計算機アーキテクチャ　83－25
（１９９０．７．１８）

並列処理用ＯＳ・ＳＫＹ－１のスレッドインタフェース

齊藤　雅彦　　上脇　正　　山口　伸一朗
㈱日立製作所　日立研究所

共有メモリ型マルチプロセッサにおける並列処理の単位として、プロセス内部で並列に動作するスレッドが注目されている。しかし、従来のプロセスに加え、新たな概念であるスレッドを導入することは、ユーザに混乱を招く可能性がある。我々はスレッドのユーザインタフェースを向上させるため、各種の並列処理モデルに基づいて、スレッドインタフェース“ｍ”を開発した。“ｍ”では、並列処理での有効なスレッド制御方式として、複数のスレッドをまとめて制御するグループの概念を新たに導入した。“ｍ”を評価した結果、性能低下を起こすことなく、プログラムを並列処理用に改造するための期間を約５０％削減できる見通しを得た。

Multiprocessor Operating System : SKY-1 - Thread Interface -

Masahiko SAITOH　Tadashi KAMIWAKI　Shin'ichiro YAMAGUCHI
Hitachi Research Lab., Hitachi, Ltd.
4026 Kuji-cho, Hitachi-shi, Ibaraki-ken, 319-12 Japan
miyabi@hrl.hitachi.co.jp

In a shared memory multiprocessor, the concept of threads which execute the subroutines concurrently in a process is effective in performance improvement. But, it makes the program somewhat complicated to offer threads in addition to processes. We developed thread interface “m” which improves user interface of threads. “m” includes a new notion : ‘thread group,’ which controls several threads togather. By using “m” and its ‘thread group,’ the programmer can make parallel programs more easily than using processes or threads without performance degradation.

1．はじめに

並列処理は計算機性能の向上に欠かせない技術である。

近年、計算機性能に対する要求は単一プロセッサの性能向上を上まわって大きくなってきている。画像処理、オンライントランザクション処理、データベース管理等の分野において、その傾向が著しく、これらの分野においては並列処理によって計算機性能を向上させることが急務である。この並列処理技術として、複数個のプロセッサを有して並列に動作させるマルチプロセッサ技術が注目されている。

マルチプロセッサシステムにおいては、ＯＳが計算機システムの並列処理機能を十分に活用し、プログラム、サブルーチンをできる限り並列に実行させる必要がある。我々はマルチプロセッサの基本ソフトウェアとして並列処理用ＯＳ・ＳＫＹ－１（System Kernel for You -1）を開発した。[1)~4)] ＳＫＹ－１では、サブルーチンレベルの並列処理にも対応するために、従来の処理単位であるプロセスに加え、負荷の軽いスレッド[5)]の概念を導入している。

しかし、プロセスの概念に加えて、新たにスレッドの概念を追加することは、ユーザにとってプログラミングの複雑さが増すことになる。たとえスレッドの使用によりプログラムの実行が高速化されるとしても、そのスレッドが使い難いものであれば、スレッドの普及は困難である。このため、ＳＫＹ－１では、スレッドのユーザインタフェース向上策として、スレッドインタフェース“m”を開発した。我々は各種の並列処理プログラムを分析した結果、“m”において、複数個のスレッドをまとめて制御する「グループ」の概念を導入した。

2．ＳＫＹ－１での並列プログラミング

近年並列プログラミング方法論（Parallel Programming Paradigm）について議論が盛んになっている。ＳＫＹ－１で並列プログラミングの方法を決定するにあたって、以下の２つの課題を考察した。

1)問題に内在する並列性をどのように表現するか；

2)表現された並列性をどのように計算機の並列処理機能へ展開するか；

前者を「並列処理モデル」、後者を「並列処理方式」と呼ぶ。以下、この２つに関して説明する。

2．1．並列処理モデル

ＳＫＹ－１では、並列処理を表現するモデルとして、以下に示す２つの並列処理モデルを考察した。[8)]

