計算機アーキテクチャ 117－7
（1996．3．5）

マルチスレッド言語のための実行時ライブラリの実装

堀 敦史[†1] 手塚宏史[†1] 石川 裕[†1]
高橋俊行[†3] 曽田哲之[†2] 堀川 勉[†4]
小中裕喜[†1] 前田宗則[†1]

昨今のLAN技術の発達によりワークステーションを高速で接続することで，従来の並列マシンに匹敵するほどの通信性能を確保できることが認識されつつある．通信が高速になった分，スレッドにまつわるオーバヘッドも低減されるべきであるという考えに基づき，我々は高速なスレッドライブラリを実装した．その結果，ローカルスレッド呼び出しに1.4マイクロ秒，リモートスレッド呼び出しに30マイクロ秒という高性能を得た．本稿では，我々が実装した高速スレッドライブラリにおける高速化の技法について述べる．

Runtime Library Implementation for Multi-threaded Language

ATSUSHI HORI ,[†1] HIROSHI TEZUKA ,[†1] YUTAKA ISHIKAWA ,[†1]
TOSHIYUKI TAKAHASHI ,[†3] NORIYUKI SODA ,[†2]
TSUTOMU HORIKAWA ,[†4] HIROKI KONAKA [†1]
and MUNENORI MAEDA[†1]

With recent LAN technologies, cluster of workstations connected by a high-speed LAN can achieve high performance comparable to parallel machines. Based on the assumption in which as higher the speed of LAN, as lower the overhead thread implementation, we have implemented a low-overhead thread library. With our runtime library, it takes 1.4 μsec to invoke a local thread and 30 μsec to invoke a remote thread. In this paper, we describe our implementation techniques to achieve such performance.

1. はじめに

我々は，マルチスレッド言語であるMPC++[1)~3)]のための実行時ライブラリをワークステーションクラスタ上に実装した。対象としたワークステーションクラスタは，9台のSun Microsystems社製のSparcStation 20（クロック周波数75MHz）をMyricom社のMyrinet LAN[4)]で結合したものである．LAN技術の発達によりMyrinetのようなギガビットクラスの高速LANの出現により，スレッドのオーバヘッドは通信の高速化にみあった分だけ低減されるべきと考える．

このような考えに基づき，我々は実行時ライブラリの可搬性を著しく損なわない限りにおいて，できるだけ速度を重視したスレッドライブラリを開発した．その結果，スレッド切替時間が1.4 μsec，遠隔スレッドの起動が30 μsecという高速なスレッドライブラリを実現できた．本稿ではそこで実現されたいくつかの技法について論じる．本稿での報告される数値は全て，先に述べたワークステーションクラスタ上での計測結果である．

2. MPC++

簡単なMPC++プログラムの例を**図1**に示す．この例では，`pong()`関数をスレッドとして`REMOTE_OR_LOCAL`変数で示されたプロセッサ上に起動しその終了を待つ、という処理を`ping()`関数の引数で指定された`loop`回数だけ繰り返す．

図2の`ping()`関数の定義は図1のものと等価である．3行目で宣言されている`entry`という型は，スレッド間の同期のためにMPC++で導入されたものである．`entry`型の変数`ent`は関数スレッド`pong()`の呼出時に指定されることで，`pong()`スレッドが非同

†1 新情報処理開発機構つくば研究センタ
Tsukuba Research Center, Real World Computing Partnership
†2 （株）SRA
Software Research Associates, Inc.
†3 東京大学
University of Tokyo
†4 （株）管理工学研究所
Kanri Kogaku Kenkyusho, Ltd.

```
1 #include <mpcxx.h>
2 void pong( void ) { return; }
3 void ping( int loop ) {
4   int i;
5   for( i=0; i<loop; i++ ) {
6     pong()@[REMOTE_OR_LOCAL];
7   }
8 }
```

