部分計算を用いたMPIプログラム最適化システムOMPI

小川宏高† 松岡聡†

MPI（Message Passing Interface）は並列計算機を用いた高性能計算のためのメッセージ通信ライブラリの標準として広く普及しつつある．MPIは様々な通信方式を強力かつ柔軟に支援している反面，そのAPIの複雑さゆえに大きなソフトウェアオーバヘッドがかかり，MPIの適用範囲を定型的な粗粒度計算に限定する結果となっている．我々のOMPIはプログラム中のMPIライブラリ呼び出しにかかわる静的情報を用いて部分計算を行うことで余分なオーバヘッドの大部分を削減するシステムである．また，本システムでは部分計算のみでは不可能なさらなる最適化のために「テンプレート関数」を導入して利用する．ベンチマークを行った結果，わずかな実行コード量増加で通信主体の並列プログラムの性能が2倍に向上するとともに，従来の動的な最適化手法との比較でも大きな性能向上が見られた．

OMPI: A Compile-time Optimizer for MPI Programs

HIROTAKA OGAWA† and SATOSHI MATSUOKA†

MPI is gaining widespread acceptance as a standard for message passing in high-performance computing, due to its powerful and flexible support of various communication styles. However, the complexity of its API poses significant software overhead, and as a result, applicability of MPI has been restricted to rather regular, coarse-grained computations. Our OMPI (Optimizing MPI) system removes much of the excess overhead by employing partial evaluation techniques, which exploit static information of MPI calls. Because partial evaluation alone is insufficient, we also utilize *template functions* for further optimization. Benchmarks show that OMPI improves execution efficiency by as much as factor of two for communication-intensive application core with minimal code increase. It also performs significantly better than previous dynamic optimization technique.

1. はじめに

MPPやWSクラスタをはじめとする並列計算機の増加とともにPVM，MPI[5]に代表される標準化されたメッセージ通信ライブラリが普及しつつある．これらは新規に並列プログラムを作成するためだけではなく，既存のアプリケーションの並列化ツールやHPFのような並列言語実装用の実行時ライブラリとしても利用されている．特に様々な通信方式—コミュニケータを介した複数の通信コンテクスト，様々な同期・非同期通信モード，派生データ型，集団通信機能等—を強力かつ柔軟に支援したMPIは，公開から3年を経過した現在，名実ともに標準のメッセージ通信仕様として定着してきている．

MPIの実装は多数なされているが，APIの仕様から本質的に生じるソフトウェアオーバヘッドが大きいため，P4，PVM等の既存の通信ライブラリ実装に比べても効率が悪い．高速ネットワークを採用したハードウェアレイテンシの小さいシステムにおいては，通信速度の恩恵はソフトウェアオーバヘッドによって相殺されてしまうため，問題はいっそう深刻である．この問題はMPPのみならずMyrinetのような安価・高速なギガビットネットワークを用いたクラスタシステムにも該当する．

このため，MPIの利用は定型的で粗粒度な計算主体の応用に限定され，細粒度で非定型的な問題を効率良く実行するのは困難であり，メッセージ通信ライブラリの標準化によって本来期待される対応プラットフォームおよび応用領域の拡大の大きな障害となりうる．

一方，メッセージ通信のソフトウェアオーバヘッドを削減する研究は従来から行われている[9]．Active Messages（以降AM）の場合では数μsec程度のオーバヘッドにおさえているが，問題はMPIが有するような強力な機能を欠いており，可搬性も幾分損なわれる点である．つまり，AMでは低レベルのプリミティブのみ

† 東京工業大学
Tokyo Institute of Technology

を用意しているためプログラミングがMPIに比べて困難であり，異機種混合環境やマルチスレッド環境にも対応できない．

では，MPIの柔軟かつ強力な機能と低オーバヘッド通信という相反する要求をできうる限り同時に満たすにはどうすればよいか？ 1つの解は本稿で述べるOMPI（Optimizing MPI）システムである．OMPIはMPIを用いて書かれたプログラムに対して部分計算技術を適用してソフトウェアオーバヘッドを削減することでAMに迫る性能を実現する．本システムでは，MPIライブラリ呼び出しを含んだCプログラム（以降MPIプログラム）に対して静的解析を行い，MPIライブラリ呼び出しの引数で静的に定まるものを検出して，それらの引数に特化された関数を生成する．さらにオーバヘッド削減には現状の部分計算のみでは不十分であるので，前もって（アーキテクチャ等を考慮して）最適化しておいた「テンプレート関数」を状況に応じて選択する技術と組み合わせている．この結果，本システムはMPIプログラムの一般性や可搬性を保証しつつ，コンパイル時に実行時アーキテクチャに特化された最適化を実現できる．また，システム自体もアーキテクチャに特化されるテンプレート関数部分を除いて可搬に設計されており，既存の高性能MPI実装がアーキテクチャに特化されていたのとは対照的である．

