広域MPI用の局所性を考慮した接続管理とランク割当て

斎藤秀雄[†,††] 田浦健次朗[†]

クラスタ間の接続数を制限し，直接通信できないプロセスのためにメッセージを中継する接続管理機構を提案する．また，二次割当て問題を解くことによって低通信オーバヘッドのランク割当てを見つける手法を提案する．提案手法はいずれも，短時間のプロファイリング実行から遅延と通信量に関する情報を取得することによって，煩雑な手動の設定をすることなく局所性を考慮した性能最適化を行う．これらの手法を用いて MC-MPI という広域環境用の MPI ライブラリを実装して，4 クラスタにまたがる 256 実ノード上で NAS Parallel Benchmarks を実行して性能評価を行った．その結果，10%のプロセス対でしか接続を確立しなくても全プロセス対で接続を確立した場合と同様もしくはそれ以上の性能が出た．また，局所性を考慮しない割当てを用いた場合より最大 300%高い性能が出るランク割当てを見つけることができた．

Locality-aware Connection Management and Rank Assignment for Wide-area MPI

Hideo Saito[†] and Kenjiro Taura[†]

We propose a connection management scheme that limits the number of inter-cluster connections and forwards messages for nodes that cannot communicate directly. We also propose a rank assignment scheme that finds rank-process mappings with low communication overhead by solving the Quadratic Assignment Problem. Our proposed methods perform locality-aware communication optimizations, and do so without tedious manual configuration by obtaining latency and traffic information from a short profiling run of the environment and the application. Using these methods, we implemented a wide area-enabled MPI library called MC-MPI, and evaluated its performance by running the NAS Parallel Benchmarks on 256 real nodes distributed across 4 clusters. MC-MPI was able to limit the number of node pairs that established connections to just 10% without suffering a performance penalty. Moreover, MC-MPI was able to find rank assignments that resulted in up to 300% better performance than locality-unaware assignments.

1. はじめに

近年，Wide Area Network（WAN）の帯域の増加にともない，異なる位置に存在する複数のクラスタを接続して並列計算を行う機会が増加した．複数のクラスタを用いることによって単一のクラスタを用いる場合よりはるかに多くの計算力，メモリ，ストレージが利用可能になるため，広域環境を意識した並列プログラミングシステムが必要とされている．

広域環境用の並列プログラミングシステムは，高い性能を実現しつつ，以下の要件を満たす必要がある．

スケーラビリティ システムの様々な制限（e.g.，メモリ量，ファイルディスクリプタ数，ステートフルファイアウォールのセッション数）に触れずに何千ものプロセスにスケールする必要がある．プロセス数が n のときに $O(n^2)$ 本の接続を確立する可能性があるような単純な手法はスケールしない．

広域環境の考慮 クラスタ間の通信は遮断されていることが多いことを意識する必要がある．全プロセス間で通信が可能と仮定している単純な手法は広域環境では動作しないことが多い．たまたま確立できた接続を用いてメッセージのルーティングができなければならない．

局所性の考慮 最初の 2 つの要件より，一部のプロセス対でのみ接続を確立することになる．高い通信性能を保つために，それらのプロセス対は局所性を考慮して選択する必要がある．

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 日本学術振興会特別研究員
Research Fellow of the Japan Society for the Promotion of Science

アプリケーションの考慮 接続を確立するプロセス対は，アプリケーションの通信パターンも考慮して選択すべきである．

適応性 上記の要件を，煩雑な手動の設定なしで自動的に満たすべきである．

本稿では，上記の要件を満たすためにプロファイリング実行から取得した遅延と通信量に関する情報を用いる接続管理手法を提案する．また，それらの情報を用いて低通信オーバヘッドのランク割当て（プロセスへの ID の割当て）を見つける手法も提案する．これらの手法を用いて Multi-Cluster MPI（MC-MPI）という広域環境用の MPI ライブラリを実装し，4 つの実クラスタで NAS Parallel Benchmarks（NPB）[5] を実行して性能評価を行った．

以降，プロファイリング実行について 2 章で説明し，提案する接続管理とランク割当てについてそれぞれ 3 章と 4 章で説明する．その後，5 章で評価実験について説明する．6 章で関連研究について述べた後，最後に 7 章でまとめと今後の課題を述べる．

2. プロファイリング実行

提案する 2 つの手法では遅延行列と通信行列の 2 つの行列を用いて局所性を考慮するが，それらの行列は計算環境とアプリケーションのプロファイリングを行うことによって取得する．

2.1 遅延行列

遅延行列 D の各要素 d_{ij} は，実行に用いる計算環境におけるプロセス i とプロセス j の間の遅延を表す値である（$i, j = 0, 1, 2, \ldots, n-1$）．具体的には d_{ij} をプロセス i とプトセス j の間の往復時間（RTT: Round-Trip Time）とするが，全プロセス対で RTT を測定しようとすると長時間かかってしまう．以下のような理由のため，異なるクラスタのプロセスどうしの測定が特に時間を要する．

- あるプロセスとの測定にかかる時間はそのプロセスとの RTT に比例するが，異なるクラスタのプロセスとの RTT は長い．
- 異なるクラスタのプロセスとの通信はファイアウォールによって遮断されていることがあり，その場合はタイムアウトするまでの間待たされる．

そこで提案手法では，遅延行列を高速に求めるために，遠いプロセス対の RTT を他のプロセス対の RTT を基に見積もる．この際に各プロセスが従うアルゴリズムを**図 1** に記す（各プロセス p は 4 行目の obtain_rtts を自律的に実行する）．

この見積りアルゴリズムは，3 つのプロセス p，q，r の間の RTT について三角不等式

$$|rtt(p,r) - rtt(r,q)| < rtt(p,q) < rtt(p,r) + rtt(r,q)$$

が成立することが多いことに基づいている．具体的には，プロセス p が他のプロセス r と RTT 測定を行う際に，p にとって RTT が未知であるが r にとっては既知であるすべての q について，p は r から $rtt(r,q)$ を受信する（図 1 の 9 行目）．そして，測定した $rtt(p,r)$ と受信した $rtt(r,q)$ を用いて以下のように $rtt(p,q)$ を見積もる（図 1 の 10–16 行目）．

- $rtt(p,r) > \alpha rtt(r,q)$ の場合：$rtt(p,q)$ の見積りを $rtt(p,r)$ とする．
- $rtt(p,r) < \frac{1}{\alpha} rtt(r,q)$ の場合：$rtt(p,q)$ の見積りを $rtt(r,q)$ とする．
- それ以外の場合：$rtt(p,q)$ は直接 q と測定するか，r 以外のプロセスと測定を行う際に見積もる．