図1 MPC++ プログラムの例 (1)
Fig. 1 MPC++ program (1)

```
1 void ping( int loop ) {
2   int i;
3   entry() ent;
4   for( i=0; i<loop; i++ ) {
5     pong()@(ent)[REMOTE_OR_LOCAL];
6     ent()::;
7   }
8 }
```

図2 MPC++ プログラムの例 (2)
Fig. 2 MPC++ program (2)

期に呼び出される．6 行目で，pong() スレッドの終了を待つ．

MPC++ コンパイラはフロントエンド処理系とバックエンド処理系から成り，バックエンド処理系においてターゲットマシンに応じた最適化がなされる．ワークステーションクラスタのような一般的なマシンの場合には，フロントエンド処理系は C++ のコードを生成し，それを GNU g++ でコンパイルする方式とした．図 1 のプログラムの C++ 中間コードの出力例を**図 3** に示す．

関数 ping() における処理では，まずエントリオブジェクトを初期化し（17 行目），そのエントリに対するトークン（意味的には，トークンとはエントリに対する大域的な参照のことである）を渡して，pong() スレッドを呼び出す（23 行目）．__threaded_pong() はコンパイラが生成した関数であり，スレッドとして呼び出される関数 pong() の wrapper 関数になっている．引数は構造体にパックされて渡される．__threaded_pong() 関数では，引数構造体をアンパックし（4 行目），スレッドの本体である pong() 関数を呼び出す（8 行目）．関数から戻って来たらスレッドの終了を通知するためのトークンを取り出して（9 行目），スレッドの起動元に通知する（11 行目）．そして最後にスレッドを消滅させる（12 行目）．

図 1 プログラムの 1 行目で指定されている include ファイルには，この MPC++ プログラムの実行に必要なライブラリに関する各種定義だけでなく，MPC++ コンパイラが必要に応じて実行時ライブラリ関数の呼び出しなどに変換するための「メタプログラム」が定義されている．MPC++ コンパイラはメタプログラムを通じ，実行時ライブラリに合わせた C/C++ コードを出力することができる．さらに MPC++ のメタプログラムは，C/C++ に対する言語拡張をユーザに解放

```
1 void pong(void){return ;}
2 static void __threaded_pong() {
3   extern void pong();
4   struct _arg_threaded_pong *argp =
5     (struct _arg_threaded_pong *)
6       _mpcGetArgp();
7   MPCtoken *tkn;
8   pong();
9   tkn = _mpcGetReturnToken();
10  if (tkn!=0)
11    _mpcSendToToken(tkn, (char *) 0, 0);
12  _mpcThreadTerminate();
13 }
14 void ping(int loop){
15   int _retval___SYMO_;
16   MPCtoken *_tkn___SYMO_;
17   MPCentry _ent___SYMO_
18     (&_tkn___SYMO_,
19      (char *) &_retval___SYMO_,
20      sizeof((_retval___SYMO_)));
21   int i;
22   for(i = 0; i<loop; i++){
23     (_mpcRemoteAsyncInvoke(1,
24           (void (*)(void)) __threaded_pong,
25           (char *) 0,
26           0,
27           (MPCtoken *) _tkn___SYMO_),
28      _mpcWait(&_ent___SYMO_),
29      _retval___SYMO_ );
30   }
31 }
```

図3 MPC++ コンパイラが生成した C++ コード
Fig. 3 Intermediate Output of MPC++ Compiler

するため，新たな並列構文の導入などが容易に記述かつ試行することが可能となっている[2),3)]．

3. Runtime Library

MPC++ の実行時ライブラリはポータビリティを考えて，通信部分とそれ以外の部分分離可能になっている．現在までに UDP と Myrinet を通信に用いるバージョンが開発されている．他のプロセッサ上のプロセスは実行時ライブリが rsh コマンドを用いて生成される．また，SIGSEGV などの異常を知らせるシグナルを実行時ライブラリが検知した場合や，実行時ライブラリ自身が異常状態に陥った場合は，XTerm/GDB プロセスを立ち上げ，自分自身に attach するようにしている．このリモートデバッグ方式は非常にシンプルではあるが，分散環境におけるデバッグの 1 手法として非常に有効である．

