ハイパフォーマンス
コンピューティング 77－8
（1999. 8. 2）

MPC++ Multi-Thread Template Library の MPI による実装と性能評価

栄 純明[†] 石川 裕[††]
松岡 聡[†] 小川宏高[†]

並列言語 MPC++ は新情報処理開発機構で開発されたクラスタシステムソフトウェアである SCore 以外の環境では利用できなかった．我々はプラットホームポータビリティのために，通信レイヤに MPI を用いて MPC++ を実装したが，MPI の特性はプラットホーム毎に異なるため，性能の可搬性が問題となる．測定の結果データサイズが 8Kbytes 以上の時には通信のオーバヘッドは無視できることが分かった．さらに Nas Parallel Benchmarks の CG カーネルベンチマークでは，ノード数が少ないときには MPC++ on MPI で書かれたものが MPI で書かれたものに性能面で迫る．しかしノード数が増加すると，ネットワーク性能の低いコモディティプラットホームでは性能が著しく低下するのに対し，高速なネットワークを持つ MPP 上と，高性能なネットワークを持ったコモディティプラットホーム上ではスケールした．この結果 MPC++ on MPI は高性能なネットワークを持ったプラットホーム上ではその有効性が確認できたが，コモディティネットワークに対しては更なる最適化が必要であることが分かった．

Implementing and Evaluating the MPC++ Multi-Thread Template Library on Multiple MPI Platforms.

YOSHIAKI SAKAE,[†] YUTAKA ISHIKAWA,[††] SATOSHI MATSUOKA[†] and HIROTAKA OGAWA[†]

Our parallel programming language MPC++ has been only available on the SCore cluster system software developed at the Real World Computing Partnership. In order to achieve better portability amongst multiple platforms, the scope of MPC++ is being widened via implementation using MPI as the underlying communication layer. This brings up the question of applicability, since MPI performance varies considerably on different platforms. Our evaluation results show that the communication overhead is negligible when the data size is larger than 8 Kbytes. Furthermore, the CG kernel benchmark of Nas Parallel Benchmarks written in MPC++ using MPI achieves a comparable speed to one written in MPI when the number of nodes are small. However, increase in the number of nodes causes severe loss of performance for commodity platforms with low network performance, while it continues to scale well on those with high-performance networks, as well as MPIs on MPPs with fast communication infrastructure. These results suggest that, although MPC++ on MPI is viable on high-performance platforms, we need further research on optimizing for commodity networks.

1. はじめに

コモディティハードウェアとギガビット級のネットワークで構成されるクラスタ型並列計算機が注目を浴びている．コモディティハードウェアを使用したクラスタ型並列計算機では，特殊な OS ではなく単体でも使用していた OS が使用できる・導入コストが安い・技術の進歩の恩恵を受けやすい，と言った利点がある．

クラスタ型並列計算機上では開発言語として C/C++ や Fortran に，MPI もしくは PVM のようなメッセージ通信ライブラリを組合わせて用いることが多いが，MPI や PVM ではメッセージの受け渡しを直接プログラマが記述しなくてはならないため，大規模なプログラムを書くには負担が大きい．そこでプログラマが直接メッセージ送受信の記述をしなくてもすむように，より高いレベルで抽象化したプログラムを記述できる並列言語が開発されてきた．

新情報処理開発機構[1] ではパーソナルコンピュータやワークステーションと，ネットワークに Myricom 社[2] の Myrinet を用いたクラスタ型並列計算機を構築し，その上で SCore[3] と呼ばれる並列環境を実

† 東京工業大学 情報理工学研究科 数理・計算科学専攻
Tokyo Institute of Technology
†† 新情報処理開発機構
Real World Computing Partnership

現するシステムソフトウェアを開発している．SCore 上のプログラム開発，また SCore 自身の記述には，C++ のテンプレート機能を用いて実現した並列言語 MPC++[4] が新情報処理開発機構で開発され使用されている．

