社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－HPC－116 (1)
2008／8／5

MPI通信ライブラリの最適化と性能評価

松葉浩也[†] 野村哲弘[††] 石川裕[†,††]

MPI通信ライブラリの実装の一つであるYAMPIを最新の低遅延ネットワークに対応させるため、効率的な通信インタフェースの設計と実装の最適化を行った。新しい通信インタフェースはMyrinet-10GのMX通信ライブラリに代表されるようなマッチング機構を持った通信デバイスを効率的に扱うことが可能である。また、実装の最適化としては通信状態の管理を行うオブジェクトの確保と解放を高速化し、送受信に必要な命令実行数を削減した。性能評価では最適化によって通信遅延が1.5から1.7マイクロ秒削減されたことが示された。アプリケーションベンチマークでは最大4.6%の高速化を達成した一方、まったく効果のないアプリケーションもあった。

Optimazation and Evaluation of an MPI Communication Library

Hiroya Matsuba,[†] Akihiro Nomura[††] and Yutaka Ishikawa[†,††]

An implementation called YAMPI is optimized in order to adopt high-speed interconnects, such as Myrinet-10G. We design a new efficient communication interface for the Myrinet network and the overall implementation of YAMPI is optimized by reducing the overhead to allocate and deallocate objects that keeps the statuses of outstanding communication. Results of performance evaluation show that communication latency is reduced by 1.5 to 1.7 μs and application benchmarks run faster at most by 4.6%, while some benchmarks show that our optimization has no effect to performance of these benchmarks.

1. はじめに

MPI通信ライブラリが分散メモリ型並列計算機における標準的な通信ライブラリとなって10年以上が経過しており、並列アプリケーションを作成する際の事実上の必須ライブラリとなっている。このMPI通信ライブラリには様々な実装が存在し、MPIのリファレンス実装である米国アルゴンヌ国立研究所によるMPICH[7]をはじめとして、OpenMPI[6),16)]や並列計算機メーカーによる独自のMPIなどが一般的に使われている。MPI通信ライブラリの性能は並列計算機全体の性能を左右する要素の一つとなるため、高速化に対する努力も各所で行われている。例えば、米国オハイオ州立大学のグループがMPICHをInfiniband[9)]に最適化したMVAPICH[10),11)]や、フランスINRIAのグループが共有メモリをはじめMPICH2の通信全般を最適化したMPICH2-Nemesis[4)]などが代表的である。

YAMPI[21)]はPCクラスタにおける実運用および基盤システムソフトウェアの研究を目的に作成されたMPIの実装であり、GridMPI[20)]のコアとしてグリッド環境向けMPI実装としても使用されているライブラリである。YAMPIは多くの種類のOS上で安定に動作する完成度の高いライブラリに仕上がっているが、2003年のYAMPI登場以降にも前述のMPICH2-Nemesisのように細部にわたって高速化のための工夫が凝らされたライブラリが登場したり、遅延が5マイクロ秒を下回る10Gbpsクラスのネットワークが主流になるなど、MPIをとりまく環境は変化しており、YAMPIも最新環境にあわせた高速化が必須課題となっている。

本稿ではYAMPI通信ライブラリを高速化するため、設計と実装を見直し、YAMPI3と呼ばれる新しいバージョンを作成した。具体的にはYAMPI3では、(1) 重複するマッチング処理の排除 (2) 実装の軽量化、の2点において最適化を行った。具体的には (1) はMyrinet-10G用ライブラリであるMX通信ライブラリを効率よく使用するための通信インタフェースの見直しであり、(2) はインストラクション数が他のMPIに比べて多くなっているYAMPI全体の実装を見直すことによる軽量化である。

これらの最適化を行った上で、YAMPI3の性能を

† 東京大学情報基盤センター
Information Technology Center, The University of Tokyo
†† 東京大学情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

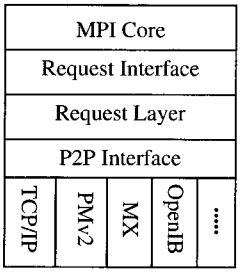

図 1 YAMPI のソフトウェアアーキテクチャ

測定した結果、MPI_Isend および MPI_Irecv に必要な実行命令数は 63%削減されたことが示された。これにより TCP を用いた場合の遅延が 1.5 マイクロ秒削減され、Myrinet-10G では通信インタフェースの見直し効果を含めて 1.7 マイクロ秒の遅延削減に成功した。一方、HPC Challenge Benchmarks[8] および NAS Parallel Benchmarks[1] を用いたアプリケーションレベルベンチマークでは前者で最大 4.6%の高速化を達成したものの、後者では効果はまったく見られず、マイクロ秒単位の遅延を削減してもアプリケーション性能に大きな影響はないことがわかった。