MPC++ のスレッド実行に必要とされる実行時ライブラリの機能としては， i) スレッドの生成および消去，ii) スタックを保存したスレッド実行の中断および再開，iii) リモートメモリアクセス，iv) および上記操作に伴う通信，が必要となる．具体的には，**表 1** にあるような関数が提供されている．

エントリとトークンによる待ち合わせを除き，これらの機能はいずれもマルチスレッドの実現に必要なも

表1　実行時ライブラリが提供する関数
Table 1　Functions provided by the runtime library

関数名	関数の説明
_mpcSelfPE	プロセッサ番号を返す
_mpcNumPE	プロセッサ数を返す
_mpcInitEntry	エントリを初期化する
_mpcGetReturnToken	スレッドのトークンを返す
_mpcThreadTerminate	スレッドを終了する
_mpcGetArgp	引数構造体ポインタを返す
_mpcSelfAsyncInvoke	ローカルスレッドの起動
_mpcRemoteAsyncInvoke	リモートスレッドの起動
_mpcWait	トークンを待つ
_mpcSendToToken	トークンに値を束縛する
_mpcRemoteMemRead	リモートメモリのコピー
_mpcRemoteMemWrite	リモートへのメモリコピー

のばかりであることから，MPC++ 以外のマルチスレッド言語への応用も可能と考える．

3.1 Thread Switching

MPC++ 実行時ライブラリでは，スレッド切替の回数を減らすため，コルーチン的にスレッドが切り替わる．UNIX でスレッド切替を実現する場合，`setjmp()/longjmp()` を用いるのが普通である．しかしながら，`setjmp()/longjmp()` は，シグナルの設定をも退避／復帰する．我々の計測では，シグナルの退避／復帰を行なわない `_setjmp()/_longjmp()` との差は約 20 μsec あることが判明した．シグナル設定の退避／復帰は，仮想プロセッサとしてのスレッドを実現するためには必須である．しかしながら，20 μsec は Myrinet の通信レイテンシに比べ（後述）大きなオーバヘッドであるため，MPC++ ではシグナル設定の退避／復帰はおこなわないものとした．

一方，Sparc プロセッサのようにレジスタウィンドウを有する場合，スレッド切替時にはレジスタウィンドウをフラッシュする必要がある．SunOS ではこのためのシステムコールが存在しており（`ST_FLUSH_WINDOWS`），`_longjmp()` 内で実際にこのシステムコールが呼ばれている．計測によると，`ST_FLUSH_WINDOWS` のシステムコールにはおよそ 5 μsec 必要であることが判明した．この時間は `getpid()` に要する時間にほぼ等しいことから，5 μsec はシステムコールのオーバヘッドであると推測される．`ST_FLUSH_WINDOWS` システムコールは，プロセッサが有するレジスタウィンドウの数をユーザが知る必要がないために導入されたものであるが，その代償としてシステムコールのオーバヘッドを受け入れなければならない．

そこで，レジスタウィンドウの数を既知のものとし，`ST_FLUSH_WINDOWS` システムコールを呼ばない方法を考える．具体的には図4 に示したアセンブラプログラムにあるように，レジスタウィンドウの数より 1 減じた回数だけ `save` 命令を続けて発行し，その後で `restore` 命令を同じ回数繰り返す．つまり，レジスタウィンドウを一旦オーバフローさせることで，レジスタウィンドウのメモリへの退避を実現している．

```
 1 ! void _mpc_longjmp( jmp_buf, int )
 2 ENTRY(_mpc_longjmp)
 3     save %sp,-SA(WINDOWSIZE),%sp;
 4       :          ! NWINDOWS-1 回繰り返す
 5     save %sp,-SA(WINDOWSIZE),%sp;
 6     restore;
 7       :          ! NWINDOWS-1 回繰り返す
 8     restore;
 9     ld  [%g1 + 0x8], %sp
10       :          ! fp などの復帰
11       :          ! i レジスタの復帰
12       :          ! g レジスタの復帰
13     retl         ! return( int )
14     mov %o1, %o0 ! delay slot
```