1)パイプラインモデル：

複数個のプログラム、サブルーチン等が流れ作業で処理を進める。

2)同時処理モデル：

複数個のプログラム、サブルーチン等が、独立に計算を行う。我々は同時処理モデルをユーザの表現する形態により、さらに、以下の５つの並列処理モデルに分類、考察した。

a)単純並列処理モデル：

複数のプログラム、サブルーチン等が、個別のデータを用いて、独立に計算を行う（図１a））。

b)ループモデル：

ＤＯ－ＡＬＬ型のループによる配列計算等において、並列実行可能な配列要素の演算を、ループ分解して処理する（図１b））。

c)サーバモデル：

特定対象を管理するサーバは基本的に、１つの依頼を処理中、他の依頼を処理することができない。このため、同一サーバを複数個存在させ、並列処理を行う（図１c））。

d)ブロードキャストモデル：

サーバ等のプログラムが各種存在する場合、これらのプログラムをまとめて起動／終了させたい場合がある（図１d））。例えば、グラフィックスシステムにおいて、座標変換の処理を独立なＸ、Ｙ、Ｚ座標変換に分割し、並列に実行する場合に対応する。

e)木構造（再帰計算）モデル：

パズルの全開探索木等では、探索過程で親プロセスが複数個の子プロセスを生成し、部分問題を並列に解くことにより、全ての解を求めることが可能となる（図１e））。この場合、木の葉にあたる処理が、その時点において並列に実行され得る処理である。

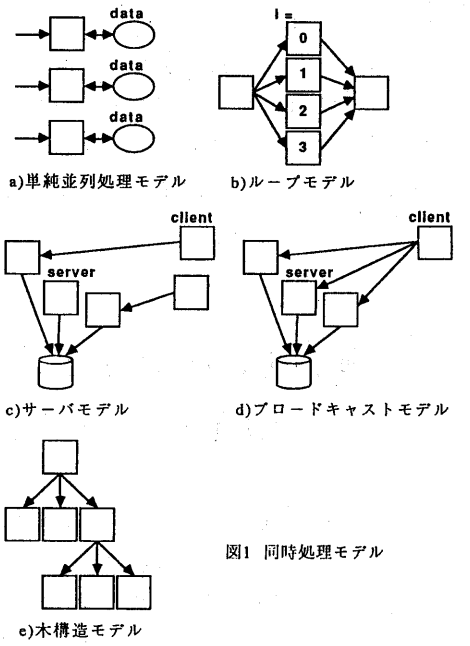

図1 同時処理モデル

2．2．並列処理方式

並列処理方式とは、計算機システムの階層構成（図２）の各階層において、どのように並列処理を支援するかを意味する。ＳＫＹ－１の現在の並列処理方式としては、ＯＳレベル、ライブラリ／サーバレベルに重点を置いている。

（１）ＯＳレベルの並列処理方式

従来の並列処理単位であるプロセスはプログラムレベルの並列処理に適している。これに対して、共有メモリ型マルチプロセッサにおいては、サブルーチンレベルの並列処理が効率的である。このための新しい並列処理方式として、プロセス内の実行環境を共有して並列処理を行うスレッド

の概念がある。[5)~7)] ＳＫＹ－１でも並列処理の単位として、従来のプロセスに加え、スレッドを導入した。

スレッドはプログラムカウンタ、スタックポインタ、汎用レジスタ等を有した論理プロセッサであるとみなすことができる（図３）。同一プロセス内のスレッド切換え時には、実行環境を切り換える必要がなく、プロセスに比べて、切換え処理を高速化することができる。スレッドを生成する場合においても、仮想記憶管理等のテーブルを生成、もしくは、複製する必要がない。また、データの共有が容易という利点もある。

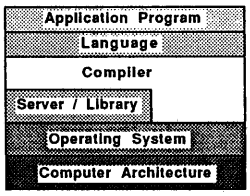

図2 計算機システムの階層構成

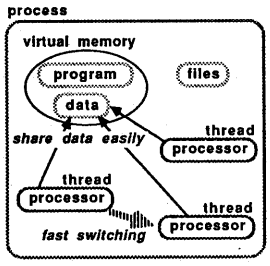

図3 スレッドによる並列処理

（２）ライブラリ／サーバレベルの並列処理方式

並列処理モデルのうち、パイプラインモデルと単純並列処理モデルはスレッドの基本機能により容易に表現できる。しかし、これら以外の並列処理モデルでは、基本機能のみで並列処理を記述することは難しい。

このため、ＳＫＹ－１ではライブラリ／サーバレベルにおける並列処理のインタフェースとして、スレッドインタフェース“m”を開発した。“m”では、複数個のスレッドをまとめて制御の対象とする「グループ」という概念を導入した。我々は各種並列処理モデルを分析し、ユーザがプログラムを記述し易くするためには、「複数個の処理をまとめて制御する」ことが基本的であるとの結論に達している。

