システムソフトウェアと
オペレーティング・システム 61−13
（１９９３．８．１８）

分散要求管理を用いるアクティビティ方式並列実行機構

本橋 健†, 中畑 昌也†, 中山 泰一☆, 永松 礼夫†,
出口 光一郎†, 森下 巖†

† 東京大学工学部　☆ 電気通信大学電気通信学部

細粒度の並列処理可能なタスクを実行するための並列実行管理機構であるアクティビティ方式において、プロセサ台数や並列プログラムのタスク数が多くなる場合でも効率が低下しないスケジューリング手法を検討した。タスク生成要求キューであるアクティビティキューへのアクセス競合を解消するためには、個々のプロセサがアクティビティキューを持ち、通常は自分のキューのみを操作する分散方式を採用する。アクティビティ等の保存領域の増加を解消するためには、個々のアクティビティキューの取り出し方式をLIFOにする。また、個々のプロセサのアクティビティキューが空になった場合には、他のアクティビティキューからFIFOで取り出すことで仕事の補充を効率よく行なうことができる。これらの手法を実験で検証し、プロセサ自身のアクティビティキューからはLIFOで取り出すが、他のプロセサのアクティビティキューからはFIFOで取り出す方式がもっとも効率よく行なわれることを示した。

An Activity-Based Parallel Execution Mechanism Using Distributed Activity Queues

Takeshi Motohashi†, Masaya Nakahata†, Yasuichi Nakayama☆,
Leo Nagamatsu†, Koichiro Deguchi† and Iwao Morishita†

† Faculty of Engineering, University of Tokyo
☆ Faculty of Electro-Communications, University of Electro-Communications

This paper describes an activity-based mechanism for high performance parallel execution of a large number of fine grain tasks on a shared memory machine with large number of processors. In this mechanism, we employ distributed activity queues to reduce access contention to a single activity queue. Each light-weight process has its local activity queue for management of task execution. A local queue access is done by the LIFO order to reduce memory consumption for the storage of activities generated. When a process finds its local queue empty, it tries a remote access to another process's queue. This access is done by the FIFO order. Simulation studies have been done for the four cases of queue access orders. The results show that the best performance is obtained by the combination of LIFO for the local queue access and FIFO for the remote queue access.

1 はじめに

並列計算機において並列実行可能な多くのタスク (並列実行単位) を同時に実行する方法のひとつとして軽量プロセスを使用するものが広く用いられてきた。しかし、軽量プロセスを用いてもその生成と消滅、切替には時間がかかり、またメモリも消費するため、最近では軽量プロセスの生成・切替等を必要最小限にする手法の提案がなされている [1][2]。

我々は、共有メモリ型並列計算機において多数の細粒度のタスクを実行する際に、あらかじめプロセサ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おさえる手法であるアクティビティ方式を提案した [3][4]。応用プログラムからのタスク生成要求は、アクティビティと呼ばれる手続きと引数の組のかたちで保存され、順次軽量プロセスに割り当てられ実行される。

アクティビティ方式では軽量プロセスの生成・切替を極力少なくするため、メモリ使用量やスケジューリングコストを低減できる利点がある。しかしタスク数やプロセサ台数が非常に多くなると、アクティビティ等の保存領域の増大やそれを保存するキューへのアクセス競合によるスケジューリングコストの増加が問題になる。

アクセス競合を解消するには、アクティビティキューを各プロセサごとに用意すればよい。メモリ使用量を軽減するには、タスクツリーの実行を進めていく際にプロセサ台数までは幅優先ですばやくタスクを分配し、その後は深さ優先でタスクのひろがりをおさえるのが望ましい。ただし、プロセサのアクティビティキューが空になった際には、他のキューからの仕事の補充が効果的に行なわれなければならない。

上記のタスクスケジューリングを実現する方法として、通常は自分のキューから LIFO 順序でアクティビティを取り出し、自分のキューが空となった場合には他のキューから FIFO 順序でアクティビティを取り出す分散キュー管理方式が考えられる。

