ハイパフォーマンス
コンピューティング 67－11
（1997. 8. 19）

SMP クラスタでの共有/分散融合プログラミング

田中良夫† 松田元彦† 安藤 誠†
久保田和人† 佐藤三久†

我々は対称型マルチプロセッサ (SMP:Symmetric Multi Processor) であるサーバ型 PC をネットワークで結合した SMP クラスタ COMPASS を構築した．SMP クラスタのアーキテクチャは分散メモリアーキテクチャと共有メモリアーキテクチャの中間的な形態であり，その性能特性に関する報告はまだされていない．本稿では，COMPASS における共有/分散融合プログラミング上の経験およびその初期的な評価について報告する．バスへのアクセスが集中するようなプログラムにおいては，バスの性能がネックとなり高い並列化効率を得ることはできなかった．SMP クラスタのノードの共有メモリ・バスの容量が十分な場合は有効であるが，今回の Pentium Pro を用いた SMP クラスタの実験では，共有メモリ・バスへのアクセスの集中が原因と思われる障害により SMP ノードの利点を生かすことができなかった．詳細については調査中である．

A Hybrid Shared Memory/Message Passing Programming on SMP Cluster

Yoshio Tanaka ,† Motohiko Matsuda ,† Makoto Ando ,†
Kazuto Kubota † and Mitsuhisa Sato†

We developed a cluster system COMPASS which consists of Symmetric Multi Processor systems. The SMP cluster has characteristics of both distributed memory architecture systems and shared memory architecture systems. In this paper we report a hybrid shared memory/message passing programming on COMPASS and its preliminary evaluation. SMP clusters can provide high performance if shared memory bus of their nodes provide high performance. However, our SMP cluster cannot provide high performance for some programs which include overconcentration of access to shared memory bus because the performance of shared memory bus becomes a bottleneck. We are now under investigation for more details.

1. はじめに

並列計算機のアーキテクチャは分散メモリアーキテクチャと共有メモリアーキテクチャに大きく分類することができる．分散メモリアーキテクチャの並列計算機におけるプログラミング技法の代表的なものとして，メッセージパッシングによるプログラミングがある．これは，MPI[2]や PVM などのメッセージ通信ライブラリを用いてプログラミングを行なうものである．また，共有メモリアーキテクチャの並列計算機においてはマルチスレッドの機能を用いたプログラミングが一般的である．それぞれのアーキテクチャを，プログラミングおよびその性能という点から比較すると，以下のようなことが言える．

- 分散メモリアーキテクチャ
  - (1) メッセージ送受信時に陰に同期がとられる．
  - (2) 通信のオーバーヘッドが大きい．
  - (3) バンド幅はネットワークの性能に依存する．
- 共有メモリアーキテクチャ
  - (1) プログラム中で陽に同期を取る必要がある．
  - (2) critical region で排他処理が必要．
  - (3) 通信のコストが低く，オーバヘッドも小さい．
  - (4) バンド幅はバスの性能に依存する．
  - (5) 局所性を高くし cache を効率良く利用することができれば高い性能を得ることができる．

我々は対称型マルチプロセッサ (Symmetric Multi Processor, SMP) システムをネットワークで結合した SMP クラスタ COMPASS(Cluster Of Multi Processor Advanced SyStem) を構築した．SMP クラスタにはコンパクトな大きさで多くのプロセッサを搭載でき，保守，管理も容易になるという利点がある．SMP クラスタのアーキテクチャは分散メモリアーキテクチャと共有メモリアーキテクチャの中間的な形態であり，各ノード間ではメッセージパッシングにより通信を行ない，各ノード内のプロセッサ間では共有メモリを介し

† 新情報処理開発機構
Real World Computing Partnership

て通信を行なう．各アーキテクチャの利点をうまく引き出すことができれば，システムとして高い性能を得ることができる．並列システムにおいて高い性能を得ることができるかどうかは，プログラムの性質とプログラミングの方法に依存する．本研究は，SMP クラスタにおいて高い性能を引き出すために，まず SMP クラスタの性能特性を明らかにすることを目的とする．我々は共有プログラミングと分散プログラミングを融合してプログラミングを行ない，COMPASS 上で実行した．本稿では，共有/分散融合プログラミング上の経験および，その結果どの程度の性能を得ることができたかに関して報告する．また，プログラムの性質とシステムとの適合性に関して考察する．

次節では我々が開発した COMPASS の仕様を示す．3 節では COMPASS の基本性能に関して述べる．4 節では分散プログラミングと共有プログラミングを融合したプログラミングに関して 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を例に説明し，5 節ではそれらのプログラムの実行結果を示す．6 節では実験結果をもとに COMPASS の性能およびプログラムの性質とシステムとの適合性に関して考察し，最後に結論と課題を述べる．