分散処理を主目的とするＯＳにおいては、Stanford大学のV、[9)] 京都大学のR^{2}[10)]のように、既にグループをプログラムレベル（プロセス）の並列処理に適用したものがある。これに対して、ＳＫＹ－１ではグループをサブルーチンレベル（スレッド）の並列処理に適用した。すなわち、ＳＫＹ－１のグループは並列処理モデルのうち、分散処理的なサーバモデル、ブロードキャストモデルだけでなく、ループモデル、木構造モデルに対しても一様に適用されるものである。各並列処理モデルにグループを適用すると、図４a)～d)に示すような形式となる。

３．スレッドインタフェース“m”

３．１．設計方針

“m”の実現方針として、以下の条件を念頭においた。

1)ＯＳを肥大化させないこと；

2)多様で柔軟な機能を実現可能であること；

“m”では、並列処理支援のインタフェースという立場上、排他制御などのため、システムコールの実行が不可避とな

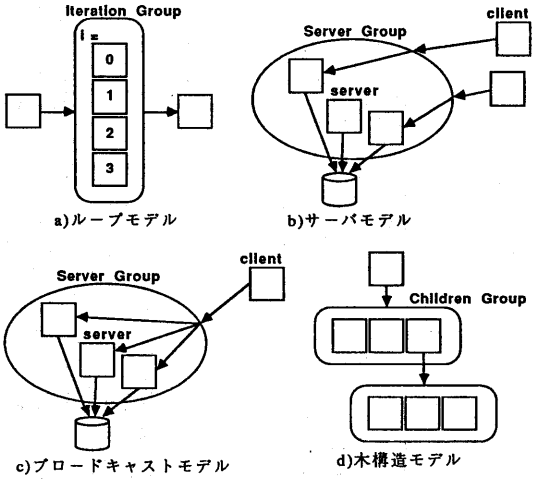

図4 同時処理モデルとグループ

る。このため、速度を重視すると、全ての必要な機能をシステムコールで実現する方法が最もよい。しかし、高速化のみを追求し、全てのスレッド制御機能をシステムコールで実現するようになれば、ＯＳがますます肥大化することになる。また、機能の追加や変更が行われるたびにＯＳのバージョンアップが必要となる。したがって、ＳＫＹ－１においては、基本的な機能のみをシステムコールで用意し、高度な機能は基本システムコールの組合せで実現した。具体的には、以下に示す２つの方式で“m”を実現、評価した。

a)ライブラリ方式：

プログラム中で使用するスレッド制御用関数はオブジェクトプログラム生成時にプログラム内に組み込む。関数内では基本システムコール（２３種類）の組合せにより必要な機能を実現する。ライブラリにおける排他制御も基本システムコールによる（図５a））。なお、グループの生成／削除、メンバ管理等、グループ管理の基本的な機能はＯＳ内で実現した（２３種類中８種類）。

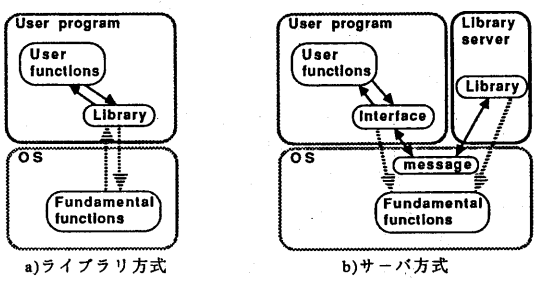

図5 ライブラリ実現方式

b)サーバ方式：

上記ライブラリをサーバプロセス（ライブラリサーバを“M”と呼ぶ）として構成し、ユーザプログラムはライブラリサーバとの通信により、機能を実行する（図５b））。この方式は、ＯＳ機能の分散化（Kernelized Kernel）の流れに沿うものとして位置付けている。サーバ方式では、