MPC++ で開発された有用なプログラムは多く，SCore 以外の環境で MPC++ プログラムを効率を損なうこと無く動作させたいという要求は高い．解決方法として，1) SCore 環境全体を Myrinet 以外のネットワーク上に移植する，2) MPC++ の通信部分を変更して MPI のような汎用的な通信ライブラリ上に実装する，の 2 つが考えられるが，MPI は現在すでに SMP でも MPP でもコモディティーハードウェアを用いたクラスタでも，様々なところで動作しているため，後者のアプローチの方がポータビリティという観点から勝っている．そこで我々は MPC++ Version 2.0 level 0 (MPC++ MTTL) を MPI 上に実装 (MTTL-MPI[5]) し，評価を行った．

以下，2 章で MPC++ MTTL の概要を述べ，3 章で実装の説明する．4 章で性能評価を行い，関連研究に関して 5 章で触れ，6 章で本稿をまとめる．

2. MPC++ MTTL の概要

MPC++ Version 2.0 は level 0 と level 1 からなり，level 0 では C++ 言語仕様を拡張，変更することなく，C++ の持つテンプレートおよびクラス機能を用いて，分散メモリ型並列計算機上でのプログラミングを容易にするために，関数の同期および非同期呼び出し機能・同期構造体・グローバルポインタなどを提供している．level 1 ではメタレベルアーキテクチャやアプリケーションに特化した言語拡張を行う．

本研究では MPC++ Multi-Thread Template Library (MTTL) と呼ばれる level 0 を MPI 上に実装した．以下 MPC++ MTTL の説明を行う．

2.1 MPC++ のプログラミングモデル

MPC++ は分散メモリ環境での SPMD プログラミングモデルをサポートする．各プロセスには複数のスレッドが含まれ，これらはプリエンプティブではない．つまり，スレッドの実行は同期を行うかその実行が終了するまで止ることはない．

全ての変数は各プロセスに局所的である．ファイルスコープで定義された変数は各プロセスに割付けられる．

MPC++ プログラムのメインルーチンである `mpc_main` はファイルスコープ変数の初期化後に，プロセッサ 0 だけで実行される．他のプロセッサ上のプロセスは，ファイルスコープ変数の初期化後メッセージハンドラの実行に入り，メッセージを待つ．

2.2 MPC++ MTTL の提供する機能

2.2.1 invoke, ainvoke 関数テンプレート

`invoke` 関数テンプレートは，同期的にローカルもしくはリモート関数呼び出しを提供する．つまり `invoke` を呼び出したスレッドは `invoke` した関数から帰ってくるまでブロックされる．`ainvoke` 関数テンプレートは非同期呼び出しを行う．`(a)invoke` の呼び出しには新しいスレッドの作成と，スレッドのコンテクストスイッチを伴う．

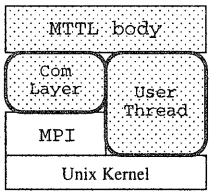

図 1 MTTL-MPI の構造

2.2.2 同期構造体 Sync クラステンプレート

multiple readers/writers 通信モデルを実現するために FIFO 型の通信バッファとして振る舞う同期構造体 Sync クラスを提供する．writer がデータを書き込むと Sync オブジェクトの queue に入れられ，reader がデータを読むと queue の先頭からデータを削除する．reader が読み出すときに queue にデータがないときには reader スレッドはブロックされる．また peek 命令によって queue からデータを削除せずに読み出すこともできる．そのほか queue の長さを得る queueLength 命令なども提供する．

2.2.3 GlobalPtr クラステンプレート

GlobalPtr クラステンプレートは，ほかのプロセッサ上のメモリを参照するためのグローバルポインタの機能を提供する．GlobalPtr クラスでは配列へのグローバルポインタ，グローバルポインタへのグローバルポインタ，グローバルポインタの指すリモートオブジェクトのメンバファンクションの起動，リモートメモリ read/write なども実現している．

3. MTTL-MPI の実装

MTTL-MPI の実装はポータビリティを高めるため，MTTL を変更し図 1 に示すように通信レイヤ，スレッド部分を分離している．スレッドはユーザスレッドとして実装してあり，この部分のみがプラットホーム依存である．

3.1 通信レイヤ

MTTL-MPI は通信レイヤに関してポータブルにするために，通信レイヤに対するラッパーとして働く 8 個のプリミティブを用意し，上位レイヤからはそれを用いて通信を行う．

