ハイパフォーマンス
コンピューティング 65－13
（1997．3．6）

Collective 通信を用いたデータ並列プログラムの性能予測

田中良夫[†] 久保田和人[†]
佐藤三久[†] 関口智嗣[††]

データ並列のプログラムは，プロセッサ内での局所計算と他のプロセッサとの通信の2種類の処理により構成される．また，プロセッサ間の通信は1対1通信ではなく，broadcast や reduction のような collective 通信により行なわれる．我々は様々な並列計算機上で各種の collective 通信の性能を測定してきた．本稿では，それらの collective 通信の基本性能を用いたデータ並列プログラムの性能予測に関して，その方法および実験結果を報告する．本手法により，プログラムと並列計算機との適合性を事前に知ることができる．

Performance Prediction of Data-Parallel Programs Using Collective Communication Performance

Yoshio Tanaka ,[†] Kazuto Kubota ,[†] Mitsuhisa Sato [†]
and Satoshi Sekiguchi[††]

Data-parallel programs are constructed by two parts: *local computation* and *interprocessor communication*. The communications are not point-to-point communications but collective communications such as *broadcast* and *reduction*. In our previous works, we measured the performance of collective communications on various parallel systems. In this paper, we propose the performance prediction method of data-parallel programs using collective communication performance and report the experimental results.

1. はじめに

データ並列のプログラムは，プロセッサ内部で行なわれる局所計算と，他のプロセッサとの通信により構成される．また，その通信は1対1通信ではなく，*broadcast* や *reduction* のような全プロセッサが通信に参加する **collective 通信**により行なわれる．Collective 通信の性能は並列計算機を特徴づけるデータを与え，応用プログラムの性能に大きな影響を与える．我々は今まで行なってきた性能評価研究[2]の一環として，いくつかの並列計算機上で collective 通信の性能を測定した[4),5)]．それらのデータはデータベースとして保存され公開されており☆，システムの性能評価などに利用できるようになっている．

本稿では，これらのデータの利用法の1つとして，データ並列プログラムの性能解析を行なう方法を提案する．プログラムの性能を正確に理解することにより，ネットワークの性能がプログラム全体に対してどのように寄与しているかなど，プログラムと並列システムとの適合性を事前に知ることができる．その結果を

- 新しいシステムの選択，改良の際の手がかり
- プロセッサ数を変更したときの挙動の予測（スケーラビリティ）
- ボトルネックの発見

などに利用することができる．

我々はソースプログラムを解析し，自動的にプログラムの性能解析を行なうようなツールの開発を目指している．並列プログラムは逐次的な（各プロセッサ内部で処理する）計算とプロセッサ間で行なわれる通信から構成される．逐次的な部分の性能はプロセッサの性能に依存しており，第1次的な近似としては実行サイクルと実行命令数などから予測することができる．通信部分の性能は，要素プロセッサ自体の性能に加えて，プロセッサ数やそれらを結合するネットワークの形態や性能，インタフェースなど様々な要素が影響を与えるため，その解析は困難である．そこで我々は通信部分の性能解析に collective 通信の性能データベースを利用してデータ並列プログラムの性能解析を行なう方法を提案する．データ並列プログラムは単純に局所計算部分と通信部分に分けてモデリングすることができる．本稿は単純にモデリングを行なって性能解析

† 新情報処理開発機構
Real World Computing Partnership
†† 電子技術総合研究所
Electrotechnical Laboratory
☆ http://www.rwcp.or.jp/lab/mpperf

を行なう方法を提案し，いくつかの事例を用いてその妥当性を示す．

次節では性能解析の方法に関して詳しく説明する．3節では性能解析の具体例として，行列の乗算，CG法による連立1次方程式の解法およびradixソートのプログラムに関して性能解析を行ない，実際に実行した時間との比較を行なった結果に関して報告する．4節では考察を行ない，最後に結論と課題を述べる．

2. 性能モデリングと性能予測

我々はデータ並列プログラムの通信はほとんどがcollective通信であることに着目し，データベースに蓄えられているcollective通信の基本性能データをもとにデータ並列プログラムの性能解析を行なう方法を提案する．データ並列のプログラムは局所計算と通信部分に分けることが容易であり，また各ノードのロードバランスも均一であるため，その性能解析は単純な方法で行なうことができる．たとえばCG法を用いた連立1次方程式の解法の場合，局所計算としては行列とベクタの乗算，ベクタの内積およびベクタの加算が行なわれ，通信としてはベクタの *complete exchange* および *reduction* が行なわれる．我々はデータ並列のプログラムを局所計算と通信部分とに分けるというモデル化を行ない，そのモデルを用いて局所計算と通信のそれぞれにかかる時間を別々に見積もり，プログラムの性能予測を行なう．モデル化および性能予測の手順は以下のようになる．