表1 SKY-1スレッドインタフェース"m"の関数一覧

	single thread control	multiple threads control (group control)
fork / terminate	thd_fork(func,args) thd_fork_alloc(stksiz,func,args) thd_exec(tid) thd_exit(status) thd_terminate(tid,status)	grp_fork(key,num,func,args) grp_fork_alloc(key,num,stksiz,func,args) grp_exec(gid) grp_terminate(gid,status)
scheduling	thd_join(tid) thd_yield() thd_sleep(time) thd_suspend(tid) thd_resume(tid)	grp_join(gid) grp_join_all(gid) thd_costop() thd_cocall(tid) thd_coend_grp(gid) grp_suspend(gid) grp_resume(gid) grp_proc_set(vp,gid) grp_proc_reset(gid) grp_barrier()
grouping		grp_alloc(key) grp_alloc_one(key,tid) grp_free(gid) grp_add(gid,tid) grp_delete(gid,tid) grp_id_thd(key) grp_member(gid,tid)
exclusion	mtx_alloc(key) mtx_free(mid) mtx_lock(mid) mtx_lock_try(mid) mtx_unlock(mid) mtx_unlock_thd(tid,mid) cnd_alloc(mid,key) cnd_free(cid) cnd_wait(cid) cnd_wait_thd(tid,cid) cnd_signal(cid) cnd_signal_thd(tid,cid) cnd_broadcast(cid)	mtx_unlock_grp(gid,mid) cnd_wait_grp(gid,cid) cnd_signal_grp(gid,cid) cnd_broadcast_grp(gid,cid)
message	msg_send(tid,buf,size) msg_receive(tid,buf,size) msg_receive_all(ptid,buf,size)	msg_send_grp(gid,buf,size) msg_broadcast_grp(gid,buf,size) msg_receive_grp(gid,buf,size)
signal	thd_signal(type,func) thd_kill(tid,type) thd_alarm(time) thd_pause() thd_sighold(type) thd_sigrelse(type) thd_sigignore(type)	
etc.	gettid() thd_exist(tid) thd_list(tlist,maxsiz) msg_list_send(tlist,maxsiz) msg_list_receive(tlist,maxsiz) thd_errno()	grp_exist(gid) thd_list_grp(gid,tlist,size)

メッセージ通信により同期、排他制御が容易に実現できるため、ＯＳ内に用意する機能が少なくてすむ。具体的には、スレッドの生成／終了／停止／再開およびスレッド間メッセージ通信だけでよい（１２種類）。また、この方式では、ＯＳ内のグループ管理機能は使用せず、全てライブラリサーバ“M”内で管理する。

３．２．構成

スレッドインタフェース“m”はグループ制御を行う関数を含めて、表１に示す６９個の関数から構成される。これらを組み合せて制御することにより、きめ細かなプログラミングが可能となる。

３．３．スレッドグループ制御

グループ機能を使用して複数個のスレッドをまとめて制御することにより、各種並列処理モデルを容易にプログラミングできる。以下、各並列処理モデルにグループを適用するため、“m”において導入したグループ制御機能を説明する。

1)グループ（複数スレッド）の生成／終了待ち

ループモデル、ブロードキャストモデル、木構造モデルに対しては、複数スレッド（グループ）の生成／終了待ち等の機能が必要である。図６にループモデルを使用した配

```
#define ROW1 50
#define COL1 50
#define ROW2 COL1
#define COL2 50
int gid;   /* Thread group ID */
int rownum=0;   /* Row number */
int row_lock;   /* Lock ID of 'rownum' */
double matrix1[ROW1][COL1];
double matrix2[ROW2][COL2];
double result[ROW1][COL2];

main()
{
  ..... Input matrix .....
  row_lock=mtx_alloc(0);   /* Get lock ID */
  /* Calculation threads fork */
  grp_exec(gid=grp_fork(0,ROW1,calc));
  grp_join_all(gid);   /* All threads end? */
  ..... Output matrix .....
}

calc()
{
  int i,j,row;
  double sum;
  mtx_lock(row_lock);
  row=rownum++;   /* Execute 'Fetch&Add' */
  mtx_unlock(row_lock);
  for(i=0; i<COL2; i++) {
    sum=0;
    for(j=0; j<ROW2; j++)
      sum+=matrix1[row][j]*matrix2[j][i];
    result[row][i]=sum;
  }
}
```

図6 ループモデル使用例

列計算の例を示す。ループ開始点でgrp_fork、grp_exec関数によりグループを生成／起動し、終了点でgrp_join_all関数により、グループ全体の終了を待つ。