MPI 以外の通信レイヤ上に移植するためには，この 8 個の通信プリミティブを，使用する通信レイヤを用いて実装すればよく，必要な変更量は，MPI の場合でコードにして約 50 行である．8 個の通信プリミティブを表 1 に示す．

3.2 ユーザスレッド

MTTL-MPI のスレッドはユーザスレッドとして実装されている．現在 Hitachi SR2201, SunOS 4.1.x, SunOS 5.5.x/sparc, NetBSD/i386, Linux/i386 上のユーザスレッドが実装済で，また NEC Cenju-4 上へ移植中であるが，他のプラットホームに移植するためにはこのユーザスレッド部，具体的にはプログラム内での大域ジャンプを行う setjmp, longjmp と jmp_buf のマクロ定義を各プラットホーム向けに変更すればよ

表 1 MTTL-MPI の通信プリミティブ

_pmttl_cominit	通信の初期化を行う
_pmttl_comfinish	通信の終了処理を行う
_pmttl_prob	受信メッセージのプローブ
_pmttl_asend	非同期送信
_pmttl_nsend	ノンブロッキング送信
_pmttl_senddone	ノンブロッキング送受信の完了確認
_pmttl_brecnode	特定のノードからの受信
_pmttl_brecany	任意のノードからの受信

く，Linux/i386 の場合で 10 行程度である．

スレッドの管理は各プロセッサ上で Thread クラスのスタティック変数を用いて行われる．管理されるデータは，動作できる状態にあるスレッドの数・サスペンドしているスレッドの数・消滅予定のスレッドへのポインタ・現在動作中のスレッドへのポインタなどである．

各スレッドオブジェクトは queue として管理されており，それぞれ queue 上の前後のスレッドへのポインタ，送信・受信バッファ，スタック，そのスレッド上で実行する関数へのポインタとその関数の引数へのポインタなどを管理する．

3.3 メッセージハンドラ

MTTL の機能は全てメッセージハンドラベースである．すなわち，各プロセッサ上では少なくとも 1 つのメッセージハンドラ (`_mpcRecHandler`) がメッセージの到着を待ち，到着したメッセージのタイプに応じてディスパッチする．たとえばプロセッサ 0 からプロセッサ 1 へのリモートメモリ write の場合には，

(1) リモートのプロセッサ番号・データのアドレス，送信データのローカルのメモリアドレス，データのサイズ，メッセージのタイプを構造体にパックする．

(2) データサイズがある値より小さいときには，write するデータもその構造体にパックし，その構造体をプロセッサ 1 に送信．そうでない場合には，その構造体とデータ本体を別々に送信．

(3) プロセッサ 1 では受信したメッセージをメッセージハンドラがアンパックし処理する．ここではメッセージのタイプがリモート write であるので，リモート write の処理を行う．

(4) 受信した構造体から取り出したデータサイズがある値より小さいときにはデータもその構造体内にあるので，それを指定されたアドレスにコピーして完了．そうでない場合には，別にデータが送られてきているので，それを指定されたアドレスで受信して完了．

という動作になる．これは MPI の eager/rendezvous protocol 同様，データが小さいときには send/receive で動作して，大きいときには先に制御データのみを送ることにより，受信側のコピーによる性能の低下を回避している．

3.4 MTTL-MPI のオーバヘッド

メッセージパッシング型の並列環境である MPI 上にスレッドベースの並列言語である MPC++ を実装するため，MPI を挟むことによるオーバヘッドが避けられない．極力オーバヘッドの無いよう実装したが，リモートメモリ write 時でメッセージ識別子，リモートノード番号，リモートアドレス，データサイズなどからなる少なくとも 16bytes のヘッダがつけられる．同様にリモートメモリ read 時には少なくとも 24bytes のヘッダがつけられる．`(a)invoke` や Sync オブジェクトに対する操作に関しても同様である．データ転送量が少ないとき，つまりリモートメモリ read/write で小さいデータのやり取りをするときや，`(a)invoke` 時にこのオーバヘッドが dominant になってくると考えられる．

表 2 新情報処理開発機構 PCC-II の仕様

CPU	Pentium Pro 200MHz
Cache	512KB
Chipset	440FX
Memory	EDO 256MB
Myrinet Link Speed	160MB/sec
Myrinet Memory	1MB
OS	Linux 2.1.119

