MPC++-on-MPI のコモディティクラスタ環境における評価

栄 純明[†] 石川 裕[††]
松岡 聡[†] 高橋俊行[††]

MPC++ のように finer-grained なマルチスレッド・リモートメソッド起動・グローバルメモリリード/ライトや同期構造体を言語レベルでサポートする並列言語は，C + MPI の様なものに比べて，表現力・記述性が高い．finer-grained な言語の構造上，こういった並列言語は従来高速で高価なネットワーク上の専用のユーザレベル通信ライブラリを用いて実装されてきた．一方で，これらの並列言語が広く受け入れられるためには，比較的安価なネットワークを使用したコモディティハードウェア上のポータブルなメッセージ通信ライブラリを用いて実装される必要がある．しかし，コモディティハードウェア上に実装された並列言語で (1) 既存の並列プログラムを容易に記述することができるか，(2) その場合，コモディティハードウェア/ソフトウェアを用いたことによる性能の低下はどれくらいになるのか，(3) 専用のハードウェア/ソフトウェアを用いて実装したものに比べてどれくらいのスケーラビリティを得ることができるのか，を系統的に検証した例はほとんどない．我々は，このようなコモディティな実装の有効性を検証するため，MPC++ を Myrinet と 100Base-T Ethernet という費用や性能の異なるネットワーク上の，異なる MPI の実装上に移植した．さらに，NASPAR のアプリケーションを MPC++ 上に移植し，ベンチマークを行った．その結果，(a) 既存の MPI で記述されたプログラムを MPC++ に移植するのは容易であること，(b) MPC++ の通信レイヤに MPI を使うことによるオーバーヘッドは小さく，NASPAR アプリケーションでは無視できる程度であること，(c) データセットが大きいときには 100Base-T 上の MPC++/MPI でも Myrinet 上の C+MPI やオリジナルの専用実装である MPC++ on PM/Myrinet と同等の性能を示すことが分かった．

Evaluation of MPC++-on-MPI on Commodity Cluster Environment

Yoshiaki Sakae,[†] Yutaka Ishikawa,[††] Satoshi Matsuoka[†]
and Toshiyuki Takahashi[††]

Parallel Programming Languages such as MPC++ which facilitates finer-grained multi-threading, remote method invocation, global memory read/write, and synchronized data structures at the language level, have often been claimed as being allowing his parallelism to be expressed in much richer, easier style than programming with libraries such as C + MPI. Due to reliance on language mechanisms which are finer-grained, such languages have traditionally been implemented only on specialized user-level libraries on top of fast, expensive networks. On the other hand, in order for such languages to gain common acceptance, they must be implemented on top of portable messaging libraries running on commodity hardware with substantially less expensive networking. However, little systematic studies have been done as to identify (1) whether the languages allow easy expression of traditional parallel programs, and (2) in such a case, how much performance one loses by using commodity software/hardware, and (3) the degree of scalability compared to dedicated software/hardware implementations. In order to verify the viability of commodity implementation, we ported the MPC++ language on top of different breeds of MPI, to be executed on two networks of substantial performance/cost difference, namely, Myrinet and 100Base-T Ethernet. We then investigated whether some NASPAR applications can be ported “naturally” on top of MPC++, to be benchmarked in such a environment. Results were quite positive for MPC++ and its commodity implementation, namely (a) the port was quite effortless, (b) the small penalty caused by the additional MPI layer was negligible for NASPAR applications, and (c) for large data sets, MPC++/MPI running on the 100Base-T network was surprisingly competitive to both the C+MPI on Myrinet, the original dedicated implementation of MPC++ on PM/Myrinet. The results are quite promising for wider-spread acceptance of higher-level parallel languages on commodity clustering environments.

1. はじめに

コモディティハードウェアから構成される Beowulf[17),19)] 型のクラスタ型並列計算機がユーザビリティーの高さ，費用対効果の大きさ，技術の進歩の恩恵を受けやすいと言った特徴から研究用のプラットホームとしてだけでなく，商用の環境としても普及することが期待されている．一方，クラスタ環境での開発言語を見ると，MPI のような比較的低レベルなメッセージ通信ライブラリは普及しているものの，スレッド・リモートメソッド・グローバルメモリなどのより高いレベルの abstraction をサポートした並列言語は多くない．

こういった並列言語はもともと，細粒度スレッドおよび高速メッセージパッシングを言語モデル上仮定しており，コモディティネットワーク (Ethernet など) 上の通信ライブラリ (MPI, PVM, TCP/IP など) では不十分であると，従来考えられてきた．たとえばこの種の並列言語には，Real World Computing Partnership[1)](RWCP) の MPC++[11)]，UCB の Split-C[13)]，UIUC の Charm++[12)] などがあるが，MPC++ は高速なネットワーク (Myrinet[2)]) と専用の通信ライブラリ (PM[21)]) を想定しており，Split-C はやはり専用の通信ライブラリである Active Messages[14)] を想定している．Charm++ は UDP/IP をサポートしているが評価に用いているアプリケーションが，もともと非常に並列性の高いものであったり，ネットワーク性能の低さの影響を受けづらいものであったり，通常の SPMD 型のアプリケーションでの評価が十分とは言えない．

また，これらの並列言語に関して (1) 従来の並列プログラムを容易に記述できるだけの十分な表現力を持っているか，(2) 実装にコモディティハードウェア/ソフトウェア，特にコモディティネットワーキングレイヤを用いたときに，どの程度の性能が期待できるのか，(3) 専用実装したものと比較してどの程度のスケーラビリティを得ることができるのか，を明らかにする系統的な研究はほとんど行われていない．これはコモディティクラスタ上に実装されるこれらの並列言語が，コモディティクラスタと同様に広く使用されるようになるのに必要なことである．

我々はファーストステップとして並列言語のコモディティな実装の有用性を確認した．まず，MPC++ を費用と性能の異なる Myrinet および 100Base-T Ethernet 上で動作する MPI 上に効率的かつ可搬性を有するよう移植した．つぎに，NAS Parallel Benchmark 2.3[6)](NPB-2.3) から CG と IS を「自然な形で」MPC++ 上に移植可能であるか否か検証し，それを用いてベンチマークを行った．さらに，MPI を通信レイヤとして用いたことによるオーバーヘッドを解析し，その性能をオリジナルの専用実装である MPC++ と比較した．その結果，(1) C+MPI で書かれたものを移植するのに，デバッグも含めて数時間しか要しないこと，(2) MPC++ on MPI に移植したアプリケーションが a) オリジナルの C+MPI で記述されたもの，b) 通信レイヤとしてユーザーレベル通信ライブラリである Myrinet 上の PM を用いた MPC++ 上で動かした場合，の両者に比較して同等にスケールすること，(3) 100Base-T Ethernet における MPC++ on MPI 上でも，特に問題サイズが大きいときにはオリジナルの MPC++ 上のものと同等のスケーラビリティを示すこと，を確認した．

