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[†]

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生成要求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，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，個々のアクティビティキューの取り出し方式にLIFO順序を採用する．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順序で取り出すことで仕事の補充を効率よく行う．シミュレータを用いて実験を行い，上記の方式により効率のよいスケジューリングが実現でき，またメモリ使用量も小さくできることを確認した．

An Activity-Based Parallel Execution Mechanism Using Distributed Activity Queues

Takeshi Motohashi,[†,*] Masaya Nakahata,[†,**] Yasuichi Nakayama,[††]
Leo Nagamatsu,[†] Koichiro Deguchi[†] and Iwao Morishita[†]

This paper describes an activity-based scheduling mechanism for high performance parallel execution of a large number of fine grain tasks on a shared memory machine with a 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 is accessed by the LIFO order to reduce memory consumption for the storage of activities. When a local queue is empty, the process tries an access to remote process's queue. This access is done by the FIFO order to redure remote queue accesses. Experimental simulations have been done for various combinations of the queue structure and the accessing order mechanisms. The best performance was obtained by the combination of LIFO for the local queue access and FIFO for the remote queue access.

1. はじめに

多数のプロセサを有する共有メモリ型並列計算機において，多数の細粒度な並列処理単位（タスク）を実行させる際，効率の良い並列実行管理機構を用意することが性能を向上させるための重要な課題である．現在，システムソフトウェアの分野において多くの試みがなされている[1]．

タスクを同時に実行する手法として広く用いられている軽量プロセス生成方式は，従来のUNIXのプロセス生成方式と比較するとコストが小さい．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．そこで，軽量プロセスの生成・切替のコストを軽減する実現方式の提案がなされている[2),3)]．

軽量プロセス生成・切替を極力少なくする手法として，我々は，共有メモリ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アクティビティ方式を提案した[4,5)]．

アクティビティ方式では，タスク生成要求をすぐに

† 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
†† 電気通信大学電気通信学部情報工学科
Department of Computer Science, The University of Electro-Communications
* 現在，日本電信電話(株)
Presently with Nippon Telegraph and Telephone Corporation
** 現在，(株)日立製作所
Presently with Hitachi, Ltd.

実行せず，アクティビティという形でいったんキューに保存する．そして各プロセサに一つずつ用意された軽量プロセスは一つのタスクの実行を完了したらキューからアクティビティを取り出して新たなタスクの実行を開始する．このようにしてタスク実行の際に軽量プロセスの生成・切替を行わないことでスケジューリングコストを小さくする．しかし，プロセサ台数が多くなると共有キューであるアクティビティキューを操作することによるアクセス競合が発生する．また，タスク生成要求が多数発生するとアクティビティのキューへの保存に多量のメモリを消費するという問題があった．

この二つの問題を解決するには，キューの分散化によるアクセス競合の低減が必要であり，また，後に説明するようないわゆる深さ優先のスケジューリングによる保有アクティビティ数の低減が必要である．ここで提案する方式では，キューの分散化としてキューを個々のプロセサに一つずつ用意することにする．また，深さ優先のスケジューリングを行うにはキューからの取り出し方式を LIFO (Last In First Out) 順序とすればよい．

ただし，キューを個々のプロセサに一つ持たせた場合，あるキューが空になった際に他のキューからアクティビティを補充する必要がある．そしてそのような補充は効果的に行わねばならない．そのため，他のキューからのアクティビティの取り出しを FIFO (First In First Out) 順序で実行する．FIFO 順序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が得られる．

ネストした fork-join 型の応用プログラムではより過去に生成されたアクティビティほど，実行の際に fork を繰り返すことで多くのアクティビティを生成すると考えられる．したがって，過去に生成されたアクティビティを取り出すほうが結果として一度の取り出しでより多くのアクティビティを得られることが期待できる．

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[6] でも採用されており，一般に有効な方法として知られている．

本論文では，分散アクティビティキューの管理において取り出し方式が性能に与える影響をシミュレーション実験で評価し，このような方式が効果的であることを確認した．評価には，共有キューと分散キューの取り出し方式に対して LIFO 順序と FIFO 順序を使用し比較した．その結果，分散アクティビティキューにおいて自分のキューからは LIFO，他のキューからは FIFO で取り出す本方式がスケジューリングコストやメモリ使用量をもっとも低減できることが確認された．