表 3 presto クラスタの仕様

CPU	Pentium II 350MHz
Cache	512KB
Chipset	440BX
Memory	SDRAM 128MB
NIC	Intel EthernetExpress Pro 10/100
OS	Linux 2.2.10

4. 性能評価

前章までに説明した MTTL-MPI の性能評価を行った．MTTL-MPI の通信レイヤの MPI として，SCore 上に実装された MPICH-PM[6]，SR2201 上の MPI (MPICH-sr2201)，100Base TCP/IP 上の MPICH (MPICH-p4) を用いた．またオリジナルの MPC++ on SCore でも評価を取った．以降ではそれぞれを単に，MTTL/MPICH-PM, MTTL/MPICH-sr2201, MTTL/MPICH-p4, MPC++/SCore と表記することにする．

4.1 評価環境

MTTL/MPICH-PM および，MPC++/SCore の測定には，新情報処理開発機構の PC Cluster II を用いた．各ノードの性能は**表 2** に示す通りである．コンパイラには egcs-1.0.2 release を用い，最適化オプションは -O6 である．

100Base TCP/IP の環境には我々の研究室の PC クラスタ（presto クラスタ）を用い，各ノードの性能は**表 3** に示す通りである．各ノードは 100Base Switch で**図 2** のように接続してある．使用した MPI は MPICH-1.1.2 でデバイスには ch_p4 を使用している．コンパイラには egcs-1.1.2 を用い，最適化オプションは -O6 である．

SR2201 の仕様を**表 4** に示す．使用した MPI は MPICH-1.0.11 を元に低レベルの通信に SR2201 の remote DMA を用いるように作り替えたものである．コンパイラには gcc-2.8.1 を用い，最適化オプション

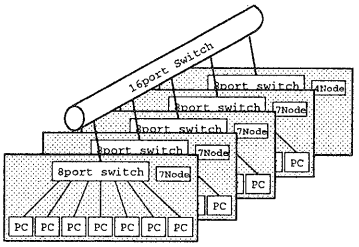

図 2 presto クラスタの Ethernet トポロジー

表 4 SR2201 の仕様

演算性能	0.3 GFLOPS/PE
L1 cache	16KB + 16KB
L2 cache	512KB + 512KB
Memory	256MB/PE
ネットワークトポロジー	3 次元クロスバー
PE 間転送速度	300MB/sec

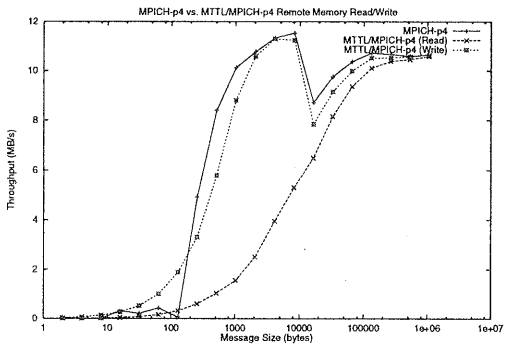

図 3 Throughput: 100Base-T

は -O2 である．ただし CG に関しては -O2 では実行時エラーが出てしまうため -O とした．

4.2 MTTL/MPICH-p4 の基本性能

MTTL-MPI を MPICH-p4 上で実行した際のスループットを，MTTL/MPICH-p4 のリモート read/write, MPICH-p4 の MPI_Send, MPI_Recv で測定した．結果を図 3 に示す．

MTTL/MPICH-p4 でライトオペレーションの方がリードオペレーションよりスループットが高い．これはリードオペレーションのあとには操作の終了の確認のために同期を取る必要があるためである．

MPICH-p4 において 256bytes より小さいデータサイズ時に性能がでていないのは，Linux 2.2.x の TCP/IP の実装において TCP_NODELAY オプションを指定しても状態によってユーザデータが TCP/IP パケットにまとめられる場合があるからだと思われる．

3.4 節で述べたようにリモート write 時のオーバヘッドはメッセージヘッダのサイズ，およびヘッダの処理にかかる時間であり，送信するデータサイズが小さいときには 30% 近くものオーバヘッドになってしまっているが，データサイズが大きくなれば無視できるほどのオーバヘッドしかないことが分かる．