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 (MTTL-MPI) 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．すなわち，スレッドの実行は同期を行うか，プログラマが明示的に yield するか，その実行が終了するまで止らない．

すべ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．したがってプログラム実行時の変数参照は，そのスレッドの実行されているプロセス内のアドレススペースでのアクセスであり，他のプロセスの変数にア

† 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

クセスするためには後述の *global pointer* の機能を用いる．

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，メッセージの到着を待つ．

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．`invoke`, `ainvoke` 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．

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 の先頭から削除される．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．またメンバ関数 peek() 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() 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

利用する下位の通信レイヤから独立にするため，MPI 程度の抽象化がなされた通信ライブラリを想定し，**表 1** に示す 8 個のラッパールーチンを用意し，これらを用いて通信を行うように MTTL を変更した．これらのプリミティブはほぼ 1 対 1 で同等の MPI 命令にマッピングされており，たとえば，`_pmttl_asend` は `MPI_Send` に，`_pmttl_nsend` は `MPI_Isend` に，`_pmttl_brecnode` と `_pmttl_brecany` は `MPI_Recv` にそれぞれマップされる．

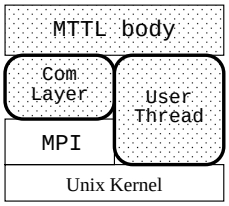


図 1 MTTL-MPI の構造
Fig. 1 Structure of MTTL-MPI

表 1 MTTL-MPI の通信プリミティブ
Table 1 The primitives of MTTL-MPI communication

`_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

なお，本稿では通信レイヤとして MPI を用いたものしか扱わないが，我々は通信レイヤとして VIA[9] を用いる試み[24] も行っており，下位の通信ライブラリの差異を本レイヤで吸収している．

3.2 ユーザスレッド

スレッドは setjmp, longjmp を用いたユーザレベルスレッドとして実装している．スレッドのスケジューリングは non-preemptive で，同期のために wait するか，スレッドが終了するか，明示的に yield するなどしない限りスレッド切り替えは起らない．

Thread クラスのオブジェクトが個々のスレッドを表現していて，そのスレッド上で実行する関数・通信に用いるバッファ・スタック・PC カウンタなどのコンテキストを Thread オブジェクトに保存する．また，各 Thread オブジェクトは queue で管理され，スレッド切り替えはこの queue の操作と longjmp によって実現される．

ユーザスレッドに関してはオリジナルの MPC++ から構造的に大きな変更はない．

3.3 実 行 例

remote invocation を例にとって MTTL-MPI の実行モデルを説明する．**図 2** は MTTL-MPI のシステム

```
int main(int argc, char **argv)
{
    _pmttl_thinit();
    _pmttl_cominit(&argc, &argv);
    if (myNode == 0) {
        mpc_main(argc, argv);
    } else {
        do {
            _mpcRecHandler(0, 0);
            if (Thread::numReadyThreads > 0) Thread::resched(0);
        } while (_mpc_exiting == 0);
    }
    mpc_exit(0);
    while (_mpc_exiting != numNode) {
        _mpcRecHandler(0, 0);
        if (Thread::numReadyThreads > 0) Thread::resched(0);
    }
    _pmttl_comfinish();
    exit(0);
}
```

図 2 システムコード
Fig. 2 System Code

```
#include <mpcxx.h>

int foo(int, int);

int mpc_main(int argc, char **argv)
{
    int i;

    invoke(i, 1, foo, 1, 2); // invoke foo on node 1
    return 0;
}
```

図 3 ユーザーコード
Fig. 3 User Code

のコードで，起動時に参加する全ノードで実行される．**図 3** がユーザーのメイン関数であり，`mpc_main` という名前を使う．

(1) スタートアップ後最初に，スレッドオブジェクトと通信レイヤの初期化が全ノード上で行われる．最初のスレッドオブジェクトに登録される関数はメッセージハンドラー `_mpcRecHandler` である．

(2) 次に node 0 上で `mpc_main` が呼び出され，それ以外のノードでは `_mpcRecHandler` が実行される．

(3) 続いて node 0 上で実行されている `mpc_main` から関数テンプレート invoke (invoke に関しては後述する) が呼び出される．関数テンプレート invoke を用いることによって，関数をリモートもしくはローカルで (必要であれば) 新たにスレッドを作成し，その上で実行することができ，呼び出したスレッドの実行は invoke から返るまでブロックする．この例では，node 1 で関数 `foo(1, 2)` を呼び出し帰り値を node 0 のローカル変数 `i` で受けている．

(4) node 0 以外では `_mpcRecHandler` で常にメッセージの到着を待つ．

(5) 最後に node 0 が `mpc_exit()` で終了メッセージを全ノードに送り，全ノードの終了を待ってから `_pmttl_comfinish()` で通信レイヤの後処理をし，プログラムの実行を終了する．

図 4 に invoke を行った際のプログラムの流れ図を示し，以下これに沿って説明する．

(1) node 0 で `invoke(foo)` が実行されると，メッセージタイプ・リモートノード番号・呼び出される関数のアドレスなどの制御情報が `com_buf` と呼ばれる構造体に格納される．

(2) `com_buf` をリモートノード (この例では node 1) に送信 (`_pmttl_asend()`)．ここで `com_buf` が一定のサイズより大きいときにはバッファコピーのオーバーヘッド，領域が大きくなるのを避けるため，制御情報とデータ本体に分割して送信 (`_pmttl_asend()` および `_pmttl_nsend()`) する．invoke を呼び出したスレッドは，関数 foo の返り値を Sync オブジェクトを通して読み出そうとするが，この次点ではまだ返り値が node 0 に届いていないためブロックする．これによりコンテキストスイッチが起こり，メッセージハンドラ `_mpcRecHandler` に制御が移る．

(3) node 1 では `_mpcRecHandler` で `com_buf` を受信し (`_pmttl_brecany()`)，制御情報を取り出す．ここで `com_buf` が一定のサイズより大きいときには制御情報を `_pmttl_brecany()` で，データ本体を `_pmttl_brecnode()` で受信する．

(4) 取り出した制御情報からメッセージタイプを得て，以下の処理を dispatch する．ここでは "remote invocation" であるため，(必要であれば) スレッドを作成しその上で関数 foo を実行する．

(5) foo の終了後返り値を node 0 に返す (`_pmttl_asend()`)．

(6) node 0 では先ほどアクティブになった `_mpcRecHandler` が node 1 からの foo の返り値を受取り，Sync オブジェクトに書き込む．

(7) ここで再びコンテキストスイッチが起り，invoke を行ったメインスレッドに制御が移り，foo の返り値を取り出して処理は先に進む．

3.4 MTTL-MPI のオーバーヘッドの要因

高レベルな通信ライブラリである MPI 上に MPC++ を実装するため，より下位の通信レイヤを直接用いるのに比べて，MPI を挟むことによるオーバヘッドが避けられない．無駄なバッファコピーの回避，