以下，まずアクティビティ方式の概要について述べる．次に，上で述べたスケジューリングの問題を整理し，その解決法としての本方式の詳細を述べる．そして本方式の評価のためのシミュレーション実験について述べ，結果を示す．

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[4] は並行実行可能なタスクの生成要求とタスクの実行を独立に処理することにより効率向上を図る並列実行機構である．

アクティビティ方式の実行機構は，あらかじめ軽量プロセスを各プロセサに一つずつ割り当てて用意しておく．応用プログラムから並列実行可能なタスクの生成が要求されると，いったん要求の形のままキューにつないでおく．このタスク生成要求の情報をアクティビティと呼ぶ．そして，タスクを実行していない軽量プロセスが生じるとキューからアクティビティを取り出すことでタスクは実行される（**図 1**）．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．キューに入れられたアクティビティはいつ実行開始されてもよい．

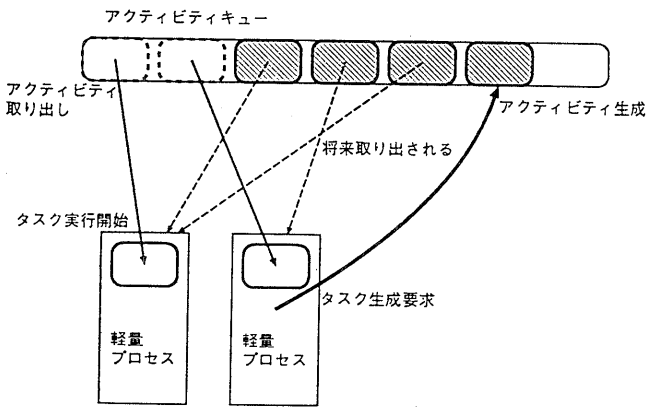


図 1 アクティビティ方式によるスケジューリング
Fig. 1 Scheduling in Activity-based processing mechanism.

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う方式と比較すると，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により，タスク切替のコストも小さくできる．さらに，軽量プロセスを一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，メモリ使用量の低減を図ることもできる．

2.2 「遺言」方式での改良

アクティビティ方式は汎用の並列実行機構であるが，基本のアクティビティ方式のままではタスク間の同期待ち合わせによってたびたび中断する応用プログラムで軽量プロセス生成・切替を行わざるを得ないため，顕著な効率向上が望めない場合がある．そこで，同期待ち合わせのうち，並列プログラムを記述する際よく採られる手法である fork-join 型に現れる親タスクと子タスク間の同期待ちについて，アクティビティ方式の利点を損なわずに効率向上を行うことのできる「遺言」方式を提案した[5]．

並列プログラムを記述する際，問題を複数の問題に分割し，それらを並行実行させる手法がしばしば使用される．分割された複数の問題をさらに再帰的に分割するように記述し，さらに分割された複数の問題がすべて実行完了した後に，後処理を行うことが多い．このような問題の記述法をネストした fork-join 型と呼ぶ．ネストした fork-join 型の応用プログラムでは再帰的な分割をしていくことで並列実行可能な問題数は非常に大きなものとなる．この場合，分割された問題の依存関係は分割をした親と分割された子との間でのみ発生する．

「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う後処理（「遺言」）を残し，その親タスクの実行を終了する．生成された子タスクのうち一番最後に終了したものを実行した軽量プロセスがその「遺言」を取り出して実行する．親タスクは子タスクの実行完了を待つ必要がないため，同期待ち合わせによる軽量プロセスの中断は起きない利点がある．

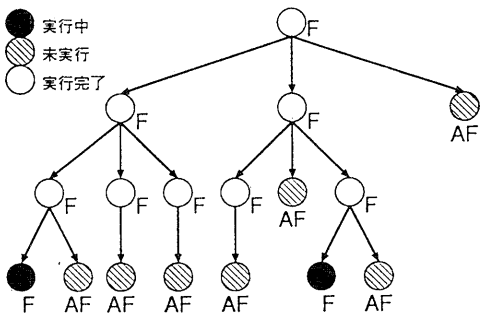
「遺言」方式の実装の際に，家系記述子（Family Tree Descriptor，以下 FTD とする）という構造体を導入した．これは，生成されたすべてのタスクの親子関係と「遺言」を記録するためのもので，すべてのタスクは必ず対応する FTD を持つ．このためアクティビティ以外にも FTD の保存領域が必要となるが，両者を合わせても軽量プロセスが必要とするメモリ量よりはるかに小さい．

