2004－HPC－99　(20)
2004／7／31

YAMPII もう一つの MPI 実装

石　川　　裕†

並列研究基盤ソフトウェアを実現するとともに、アプリケーションユーザにも使用できる完成度の高い MPI 通信実装を提供することを目標に、YAMPII (Yet Another MPI Implementation) と呼ばれる MPI 通信ライブラリが開発されている。様々な並列計算機環境に容易にポーティング出来るように、プロセス生成機構、通信機構をモジュール化している。現在、YAMPII は、低レベル通信機構として、TCP/IP ならびに SCore クラスタシステムソフトウェア上の PMv2 通信機構が使用できる。NAS 並列ベンチマークによる予備評価では、YAMPII は既存 MPI と同等あるいはそれ以上の性能を達成している。

YAMPII – Yet Another MPI Implementation –

Yutaka Ishikawa†

YAMPII, Yet Another MPI Implementation, has been designed and implemented in order to be a research vehicle as well as providing the application users a stable high performance MPI implementation. YAMPII consists of several portable modules including process creation and point to point communication so that it is easily ported to any parallel computing environment. The current YAMPII supports the TCP/IP communication layer and the PMv2 communication layer of the SCore cluster system software. The preliminary evaluation result using the NAS parallel benchmarks shows that YAMPII achieves comparable or better performance than existing MPI implementations.

1. はじめに

1994 年に MPI-1 通信ライブラリ [1] が規格化されると共に、ベンダによる MPI 通信ライブラリ、MPICH[5] や LAM/MPI[6] などの無償で入手可能な MPI 実装が公開され、MPI は広く普及している。特に、MPICH は単なる無償ソフトウェアという位置づけでなく、MPI の規格化のプロセスの中で実装され、実質の参照モデルとなっている。MPICH は様々なマシンに移植できるように、ADI (Abstract Device Interface) と呼ばれる層を規定している。実際、多くの並列計算機環境への移植や研究ツールとして様々な試みが行なわれている。

経済産業省の「リアルワールドコンピューティング」プログラムを推進した新情報処理開発機構では、1995 年から SCore クラスタシステムソフトウェアの開発を行ない、MPICH を SCore に移植した [4]。SCore が提供する低レベル通信 PMv2 は、TCP/IP に比べて高い通信バンド幅および低い通信遅延を実現している。

高性能並列環境をユーザに提供していくためには、MPICH のバージョンが更新される度に、更新に追従して再移植を行うコストは非常に高く、継続してシステムを保守していくことが困難になっている。バージョンの更新に追従しないという選択肢も考えられるが、バグ修正や性能向上が行なわれていることが多いため、追従を余儀なくされる。

このような背景の中、MPICH から脱却し今後の研究基盤ソフトウェアを実現するとともに、アプリケーションユーザにも使用できる完成度の高い MPI 通信実装を様々なプラットフォームで提供すべく、著者は 2002 年から MPI 通信ライブラリをスクラッチから開発している。本通信ライブラリは、YAMPII(Yet Another MPI Implementation) と名付けた。現在、YAMPII では、MPI-1.2 機能が実現されている。

動作検証を優先したため、ポータビリティのための枠組や最適化が不十分である。特に集団通信の最適化が不十分ではあるが、YAMPII ソフトウェアを LGPL に基づいて公開し、多くのフィードバックを得ようとしている。本稿において、YAMPII の設計、実装の現状について述べると共に現状の性能を報告する。YAMPII ソフトウェアは、YAMPII ホームページ [11] からダウンロード可能である。

2. 設計上の考慮点

「アプリケーションユーザに対する完成度の高い

† 東京大学大学院情報理工学系研究科

MPI 通信実装を様々なプラットフォームで提供する」という YAMPII 実現の目標に対して、他の通信ライブラリとの互換性およびポータビリティの 2 つについて検討する。さらに、本節では、MPI 通信セマンティックスを効率良く実現する上での考慮点についても述べる。