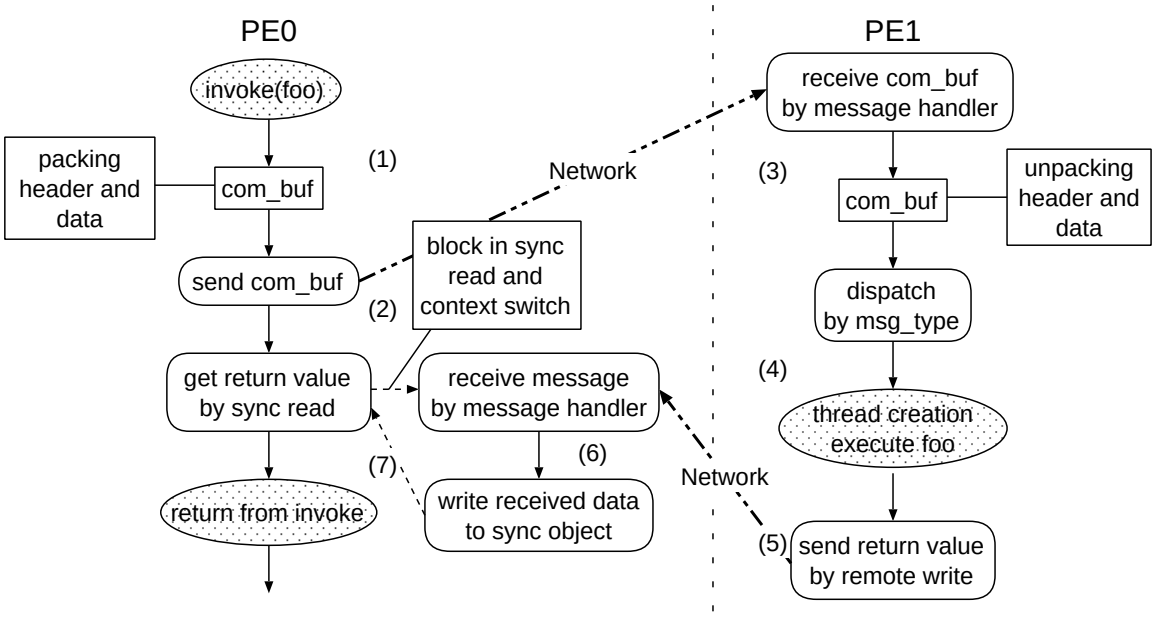


図 4 invoke の処理の詳細
Fig. 4 The datails of invoke

インライニングなど極力オーバヘッドの無いよう実装したが，たとえばリモートメモリ 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．`invoke`, `ainvoke` 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，`invoke`, `ainvoke` 時にこのオーバヘッドが dominant になってくると考えられる．

また Reduction, Barrier などの集団通信も MPI のものを用いずに MTTL のレベルで実装している．これは集団通信をサポートしていない通信レイヤ (e.g. VIA) に対しても可搬性を確保するためであるが，より下位のレベルで実装するのに比べ余分なメッセージ・スレッドの切り替えに起因するオーバーヘッドも考えられる．

4. 評価方法と環境

(1) MPC++ で従来の SPMD 型の並列プログラムを容易に記述できるか否か，(2) コモディティソフトウェア/ハードウェアを用いることによってどの程度の性能低下があるか，(3) 専用のソフトウェア/ハードウェアを用いた実装と比較してどの程度のスケーラビリティを得られるか，を確認するため我々はまず 3 節に示したように MPC++ をポータブルに実装した．つぎに，(1) を確認するため NPB-2.3 から CG, IS を MPC++ 上に移植した．さらに (2)(3) を確認するため，これらのアプリケーションを用いて以下に挙げる要素を変えてベンチマークを行った．

- プロセッサ数 (1–32),
- 問題サイズ (Class A, Class B),
- 通信レイヤ (MPI vs. PM),
- MPI の実装 (LAM[3),7),8)] vs. MPICH[4),10)]),
- 低レベルメッセージングレイヤ (TCP/IP vs. PM),
- ネットワークインターフェース (Switched 100Base-T Ethernet vs. Myrinet).

結果は 5 回計測したうちの最速のものを示す．ここでは，性能上の理由などから上記のすべての組み合わせに関して計測したわけではない．たとえば，100Base-T Ethernet 上では MPICH より LAM の方が予備的な性能測定，CG, IS のテストにおいて高い性能を示しているので 100Base-T では MPI の実装として LAM を基本に， CG の Class B に関しては MPC++/LAM

で不具合☆が見られたため Ethernet 上での PM の実装である PM-Ether[20] を用いた MPI の実装 MPICH-PM/PM-Ether を用いた．一方で Myrinet 上では PM を下位の通信プロトコルとして使用している LAM-PM は存在しないため，MPICH-PM[16] を MPI として用いている．

以降，MPI の実装として LAM を用いたものを MPC++/LAM，MPICH-PM/PM-Ether を用いたものを MPC++/MPICH-PM/PM-Ether，MPICH-PM を用いたものを MPC++/MPICH-PM，またオリジナルの PM を用いた MPC++ の実装を MPC++/native-PM と表記する．

実際の組合わせは 5 節，**表 3** を参照のこと．

4.1 MPC++ による NPB CG, IS の記述

NPB-2.3 のベンチマークは SPMD 型の並列プログラムとして書かれている．このうち IS が C+MPI で記述されている以外はすべて，Fortran+MPI で記述されており，それぞれ問題サイズの違いにより小さい方から Class A, B, C と問題セットが設定されている．

NPB-2.3 の仕様[6] に沿ってスクラッチから MPC++ 版を記述することも考えたが，以下の 3 つの理由により移植という手法を用いた．(1) スクラッチから記述することによってコードが大きく異なってしまうと，各実装における下位の通信レイヤの差異を正確に見るのが難しくなってしまう点，(2) 実装のコストが大きくなるであろう点，(3) それにも関わらずスクラッチから記述したとしても最終的なプログラムの構造には大きな違いは見られないであろう点．移植は C++ で行うという都合上 CG, IS に関して行った．CG に関しては電総研の田中氏による C+MPI+Thread による実装をベースとした．