FTD はタスクの生成要求の際に生成され，子タスクの生成数や子タスクの実行完了数，そして「遺言」と呼ばれる後処理の情報を管理する．すべての子タスクが実行完了し「遺言」も実行完了するまで，FTD の領域は解放されない．アクティビティはキューから取り出されタスクの実行が開始されると領域を解放できるのに対して，FTD は，実行完了していないタスクに対応する FTD と，その祖先のすべての FTD を保有する必要がある（**図 2**）．

3. 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」を付け加えた方式を使用すると，軽量プロセスの生成・切替を最小とするのでスケジューリングコストやメモリ使用量を低減できる利点がある．しかし，それでもプロセサ台数やタスク生成要求数が非常に多くなると，キュー操作のアクセス競合によるオーバーヘッドとメモリ使用量の増加が起きるようになる．

プロセサ台数が増えると，アクテビティを保存しているキュー操作のため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．また，タスク生成要求数が非常に多くなるとそれにともなって保有すべきアクティビティや FTD も増加し，より多くのメモ



A：保有するアクティビティ
F：保有する FTD

図 2 応用プログラム実行中のある時刻に保有されているアクティビティと FTD．FTD はアクティビティよりも多くの個数を保有する必要がある．

Fig. 2 Activities and FTD's being stored at moment in executing a program. The number of FTD is always larger than that of Activity.

リが必要となる．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 キューとして行っていたためであった．

キューの取り出し方式を FIFO 順序にすると，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，タスク実行順序はタスクツリーで考えると，タスクの分布が横方向に広がる，いわゆる幅優先で行われる傾向にある．

幅優先のタスク実行順序である場合に，ネストした fork-join 型の応用プログラムでは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，一つ一つのアクティビティの大きさが小さくても保有しきれなくなる（**図 3**）．

2.2 節で述べたように，アクティビティ方式の改良である「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 はすべて保有されることになる．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 を保有することになるため，メモリ使用量も大きなものになる．

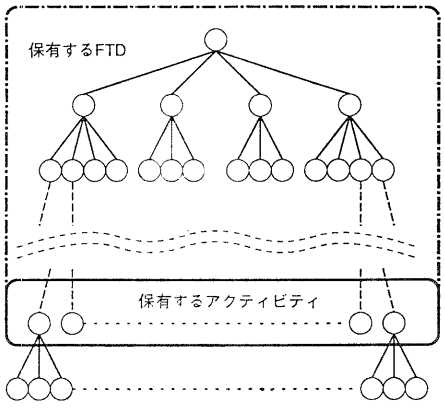


図 3 共有 FIFO キューを使用した場合に保有しなければいけないアクティビティと FTD
Fig. 3 Activities and FTD's which must be stored for the FIFO order management of an Activity queue.

4. 提案する分散アクティビティキューを用いる方法

4.1 分散アクティビティキューを用いる方法

ここでは，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する．

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キューにつながれることになる．新たなアクティビティを取り出すときも，同様に自分のキューを用いる（**図 4**）．

自分のキューに保有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．

4.2 分散キューにおける取り出し方式

アクティビティキューの取り出し方式を FIFO 順序で行った場合，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となる．このキューの取り出し方式を LIFO 順序にすると，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ことになる．LIFO キ

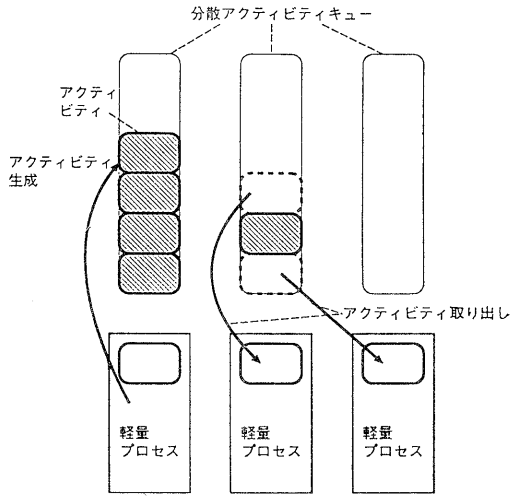


図 4 分散アクティビティキューの概念図
Fig. 4 Schematic diagram of distributed Activity queues.