2. 既存 YAMPI の問題点

本章では既存の YAMPI version 2（以下 YAMPI2）における問題点を指摘する。

2.1 マッチング処理の重複

MPI のメッセージには tag および communicator で表される通信コンテキストがあり、MPI の受信命令はコンテキストが一致するメッセージのみを受けとることとなっている。そのため MPI の実装には受信リクエストのキューと受け取ったメッセージを照合する機構が実装されている。この照合処理をマッチングと呼ぶ。一般にマッチング機能を提供する通信ライブラリは、ネットワークカードから送られてくる受信データを一旦ライブラリ内の一時バッファに受け取る。その上でそのメッセージに付けられているタグを確認し、一致する受信命令を探した上で受信命令で指定されている本来の受信バッファに受信データをコピーする (ここではリモートメモリアクセスは考えないこととする)。つまりマッチング機能を提供する場合、受信データのコピーは必須である。

ここで YAMPI2 のソフトウェアアーキテクチャを図 1 に示す。この図の中の Request Layer と書かれた部分がマッチング機構を持つ部分である。前述した受信データのコピーはこの層で発生する。この図に示されているように YAMPI2 ではすべての通信デバイスが Request Layer の下層にある P2P Layer につながっている。この設計はデバイスに依存した通信ライブラリが Point-to-Point の低レベル通信機構を提供することを前提とした設計であるが、実はすべての通信デバイスがこのような低レベル通信機構を提供しているわけではない。低レベル通信機構を提供しないデバイスの代表が Myrinet-10G[14] である。Myrinet-10G は MX[13] と呼ばれる通信ライブラリを用いて使用するが、MX は Myrinet-10G 用のデバイスドライバの機能にとどまらず、MPI から使用することを前提としたマッチング機構を備えている。

YAMPI2 から MX を使うにあたり問題となるのはこのマッチング機構の重複である。前述のようにマッチングはデータのコピーを伴うものであり、MX もマッチングのためにライブラリ内でデータのコピーを行うことがある。このようなライブラリを Request Layer の下層に置くとデータのコピーが 2 回となってしまい、MX のマッチング機構を直接使用した MPICH-MX[12] などの MPI 実装に比べて性能が悪くなってしまう。

ここでは Myrinet(MX) 対応における問題点としたが、MX と同じようにハードウェア固有の通信ライブラリがマッチング機能を持つ例は他にもあり、Quadrics の QsNet[17]、Cray の SeaStar が代表的である。これらに対応した実装は YAMPI には存在しないが、対応することがあるとすれば MX と同じ問題が発生する。この問題を解決すべく、マッチング機構を持つライブラリを効率よく使用するインタフェース設計が必要である。

2.2 ステップ数の多い実装

YAMPI2 は MPICH2 と比較して全体的に処理のステップ数 (実行インストラクション数) が多くなっている。予備調査としてハードウェアカウンタを用いて 4byte の MPI_Isend に要するステップ数を計測したところ、TCP 上の MPICH2 (Nemesis[4] を使用) が 516 インストラクションであるのに対し、YAMPI2 では同じく TCP を使用する場合で 1542 インストラクションとなっている (両者ともユーザーモードの部分のみを測定しているため、TCP プロトコル処理は含まない)。このステップ数の差を分析し軽量化した実装にすることで、性能が向上する可能性がある。

3. 最 適 化

本節では前節で指摘した既存 YAMPI の問題を軽減するための最適化手法について述べる。本節で述べる最適化を行った YAMPI を YAMPI3 と呼ぶこととする。

3.1 新たな低レベル通信インタフェース

Myrinet-10G 向けの MX 通信ライブラリをはじめとしたマッチング機能を持つ通信ライブラリを効率的に使用するために、YAMPI2 の Request Layer から下 (前節の図 1 参照) の設計を変更した。新たに設計

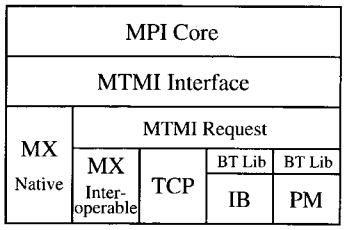

図 2 YAMPI3 のソフトウェアアーキテクチャ

したプロトコルスタックを図 2 に示す。YAMPI3 においては YAMPI2 の Request layer の置き換えとして、MPI Core からは実装を完全に隠蔽した中間通信ライブラリを作成した。この新たな Request Layer を MTMI (MPI-Transport Matching Interface) と呼ぶ。