ここで，α は任意の値をとれるパラメータであるが，変化させることによってと見積りの正確さと測定に要する時間のバランスを調節することができる．これについては 5.2 節でより詳しく述べるが，本稿の評価実験では $\alpha = 5$ とする．

```
1:  // rtt(p,r): p と r の間の RTT
2:  // S_p: p にとって RTT が未知なプロセスの集合
3:
4:  obtain_rtts():
5:    S_p = { 全プロセス };
6:    while S_p ≠ φ:
7:      S_p から無作為に r を選択する;
8:      rtt(p,r) = measure_rtt(r);
9:      T = receive_rtts(r);
10:     foreach (q, rtt(r,q)) ∈ T:
11:       if rtt(p,r) > α rtt(r,q):
12:         rtt(p,q) = rtt(p,r);
13:         S_p = S_p − {q};
14:       else if rtt(p,r) < (1/α) rtt(r,q):
15:         rtt(p,q) = rtt(r,q);
16:         S_p = S_p − {q};
17:     S_p = S_p − {r};
18:
19: measure_rtt(r):
20:   rtt(p,r) を測定し，返す;
21:
22: receive_rtts(r):
23:   r から {(q, rtt(r,q)) | q ∈ S_p ∧ q ∉ S_r} を
24:   受信し，返す;
```

図 1 RTT 測定・見積りの際に各プロセス p が従うアルゴリズム
Fig. 1 Algorithm followed by each process p for RTT measurement and estimation.

2.2 通信行列

通信行列 T の各要素 t_{ij} は，実行するアプリケーションにおけるランク i とランク j の間の通信量を表す値である（$i, j = 0, 1, 2, \ldots, n-1$）．多くのアプリ

ケーションは実行を通して似た通信を繰り返すので，提案手法ではアプリケーションを短時間実際に実行して，その間にランク i からランク j に送信されたメッセージ数を t_{ij} とする．たとえば，反復法を用いたアプリケーションでは 1 イテレーションだけ実行してメッセージ数を数える．

3. 局所性を考慮した接続管理

3.1 概　　要

提案する接続管理では，全プロセス対で接続を確立するのではなく，一部のプロセス対でのみ接続を確立して，それらの接続を用いてルーティングレイヤを構築する．ルーティングを行うことによって，一部のプロセスの間の通信が遮断されていても，すべてのプロセスが互いに通信を行えるようになる．また，確立する接続の数を制限することによってスケーラビリティが向上する．たとえば，ステートフルファイアウォールや NAT サーバは確立された接続を記憶するが，記憶できる数には限りがある．また，TCP で高い帯域を実現するためにはバッファサイズを接続の帯域遅延積より大きくしなければならないため，多数の広域接続を確立するためには多くのメモリが必要である[15]．

提案する接続管理では，2 つの手法の組合せによって接続数を制限しても高い通信性能を保つ．1 つは確立する接続を局所性を考慮して選択することであり，これには遅延行列 D と通信行列 T を用いる．もう 1 つは接続を必要になったときに初めて確立すること（遅延接続確立）である．

遅延接続確立は MPICH[6] などの単一クラスタ用のメッセージパッシングシステムでも用いられてきた手法であり，各プロセスが少数のプロセスとしか通信を行わない場合は接続数が制限される．実際多くの科学技術計算アプリケーションでは各プロセスが少数のプロセスとしか通信を行わない．たとえば，NPB の BT，EP，LU，MG，SP では**表 1** に示すように通信を行うプロセス対は 6%以下である．しかし，他のアプリケーションには全対全で通信を行うようなものもあり，それらでは遅延接続確立を用いても全プロセス対で接続が確立されてしまう．そのようなアプリケーションの例としては NPB の IS（基数ソート），n 体問題，ランダムワークスティーリングに基づいた負荷分散があげられる．我々の提案する遅延接続確立では，事前に少数の接続を用いてルーティングテーブルを構築しておくことによって確立される接続の数に上限を設ける．また，通信が遮断されている可能性がある広域環境では単純な遅延接続確立はうまく動作しないが，我々の提案手法では事前に接続性を調べることによって遅延した接続確立が失敗することを防ぐ．

表 1 NPB で通信を行うプロセス対の数
Table 1 Process pairs that communicate in the NPB.

Benchmark	Process Pairs
BT (Block Tridiagonal)	5.5%
EP (Embarrassingly Parallel)	0.78%
IS (Integer Sort)	100%
LU (Lower Uppper)	4.6%
MG (Multi Grid)	5.4%
SP (Scalar Pentadiagonal)	5.5%

アプリケーション起動時（*MPI_Init*）の動作は以下のとおりである．

エンドポイントの交換　全プロセス間でエンドポイント（IP アドレスとポート番号）を交換する（3.2 節）．

バウンディンググラフの構築　各プロセスは D と T を基に少数の他のプロセスを選択し，それらにルーティング用の一時接続を確立することを試みる．確立が成功した全接続の集合ををバウンディンググラフと呼ぶ（3.3 節）．

ルーティングテーブルの構築　バウンディンググラフ上で全プロセス間の経路を計算する（3.4 節）．

スパニングツリーの構築　バウンディンググラフのスパニングツリーを計算し，スパニングツリーに属さない一時接続はすべて切断する（3.5 節）．

アプリケーションはスパニングツリーの接続のみ確立された状態で開始し，必要に応じてバウンディンググラフ上で隣接するプロセス間で本接続を確立する（3.6 節）．バウンディンググラフ上で隣接しないプロセス間で接続を確立することはない．また，スパニングツリーを用いてアプリケーショントラヒックをルーティングすることはなく，スパニングツリーは遅延接続確立を支援するためにのみ用いる．

3.2 エンドポイントの交換

エンドポイント交換には GXP[16] というグリッドシェルを用いる．GXP は複数のクラスタにまたがるノードにツリー上に SSH ログインを行うので，GXP からジョブを投入するとログインツリーを用いてエンドポイントを交換することができる．エンドポイントを交換する手段（e.g.，マスタプロセスへの接続）さえ提供されていれば，他のジョブ投入手法を用いることも可能である．

ファイアウォールなどによって通信が遮断されていることがあるため，これらのエンドポイントの中には到達不能なものある．しかし，バウンディンググラフを構築する際に接続性を調べるので，遅延接続確立は

到達可能なエンドポイントに対してのみ行うことができる．

3.3 バウンディンググラフの構築

バウンディンググラフを構築するために，各プロセス p は以下のように他のプロセスを選択し，それらに一時接続を確立することを試みる．ただし，n は全プロセスの数，β は接続密度を制御するパラメータとする．また，p 以外の $n-1$ 個のプロセスを d_{pq_i} の昇順に $q_1, q_2, \ldots, q_i, \ldots, q_{n-1}$ とする．