2.1 互 換 性

MPI 通信ライブラリの規格では、バイナリレベルでの互換性が保証できる詳細な規格化が行なわれていない。ここでいうバイナリレベルでの互換性とは、関数インターフェイスだけでなく、エラー番号を含むエラー処理の仕様、API で使用されるデータ構造が共通であるということを意味する。例えば、MPI_COMMWORLD や MPI_COMM_NULL などの定数値は実装依存である。エラーのシンボル名は決まっているが、整数値は定められていない。MPI_Status 構造体のメンバ名もサイズも決まっていない。

様々な MPI 通信ライブラリ実装が存在するが、バイナリレベルでの互換性がないために、ある MPI 通信ライブラリ実装環境を用いてコンパイルされたバイナリオブジェクトは、他の MPI 通信ライブラリ実装環境とはリンクすることが出来ない。このため、MPI アプリケーション開発者はソースプログラムを配布するか、MPI 通信ライブラリ実装を固定した環境でコンパイルされたオブジェクトファイルを提供するしかない。

バイナリ互換性を阻害する部分は、アプリケーションプログラムでインクルードされる `mpi.h` に含まれている。そこで、`mpi.h` が公開されている通信ライブラリの実装に関しては、その `mpi.h` ファイルを使って通信ライブラリを生成して使うようにすれば良い。例えば、図 1 において、mpi-A と mpi-B という 2 つの実装があると仮定する。mpi-A が提供している mpi.h、mpi-B が提供している mpi.h をそれぞれ使って YAMPII 通信ライブラリを作成する。それぞれ作成したライブラリを YAMPII-A、YAMPII-B と呼ぶことにする。mpi-A 環境で作られたアプリケーションが、実行時に共有ライブラリを使用するように作られているならば、YAMPII-A 通信ライブラリを使って実行できる。

あらゆる MPI 実装の mpi.h ファイルを使ってコンパイルできるようにすることは不可能である。例えば、MPI 関数がマクロ文やインライン関数で定義されていて MPI 実装固有のライブラリが呼ばれていたり、YAMPII が想定する MPI_Status 構造体サイズよりも小さいサイズの場合である。現在の YAMPII 実装では、MPICH の mpi.h でもコンパイルできるように考慮している☆。

MPI 通信ライブラリでは、バイナリ互換性の問題だけでなく、MPI 通信ライブラリにおける通信セマンティックスが曖昧である。このため、ある MPI 実装ではデッドロックを引き起こさないアプリケーションでも、別の実装によってはデッドロックを引き起こすこともあり得る。この問題については、MPICH で動く通信パターンは、それが実装依存であっても、動作させる必要があるという実装方針をとった。

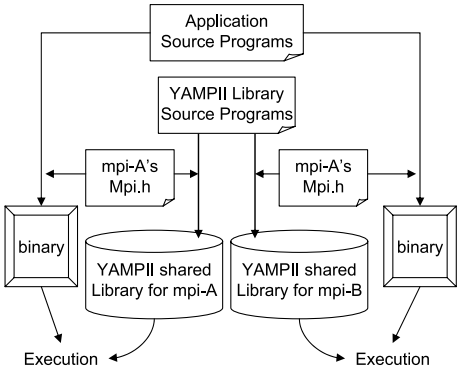


図 1 バイナリ互換

2.2 ポータビリティ

ポータビリティをあげるためには、実行環境による実装依存性を考慮し、様々な実行環境上での移植を可能とする必要がある。実行環境に依存する部分は、プロセス生成機構、プロセスに対する情報設定機構、プロセス間での通信路確立ならびに通信機構の 3 つがある。