以下に C+MPI で記述された CG, IS を MPC++ で記述する際に行う主な MPI 版からの変更点をあげる．

- メインスレッドの変更．SPMD で記述された MPI プログラムでは各ノードは等しくプログラムを実行し，自分のノード番号によって仕事を分担することによって全体のタスクをこなす．一方 MPC++ では SPMD 型のプログラミングモデルであるものの，メインのスレッドを実行するのは基本的に 0 番ノードだけで，ほかのノードは 0 番ノードから remote invocation によって仕事を割り当てられ全体のタスクをこなす．したがって MPC++ で記述する際には，初期化部分・カーネルループ部分などの単位でタスクを切出し，0 番ノードからほかのノードにそのタスクを remote invoke するように変更する．
- メッセージ送受信の変更．MPI のデータの送受信では，送信側・受信側双方で MPI_Send, MPI_Recv などの操作をプログラマが行う必要があるが，MPC++ ではリモートメモリリード/ライトを用いて一方のノードからの操作でデータの移動が行える．ただし，あらかじめリモートメモリのアドレスを把握しておく必要があり，注意を要する．
- そのほか，集団通信などの MPI 命令の単純な MPC++ への置き換え．

上記の方法に従うことで，C+MPI で記述された NPB プログラムを素直に MPC++ に書き直すことが可能であることを確認した．また，スクラッチから記述したとしても，アルゴリズムが同じならば同等の記述になると強く推測される．

4.1.1 Port of CG

CG は対称正定値行列の固有値の近似解を conjugate gradient(CG) 法を用いて求める問題で，NPB-2.3 では Fortran + MPI で記述されている．本研究では MPC++ 版と直接比較するために，電総研の田中氏による C+MPI+Thread による実装をベースに，一旦 C++ + MPI で書き直し，それを MPC++ に移植した．C++ + MPI 版からの主な変更は以下の 3 点である．カッコ内は変更したおおよその行数である．

- スタートアップ後カーネルループ部分を呼ぶ部分までの変更 (約 30 行)
- カーネルループ内の内積で必要なリダクション演算を MPC++ のリダクションへ変更 (約 10 行)
- ベクトルのアップデートに必要な配列の送受信を MPC++ のリモートメモリライトに変更 (約 15 行)

C++ + MPI 版 CG の MPC++ への移植は上記の手順で素直に行えて，時間にして正味 1 時間くらいである．必要なコード変更量は全部でおよそ 800 行中 60 行であった．

4.1.2 Port of IS

IS は整数の整列問題で，そのカーネル部分はノード内のヒストグラムの作成・全体のヒストグラムの集計・各階級の割り当てのプロセッサへのデータ移動・階級内でのソーティング・部分列の検証からなる．このうち通信が必要となるのはヒストグラムの集計と，移動デー

☆ ノード数を 16 ノード以上に増やしたときに極端に性能が落ちるもので，この際 CG の解は収束するものの残差が触れてしまうという現象が見られるもの．これは MPC++/LAM の Class B で 16 ノード以上に増やしたときのみに見られるもので，他の組合わせでは起らない．詳しい原因は現在調査中．

表 2 Presto クラスタの仕様
Table 2 The specification of Presto cluster

CPU	Pentium II 350MHz
Cache	512KB
Chipset	440BX
Memory	SDRAM 256MB
NIC	Intel EthernetExpress Pro 10/100
NIC	Myrinet M2M-PCI64A-2

タ数の交換，各階級の割り当てのプロセッサへのデータ移動の 3 個所であり，オリジナルの C + MPI 版では MPI_Reduce, MPI_Alltoall, MPI_Alltoallv を用いて実装されている．MPC++ に移植する際の変更点は以下の 6 点である．

- スタートアップ後カーネルループ部分を呼ぶ部分までの変更 (約 30 行)
- ヒストリ集計部分を MPC++ の Reduction で実現 (2 行)
- 移動データ数の交換を MPC++ のリモートメモリライトで記述 (6 行)
- データ移動の前準備を MPC++ のリモートメモリライトで記述 (5 行)
- データ移動を MPC++ のリモートメモリライトで記述 (6 行)
- チェックルーチン内の 4 バイトデータ転送．(3 行)

IS の場合は特にデータ構造が単純なので短時間 (約 1.5 時間) で移植を行えた．必要なコード変更量はおよそ 1000 行中 50 行程度であった．

4.2 環　　境

評価には我々の研究室の Presto クラスタ 32 ノードを用いた．各ノードの仕様は**表 2** のとおりで，Ethernet は Planex 社製 32 ポート Switching Hub (FHSW-3232NW) で，Myrinet は Myricom 社の M2M-OCT-SW8 × 2 台で接続されている．

OS は Linux 2.2.14，Myrinet のクラスタ環境に SCore 3.0，Ethernet 上の MPI には LAM 6.3.2 を使用した．

コンパイラには pgcc[5] を用い，コンパイルオプションは -O6 -mcpu=i686 -march=i686 -malign-double -fstrength-reduce -funroll-loops -fexpensive-optimizations とした．

基本性能として**図 5**，**図 6** に LAM, MPICH-PM では MPI_Send, MPI_Recv によるスループットを，MPC++/LAM, MPC++/MPICH-PM, MPC++/native-PM ではリモートメモリライトのスループットを示す．

MPI のスループットの計測には NetPIPE[18] のストリームモードを，MPC++ のリモートライトの計測には**図 7** のようなコードを用いた．

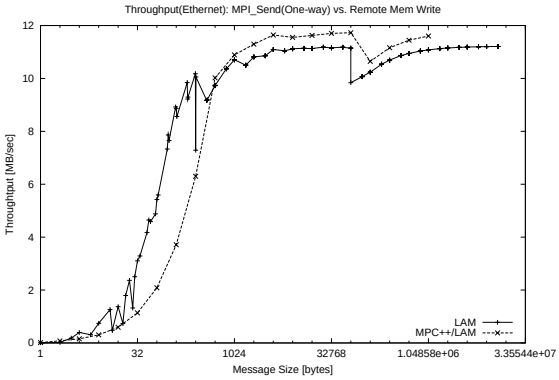


図 5 バンド幅 (Ethernet): LAM vs. MPC++/LAM
Fig. 5 Throughput(Ethernet): LAM vs. MPC++/LAM

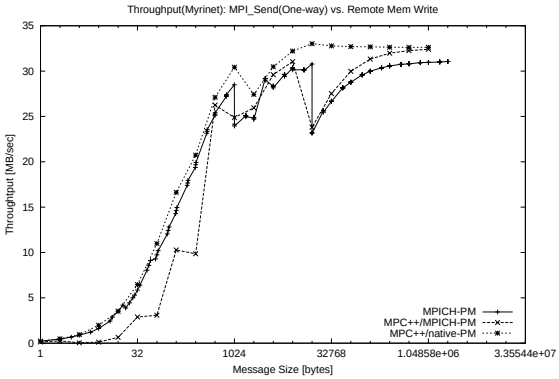


図 6 バンド幅 (Myrinet): MPICH-PM vs. MPC++/MPICH-PM vs MPC++/native-PM
Fig. 6 Throughput(Myrinet): MPICH-PM vs. MPC++/MPICH-PM vs MPC++/native-PM

```
for (int d_size = MIN; d_size <= MAX; d_size *= 2) {
  timer.start();
  if (myPE == 0) {
    for (int i = 0; i < ntimes; i++) {
      gp.nwrite(data, d_size);
    }
  }
  barrier.exec();
  timer.stop();
}
```