- $q_1, q_2, \ldots, q_{\beta-1}$ の $\beta-1$ 個のプロセスはすべて選択する．
- $q_{2^{j-1}\beta}, q_{2^{j-1}\beta+1}, \ldots, q_{2^j\beta-1}$ の $2^{j-1}\beta$ 個のプロセスから，β 個を選択する（$j = 1, 2, \ldots, \log_2 \frac{n}{\beta}$）．ただし，$q_k$ が選択される確率は
$$\frac{t_{pq_k}}{\sum_{l=2^{j-1}\beta}^{2^j\beta-1} t_{pq_l}}$$
に比例する（$k = 2^{j-1}\beta, 2^{j-1}\beta+1, \ldots 2^j\beta-1$）．

確立が成功した一時接続の集合がバウンディンググラフであるが，このように RTT に従って選択するプロセスを指数的に減少させることによって，近いプロセスとは密に接続を確立しつつ，全体の接続数を制限できる．また，t_{pq_k} が大きい q_k を高い確率で選択することによって，頻繁に通信することが予想されるプロセスの間で多数接続を確立することができる．

ここで，確立される接続の数の上限を求めるために，ファイアウォールなどによって通信が遮断されていることはなく，すべての一時接続は成功したとする．また，簡単のために，n 個のプロセスが c 個のクラスタに均等に分散されており，n は 2 のべき数であるとする．すると，以下のように各プロセスは $O(\log n)$ 本の接続を確立し，クラスタ間では合計 $O(n \log c)$ 本の接続が確立される．

- 各プロセスは $\beta\log_2 \frac{n}{\beta} + \beta - 1$ 本の接続を確立する．
- これらの接続のうち $\beta\log_2 c$ 本がクラスタをまたぐので，クラスタ間ではすべてのプロセスによって合計 $\beta n\log_2 c$ 本の接続が確立される．

ファイアウォールなどによって通信が遮断されている場合はこれより少ない数の接続が確立される．β の値および通信が遮断されている場所によってはバウンディンググラフが非連結になる可能性があるが，5.3 節で述べるように実用的には十分高い確率で連結になる．

3.4 ルーティングテーブルの構築

バウンディンググラフを構築した後，RTT をメトリックとした距離ベクトル型ルーティングを行い，バウンディンググラフの枝を用いた全プロセス間の最短経路を求める．ルーティングを行う際には，各接続がどのプロセスによって確立されたかは考慮しない（TCP 接続は確立された後は双方向に利用可能である）．ただし，遅延接続確立の際には接続が確立できる方向が重要なので，バウンディンググラフ構築時にどちらの方向に一時接続が確立されたかは記憶しておく．

3.5 スパニングツリーの構築

ルーティングテーブルを構築した後，一時接続は制御メッセージ送受信用のスパニングツリーを除いて切断する．ルーティングテーブルを用いればバウンディンググラフからスパニングツリーを生成するのは容易であり，ある 1 つのプロセスを根とすれば，そのプロセスから他のすべてのプロセスへの経路に含まれる枝の集合がスパニングツリーになる．また，バウンディンググラフの場合と同様に，RTT をメトリックとした距離ベクトル型ルーティングを行い，スパニングツリーの枝を用いた全プロセス間の最短経路を求める．

3.6 遅延接続確立とメッセージの中継

アプリケーショントラヒックをルーティングする本接続は，メッセージを送信する際にオンデマンドに確立する．この際に用いるアルゴリズムを**図 2** と**図 3** に示すが，基本的な動作は次のとおりである．プロセス p がプロセス q にメッセージ msg を送信する場合，最初に p が forward_app_message(p, q, msg) を実行し（図 2 の 5 行目），それ以降 p から q への経路上の各プロセスが順に forward_app_message を実行す

```
1:  // BG: バウンディンググラフ
2:  // ST: スパニングツリー
3:  // req: 接続確立を要求するメッセージ
4:
5:  forward_app_msg(src, dst, msg):
6:    next = get_next_hop(BG, dst);
7:    if not connected(next):
8:      if connectable(next):
9:        next に向けて本接続を確立する;
10:     else:
11:       next2 = get_next_hop(ST, next)
12:       send(ST, self, next, next2, req);
13:       next から本接続が確立されるのを待つ;
14:   send(BG, src, dst, next, msg);
15:
16: get_next_hop(G, dst):
17:   G 上の dst へのネクストホップを返す;
18:
19: connected(next):
20:   next との本接続がすでに確立されている場合
21:   TRUE を，それ以外の場合 FALSE を返す;
22:
```

図 2 オンデマンドに接続を確立しながらメッセージを転送するためのアルゴリズム（前半）
Fig. 2 Algorithm for forwarding messages while establishing connections on demand (first half).

```
23: connectable(next):
24:     next との一時接続を確立したのが self である
25:     場合 TRUE を，それ以外の場合 FALSE を
26:     返す.
27:
28: send(G, src, dst, next, msg):
29:     msg を G の接続を用いて next へ転送する;
30:     next は msg を受信すると，
31:     handle_msg(G, src, dst, msg) を実行する;
32:
33: handle_msg(G, src, dst, msg):
34:     if G == BG:
35:         if dst == self:
36:             msg をアプリケーションに届ける;
37:         else:
38:             forward_app_msg(src, dst, msg);
39:     else: // G == ST
40:         if dst == self:
41:             connect(src);
42:         else:
43:             next2 = get_next_hop(ST, dst);
44:             send(ST, src, dst, next2, req);
```

図 3 オンデマンドに接続を確立しながらメッセージを転送するためのアルゴリズム（後半）

Fig. 3 Algorithm for forwarding messages while establishing connections on demand (second half).