(1) **プログラムのモデル化**
プログラムを局所計算部分と通信部分とに分割する．典型的なデータ並列のプログラムは以下のような仮想コードにより記述できる．

```
while ( condition ) {
    LocalComputation();
    Communication();
}
```

LocalComputation() は局所計算を行なう部分であり，*Communication()* が通信を行なう部分である．

(2) **局所計算部分の時間の見積もり**
局所計算部分の時間を見積もるためには，*LocalComputation()* にかかる時間を見積もれば良い．この方法としては命令数を数えるなどの方法が考えれるが，正確に見積もるためには条件分岐の扱いやキャッシュのヒット/ミスによってメモリ参照にかかる時間が変わってくるなどの，様々な問題をクリアする必要がある．今回は局所計算部分の時間の見積もりに関してはプログラムコードより時間を見積もるのではなく，実際に局所計算の処理部分だけを行なうプログラムを実行することにより時間を見積もるという方法を採用した．具体的には *LocalComputation()* のみからなるプログラムを実行し，その時間を計測する．

(3) **通信部分の時間の見積もり**
Communication() にかかる時間はデータベースに蓄えられているcollective通信の基本性能データを用いて行なうことができる．

(4) **プログラムの性能予測**
(2) および (3) で得られたデータよりプログラムの性能を予測する．

我々の方法でどの程度正確に予測することができるかを示すためにいくつかの事例に関して実際に予測を行ない，実測値との比較を行なった．ただし，ここでは局所計算部分と通信部分の切り分けは手で行ない，また，前述のように局所計算にかかる時間はプログラムを解析して見積もるのではなく，実際に局所計算部分のみを処理するプログラムを実行することによって求めている．

並列プログラムの性能予測に関しては，コンパイル時にプログラムの構造を解析して性能予測を行なう方法[6]などが提案されている．これらの方法の多くはプログラムの構造を解析してプログラムを簡単な (抽象化された) 中間コードに変換し，そのコードを用いてプログラムの実行にかかる時間を予測するという方法をとっている．この方法はプログラムを全く実行することなく性能を予測できるという利点がある反面，各部分の予測値がかなりおおまかになってしまうという欠点がある．たとえば，文献[6]の方法では通信時間のコストを見積もるために *delay* や *use* など何種類かのオペレーションを用いてプログラムを抽象化しているが，このような方法では *reduction* や *complete exchange* など全プロセッサが同時に通信に参加するような通信のコストを見積もることは難しい．

3. 事　　例

本節では，実験に用いたプログラムを例に性能予測の方法を具体的に説明する．実験は66ノードのParagon[1]を用いて行なった．示される値はいずれも数回実験を行なったうちの最良値である．

3.1 行列の乗算

行列の乗算を *shift* を用いてデータ並列により行なう．行列のサイズを変化させて，PE数が32および64台の場合に関して性能予測を行なった．行列 A と行列 B をそれぞれ行方向と列方向に分割し，A_i および B_i を i 番目のプロセッサに持たせる．このようにデータを各プロセッサに保持させた場合，行列の乗算のプログラムの主要部分の概略は以下のように記述できる．

PEはプロセッサの台数である．*ComputeSubmatrix()* においては，各プロセッサは自分が保持しているAとBの行列の乗算を行なう．次の *Shift()* におい

```
while ( condition ) {
    ComputeSubmatrix(A_i, B_j);
    Shift(B_j);
    UpdateMatrix(B_j);
}
```

ては各プロセッサは *shift* オペレーションを使って自分が保持している B の行列を隣 (自分の番号より 1 大きな番号) のプロセッサに送り，自分は逆隣 (自分の番号より 1 小さな番号) のプロセッサからそのプロセッサが保持していた行列を受信する．*UpdateMatrix()* では受信の際に用いた一時的なバッファからデータ保持用のバッファに受信内容をコピーする．モデル化および性能予測の手順を以下に示す．