- プロセス生成機構
  MPI プロセスの生成方法は実行環境に大きく依存する。ssh を介して生成できる環境もあれば、SCore のように独自の並列プロセス生成機構を提供している環境もある。SCore の実装のように、各計算ノードで独自デーモンプロセスを生成して実現する方法を選択することもあるだろう。
- 情報設定機構
  プロセス生成時、各プロセスに固有の情報を与える必要がある。プロセスのランク、全体のランク数、他のプロセスとの通信路確立のための情報などがある。MPI 1.2 以降、MPI_Init 引数に、プログラム引数が格納されている `argv` を渡さなくても良い仕様になった。このため、実行時オプションで、これら情報を MPI 通信ライブラリ側に渡すことが出来ない。Unix 系 OS で、これら情報を渡すには、i) プロセス環境変数として渡すか、ii) 情報提供用にプロセス生成側とのパイプを作るか、あるいは、iii) 特別なデーモンプロセスを作りそのプロセスと通信する、などの方法が考えられる。
- プロセス間通信機構
  TCP/IP などのコネクション型通信路を使用する

☆ 実際の動作確認はまだ行なわれていない。

表 1　MPI における受信メッセージ指定

タグ	送信プロセスのランク
特定の値	特定のランク
特定の値	指定せず (ANY)
指定せず (ANY)	特定のランク
指定せず (ANY)	指定せず (ANY)

場合には、全てのプロセスとの間でコネクションを張る必要がある。このために他のプロセスの IP アドレスとポート番号を知る必要がある。

これら実行環境依存性のある機構に関しては、カスタマイズ可能な枠組を提供する。

2.3　メッセージ処理

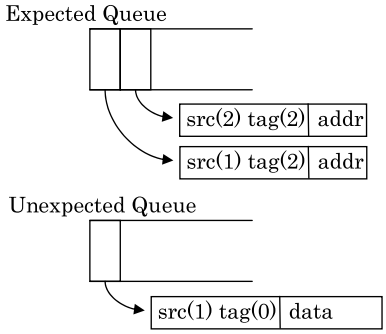
MPI 通信ライブラリは、コミュニケータによる通信相手の集合上での送受信ととらえることができる。受信側は、タグと送信プロセスのランクを指定してメッセージを受信する。この指定の組合せには、表 1 に示す通り 4 通りある。送信側のランクを指定せずに任意の送信元からメッセージを受信することが可能であるという特徴を持つ。さらに受信操作にはブロック、ノンブロックの 2 種類が提供されている。

MPI 通信ライブラリを TCP/IP プロトコル上で実装する場合には、socket インターフェイスを用いて実装することになる。他のプロセスとの間にコネクションを確立した上で、全てのコネクションからのメッセージを受信し、メッセージの内容を調べ、送信元のランクおよびタグとユーザの受信要求とのマッチングをとる。MPI 通信ライブラリのセマンティックスを実装するためには、到着したデータを全て検査する必要がある。

このために、MPI の実装では、一般に、メッセージ管理用に unexpected queue と expected queue (request queue) という 2 つの queue を用意する。

- ユーザプログラムが MPI_Recv などのメッセージ受信要求を発行した時、当該メッセージが到着しているかどうか、unexpected queue を調べる。もし、unexpected queue に当該メッセージが格納されていれば、そのメッセージを取り出す。さもなければ、その要求は expected queue に格納される。
- メッセージが到着した時、expected queue に該当するメッセージ待ち要求があれば、expected queue から取り出しデータを格納する。そもなければ、メッセージは unexpected queue に格納される。

図 2 では、2 つの MPI_Irecv 命令が発行され、それぞれ、送信元のランクが 1 でタグが 2、送信元のランクが 2 でタグが 2、のメッセージを受信しようとして、expected queue に要求が格納されている。一方、ランク 1 のプロセスがタグ 0 を持つメッセージを送信してきたが、受信側では当該メッセージを受信する要求が発行されていないので、そのメッセージは unexpected queue に格納される。




```
while(select(...)) {
 read(...);
 if(the corresponding message request
  is in the expected queue) {
  copy the message to the user's receve area.
 } else {
  enqueue the message to unexpected queue.
 }
}
```