ることによって（図 3 の 38 行目），*msg* は再帰的に *q* に届けるられる．forward_app_message を実行するプロセスは，ネクストホップ *next* との本接続がすでに確立されている場合はただちに *msg* を転送することができるが（図 2 の 14 行目），まだ確立されていない場合は確立してから（図 2 の 8–13 行目）*msg* を転送する．

本接続を確立する際，バウンディンググラフ構築時に *next* との一時接続を確立したのが *self* である場合は，本接続も *self* から *next* に向けて確立することができる（図 2 の 9 行目）．しかし，一時接続を確立したのが *next* である場合は，*self* から *next* に向けて確立できるかは分からない．そこで，スパニングツリーを用いて *self* から *next* にメッセージを送信して，*next* から *self* に向けて本接続を確立してもらう（図 2 の 11–13 行目）．

通信を行うプロセス対が多い場合はバウンディンググラフのすべての枝に沿って本接続が確立されるが，少ない場合は実際に使われる接続のみ確立される．ルーティングテーブル構築時にはバウンディンググラフのすべての枝に沿って一時接続が確立されるが，それらは本接続のように大きな TCP バッファを必要としない．

また，メッセージ中継時に小さいメッセージは図 3 の 38 行目のようにメッセージ全体を受信してから転送するが，大きいメッセージはセグメント単位で受信・転送を繰り返すことによってスループットを向上させる．

4. 局所性を考慮したランク割当て

4.1 概　　要

よく用いられるランク割当て手法に，プロセスをホスト名もしくは IP アドレスに基づいてソートして，その順にランクを割り当てるというものがある．この手法は，通信の多くはランクが近いプロセスの間で行われ，ホスト名や IP アドレスが近いプロセスの間の通信コストは低いという仮定に基づいている．しかし，実際にはアプリケーションの通信パターンには様々なものがある．また，WAN では IP アドレスと通信コストの間に深い関係があることはあまり期待できない．Virtual LAN（VLAN）や Virtual Private Network（VPN）などの仮想化技術が用いられている場合は，同じ LAN に存在するプロセスの間の通信コストが低いかすら分からない．

提案手法ではこのような単純な仮定はせずに，通信オーバヘッドを最小化するという問題をより直接的に扱う．そのためにプロファイリング実行で得た通信行列 T と遅延行列 D を用いてランク割当て問題を二次割当て問題として表現する．

4.2 二次割当て問題による表現

提案するランク割当てでは，各プロセス対の通信オーバヘッドを通信量と通信コストの積とし，アプリケーション全体の通信オーバヘッドを全プロセス対の通信オーバヘッドの和とする．また，通信量と通信コストを以下のように定義する．

- 通信量：T の要素 t_{ij}
- 通信コスト：3.4 節で求めた経路に沿って D の要素を加えたもの（δ_{ij} とする）

すると，通信オーバヘッドを最小化するランク割当てを見つけるという問題は，

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} t_{ij}\delta_{p(i)p(j)}$$

を最小化する $\{0, 1, 2, \ldots, n-1\}$ の並べ替え p を求めるという問題になる．この最適化問題は二次割当て問題として知られており，Koopmans らによって最初に紹介されたものである[11]．

遅延に基づく通信コストを用いる理由は以下のとおりである．2 つのプロセスの RTT とその 2 つのプロセスの間にあるリンクの数には深い関係がある．また，2 つのプロセスの間にあるリンクの数が多いほど，その 2 つプロセスの通信と他のプロセス対の通信が衝突する可能性は高い．したがって，通信量の多いランク

対を RTT の短いプロセス対に割り当てることによって，遅延だけでなく帯域についても最適化することになる．

帯域に基づく通信コストを用いることも考えられるが，2 点間の帯域では通信の衝突がモデル化されないため，2 点間の遅延を用いる場合ほどうまくいかない．

4.3 二次割当て問題の解法

二次割当て問題は NP 困難であり，n が大きな問題を厳密に解くことはできないが，ヒューリスティクスを用いることによって短時間で良い近似解を得ることができる[2),4)]．提案手法では Resende らによって開発された GRASP（Greedy Randomized Adaptive Search Procedure）という手法に基づくライブラリ[13)] を用いることによって二次割当て問題の近似解を得る．公開されている二次割当て問題のセットである QAPLIB[3)] を用いてこのライブラリの性能を調べたところ，256 以下の n に対して 1 秒以内に既知の最良解の 1–2%以内の近似解を得ることができた．ただし，問題サイズが n のときは二次割当て問題の計算に用いることができるノードも n あるので，n 並列に計算を行う☆．

5. 評価実験

5.1 概　　要

提案する 2 手法を用いて Multi-Cluster MPI（MC-MPI）というメッセージパッシングライブラリを実装し，**図 4** に示すような 4 クラスタにまたがる 256 実ノードを用いて評価した．各クラスタの隣の値はクラスタ内の RTT とバンド幅を示し，クラスタを結ぶ線の隣の値はクラスタ間の RTT とバンド幅を示す．また，クラスタを結ぶ線の矢印は許可されていた通信の方向を示す．4 クラスタのうち 1 つはファイウォールによってフィルタリングが行われており，外向きに接続を確立することは許可されていたが，内向きに接続を確立することは禁止されていた．また，各ノードの CPU は 2.4 GHz の Xeon もしくは 1.86 GHz の Pentium M であり，OS は Linux であった．TCP の送信バッファ・受信バッファは，クラスタ A とクラスタ C の間の帯域遅延積にほぼ等しい 256 KB にした☆☆．

5.4 節および 5.5 節の実験では NAS Parallel Benchmarks（NPB3.2-MPI）を用いた．用いた 6 つのベンチマークのうち BT，EP，LU，MG，SP はクラス D を用いたが，IS はクラス D が存在しないためクラス C を用いた．

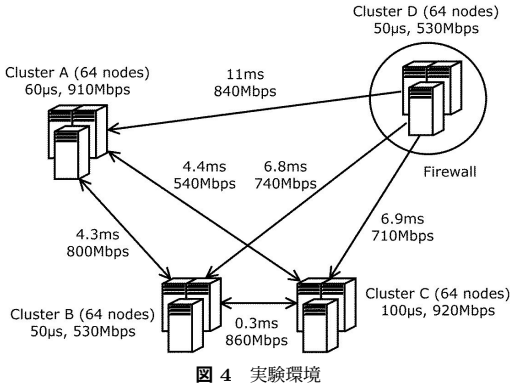

図 4 実験環境
Fig. 4 Experimental environment.

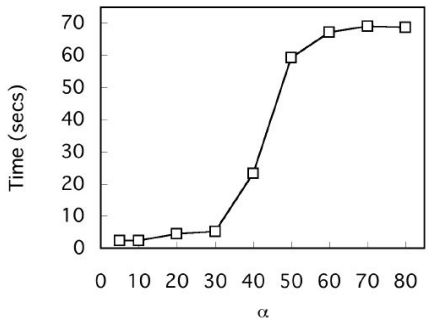

図 5 様々な α を用いたときに遅延行列を取得するのに要した時間
Fig. 5 Time required to obtain the traffic matrix with various values of α.

