共有メモリ型並列機における細粒度並列処理のための新しいアクティビティ方式並列実行機構

中山泰一[†*] 永松礼夫[†]
出口光一郎[†] 森下巖[†]

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処理単位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が提唱されている．その利点として，どのような形式の並列プログラムにも適用でき，処理単位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ことが確認されている．しかしながら，ネストした fork-join 形式の並列プログラムにおいて，親処理単位が再帰的に子処理単位を生成していき，しかも，それぞれの親処理単位がそのすべての子処理単位の実行の完了を待って後処理を実行する場合，子処理単位の完了待ち合わせによる多数のサスペンドが発生し，従来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．本論文では，上記形式のプログラムの実行効率をも向上させるため，従来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，後処理を子処理単位に「遺言」して親処理単位は実行を完了し，最後に処理を終える子処理単位を実行し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試作し，シミュレーションにより性能評価した．その結果，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．

A New Activity Based Execution Mechanism for Fine Grain Parallel Processing on Shared Memory Machines

Yasuichi Nakayama,[†*] Leo Nagamatsu,[†] Koichiro Deguchi[†]
and Iwao Morishita[†]

An activity based execution mechanism was proposed for effective execution of a large number of fine grain procedures on shared memory machines. This mechanism deals with an execution request and its execution separately to reuse a light-weight process prepared for a previous procedure execution. However when a process suspends for waiting the completion of all the child procedures, a new additional light-weight process must be created. In this paper, a new construct called "make a will" is introduced to reduce the number of these additional light-weight process creations. When the post-processing of a procedure is declared by using the construct, it is executed after the completion of all its child procedures by utilizing the light-weight process used for the last child procedure. It is shown that both the execution time and memory consumption are reduced by the new mechanism.

1. はじめに

汎用高並列計算機[1]を用いて非常に多数の細粒度の処理単位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．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処理単位を実行させる方式ではあまりにオーバヘッドが大きく，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が提案されている[2),3)]．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し，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

† 東京大学工学部計数工学科
Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo
* 現在 電気通信大学電気通信学部情報工学科
Presently with Department of Computer Science, Faculty of Electro-Communications, University of Electro-Communications

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処理単位が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．

田胡ら[4]は，共有メモリ型並列機を対象として，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って無用の軽量プロセスの生成を防止する方式を提案した（これをアクティビティ方式とよぶ）．これは，並列に実行すべき処理単位が発生することに軽量プロセスを生成し，処理単位の処理が完了するとこれを消滅させる方式ではなく，並列に実行すべき処理単位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個の処理単位の実行が完了するごとにキューから要求を1個取って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．処理単位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ことが確認されている．

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，親処理単位が再帰的に子処理単位を生成していき，しかも，それぞれの親処理単位がそのすべての子処理単位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，親処理単位が子処理単位の完了待ち合わせをすることにより多数のサスペンドが発生し，従来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．

本論文では，アクティビティ方式の利点を活かしつつ，上記形式のプログラムの実行効率をも向上させるため，従来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，後処理は自分では実行しないことにして「遺言」として子処理単位に伝達し，この「遺言」は最後に処理を終える子処理単位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．親処理単位は「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．上記形式のプログラム以外の場合でも，従来のアクティビティ方式とほぼ同一の実行効率が得られるものである．

以下，まず，従来方式の利点と問題点を整理し，改良方式の具体的な実行機構を提案する．つぎに，試作した改良方式管理機構のシミュレーションによる性能評価の結果を報告する．シミュレーション例では，処理時間は数 10% 短縮され，スタック領域のメモリ消費は 1/50 以下に節減された．

2. アクティビティ方式

2.1 アクティビティ方式の基本原理

アクティビティ方式では，並列に実行すべき処理単位の発生と，処理単位の実行機構の用意を別個に取扱う．並列実行可能な処理単位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，並列に実行させる処理単位が発生すると，

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処理単位を発生すると，**図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

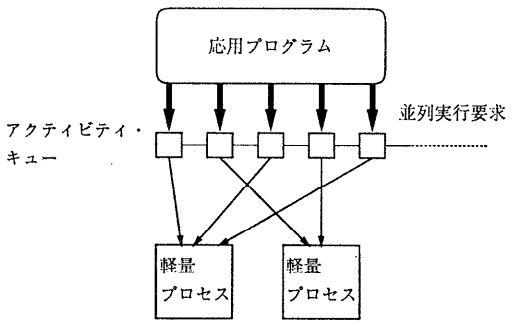


図1 アクティビティ方式による並列処理機構
Fig. 1 Activity based parallel execution mechanism.