図 2　MPI 通信ライブラリの実装方法

2.3.1　データコピー回数

上記メッセージ管理方法では、メッセージ受信時、メッセージのヘッダ部分を読み込んで送信元およびタグのマッチングを行ない、expected queue に要求が格納されていれば、メッセージはユーザが要求している受信領域にコピーすることが出来る。

一方、メッセージ受信時、expected queue に該当する要求がなく unexpected queue に格納される時、メッセージは MPI 通信ライブラリ内のバッファにコピーされる。その後、ユーザの受信要求により、unexpected queue からユーザが要求している受信領域にコピーされる。すなわち、unexpected queue にメッセージが格納されると、2 回のデータコピーが生じることになる。

2.3.2　ランデブープロトコルによるコピー削減

大規模のデータを送信する場合に、メモリコピーに伴う遅延の増大およびバンド幅の低下を防ぐため、あるいは、受信側の通信バッファを確保するために、ランデブープロトコルが使われることがある。本プロトコルとリモートメモリアクセス機能を有する NIC を使用すると、以下の手順により、CPU によるメモリコピーなしにデータの授受が行なえる。

(1)　送信側はまず送信要求を受信側に伝える。
(2)　受信側では、アプリケーションから対応する受信要求が発行された時に、受信アドレスと共に送信側に応答メッセージを送る。
(3)　送信側は、受信アドレスを使って、リモートメモリアクセス機能により送信データを直接受信

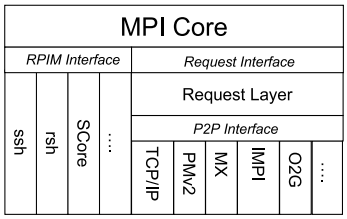


図 3 YAMPII ソフトウェアアーキテクチャ

領域に転送することが可能となる。

このように、ランデブープロトコルでは、データ転送以外にメッセージが一往復する。Grid など通信遅延が大きい環境では、ランデブープロトコルのオーバヘッドが大きくなり、かえって性能劣化が生じる [8]。

2.3.3 YAMPII におけるコピー削減

以下のようなプログラムでは、ユーザはメッセージ到着時に expected queue には既に受信要求が入っていて、データがアプリケーションのデータ領域に直接コピーされることを期待する。

```
MPI_Irecv(rb, size, MPI_BYTE, peer, ...);
MPI_Bsend(sb, size, MPI_BYTE, peer, ...);
```

しかし、実際には、各計算ノードにおける実行タイミングのずれにより、メッセージ到着時には、対応する受信要求が発行されずに、unexpected queue に格納され、結果としてデータが 2 回コピーされることがある。

YAMPII では、このような状況において、可能な限り 2 回のコピーが入らないようにする。受信したメッセージを unexpected queue にコピーしているフェーズにおいて、アプリケーション側が当該メッセージの受信要求を発行する可能性がある。YAMPII では、この時、既に受信されたデータをユーザが指定したメモリ領域にコピーし、その後受信される残りのデータは、直接ユーザメモリ領域に書く。

3. ソフトウェアアーキテクチャ

図 3 に YAMPII のソフトウェアアーキテクチャを示す。MPI Core 層では、グループ、コミュニケータ、トポロジなどの機能が実現される。MPI Core 部の下位レイヤに RPIM(Remote Process Invocation Mechanism) 層、Request 層が定義されていている。これらの層が提供する機能を利用して MPI Core 部が実現される。Request 層の下位層にさらに P2P 層が定義され、様々な通信ライブラリを利用できる。

RPIM 層はリモートプロセスの生成及び全てのリモートプロセスとの間の通信路の確立を提供する。現在、ssh, rsh を使用した RPIM 層が実現されている。

Request 層は、MPI Core 層からのメッセージ送信／受信要求を受け付け、P2P 層を用いて MPI 通信データや制御メッセージの授受を行なう。また、P2P 層からのメッセージ到着通知を受けて、MPI Core 層の受信要求に対応する。