また，提案手法を用いると本実行に加えてプロファイリング実行にも時間がかかるが，長時間にわたって実行するアプリケーションにおいてはプロファイリング実行が占める割合は小さい．たとえば BT の場合，本実行にかかる時間は 1,700 秒程度であるが，RTT 測定・見積りにかかる時間はその 0.13%の 2.2 秒である．また，BT は 250 イテレーションからなるので，メッセージ数を数えるための 1 イテレーションのオーバヘッドは 0.4%である．

5.2 遅延行列の取得時間

本節の実験では，様々な α を用いたときに遅延行列を取得するのに要する時間を測定した．**図 5** に結果を示す．α が大きい場合は異なるクラスタのノードの間で測定が数多く試みられ，その一部がファイアウォールによって遮断されていてタイムアウト待ちが多発したため，遅延行列を求めるのには長時間要した．本章の評価実験ではこのようなタイムアウト待ちを避けるために，$\alpha = 5$ とした．$\alpha = 5$ の場合，最大 25%の見積り誤差があるが，クラスタ内の RTT とクラスタ間の RTT にはこれより大きな差があるため，提案す

☆ GRASP は容易に並列化できる手法である．
☆☆ Linux は内部で同じだけのメモリを用いるため，実際に消費されるメモリ量は 512 KB である．

る接続管理とランク割当てには十分な正確さである．

5.3 バウンディンググラフの非連結確率

次に，一部の通信が遮断されていてもバウンディンググラフが十分高い確率で連結になるということを示すために，シミューレションを行った．このシミュレーションでは，下記の 2 つの環境でバウンディンググラフが非連結になる確率を求めた．

- *1FW*：図 4 の実験環境の RTT を用いるシミュレーション環境．ファイアウォールも実際の実験環境と同様に，クラスタ D の周りに内向きの通信を遮断するものがある．
- *3FW*：RTT は *1FW* と同様に図 4 の実験環境のものを用いるが，通信が遮断されている場所は実際の実験環境より多いシミュレーション環境．クラスタ A 以外の 3 個のクラスタの周りにファイアウォールがある（クラスタ A の周りにもあると，クラスタどうしを連結にすることが不可能になってしまう）．

これらの環境は 256 ノードからなるが，プロセス数の影響も調査するために，$8, 16, 32, \ldots, 256$ ノードでプロセスを立ち上げる場合のそれぞれについてシミュレーションを行った（各クラスタでは同数のノードを用いた）．

各プロセス数において 10^8 回バウンディンググラフを作成したところ，*1FW* では $\beta = 1$（最もバウンディンググラフが非連結になる確率が高い値）としても，バウンディンググラフが非連結になることはなかった．一方，より多くの通信が遮断されていた *3FW* では，**図 6** に示すようにバウンディンググラフが非連結になることがあった．しかし，*3FW* でもプロセス数が増加すると非連結になる確率は下がり，$\beta = 1$ の場合でも 128 プロセス以上では非連結になることはなかった．また，$\beta = 2$ とすると非連結になる確率はさらに下がり，$\beta = 4$ とするとどのプロセス数でもバウンディンググラフが非連結になることはなかった．

5.4 接続管理の性能

次の実験では，提案する接続管理を評価するために，確立する接続の数を変化させたときの NPB の性能を測定した．提案手法を用いた場合（*MC-MPI*）に加えて，バウンディンググラフ構築時に遅延・通信量を考慮せずにランダムにプロセスを選択する手法（*Random*）を用いた場合についても測定を行った．*MC-MPI* では **表 2** のように β を選択することによって接続数を制御し，*Random* では同数の接続をランダムに選択した．

図 7 に結果を示す．横軸は各プロセスが確立する接続のパーセンテージの上限を表す（遅延接続確立を用いるため，実際に確立される接続の数は横軸に表示されている値より少ない）．縦軸は全プロセス対で接続を確立した場合の性能に対する相対性能を表す．

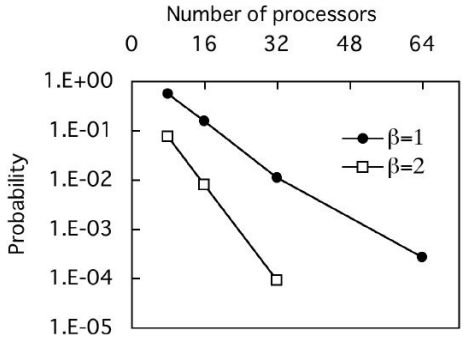

図 6 バウンディンググラフが非連結になる確率
Fig. 6 Probability of the bounding graph being disconnected.

表 2 様々な β によって選択される接続のパーセンテージ（$n = 256$）
Table 2 Percentage of connections selected with various values of β ($n = 256$).

β	Percentage	β	Percentage
2	12.2%	21	60.6%
4	20.1%	28	69.4%
7	30.9%	38	79.7%
10	39.1%	55	89.9%
15	50.7%	128	100%

BT，LU，MG，SP の 4 ベンチマークは，*Random* を用いた場合は接続数を制限すると中継されるメッセージが多数発生して，性能が大きく低下した．一方，*MC-MPI* を用いた場合は 10%のプロセス対でしか接続を確立しなくても性能はまったく落ちなかった．これらのベンチマークでは各プロセスが少数のプロセスとしか通信しないため，確立する接続を適切に選択することによって少数の接続でもすべてのメッセージが直接届けられるようにできたためである．たとえば，LU は Successive Over-Relaxation（SOR）を行うベンチマークなので，各プロセスは主に 4 プロセスとしか通信を行わない．

EP はほとんど通信をともなわないベンチマークであるため，*MC-MPI* を用いた場合も *Random* を用いた場合も接続数によらず性能は変わらなかった．

IS は *MC-MPI* を用いた場合も *Random* を用いた場合も，接続を制限した方が高い性能が出た．これは IS が全対全通信をともなうベンチマークであり，多数の接続を用いて全対全通信を行うと輻輳が生じてしまうためである．全対全通信では多数のノードが同時

