提案した。

さらに、複数の応用プログラムが同時に動作しており、かつ、各応用プログラムがアクティビティ方式を用いて細粒度の並列処理を行う場合について考え、その場合の管理系の実現方式を検討した。

オペレーティング・システムが提供するシステム機能自体の並列実行については、オペレーティング・システムをプロセス・ネットワーク方式で実現することにより、システム機能自体の並列処理を試みた。システム・コール時の応答時間を向上させることをめざして並列性の抽出について考察し、プロセス・ネットワークを各プロセッサに分散配置するための指針を与え、この指針による分散配置が適切なものであることを実験により検証した。

今後の課題としては、次のようなものがある。

- 高並列の共有メモリ型マルチプロセッサ・システムの実機を用い、アクティビティ方式の並列実行管理機構の性能評価実験を行う。改良したアクティビティ方式は、より大規模な問題を解くにつれて、軽量プロセスを直接利用する方式、および、基本のアクティビティ方式に比べてよりよい実行効率が得られると予想される。並列応用プログラムについてもさまざまな種類のものを用い、アクティビティ方式の有効性を確認する。

- 複数の並列応用プログラムが動作する環境におけるアクティビティ方式の試作システムを、**3.3**と同様、高並列の共有メモリ型並列機シミュレータ上で実現中である [Koba.S93]。応用プログラムにまたがったコンテクストの切替えをアクティビティの完了の時点に行うことをしてやれば、複数の並列応用プログラムが動作する環境でも、アクティビティ方式が有効に動作することを確認する。また、グローバル・スケジューラが各応用プログラムにどのようにプロセッサ台数に割り当てるかを変えながら、より効率の向上が図れるようなプロセッサ割り当て法について検討する。

- 本研究ではオペレーティング・システムのシステム機能をカーネル・レベルに置いて分散配置法を検討したが、システム機能を利用者レベルに置いた場合についても、分散配置と並列化に関する検討を行う。

謝 辞

私が博士論文を作成するにあたり、暖かく見守り御指導してくださった指導教官の森下巖教授に心から感謝します。先生には、私が森下研究室に配属されてから5年間公私にわたりお世話になりました。

また、日頃より多くの御助言、御指導をいただいてきた出口光一郎助教授に深く感謝します。井上博允教授、田中英彦教授、武市正人教授には論文審査を通じて有益な御指摘、御指導をいただきました。深く感謝します。

私がオペレーティング・システムに関する研究を行う端緒を開いて下さり、熱心に御指導くださった田胡和哉助手[1]に感謝します。本研究全体を通して、その場その場で適切な御助言をいただき、つねに頼りにさせていただいていた永松礼夫助手に感謝します。

鈴木守助手、津村幸治君[2]、青木伸君[3]、木下敬介君をはじめ、森下研究室のスタッフの方々には、私の大学院での研究生活に多くの御協力をいただきました。とくに、数藤義明君[4]、檜垣博章君[5]、小林伸治君、小林健一君[6]、白木光彦君[7]、中畑昌也君、本橋健君には、本研究におけるシステム実現などに御協力いただきました。感謝します。また、6章における実験に用いたITCシステムを提供いただいた、沖電気工業(株)システム開発センターに感謝します。

最後に、私が博士課程に進学して研究することを許し、長期にわたり援助をしてくれた両親と、博士論文を書き上げる間そばで協力してくれた妻、代志子に感謝します。

現在の所属は次のとおり。

[1] 日本IBM(株)東京基礎研究所　[2] 千葉大学工学部情報工学科

[3] (株)リコー中央研究所　[4] キヤノン(株)情報システム研究所

[5] NTT(株)ソフトウェア研究所　[6] 東京大学大学院情報工学専攻田中研究室

[7] (株)日立製作所宇宙技術推進本部

参考文献

[Ander92] Anderson, T. E., B. N. Bershad, E. D. Lazowska and H. M. Levy:
Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism,
ACM Trans. Computer Systems, Vol.10, No.1, pp.53-79(1992).

