共有メモリ型並列機のためのアクティビティ方式を用いる並列実行環境†

田胡和哉††* 槍垣博章††** 森下巖††

共有メモリ型マルチプロセッサを対象とした，並列処理の実行環境について提案する．並列処理の管理方式として，並行プロセスに基づく方式が広く用いられている．プロセスを用いることにより，プロセッサ数やプロセッサ割り当てのスケジューリング方式とは独立に並列処理プログラムを作成することが可能になる．しかしながら，その一方において，プロセスを実現するために費やされるプロセッサ時間，および，メモリ領域は小さくない．特に，高並列機において，細粒度の並列処理を行おうとすると，大きな問題となる．これを軽減するために，並列実行可能な単位の各々にプロセスを割り当てるのではなく，実行単位の発生と，その単位を実行する機構の生成を別個のものとして扱う方式を提案する．並列処理可能な単位を，アクティビティと名付ける．応用プログラムは，アクティビティ生成の形式で並列処理の開始を管理系に依頼する．アクティビティは，生成されると，アクティビティ・キューにいったん蓄えられる．アクティビティの実行機構として，利用可能なプロセッサ数に等しい数の軽量プロセスをあらかじめ用意する．これらのプロセスを複数のアクティビティを実行するために繰り返し利用することにより，論理的にはプロセスを直接利用するのと同等の環境を，より少ないオーバヘッドで実現できる．実際に，市販のマルチマイクロプロセッサ・システム上で並列処理環境を実現し，並列処理を行う応用プログラムを実行した．その実行時間を，従来の，プロセスを直接利用する方式と比較したところ，オーバヘッドを大きく削減できることが確認できた．

1. はじめに

多数の高能力マイクロプロセッサを結合することにより，処理能力が高く，かつ，コスト性能比の優れた汎用高並列の計算機を実現することができる[1]．たとえば，十台程度の高能力プロセッサをキャッシュメモリを介して主メモリに結合した並列機が開発されている．また，数百台規模のプロセッサ要素を多段結合ネットワークを介して同数のメモリ要素と結合する構成の並列機の開発も進められている[2),3)]．本論文では，特に，後者の共有メモリ型汎用高並列機を対象とする．

高並列機を利用しようとする場合，もちろん，高並列度の応用プログラムを組織的に記述するための記述言語の確立も重要になるが，これと同時に，高並列の処理を効率よく管理，サポートするシステム・ソフトウェアの提供も不可欠である．とくに，並列に実行すべき処理の単位が小粒度であり，かつ，並列度が非常に高い場合には，効率のよいサポートを実現しないかぎり，高並列機の利点を生かすことができない．

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，並列に実行すべき一単位の処理をプロセスとして取り扱うものである．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．実行の過程では，メッセージ通信により相互のデータ交換や同期を行うことができる．これは，概念的に単純明快であり，かつ，一般性の高い方式である．現在，その管理方式は完全に確立されている．

しかし，注意すべきは，高並列機を対象とする応用プログラムでは，並列実行の単位をできるだけ小さくとって，並列度が十分高くなるようプログラミングする必要があることである．この結果，プログラム当りの並列実行の単位数が大きくなるとともに，プロセッサ当りの同期処理の頻度や並列実行単位の生成頻度が増加する．このような条件下では，並列に実行すべき処理が発生するごとにプロセスを生成すると，そのオーバヘッドが無視できなくなる．最近では，たとえばMACHシステムにおけるスレッドの実現[4)]にみられるように，プロセスをできるだけ軽量化した軽量プロセスの使用も提案されているが，それでも，軽量プロセスの生成と消滅，および，その並列実行の管理にはかなりのプロセッサ時間を消費する．実際，単位となる処理の粒度がそのオーバヘッド量と同一オーダになると大きな効率低下を引き起こす．

また，軽量プロセスでも，少なくともある量のスタック領域を割り当てなければならないので，メモリ資源の浪費も大きな問題となる．特に，多数のプロセ

† Activity: A Construct for Parallel Processings on Shared Memory Multiprocessor Machines by KAZUYA TAGO, HIROAKI HIGAKI and IWAO MORISHITA (Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科
* 現在 日本 IBM 東京基礎研究所
** 現在 NTT