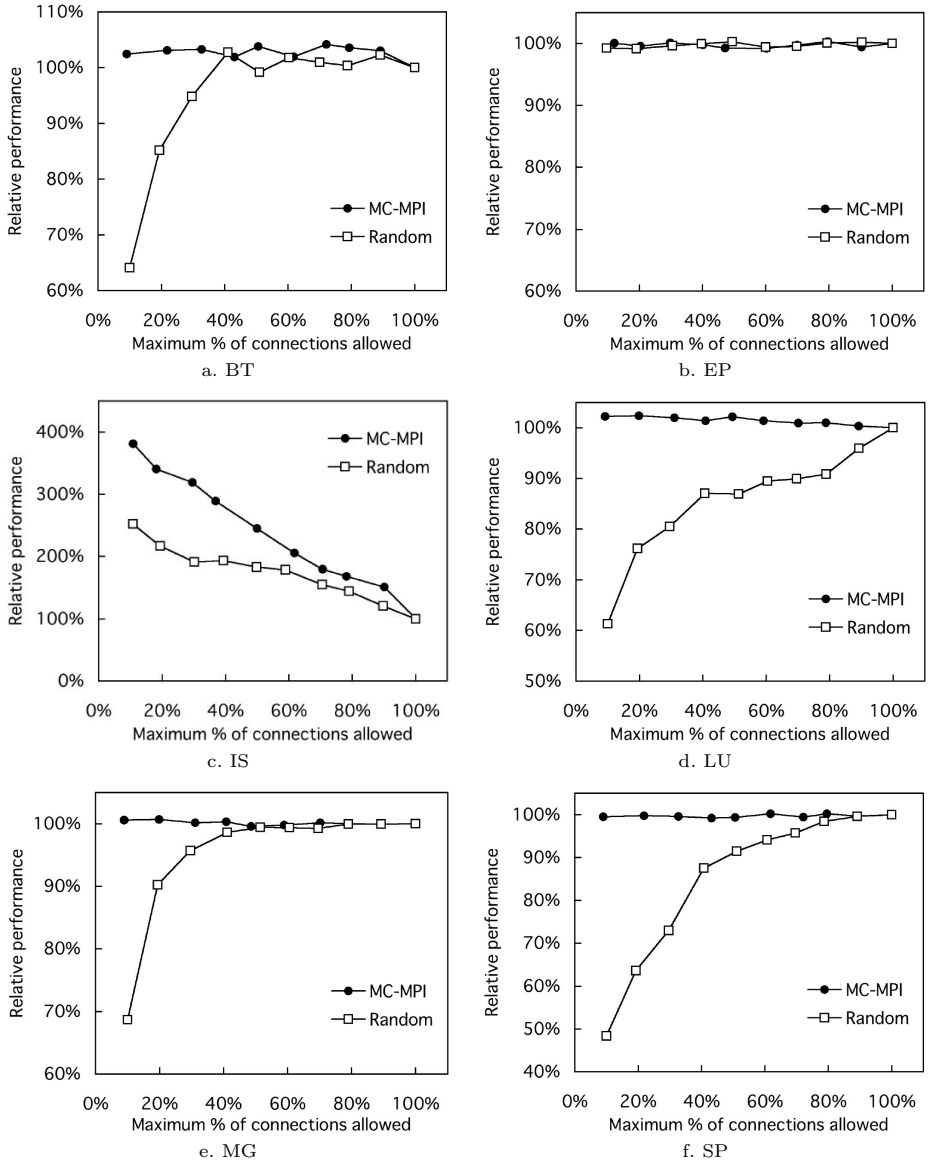

図 7 接続を制限したときの NPB の性能
Fig. 7 Performance of the NPB when the number of connections was limited.

に通信を行うため，各ノードが利用できる帯域は限られている．しかし，TCP は各ノードで独立に動作するため，全対全通信の開始時には他のノードの通信を考慮せずに高速にパケットを送出してしまう．その結果，接続を多数張っている場合はクラスタ間で輻輳が生じてしまう．一方，接続を少数しか張っていない場合は，アプリケーションレイヤで全対全通信を行っても TCP レイヤでは少数のノードしか通信しないため，輻輳は生じない．

多数の接続を用いて全対全通信を行うと輻輳が生じてしまう様子を確認するために，接続数およびバッファサイズを変更したときの IS の性能を測定した．**図 8** に結果を示す．横軸は TCP の送信バッファ・受信バッファの大きさを表す．縦軸は図 7 と同様に，バッファサイズを 256 KB にして全プロセス対で接続を確立した場合に対する相対性能を表す．また，接続数に関しては，20%，60%，100%のプロセス対で接続を確立した場合について結果を示す．100%の場合は，バッファサイズを 32 KB 以上にするとパケットが高速に送出できてしまうため輻輳が生じて性能が大きく低下した．一方，60%の場合は性能低下はよりなだらかであり，20%の場合は性能低下はまったくなかった．

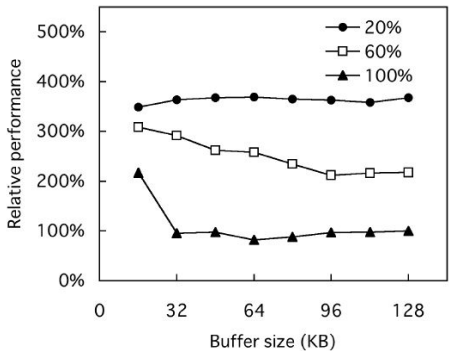

図 8 接続数およびバッファサイズを変更したときの IS の性能
Fig. 8 Performance of IS with varying numbers of connections and buffer sizes.

また，バウンディンググラフを用いた遅延接続確立を評価するために，提案手法（*MC-MPI*）を用いた場合と MPICH のような接続管理手法（*MPICH-like*）を用いた場合を比較した．*MC-MPI* では各プロセスが確立する接続の数が 30%に制限されるように β を選択した．*MPICH-like* ではバウンディンググラフを用いずに遅延接続確立を行った（バウンディンググラフ構築時に各プロセスが他のプロセスを 100%選択したと考えてもよい）．*MPICH-like* は全対全で通信が行えないと動作しないので，この実験のときのみクラスタ B のファイアウォールを無効にした．

図 9 に結果を示す．IS 以外のベンチマークでは *MC-MPI* を用いた場合も *MPICH-like* を用いた場合も同様の数の接続が確立され，性能も同等であった．通信パターンが全対全である IS では，*MPICH-like* を用いた場合はすべての接続が確立されてしまったが，*MC-MPI* を用いた場合は 30%の接続しか確立されなかった．その結果，*MC-MPI* は *MPICH-like* に対して 180 MB メモリを節約できた．また，前述したように接続を制限すると輻輳が生じることを回避できるため，性能も *MC-MPI* の方が *MPICH-like* より良かった．

5.5 ランク割当ての性能

次に，提案するランク割当てを評価するために，様々なランク割当てを用いたときの NPB の性能を測定した．

図 10 に結果を示す．ここで，*Random* はランダムな割当て，*MC-MPI* は提案する D と T に基づく割当て（4.2 節と 4.3 節），*Hostname* はホスト名に基づく割当て（4.1 節）である．また，*Hostname* (*Best*) と *Hostname* (*Worst*) は，クラスタ単位でホスト名を入れ替えてホスト名に基づく割当てを行ったときに，24 通りの割当ての中で最も性能が良かったものと最も性能が悪かったものである．