(1) **プログラムのモデル化**

```
LocalComputation()
{
    ComputeSubmatrix(A_i, B_j);
    UpdateMatrix(B_j);
}
Communication()
{
    Shift(B_j);
}
```

(2) **局所計算部分の時間の見積もり**
LocalComputation() のみを処理するプログラムを実行し，その時間を計測する．

(3) **通信部分の時間の見積もり**
通信は *shift* のみであるので，それぞれのバイト数に対する *shift* の性能をデータベースより求める．

(4) **プログラムの性能予測**
(2) および (3) で得られたデータを繰り返しの回数 (PE 数) 倍してそれぞれの処理にかかる時間を見積もり，それらを加えることによってプログラムの性能を予測する．

表 1および表 2に結果を示す． LP time は 1 回の局所計算にかかった時間，CP time は *shift* にかかる時間，pretime は LP time と CP time を加えたものを PE 数倍したものであり，予測される時間を表す．exetime は実際に行列の乗算プログラムを実行した結果，error は pretime と exetime の差の割合を示した ((exetime - pretime) / exetime) ものである．64 台で行列のサイズが小さい場合は error の値が大きいくなっているが，それ以外の error は 1%前後であり，かなり正確に予測できている．PE 台数が多いのに転送するメッセージのサイズがが小さい場合は collective 通信の基本性能を測定する場合にもその結果にかなりゆらぎが見られ，1 度の通信にかかる時間が一定ではないと考えられる．メッセージサイズが小さい場合には全実行時間に対する通信時間の割合がかなり大きいため，その影響を大きく受けて error が大きくなってしまうと考えられる．実際には大きなサイズのデータに対して性能予測を行なうと考えられるのでデータサイズが小さい場合に error が大きくなるのは問題ではないが，このプログラムではデータサイズが大きい場合には通信時間の割合が数% とかなり小さくなってしまい,「実行せずに性能予測できる」という利点が大きく現れない．

3.2 CG(疎行列) カーネル

今回の実験では行列を行方向に分割して各プロセッサに持たせた．データは疎行列である．行列のサイズは 1400(double) であり，PE 数が 32 台と 64 台の 2 通りの場合に関して性能予測を行なった．行列 A のデータを行分割して各プロセッサに持たせ，ベクタ V は各プロセッサに持たせる．CG 法のプログラムの主要部分の概略は以下のように記述できる．

```
CopyVector(V, r);
CopyVector(V, p);
SetVector(0.0, z);
Dotpro(V, V);
while ( condition ) {
    Matvec(A_i, V);
    Dotpro(p, q);
    AddVector(p, z);
    AddVector(q, r);
    Dotpro(r, r);
    AddVector(r, p);
    CollectVector(p);
}
CollectVector(z);
Matvec(A_i, z));
AddVector(x, r);
Dotpro(x, z);
Dotpro(x, z);
Dotpro(r, r);
```

p, q, r, zは一時的に用いるベクタである．*Matvec()* においては，各プロセッサは自分が保持している行列とベクタの乗算を行なう．*Dotpro()* ではベクタの内積を求めるが，その際に各プロセッサが保持しているベクタの内積を求めたあとに各プロセッサ間で *reduction* を行なう．*LocalSum()* は各プロセッサが保持しているベクタの加算を行なう．*CollectVector()* は各プロセッサが保持しているベクタを *complete exchange* によって全対全で交換する．ここでは疎行列は 0 以外の要素をベクタに格納する表現 (Row Compressed Format) を用いているため，実際に交換するのは送るベクタの開始インデックス (int)，要素数 (int)，および実際のデータである．モデル化および性能予測の手順を以下に示す．

(1) **プログラムのモデル化**
行列の乗算と同様に，*LocalComputation()* と *Communication()* を定義する．

(2) **局所計算部分の時間の見積もり**
LocalComputation() のみを処理するプログラムを実行し，その時間を計測する．

(3) **通信部分の時間の見積もり**
通信は *Dotpro()* で行なわれる *reduction* と *Col-*

表1　Paragon における行列の乗算の性能予測 (32 台)

Matrix size	LP time (msec)	CP time (msec)	pretime (msec)	exetime (msec)	error (%)
64	0.254	0.211	14.88	18.98	21.60
128	1.803	0.556	75.49	74.94	-0.73
384	48.46	2.561	1632.67	1613.43	-1.19
512	113.10	3.271	3723.87	3737.71	0.37

表2　Paragon における行列の乗算の性能予測 (64 台)