図 7 マイクロベンチマーク (MPC++ remote mem write)
Fig. 7 Micro Benchmark (MPC++ remote mem write)

5. 評 価 結 果

MTTL-MPI の通信レイヤである MPI に対するオーバーヘッド，Myrinet においては専用実装である

表 3 計測したデータ
Table 3 The combination of measured datas

	Ethernet	Myrinet
CG	Fortran/LAM	Fortran/MPICH-PM
	C++/LAM	C++/MPICH-PM
	MPC++/LAM	MPC++/MPICH-PM
		MPC++/native-PM
IS	C/LAM	C/MPICH-PM
	MPC++/LAM	MPC++/MPICH-PM
		MPC++/native-PM

MPC++/native-PM との比較，CG では Fortran から C++ に書き直した影響を見るため CG, IS の Class A, B それぞれに関して表 3 のような組合わせで評価を行った☆．ただし，Class B に関してはノード数 1 のときにはメモリなどの制限から計測できていない．

図 8，**図 9** が CG のノード数に対する経過時間のグラフである．CG ではノードの増加に伴ってノード当りの通信回数，通信量はともに増加するが，Myrinet 上ではいずれの組み合わせも 32 ノードまで妥当なレベルでスケールしている．MPC++/MPICH-PM と (MPC++ のオリジナルである) MPC++/native-PM を比較しても全く同等である．

Class A と Class B を比べると Class A では Ethernet, Myrinet ともに 32 ノードではノード当りのワーキングセットが小さく，通信時間が支配的になりつつあり，これ以上ノード数を増やしてもそれほどスケーラビリティは期待できない．一方 Class B では，CG は問題サイズ n の増加にともなって計算が $\mathcal{O}(n^2)$ のオーダーで増加するのに対して，通信は $\mathcal{O}(\log(n))$ でしか増えないため，32 ノードでも Class A ほど通信時間が支配的にはならず，なおスケーラビリティが期待できる．

また，Ethernet と Myrinet で Class A, B ともにスケーラビリティでは Myrinet が勝るものの，実際の実行時間ではそれほど大きな差は認められない．

図 10，**図 11** が IS のノード数に対する経過時間のグラフである．IS ではノード数の増加に伴ってノード当りの通信時間の割合が増加し，計算量に比べて通信量が支配的になる．Myrinet 上では Class A, B ともに 32 ノードまでスケーラビリティを維持している．MPC++/MPICH-PM と MPC++/native-PM の性能も変わらない，もしくは MPC++/MPICH-PM の方が高い性能を示している場合もあるほどである．Ethernet 上では Class B こそ 32 ノードまでスケールしているものの，Class A では 32 ノードではワーキングセットが通信量の増加に対して小さくなり過ぎていて，通信が支配的になり Myrinet に比べてスケールしていない．

☆ CG (Class B: Ethernet) に関しては 4 節で触れた通り MPC++/LAM のかわりに MPC++/MPICH-PM/PM-Ether を用いた．

表 4 CG (Class B) 32 ノード
Table 4 CG (Class B) 32 nodes

C++/LAM	257 [sec]
MPC++/MPICH-PM/PM-Ether	265 [sec]
C++/MPICH-PM	175 [sec]
MPC++/MPICH-PM	194 [sec]
MPC++/native-PM	210 [sec]

IS の Class A のノード数が少ないときに，MTTL-MPI 上での実装と MPI 上での実装にやや差が見られるものの，そのほかに関しては MTTL-MPI 版は MPI にほぼ等しい値を示している．

6. 考　　察

この評価で確認すべき点は 1)MPC++ を MPI 上に実装することによるオーバーヘッドと，2)100Base Ethernet を用いることのオーバーヘッドの 2 点である．

1) に関しては図 5, 図 6 からメッセージサイズが 512Byte 以下の時に最大で 50%のオーバーヘッドであることが分かった．このオーバーヘッドの要因は 3.4 節で示したものである．しかしこれはマイクロベンチマークの結果であり，現実的なアプリケーションではそれほど問題にならないことが，CG, IS のベンチマークで確認できた．

2) に関して，典型的な SPMD 型のプログラムを MPC++ で「自然な」記述をすると，100Base Ethernet 上の MPI ではレイテンシ・スループットなどが原因でスケールしないかも知れないという懸念があったが，Class B だと CG 32 ノードで**表 4** のようにそれほど大きな差はついていない．

したがって，実際的なデータセットであれば MPC++ のような言語で記述し，コモディティクラスタを用いても 32 台程度ならばそれほど性能的に差はつかなく，オーバーヘッドは小さいと言える．

一方 Class A で見ると，ノード数が増えたときの相対的な差が大きくなっている．これは極端にデータセットが小さいためであり，実アプリケーションという観点からは非現実的だが，より大きくノード数をスケールしたときの試金石として詳細に分析する．

6.1 CG (Class A: Ethernet)

C++/LAM 版 CG の通信の内訳を**表 5** に，

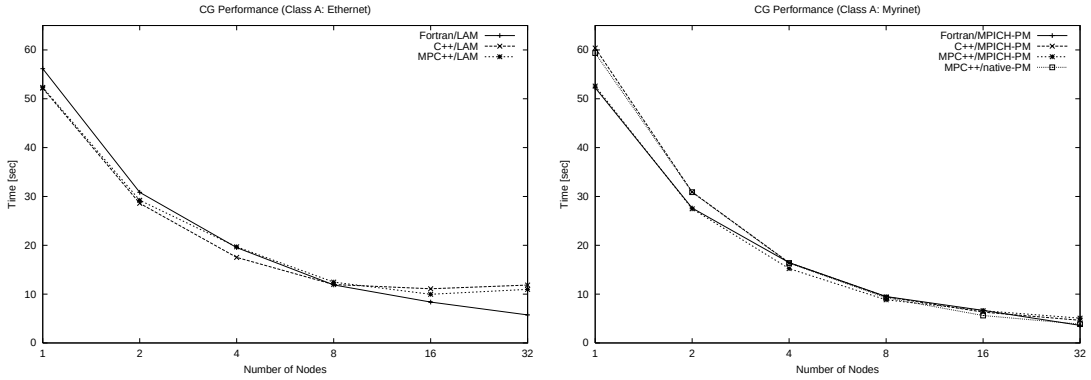


図 8 CG Performance (Class A)
Fig. 8 CG Performance (Class A)