スを用いた高並列度のプログラムでは，メモリ資源の消費全体に占めるスタック領域の比率が増大し，割り当てた領域の大半がスタックによって消費されることもしばしば起る．

一方，FORTRAN などで記述された数値計算プログラムを対象とする場合，これと全く異なるアプローチも提案されている．プログラム中でもっとも処理量が大きいのは，大規模な配列データの各要素に一定の処理を行うループ文であり，DO_ALL 文などの形で記述して並列に実行させる．この時，すべてのプロセッサに，配列の要素を読み込んでは同一処理を実行するという動作を，すべてのデータに対する処理が完了するまで繰り返させる．その際，各プロセッサにはつぎに処理すべきデータを互に重複しないように決定させなければならない．これには，データのインデックス変数を共有変数として主メモリに格納しておき，Fetch and Add 命令によってつぎに処理すべきデータのインデックスを得る方法を用いる．DO_ALL 文の実行中，システム・コールが発生しないので，非常に効率の高い並列実行となる[2]．しかし，これは，主に数値計算における配列を対象とした演算の並列化を念頭においた方式であり，他の分野への適用は容易でない．

本論文では，Scheme[5] 等の関数型言語，Multilisp[6] 等の並列 LISP，ABCL/1[7] 等の並列オブジェクト指向言語等の，高い並列度を生ずる並列記述言語で記述されたプログラムを効率よく実行することを目的として，新しい並列実行の管理方式を提案し，この管理方式を実現する管理系の設計例とその評価結果を報告する．この管理系の実体は，応用プログラムと同一の実行環境で動作する一群のライブラリ関数であり，下部機構のオペレーティング・システムとしては，既存のもの，たとえば，UNIX を無変更で使用できる．すなわち，オペレーティング・システムは従来のものをそのまま利用しつつ，少数のライブラリ関数を付加することによって，効率のよい汎用並列実行環境を提供しようとするものである．

本方式では，並列に実行すべき処理単位の発生と，資源を割り当てて処理単位の実行機構を用意することを別個に取り扱い，並列実行可能な単位が生じると同時にその実行のためにプロセスを生成することはしない．応用プログラムが実行を要求した，並列実行可能な処理単位をアクティビティとよび，実行機構自体とは独立のものと考える．プログラムがつぎつぎとアクティビティを生成すると，管理系はこれをアクティビティ・キューに結合して保持する．アクティビティの実行機構としては，利用可能なプロセッサ数と高々同数の軽量プロセスをあらかじめ用意しておく．この軽量プロセスは，未実行のアクティビティをアクティビティ・キューからデキューしては実行するという動作を繰り返す．

2. 管理方式の基本原理

2.1 従来の並行プロセス方式

応用プログラムは，並列に実行すべき処理が発生すると，プロセス生成を要求し，そのプロセスに処理を実行させる．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，周知のように，

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数とは独立に任意に生成できる，
2) 並列実行単位に対するプロセッサ割り当て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，

などの決定的な利点がある．

しかしながら，並列に実行すべき処理が発生するごとにプロセスを生成する方式は，プロセスとして軽量プロセスを採用しても，なお，プロセッサ時間とメモリ空間の消費量が大きい．すなわち，

1) プロセス生成時には，スタック領域の割り当て，スタック・イメージの設定，pcb (process control block) 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) 実行中には，スタックの利用にともなうページ・フォールト処理，
4) 実行完了時には，プロセス消滅の操作，

などが必要であり，しかも，

5) プロセスの管理系へのアクセスは直列化されるので，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，並列に実行すべき処理が高並列度，小粒度の場合は大きなオーバヘッドとなる．