2)グループへのメッセージ通信／放送

サーバモデルについてはグループへのメッセージ通信、ブロードキャストモデルについてはグループへのメッセージ放送の機能を導入した。サーバモデルの例を図７に示す。サーバスレッドを複数個生成して、サーバグループとする。グループへのメッセージ通信（msg_send_grp）が行われると、“m”がサーバグループ中で暇なスレッド（メッセージ受信待ち状態であるスレッド）を捜し、そこにメッセージを転送する。

3)グループ内での排他制御／同期

グループ内で並列処理中に、同一制御対象を扱う場合がある。このとき、グループ内で排他制御、同期を行うことが可能となれば、便利である。“m”では、a)グループ内でのMonitor制御、[11] バリア同期；b)グループの強制停止／再開機能；c)グループの単一プロセッサへの固定；という３種類の機能を用意した。このような排他制御を使用することによって、ユーザはスレッドの複雑な制御を容易に実現できる。

４．評価

“m”の評価においては、どれくらい使い易くなったかが最も重要な観点となる。しかし、このスレッドインタフェースを導入した結果、性能が著しく低下することは避けなければならない。そこで、次の２点から評価を行った。

・性能向上／低下率

・ユーザインタフェース改善度

```
#define CLIENT 10
#define SERVER 15
#define MAXBUF 32
int gid;   /* Server group ID */

main()
{
  /* Server & Client threads fork */
  grp_exec(gid=grp_fork(0,SERVER,server));
  grp_exec(grp_fork(0,CLIENT,client));
}

server()
{
  char buf[MAXBUF];
  int tid;
  for(;;) {   /* Loop forever */
    msg_receive_all(&tid,buf,MAXBUF);   /* Receive from any thread */
    ..... SERVER WORKS .....
    msg_send(tid,buf,MAXBUF);   /* Reply */
  }
}

client()
{
  char buf[MAXBUF];
  ..... CLIENT WORKS .....
  msg_send_grp(gid,buf,MAXBUF);   /* Message to server group */
  msg_receive_grp(gid,buf,MAXBUF);   /* Reply from server group */
  ..... CLIENT WORKS .....
}
```

図7 サーバモデル使用例

4．1．性能評価

（1）並列処理方式による評価

本研究では、第2章で記述した並列処理方式の観点から、性能評価を行った。並列処理方式として、“m”を含めて、以下の3者を比較した。

1)プロセスを並列処理単位とした場合；

2)SKY－1のシステムコールのみを使用し、スレッドを並列処理単位とした場合；

3)スレッドインタフェース“m”（ライブラリ方式）を使用し、スレッドを並列処理単位とした場合；

なお、2)はスレッド制御の基本機能のみを利用した場合を意味する。表2に上記3種類の並列処理方式によるプログラムの性能測定結果について示す。従来のシングルプロセッサ用プログラムを1とした場合の速度比である。

測定結果から、全体的に、スレッドを使用して並列処理を行えば、プロセスを使用して並列処理を行う場合に比べて、速度的に有利となることが示されている。プロセッサ4台で評価すると、最大17％性能が向上している。

システムコールのみを使用した場合と“m”を使用したプログラムとでは、性能差はほとんどない。これらのプログラムにおいて使用されている“m”の関数は、スレッドの生成／終了、グループ管理、バリア同期、排他制御といった一部の関数のみであるが、他の関数においても、関数内部で実行するシステムコール数はほぼ同一である。すなわち、“m”の性能上の目標：「“m”を使用しても、性能が低下することはない」は達成されていると考える。後述するように、“m”では、グループ制御などの使用により、スレッドのユーザインタフェースが向上している。記述性の向上が結果的にシステムコールの実行回数減少につながり、“m”のオーバヘッドを補っていると考える。

（2）“m”の実現方式による評価

第3章で述べたように、現状の“m”の実現方式はライブラリ方式、サーバ方式の2つである。これらの方式では、並列処理支援のインタフェースという性格上、関数内で数回程度までの基本システムコールを実行する必要があり、関数をシステムコールで実現する方式に比べて、速度的に不利となる。本研究では、“m”を「どのように実現するか」によって、性能がいかに変化するかを評価した。評価する実現方式は以下の3方式である。

1)ライブラリ方式；

2)サーバ方式；

3)システムコール方式（“m”の関数をシステムコールに組み込む。なお、評価においては、システムコールとして組み込む関数を、プログラム中頻繁に使用されるバリア同期、排他制御の関数のみとした）；