P2P 層は 1 対 1 通信機能を実現する。P2P 層では、構造化されたデータが確実に相手に到着することを保証する。現在、TCP/IP ならびに SCore が提供する低レベル通信ライブラリである PMv2 が実装されている。

4. 実　　装

本節では、前節で述べたソフトウェアモジュールのインターフェイスについて説明する。

4.1 RPIM インターフェイス

MPI プロセスを生成する関数は直接呼ばれずに、`_yampiProcCreate` 関数ポインタ経由で呼ばれる。実装環境に応じて、本変数を変更することが可能である。

TCP/IP を使用したデフォルトのプロセス生成関数では、以下の関数群が使用される。環境変数関係では、生成および削除用の関数として `_YampiCreateEnv`、`_YampiDeleteEnv` 関数、環境変数の設定関数である `_YampiSetEnv`、環境変数の値を得る関数である `_YampiGetEnv` が提供される。リモートプロセス生成用に
`_YampiSpawnProc` 関数、リモートプロセスにシグナルを送るための関数として `_YampiSendSignal` 関数が提供される。これらの関数も最終的には関数ポインタ変数経由で実際の関数を呼び出す仕組みになっているためカスタマイズが可能である。

4.2 Request インターフェイス

Request インターフェイスは、MPI Core が使用するインターフェイスであり、expected queue および unexpected queue に対する操作関数、これら queue の要素である YamReq 構造体、YamReq 構造体に対する関数から定義される。YamReq 構造体メンバの一部以下に示す。

- yreq_state メッセージ授受の状態
- yreq_hndl p2p 層のハンドル
- yreq_mtype バッファ通信か同期／非同期通信かを表現するメッセージタイプ
- yreq_id メッセージ ID
- yreq_error エラー状態
- yreq_bufsize 受信バッファサイズ
- yreq_actsize 受信データサイズ
- yreq_extdata 受信データ領域

YamReq 構造体を生成、初期化、削除する関数として、`_YampiReqCreate`、`_YampiReqSetup`、`_YampiReqDelete` が用意されている。

queue 操作の関数は、関数ポインタ変数で定義されている。これにより、P2P 層のインターフェイスおよび P2P 層の作り全体が変更可能である。

- `_yampiPostReq`

送信あるいは受信要求を行なうための関数。送信要求では、送信用 YamReq を send queue に追加するだけである。

- `_yampiCancelReq`
  キャンセル要求を行なうための関数。
- `_yampiTestReq`
  メッセージが到着しているかどうかを検査する関数。

4.2.1 P2P 層に対するサービス

P2P 層に対して、`_YampiFinishRecv`、`_YampiFinishSend` 関数が提供される。`_YampiFinishRecv` 関数は、メッセージ受信が完了した時に、そのメッセージを Request 層に渡すための関数である。`_YampiFinishSend` 関数は、メッセージ送信が完了した時に、Request 層に通知するための関数である。

4.3 P2P インターフェイス

P2P 層を初期化するための関数として、`_YampiP2PInit` 関数がある。`_yampiPollandExec` および `_yampiFinalization` は、関数ポインタ変数である。`_yampiPollandExec` では、他の全てのプロセスからのメッセージ到着の有無を調べ、メッセージが到着していたならばコネクションハンドラ (第 4.3.2 節で述べる) で定義される受信関数を呼び出す。`_yampiFinalization` が指している関数は、MPI 終了時に呼ばれる関数である。本関数では、現在受信中あるいは送信中のメッセージ完了を待ち、P2P 層において全てのメッセージ授受が完了させる。

4.3.1 コネクション確立のための関数ポインタ

TCP/IP のようなコネクション型 1 対 1 通信機能を使用する場合には、初期化時に他の全てのプロセスとコネクションを確立する必要がある。このための関数ポインタが定義される。