実行開始時は空いたプロセサに順次アクティビティを渡していく。各プロセサは自分で生成したアクティビティを自分のキューに保存し、そのキューから LIFO で取り出すため、深さ優先でタスクの実行を行なうことができる。

また、自分のキューにアクティビティがなくなり他のキューから取り出す際、FIFO で取り出すことでより過去に生成されたアクティビティを得られる。タスクツリーが極端に偏っていない場合、より過去に生成されたアクティビティからはより多くのタスクが生じると考えられる。一度の取り出しでできるだけ多くの仕事を得られるようにすることで他のキューから取り出す回数を減らすことができる。

自分のキューから LIFO、他のキューからは FIFO で取る上記の方式は並列関数型言語を対象とする Lazy Task Creation[5] で採用されている。我々はこの方式をアクティビティ方式に適用することを考え、分散アクティビティキューの取り出し方式が性能に与える影響を評価することにした。共有キューと分散キューの取り出し方式を LIFO と FIFO の組合わせとしてその効果を調べた。その結果、分散アクティビティキューにおいて自分からは LIFO、他からは FIFO で取り出した際に、スケジューリングコストやメモリ使用量がもっとも低減できることが確認された。

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 [3] では並列実行可能なタスクの発生とタスクの実行を別々に行なう。

応用プログラムから並列実行可能なタスクの生成が make_child() プリミティブによって要求されると、そのタスクをいったん要求時の形 (アクティビティ) のままキューにつないでおく。そして、空いた軽量プロセスにそのアクティビティをスケジュールすることでタスクは実行される。アクティビティは実行すべき手続きとその引数の組で構成されている。

アクティビティ方式の実行機構としては、プ

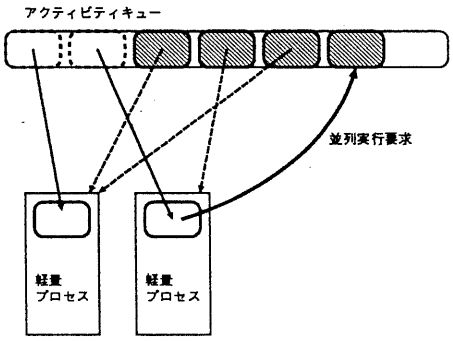

図 1: アクティビティ方式

ロセサ数と同数の軽量プロセスをあらかじめ用意しておく。軽量プロセスはアクティビティスケジューラからアクティビティをひとつ受け取り、実行する。タスクの実行が完了したら、再びアクティビティスケジューラから新たなアクティビティを受け取る。

こうして、タスク生成の要求はいったんアクティビティという形で保存され、そのアクティビティを軽量プロセスが順次実行することで、タスクの実行が行なわれる (図 1)。

応用プログラムがタスク生成要求をしたときにすぐに軽量プロセス生成を行なう場合と比較して、アクティビティ方式は軽量プロセス生成にかかる時間を短縮できる。また、空いた軽量プロセスにアクティビティを順次割り当てることでタスク切替のコストも小さい。また、軽量プロセスをひとつ用意するのに比べて、アクティビティが占めるメモリ量は非常に小さいため、メモリの低減を図ることができる。

2.2 「遺言」方式

アクティビティ方式では、タスクの実行中に同期等で中断すると、実行していた軽量プロセスをいったん退避して新たな軽量プロセスを生成する。プロセサはその軽量プロセスを用いて別のタスクの実行を行なう。そのためタスク間の同期待ち合わせによって中断するような応用プログラムでは顕著な効率向上が望めない。