2.2 アクティビティ方式

上記の利点を生かしつつ，欠点を除去するにはどのような管理方式を採用すればよいかが問題である．本

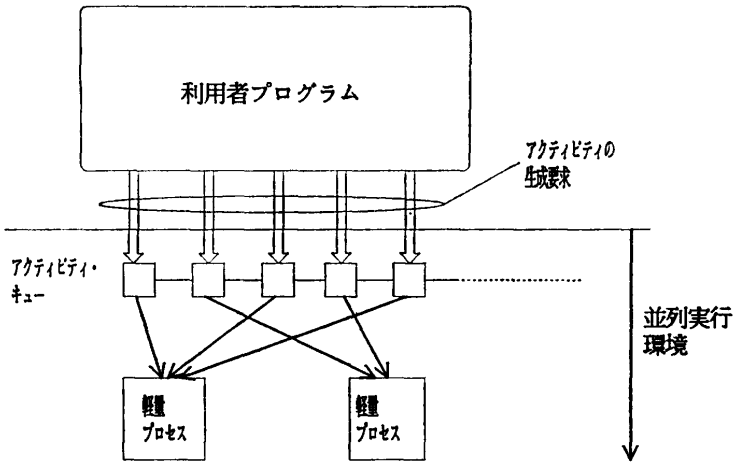


図 1 提案方式による並列処理環境
Fig. 1 Parallel processing environment by the proposed method.

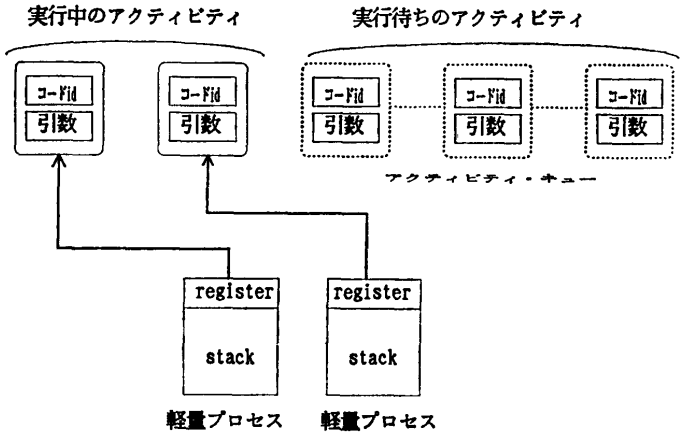


図 2 アクティビティの実行機構
Fig. 2 Execution mechanism of activities.

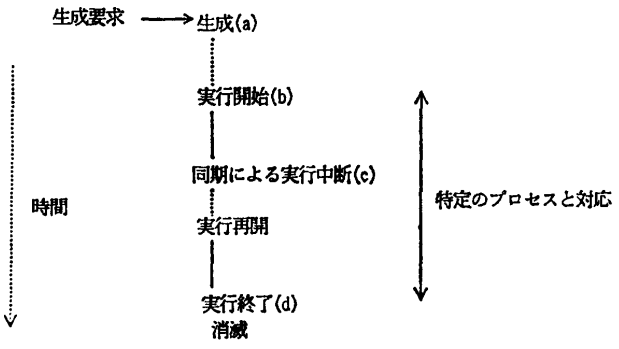


図 3 アクティビティの状態の遷移
Fig. 3 Lifetime of an activity.

論文では，アクティビティ方式とよぶ新しい方式を提案する．

いま，一つの応用プログラムの実行中に，ある処理の並列実行が要求されたとする．本方式では，この処理の実行のために一つの実行機能を用意することはしない．本方式では，これを一つのアクティビティの生成と考え，**図1**に示すように，アクティビティ・キューに接続する．一方，実行機構としては，ある数の軽量プロセスをあらかじめ用意しておく．用意する軽量プロセスの個数は，オペレーティング・システムによってこの応用プログラムに割り当てられたプロセッサの個数と同数とする．それ以上の軽量プロセスを用意しても，並列に実行させることができないから，無駄である．

軽量プロセスはアクティビティを管理系に要求する．管理系はアクティビティを1個取って配分する．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，またつぎのアクティビティを要求する．以下同様である．このように，あらかじめ用意され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消費されるプロセッサ時間を節減できる．

アクティビティ・キューの各要素は，実行すべきプログラムの識別子（コード id）と，それへの引数だけであり，アクティビティ1個あたりのメモリ消費量は小さい．アクティビティが多量に発生しても，メモリ容量ネックは発生しにくい．また，管理系が実行しなければならない処理は，アクティビティ・キューへのエンキューとデキューだけであり，ボトルネックが発生しにくく，処理量も小さい．