SCore 環境においては、リモートプロセスの生成ならびに通信路の確立は、SCore システムが行なうため、これら関数は定義されない。

4.3.2 コネクションハンドル

他のプロセスとの通信に使用されるコネクションハンドルが以下のように定義されている。ここでは主要なメンバ変数を紹介する。

- `ycon_type`
  Request 層が下位層のネットワークタイプを調べるために使われる。
- `ycon_rank`
  本コネクションで送信できる相手プロセスの rank。
- `ycon_topsend`
  Request 層から呼び出される送信のための関数のアドレスが格納される関数ポインタ変数。
- `ycon_netpush`
  P2P 層において使用される関数ポインタ変数。コネクションがビジー状態でデータが送信できない場合、未送信データは待ち行列に格納される。送信可能になった時に、本関数ポインタ変数が指している関数が呼ばれ、未送信データが送信される。
- `ycon_netpull`
  受信メッセージを受信する関数のアドレスが格納される関数ポインタ変数。デフォルトの Request 層のデフォルト実装では、queue 操作関数の一つである `_yampiPollandExec` 関数の中から呼び出される。
- `ycon_netcancel`
  MPI_Cancel を実現するための関数で、YamReq 上位層から呼び出される。
- `ycon_close`
  コネクションを閉じるための関数のアドレスが格納される関数ポインタ変数。

このように P2P 層が提供する送信インターフェイスは、単純な送受信機能である。ポイント間でのフロー制御やリモートメモリアクセス機能を用いた通信機能は、P2P 層で実現される。これらを実現するための支援機能は上位層では提供されない。

表 2 評価環境

CPU	Dual Intel Xeon 2.8 GHz
Network	Broadcom 5700 (1G Ethernet)
Kernel	Linux 2.4.21
Distribution	Fedora Core 1
SCore	5.7 (development version)

5. 関連実装

MPICH[5] も LAM/MPI[6] もポータビリティのために階層構造となっているが、YAMPII も含めてそのインターフェイスの定義は異なっている。

MPICH は、ADI (Abstract Device Interface) と呼ばれるインターフェイスによってポータビリティを実現している。ADI の実装である channel interface では、5 つの基本関数を移植すれば良いように設計されている。 channel interface では、イーガー (Eager)、ランデブ、ゲットプロトコルが提供されている。MPICH のポーティング例では、SCore 環境に移植された MPICH-SCore や Grid 環境用の MPICH-G2 などがある。

LAM/MPI[3] は、4 つのコンポーネントから構成される。RPI (Request Progression Interface) コンポーネントは MPI の一対一通信のバックエンド通信を提供する。Coll コンポーネントは、MPI の集団通信のバックエンド通信を提供する。CR は、checkpoint および restart 機能が実現されているコンポーネントであり、Boot コンポーネントはプロセス生成のためのコンポーネントである。

6. 評　　価

YAMPII の性能を既存 MPI 実装である MPICH-

表 3 NPB Class B (8PE)
単位:Total Mops/sec (括弧内は MPICH-P4 の性能を 1 とした時の相対値)

	YAMPII(TCP/IP)	YAMPII(SCORE)	MPICH-SCORE	MPICH-P4
CG	1355.92 (1.22)	1198.76 (1.08)	1173.77 (1.06)	1111.36 (1)
EP	48.30 (1.00)	48.15 (1.00)	44.68 (0.93)	48.18 (1)
IS	32.12 (0.73)	10.58 (0.24)	80.05 (1.81)	44.26 (1)
LU	2671.00 (1.06)	2620.56 (1.04)	2625.50 (1.04)	2518.20 (1)
MG	2700.67 (1.02)	2465.66 (0.93)	2644.82 (1.00)	2654.09 (1)

P4 ならびに MPICH-SCore と比較する。MPICH-P4 は、TCP/IP を使用した MPICH である。MPICH-SCore は、MPICH を SCore 環境に移植したものである。表 2 に評価環境を示す。2 way SMP であるが、評価では一ホスト 1CPU に MPI プロセスを割り当てる。