並列プログラムを記述する際によく取られる

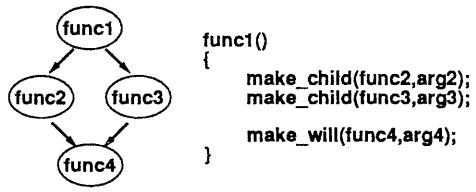

図 2: 「遺言」の記述と動作

手法として fork-join 型の応用プログラムがある。あるタスクが生成したすべての子タスクについて、最後に終了した子タスクの後に後処理を行なうという記述をする。fork-join 型応用プログラムでは親子間のタスクで同期を取る必要がある。

アクティビティ方式の改良である「遺言」方式を用いると、親子関係による同期待ち合わせに対して、従来のアクティビティ方式と同様に効率が上がることが示されている [4]。

「遺言」方式では、ある親タスクは、生成したすべての子タスクの実行完了後に行なう後処理 (「遺言」) を残し、そのまま実行を終了する。生成された子タスクのうち一番最後に終了したものが、その「遺言」を取り出し実行する。親タスクは子タスクの実行完了を待つ必要がないため、タスクの中断は起きない。

「遺言」も手続きと引数との組となっており、親タスクの最後で make_will() プリミティブを用いて指定される (図 2)。

「遺言」方式をインプリメントするために、アクティビティ方式に家系記述子 (Family Tree Descriptor, 以下 FTD とする) という構造体を導入した。これは、生成されたすべてのタスクの親子関係と「遺言」を記録するためのもので、すべてのタスクはかならず対応する FTD を持つ。

「遺言」方式を導入するとアクティビティのほかに FTD の保存領域が必要となるが、これらは軽量プロセスが必要とするメモリ量よりも非常に小さい。またアクティビティや FTD の操作によるタスク切替コストも軽量プロセス切替より小さなものである。よって、「遺言」方式は親子間の待ち合わせをする応用プログラム

でもアクティビティ方式の利点を損なうことなく実行できる。

3 タスク生成要求の分散管理

3.1 今までのアクティビティキュー管理方式の問題点

アクティビティ方式およびその改良である「遺言」方式を使用すると、軽量プロセスの生成・切替を最小とするためメモリ使用量やスケジューリングコストを低減できる利点がある。しかし、それでもタスク数やプロセサ台数が非常に大きくなると、メモリ使用量の増加やアクセス競合が起きるようになる。

タスク数が非常に大きくなるとそれにともなってアクティビティやFTDも増加し、より多くの保存領域が必要となる。また、プロセサ台数が増えると、アクティビティを保存しているキューへのアクセスが競合を起こし、スケジューリングコストが増加する。これらの問題はキューの管理方式によるものであり、タスク生成要求であるアクティビティの管理を共有FIFOキューとして行なっていたためである。

FIFOキューは最初に保存されたものが最初に取り出される。タスク実行順序としてはより過去に生成されたものが先に実行されることになる。このようなFIFOキューを使用すると、タスク実行順序は幅優先で行なわれる傾向にある(図3)。

幅優先のタスク実行順序を行なうと、同時実行可能なアクティビティの最大数はタスクツリーの幅程度になる。問題の規模が大きくなるとこの数は莫大になり、アクティビティのひとつひとつの大きさが小さくても保持しきれなくなる。

また「遺言」方式においては、タスクツリーの各タスクに対応するFTDを持つ。FTDは対応するタスクの子タスク(子孫タスク)がすべて終了し、かつ「遺言」の実行が完了するまで解放することができない。よって、タスクツリーの末端のタスクが実行完了されないかぎり、その祖先のタスクに対応するFTDはすべて保持されることになる。幅優先のタスク実行順序では、最大すべてのタスクに対応するFTDを保持することになるため、莫大なメモリを必要とすることになる。