本システムの有効性を確認するために，64PEの富士通AP1000上でPing-pongによる基本ベンチマークと通信パターンの異なる複数の数値アプリケーションコアによるベンチマークを実施した．1対1通信レイテンシは338 μsecから76 μsecに短縮するとともに，通信主体の数値計算コアでは2倍の速度向上を達成した．また，引数キャッシングによる従来の実行時最適化との比較でもOMPIの方が軒並高速となった．

2. ソフトウェアオーバヘッド削減に関する検討

まず，ソフトウェアオーバヘッドが生じる原因をメッセージ通信一般の問題とMPI固有の問題に分けて検討する．その後，静的情報が利用できる場合にそれらのオーバヘッドを削減する機会について述べる．

2.1 メッセージ通信一般のオーバヘッド

多くのメッセージ通信ライブラリではメッセージに関する情報は実行時にしか得られないため，基本的に受信バッファは何らかの方法で動的に確保し，受信したメッセージはそのバッファに複写しなければならない．実行時の工夫によってメッセージ到着以前に受信

```
MPI_Send
 (buf, count, type, dest, tag, comm)
  void *buf;            /* Send Buffer */
  int count;            /* Data Count */
  MPI_Datatype type;    /* Data Type */
  int dest;             /* Rank
                           (Target Process) */
  int tag;              /* Message Tag */
  MPI_Comm comm;        /* Communicator */

MPI_Recv
 (buf, count, type, source, tag, comm, status)
  void *buf;            /* Receive Buffer */
  int count;            /* Data Count */
  MPI_Datatype type;    /* Data Type */
  int source;           /* Rank
                           (Target Process) */
  int tag;              /* Message Tag */
  MPI_Comm comm;        /* Communicator */
  MPI_Status *status;   /* Status */
```

図1 MPIのAPIの例
Fig. 1 Examples of API of the MPI.

が発行されたときに限り，バッファ複写を省略できるが，先行送信，計算，受信の順に実行するレイテンシの隠蔽を意図したプログラムを記述した場合，この種の工夫による恩恵は得られない．

一方，複写オーバヘッドを完全に削減する手法としてはAMのように受信バッファアドレスを送信側が指定するものが一般的だが，柔軟性が損なわれる．たとえば，多くのメッセージ通信ライブラリが実現しているタグによるメッセージのフィルタリングは不可能であり，またすべての計算ノードでメモリ配置が同じという仮定が必要になるため，SPMDスタイルのプログラムに限定されてしまう．

2.2 MPIに固有のオーバヘッド

MPIの設計方針の1つは様々な並列アプリケーションに適用できるような，豊富な機能を提供することである．これを反映してMPIのAPIは多数のパラメータを引数として与え，その値がすべて実行時にしか分からないという特徴がある．たとえば，最も単純な1対1通信でも6，7個の引数を取る（図1）．MPIの様々な機能を実現できるのはAPIの複雑さの恩恵であるが，同時に実行時に範囲検査や作業領域の動的確保やエラー検査処理等が必要となり，オーバヘッド増大の原因になる．

ここでメッセージ通信時に必要な4個組のパラメータ（以降，通信セット）を定義する．通信セットはPVMのような単純なAPIのライブラリではほぼ直接的に得られるものであるが，MPIでは下に示すような処理が必要になり，オーバヘッドとなる．

CommBuf： 送信・受信バッファの先頭アドレス
引数 buf で直接指定される

CommSize： 通信量のバイト数
type と count から得られる．type が派生データ型の場合は型ごとの正確なサイズを得るために structure をトラバースすることになり，処理が複雑になる．

CommNode： 通信相手の物理ノード番号
comm で指定されたコミュニケータで指定されるプロセスグループの rank 番目のプロセスの物理ノード番号を得る必要がある．プロセスグループが既定の全ノード含むグループでない場合，ランク–ノード番号の対応表を引く処理が必要となる．

CommTag： メッセージの受信条件を付加するタグ
MPI ではコミュニケータ comm とタグ tag で受信条件を指定するため，この2つからユニークなタグを生成する．

2.3 ソフトウェアオーバヘッド削減の機会

これまでの高性能メッセージ通信ライブラリの実現には実行時に得られる動的情報のみを用いてソフトウェアオーバヘッドの削減を行うという制限があった．これに対し，コンパイル時に得られる静的情報を利用することで3章以降で述べるようなオーバヘッドの削減が可能になる．ここでは様々な種類のオーバヘッドについて検討し，静的情報を用いてそれらを除去する機会について議論する．

静的情報が利用できないことによるオーバヘッドは以下の4つに分類できる．

- バッファ間複写のコスト（特に AM に代表される複写の省略手法が適用不能）
- 動的なバッファの確保・解放や管理のコスト
- 値の範囲検査等の動的な条件の検査のコスト
- 通信セットの計算のコスト