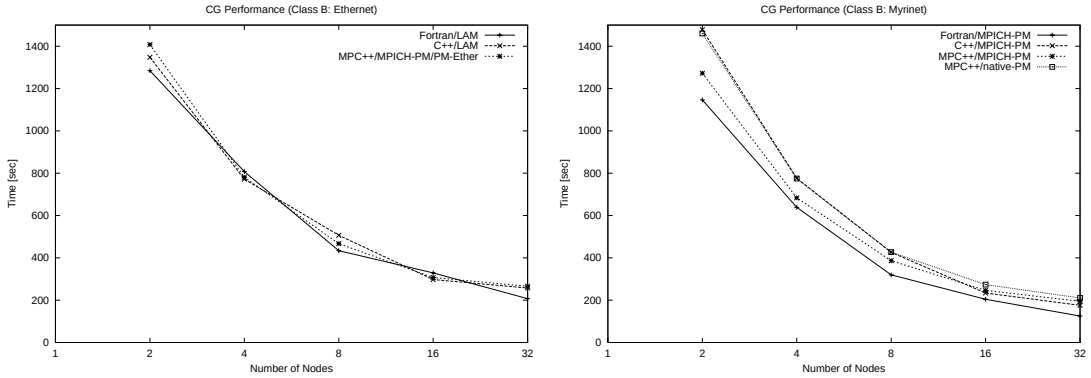


図 9 CG Performance (Class B)
Fig. 9 CG Performance (Class B)

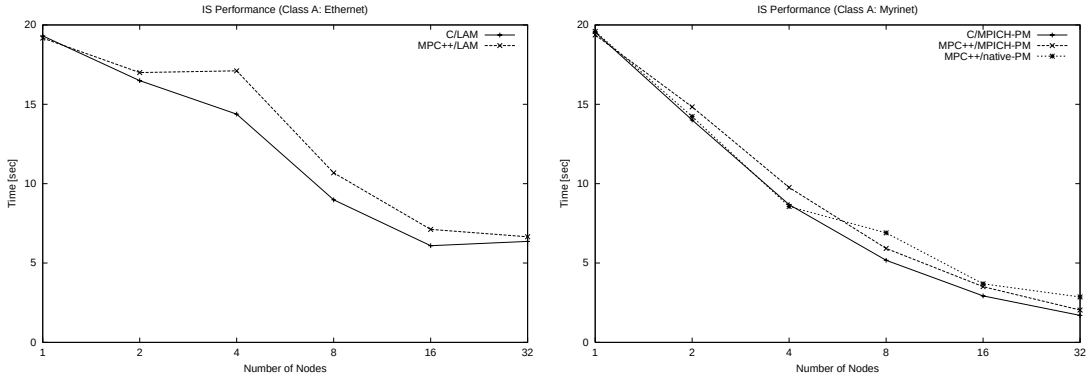


図 10 IS Performance (Class A)
Fig. 10 IS Performance (Class A)

MPC++/LAM 版のものを**表 6**☆に示す．通信回数はノードあたりの平均通信回数で，集団通信に関してはノード全体で 1 回と数える．“AVG” は 1 メッセージの平均メッセージ長，“Total” はノード当りの総メッセージ量であり，送信時のもののみカウントしている．

☆ MTTL-MPI の通信プリミティブとの対応は 3.1 節を参照のこと．

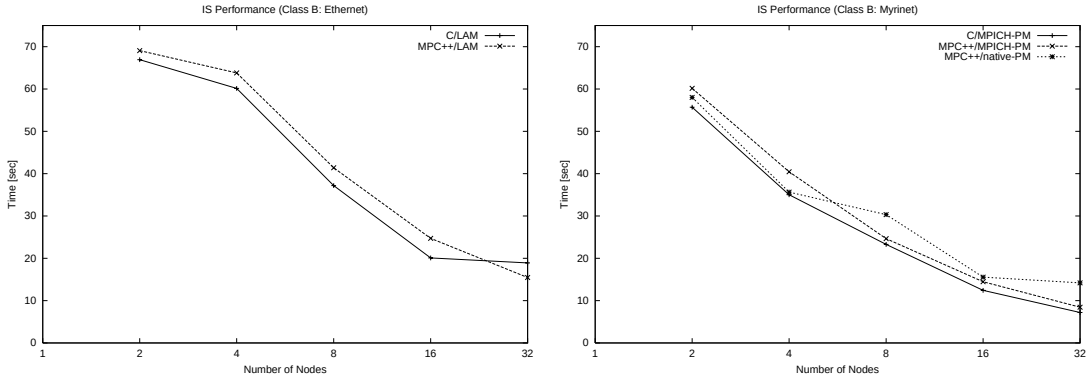


図 11 IS Performance (Class B)
Fig. 11 IS Performance (Class B)

表 5 CG (Class A: C++/LAM) の通信の内訳 (1 ノード当り)
Table 5 CG (Class A: C++/LAM): The breakdown of communication (per node)

Nodes	2	4	8	16	32
Send	1,200	2,790	5,160	9,090	16,140
Irecv	1,200	2,790	5,160	9,090	16,140
Wait	1,200	2,790	5,160	9,090	16,140
AVG(KB)	18.2	11.7	7.4	4.5	2.6
Total(MB)	20.8	31.3	36.5	39.1	40.4

また，それぞれの版の CG の計算時間と通信時間の内訳を**図 12** に示す.

C++/LAM 版 CG (表 5) と，MPC++/LAM 版 CG (表 6) を比較すると，前者に比べて後者は平均のメッセージサイズが小さくなり，転送回数の増加が見られる．また，MTTL の制御メッセージの転送も増加するため全体のメッセージ量としても若干の増加が見られる．しかし，図 5 によるとこの程度の大きさのメッセージサイズがあれば MPC++/LAM 版 CG でそれほどオーバーヘッドはなく，図 12 にあるようにほぼ同じ傾向を示す.

16 ノード以上の時に Isend, Test が 0 なのは MTTL-MPI の実装でメッセージサイズによって通信を切り替えているためである.

CG(Class A: Ethernet) の場合 32 ノードで通信時間が支配的になりつつある状況であり，C++/LAM 版 MPC++/LAM 版ともにここがスケーラビリティの限界だと言うことが分かる.

6.2 IS (Class A: Ethernet)

C/LAM 版 IS の通信の内訳を**表 7** に，MPC++/LAM 版のものを**表 8** に示す.

また，それぞれの版の IS の計算時間と通信時間の内訳を**図 13** に示す.

表 6 CG (Class A: MPC++/LAM) の通信の内訳 (1 ノード当り)
Table 6 CG (Class A: MPC++/LAM): The breakdown of communicationn (per node)

Nodes	2	4	8	16	32
Send	1,591	3,572	6,333	10,654	18,095
Isend	390	1,170	2,730	0	0
Recv	1,981	4,742	9,063	10,654	18,095
Iprobe	1,598	3,575	6,334	10,654	18095
Test	390	1170	2,730	0	0
AVG(KB)	10.80	6.77	4.14	3.87	2.37
Total(MB)	20.9	31.4	36.6	39.4	40.8

図 13 から C/LAM 版，MPC++/LAM 版ともにノード数が 2 を越えるとすでに通信が支配的になっていることが分かる．IS(Class A: Ethernet) の場合 16 ノードがスケーラビリティの限界であると言える．CG に比べて MTTL-MPI のオーバヘッドが確認できるレベルではあるが，絶対的な実行時間を考えると許容できないものではない.