[Black90] Black, D.:
Scheduling Support for Concurrency and Parallelism in the Mach Operating System,
IEEE Computer, Vol.23, No.5, pp.35-43 (1990).

[Cheri88] Cherinton, D. R.:
The V distributed system,
Communications of the ACM, Vol.31, No.3, pp.314-333 (1988).

[Degu90] Deguchi, K., K. Tago and I. Morishita:
Integrated Parallel Image Processings on Pipelined MIMD Multi-Processor System PSM,
Proc. 10th International Conference on Pattern Recognition, pp.442-444 (1990)

[Ginge87] Gingell, R. A., J. P. Moran and W. A. Shannon:
Virtual Memory Architecture in Sun OS,
Proc. USENIX Summer, pp.81-94 (1987).

[Higa90] 檜垣博章:
共有メモリ型マルチプロセッサのための並列実行方式の研究,
東京大学工学部計数工学科卒業論文 (1990).

[Hori91] 堀川隆:
バイブリッド・モニタ手法を用いたシステム動作の測定・解析,
情報処理学会オペレーティング・システム研究会報告, 91-OS-50-10
(1991).

[Imamu92] 今村信貴, 桑山雅行, 宮崎輝樹, 林茂昭, 福田晃, 富田眞治:
並列オペレーティング・システム K1 の設計と実現
— フリー・プロセッサ・キューを用いたプロセッサ管理 — ,
並列処理シンポジウム JSPP'92 論文集, pp.305-312 (1992).

[Koba.K92] 小林健一, 中山泰一, 永松礼夫, 森下巖:
共有メモリ型並列機のためのアクティビティ方式並列実行機構 (2)
— 環境切替により後処理実行を行う方式の提案 — ,
第 45 回情報処理学会全国大会論文集, 3P-2 (1992).

[Koba.S93] 小林伸治:
複数の並列応用プログラムをサポートするアクティビティ方式
スケジューラの研究
東京大学大学院工学系研究科情報工学専攻修士論文 (1993).

[Maeka91] 前川守, 所真理雄, 清水謙太郎:
分散オペレーティング・システム — UNIX の次にくるもの —,
共立出版 (1991).

[Mohr91] Mohr,E., Kranz,D.A. and Halstead,R.H.:
Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs,
IEEE trans. Parallel and Distributed Systems, Vol.2, No.3, pp.264-280 (1991).

[Mori90] 森下巖:
多段結合ネットワークを用いる超並列マシンのためのパイプライン化 MIMD プロセッサ,
情報処理学会論文誌, Vol.31, No.4, pp.523-531 (1990).

[Mulle90] Mullender, S. J. and G. van Rossum:
Amoeba: A Distributed Operating System for the 1990s,
IEEE Comput., Vol.23, No.5, pp.44-53 (1990).

[Naga91] 永松礼夫, 数藤義明, 森下巖:
多重命令流プロセッサを用いた共有メモリ型並列機の性能評価
— グラフ探索問題の処理時間 —,
電子情報通信学会コンピュータシステム研究会報告, CPSY91-29(1991).

[Nakah93] 中畑昌也, 本橋健, 中山泰一, 永松礼夫, 出口光一郎, 森下巖:
アクティビィティ方式並列実行機構の共有メモリ型並列機への実装と評価,
第 46 回情報処理学会全国大会論文集, 4F-1 (1993 発表予定).

[Nakay90a] 中山泰一, 田胡和哉, 森下巖:
並列型 OS の設計と実現,
情報処理学会オペレーティング・システム研究会報告,
90-OS-47-1 (1990).

[Nakay90b] 中山泰一, 田胡和哉, 出口光一郎, 森下巖:
並列型 OS の実現と評価,
情報処理学会計算機アーキテクチャ研究会報告, 90-ARC-83-28 (1990).