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，処理量が小さくボトルネックを生じにくい

図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．

2.2 従来方式の問題点

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．

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，親処理単位が再帰的に子処理単位を生成していき，しかも，それぞれの親処理単位がそのすべての子処理単位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，親処理単位が子処理単位の完了待ち合わせをすること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．

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．

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3**に示すように，まず6個の子処理単位を作り，そのそれぞれが5個の子処理単位を作り，と繰り返すものとなり，全体で 1237 個の処理単位が作られる．そのうち517個の処理単位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処理単位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．

3. 改良方式の提案

前章に述べた問題点は，田胡ら[4]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

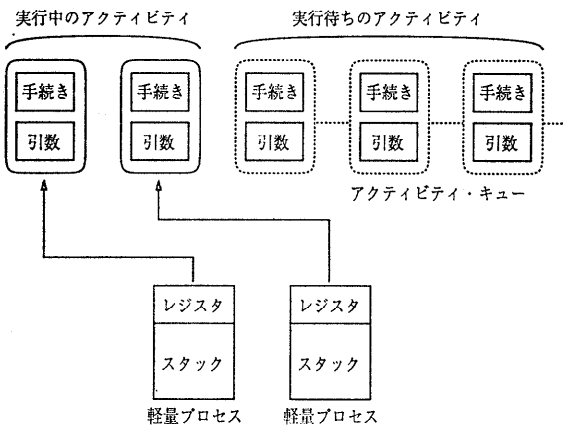


図2 アクティビティとその実行機構
Fig. 2 Activities and the execution mechanism for them.

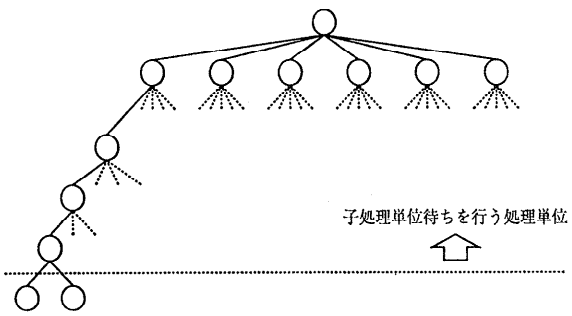


図3 7都市巡回セールスマン問題における処理単位の実行要求
Fig. 3 Execution request for child procedures in a 7-city traveling salesman problem.

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．

本論文では，親処理単位は後処理を「遺言」として子処理単位に伝達して実行を完了し，最後に実行を完了した子処理単位にその「遺言」を実行させる方式を提案する．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，従来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，従来方式とほぼ同一の効率で実行できる．

3.1 「遺言」の作成

具体的には，まず，処理単位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4**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．親処理単位が子処理単位の完了を待って後処理を実行するしないにかかわらず，子処理単位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．これにより子処理単位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．子処理単位が発生するごとに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，親処理単位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5**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」の領域に後処理の要求が格納され，同時に親処理単位は「遺言」を遺したまま強制的に終了される．

親処理単位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，そのまま次の処理単位の実行を開始することができる．

3.2 「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，子処理単位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，子処理単位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，終了後に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，子処理単位を実行した軽量プロセスには処理の終了後，親処理単位のFTDを参照して，自分が実行していた処理単位が親処理単位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処理単位が最後に処理を終えた子に当たる

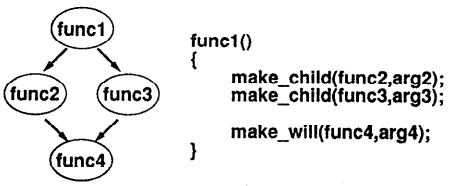


図5 子処理単位 func2, func3 の生成と，「遺言」func4 作成のシステム・コール
Fig. 5 System calls for creating a child and "making a will".

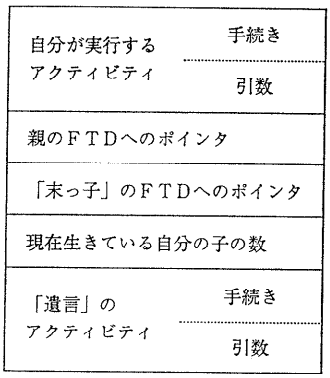


図4 家系記述子（Family Tree Descriptor）の構造
Fig. 4 Structure of the Family Tree Descriptor (FTD).