軽量プロセスの内部状態は，**図2**に示すように，レジスタとスタックで表現される．アクティビティが終了すると，対応するプロセスの内部状態はすべて不要になる．したがって，プロセスは，スタック・ポインタを初期化するだけで新しいアクティビティの実行を開始できる．

個々のアクティビティは，**図3**に示す過程を経て実行される．生成されたアクティビティは，軽量プロセスの割り当てを待ち(a)，軽量プロセスが割り当てられると実行が開始される(b)．軽量プロセスとの対応関係は，実行が完了するまで変らない．場

合によっては，実行途中において同期命令を発行して実行を一時中断する(c)．アクティビティの実行が完了することにより，軽量プロセスとの対応関係が解消され，アクティビティは消滅する(d)．

不幸にして同期による実行の中断が生ずると，管理系はその軽量プロセスを待ち状態に置き，新しい軽量プロセスを1個生成して，実行を中断した軽量プロセスの代りにこれを走行させる．

資源の消費は，用意した軽量プロセスの数によって制御できる．新たなプロセッサが割り当てられた場合には軽量プロセスを生成する．逆に，新しいタスクが発生して，システム全体のプロセッサ資源やメモリ資源が不足している場合には，一つのアクティビティが完了した時点でその軽量プロセスを消滅させればよい．

2.3 アクティビティ方式の適用対象

本方式は，論理的には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が，メリットが大きくなるのは下記の場合である．

(1) 処理単位の生成頻度

生成頻度が大きくてもよい．生成のオーバヘッドが軽量プロセスを直接使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 N よりはるかに大きくてもよい．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によって同期をとることが望ましく，その実行途中で同期をとることが少ない方がよい．新たな並列実行単位の生成が少なく，実行途中で頻繁に同期をとる形式のプログラムでは，実行効率がプロセスを直接用いる方法と等しくなってしまう．

以上のことから，実行途中で同期をとることが少ないアクティビティを積極的に生成することによって並列度を高めるように，応用プログラムを作成することが望ましい．

2.4 アクティビティ方式の利用法

(1) 対象言語

本方式は，関数型，並列 LISP，並列オブジェクト指向型等の，並列記述言語の実行時環境として利用するのが有効である．これらの言語を用いることにより，きわめて高い並列度を得ることができる．これは，高並列機を利用するうえで，有用な性質である．その一方において，プロセッサ数に比べてあまりに過度な並列度を生ずると，そのままでは実行が困難になる[8]．提案方式は，この問題を軽減する一つの方法を与える．

関数型の言語は，計算結果が関数の評価順序によらない性質を持つ．これを利用して，複数の関数を並列に評価することにより並列処理を実現することが試みられている．この方式では，特に，関数の再帰呼出しにより，きわめて高い並列度を生ずる．たとえば，1回実行されるたびに n 回同一関数を呼び出す関数が，k 段再帰的に実行されると，n^k 個の関数実行が並列に行われる可能性がある．これは，高並列機のプロセッサ数より容易に多くなりうるので，プログラム上で，関数を順序的に実行するか並列に実行するかを陽に制御するか，あるいは，実行環境がその制御を行う必要がある．

並列 LISP の一つである Multilisp では，future コンストラクトにより，関数型の言語と同様に，関数を単位とする並列処理を可能にしている．ここでも，高い並列度が得られる．

並列型オブジェクト指向言語の一つである，ABCL/1 では，未来型，および，過去型のメッセージング機構を実現している．ここでは，メソッドを単位とする並列処理が行われ，関数型と同様に高い並列度が得られる．

(2) 対象計算機

提案方式は，共有メモリ型の汎用高並列機上で，(1)で述べた言語を効率よく実行することを目的としている．多段結合ネットワークを用いることにより，数百台規模の共有メモリ型高並列機を実現することができる．現在，筆者らの研究室では，このような計算機の開発を進めている[9]．

3. プリミティブの設計と利用例

本方式による並列処理の管理機構は，実行時ライブラリとして実現する．その機能は，プリミティブを関数呼出しの形式で呼び出すことにより利用する．プリミティブの設計と，それを利用した並列処理プログラムの記述例について述べる．

3.1 プリミティブ

プリミティブとして，Ainit, Aalloc, Aexit, Await, Asuspend, Aresume を用意する．

1) Ainit

管理系全体を初期化する.