MTMI はマッチングを含んだインタフェースを持っており、MX などのマッチング機能付き通信ライブラリに対してはデータ型の変換程度の軽い処理のみを行うだけですべての処理を下位通信ライブラリに移譲する (図中の MX Native)。この場合は送受信キューの管理やマッチング処理はすべて MX などの下位通信層が行う。これによりデバイス依存通信ライブラリが持つ豊富な機能を生かすことが可能となり、このようなデバイスを使う際の性能は YAMPI2 に比べて向上する。

MTMI タイプのインタフェースの問題はマッチング機能を持たないデバイスに対応するための実装が多くなる点にある。そこで MTMI では送受信キューの管理、マッチング処理など多くのデバイスに共通する処理はライブラリとして提供している (図中の MTMI_Request)。さらに SCore システムソフトウェアで使われる PM[19] や Infiniband 用の OFED[15] のような高速インターコネクトは、多くの共通点があり、MPI から使用するための実装はほぼ共通化できるためそのための共通実装も提供している (図中の BT lib)。これによりデバイス依存の実装規模は YAMPI2 と同程度に抑えられている。

3.2 実装の軽量化

ハードウェアカウンタを用いて YAMPI2 の MPI_Isend、MPI_Irecv、MPI_Wait (すでに完了しているリクエストの Wait) に要するステップ数の内訳を測定すると表 1 のようになる。この表からわかるように、YAMPI2 では Request Layer の処理に入る前段階でリクエストを管理するオブジェクトの作成とハッシュへの登録に多くのステップを要していることがわかる。また、Wait でもリクエスト管理オブジェクトの解放に多くのステップを費やしていることがわかる。ここで「リクエスト」とは処理中の送受信命令を意味し、例えば MPI_Isend 関数が呼び出されると

表 1 YAMPI2 のステップ数

	Isend	Irecv	Wait
リクエストの作成	367	367	-
ハッシュへの登録	428	428	-
Type の処理	195	148	-
Request Layer	422	87	302
リクエスト検索	-	-	47
リクエスト解放	-	-	358
その他	130	120	43
合計	1542	1124	750

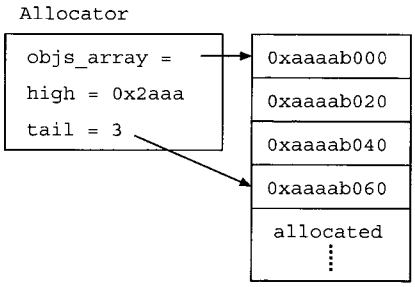

図 3 YAMPI3 の Slab Allocator

MPI はこの命令の状態を管理するためのオブジェクトを確保し、送信が終わるまで規定の手順で状態を遷移させる。また、このオブジェクトの ID をユーザープログラムに返し、ユーザーは MPI_Test などで処理中の送信命令の状態を調べるときにこの ID を使用する。

YAMPI3 ではこのリクエスト管理オブジェクトの生成および解放を高速化するため、Slab Allocator[3),5)] と呼ばれる方式を採用した。図 3 に示すように Slab Allocator (特に Linux で採用されている実装) はあらかじめ一定数のオブジェクトを確保しておき、そのオブジェクト群のアドレスを配列に保持しておく。そして、確保要求が来た場合はその配列の末尾にあるアドレスを返し、解放要求が来た場合には解放されたオブジェクトのアドレスを配列の末尾に加える。これは生成と解放がいずれも一回のメモリアクセスで完了する高速な方法として知られている。

さらに、YAMPI3 では割り当て済みのオブジェクトをハッシュテーブルでは管理せず、オブジェクトのアドレスの下位 32bit を直接 ID としてユーザープログラムに渡すこととした。ユーザープログラムから ID が渡されたときは別途保存してある上位 32bit と合わせることでオブジェクトのアドレスを求めることができるため、ハッシュアクセスのオーバヘッドをゼロにすることができる。64bit アーキテクチャの計算機上でもこの方法を安全に動作させるため、Slab Allocator の実装の中で、すべてのオブジェクトが同じ上位 32bit を持つような割り当てを行っている。