上記3つの方式により、“m”を実現し、性能を測定した。表3に3つの実現方式の性能測定結果を示す。表3はライブラリ方式を1とした場合の性能比を示している。

表2 性能評価

Program type			CPU台数							
			プロセス数一定				プロセス数=CPU台数			
program	model	method	1	2	3	4	1	2	3	4
LINPACK (300*300)	Loop model	process	0.79	1.38	1.40	1.90	0.94	1.59	1.87	1.90
		thread	0.85	1.45	1.53	2.23	0.94	1.66	2.09	2.23
		"m"	0.83	1.46	1.53	2.12	0.96	1.65	2.03	2.12
	Broadcast model	process	0.87	1.44	1.55	2.20	0.95	1.67	2.09	2.20
		thread	0.87	1.47	1.58	2.25	0.94	1.65	2.10	2.25
		"m"	0.92	1.55	1.59	2.22	0.98	1.74	2.15	2.22
MANDEL (360*360)	Loop model	process	0.97	1.95	2.60	3.06	0.99	1.96	1.54	2.60
		thread	0.98	1.96	2.61	3.10	0.99	1.96	1.55	2.60
		"m"	0.97	1.96	2.60	3.09	0.99	1.96	1.54	2.60
	Broadcast model	process	0.95	1.92	2.89	3.81	0.98	1.97	2.94	3.92
		thread	0.95	1.92	2.90	3.85	0.98	1.97	2.96	3.92
		"m"	0.96	1.92	2.89	3.84	0.98	1.97	2.95	3.92

LINPACK ： 連立一次方程式解法(プロセス数4／可変)
MANDEL ： マンデルブロー集合計算(プロセス数8／可変)

表3 "m"の実現方式による評価

Program type		function	program CPU台数			
program	"m"		1	2	3	4
Barrier (LINPACK broadcast model)	Library	1.00	1.00	1.00	1.00	1.00
	Server	1.56	0.97	1.03	1.10	1.16
	System call	4.67	1.08	1.20	1.26	1.48
Lock (MANDEL broadcast model)	Library	1.00	1.00	1.00	1.00	1.00
	Server	1.05	0.98	0.97	0.96	0.95
	System call	2.19	1.01	1.01	1.01	1.02
Barrier & Lock (NEURO broadcast model)	Library	—	1.00	1.00	1.00	1.00
	Server	—	0.85	0.98	1.01	1.10
	System call	—	1.18	1.54	1.60	2.01

LINPACK ： 連立一次方程式解法(プロセス数4)
MANDEL ： マンデルブロー集合計算(プロセス数8)
NEURO ： ホップフィールド型ニューラルネットによる巡回セールスマン問題(プロセス数4)

関数をシステムコールとしてＯＳ内部に実現すれば、ライブラリ方式に比べて、単体性能では２～５倍、プログラム全体の性能としては最大２倍速度が向上することが判明した（この性能差はバリア同期、Monitor制御の関数をシステムコールとして構成するか否かの相違である）。

プロセッサ数を増加していくと、バリア同期に関しては、サーバ方式がライブラリ方式に比べて性能的に１０～１６％高いが、大差ではない。単純な排他制御であるMonitor制御では、逆に、サーバ方式が性能的に劣ることもある。ライブラリサーバ“M”とのメッセージ通信自体がシステムコールであること、および、そのシステムコールがそれほど高速でないことが原因と考える（同期型メッセージ通信であるため、ＵＮＩＸ*）のメッセージ通信より遅い）。

しかしながら、将来的には、ＯＳ機能の分散化が進み、これによって、メッセージ通信は高速化されていくものと考える。通信が高速化されることによって、サーバ方式の性能はシステムコール方式にある程度近づく。これに対し、他方式ではＯＳ機能の分散化による利益は少ない。将来のスレッドインタフェースの構成としては、高速化が要求されるもののみシステムコール化し、それ以外の機能に関してはサーバ方式を採用する必要があると考える。

４．２．ユーザインタフェースの評価

ユーザインタフェースに対する評価尺度として、プログラムの開発期間を用いることが望ましい。しかし、本研究では“m”の評価のため、各種の並列処理方式でプログラムを作成、比較している。開発期間を評価尺度とするためには、複数の並列処理環境下において、プログラミングを行う個人の技量差が表面に現われないほど大きなプログラムを作成させるなど、同一（同程度）のプログラム作成条件を形成しなければならない。