表 3 に、NAS Parallel Benchmarks (Class B) の実行結果を示す。表中、YAMPII(TCP/IP) は P2P 層として TCP/IP を使用した場合、YAMPII(SCORE) は SCore 上の PMv2 を P2P 層として使用した場合の性能である。MPICH-P4 による実行では、ソケットバッファサイズが YAMPII のデフォルトのソケットサイズと同じ 64KBytes になるように設定している。

8 台の計算ノードによる小規模クラスタにおける結果であるため、本結果により YAMPII の優位性を結論付けるものではないが、IS を除き、YAMPII(TCP/IP) が、YAMPII(SCORE)、MPICH-SCORE、MPICH-P4 と同等あるいはそれ以上の性能が達成されていることがわかる。IS 性能が悪いのは、Alltoall 通信の最適化が行なわれていないためである。また、YAMPII(SCORE) の性能が低いのも最適化が不十分なためである。

7. おわりに

本稿では、YAMPII の設計、実装、NAS Parallel Benchmarks による性能評価を行なった。YAMPII は既存 MPI 通信ライブラリである MPICH 通信ライブラリよりも高い性能を実現している。

YAMPII は、NaReGI プロジェクトの GridMPI の核としても使用されている [10]。GridMPI では、P2P 層で IMPI (Interoperable MPI) 通信機能が実現されている。また、ソケットインターフェイスの問題点を克服する目的で設計された O2G 通信機構の開発でも YAMPII が使用されている [9]。

YAMPII は、MPICH 配布に付随している Test Suite および Intel 社から配布されている Test Suite を用いて検証が行なわれている。現在の実装は、MPI 規格に対する完成度を優先したため、多くの最適化が残されている。例えば、集団通信のアルゴリズムは、通信遅延およびバンド幅、計算ノードの台数、データサイズに応じて最適なアルゴリズムに切替えなければならないが、現在の実装は固定されている。TCP/IP 環境でのプロセス生成機構は単純なアルゴリズムのみを提供している。共有メモリを利用する通信機構も未実装である。SCore 環境に関しては、PMv2 が提供するリモートメモリアクセス機能を利用していない。

今後、これら最適化を進めると同時に MPI-2 機能を実現していく。MPICH や LAM/MPI 環境下で作られたバイナリが YAMPII 環境で稼働するかどうかの検証も今後行なう。

謝　辞

産業技術総合研究所の松田氏には、YAMPII を使った O2G の開発ならびに GridMPI の開発を通して、YAMPII の改良に協力して頂いている。ここに感謝する。

参 考 文 献

1) Message Passing Interface Forum, http://www.mpi-forum.org
2) William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, “A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard,” Parallel Computing, vol. 22 number 6, pp. 789–828, 1996.
3) Jeffrey M. Squyres and Andrew Lumsdaine, “A Component Architecture for LAM/MPI,” Proceedings of 10th European PVM/MPI Users’ Group Meeting, Lecture Notes in Computer Science, no. 2840, Springer-Verlag, pp. 379–387, 2003.
4) PC Cluster Consortium, http://www.pccluster.org
5) MPICH, http://www-unix.mcs.anl.gov/mpi/mpich/
6) LAM/MPI, http://www.lam-mpi.org/
7) 石川、松田、工藤、手塚、関口、GridMPI – 通信遅延を考慮した MPI 通信ライブラリの設計、情報処理学会、SWOPP’03, 2003.
8) Motohiko Matsuda, Yutaka Ishikawa, and Tomohiro Kudoh, “Evaluation of MPI Implementations on Grid-connected Clusters using an Emulated WAN Environment,” CCGRID2003, 2003.
9) 松田、石川、工藤、手塚、MPI 通信モデルに適した非同期通信機構の設計と実装、SACSIS 2004, 2004.
10) GridMPI, http://www.gridmpi.org/
11) YAMPII, Yet Another MPI Implementation, http://www.ilab.is.s.u-tokyo.ac.jp/yampii/
12) Myrinet Express(MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, 2003. (http://www.myri.com/scs/MX/doc/mx.pdf)