図4　レジスタウィンドウのフラッシュ
Fig. 4　Flushing of Register Windows

図5 は，普通の `call/return`，`_setjmp()/_longjmp()`，および `ST_FLUSH_WINDOWS` を用いない `_mpc_setjmp()/_mpc_longjmp()` に要する時間を，整数引数の数が 4 の場合と 16 の場合のそれぞれについて，関数呼出の深さを横軸に，制御が戻ってくるまでに要した時間を縦軸にとったグラフである．また，gcc のコンパイルオプションとして，レジスタウィンドウを使う場合（-O）と，使わない場合（- O -mflat）でも比較をした．

左側のグラフでレジスタウィンドウを使用するコードの場合，レジスタウィンドウのオーバフローの影響がかなり大きいことに注目される．また，`_setjmp()/_longjmp()` と `_mpc_setjmp()/_mpc_longjmp()` での比較では，およそ 6 μsec の差でグラフは並行線となっており，`ST_FLUSH_WINDOWS` を省いた効果が現れている．一方，レジスタウィンドウを使わない場合（右側のグラフ）では，`_mpc_setjmp()/_mpc_longjmp()` のオーバヘッドは単なる `call/return` のそれにかなり接近していることが分かる．

マルチスレッド言語のようにスレッド切替が頻繁に生じるような場合は，レジスタウィンドウを使わない方が効率が良い場合があることを示唆するものと考えられる．`_mpc_setjmp()/_mpc_longjmp()` の実装では，-mflat オプションでコンパイルしたコードの混在を許しているため，スレッド切替の頻度やスレッド内での関数コールの深さなどを考慮して，-mflat コンパイルオプションを選択することが可能である．

3.2 Communication

現在，MPC++ 実行時ライブラリは Myrinet あるいは UDP を用いて実装されている．以下は Myrinet を用いたものについての説明である．Myrinet の諸元を 表2 に示す．

我々は Myrinet[4] 上に専用のドライバソフトウェアを開発し，これを PM と呼んでいる．PM の設計において，i) OS への割込，システムコール，および ii) packetize，depacketize 時におけるメモリコピー，

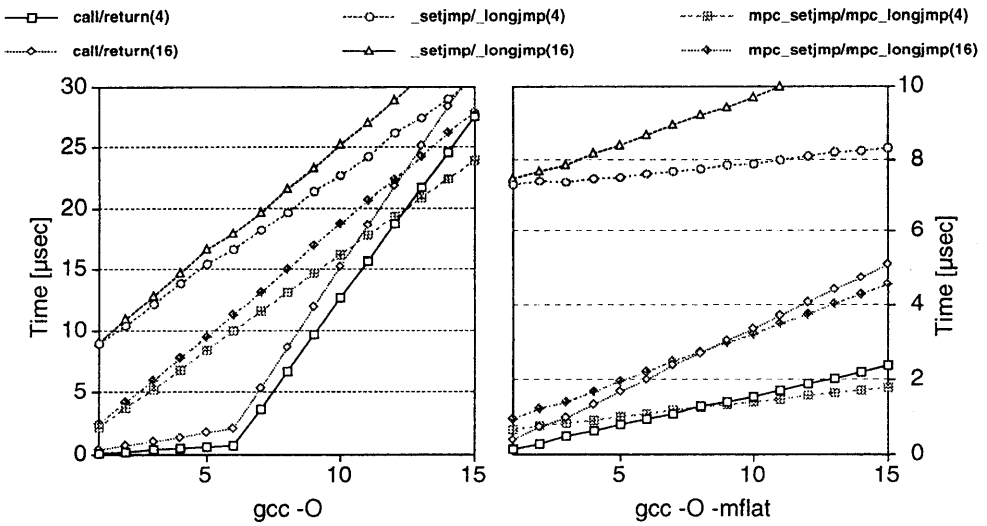

図5　大域ジャンプの時間
Fig. 5　Long-jump Time

表2　Myrinet の諸元
Table 2　Myrinet Specifications

Link Speed	80 MB/sec x 2 (Full-Duplex)
Host Interface	S-Bus (PCI)
Network Topology	Arbitrary
Switch Routing	Cut Through
Packet Length	Arbitrary

http://www.myri.com より抜粋.