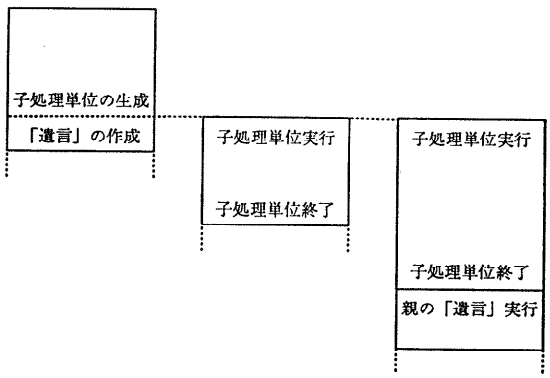


図6 「遺言」の作成とその実行のタイミング
Fig. 6 Timing of "making a will" and its execution.

場合には，**図6**に示すように親の処理単位の「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，さらに，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7**を用いて説明する．処理単位Eは処理単位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，その親処理単位である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7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，その親処理単位である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Gを実行していた軽量プロセスが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，先祖を手繰って「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・キューから処理単位を取ることを行う．

もちろん，「遺言」の中でも子処理単位の生成や「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4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，元の親処理単位の延長として「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処理単位の親は，元の親処理単位である．

3.3 「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，親処理単位は後処理を「遺言」の形で残し，最後に処理を終えた子処理単位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，親処理単位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，新しい処理単位の実行開始が可能になる．

その軽量プロセスは，新しい処理単位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，親処理単位には必ず自分が生成した子処理単位が存在するから，ここでは，親処理単位を実行していた軽量プロセスが，最後に生成した子処理単位（「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処理単位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」以外の子処理単位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

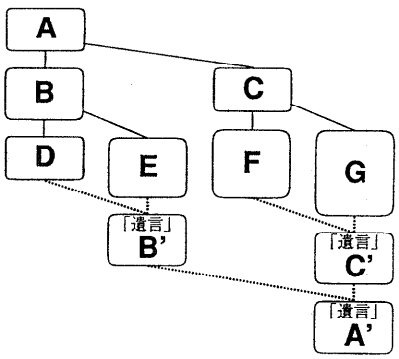


図7 「親の親」の「遺言」の実行
Fig. 7 Execution of "grand parent's will".

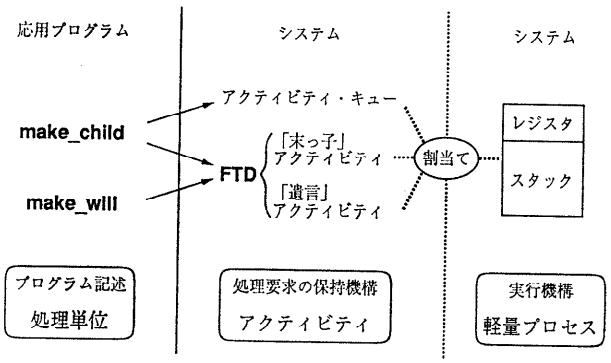


図8 提案方式の構成
Fig. 8 Overview of the proposed mechanism.

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．

処理単位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のための管理機構の構成を**図8**に示す．従来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9**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，処理単位を実行している間に子処理単位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．処理単位のツリーにおいて，葉に当たる処理単位の場合には「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処理単位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にも対応でき，その場合には「遺言」は同じ軽量プロセスによってただちに実行される．

4. 実　　験

4.1 実験環境

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．

実験に用いたシミュレータは，筆者らの研究室において共有メモリ型並列機の性能を評価するために試作したものである．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るものとしてある．すなわち，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[5),6)]．

共有メモリ型並列機において，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，各プロセッサは遅延なしで共有メモリにアク