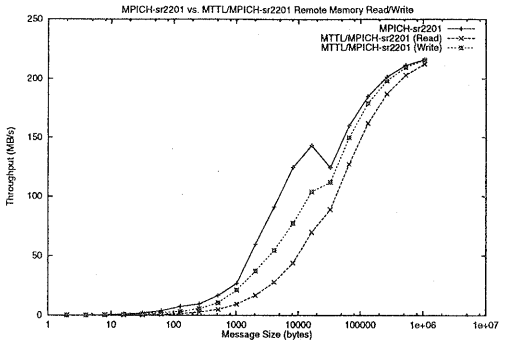

図 4 Throughput: SR2201

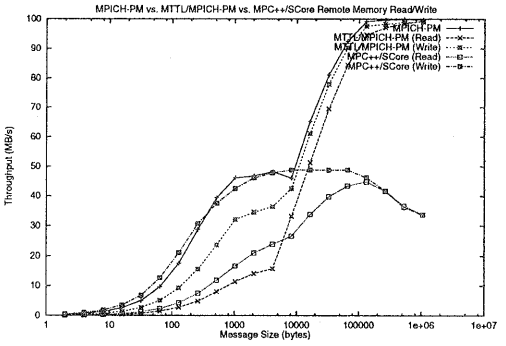

図 5 Throughput: Myrinet

4.3 MTTL/MPICH-sr2201 の基本性能

MTTL-MPI を MPICH-sr2201 上でのスループットを先と同様に測定した．結果を図 4 に示す．

MPICH-p4 における結果と同様の傾向を示している．

4.4 MTTL/MPICH-PM の基本性能

MTTL-MPI を MPICH-PM 上でのスループットを先と同様に測定した．結果を図 5 に示す．なお，ここでは比較のために MPC++/SCore でのリモート read/write の結果も含める．

MPICH-PM, MTTL/MPICH-PM の間にはやはり MPICH-p4, MPICH-sr2201 での測定結果と同様の傾向が見られる．

MTTL/MPICH-PM と MPC++/SCore を比べるとデータサイズが 8KB 以下の時には後者は前者に含まれる MPI を挟むことから来るオーバヘッド無しに実装されているためスループットが高い．

MPICH-PM と MPC++/SCore のリモート write を比べると，これらは同じレベルで実装されているため 8KB 以下の時にはほぼ同等の性能である．

MPICH-PM, MTTL/MPICH-PM のスループットが 8KB 付近から急激に上がっているのは，Myrinet の通信ドライバである PM[7] がデフォルトで 8KB 以上のデータの時にゼロコピー通信 [8] を行うためである．MPC++/SCore では現在は実装の都合上ゼロコ

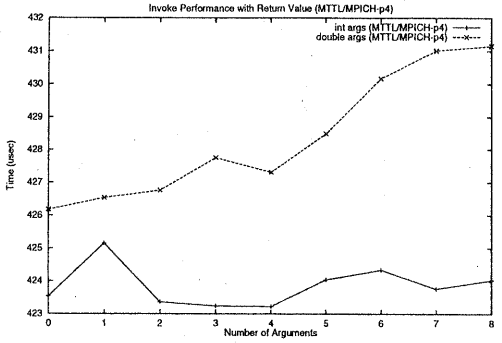

図 6 Remote Invocation (MTTL/MPICH-p4)

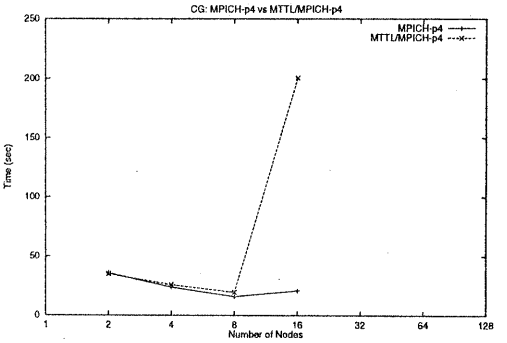

図 8 CG: 100Base-T

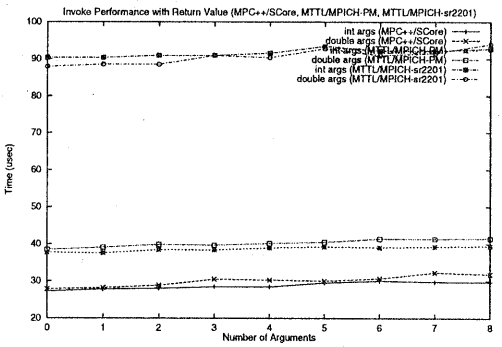

図 7 Remote Invocation (MPC++/SCore, MTTL/MPICH-PM, MTTL/MPICH-sr2201)