[Nakay91] 中山泰一, 田胡和哉, 森下巖:
プロセス・ネットワークによる OS 内部の並列実行,
並列処理シンポジウム JSPP'91 論文集, pp.317-324 (1991).

[Nakay92a] 中山泰一, 田胡和哉, 森下巖:
プロセス・ネットワークとして実現した UNIX カーネルの並列動作によるシステム・コール・レスポンス時間短縮の試み,
情報処理学会論文誌, Vol.33, No.3, pp.330-337 (1992).

[Nakay92b] 中山泰一, 永松礼夫, 森下巖:
共有メモリ型並列機のためのアクティビティ方式並列実行機構の研究
— タスクの親子関係を利用する後処理実行機構の導入 — ,
情報処理学会オペレーティング・システム研究会報告, 92-OS-55-3 (1992).

[Nakay92c] 中山泰一, 白木光彦, 永松礼夫, 森下巖:
共有メモリ型並列機のためのアクティビティ方式並列実行機構 (1)
— 子プロセス待ちにおいて後処理を分割し性能改善を図る方式 — ,
第 45 回情報処理学会全国大会論文集, 3P-1 (1992).

[Nakay93] 中山泰一, 永松礼夫, 出口光一郎, 森下巖:
共有メモリ型並列機のための新しいアクティビティ方式並列実行機構,
情報処理学会論文誌, Vol.34, No.5 (1993 掲載決定).

[Ouste88] Ousterhout, J. K., A. R. Cherenson, F. Douglis, M. N. Neson and B. B. Welch:
The Sprite Network Operating System,
IEEE Computer, vol.21, No.2, pp.23-36 (1988).

[Rashi86] Rashid, R. F.:
Threads of a New System,
UNIX Review, vol.4, No.8, pp.37-49 (1986).

[Rees86] Rees, J. and W. Clinger:
Reviced^3 Report on the Algorithmic Language Scheme,
ACM SIGPLAN Notices, Vol.21, No.12, pp.37-79 (1986).

[Shimi89] 清水謙太郎:
分散オペレーティング・システムの研究調査,
情報処理学会オペレーティング・システム研究会報告, 89-OS-45-1 (1989).

[Shinj92] 新城靖, 清木康:
並列プログラムを対象とした軽量プロセスの実現方式,
情報処理学会論文誌, Vol.33, No.1, pp.64-73 (1992).

[Sun88] Sun microsystems:
System Services Overview, Part No.: 800-1753-10, pp.71-106 (1988).

[Tada90] 多田好克, 寺田実:
移植性・拡張性に優れたCのコルーチンライブラリ実現法,
電子情報通信学会論文誌, Vol.J73-D-1, No.12, pp.961-970 (1990).

[Tada92] 多田好克:
機種に依存しない利用者 threads ライブラリ,
情報処理学会プログラム - 言語・基礎・実践 - 研究会報告, 92-PRG-8-22 (1992).

[Tago84] 田胡和哉, 益田隆司:
OS の構造記述に関する一試み,
情報処理学会論文誌, vol.25, No.4, pp.524-534 (1984).

[Tago89a] 田胡和哉, 中山泰一:
軽量なプロセスの集合によるマルチプロセッサ用 OS の設計,
第 38 回情報処理学会全国大会論文集, 5N-1 (1989).

[Tago89b] 田胡和哉, 中山泰一:
分散並列システムの構成とその実現方式,
情報処理学会オペレーティング・システム研究会報告, 89-OS-45-8 (1989).