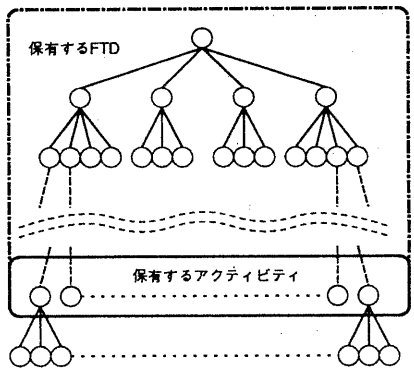

図3: 幅優先とアクティビティ,FTD

また、共有キューにおいて複数のプロセサが同時にキューを操作しようとするときはキュー操作の一貫性を保たねばならない。そのもっとも簡単な解決法としてキューをロックする方法を採用した。しかし、ロックを取るのに失敗した場合にプロセサは待たされ、プロセサ数が多くあるとこの無駄は顕著に現れる。

3.2 FIFOキューとLIFOキュー

アクティビティキューの取り出しをFIFO方式で行なった場合、前述のようにアクティビティやFTDに対して莫大な保存領域が必要になる。このキューの取り出し順序をLIFOにすると、必要なメモリ量は小さいものになると考えられる。

LIFOキューでは最後に保存されたものが最初に取り出される。タスク実行順序としてはもっとも最近生成されたものが先に実行されるため、LIFOキューを使用するとタスク実行順序は深さ優先で行なわれる傾向にある。

深さ優先では自分の子タスクを先に実行するため、生成されるタスクはタスクツリーのある縦一列の領域にのびる(図4)。よって、幅の非常に広いタスクツリーを持つ応用プログラムであっても保持しなければいけないアクティビ

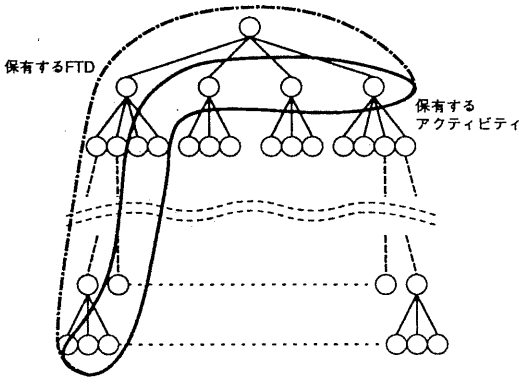

図 4: 深さ優先とアクティビティ,FTD

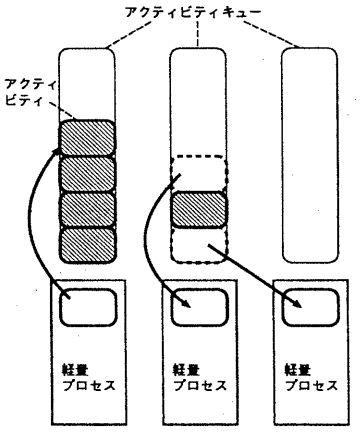

図 5: 分散アクティビティキュー

ティの最大数は小さくなり、保存領域が少なくて済む。

また深さ優先探索ではタスクツリーの末端のタスクを先に実行しようとするため、末端のタスクが実行完了するとそのタスクの FTD は解放できる。末端の子タスクを生成した親タスクの FTD も、その子タスクがすべて実行完了し、かつ「遺言」がなければ解放できることになる。よって、必要な FTD の最大数もおさえることができ、FTD の保存領域も小さく済むようになる。

3.3 分散アクティビティキューを用いる方法

タスク生成要求を保存するアクティビティキューへのアクセス競合の問題は、キューを個々のプロセサごとに持たせる分散キュー方式とすることで解決できる。分散キューにより特定のキューに集中してアクセスすることがなくなり、アクセス競合が起きにくくなる。

キューを分散させた場合、各プロセサが生成したアクティビティはそのプロセサの持つアクティビティキューにつながれることになる。タスクの実行が完了して新たなアクティビティを取り出すときも、同様に個々のキューを用いる。

自分のキューに保存しているアクティビティがなくならない限り、各々のプロセサは自分のキューのみを操作する。キューにアクティビティがなくなった場合にのみ他のプロセサのアクティビティキューを探しにいくため、キュー操作におけるアクセス競合は低減できる。