2) Aalloc

Aalloc によって，並列アクティビティの起動要求を登録する．第1引数によって，実行すべきプログラムのエントリ・アドレスを指定する．第2，3引数によって，プログラムの起動時に渡す引数を二つまで指定する．第4引数によって並列アクティビティの起動モードを指定する．この引数の MULTI ビットが0のときには，生成されたアクティビティに一つのプロセスが割り当てられる．1のときには，すでに生成されている可能な限りのプロセスが割り当てられる．

Aalloc は，アクティビティの識別子を返す．これは，Await，Aresume によって利用される．

3) Aexit と Await

Aexit によって，起動されたアクティビティの実行を終了する．もし，Aexit の引数が0でなければ，実行の終了を，それを起動したアクティビティに知らせる．起動したアクティビティは，Await によってこれを待ち合わせる．これを用いて，fork-join 型の同期が実現できる．

4) Asuspend と Aresume

Asuspend によって，アクティビティの実行を一時中断する．その再開は，Aresume によって実現される．Aresume は，アクティビティの識別子を指定して実行可能状態にする．これを用いて，PV 命令，通信機構，モニタ機構等の同期，通信コンストラクトを構築できる．

3.2 記 述 例

上記プリミティブは，並列記述言語の実行時環境として利用するのが一般的であるが，アプリケーション・プログラムから直接呼び出して利用することもできる．高い並列度を生ずるプログラムの簡単な例として，クイック・ソート・プログラムをあげる．

提案方式を用いて実現した，クイック・ソート・プログラムの一例を**図4**に示す．クイック・ソート・プログラム qsort は，ソートすべき配列 buf と，その長さ size を引数として持つ．size がしきい値 THRESHOLD より大きければ，関数 div によって，buf をキー値より大きい部分と小さい部分の二つのエレメントに分割する．Aalloc を用いて，それぞれについて並列に qsort 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 によってソートを行う．

```
qsort(buf, size)
{
  register  newsize;

  if(size > THRESHOLD) {
    newsize = div(buf, size);
    Aalloc(qsort, buf, newsize, 0);
    Aalloc(qsort, buf+newsize,
        size-newsize, 0);
  } else
    bsort(buf, size);
}
```

図4 並列クイック・ソート・プログラム
Fig. 4 Parallel quick sort program.

ここで重要な点は，並列度がきわめて高くなることである．たとえば 10^6 個のデータのソートで，しきい値が 20 byte であれば，最悪の場合 $5*10^4$ 個のエレメントに分割され，それぞれに bsort が並列に適用されることになる．それぞれにプロセスを割り当てる方法では，すぐにスタック領域が不足してしまう．そこで，プロセスを直接用いる方法では，このような単純なプログラムは破綻をきたし，アプリケーション・プログラム・レベルでプロセスの生成を制限するような制御を行う必要が生ずる．

これに対して，提案方式では，並列処理要求を保持するためにわずかな領域しか必要としない．そこで，このプログラムは，破綻をきたすことなく，そのまま実行することができる．

4. 管理機構の実現

3章で述べたプリミティブの機能を実現する管理機構は，軽量プロセスの管理機構と，アクティビティの管理機構から構成される．軽量プロセスがすでに実現されているシステムでは，それをそのまま利用できる．

提案方式の特性について検討するために，市販の共有メモリ型マルチプロセッサ上に管理機構を実現したので，その実現方式にについて述べる．マルチプロセッサの仕様の概要を**表1**に示す．UNOS オペレーティング・システムは，UNIX システムとほぼ互換であり，ここで利用した機能はすべて UNIX システム

表1 対象システムの仕様
Table 1 Specification of the target system.