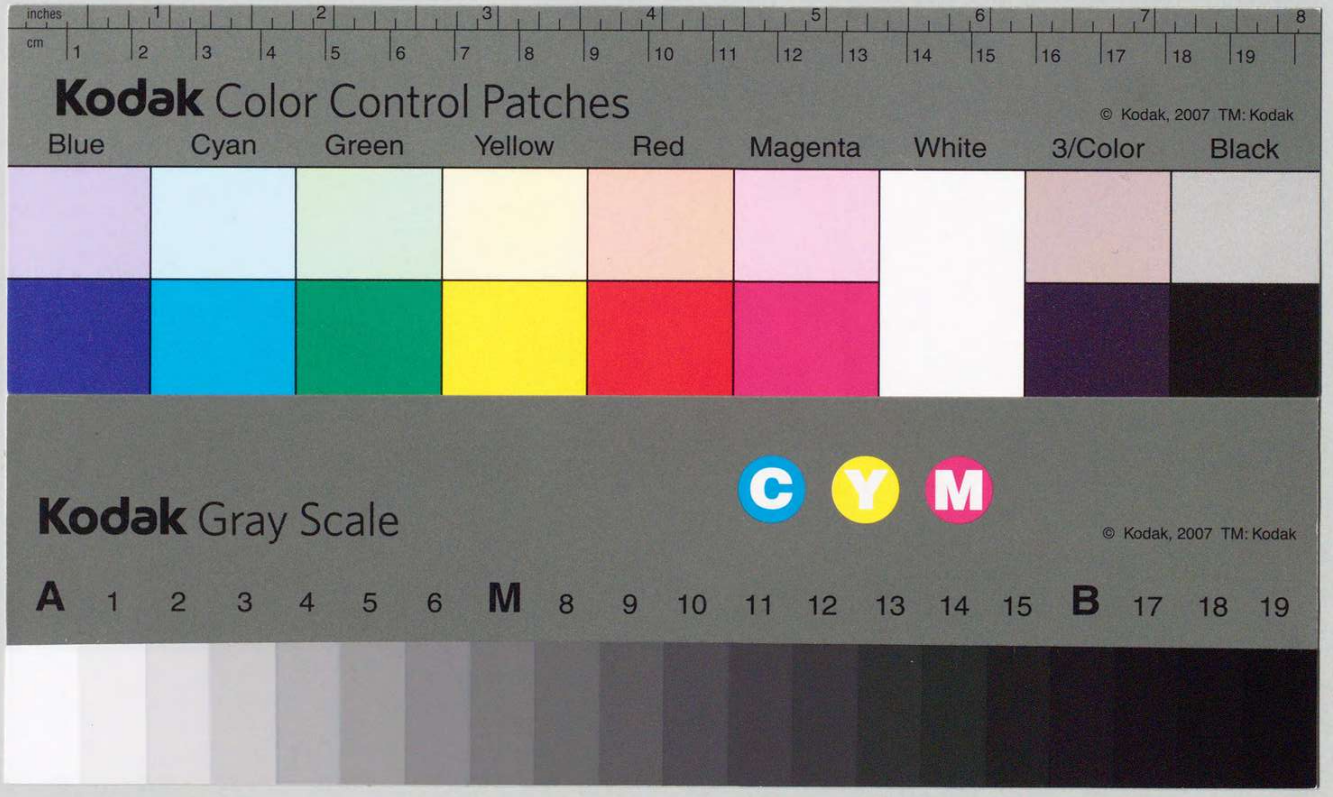
[Tago91] 田胡和哉, 檜垣博章, 森下巖:
共有メモリ型並列計算機のためのアクティビティ方式を用いる並列実行環境,
情報処理学会論文誌, Vol.32, No.2, pp.229-236 (1991).

[Tanen85] Tanenbaum, A. S.:
Distributed Operating Systems,
ACM Computing Surveys, Vol.17, No.4, pp.419-470 (1985).

[Tomi86] 富田眞治:
並列計算機構成論, 昭晃堂 (1986).

[Tsune92] Tsunedomi, K., K. Murakami, A. Fukuda and S. Tomita:
A Message-Pool-Based Parallel Operating System for the Kyushu University Reconfigurable Parallel Processor,
Journal of Information Processing, Vol.14, No.4, pp.423-432 (1992).

[Tucke89] Tucker, A. and A. Gupta:
Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors,
Proc. 12th ACM Symp. Operating Systems Principles, pp.159-166 (1989).


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19