前述のように、キューの取り出し方式に関して FIFO、LIFO の 2 種類がある。分散キューの場合、自分のキューから取り出す場合と、他のキューから取り出す場合のそれぞれに対して FIFO、LIFO の 2 種類、計 4 通りの取り出し方が考えられる。そのなかで、自分のキューからは LIFO、他のキューからは FIFO で取り出す方式がもっともよいと考えられる (図 5)。

その理由としては、まず、個々のアクティビティキューからの取り出し方式を LIFO にすると、生成タスク数が増加してもアクティビティや FTD の保存領域の増加は小さいものとなるためである。自分からのキューの取り出しに LIFO を使用することで、分散キュー方式における各キューのアクティビティの最大数をおさえることができる。

また、他のアクティビティキューからは FIFO で取り出すようにすると、他から取りにいく回数がより少なくなると考えられる。タスクツリーの形が非常に不均等でないのなら、より過去に生成されたアクティビティのほうがより多数のアクティビティを生成することが多い。他から取り出したアクティビティによって多くのアクティビティを生成してそれを実行すれば、

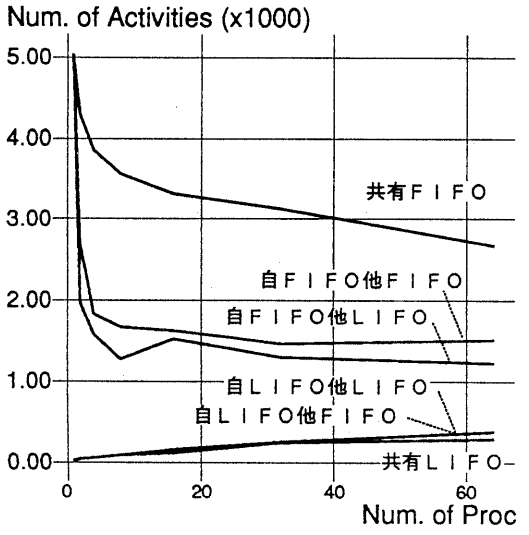

図 6: アクティビティ最大数

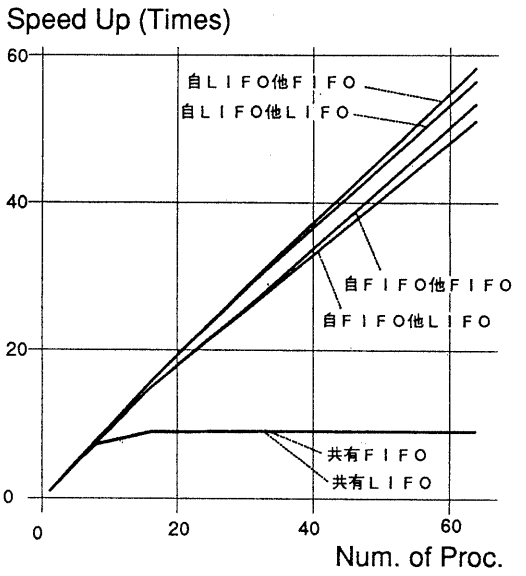

図 7: プロセサ数とスピードアップ

次に他のアクティビティキューから取り出すまでの間隔が長くなる、と考えられるためである。

他のアクティビティキューから取り出す回数が少なくなれば、それだけキューへのアクセス競合も少なくなるはずであり、スケジューリングコストが低減できると考えられる。

4 実験と評価

以上の議論をもとに実験を行なった。実験はSPARC のプロセサと理想共有メモリのシミュレータ上で行なった。プロセサ台数は 1 台から 64 台までを設定した。

問題としては、枝刈りなしで 8 都市の TSP (Travelling Salesman Problem) を使用した。タスク生成要求の管理方式として、共有キューと分散キューの各々において、FIFO、LIFO の取り出し方式を調べることとする。