オブジェクト管理の効率化の他、YAMPI3 は MPI

表 2 測定用コンピュータの仕様

CPU	Opteron 2214 (Dual Core, 2.2GHz) x 2
Chipset	Broadcom HT1000+HT2100
Memory	DDR2-667 Dual Channel (4GB)
Network	Myrinet-10G
OS	CentOS 5 (Linux 2.6.18)

表 3 YAMPI3 のステップ数

	Isend	Irecv	Wait
リクエストの作成	20(-347)	20(-347)	-
ハッシュへの登録	0(-428)	0(-428)	-
Type の処理	67(-128)	143(-5)	-
Request Layer	433(+11)	223(+136)	141(-161)
リクエスト検索	-	-	0(-47)
リクエスト解放	-	-	128(-230)
その他	48	22	82
合計	568(-974)	408(-716)	352(-398)

表 4 片道遅延 (マイクロ秒)

YAMPI2(MX)	4.78	YAMPI3(MX)	3.08
YAMPI2(TCP)	21.80	YAMPI3(TCP)	20.30

のデータ型を扱う関数も最適化を行った。これは YAMPI2 の実装に本質的な変更を加えたわけではないが、重複して呼び出されている関数があったためそれを 1 回にし、頻繁に使用される関数をインライン展開した。

4. 性能評価

本章では YAMPI3 で新たに導入した 2 点の最適化についてその性能評価を行う。評価に用いたのは 32 台のクラスタであり、各ノードの仕様は表 2 のようになっている。Myrinet-10G は Ethernet としても使用できるが、本測定では HPC 向けの Myrinet モードで動かしており、本章で後に述べる Myrinet 上での TCP 性能とは Myrinet モードのドライバが提供する Ethernet エミュレーションを用いての測定値である。

4.1 ステップ数

YAMPI3 において MPI_Isend、MPI_Irecv、MPI_Wait に要するステップ数を表 3 に示す。括弧内は YAMPI2 との差である。Slab Allocator の導入により、リクエスト管理オブジェクトの確保と解放に要するステップが Isend、Irecv で 63%、Wait で 53% 削減された。

4.2 通信遅延

通信遅延の測定のため 4 バイトメッセージの Ping-Pong に要する時間を測定した。測定値を 2 で割った片道遅延を表 4 に示す。結果は 10 回測定を行った平均値である。MX、TCP はそれぞれ MX を使用した Myrinet-10G での性能および Myrinet-10G の Ethernet エミュレーション上で TCP を使用した場合の性能である。

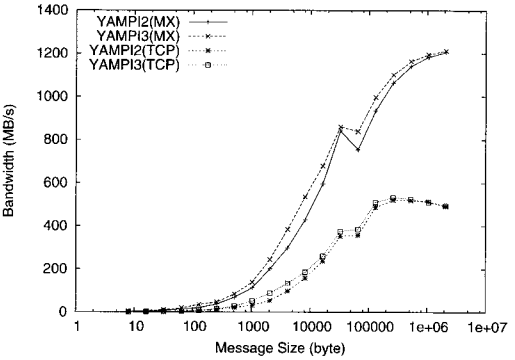

図 4 バンド幅

YAMPI2 と YAMPI3 を比較すると MX で 1.7 マイクロ秒、TCP で 1.5 マイクロ秒遅延が短くなっている。前述したように YAMPI3 では送受信に必要なステップ数が YAMPI2 より少なくなっているため、その差が遅延に現れている。さらに MX では YAMPI2 における Request Layer を YAMPI3 は完全にバイパスしているため、TCP より削減幅が大きい。

4.3 バンド幅

バンド幅測定のため 2 台のノードに一個ずつ配置した 2 つのプロセス間で Ping-Pong を行い、所要時間を測定した。図 4 に結果を示す。

MX に関してはほとんどすべてのメッセージサイズにおいて YAMPI3 が YAMPI2 より高速である。特に 32KB 以下のメッセージで差が大きいが、これは第 2 章で指摘した YAMPI2 におけるインタフェースの問題である。MX がリモートメモリアクセスを行わない 32KB 以下の通信では YAMPI2 は受信時に 2 回のデータコピーを行っているため、その分のオーバヘッドがバンド幅として現れている。MX のメッセージサイズの大きな部分では YAMPI2、YAMPI3 ともにハードウェアの理論最高値に近い 1200MB/s 程度の性能が得られている。

TCP でも YAMPI3 は YAMPI2 よりも高いバンド幅を示している。これは遅延が短くなっているためである。メッセージサイズが 64KB の点でバンド幅の伸びがなめらかでないが、これはランデブー通信への切り替えのためである。YAMPI2、YAMPI3 ともピークバンド幅は理論性能の半分程度にとどまっている。これは TCP のプロトコル処理や送受信の際に発生するデータコピーのオーバヘッドのためである。