2. COMPASS の仕様

COMPASS は 4 台の Pentium Pro を搭載したサーバ東芝 GS700 を 8 台 100Base-T Ethernet と Myrinet[1] により接続したシステムである．Ethernet により接続される場合は，各ノードは ether switch を介して接続される．GS700 は 4-Way SMP の Pentium Pro PC(200MHz，512KB キャッシュ，128MB 主記憶) である．本稿で述べる基本性能および実験結果などは，すべて Ethernet を介して接続したものを使用したものである．各ノードの OS は Solaris 2.5.1 である．

3. COMPASS の基本性能

本節ではバスの性能などの各ノードの基本性能および，MPI を用いた通信性能などのノード間の基本性能を示す．

3.1 バスの性能

表 1にバスのバンド幅を，表 2に複数のスレッドで bcopy を行なった場合のバンド幅を示す．

表 1　バスの性能

read(MB/s)	write(MB/s)	copy(min/max)(MB/s)
208.1	78.2	35.5/44.7

バンド幅は各スレッドごとのバンド幅であり，全体のバンド幅は全スレッド合計のバンド幅である．スレッド数が増えても全体としてのバンド幅は一定であることが分かる．

表 2　複数スレッドで bcopy を行なった場合のバンド幅

スレッド数	バンド幅 (MB/s)	全体のバンド幅 (MB/s)
1	71.31	71.31
2	37.06	74.12
3	24.75	74.25
4	18.56	74.24

3.2 スレッド間のバリア同期

表 3にスレッド間でのバリア同期にかかる時間を示す．

表 3　スレッド間のバリア同期

スレッド数	2	3	4
バリア同期の時間 (usec)	1.222	1.761	1.960

mutex 変数を使わずにマスタースレッドが全スレーブスレッドがバリアに入ったのを検出するアルゴリズムであるが，ノード間のバリア同期よりもかなり高速にバリア同期がとれることが分かる．

3.3 ノード間の通信性能

図 1に *point-to-point* の通信性能を，表 4に *MPI_Barrier()* を用いた場合のノード間のバリア同期にかかる時間を，図 2に 4 ノードおよび 8 ノードで pairwise アルゴリズム[4]および *MPI_Alltoall()* を用いて *complete exchange* を行なった場合の性能を示す．

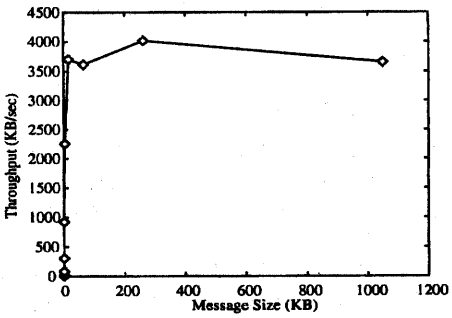

図 1　point-to-point の通信性能

表 4　ノード間のバリア同期

ノード数	2	4	8
バリア同期の時間 (msec)	0.493	1.066	1.645

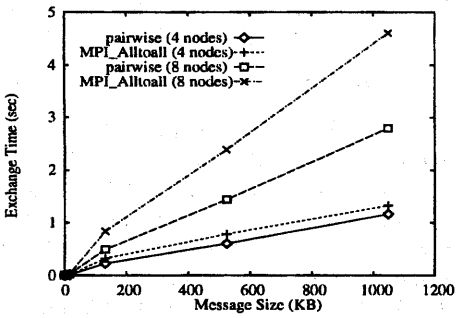

図2 Complete exchange におけるデータ量と通信時間

4. 共有/分散融合プログラミング

SMP クラスタは共有メモリ並列計算機がネットワークで接続され，全体として分散メモリ並列計算機を構成している．8 台の各ノードには 4 台のプロセッサが搭載されているため，全体では 32 プロセッサを備えていることになる．分散プログラミングの技法のみを用いてプログラミングを行なう方法も考えられるが，これらのプロセッサを有効に利用するために，分散プログラミングの技法と共有プログラミングの技法を融合してプログラミングを行なった．本節では CG 法による連立 1 次方程式の解法および radix ソート[4]のプログラムを例に，共有/分散融合プログラミングの方法に関して説明する．

4.1 疎行列 CG カーネル

本稿で実装した CG カーネルは Nas Parallel Benchmarks[3]の実装に基づいている．CG カーネルは反復法による連立 1 次方程式の解法の 1 つであり，行列とベクタの乗算，ベクタの内積およびベクタの加算などの処理が繰り返し行なわれる．行列のデータを各プロセッサに分配することにより，特に行列とベクタの乗算やベクタの内積，加算などの処理に関して高い並列化効率を得ることができる．CG カーネルの分散メモリ並列計算機および共有メモリ並列計算機における実装はそれぞれ以下のような方法になる．