Matrix size	LP time (msec)	CP time (msec)	pretime (msec)	exetime (msec)	error (%)
128	0.51	0.194	45.06	47.93	5.99
256	3.78	0.510	274.52	280.12	2.00
384	12.54	0.936	862.46	871.00	0.98
512	29.22	1.536	1968.38	1982.53	0.71

lectVector() で行なわれる *complete exchange* であるので，それぞれのデータサイズに対するそれらの通信の性能をデータベースより求める．

(4)　プログラムの性能予測
reduction にかかる時間を T_{red}，*complete exchange* にかかる時間を T_{ex}，局所計算にかかる時間を T_{local} とすると，予測される時間 pretime は繰り返しの回数を $NITCG$ とすると

$$pretime = (2T_{red} + T_{ex}) * NITCG + 4T_{red} + T_{ex} + T_{local} \quad (1)$$

により求めることができる．

図1に予測時間と実測時間を示す．予測される時間は各通信にかかる時間が分かるように示されている．局所計算にかかる時間に比べて通信にかかる時間が大きな割合を占めることが分かる．32 台と 64 台の場合を比較すると，局所計算にかかる時間はは 64 台の方が短いが，大きな割合を占める通信にかかる時間が 64 台の方が倍近くかかってしまい，結果として 32 台の方が約半分の時間で実行できることが分かる．

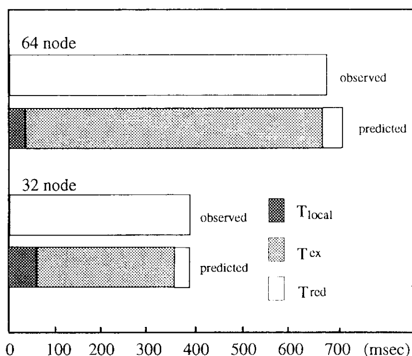

図1　Paragon における CG カーネルの性能予測

3.3 radix ソート

radix ソートは各データを n 進数で表し，各桁ごとのソートを繰り返すことにより全体のソートを行なう方法である．効率化のために n としては 2 の巾乗を用いるのが一般的である．データの取り得る範囲があらかじめ分かっている場合には非常に効率的なソートであり，容易に並列化することができ，十分なデータ数があればプロセッサ台数に対してスケーラブルな性能を期待することができる．以下に 4 プロセッサで radix を 4 とした場合の最下位桁のソートを例に，ソートの手順を説明する．

(1)　各 PE でローカルに最下位桁が 0,1,2,3 である要素数を数える (local count).

(2)　(1) の結果の自分より PE 番号の小さい PE までの部分和を求める (scan count).

(3)　もっとも番号の大きな PE が (2) で得られた scan count に対してその累計を求め，結果を全 PE に配る (global count).

(4)　各 PE は番号が 1 大きな PE に対して自分の scan count の値を送り，(2) で得られた global count と番号が 1 小さな PE から受け取った scan count の値を元に転送アドレスを求める (transfer address)．ある数値の転送アドレスを求めるには，その数値より 1 小さな数値の global count(数値が 0 の場合は 0) に，(4) で送られてきた scancount の値を加えることにより求めることができる．

(5)　各データを上の転送アドレスから転送する (transfer).

PE 数を 32 台と 64 台の 2 通りの場合に関して実験した．全データ数は 65536 個であり，各 PE が保持するデータ数は 32 台の場合は 2048 個，64 台の場合は 1024 個とした．各データは 11bit に収まるように設定し，radix は 2 の 8 乗で行なったため，2 回のソートを行なえば良いことになる．

radix ソートのプログラムの主要部分は以下のように記述できる．

```
LocalCount(data, count);
VectorScan(count);
GlobalCountAndBroadcast(count, sum);
Shift(count);
DetermineRank(count, sum, sendcount, senddata);
CompleteExchange(sendcount, senddata);
```

data は各 PE がデータを蓄える配列，*count* は各数

値ごとの要素数を格納する配列，*sum* は global count を保持する配列，*sendcount* および *senddata* はデータの送信に用いる領域である．*LocalCount()* において，各プロセッサは処理 (1) を行なう．*VecorScan()* は (2) の scan count を求める処理を行なう．*GlobalCountAndBroadcast()* ではもっとも番号の大きな PE が global count を求め，結果を全 PE に *broadcast* する．*Shift()* では (4) で行なう「自分の scan count を隣の PE に送る」という処理を行なう．*DetermineRank()* で各要素の転送アドレスを求め，*CompleteExchange()* で実際の転送を行なう．モデル化および性能予測の手順を以下に示す．