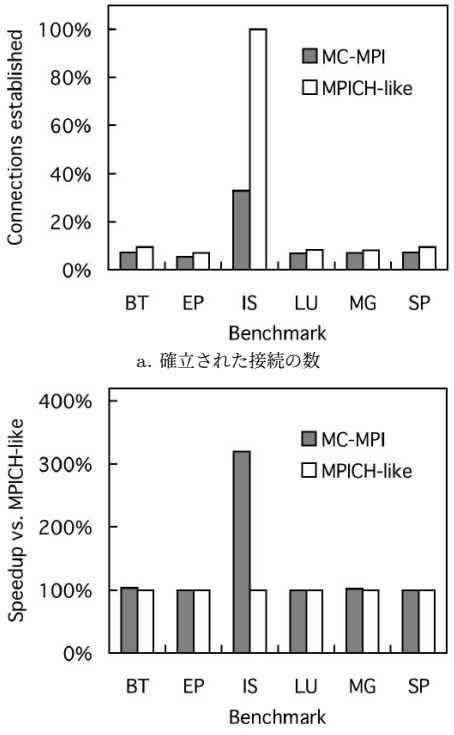

図 9 遅延接続確立手法の比較（*MC-MPI* vs. *MPICH-like*）
Fig. 9 Comparison of lazy connection establishment methods (*MC-MPI* vs. *MPICH-like*).

LU ではホスト名に基づく割当て（*Hostname*, *Hostname* (*Best*), *Hostname* (*Worst*)）を用いた場合は *Random* を用いた場合より 110%から 200%高い性能が出た．これは LU がランクが近いプロセス間の通信が多いベンチマークであり，ホスト名に基づく割当てが同じクラスタのプロセスに近いランクを割り当てるためである．ホスト名に基づく割当ての中でも近いクラスタに近いランクを割り当てたかによって性能に差が生じているが，*MC-MPI* は *Hostname* (*Best*) とほぼ同じ性能を出すことができた．MG でも同様の結果が得られた．このようにクラスタの数が 4 つでも良い割当てを手動で見つけるのは困難であり，クラスタの数が多くなると *MC-MPI* のような適応的な手法はさらに有用になる．

BT では *MC-MPI* を用いた場合は *Hostname* (*Best*) を用いた場合よりさらに 20%高い性能が出たが，これは BT が遠いランク間でも頻繁に通信を行うためである．たとえば，ランク 0 が主に通信を行うランクは 1，15，16，31，240，241 である．SP でも同様の理由で *MC-MPI* を用いた場合の方が *Hostname* (*Best*) を用いた場合より 10%高い性能が出た．

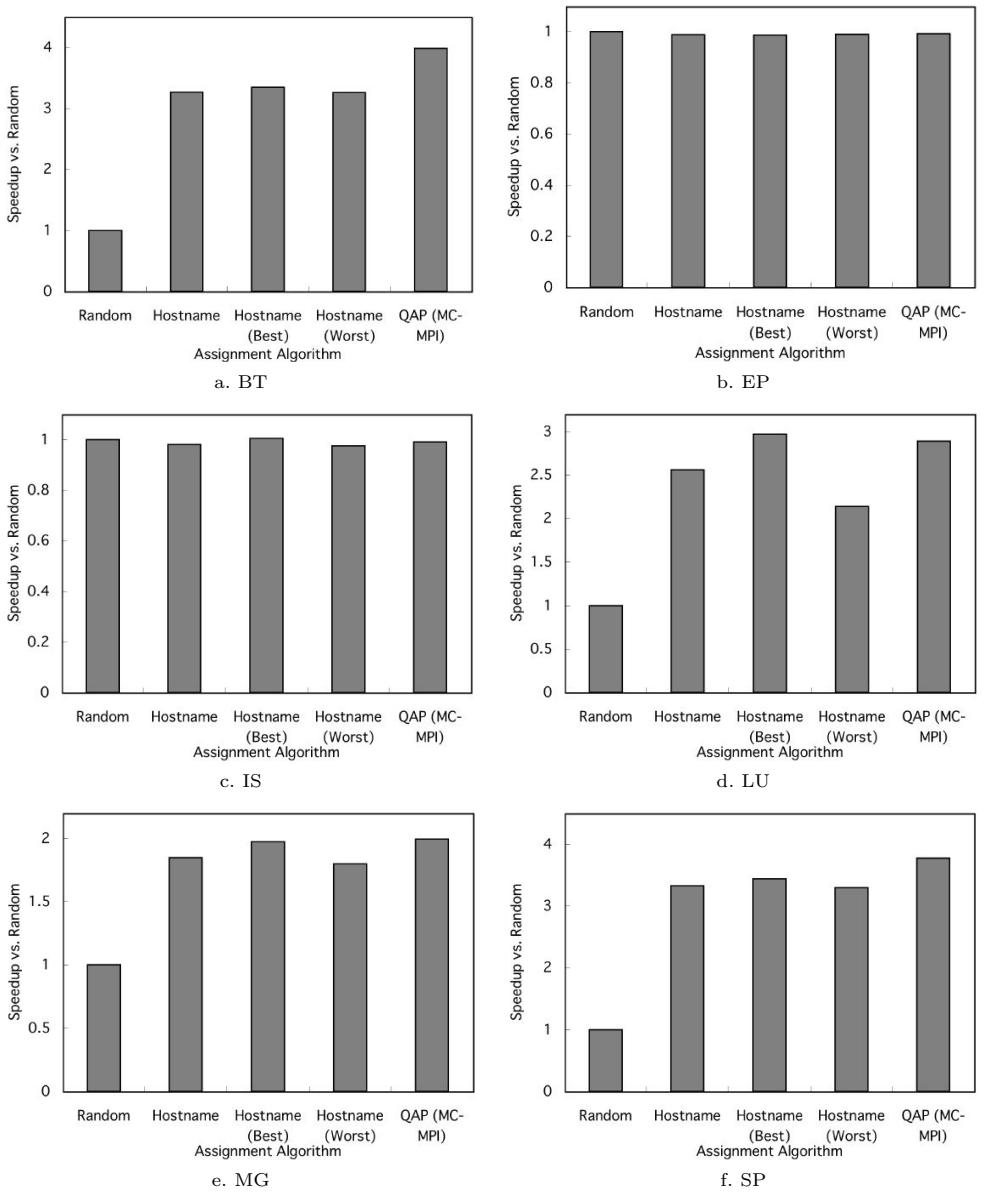

図 10 様々なランク割当てを用いたときの NPB の性能
Fig. 10 Performance of the NPB with various rank assignments.