このため、本研究では別の評価尺度として、従来のシングルプロセッサ用のプログラムに対する改造の度合いを選択した。これによって、並列処理プログラムの開発期間の比をある程度予測することが可能である。

評価に用いたプログラムは性能評価で使用したプログラムと同一である。表４にこれらのプログラムの大きさとシングルプロセッサ用プログラムからの改造量を示す。

表４　ユーザインタフェースの評価

Program type		Line number method			
program	model	original	process	thread	"m"
LINPACK	Loop	1013	1091(102)	1189(198)	1068(77)
	Broadcast	1013	1200(218)	1200(214)	1098(111)
MANDEL	Loop	80	145(58)	205(114)	137(52)
	Broadcast	80	170(80)	181(92)	123(42)
NEURO	Broadcast	1556	2407(1090)	2453(1158)	1839(559)

プログラムは表３と同じ
()内が変更された行数を示す
プログラム作成者はＣ言語の初心者である

“m”を使用すると、改造量は他並列処理方式と比べて１／２程度となる。単純に換算すれば、並列処理プログラムに改造する期間が１／２程度となることが予想される。また、スレッドをシステムコールのみで制御する場合には、プロセスに比べて明らかにユーザインタフェースが低下している。しかしながら、“m”を用いることにより、プロセスと同程度またはプロセス以上のユーザインタフェースを提供することが可能であることが証明された。

５．おわりに

並列処理用ＯＳ・ＳＫＹ－１では、プロセス内で実行環境を共有して並列に動作するスレッドを導入した。スレッドを用いることにより高速化が図れる反面、複雑なプログラミングが要求されるため、ユーザインタフェースが著しく低下する。そこで我々は、スレッドの普及策の一環として、スレッドインタフェース“m”を開発した。

“m”では、スレッドのグループ化機能を有し、互いに関連するスレッド同志をまとめることにより、ユーザインタフェースを高めている。“m”とそのグループ機能を使用することにより以下の利点が得られた。

1)従来のプログラムを並列処理プログラムに改造するための期間を１／２程度に縮小させることができる。

2)性能的にはプロセスを最大１７％上回り、かつ、スレッド制御用システムコールを使用する場合と同程度となる。

また、本研究では、将来ＯＳの構成を考え、“m”をサーバプロセス化したライブラリサーバ“M”を開発、評価し、その有効性を確認した。

*）ＵＮＩＸはＡＴ＆Ｔ社のＢｅｌｌ研究所で開発され、ＡＴ＆Ｔ社がライセンスしているＯＳです。

参考文献

１）山口伸一朗他：並列処理用ＯＳ・ＳＫＹ－１　－開発構想－，情報処理学会第３９回全国大会論文集

２）齊藤雅彦他：並列処理用ＯＳ・ＳＫＹ－１のメモリ管理方式，情報処理学会第３９回全国大会論文集

３）上脇正他：並列処理用ＯＳ・ＳＫＹ－１のスケジューリング方式，情報処理学会第３９回全国大会論文集

４）齊藤雅彦他：並列処理用ＯＳ・ＳＫＹ－１のスレッド制御用ライブラリ，情報処理学会第４０回全国大会論文集

５）Accetta,M. et.al.：Mach:A New Kernel Foundation for UNIX Development, Proceedings of USENIX 1986 Summer Conference, pp.93-112

６）下山智明：Sun Lightweight Processプログラミング，UNIX MAGAZINE, pp.90-115, 1988/7

７）穂積元一：ＴＯＰ－１オペレーティングシステム，情報処理学会オペレーティングシステム研究会報告Ｎｏ．４０，1988/9

８）村岡洋一：並列処理，昭晃堂，pp.10-16

９）Cheriton,D.R.：Distributed Process Groups in the V Kernel, ACM Trans. Computer Systems Vol.3 No.2, pp.77-107

10）大久保英嗣他：実時間オペレーティングシステムR²/Ｖ２におけるタスクグループの実現，情報処理学会論文誌 Vol.31 No.2, pp.275-287, 1990/2

11）Hoare,A.C.R.：Monitors:An Operating System Structuring Concept, Comm. ACM Vol.17 No.10, pp.549-557