したがって，MPI の引数の値が静的に決定すれば上記の計算コストを削減してオーバヘッドを減らすことができる．実際引数のうちのいくつかが静的に分かった場合，以下のようなコスト削減が可能になる．

CommSize（type と count）が静的に分かる場合

- 実行時のエラー検査，メッセージサイズの計算の省略
  特に派生データ型の場合に有効であり，基本データ型の場合でもエラー検査が省略可能．
- メッセージバッファの静的確保および再利用
  ready モード以外でノンブロッキング転送で DMA コントローラを用いる場合，専用のバッファが必要になる．このバッファを静的に確保したり，再利用することにより，バッファ管理のためのオーバヘッドが大きく削減される．
- 最適な通信手続きの選択
  様々な最適化手法が導入できる．たとえば，メッセージサイズに応じて push/pull ベースの通信手続きを選択したり，AP1000 の Line-sending のようにメッセージ通信専用のハードウェアが利用できる場合に送信側のバッファリングを省略することできる[6]．
- エラー検査/ハンドリングの単純化
  エラー検査およびハンドリングコードの単純化や省略による効果が期待できる．たとえば，引数の範囲エラーがコンパイル時に分かれば，システムの堅牢性を高められる．また，短いメッセージ転送に関して十分信頼性があれば，すべてのサイズの転送に必要になるようなエラーハンドリング処理はバイパス可能である．

CommNode（rank と comm）が静的に分かる場合

- 実行時の物理ノード番号計算の省略
  すでに述べたとおり，PVM のような単純なメッセージ通信ライブラリに比較して，MPI では各通信コンテクストにおける通信相手の対応をとる手間が大きい．Rank と Comm が既知であればその大部分が除去できる．また，Comm のみしか分からない場合にも各ランク番号に特化された探索ルーチンを生成できる．
- 自分自身への通信のローカルメモリ複写への変換，ハンドラの省略
  通信相手の対応をとるために通信コンテクストを保持するハンドラ構造体を生成して適切に処理する必要がある．SPMD プログラムでしばしば起きるような自分自身への通信の場合，転送処理をローカルメモリ複写で済ますことでハンドラに関わる処理を大きく削減できる．
- エラー検査/ハンドリングの単純化
  上記と同様の理由による．

3. OMPI システムの概要

2章で述べた最適化の機会を利用するために，我々は MPI プログラムを部分計算技術を用いて最適化するシステム OMPI を提案する．OMPI は C 言語で書かれた MPI プログラムに対するプリプロセッサとして機能し，大半が可搬に実現されている．またコンパイラ・OS・ハードウェア等への変更はいっさい不要である．4章で示すようにこのシステムによって，MPI の可搬性や柔軟性と AM に迫る低オーバヘッド通信

の両立を図れる．

3.1 OMPIの最適化システム構成の概要

OMPIの最適化システムは，以下に示す一連の処理をユーザの介在なしに自動的に行う：

(1) ユーザが書いたMPIプログラムに対して静的解析を行い，MPIライブラリに渡される引数に関する静的情報を得る．

(2) ソースプログラムから呼び出されるMPIライブラリ関数を得られた静的情報に関して部分計算して特殊化する．

(3) 特殊化されたMPIライブラリを呼び出すようにソースプログラムのMPI呼び出しを書き換える．

しかし，2章で述べたようなオーバヘッド削減を行うためには現状の静的解析・部分計算技術では不十分であり，MPI実装者が特定のマシン用に特化された最適化手法を導入する機会もない．我々は対策として，(1) 引数の静的情報の可能なパターンに対応して特殊化したMPIライブラリ関数（以降テンプレート関数）をあらかじめ用意しておく．(2) 自動的な部分計算に加えて，引数の静的情報に対応したテンプレート関数の定義を生成してインライン展開する，という手法をとる．4章で述べるが，実際にこの手法はうまく機能し，通常の部分計算と，部分計算では不可能な最適化を組み合わせて利用できるという利点を得た．欠点は特定のハードウェアごとにテンプレート関数を用意する必要がある点だが，その負担はコードの再利用やツールによる支援によっておさえることができる．

3.2 SUIF

現在のOMPIシステムはStanford UniversityのM. Lamらによるコンパイラツールキット SUIF [10] を拡張する形で実現されている．このツールキットは，SUIFと呼ばれる中間表現形式の定義や拡張・組合せ自由なコンパイラのパス等から構成される．SUIF形式とは，大域変数や局所変数，プロシージャ名を管理している階層化されたシンボル表と，抽象構文木または4つ組形式によるプロシージャ本体の定義からなる中間表現形式である．各パスはプログラムのSUIF形式を入力として，解析情報を付加したプログラムやプログラム変換を行った結果を再びSUIF形式で出力する．