(1) **プログラムのモデル化**
行列の乗算と同様に，*LocalComputation()* と *Communication()* を定義する．

(2) **局所計算部分の時間の見積もり**
LocalComputation() のみを処理するプログラムを実行し，その時間を計測する．

(3) **通信部分の時間の見積もり**
通信は *VectorScan()* で行なわれる *vector scan*, *GlobalCountAndBroadcast()* で行なわれる *broadcast*, *Shift()* で行われる *shift* および *CompleteExchange()* で行われる *complete exchange* であるので，それぞれのバイト数に対するそれらの通信の性能をデータベースより求める．

(4) **プログラムの性能予測**
vector scan にかかる時間を T_{vs}，以下同様に *broadcast* は T_{bc}，*shift* は T_{sh}，*complete exchange* は T_{ex}，局所計算にかかる時間を T_{local} とすると，予測される時間 pretime は

$$pretime = T_{vs} + T_{bc} + T_{sh} + T_{ex} * 2 + T_{local} \quad (2)$$

により求めることができる．

図 2に予測時間と実測時間を示す．予測される時間は各通信にかかる時間が分かるように示されている．32 台と 64 台の場合を比較すると，局所計算は 32 台の方が倍近くかかるが，通信時間は 64 台の方がおおくかかり，結果としていずれもほぼ同じ時間で実行できることが分かる．

4. 検　　討

実験より，プログラムが小さなサイズのデータに対して行なう通信を含んでいる場合には PE 台数が大きくなると誤差が大きくなるという傾向が見られるものの，全体としては予測値と実測値との誤差は数パーセント程度と高い精度で予測できることが分かり，我々の性能予測の方法の妥当性が示された．我々の方法はプログラムを局所計算部分と通信部分とに分割してそれぞれにかかる時間を実験やデータベースより求めて加えるというものであるが，今回の実験では局所計算部分と通信部分との切り分けは手で行ない，局所計算部分の時間の見積もりには実際に局所計算部分の処理を行なうプログラムを実行して求めた．

4.1 スケーラビリティ解析

今回の例では局所計算の計算量はデータサイズの線形関数として表されるため，局所計算部分の性能モデルをデータサイズをパラメータとして表し，局所計算にかかる時間を計測せずに，すでに得られているデータと性能モデルより求めることも可能である．たとえば行列の乗算の局所処理の計算量は行列のサイズ N の 3 乗に比例することを利用すれば，64 台の場合の局所処理にかかる時間は実際に計測せずに 32 台の場合の局所処理にかかる時間をもとに求めることができる．ただし，この方法では計算量がデータサイズのどのような関数として表されるかが明らかにする必要がある．データサイズをパラメータとする性能モデルを構築できれば，プロセッサ数を変えたときのスケーラビリティなどを解析することができる．

4.2 異なる platform での予測

通信部分の時間はデータベースより求めているため，簡単な方法で別の並列システム上での挙動もある程度予測することができる．たとえば，CG カーネルをワークステーションクラスタ[3]上で実行した場合にどのような性能が得られるかを考える．データベースに保存されているワークステーションクラスタ上での *reduction* および *complete exchange* の性能およびそれらの値から求めた通信時間 T_{com} を表 3に示す．単位は msec である．CG カーネルの通信時間 T_{com} は

$$pretime = (2T_{red} + T_{ex}) * NITCG + 4T_{red} + T_{ex} \quad (3)$$

により求められる．

表 3　ワークステーションクラスタおける通信性能

PE 台数	T_{red}	T_{ex}	T_{com}
32	9.82	63.47	2180.5

Paragon で 32 プロセッサの場合には通信時間は約 327msec であったのに対し，ワークステーションクラスタでは 2180msec もかかることが予測される．局所計算にかかる時間は Paragon で約 60msec 程度であるので，ワークステーションクラスタでは通信部分の性能がボトルネックとなって CG カーネルの性能はかなり劣化することが予測される．

4.3 性能解析ツール

今後プログラムを構文解析して自動的に性能解析を行なうツールを開発する上で，いくつかの問題点がある．

- 局所計算にかかる時間を正確に見積もるのは難しい．たとえば，データがキャッシュにのっているかいないかによってデータへのアクセス時間は大幅に変わってくる．
- 条件分岐など動的な要素を含む部分の時間の見積