ューを使用するとタスク実行順序はタスクツリーの横方向のものではなく，下方のものが先向に実行される，いわゆる深さ優先で行われる傾向にある．この場合，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 5**)．よって，ネストした fork-join 型のような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有しなければいけないアクティビティの最大数は小さくなり，保存領域が少なくて済む．

また深さ優先で行うと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，さらに「遺言」の実行を終えると解放できることになる．よって，必要な FTD の最大数も小さくなり，FTD の保存領域も少なくて済むようになる．

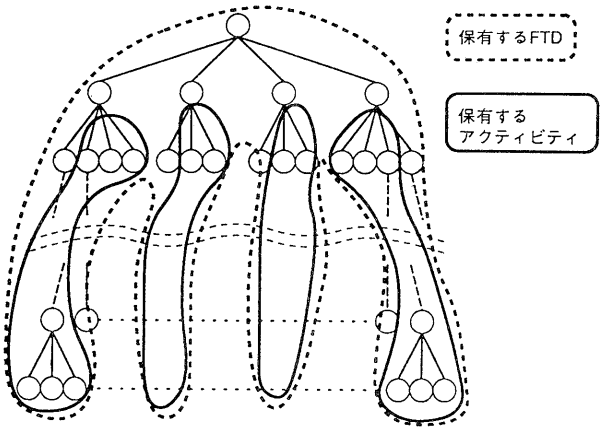


図 5 分散キューを LIFO で取り出すときに保有するアクティビティと FTD
Fig. 5 Activities and FTD's stored for the LIFO order management and distributed queue mechanism.

4.3 取り出し方式の検討

分散キューにおけるキューの取り出し方式を各々のプロセサの持つキューから LIFO 順序で取り出すようにする．また，他のプロセサの持つアクティビティキューからは FIFO の順序で取り出すようにすると，他のキューから取り出す回数がより少なくなると考える．タスクツリーの形が極端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からである．すなわち，他のキューから取り出したアクティビティが多くのアクティビティを生成してそれを実行すれば，次に他のキューから取り出すまでの間隔が長くなる．他のキューから取り出す回数が少なくなれば，それだけキュー操作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．

このように他のキューからの取り出し方式と得られる仕事の総量とは密接にかかわっている．**図 6** は応用プログラムが実行を開始してから終了するまでの間に，他のキューからアクティビティを取り出す場合，どのような取り出しがなされれば望ましいかを概念図として表したものである．

横軸が他のキューからアクティビティが取り出された時刻，縦軸はそのアクティビティによって実行されるタスクのタスクツリーの根元からの深さであり，他のキューからアクティビティを取り出すごとに，そ

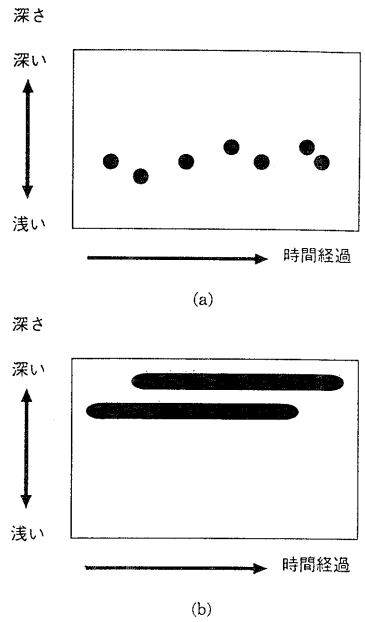


図 6 他のキューからの取り出し時刻と深さの概念図 (a) 望ましい例 (b) 望ましくない例
Fig. 6 Typical event charts of remote queue accesses. Timings and depth distributions of access is shown for (a) a desirable case, and (b) an undesirable case.