システム名	Universe
オペレーティング・システム	UNOS
プロセッサ	68020
プロセッサ数	2
方式	共有バス方式
システム・クロック	16 MHz
主記憶	16 Mbyte

においても提供されている．軽量プロセスの管理機構は提供されていないので，それも含めて実現した．

管理機構が提供するプリミティブは，C言語から利用できるようになっている．管理機構自体も，その大半がC言語によって記述されており，マルチプロセッサ用の特殊命令，および，プロセスのコンテクスト・スイッチを行う部分のみがアセンブリ言語によって記述されている．

（1） 軽量プロセスの管理機構の実現

図5に示すように，UNIXが提供する，複数の利用者プロセス間で共有可能な記憶領域を割り当てる機能を用いて，軽量プロセスの管理機構を実現した．応用プログラムの大域変数領域，および，管理機構の制御変数は，この，共有変数領域上にとる．一方，軽量プロセスのスタック領域は，各利用者プロセスのスタック領域を分割して割り当てる．

軽量プロセスの状態，および，スタック・ポインタを保持するpcbを，軽量プロセスごとに割り当てる．pcbのリストによって，実行可能リスト，空きプロセス・キュー，および，待ちキューを実現する．

この環境では，プロセッサ割り当てのスケジューリングは，オペレーティング・システムの，利用者プロセスに対するプロセッサ割り当て機能に依存している．プログラムの並列度が低下した場合には，同期のためのシステム・コールによって利用者プロセスの一つの実行を中断することにより，プロセッサを開放する．この利用者プロセスの実行を再開することにより，プロセッサが再度割り当てられる．

（2） アクティビティの管理機構の実現

アクティビティのすべての管理は，実行時ライブラリのプリミティブによって実行される．その実現方式の概要は，以下のとおりである．

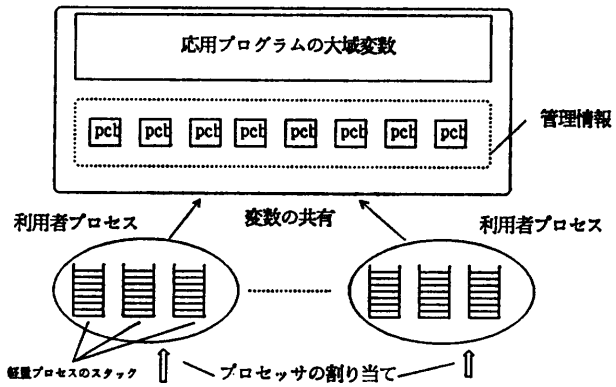


図5 軽量プロセスの実現
Fig. 5 Implementation of light-weight processes.

1) Ainit

制御表を初期化する．

2) Aalloc

アクティビティのエンキューを行う．また，実行中断している利用者プロセスがある場合には実行再開して，割り当てプロセッサ数を増やす．

これによってプロセッサが割り当てられると，それは，空きプロセス・リストから軽量プロセスを割り当てて，アクティビティの実行を開始する．空きプロセスがない場合には，効率の点からいくつかの軽量プロセスをまとめて生成し，あまった分は空きプロセス・キューにつないでおく．

3) Aexit

アクティビティが終了すると，アクティビティ・キューが空でない場合にはつぎのアクティビティの実行を開始する．空の場合には，実行可能リストから軽量プロセスを取り出して実行する．どちらも空の場合には，軽量プロセスを空きプロセス・キューにつないで，プロセッサを開放する．

4) Asuspend, Await

待ち状態の判定を行い，アクティビティの実行を中断すべき場合には，対応する軽量プロセスを待ち状態にし，そのプロセスはそのまま放置する．さらに，アクティビティ・キューが空でなければ，軽量プロセスを割り当てて他のアクティビティの実行を開始する．

5) Aresume

アクティビティの待ちが終了した場合には，対応する軽量プロセスを実行可能リストにつなぐ．

5. 性能評価

提案方式による並列処理の管理機構とは別に，従来のプロセス方式の管理機構（軽量プロセスを直接利用するプリミティブを提供するもの）をも実現し，両者の性能の比較を行った．二つの環境を用いて，図4に示したクイック・ソート・プログラムを実行した．軽量プロセスを直接利用する場合には，Aalloc の代わりに，プロセスを生成するプリミティブを利用する．