キューのアクセスに際しては、操作するキューにロックをかけることとする。

まずは、必要とするアクティビティの最大数を見る (図 6)。図の横軸はプロセサ台数であり、縦軸は保有するアクティビティの最大数である。共有 LIFO、および分散キューでの自分のキュー取り出しを LIFO にする方式では、最大数は小さいものとなる。これらはプロセサ数が増加するとそれにともなってアクティビティ最大数も増加するため、並列度を保つことができると考えられる。

ここでは示さないが、FTD の最大数もアクティビティのそれと同様な割合を示す。

次に、プロセサ台数を増やした際のスピードアップを見る (図 7)。共有キュー方式では FIFO、LIFO ともにプロセサ数を増やしてもスピードアップは頭打ちになるが、分散キュー方式ではプロセサ数とともにスピードアップしていることがわかる。共有キュー方式での頭打ちの主要因はキューへのアクセス競合のためである。

分散キュー方式のなかで性能を比べると自分のキューからは LIFO、他のキューからは FIFO で取り出す方式がもっとも良いことがわかる。この主要因は、他から取り出す回数が少ないので、キュー探索コストやアクセス競合によるオーバーヘッドを低減できるためである。

他のプロセサのアクティビティキューから取り出した回数をプロセサ台数を変えて比較する

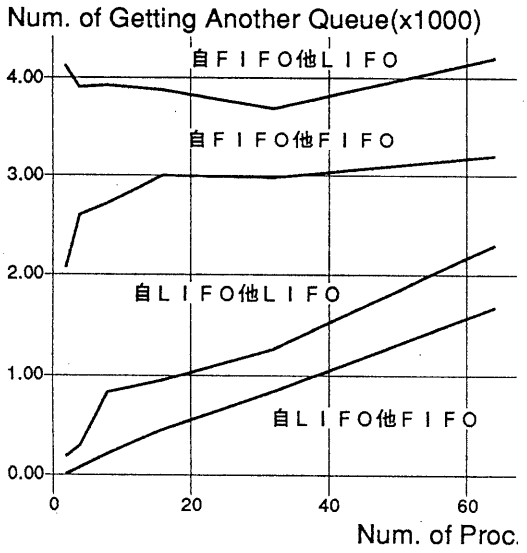

図 8: 他のキューから取った回数

(図 8)。プロセサ自身のアクティビティキューから FIFO で取る方式は、他のキューからの取り出し方式が FIFO か LIFO かにかかわらず、常に取り出す回数が多い。プロセサ自身のキューから LIFO で取る際は、プロセサ台数が小さいときには回数は少ない。プロセサ台数が増えるに従って取り出し回数が増加するが、他からのキューの取り出し方式が FIFO のときに、より少ないことがわかる。

プロセサ間をまたがったタスクの移動がどのくらいの頻度で、また、どのくらいのタスク粒度で起こったかを見るため、各構成について図 9のような実行記録図を作成した。横軸は処理開始から終了までの時間であり、縦軸はキューから取られたタスクがタスクツリーのどの深さであったかを示す。TSP 問題のように、どの leaf も同じ深さにあるようなタスクツリーである場合、ノードの深さとそれ以下にあるノードの総数は対応し、そのノードから起動される仕事の総量のよい指標となると思われる。そこで、この深さをキューから 1 回のアクセスで取り出された仕事の粒度の目安とした。図中で、ひとつの点が他プロセサのキューからの 1 回の取り出しを示している。