上述のようにOMPIはターゲットマシンのバックエンドCコンパイラに渡す前に，最適化されたMPIプログラムを生成するプリプロセッサとして構築される．SUIFはこの目的に適っており，開発時間の大幅な短縮が可能になるとともに可搬性を保証する．また，

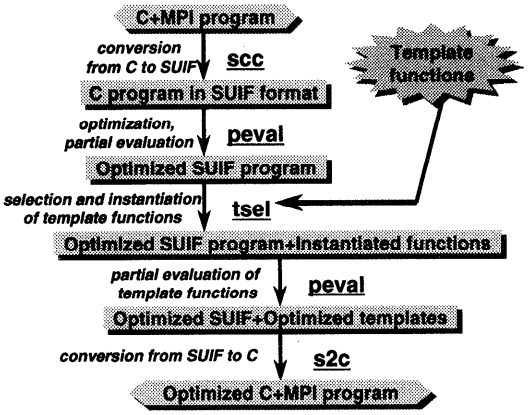


図2 OMPIシステムの概要
Fig. 2 Overview of OMPI system.

他の研究の関連する最適化手法と簡単に組み合わせてMPIプログラムをさらに最適化することも可能である．

3.3 最適化パス

ここで最適化パスについてより詳細に述べる（図2）：

(1) MPIを使って書かれたプログラムを，SUIFに付属するsccというツールでSUIF形式に変換する．

(2) SUIF形式のプログラムに対して，各種の最適化や部分計算によるプログラム変換パス（peval）を施す．pevalはSUIFに付属するporkyというツールを拡張し，定数式の計算や特化されたプロシージャ間解析を行えるようにしたものである．

(3) テンプレート関数選択パス（tsel）を適用し，適切なテンプレート関数を選択して，関数定義を生成する．

(4) ここまでの処理で特化されたプログラムに対し，再度pevalパスを適用し，生成されたテンプレート関数内部に対して静的情報を使った部分計算を施す．

(5) SUIFに付属するs2cというツールでSUIF形式からCプログラムに変換する．

3.3.1 プログラム解析・最適化パス peval

pevalは本システムの核となる解析・最適化パスであり，SUIFで使える様々なスカラ最適化器を再利用・拡張して実現している．このパスでは，constant folding, constant propagation, common sub-expression elimination, dead code elimination, if hoisting等を始めとする多数の基本的な最適化をソースレベルのプログラム変換として実現する．また，基本型に関し

てはプロシージャ間のデータフロー解析も行う．

すでに述べたようにpevalパスはテンプレート関数選択パス（tsel）の前後2回適用される．前のpevalの目的はtselパスに必要な情報を可能な限り収集することである．MPIでは通信セットを求めるのに必要な情報は大域変数ではなくすべて引数として渡されるため，引数部分に定数や静的に計算できる式が現れれば，それを利用してテンプレート関数の選択を行える．また，後のpevalは生成されたテンプレート関数内部の最適化のために行われる．

3.3.2 テンプレート関数選択パス tsel

tselはMPIライブラリ呼び出しの各引数に関する情報から，適切なテンプレート関数を選択してプログラムに付加する．具体的には，プログラムの先頭から走査してMPIライブラリ呼び出しを見つけると以下の処理を行う：

(1) MPIライブラリ呼び出しの各引数に関する情報（静的・動的，値の範囲）を集める．
(2) 引数情報を元に，適したテンプレート関数を選択する．
(3) ライブラリ呼び出しをテンプレート関数呼び出しに書き換える．
(4) テンプレート関数をプログラムに逐一展開する．

(2)において，すべての引数情報のパターンに対するテンプレート関数を実装者が用意するのは，仮に半自動化する手段が提供されたとしても，現実的ではない．

我々は2章で述べた通信セットをベースにした単純な方法を採用した．まず，MPIの引数をCommSizeとCommNodeとそれ以外を決定する引数グループ☆に分類し，各通信セットを決定する引数グループすべてが静的であればその通信セットは静的とし，それ以外の場合には動的とする．さらにCommSizeが静的に求まる場合にはその値の大小に応じてshort/long messageに分類し，CommNodeに関してもremote/localに分類する．

このようにして，**表1**に示す9通りの場合のテンプレート関数を用意すれば十分であり，また必ずしもこれらのテンプレート関数のすべてを準備する必要はない．これは，どのテンプレート関数もより一般的なバージョンのテンプレートで置き換え可能であるためである．たとえば，X_long_remoteはX_longで代用しても機能上の問題はない．

☆ 通信セットとMPIの引数との対応は2.2節で述べたとおりである．したがって，たとえばCommSizeはtypeとcountによって決定される．

表1 用意するテンプレート関数の一覧
Table 1 List of building template functions.