をできるだけ避けることに留意した．図6に PM におけるデータの流れを示す．カーネル内の送受信バッファをユーザ空間にメモリマップすることでコピーを防いでいる．実行時ライブラリはこのメモリマップされたバッファ上でパケットを生成あるいは分解する．Myrinet インターフェイスボード上の制御プロセッサは，メインのプロセッサとは非同期にメッセージの送受信をおこなう．送受信バッファは FIFO バッファとなっており，メッセージの送信順序は保存される．Myrinet は原理的に伝送途中でのパケット喪失はなく，ハードウェアでフロー制御を行なっていることから，PM での信頼性向上のためのプロトコルはない．しかしながら，長時間パケットがブロックされるとハードウェアのタイムアウトによりリセットが生じるため，PM レベルで別途フロー制御をおこなっている．

PM が提供するメッセージ送受信に関する関数インターフェイスを表3に示す．図7に PM におけるパケットの往復時間（ラウンドトリップ，往復とも同じパケット長）と，バンド幅（片道）の計測結果を示す．パケット長が 8 バイトの場合の転送時間は片道 24 μsec であり，4096 バイトにおけるバンド幅は，約 32 MB である．

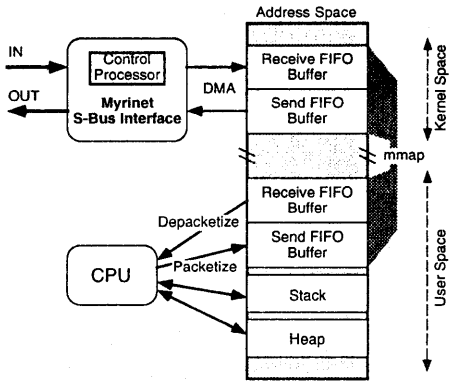

図6　PM におけるデータの流れ
Fig. 6　Data Flow in PM

表3　PM ライブラリが提供する関数
Table 3　Functions provided by PM library

関数名	関数の説明
_pmGetSendBuf	送信のための領域を確保する
_pmSend	確保した領域を送信する
_pmReceive	受信した領域を返す（もしあれば）
_pmPutReceiveBuf	受信領域を解放する

Deadlock の回避

MPC++ のようにメッセージの送受信が任意のタイミングで発生するような場合，互いに送信し合う通信バッファが満杯となることで，デッドロックを生じる可能性がある．そこで，メッセージ送信のためのバッファ領域確保が（満杯により）失敗した場合，確保可能になるまでメッセージの受信処理をおこなうことにより，この種のデッドロックは回避可能である（図8）．

MPC++ 実行時ライブラリでは，リモートメモリア

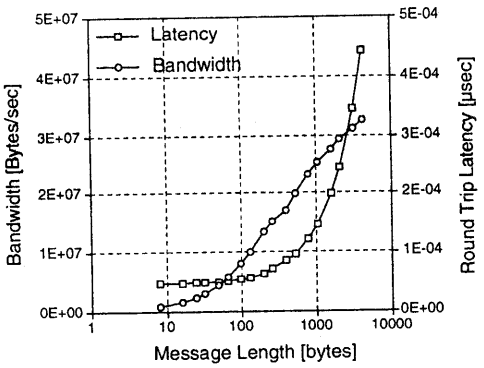

図 7　PM の転送性能
Fig. 7　PM Performance Curves

クセス要求に対するサービスは，スレッドを生成せずにおこなっている．このため，送信バッファ確保に失敗した時の受信（図 8，5 行目）において，リモートメモリアクセスに対する返答メッセージは送信できない．新たにデッドロックが生じる可能性があるからである．このため，このような場合のリモートメモリアクセスに対する返答メッセージは，別なキューを設け，送信バッファが確保可能になるまで送信を遅らせている．

```
1 void *alloc_send_buffer( int length ) {
2   void *bufp;
3   while( _pmGetSendBuf( &bufp, length, ... )
4          == ENOBUF )
5     receive_messages( ... );
6   return( bufp );
7 }
```