の時刻とそのタスクの深さを表す点をプロットしたものである．上記のことから望ましい状態は，他のキューからの取り出しである点が少なく，深さの浅いと

ころに点が存在する場合である（図6（a））．逆に望ましくない例としては点が多く，かつ深さの深いところに点が密集する場合である（図6（b））．

図6（a）のような場合において，アクセス競合が少ないため，効率が良くなると考える．

5. 実験と評価

5.1 実験環境

以上の議論を検証するために実験を行った．実験はSPARCのプロセサと1メモリアクセスが1クロックで行える理想共有メモリのシミュレータを用いた．プロセサ台数は1台から64台までを設定した．

応用プログラムには，ネストしたfork-join型として記述したTSP（Travelling Salesman Problem）を使用した．実験に際してTSP問題は8都市の枝刈りのないものを使用した．

タスク生成要求の管理方式として，全体で一つのキューを共有する（共有キュー）方式とそれぞれのプロセサごとにキューを持つ（分散キュー）方式について取り出し方式を含めて効率を調べた．

分散キューでは自分のキューおよび他のキューからのアクティビティの取り出しをそれぞれFIFO, LIFO（自LIFO他FIFOなどと表す）とする4通りの取り出し方式のものを用意した．また，共有キューではFIFOとLIFOの2通りの取り出し方式のものを用意した．また，キューのアクセスに際しては，操作する際キューにロックをかけることとした．

5.2 実験結果

まず，**図7**にプロセサ台数を増やした際のスピードアップについて1台で行った際の全実行時間を1として調べた結果を示す．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．キューへのアクセス競合のないプロセサ数1台のときは，全実行時間の4割がシステム時間となり，キューにロックを行う操作はその1/4となった．さらにキューへのアクセス競合が起きるプロセサ数64台の場合では，キューのロック待ちの時間は全体の85%になってしまう．共有キュー方式でのスピードアップの頭打ちとなる主要因はキュー操作のアクセス競合のためであり，分散キュー方式ではこれが解決できることが確かめられた．

図7において分散キュー方式のなかでのスピードアップ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（自LIFO他FIFO）がもっとも良いことがわかる．この主要因は，他のキューから取り出す回数が少ないので，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と考えられる．

取り出し回数について調べるため，**図8**に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した結果を示す．自分のキューから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かおよびプロセサ台数にかかわらず，取り出す回数が常に多い．自分のキューから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．自分のキューからLIFOで取る方式のうち，他からのキューの取り出し方式がFIFOのときに，取り出し回数がより少ないことがわかる．

図9は他のキューからアクティビティを取り出した回数と，取り出したことによって移動した仕事の総量を見たものである．この図は，図6で概念図として表したものの実験で得られた結果である．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アクティビティから実行されるタスクがタスクツリーのどの深さであったかを示す．プロセサ台数は4

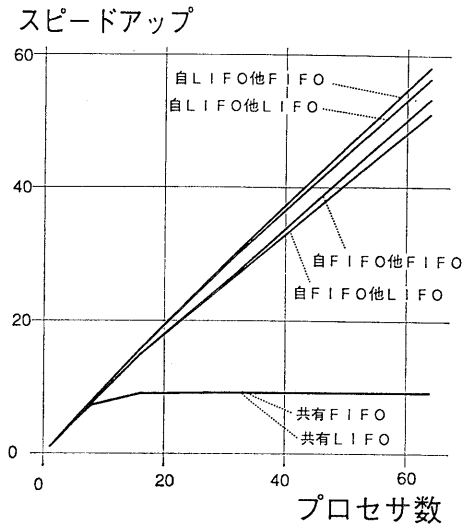


図7 キュー構造と取り出し方式の違いによるプロセサ数に対するスピードアップ比
Fig. 7 Speed-up ratio to the number of processors for various combinations of the queue structure and the accessing order mechanisms.

台とする．実験に使用した TSP 問題では枝刈りをしていないため，タスクツリーでの深さが深いタスクほどそれ以下にある子孫タスクの総数は減少する．よって，タスクツリーの深さは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総量の目安とできる．図中で，一つの点が他のプロセサのキューからの1回の取り出しを示している．

4.3 節で議論したように，これらの図において良い状態とは，点の数が少なく，密集していない場合である．逆に悪い状態とは，点の数が多く，かつ密集するような場合である．

図9の結果を見ると，

(1) 自分のキューからは FIFO で，他のキューからは 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もののアクセスが現れる．

(4) 自 LIFO 他 LIFO：アクセス量は上記の中間程度．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)の自 LIFO 他 FIFO の構成の時に得られることが確認された．