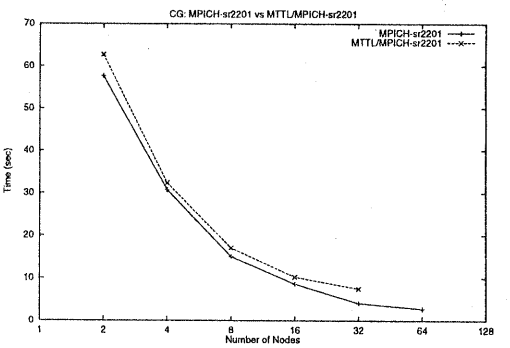

図 9 CG: SR2201

ピー通信を行っていないため，8KB を超えるデータ時には不利になっている．

4.5 リモートメソッド呼び出し

MTTL/MPICH-p4, MTTL/MPICH-sr2201 および MTTL/MPICH-PM においてリモートソッドの呼び出しにかかる時間を整数型，および実数型の引数を 0 から 8 個にわたって測定した．リモートで呼び出す関数は単にリターンするだけのものである．つまりここで測定しているのは，メソッドのアドレスおよび引数などをパックした構造体をリモートに送信し，メッセージハンドラがそれをアンパックし，関数のためのスレッドを作成・実行し，返り値を返すのにかかる時間である．**図 6**，**図 7** に結果を示す．リモートメソッドの呼び出しの際のデータの転送量は double の引数を 8 個送信し double の返り値を受信したとしても，double を 8bytes として 72bytes であり，そのほかに必要な制御用のデータを合わせても高々 100bytes 程度である．したがって先のリモートメモリ read/write の測定結果でデータサイズが小さいときの性能の違いが図 7 でも見られる．

図 6 で MTTL/MPICH-p4 の結果が大きく変動して見えるが，絶対的にかかった時間から見れば測定誤差の範囲である．

4.6 CG kernel benchmark

アプリケーションレベルのベンチマークとして Nas Parallel Benchmarks[9] より CG を用いた．本実験で用いた行列サイズは 14,000×14,000 (Class A) である．測定結果を図 **8**，図 **9**，図 **10** に示す．

MPICH-p4, MPICH-sr2201, MPICH-PM での CG は NPB2.3 として配布されているものを変更せずに用いたもので，コンパイラにはそれぞれ f77(egcs-1.1.2) -O6, f77(Hitachi 最適化 FORTRAN90) O(3), f77(egcs-1.0.2) -O6 を用いた．

図 8 で MTTL/MPICH-p4 の性能が 16 ノード時に突然落ちているのは，遅延の大きい `(a)invoke`（図 6 参照）が，プログラム中のリダクション演算を主として増加した（約 9000 回）ためと思われる．また，これに加えて図 2 から分かるようにアップリンクの帯域が 16 ノード以上になると極端に不足することも原因の 1 つと考えられる．8 ノードまでの結果を見ると MTTL/MPICH-p4 のオーバヘッドは 2–18% であり，十分なレベルにあると考える．

MTTL/MPICH-sr2201 のオーバヘッドは 16 ノードに至るまで小さく (4–15%)(図 9)，オーバヘッドは少ないと考えられる．また MPICH-sr2201, MTTL/MPICH-sr2201 ともにスケールしている．