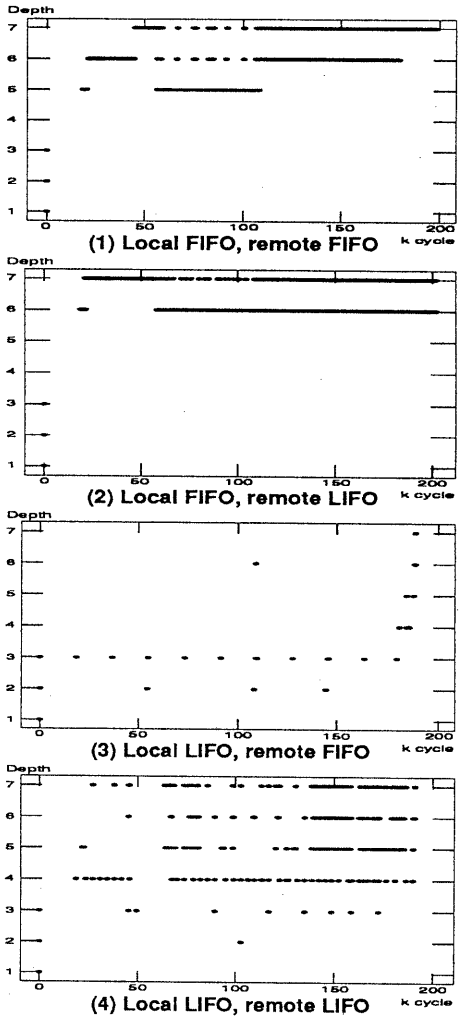

図 9: 他からの取出しの発生する時刻と深さ

この結果から、

(1) 自プロセサからは FIFO で、 他プロセサからは FIFO で取り出す場合: アクセス回数は多く、深さの深いもの (粒度の細かいもの) が多い。

(2) 自 FIFO、他 LIFO: アクセスは多く、深さは 6 と 7 の深いものだけである。

(3) 自 LIFO 他 FIFO: アクセスは少なく、深さも浅いものが中心である。仕事が終了

する間際だけ、細かい負荷バランスをとるため深いアクセスが現れる。

(4) 自 LIFO 他 LIFO: アクセス量は中くらい。いろいろの深さのものがアクセスされるが、深いものが多めである。

ということが読みとれる。したがって、我々の理想とする性質は (3) の構成の時に得られることが確認された。

またここでは示さないが、枝刈りのある TSP でも同様の結果が得られている。

5 検討

分散キュー方式では他のキューからアクティビティを探すコストがかかるため、プロセサ数が増えるとそれだけ探索のコストが増加するという問題点がある。現在は、各プロセサが自分のとなりの番号から順々に調べていく方式を取っているが、これをもっと効率的に行なうことで他のキューへのアクティビティ探索コストが減少し、システム時間が短縮できると考えられる。

効率的にアクティビティキューを探索する方法としては、個々のプロセサのキューにあるアクティビティの存在のヒントを共有する場所に用意しておく方法が考えられる。ヒントとしては、キューにアクティビティがあるかどうかのフラグや、キューにあるアクティビティの数が考えられる。

また、分散キューのほかにも共有のアクティビティキューを用意しておき、自分のキューにアクティビティがなくなった場合には共有キューに探しにいく方法も考えられる。各プロセサ個々のキューと共有キューのみにアクセスするので、個々のアクティビティキューにはロックをかける必要がなくなるためシステム時間の短縮が行なえる。

6 おわりに

アクティビティ方式において、タスク生成要求キューを個々のプロセサごとに分散して管理し、自分のキューからは LIFO、他のキューからは FIFO で取り出す方式を適用してその性能を評価した。本方式を使用すると、生成するタスク数やプロセサ数が非常に多くなっても、使用するメモリ量やキューへのアクセス競合によるオーバーヘッドを小さくできることが確認された。

今回、実験はシミュレータ上で行なったが、我々の研究室では SPARC プロセサを 4 台用いた共有バス共有メモリ型並列計算機を試作しており [6]、実機を用いた実験も行なう予定である。

参考文献

[1] Anderson, T. et al.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM Trans. Computer Systems*, Vol.10, No.1, pp.53-79(1992).

[2] 新城, 清木: 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).

[3] 田胡, 檜垣, 森下: 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).

[4] 中山, 永松, 出口, 森下: 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, pp.985-993 (1993).

[5] 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).

[6] 中畑, 本橋ほか: アクティビ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).