タスクツリーの根元に近いタスクが必ずしも多くの子孫タスクを生成しない場合でも本方式が有効であることを確認するために，枝刈りのある場合も実験し

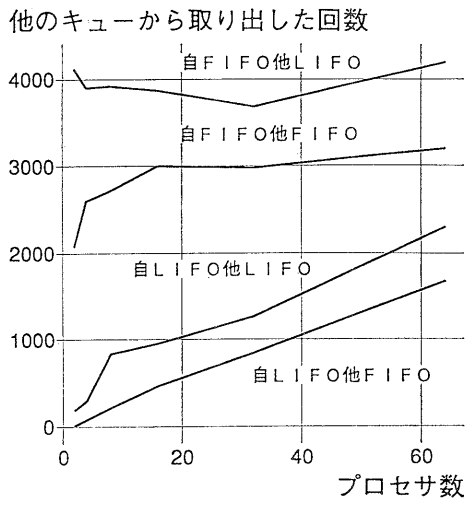


図 8 分散キュー方式における他のキューから取り出した回数

Fig. 8 Number of remote queue accesses for the accessing mechanism combinations of distributed Activity queues.

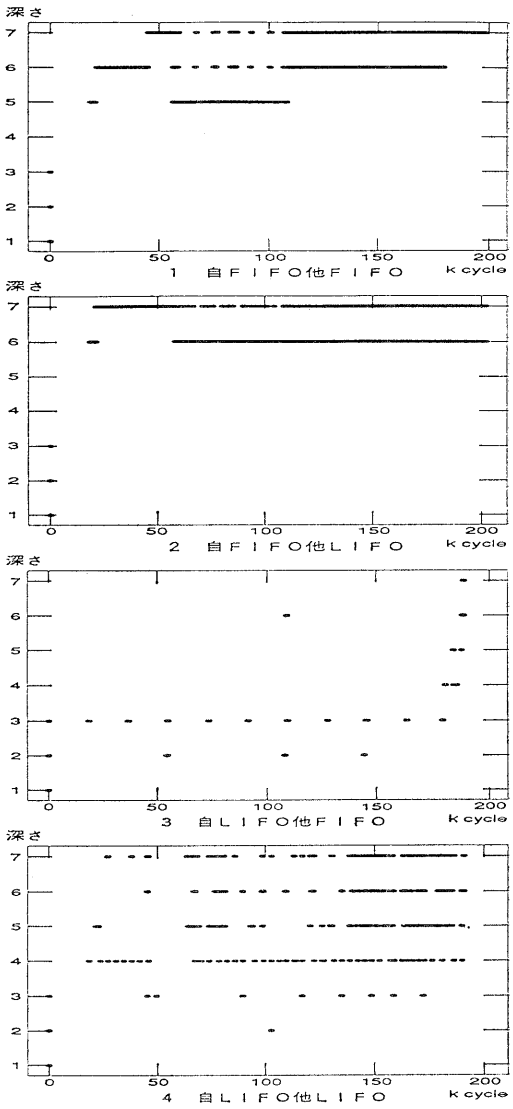


図 9 8都市 TSP を枝刈なしで解いたときの他のキューからの取り出し発生時刻と深さの記録図

Fig. 9 Event charts of remote queue accesses obtained in solving the 8 city TSP without pruning.