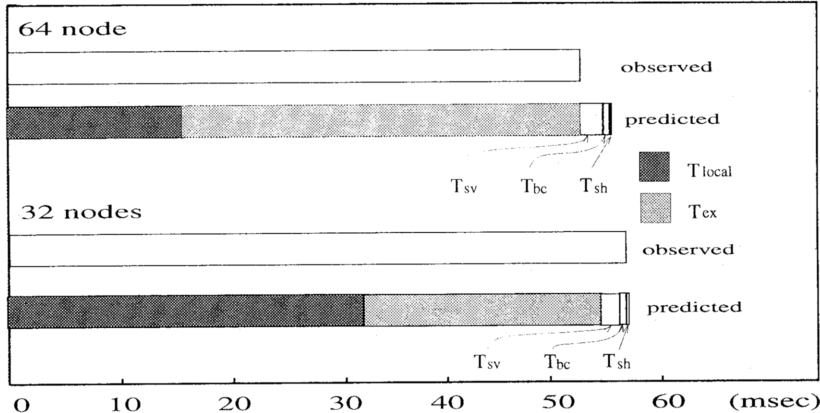

図2 Paragon における radix ソートの性能予測

もりにも注意が必要である.

- 今回は典型的なデータ並列プログラムを実験に用いたが, 非同期通信を行なって通信の遅延を隠蔽するような処理を行なっているプログラムの性能予測は今回のように局所計算部分と通信部分の予測値を加えるという方法ではできない.
- 実行時にメッセージサイズが動的に変わるような通信を行なうプログラムの場合には, あらかじめ通信の種類とメッセージの大きさからデータベースを検索するという処理を行なうことができない.

特に局所計算部分の性能予測を正確に行なうことは難しいと考えられる. 全体の計算時間に対する局所計算部分の時間の割合が小さければ誤差は隠蔽され, 通信部分の時間の見積もりが性能予測の精度に大きく影響を与えると思われるが, いずれにしてもこれらの問題を考慮しながら性能解析ツールを作成する必要がある.

5. 結論および課題

本稿では collective 通信の基本性能データを元に性能解析を行なうツールの基本方針として, 性能予測の方法を提案し, いくつかの典型的な事例について, 解析を試みた. 様々な並列計算機における collective 通信の性能はデータベースとして保持されているので, この方法を用いれば様々な並列計算機上でプログラムを実行した場合の性能特性を知ることができる. 並列システムの collective 通信の性能をデータベースとして蓄えていくことにより, それらのシステム上での並列プログラムの性能を解析, 調査することが可能となる. 今後はプログラムの構造を解析して自動的に局所計算部分と通信部分とに分割し, 前述の問題点を解決しながらそれぞれの実行時間を予測して全体の性能を予測するような性能予測ツールを開発する予定である.

謝辞

本研究に対して議論に参加頂き, 多くのアドバイスを頂きましたNPBグループの皆様と電子技術総合研究所の児玉祐悦氏に感謝致します. なお, 本研究の一部は工業技術院国際特定共同研究「ハイパフォーマンスコンピューティングシステム性能評価モデルの研究」に基づくものである.

参考文献

1) "Paragon System Administrator's Guide", Intel Corporation, (1995).
2) S. Sekiguchi and M. Sato: "A Performance Model and Metrics for Fine Grain Parallel Computing Systems", Proceedings of 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, (1996).
3) 手塚 宏史, 堀 敦史, 石川 裕: "ワークステーションクラスタ用通信ライブラリ PM の設計と実装", 情報処理学会並列処理シンポジウム JSPP'96, pp. 44–48 (1996).
4) 田中良夫, 久保田和人, 佐藤三久, 関口智嗣: "並列アルゴリズムにおける collective 通信の性能比較", 情報処理学会ハイパフォーマンスコンピューティング研究会報告, Vol. 96, No. 81, pp. 19–26 (Aug. 1996).
5) Y. Tanaka, K. Kazuto, M. Matsuda, M. Sato and S. Sekiguchi: "A Comparison of Data-Parallel Collective Communication Performance and its Application", Proceedings of HPC Asia '97, (1997). (to appear)
6) Arjan J. C. van Gemund: "Performance Prediction of Parallel Processing Systems: The PAMELA Methodology", Proceedings of ICS '93, pp. 318–327 (1993).