2個のプロセッサを用いて，$5*10^5$ 個のランダム・データをソートするのに要した時間を，**図6**に示す．生成されるアクティビティ，あるいは，プロセスの総数は，クイック・ソートからバイナリ・ソートに移行することを決めるしきい値を変

えることによって制御できる．この総数を，並列実行要求数と呼ぶことにする．このとき，全体として実行するソートの総量は同一である．同図では，比較のために，単一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．しきい値が同一のとき，再帰手続呼出しの実行総数は，並列実行要求数に等しい．縦軸は，処理全体に要した時間を time コマンドを用いて計測したものである．同様に，10^5 個のデータのソートに要した時間を図7に示す．二つの図とも，よく似た傾向を示している．アクティビティ方式が軽量プロセス方式よりも，常に処理時間が短い．

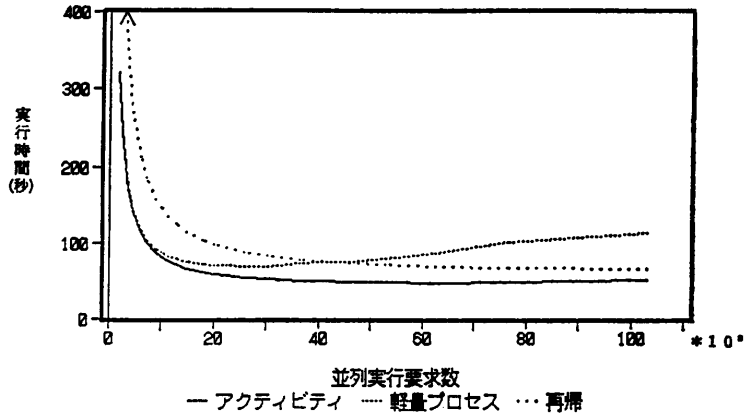


図 6 $5*10^5$ 個データのソート実行時間
Fig. 6 Execution time of sorting $5*10^5$ data.

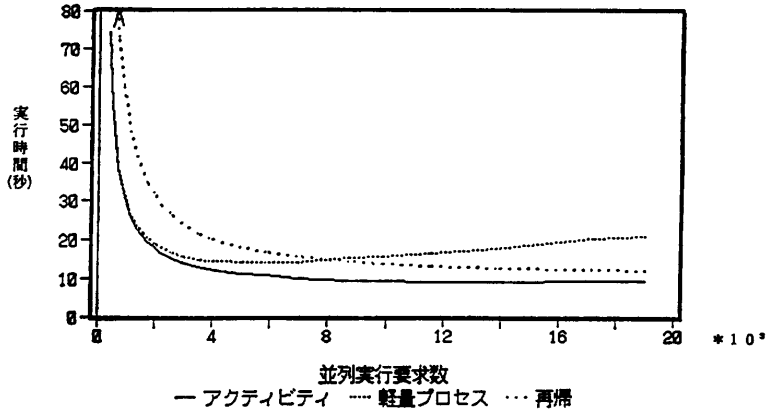


図 7 10^5 個データのソート実行時間
Fig. 7 Execution time of sorting 10^5 data.

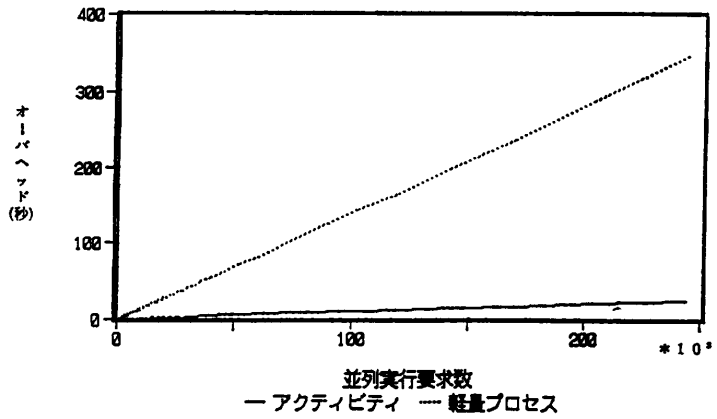


図 8 実行環境のオーバヘッド
Fig. 8 Overhead of parallel execution environment.