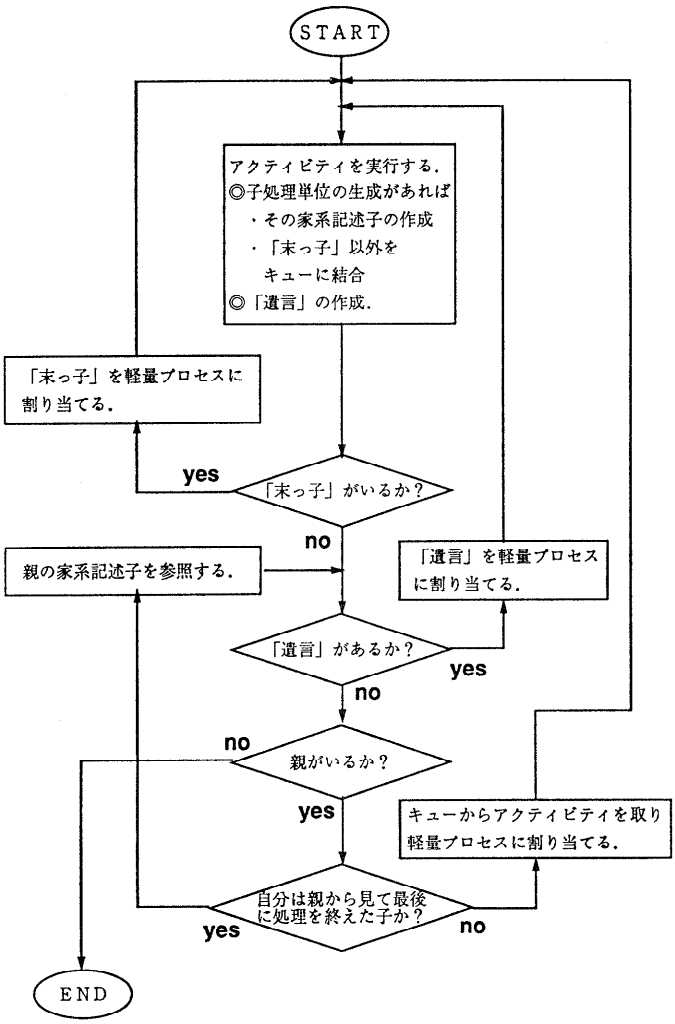


図 9 提案方式における並列実行管理機構のフローチャート
Fig. 9 Flowchart of the proposed parallel execution management mechanism.

セスできるものとみなすことができる.

本実験の目的は，従来のアクティビティ方式と提案した方式の性能を比較することが目的であるから，実験環境を単純化して，

○各プロセッサを遅延なしでアクセスできる共有メモリ，すなわち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.

4.2 実験内容

並列実行管理機構としては，

(S1) 提案方式のもの，

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

(S3) 従来方式のもの，

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，7都市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．全処理量が一定となるように枝刈りをしない場合と，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との，両方の場合のプログラムで実験した.

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

B：S2—P1

C：S3—P1

D：S3—P2

の4個の場合を評価した.

4.3 実験結果と考察

実行した処理時間の結果を**図10**，**図11**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10）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．従来方式による実験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．これは子処理単位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．提案方式の実験Aは，従来方式の実験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従来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

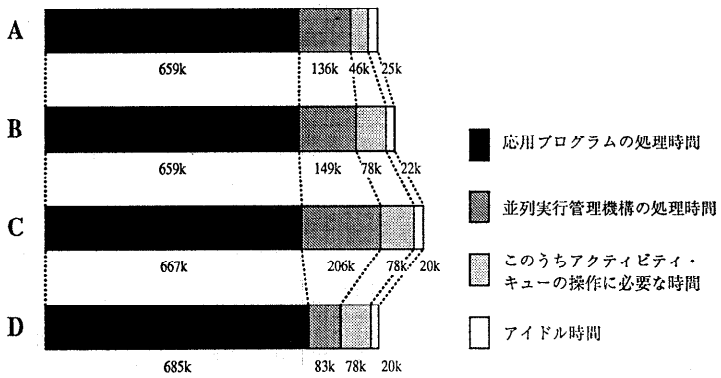


図10 7都市巡回セールスマン問題のシミュレーション結果（枝刈りをしない場合）

Fig. 10 Simulation results of a 7-city traveling salesman problem (without pruning).

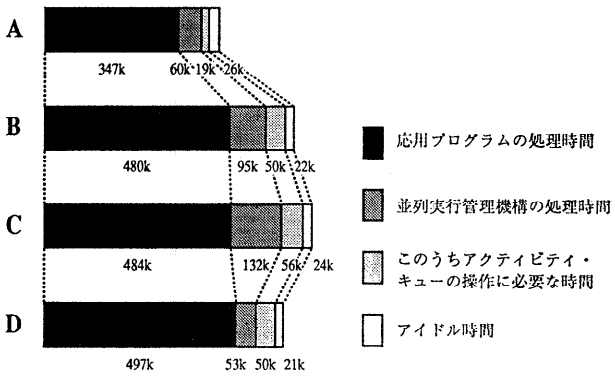


図11 7都市巡回セールスマン問題のシミュレーション結果（枝刈りをする場合）

Fig. 11 Simulation results of a 7-city traveling salesman problem (with pruning).