CommNode \ CommSize			Static: Long message	Static: Short message	Dynamic
Static	remote		X_long_remote()	X_short_remote()	X_remote()
	local		X_long_local()	X_short_local()	X_local()
Dynamic			X_long()	X_short()	X_generic()

(4)において選択されたテンプレート関数のインライン展開では，元のMPI呼び出しの引数部に現れたすべての静的な引数を展開された関数の定義内部にconst宣言して付加する．この情報は後に実行されるpevalによって部分計算のための静的情報として利用される．したがって，最も一般的なバージョンしか利用できない場合でも自動的な部分計算の恩恵を受けられる．

4. 性能評価

OMPIシステムの有効性を確認するために富士通AP1000上にプロトタイプ版の実装を行った．AP1000はノードプロセッサの性能（25 Mhz SPARC IU+FPU）に比較して通信性能（25 Mbytes/sec）が十分高い構成であり，ソフトウェアオーバヘッドによって通信性能が損なわれやすい設定である．AP1000には2つの通信モードがあり，1つは割込みベースのDMA転送であり，DMACの設定時間が大きいが送信がノンブロッキングで行える．もう1つはLine-sendingと呼ばれ，キャッシュラインにあるデータを明示的なフラッシュと同時に直接送信する方法であり，DMACが利用するバッファへの複写・DMAC設定時間・割込み処理等が省略できるが，送信側は送信が完了するまでブロックする．また，受信側は専用バッファの操作のコストが大きい．

ベースラインの比較のために基本となるMPI実装としてANLとMississippi State UniversityによるパブリックドメインのMPICH[2]を採用した．MPICHをAP1000に移植するには低レベルのADI（Abstract Device Interface）のみを実現すればよい．MPICHの性能はADIの実装に大きく依存するので，性能比較に用いるにはADIを十分効率良く実装する必要がある．我々はAP1000で標準的に提供される通信ライブラリ（以降，Nativeライブラリ）を再利用することによってこの要求を満たした．

4.1 引数キャッシングによる最適化との比較

我々の手法より一般的で単純な最適化手法として動的な引数のキャッシングがある．具体的にはMPIライブラリ関数がそれぞれ以前呼び出されたときの引数

をキャッシュしておき，そのキャッシュにヒットした場合には，以下の最適化を行ってオーバヘッドを削減することができる：

- エラー検査が省略可能
- 他の最適化に必要なパラメータチェックが省略可能
- メッセージバッファが再利用可能

引数キャッシング手法に対する我々の手法の有効性を確認するためにMPICHに引数キャッシング機能を追加したものを用意した．この機能を効率良く実現するために我々はMPIのPCR（Persistent Communication Request）機能を利用した．PCRは内部ループ等で同じ引数で同じMPIライブラリを繰返し呼び出すパターンの最適化に用いられる．具体的には毎回同じ引数を指定する代わりに，ループの前に`MPI_Send_Init/MPI_Recv_Init`で一連の引数を登録し，ループでは`MPI_Start`を繰返し呼び出せばよい．PCRを用いればキャッシュ機能は以下のように容易に実現できる：

MPIライブラリ関数の前処理 指定された引数がキャッシュしているものと一致するかどうか検査し，一致すればヒット，さもなくばミス，とする．

キャッシュミスの場合 呼ばれた関数と引数の情報をキャッシュに保存し，`MPI_Send_Init`などを呼び出してPCRのハンドルを生成し，キャッシュに保存する．その後に`MPI_Start`で通信を開始する．

キャッシュヒットの場合 キャッシュされているPCRのハンドルを使って`MPI_Start`する．

派生データ型に関しては，動的に生成された構造の比較を行うため前処理に時間を要する．単純な解決としては派生データ型をキャッシュしない，高速なマッチング関数を使う等がありうる．

4.2 AP1000へのOMPI実装の詳細

OMPIシステムをAP1000に対応させるためには，`tsel`において用いる，テンプレート関数とメッセージサイズのshort/longの境界値を適切に設定する必要がある．

プロトタイプではテンプレート関数を生成するためのツールを用意していないため，9種のテンプレート関数は手で作成している．したがって，多くのライブラリを対応させることは困難で，現在は1対1通信，放送通信等の主要な機能しか実現していない．ここでは`MPI_Send`, `MPI_Recv`について述べる．`MPI_Send`に関しては，**表2**に示すような特殊化・最適化を行う．示していない4種（`_long_remote`, `_short_remote`, `_long_local`, `_short_local`）についてはそれぞれの最適化方法を組み合わせて導出する．`MPI_Recv`に関しても同様で**表3**に示すようになる．他のMPIライブラリに関しても同様であり，したがって，半自動化ツールがあればテンプレート関数作成の手間は大きく軽減されると思われる．

表2 AP1000用のMPI_Sendの特殊化項目
Table 2 Specializing MPI_Send for AP1000.