図 10 でノード数が少ないときに MTTL/MPICH-PM の性能が高いのは，1 回の通信で転送するデー

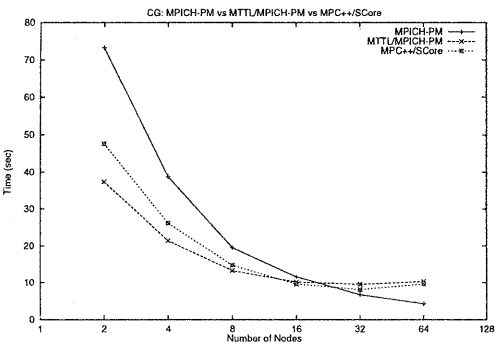

図 10 CG: Myrinet

タサイズが大きい（4 ノード時で 2.8MB など）ためゼロコピー通信が有効に働いているからだと考えられる．ノード数が増えたときにはデータサイズの小さい通信が増え MPC++/SCore の方が有利になる．MTTL/MPICH-PM, MPC++/SCore が 16 ノード以上でスケールしていない原因はリダクション演算にあると考えられる。MPICH-PM に関してはノード数が少ないときに性能がでていない理由を調べる必要がある．

5. 関連研究

1 章でも述べたが，MPC++ プログラムをポータブルにするには，1) MPC++ の実行環境である SCore を Myrinet 以外のネットワーク上に移植する，2) MPC++ の通信部分を変更してほかの汎用的な通信ライブラリ上に実装する，2 つの方法が考えられる．本研究では後者の解決法を示したが，前者の解決法を取る研究に SCore が利用している通信ライブラリ PM を UDP 上に移植した PM/UDP[10] および，Gigabit Ethernet 上に PM を実装した GigaE PM[11] などがある．

6. まとめと今後の課題

我々は MPC++ で書かれたプログラムのポータビリティを高めるため，MPC++ を汎用的な通信ライブラリである MPI 上に実装した．メッセージパッシング型の並列環境の上にスレッドベースの並列言語を実現するため，MPI を挟むことによるオーバヘッドが懸念された．また MPI の性能はプラットホーム毎に大きく違うため，性能の可搬性というのも興味深い点である．

測定の結果データ転送量の小さいときには 30% ほどのオーバヘッドがあるものの，データサイズが大きい (8KB 以上) ときには無視できるレベルであることが分かった．また，CG ではノード数が増加したときに MTTL/MPICH-p4 の性能が著しく落ちるのに対し，高性能なネットワークを利用する MTTL/MPICH-PM, MTTL/MPICH-sr2201 ではスケールした．この結果高性能なネットワーク上での MTTL-MPI の有効性は確認できたが，低速なネットワーク上では更なる最適化が必要であることが分かった．

今後の課題として，バリア・リダクションなどの集団通信の性能測定。今回測定に用いた CG を 1 プロセッサで測定してみること，CG 以外のアプリケーションで測定してみることなどがあげられる．

謝辞 NEC Cenju-4 上への移植に際して，深夜に及ぶまでサポートしていただいた NEC C&C メディア研究所の中田登志之氏に深く感謝致します．

参考文献

1) http://www.rwcp.or.jp/lab/pdslab/.
2) http://www.myri.com/.
3) Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. An Implementation of Parallel Operation System for Clustered Commodity Computers. In *Cluster Computing Conference '97*, March 1997.
4) Yutaka Ishikawa. Multi Thread Template Library – MPC++ Version 2.0 Level 0 Document –. Technical Report 012, Tsukuba Research Center, Real World Computing Partnership, September 1996.
5) Yutaka Ishikawa. The MPC++ Multi-Thread Template Library on MPI. Technical Report 005, Tsukuba Research Center, Real World Computing Partnership, October 1997.
6) Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *ACM SIGARCH ICS'98*, pp. 243–250, July 1998.
7) Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In Peter Sloot and Bob Hertzberger, editors, *High-Performance Computing and Networking '97*, Vol. 1225, pp. 708–717. Lecture Notes in Computer Science, April 1997.
8) Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *IPPS/SPDP 1998*, pp. 308–314. IEEE, April 1998.
9) http://science.nas.nasa.gov/Software/NPB/.
10) 曽田哲之, 手塚宏史, 住元真司, 堀敦司, 石川裕. 通信ライブラリ PM の UDP 上への移植と評価. In *HOKKE '99*, pp. 127 – 132. 情報処理学会, March 1999.
11) 住元真司, 手塚宏史, 堀敦史, 原田浩, 高橋俊行, 石川裕. Gigabit Ethernet を用いた高速通信ライブラリの設計と評価. 並列処理シンポジウム JSPP'99, pp. 63 – 70, 1999.