また，4 ノード以下の時に C/LAM と MPC++/LAM で通信時間にやや異なる傾向が見られるが，これは LAM の集団通信は 4 ノード以下の時には linear に行われ，4 ノードを越えるときにはツリーを組んで行われるのに対し，MTTL-MPI は常にツリーを組んで行っていることに起因すると考えている.

7. まとめと今後の課題

本研究では通信レイヤに高速なメッセージ通信ライブラリを想定した abstraction の高い並列言語を，昨今のコモディティネットワーク・ソフトウェアの上に妥当な性能・スケーラビリティを維持したまま実現できないか確認するため，MPC++ を汎用の通信ライブラリである MPI 上に実装・評価した.

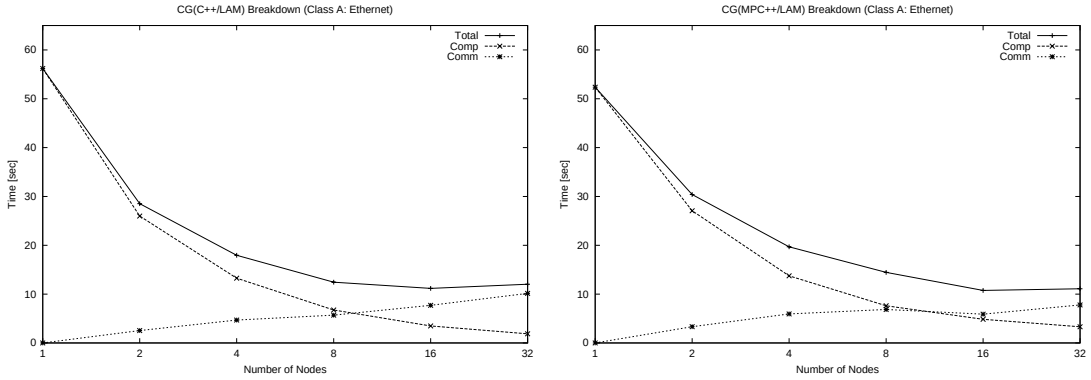


図 12 CG Breakdown (Class A)
Fig. 12 CG Breakdown (Class A)

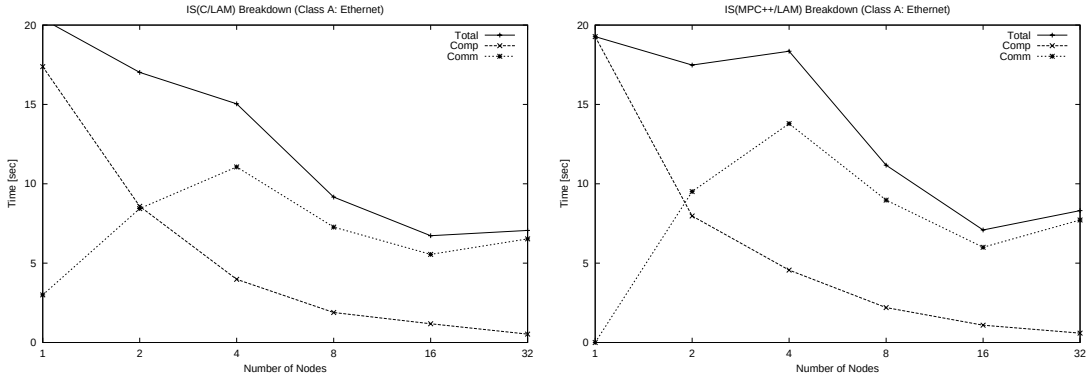


図 13 IS Breakdown (Class A)
Fig. 13 IS Breakdown (Class A)

表 7 IS (Class A: C/LAM) の通信の内訳 (1 ノード当り)
Table 7 IS (Class A: C/LAM): The breakdown of communication (per node)

Nodes	2	4	8	16	32
Allreduce	10	10	10	10	10
Alltoall	10	10	10	10	10
Alltoallv	10	10	10	10	10
AVG(KB)	2,731	2,052	1,204	660.2	372.3
Total(MB)	80.0	60.1	35.3	19.3	10.9

評価では現実的なアプリケーションにおける MPC++ の性能を示すため，NPB-2.3 の CG, IS を MPC++ に移植し，これを用いた．MPC++ に移植するのに MPI 版からの必要なコードの変更量は CG でおよそ 800 行中 60 行，IS でおよそ 1000 行中 50 行程度とわずかであることが確認できた．また，これによって MPC++ のデータ並列なプログラムに対する記述性も合わせて確認できた．

評価の結果 IS (Class A: Ethernet) においては通信が支配的になりすぎ MTTL-MPI のオーバーヘッドが確認されたが，許容できないレベルではない．それ以外の場合には MTTL-MPI の通信レイヤとして用いた MPI に特に性能的に劣るようなことはなく，また同程度のスケーラビリティを示した．さらに専用実装である MPC++/native-PM と比較しても Myrinet 上 (MPC++/MPICH-PM) はもちろん，Ethernet 上 (MPC++/LAM) でも近い性能を達成できた．

表 8 IS (Class A: MPC++/LAM) の通信の内訳 (1 ノード当り)
Table 8 IS (Class A: MPC++/LAM): The breakdown of communication (per node)

Nodes	2	4	8	16	32
Send	80	190	360	650	1,170
Isend	10	30	70	150	310
Recv	90	220	430	800	1,490
Iprobe	90,297	164,323	100,731	52,322	1,180
Test	90,267	164,193	100,511	51,972	310
AVG(KB)	910.5	279.7	83.7	24.2	6.84
Total(MB)	80.0	60.1	35.1	18.9	9.9

したがって，MPC++ のような，言語モデル上では細粒度スレッドおよび高速なメッセージ通信を仮定している言語でも，コモディティ通信ライブラリ上での実装を効率よく行うことができ，CG, IS などの SPMD 型の代表的なアプリケーションに関して性能面でのデメリットはほとんどない，と結論づけられる．

今後の課題としては我々の研究室の 64 ノード 128 プロセッサの次期クラスタ上での大規模なテストを行うこと．また，本稿では一般的なアプリケーションが MTTL-MPI でいかに効率よく動作するかということに主眼をおいていたため，MPC++ 専用に書かれたアプリケーションを用いるような言語中心のベンチマークでなく，MPC++ を実用言語として捉えたときに，ある程度ノード数の増加に対してスケールすることが一般的に知られているプログラムをベンチマークとして用いて，その言語が現実的な有効性を持つのかを議論したが，今後は MPC++ 専用に書かれたアプリケーションを用いて評価し，下位のネットワーク性能の差による影響も調べる予定である．