並列実行要求数が少ない領域において，実行時間が著しく短縮されているのは，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，並列実行要求数が増加するにしたがって実行時間の増加がみられる．これは，オーバヘッドによるものである．特に，プロセスを用いる方式では，並列実行要求数の比較的少ない領域で，すでに再帰手続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．アクティビティによる方式でも，実行時間の若干の増加がみられるが，プロセスを用いる方式よりはるかに少ない．

オーバ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，再帰手続呼出しによって記述したクイック・ソート・プログラムの性能と比較した．$5*10^5$ 個の例について，再帰手続呼出しによる実行時間を正味の処理時間，それと比べた増加分をオーバヘッドと考えて比較を行うと，図8のようになる．オーバヘッドが 1/15 程度に削減されていることがわかる．

6. むすび

プロセスは，種々の特徴を持つ複雑なコンストラク

トである．それは，多くの利点を持つと同時に，多くの資源を消費する．高並列システムでは，資源の消費を削減するために，実行方式を改善することが必要になる．並列処理において，プロセスの生成によってプログラマが行いたいのは，並列実行の開始を要求することであって，大きなコンテクストをこの時点で生成する必要はない．並列処理の実行要求のみをキューイングすることにより，より高い実行効率と資源利用効率が得られる．この方針に基づいた並列実行環境の実現方式を述べた．また，並列実行の管理系を実現し，評価を行うことにより，提案方式の有効性を示した．実際に，高並列機上で大規模な並列処理プログラムに適用し，その有効性を確認することが今後の課題である．

謝辞 本研究をまとめるにあたり，討論を通じて貴重な御助言をいただいた，東京大学工学部計数工学科の出口光一郎助教授に感謝します．また，研究に必要な計算機環境を御提供いただいた，東京大学工学部計数工学科の和田英一教授および岩崎英哉博士に感謝します．

参考文献

1) 富田：並列計算機構成論，昭晃堂 (1986).
2) Gottlieb, A., Grishman, P., Kruskal, C. P., McAuliffe, K. P., Rudolph, L. and Snir, M.: The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. Comput.*, Vol. C-32, No. 2, pp. 175-189 (1983).
3) Siegel, H. J.: *Interconnection Networks for Large-Scale Parallel Processing*, Lexington Books, Toronto (1985).
4) Rashid, R. F.: Threads of a New System, *UNIX Review*, pp. 37-49 (1986).
5) Steele, G. L., Jr. and Sussman, G. J.: The Revised Report on Scheme, AI Memo 474, MIT (1978).
6) Halstead, R. H.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 4, pp. 501-538 (1985).
7) Yonezawa, A. and Tokoro, M.: *Object Oriented Concurrent Programming*, MIT Press (1987).
8) Mohr, E., Kranz, D. A. and Halstead, R. H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *1990 ACM Conference on LISP and Functional Programming*, pp. 185-197 (1990).
9) 数藤，田胡，永松，出口，森下：多重命令流プロセッサを用いる多段結合ネットワーク結合共有メモリ型並列機のシミュレーションによる性能評価，第41回情報処理学会全国大会論文集，6-77 (1990).

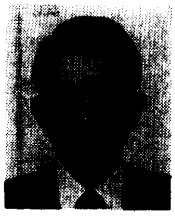
(平成2年4月16日受付)
(平成2年11月13日採録)

田胡 和哉（正会員）

昭和31年生．昭和56年筑波大学第3学群情報学類卒業．昭和61年同大学大学院工学研究科博士課程修了．工学博士．同年同大学電子・情報工学系助手．昭和63年東京大学工学部計数工学科助手．平成2年より日本IBM東京基礎研究所勤務．オペレーティング・システムの設計方式に興味を持つ．昭和60年本学会論文賞受賞．計測自動制御学会，ソフトウェア科学会各会員．

檜垣 博章

昭和42年生．平成2年東京大学工学部計数工学科卒業．同年日本電信電話(株)入社．現在，同社ソフトウェア研究所にて分散実行環境に関する研究に従事．



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．電気学会などの会員．