図 8　送信バッファ領域の確保
Fig. 8　Send Buffer Allocation

4. 予備評価

本章では，これまで説明した MPC++ 実行時ライブラリの予備評価をおこなう．**表 4** は，図 1 のプログラムを 1 プロセッサ上で実行した結果（loop 引数の値は 1,000,000）である．ここではレジスタウィンドウと ST_FLUSH_WINDOWS の関係を知るために，gcc のコンパイルオプションとして -O のみを指定した場合と -O -mflat を指定した場合，また、_setjmp()/_longjmp() を用いた場合と _mpc_setjmp()/_mpc_longjmp() を用いた場合，の 4 通りで計測した．

この結果から，i) ST_FLUSH_WINDOWS システムコールを呼ばないことで 15 μsec 以上も高速化された，ii) その結果，レジスタウィンドウを用いないことの効果が ST_FLUSH_WINDOWS なしの場合により顕著に現れた，ことが分かる．

tt ST_FLUSH_WINDOWS もレジスタウインドウも用いない場合，表 4 にあるようにローカルなスレッドの起動から終了を知るまでの時間が 3 μsec を切る程度にまで高速化できたことになる．ちなみに同じユーザレベルスレッドである SunOS が提供する LWP と比較した場合，MPC++ ランタイムライブラリのスレッド切替時間は一桁以上高速である．

表 4　Pingpong（ローカル）
Table 4　Local Pingpong

longjmp	gcc option	Time [sec]
_longjmp	-O	19.67
	-O -mflat	18.34
_mpc_longjmp	-O	4.330
	-O -mflat	2.871

表 5 は，pong() スレッドをリモートプロセッサで実行した場合の計測結果である．ここで，コンパイルオプションは -O -mflat を指定し，_mpc_setjmp()/_mpc_longjmp() を用いている．条件として，通信待ちにプロセスがブロックする場合（表中，block）とビジーウェイトする場合（表中，busy）を指定した．また，参考までに UDP（10Base-T を使用．100Base-T でもほぼ同じ）での実行結果も示す．UDP の場合において受信に aioread() を用いた非同期（割込）I/O を用いた場合も示した．

表 5　Pingpong（リモート）
Table 5　Remote Pingpong

Communication	Wait	Time [sec]
PM	block	580.3
(Myrinet)	busy	60.98
UDP	block	801.5
(10Base-T)	async-busy	822.1
	busy	626.4

ブロックウェイトによる場合，通信待ちにおいてプロセス切替のオーバヘッドおよび割込のオーバヘッドが生じる．その結果，ビジーウェイトの方が 10 倍近く高速であった．同様の現象は UDP においても見ることができる．PM を用いたビジーウェイトでは，pong() スレッドをリモートで実行した場合，1 回あたり約 61 μsec である．前述したように PM レベルでのレイテンシは往復で 48 μsec であり，表 4 に示したようにローカルスレッドの起動からその終了を受けとるまでに 3 μsec かかる．残りの 10 μsec の大半はパケットの組立と分解に要するコストと考えられる．

表 9 は非同期（終了を待たない）リモートメモリ書き込みにおける最小転送間隔とその時のバンド幅を，転送量を変えて計測したものである（コンパイルオプションは -O -mflat，_mpc_setjmp()/_mpc_longjmp()，PM，ビジーウェイト）．表 7 と違い，転送間隔が対数スケールになっている．PM は最大 4 KB まで転送を許しているため，実行時ライブラリにおいて 4 KB を越えるメッセージは複数パケッ