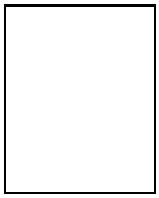
謝辞 本稿のベンチマークに用いた CG の C + MPI による実装を提供していただいた、電子技術総合研究所の田中良夫博士に感謝致します。

参 考 文 献

1) http://www.rwcp.or.jp/lab/pdslab/.
2) http://www.myri.com/.
3) http://www.mpi.nd.edu/lam/.
4) http://www-unix.mcs.anl.gov/mpi/mpich/.
5) http://www.goof.com/pcg/.
6) Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V. and Weeratunga, S.: THE NAS PARALLEL BENCHMARKS, Technical Report 007, RNR (1994).
7) Burns, G. D.: The Local Area Multicomputer, *Fourth Conference on Hypercube Concurrent Computers and Applications*, ACM Press (1989).
8) Burns, G. D., Daoud, R. B. and Vaigl, J. R.: LAM: An Open Cluster Environment for MPI, *Supercomputing Symposium '94* (1994).
9) Corp., C. C., Corporation, I. and Corporation, M.: *Virtual Interface Architecture Specification*, draft revision 1.0 edition (1997).
10) Gropp, W., Lusk, E., Doss, N. and Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Vol. 22, No. 6, pp. 789–828 (1996).
11) Ishikawa, Y.: Multi Thread Template Library – MPC++ Version 2.0 Level 0 Document –, Technical Report 012, Tsukuba Research Center, Real World Computing Partnership (1996).
12) Kalé, L. V. and Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++, *Proceedings of OOPSLA'93* (Paepcke, A.(ed.)), ACM Press, pp. 91–108 (1993).
13) Luna, S. S.: Implementing an Efficient Portable Global Memory Layer on Distributed Memory Multiprocessors, Technical report, UCB (1994).
14) Mainwaring, A. M. and Culler, D. E.: Active Message Application Programming Interface and Communication Subsystem Organization, Technical report, Computer Science Division, UCB (1995).
15) Matsuoka, S., Nikami, N., Ogawa, H. and Ishikawa, Y.: Towards a Parallel C++ Programming Language Based on Commodity Object-Oriented Technologies, *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)* (1997).
16) O'Carroll, F., Tezuka, H., Hori, A. and Ishikawa, Y.: The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network, *ACM SIGARCH ICS'98*, pp. 243–250 (1998).
17) Ridge, D., Becker, D., Merkey, P. and Sterling, T.: Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs, *IEEE Aerospace* (1997).
18) Snell, Q. O., Mikler, A. R. and Gustafson, J. L.: NetPIPE: A Network Protocol Independent Performance Evaluator, *IASTED International Conference on Intelligent Information Management and Systems* (1996).
19) Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawak, U. A. and Packer, C. V.: BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION, *International Conference on Parallel Processing* (1995).
20) Sumimoto, S., Tezuka, H., Hori, A., Harada, H., Takahashi, T. and Ishikawa, Y.: High Performance Communication using a Commodity Network for Cluster System, *High Performance Distributed Computing (HPDC) 2000* (2000).
21) Tezuka, H., Hori, A., Ishikawa, Y. and Sato, M.: PM: An Operating System Coordinated High Performance Communication Library, *High-Performance Computing and Network-*

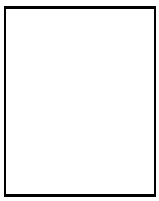
ing '97 (Sloot, P. and Hertzberger, B.(eds.)), Vol. 1225, Lecture Notes in Computer Science, pp. 708–717 (1997).
22) Wong, F. C., Martin, R. P., Arpaci-Dusseau, R. H. and Culler, D. E.: Architectural Requirements and Scalability of the NAS Parallel Benchmarks, *Proceedings of Supercomputing '99* (1999).
23) 板倉憲一, 松原正純, 朴泰祐, 中村宏, 中澤喜三郎: 超並列計算機 CP-PACS における NPB Kernel CG の評価, 情報処理学会論文誌, Vol. 39, No. 6, pp. 1757–1765 (1998).
24) 野田裕介, 栄純明, 松岡聡, 小川宏高: MPC++ Multi-Thread Template Library の様々な通信レイ ヤ上での実装と性能評価, 情報処理学会 研究報告 2000-HPC-82 (SWoPP 松山 2000), pp. 137 – 142 (2000).

(平成？年？月？日受付)
(平成？年？月？日採録)



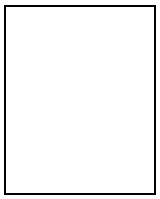
栄　純明

昭和 51 年生．平成 11 年東京工業大学理学部情報科学科卒業．現在，同大学大学院情報理工学研究科数理・計算科学専攻修士課程在学中．並列分散システム，並列プログラミング言語などに興味を持つ．ACM 学生会員．



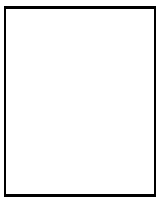
石川　裕（正会員）

1987 年慶応義塾大学大学院理工学研究科電気工学専攻博士課程終了．同年電子技術総合研究所入所．1988～1989 年カーネギー・メロン大学客員研究員．1990 年日本ソフトウェア科学会高橋奨励賞を受賞．1993 年から新情報処理開発機構に出向．並列・分散システム，適応可能並列プログラミング言語/環境/処理系，リアルタイム処理等に興味を持つ．日本ソフトウェア科学会，ACM，IEEE 各会員．工学博士．



松岡　聡（正会員）

昭和 38 年生．昭和 61 年東京大学理学部情報科学科卒業．平成元年同大学大学院博士課程中退．同大学情報科学科助手，情報工学専攻講師を経て，平成 8 年より東京工業大学情報理工学研究科数理・計算科学専攻助教授．理学博士．オブジェクト指向言語，並列システム，リフレクティブ言語，制約言語，ユーザ・インタフェースソフトウェアなどの研究に従事．現在進行中の代表的プロジェクトは，世界規模の高性能計算環境を構築する Ninf プロジェクト，計算環境に適合・最適化を目指す Java 言語の開放型 Just-In-Time コンパイラ OpenJIT，制約ベースの TRIP ユーザ・インタフェースなど．並列自己反映型オブジェクト指向言語 ABCL/R2 の研究で 1996 年度情報処理学会論文賞受賞．1997 年はオブジェクト指向の国際学会 ECOOP'97 のプログラム委員長を努める．ソフトウェア科学会，ACM，IEEE-CS 各会員．



高橋　俊行（正会員）

1993 年東京理科大学理工学部情報科学科卒業，1995 年同大学院修士課程終了，1995～1998 年東京大学理学系研究科情報科学科博士課程，1998 年より新情報処理開発機構研究員．現在に至る．プログラミング言語におけるメタレベルアーキテクチャと並列計算ソフトウェア技術に興味を持つ．理学修士．