MPI_Send	_generic()	Do nothing (essentially same as MPICH)
	_short()	Use *Line-sending* method
		Eliminate allocation/deallocation of sender buffer
		Eliminate error handling and overflow checking
		Eliminate *CommSize* calculation
	_long()	Use *DMA+Interrupt* method
		Eliminate *CommSize* calculation
	_local()	Use local copying from user buffer to system receive buffer
		Eliminate allocation/deallocation of sender buffer
		Eliminate error handling and overflow checking
		Eliminate *CommNode* calculation
	_remote()	Eliminate *CommNode* calculation

表3 AP1000用のMPI_Recvの特殊化項目
Table 3 Specializing MPI_Recv for AP1000.

MPI_Recv	_generic()	Do nothing (essentially same as MPICH)
	_short()	Use *Buffer-receiving* method correspond to Line-sending
		Eliminate allocation/deallocation of receiver buffer
		Eliminate error handling and overflow checking
		Eliminate *CommSize* calculation
	_long()	Use *DMA+Interrupt* method
		Eliminate *CommSize* calculation
	_local()	Use local copying from system receive buffer to user buffer
		Eliminate allocation/deallocation of receiver buffer
		Eliminate error handling and overflow checking
		Eliminate *CommNode* calculation
	_remote()	Eliminate *CommNode* calculation

ここで注意すべき点はOMPIにおける`MPI_Send_generic`等の（最適化されていない）テンプレート関数は，対応するMPICH実装の関数と堅牢性においても実行時間においてもほとんど同等，ということである．

また，メッセージ転送方式が複数ある場合，それらの選択はCommSizeの値と対象アーキテクチャの性質に依存してヒューリスティックに行われる．AP1000においては小さいメッセージにはLine-sending，大きいメッセージには割込み通信を用いると転送時間を最短にでき，我々が用いた64台構成のものではその境界値は60 Kbytes程度であった．`tsel`においてメッセージのshort/longの判定はこの境界値を用いて行う．

4.3 基本性能の比較

まず，Ping-pongベンチマークで基本性能に関して比較を行う．マルチプロセス環境を前提とした実装なのでメッセージ受信はポーリングではなく，割込みを用いて実現されている．測定ではレイテンシとスループットの両方を求めたが，ここでは前者のみを**図3**に示す．スループットはどの条件でもほぼ同様で

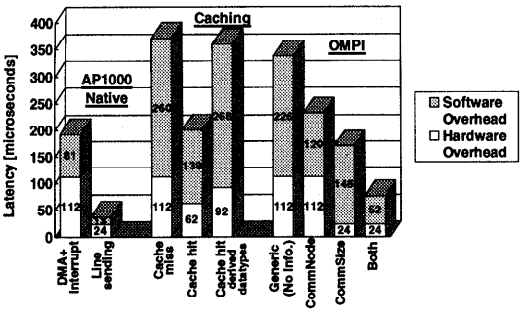


図3 Ping-pongによる通信レイテンシ
Fig. 3 Pong-pong latency results.

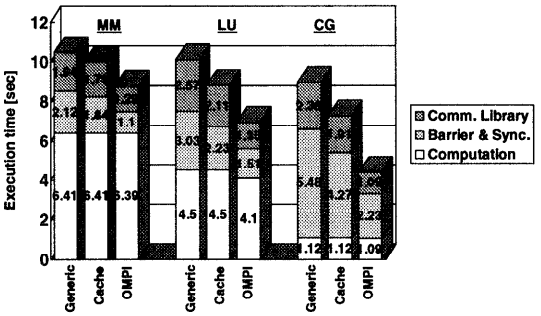


図4 数値アプリケーションコアによる結果
Fig. 4 Numeric application results.

約60 Kbytes以降，2.8 Mbytes/secに収束した．

グラフに示したレイテンシ中，ソフトウェアオーバヘッドはDMAC等へのI/O操作を行っている区間を除いた部分の所要時間を計測したものを示し，ハードウェアオーバヘッドは全レイテンシからソフトウェア部分を除いたものを示している．

左の2つのグラフはAP1000のNativeライブラリで空メッセージをPing-pongした場合のレイテンシである．割込み通信では193 μsecに対し，Line-sendingでは37 μsecとなり，ハードウェア/ソフトウェアオーバヘッドとも前者の方が大きくなっている．Line-sendingのレイテンシは，ポーリングベースのAP1000版AMは約9 μsec程度との報告[8]があるように，AMの結果に対して遜色のないものである．

中央の3つのグラフは引数キャッシングを行うMPIでの結果である．キャッシュミス時の初期設定時間が割込み通信と比較して非常に大きいが，ヒット時で基本データ型ならば，ハードウェア/ソフトウェアオーバヘッドが軽減されて割込み通信と同等になる．オーバヘッド減少の主たる要因はエラー検査や実行時の通信セットの計算の省略によるものである．PCRを利用することでキャッシュされていた通信セットを使ってほぼ直接低レベルの転送ルーチンを呼び出すことができる．一方，データ型が派生データ型の場合はソフトウェアオーバヘッドがきわめて大きくなっている．これは前述したキャッシュのマッチングをとる操作のオーバヘッドのためにキャッシングの効果が相殺されているためである．