た．応用プログラムとして枝刈りのある TSP（9都市）を使用して同じ実験を行った．**図 10** を見ると，図9と全く同様の傾向を示していることがわかる．

表1は，図 9, 10 における他のキューからの取り出し回数を表にしたものである．自 LIFO 他 FIFO のときがもっとも少ないことがわかる．

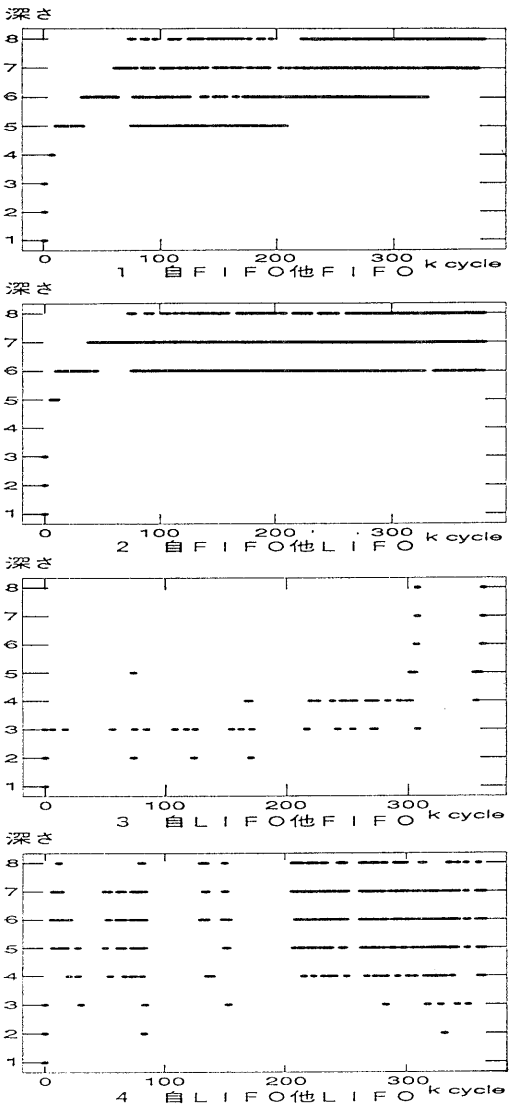


図 10　9都市 TSP を枝刈ありで解いたときの他のキューからの取り出し発生時刻と深さの記録図

Fig. 10　Event charts of remote queue accesses obtained in solving the 9 city TSP with pruning.

最後にメモリ使用量の比較を行う．**図 11** に必要とするアクティビティの最大数について調べた結果を示す．図の横軸はプロセサ台数であり，縦軸はある瞬間ごとに保有すべき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かつ自分のキューの取り出し方式を LIFO 順序にするものは，アクティビティの最大数は小さいものとなる．

同様に FTD の最大数について調べた結果が**図 12** であり，アクティビティのそれと同様な傾向を示すことが確かめられた．

6. 検　　討

4.3 節で述べたように，他のキューからアクティビティを取り出す際により過去に生成されたものを得ることで取り出し回数を減少させることができる．他 FIFO 方式はキューのなかでもっとも過去に生成した

表 1　図 9, 10 での他のキューからの取り出し回数
Table 1　Number of remote queue accesses in Fig. 9 and 10.

取り出し方式	TSP8（枝刈りなし）	TSP9（枝刈りあり）
自 FIFO 他 FIFO	2324	3862
自 FIFO 他 LIFO	3391	4930
自 LIFO 他 FIFO	35	64
自 LIFO 他 LIFO	431	828

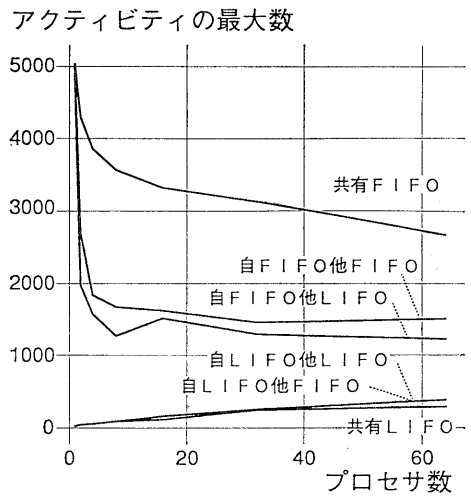


図 11　キュー構造と取り出し方式の違いによるアクティビティの最大数

Fig. 11　Maximum number of Activities stored for various combinations of the queue structure and the accessing order mechanisms.

ものを得ることはすでに議論した．

一方で自分のキューにはより過去に生成されたアクティビティを残すことを考えると，過去に生成されたものが手付かずに残る傾向にある自 LIFO 方式が適していると考えられる．それに対して自 FIFO では過去に生成されたものを使い尽くしてしまう傾向にある．実験結果をみると，自 FIFO の二つに比べて自 LIFO の二つがよいことがわかる．我々は自 LIFO をメモリ使用量削減の観点から提案してきたが，他のキューからの取り出し回数においても顕著な効果があることが確認された．

以上のように，本論文では自 LIFO 他 FIFO の組み合わせが効率のもっとも良いキュー取り出し方式であることが確認された．実はこのような LIFO での子タスクの呼び出しをするタスクの実行順序は通常の関数呼出と似ている．ただし，本手法は最初に仕事を各プロセサに分けて，その後通常の関数呼出をするような並列処理手法に比べて，次の違いがある．

最初に仕事を各プロセサに配分する手法ではプロセサへのタスクの再割当ができなくなる[6]．そのため，他のプロセサの仕事は残っているのに，別のプロセサは遊んでしまうような状況が起きる．本手法では負荷分散がうまく行える．