4.4 アプリケーションベンチマーク

本節ではアプリケーションレベルのベンチマークで YAMPI3 の最適化の効果を調べる。ベンチマークは HPC Challenge Benchmarks[8] から MPIFFT、NAS Parallel Benchmarks[1] から CG の Class D を使用す

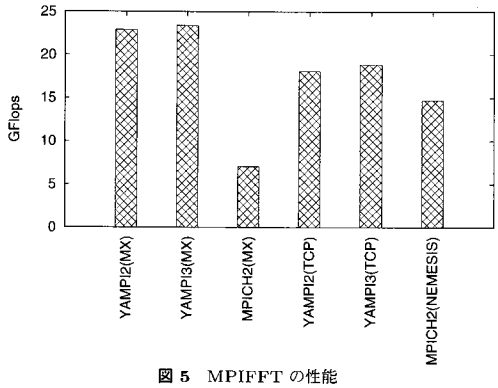

図 5 MPIFFT の性能

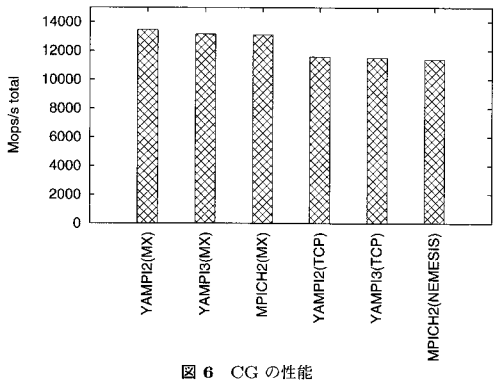

図 6 CG の性能

る。いずれのベンチマークも 32 ノードに各 4 プロセスの合計 128 プロセスで性能を測定した。また、2CPU 以上の Opteron サーバは NUMA であるため、メモリ割り当て方式によってアプリケーションの性能に差が生まれる。本測定では Linux の numactl コマンドを用いて、すべてのプロセスが常にもっとも近いメモリを使用するように設定して実行している。

MPIFFT

MPIFFT は高速フーリエ変換を行うベンチマークである。HPC Challenge Benchmaks の入力パラメータとして HPL_N=120000 を入力した (HPL とは無関係であるが、本ベンチマークは HPL のサイズをもとに他のベンチマークのサイズが決定される)。このとき MPIFFT_N は 1073741824 となっている。32 ノードではノードあたり 3.2GB 程度のメモリを使用する問題サイズであり、各ノードが 4GB のメモリを持つクラスタにおけるベンチマークとしては適切なサイズである。FFT は通信負荷の高いアプリケーションとして知られており、本測定の条件で MPIFFT ベンチマークは実行中に約 1MB の All-to-All 通信を行う。All-to-All のアルゴリズムは MPICH2 と同じものを YAMPI2、YAMPI3 双方に移植し、条件を揃えて測定を行った。図 5 に結果を示す。MX では YAMPI3 は YAMPI2 より 2.1% 高速となっている。MPICH2-MX は異常な値が出ており、この原因は不明である。TCP の場合では YAMPI3 は YAMPI2 より 4.6%、MPICH2-Nemesis と比較して 28%高速である。

CG

NAS Parallel Benchmarks の CG は共役勾配法を使用したベンチマークである。問題サイズは Class D を選択した。32 ノードの場合ノードあたり 1.1GB のメモリを使用する。このベンチマークは 8Byte の Allreduce に相当することを MPI_Isend と MPI_Irecv を用いて記述しており、この操作の性能が全体の性能に影響するため MPI 通信ライブラリは低遅延であることが求められる。また Class D の 128 ノードでは 750KB 程度の通信も発生するためバンド幅も重要である。結果を図 6 に示す。MX では YAMPI3 は 2.3%ほど YAMPI2 に劣る結果となっており、MPICH2-MX とほぼ同じ性能である。YAMPI3 や MPICH2-MX は MPI_Recv と MX の受信命令が一対一対応しているが、YAMPI2 は常に小さなメッセージに対する受信を多数 MX にリクエストした状態を保っている。CG のように非同期的に多数の小さなメッセージを交換するアプリケーションでは YAMPI2 のような方式にメリットがある可能性がある。TCP では YAMPI2 と YAMPI3 に性能差は見られなかった。