右の4つのグラフはOMPIの結果である．Genericは静的情報がない場合，CommNode，CommSizeはそれぞれ対応する通信セットが静的に計算できる場合，Bothは両方が計算できる場合である．このように得られる静的情報に応じて大きなハードウェア/ソフトウェアオーバヘッド削減が可能になる．特にCommSizeが計算できる場合には，サイズ0の空メッセージであるので通信手続きとしてLine-sendingが選択され，ハードウェアレイテンシが112 μsecから24 μsecに激減している．また，Bothの場合はGenericに比べて両オーバヘッドとも1/4程度になるという劇的な結果を得た．削減後のソフトウェアオーバヘッドはNativeライブラリに比べてまだ相当大きいが，部分計算の向上によってギャップを詰められるものと期待する．

4.4 数値アプリケーションコアによるベンチマーク

ここでは典型的な数値アプリケーションコアを用いて実際のMPIプログラムの最適化の効果を評価する．我々は通信パターンや計算・通信の割合の異なる3つのベンチマークを用意した：

MM（行列積） 行列サイズ1024×1024，ブロック-サイクリック分割(CYCLIC(16),CYCLIC)，1回あたりの通信量大，通信パターンは**一定**．

LU（連立1次方程式のLU分解による求解） 行列サイズ1000×1000，ブロック-サイクリック分割(CYCLIC(8),CYCLIC)，1回あたりの通信量中，通信パターンはピボット選択があるため**不定**．

CG（連立1次方程式のCG法による求解） 行列サイズ128×128，ドット分割(CYCLIC,CYCLIC)，1回あたりの通信量**浮動小数1個**☆，通信パターンは**一定**．

上記のベンチマークを最適化なし（Generic），引数キャッシング（Cache），OMPIでそれぞれ実行し，平均実行時間を求めた．図4では実行時間は計算時間，バリアや通信同期のための待ち時間，MPIライブラリ内での処理時間の3つに分類して示してある．

MMでは速度向上がほとんど得られていない．これはMPIライブラリ内の時間短縮の総実行時間に占

☆ このプログラムは通信主体の例として恣意的に作られたものである．

める割合が極小であるためであるが，それでもOMPIによる効果はある．一方，CGではOMPIが他の2つに大きく勝っており，特にGenericとの比較では2倍に高速化されている．通信主体で規則的な通信パターンの場合にはOMPIの効果が非常に大きいことが分かる．

LUでは，通信パターンがコンパイル時に決定できないことからOMPIが不利になると予想されたが，結果ではCacheに比べても大きく有利になっている．これは，LUの通信パターンが不規則であるためにキャッシュミスが頻発してPCRの恩恵があまり受けられず，またキャッシュの管理コストが大きなオーバヘッドになるためである．PCRに依存せずによりチューンされた動的キャッシュを実現することによってこれらの問題は解決されると思われる．一方，通信パターンが不規則であってもOMPIによる最適化の効果が得られることが分かった．

5. 関連研究

我々の知る限り，既存の研究ではMPIのすべての引数は動的に定まることを前提にしている．したがって，レイテンシを低減するための労力はもっぱらライブラリ自体をチューンすることに向けられており．性能向上のために静的なコンパイラ技術を用いた例はない．

FrankeとHochschildらによるSP1/2へのMPI実装[1]では，レイテンシがそれぞれ30 μsec，40 μsec，スループットが9 Mbytes/sec，35 Mbytes/secときわめて低レイテンシの通信を実現している．しかし，この結果はポーリングベースのものであり，実際のマルチプロセス環境では利用できない．マルチプロセスが利用できる割込みベースの実装では200 μsecとなり，けっして高速とはいえない．

小西らによるMPI/DE[4]はNECのCenju-3で動作するMach 3.0（DenEnと呼ばれる）上のMPIの実装である．Machではカーネルレベルスレッドのupcallを用いた高速な割込み処理が可能である．受信操作を割込みスレッドが行う版，ユーザスレッドを割込みハンドラとして使う版，ユーザスレッドがポーリングする版でそれぞれ140 μsec，90 μsec，60 μsecという低レイテンシ通信を実現している．しかし，Cenju-3のネットワークDMACは物理メモリ空間を操作するが，MPI/DEはMachの仮想アドレス空間で動作しており，物理アドレスを求める方法がない．このため，上記レイテンシに加えてユーザバッファとシステムバッファ間の複写が必要になり，性能を大きく犠牲にする．また，ポーリング版はマルチプロセス環境では利用できず，kernel upcallというMachの機能に依存している点も問題である．