また，1台のプロセサにのみ注目すると，関数呼び出し方式では，当然のことながら，一つのタスクの実行コストはもっとも小さくなる．アクティビティ方式[4),5)]，およびその改良である本方式はそれよりも若干のコストアップで実現できる．

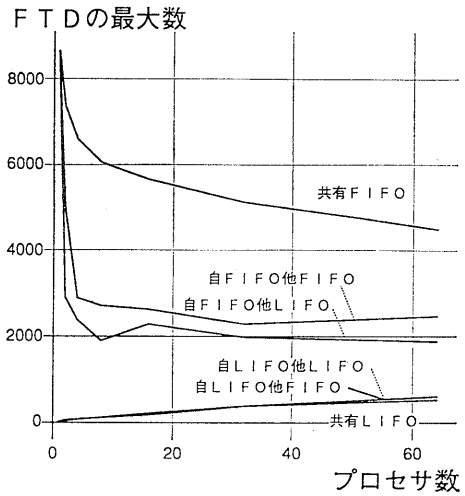


図 12 キュー構造と取り出し方式の違いによる FTD の最大数

Fig. 12 Maximum number of FTD's stored for various combinations of the queue structure and the accessing order mechanisms.

7. 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，プロセサ数や生成するタスク数が多くなっても，キューへのアクセス競合によるオーバヘッドや使用するメモリ量を小さくできることが確認された．

現在のインプリメントの分散キュー方式では，他のプロセサのキューからアクティビティを得る際に，他のキューを一つ調べてそこにアクティビティがあればそれを取り，なければさらに別のキューを調べることを繰り返す単純な設計を採用している．しかしこの設計では，プロセサ数が多い場合にアクティビティを得るまでに多くのキューを調べ，キュー探索のコストが大きくなることがある．

改良案の一つとして，あらかじめ各プロセサが他のプロセサにより取り出されるアクティビティの候補を FIFO 順序で選び，共有領域にいくつか預けておく設計が考えられる．そして，その領域から取り出すときにシステム全体でなるべく過去に生成されたアクティビティが選ばれるように工夫すれば，より効率的にプロセサ間の負荷バランスをとることができると考えられる．そのような設計をいかに小さなオーバヘッドで実現するかは今後の課題とする．

参考文献

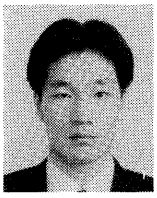
1) 福田　晃：並列オペレーティングシステム，情報処理，Vol. 34, No. 9, pp. 1139-1149 (1993).
2) Anderson, T. et al.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM Trans. Computer Systems*, Vol. 10, No. 1, pp. 53-79 (1992).
3) 新城　靖，清木　康：仮想プロセッサを支援するオペレーティング・システム・カーネルの構成法，情報処理学会論文誌，Vol. 34, No. 3, pp. 478-488 (1993).
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) 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, pp. 985-993 (1993).
6) 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).

(平成5年10月19日受付)
(平成6年6月20日採録)

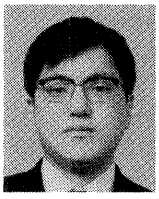
本橋　健（正会員）

1968年生．1992年東京大学工学部計数工学科卒業．1994年同大学大学院工学系研究科修士課程情報工学専攻修了．現在，NTTソフトウェア研究所第一プロジェクトチーム．並列処理，CSCW，コンピュータネットワークに興味を持つ．



中畑　昌也（正会員）

1969年東京都生．1992年東京大学工学部金属工学科卒業．1994年同大学大学院工学系研究科情報工学専攻修士課程修了．1994年4月より(株)日立製作所汎用コンピュータ事業部勤務．並列計算機の設計・開発に従事．

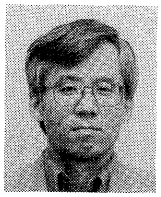


中山　泰一（正会員）

1965年兵庫県生．1988年東京大学工学部計数工学科卒業．1993年同大学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・分散処理に関する研究に従事．最近は，囲碁・将棋などのアルゴリズムの研究にも興味をもつ．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年同大学大学院修士課程修了．1984年同大学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・システムに興味を持つ．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，東京大学大学院修士課程修了（計数工学）．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年～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，画像計測，並列コンピュータの研究に従事．計測自動制御学会，電子情報通信学会，形の科学会，IEEEなどの会員．

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．