EP は通信量が少ないため，IS は通信パターンが一様であるため，どの割当てを用いても性能はほぼ同じであった．

6. 関 連 研 究

6.1 広域環境用のメッセージパッシングシステム

これまでに広域環境用のメッセージパッシングシステムが数多く提案されており，それらはネットワーク性能を考慮して広域環境で高い性能を出す．たとえば，MPICH-G2[9]，GridMPI[12]，MagPIe[10] は広域環境を意識した集合通信を提供する．しかし，これらはプロセス間で全対全の接続性を必要とするため，ファイアウォールやプライベートアドレスのある環境では動作しない．これに対して MC-MPI は全対全の接続性を必要とせず，ファイアウォールやプライベートアドレスのある環境でも動作する．

6.2 中継を行うメッセージパッシングシステム

メッセージを中継することによってファイアウォールやプライベートアドレスがある環境でも全プロセスが互いに通信できるようにするメッセージパッシ

ングシステムもいくつか提案されている．たとえば，MPICH/MADIII[1] と StaMPI[8] は手動で与えられた接続性に関する情報を用いて接続を確立したりルーティングを行ったりする．しかし，このような手動の設定を用いる手法は計算環境の規模と複雑さが増すにつれて設定量も多くなってしまう．MC-MPI は接続性にを自動的に発見して手動の設定は必要ないという点で MPICH/MADIII や StaMPI と異なる．

6.3 適応的なメッセージパッシングシステム

AMPI は仮想プロセッサを用いる適応的なメッセージパッシングシステムである[7]．AMPI は負荷分散を行うために，実プロセッサの実行時間が均等になるようにしつつ，実プロセッサ間の通信量が小さくなるように仮想プロセッサをマイグレートさせる．この手法はアプリケーションの通信パターンに基づいて適応的に性能最適化を行うという点では我々のランク割当てと似ているが，通信コストの扱いが我々の手法と異なる．AMPI は単一のクラスタ用に設計されているため，すべてのプロセッサ対の通信コストが同じだと仮定している．一方，MC-MPI は複数クラスタに対応するために，プロセス対によって通信コストが異なることを考慮する．

7. おわりに

局所性を考慮した接続管理とランク割当てを提案して，それらを用いて MC-MPI という広域環境用の MPI ライブラリを実装した．4 クラスタ 256 ノードで NPB を実行して評価を行ったところ，主な結果として以下のものが得られた．

- 10%のプロセス対でしか接続を確立しなくても，全プロセス対で接続を確立した場合と同様もしくはそれ以上の性能が出た．全対全通信を行う IS では接続を制限することによって輻輳を回避することができ，制限した方が性能が高くなった．
- 局所性を考慮しない割当てを用いた場合より最大 300%高い性能が出るランク割当てを見つけることができた．遠いランク間でも頻繁に通信が行われる BT と SP ではホスト名に基づく割当てよりもさらに 10%から 20%高い性能が出る割当てを見つけることができた．

現在はプロファイリング実行と本実行の 2 回に分けてアプリケーションを実行するが，今後は本実行の中でプロファイリングを行うための API を用意したい．また，現在 MC-MPI は 2 点間通信についてのみ局所性を考慮した性能最適化を行っているが，今後は集合通信も局所性を考慮したものを組み込んでいきたい．RTT を基に計算環境に適応するという手法は集合通信についても有効であり，実際我々は文献 14) でそのような手法を提案している．

謝辞 本研究の一部は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する新 IT 基盤研究プラットフォームの構築」の助成を得て行われた．

参考文献

1) Aumage, O. and Mercier, G.: MPICH/MADIII: A Cluster of Clusters Enabled MPI Implementation, *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp.26–33 (2003).
2) Bazarra, M.S. and Sherali, H.D.: On the Use of Exact and Heuristic Cutting Plane Methods for the Quadratic Assignment Problem, *Journal of the Operational Research Society*, Vol.33, pp.991–1003 (1982).
3) Burkard, R.E., Karisch, S.E. and Rendl, F.: QAPLIB — A Quadratic Assignment Problem Library, *Journal of Global Optimization*, Vol.10, pp.391–403 (1997).
4) Burkard, R.E. and Rendl, F.: A Thermodynamically Motivated Simulation Procedure for Combinatorial Optimization Problems, *European Journal of Operational Research*, Vol.17, pp.169–174 (1984).
5) der Wijngaart, R.F.V.: NAS Parallel Benchmarks Version 2.4, NAS Technical Report NAS-02-007, NASA Ames Research Center (2002).
6) 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.
7) Huang, C., Zheng, G., Kumar, S. and Kale, L.V.: Performance Evaluation of Adaptive MPI, *Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.12–21 (2006).
8) Imamura, T., Tsujita, Y., Koide, K. and Takemiya, H.: An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers, *Proc. 7th European PVM/MPI Users' Group Meeting*, pp.200–207 (2000).
9) Karonis, N.T., Toonen, B. and Foster, I.: MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, *Journal of Parallel and Distributed Computing*, Vol.63, No.5, pp.551–563 (2003).
10) Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A. and Bhoedjang, R.A.F.: MagPIe:

MPI's Collective Communication Operations for Clustered Wide Area Systems, *Proc. 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.131–140 (1999).
11) Koopmans, T.C. and Beckman, M.J.: Assignment Problems and the Location of Economic Activities, *Econometrica*, Vol.25, pp.53–76 (1957).
12) Matsuda, M., Kudoh, T., Kodama, Y., Takano, R. and Ishikawa, Y.: TCP Adaptation for MPI on Long-and-Fat Networks, *Proc. IEEE International Conference on Cluster Computing* (*Cluster 2005*) (2005).
13) Resende, M.G.C. and Pardalos, P.M.: Algorithm 754: Fortran Subroutines for Approximate Solution of Dense Quadratic Assignment Problems Using GRASP, *ACM Trans. Mathematical Software*, Vol.22, No.1, pp.104–118 (1996).
14) Saito, H., Taura, K. and Chikayama, T.: Collective Operations for Wide-Area Message Passing Systems Using Adaptive Spanning Trees, *Proc. 6th IEEE/ACM International Workshop on Grid Computing*, pp.40–48 (2005).
15) Snader, J.C.: *Effective TCP/IP Programming: 44 Tips to Improve Your Network Programs*, Addison Wesley (2000).
16) Taura, K.: GXP: An Interactive Shell for the Grid Environment, *Proc. 8th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp.59–67 (2005).

(平成 19 年 5 月 7 日受付)
(平成 19 年 9 月 28 日採録)

斎藤 秀雄（学生会員）
1981 年生．2006 年東京大学大学院情報理工学系研究科電子情報学専攻修士課程修了．同年より同博士課程在学中．ACM 会員．

田浦健次朗（正会員）
1969 年生．1997 年東京大学大学院理学博士（情報科学専攻）．1996 年より東京大学大学院理学系研究科情報科学専攻助手．2001 年より東京大学大学院情報理工学系研究科電子情報学専攻講師．2002 年より同助教授．2007 年より同准教授．日本ソフトウェア科学会，ACM，IEEE-CS 各会員．