Sitskyらによる AP1000 への MPI 実装[7]では，標準的に提供されるOSであるCellOSに修正を加えて放送専用ネットワークを集団通信の高速化に利用している．割込み通信に類似の in-place method と AM を利用した protocol method でレイテンシがそれぞれ 171.8 μsec，64 μsec，スループットが2.69 Mbytes/sec，14.83 Mbytes/secと報告されている．AM自体のレイテンシは9 μsec程度であるから，まだ十分にハードウェア性能を引き出したとはいえない．

6. まとめと今後の課題

MPIプログラムの通信オーバヘッドを部分計算技術を用いてコンパイル時に除去する最適化システムOMPIについて述べた．性能ベンチマークを行った結果，一般的な動的最適化手法である引数キャッシングと比較しても，本システムは高い最適化効果をあげた．特に，高性能ネットワークを持つハードウェア上での通信主体の計算については大きな効果が期待できる．

OMPIでは実現していない静的最適化手法がまだいくつかある．たとえば，完全な静的情報が得られなくても変数のとりうる値の範囲を解析できれば，様々な検査処理を省略することができる．他には通信の再スケジューリングが考えられる．たとえば，Isram[3]によって報告されているように比較的単純なアルゴリズムで複数の通信をグルーピングしたり，メッセージのベクトル化やpiggybackingの技術を利用できる．より複雑な通信再スケジューリングによって大きな効果を得ることも可能であると思われる．

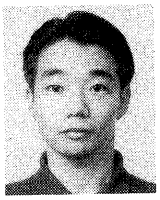
もう1つの課題はテンプレート関数の実装の労力を低減する方法で，現在我々は2つの方法を検討している．1つはソフトウェア工学の知見どおり，マシン独立な最適化部分と依存する最適化部分を分離して定義するものである．もう1つは半自動化ツールによる支援である．テンプレート関数をユーザが作成する際，引数の静的条件を少し緩めたテンプレート関数があればそれに対して最適化を行えばよい．したがって，テンプレート関数の雛型を自動生成するツールは比較的容易に実現できる．

参考文献

1) Franke, H., Hochschild, P., Pattnaik, P., Prost, J. and Snir, M.: MPI on IBM SP1/SP2: Current status and future directions, *Proc.1994*

Scalable Parallel Libraries (1994).
2) Gropp, W., Lusk, E. and Skjellum, A.: *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA (1994).
3) Islam, N., Dave, A. and Campbell, R.: Communication Compilation for Unreliable Networks, *Proc. 16th International Conference on Distributed Computing Systems* (1996).
4) Konishi, K., Takano, Y. and Konagaya, A.: MPI/DE: and MPI library for Cenju-3, *Proc. MPI Developers Conference* (1995).
5) Message-Passing Interface Forum: *MPI: A message passing interface standard*, version 1.1 (1995).
6) Shimizu, T., Horie, T. and Ishihata, H.: Low-latency message communication support for the AP1000, *Proc. 19th Annual International Symposium on Computer Architecture*, pp.288–297 (1992).
7) Sitsky, D. and Hayashi, K.: Implementing MPI for the Fujitsu AP1000/AP1000+ using Polling, Interrupts and Remote Copying, *Proc. Joint Symposium on Parallel Processing '96*, pp.177–184 (1996).
8) Taura, K., Matsuoka, S. and Yonezawa, A.: An Efficient Implementation Scheme of Concurrent Object-Oriented Languages, *Proc. 4th ACM SIGPLAN Symposium on Principles and Practive of Parallel Programming*, pp.218–228 (1993).
9) von Eicken, T., Culler, D.E., Goldstein, S.C. and Schauser, K.E.: Active Messages: A Mechnisim for Integrated Communication and Computation, *Proc. 19th International Symposium on Computer Architecture*, pp.256–266 (1992).
10) Wilson, R., French, R., Wilson, C., Amrasinghe, S., Anderson, J., Tjiang, S., Liao, S.-W., Tseng, C.-W., Hall, M., Lam, M. and Hennessy, J.: The SUIF Compiler System, Technical Report, Computer Systems Laboratory, Stanford University (1994).

(平成 9 年 11 月 5 日受付)
(平成 10 年 4 月 3 日採録)



小川 宏高

昭和 46 年生．平成 6 年東京大学工学部計数工学科卒業．平成 8 年同大学大学院工学系研究科情報工学専攻修了．平成 10 年同博士課程中退．現在，東京工業大学大学院情報理工学研究科数理・計算科学専攻助手．プログラミング言語，言語処理系，オブジェクト指向技術，並列計算機アーキテクチャ，広域分散システムに興味を持つ．ACM 会員．

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，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務める．ソフトウェア科学会，ACM，IEEE-CS 各会員．