トに分割されてバースト状に転送される．転送量が 2 KB を越えたところで飽和しているのはこの影響と考えられる．2 KB まででは表7に示した PM の転送性能の約 9 割の性能を引き出している．

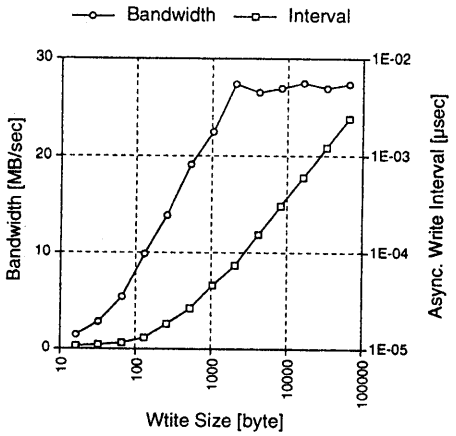

図9　非同期リモートメモリ書き込みの転送性能
Fig. 9　Async. Remote Memory Write

ま と め

昨今の LAN 技術の発達によりワークステーションを高速で接続することで，従来の並列マシンに匹敵するほどの通信性能を確保できることが認識されつつある（例えば文献[5]）．マルチスレッドを用いた並列プログラムは，通信レイテンシを隠蔽することが可能である．しかしながら，マルチスレッドのそもそもの生い立ちは，プロセスから派生したものであり，仮想プロセッサの概念に囚われていると考えることが出来る．通信が高速になった以上，スレッドにまつわるオーバヘッドも低減されるべきである．

本稿で紹介した MPC++ 実行時ライブラリは，仮想プロセッサとしてのスレッドでなく，「制御構造のひとつ」としてのスレッドを意味するものと考えることが出来よう．仮想プロセッサを実装するのに必要なシグナルハンドリングにはいくつかのシステムコールが伴い，Myrinet の通信レイテンシにくらべ大き過ぎるオーバヘッドとなる．

高速なスレッドの実装に関する研究のひとつの方向として，スレッドがサスペンドする際にスタックを保存しない方法がある[6)~8)]．この方式ではスレッドを用いたライブラリを構築できないという大きな欠点がある．我々はスタックを保存するというスレッドの実装において，いくつかの高速化の技法を適用し，スタックを保存しない方式に迫る性能を得ることが出来たと考えている．

我々の MPC++ 実行時ライブラリは，現在，富士通 AP1000 および Thinking Machines 社の CM-5 上に移植をおこなっている．今後，並列アプリケーションにより評価を行ない、さらなる改良を施す予定である．

参 考 文 献

1) 石川裕, 堀敦史, 小中裕喜, 前田宗則, 友清孝志: 並列プログラミング言語 MPC++の実現, 並列処理シンポジウム JSPP'94, pp. 105–112 (1994).
2) Ishikawa, Y.: MPC++: Massively Parallel, Message Passing, Meta-Level Programming C++, *Parallel Object Oriented Methods and Application'94* (1994).
3) Ishikawa, Y.: Meta-Level Architecture for Extendable C++, Technical Report TR–94024, RWC (1995).
4) Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N. and Su, W.-K.: Myrinet: A Gigabit-per-Second Local Area Network, *IEEE Micro*, Vol. 15, No. 1, pp. 29–36 (1995).
5) Pakin, S., Lauria, M. and Chien, A.: High Performance Messaging on Workstations: Illinoi Fast Messages (FM) for Myrinet, *Supercomputing'95* (1995).
6) Freeh, V. W., Lowenthal, D. K. and Andrews, G. R.: Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations, *OSDI*, pp. 201–212 (1994).
7) Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, *PPOPP*, pp. 207–216 (1995).
8) Sloman, B. and Lake, T.: Featherweight Threads and ANDF Compilation of Concurrency, *Euro-Par'95 Parallel Processing* (Haridi, S., Ali, K. and Magnusson, P.(eds.)), Lecture Notes in Computer Science, Vol. 966, Springer-Verlag, pp. 457–469 (1995).