5. 関連研究

MPICH2-Nemesis[4] は共有メモリを用いた通信を中心に MPICH2 の低レベル通信層を最適化した MPI の実装である。Nemesis はキャッシュを考慮した実装や、スピンロックを使わない高速ロックなどの仕組みで高速化を図っているが、本稿における MPIFFT や CG ベンチマークではこれらの最適化に効果はない。

OpenMPI は MCA と呼ばれるコンポーネントアーキテクチャを採用しているため、コンポーネント間は関数テーブルを用いた間接ジャンプとなりオーバヘッドが大きい。文献 2) はこのコンポーネントアーキテクチャのオーバヘッドを評価しており、0.3 マイクロ秒の遅延の増加が NAS Parallel Benchmarks で最大 3% の性能低下につながることが報告されている。ただしこの論文で用いている NAS Parallel Benchmarks は問題サイズが小さく、遅延の影響が相対的に大くなる設定である。

6. おわりに

本稿では MPI 通信ライブラリである YAMPI を高速化するため、MX などのマッチング機能を持つライブラリを効率よく扱うための新しいインタフェース

の設計および実装全体の最適化を行った。新しく設計したMTMI中間通信ライブラリはマッチング機能を持つデバイスに関してはそのデバイスの機能をそのまま利用し、低レベル一対一通信機能のみを提供するデバイスに関してはMPIに対してマッチング機能を提供する。これにより、多くのデバイスに対応可能なYAMPIの利点を残しつつ、MXのようなマッチング機能付きライブラリも効率的に扱えるようになった。また、実装の最適化としては通信状態を管理するオブジェクトの確保と解放をSlab Allocatorによって高速化した。

通信性能の性能評価では、本稿で述べた手法を実装したYAMPI3は従来のYAMPI2に対して実行命令数が63%少ないことが示された。この命令数削減によって通信遅延がTCPの場合で1.5マイクロ秒短縮され、MXにおいてはインタフェースの変更による効果とあわせて1.7マイクロ秒高速化された。一方、FFT(高速フーリエ変換)とCG(共役勾配法)を用いたアプリケーションベンチマークでは、前者は2.1%から4.6%の高速化が見られたものの後者ではまったく最適化の効果が見られなかった。

新しく開発したMTMI中間通信ライブラリはYAMPIへの依存はなく単独でも動作が可能であるため、MPICH2の低レベル通信層としても利用可能である。MTMIを利用したMPICH2はMPICH2-SCoreとして今後のSCoreシステムソフトウェア[18]の一部としてリリースされる予定である。

参考文献

1) D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks, 1994.
2) B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the component architecture overhead in OpenMPI. In *12th European PVM/MPI Users' Group Meeting*, 2005.
3) Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pp. 87–98, 1994.
4) Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in mpich2 using the nemesis communication subsystem. *Parallel Comput.*, Vol. 33, No. 9, pp. 634–644, 2007.
5) Brad Fitzgibbons. The linux slab allocator.
6) Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pp. 97–104, Budapest, Hungary, September 2004.
7) William Gropp and Ewing Lusk. User's guide for mpich, a portable implementation of MPI. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, Mathematics and Computer Science Division, February 1996.
8) HPCC. http://icl.cs.utk.edu/hpcc/.
9) InfiniBand. http://www.infinibandta.org.
10) J.Liu, J.Wu, S.Kini, P.Wyckoff, and D.Panda. High Performance RDMA-Based MPI Implementation over InfiniBand, 2003.
11) MVAPICH. http://mvapich.cse.ohio-state.edu/.
12) MPICH-MX. http://www.myri.com/scs/.
13) Myrinet Express. http://www.myri.com/scs/MX/doc/mx.pdf.
14) Myrinet. http://www.myri.com.
15) The OpenFabrics Alliance. http://www.openfabrics.org/.
16) OpenMPI. http://www.open-mpi.org/.
17) QsNet. http://www.quadrics.com/.
18) SCore. http://www.pccluster.org.
19) Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2: High performance communication middleware for heterogeneous network environments. In *SC2000: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.
20) Ryousei Takano, Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Fumihiro Okazaki, Yutaka Ishikawa, and Yasufumi Yoshizawa. High Performance Relay Mechanism for MPI Communication Libraries Run on Multiple Private IP Address Cluster. In *Proceedings of International Symposium on Cluster Computing and the Grid (CCGrid2008, to appear)*, 2008.
21) 石川裕. YAMPII もう一つのMPI実装. 情報処理学会研究報告 SWoPP'04, 2004.