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，提案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11）において，提案方式の実験Aは，従来方式の実験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提案方式と従来方式との比較を行った．その結果を**表1**に示す（割り当てたメモリ量で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，提案方式の実験Aは，従来方式の実験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の生成を節減したことに相当する．2章で議論したように，従来方式では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，提案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，このコストと比べスタックの消費を節減できるメリットのほうがはるかに大きい．

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処理単位の個数分を消費する．しかしながら，FTDは，そのすべての子処理単位の実行が完了し，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の個数は全処理単位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．

表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）
Table 1 Memory consumption in a 7-city traveling salesman problem.

	スタック	アクティビティ・キュー+FTD
実験A（提案方式）	1664	48328
実験C（従来方式）	105240	24032

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，従来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．

5. おわりに

従来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．本論文では，後処理を「遺言」として子処理単位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を報告した．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．

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，提案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，提案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．

本論文では，各プロセッサ要素が任意のメモリ要素に同一のコストでアクセスできる構成の場合を取り扱ったが，メモリ要素によりアクセスのコストが異なる構成のものも提案されている．この構成の場合には，処理単位のウェイティング・キューを各プロセッサ要素ごとに用意するとよい結果が得られることが知られている[7]．アクティビティ方式においても，この構成の場合には，アクティビティ・キューを各プロセッサ要素ごとに用意する設計によればキュー操作がボトルネックとなる欠点が除去される．すなわち，上記のような構成の並列機に対しても本論文で提案した方式が有効になると考えられる．

謝辞 本論文をまとめるにあたり，東京大学工学部計数工学科卒業研究としてシステム実現および評価に協力された，小林健一氏（現在，同大学院情報工学専攻），白木光彦氏（現在，(株)日立製作所宇宙技術推進

本部）に感謝する．

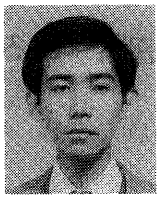
参考文献

1) 富田眞治：並列計算機構成論，昭晃堂（1986）．
2) Rashid, R. F.: Threads of a New System, *UNIX Review*, Vol. 4, No. 8, pp. 37-49 (1986).
3) 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).
4) 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).
5) 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).
6) 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，CPSY 91-29 (1991).
7) 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).
8) 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).

（平成4年7月16日受付）
（平成5年1月18日採録）

中山 泰一（正会員）

1965 年兵庫県生．1988 年東京大学工学部計数工学科卒業．1993 年 3 月同大学院工学系研究科情報工学専攻博士課程修了．工学博士．1993 年 4 月より電気通信大学情報工学科助手．オペレーティング・システム，並列・分散システムなどに興味を持つ．日本ソフトウェア科学会会員．

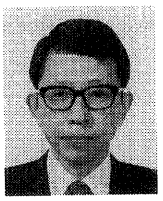


永松 礼夫（正会員）

1980 年東京大学工学部計数工学卒業．1982 年同大学院修士課程修了．1984 年同大学院博士課程単位取得退学．同年，同大学計数工学科助手，現在に至る．並列処理，計算機アーキテクチャ，分散処理，オペレーティング・システムに興味をもつ．日本ソフトウェア科学会，計測自動制御学会，ACM 各会員．

出口光一郎（正会員）

1976 年東京大学大学院計数工学修士修了．同年より東京大学工学部助手，講師を経て，1984 年山形大学工学部情報工学科助教授，1988 年東京大学工学部計数工学科助教授，現在に至る．この間，1991～1992 年米国ワシントン大学客員準教授．コンピュータビジョン，並列コンピュータの研究に従事．工学博士．計測自動制御学会，形の科学会，IEEE 各会員．



森下 巖（正会員）

1934 年生．1957 年東京大学工学部応用物理学科計測工学コース卒業．同年東レ(株)入社．計装制御システムの設計に従事．1966 年東京大学工学部計数工学科助教授，1980 年同教授，現在に至る．工学博士．1973 年 SRI 人工知能研究センタ滞在．パターン認識，信号処理，画像処理，マルチプロセッサシステムなどの研究に従事．著書「マイクロコンピュータのハードウェア」（岩波書店），「マイクロコンピュータの基礎」（昭晃堂），「信号処理」（計測自動制御学会）など．IEEE，計測自動制御学会，電子情報通信学会，電気学会各会員．