- 分散プログラミング
  - 行列を分割して各ノードに持たせる．
  - 各ノードは与えられた行列の乗算およびベクタの内積，加算を行なう．
  - ベクタの内積および解ベクトルに関しては，ノード間で *reduction* や *complete exchange* を行なってノード間での統一をはかる．
- 共有プログラミング
  - 各スレッドは共有メモリ上にある行列やベクタの担当部分を参照して乗算や内積を行なう．
  - ベクタの内積に関してはスレッド間で *reduction* を行なう．
  - 行列やベクタは共有メモリ上に置かれるため，分散プログラミングで必要とする *complete exchange* の処理は必要ない．
  - 適宜同期をとる．

共有/分散融合プログラミングは以下のような方法で行なう．

- 共有/分散融合プログラミング
  - 行列を分割して各ノードに持たせる．
  - 各ノードは複数のスレッドを生成する．
  - 各スレッドは共有メモリ上に置かれた行列やベクタの担当部分を参照して乗算や内積を行なう．
  - ベクタの内積の際には，各スレッド間で *reduction* を行なったのち，その値を用いて各ノード間で *reduction* を行なう．最後に，その結果を各スレッドに伝える．
  - 解ベクトルに関してはノード間で *complete exchange* を行ない，ノード間での統一をはかる．
  - 適宜同期をとる．
  - ノード間での通信は親スレッドのみが行なう．

このように，CG カーネルの場合には別のノードで複数のスレッドが動作していることを意識する必要がないため，共有/分散プログラミングは比較的容易に行なうことができる．

4.2 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．以下に複数のプロセッサで 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へ転送する (transfer).

radix ソートの分散メモリ並列計算機および共有メモリ並列計算機における実装はそれぞれ以下のような方法になる.

- 分散プログラミング
  - データを分割して各ノードに持たせる.
  - 各ノードは与えられたデータに関して local count, transfer address などのローカルな処理を行なう.
  - scan count, global count, transfer などの処理においては *vector scan*, *broadcast*, *complete exchange* を行なってノード間でデータの交換を行なう.
- 共有プログラミング
  - 各スレッドは共有メモリ上にあるデータの担当部分を参照して local count や transfer address などの処理を行なう.
  - scan count や global count の処理は, 他のスレッドの担当領域を参照して行なう. この処理においては各スレッドの処理が重ならないようにする必要がある.
  - transfer の処理は各スレッドが他のスレッドの担当領域を参照して並列に行なう.
  - scan count, global count, transfer などの処理のフェーズが重ならないように同期をとる.

共有/分散プログラミングは以下のような方法で行なう.

- 共有/分散融合プログラミング
  - データを分割して各ノードに持たせる.
  - 各ノードは複数のスレッドを生成する.
  - 各スレッドは共有メモリ上に置かれたデータの担当部分を参照して local count や transfer address などの処理を行なう.
  - scan count の処理は, 各ノード内で *vector scan* を行なったのち, ノード間で *vector scan* を行なう. ノード間の *vector scan* は親スレッドが行なうが, 子スレッドはノード間の *vector scan* の結果を用いて自分のデータを更新する.
  - global count の処理は親スレッドが行なう.
  - transfer の処理は, 親スレッドが行なう.
  - 共有プログラミングと同様に同期をとる.

CG カーネルと同様に radix ソートも別のノードで複数のスレッドが動作していることを意識する必要がないが, 各スレッドが保持している別のノードに送るためのデータを transfer の際に 1 つのバッファにまとめる必要があり, CG カーネルに比べるとプログラミングは若干複雑となる. しかし, CG カーネルや radix ソートのようなデータ並列プログラムは, データの分割化を段階的に行なう (まずノード間に分割し, 次にスレッド間で分割する) ことができるために, 共有/分散プログラミングは比較的容易に行なうことができる.

5. 実験結果

図 3および図 4に, CG カーネルおよび radix ソートを分散プログラミングおよび共有プログラミングにより実装した結果を示す. 分散プログラムでは各ノードでは 1 つのプロセス (スレッド) しか動作させない. ノード数が 1, 2, 4, 8 台の 4 通りについて実行時間を計測した. 共有プログラムは 1 つのノードでスレッド数が 1, 2, 4 の 3 通りについて実行時間を計測した. CG カーネルのサイズは Class A, radix ソートのデータ数は 4M 個である.

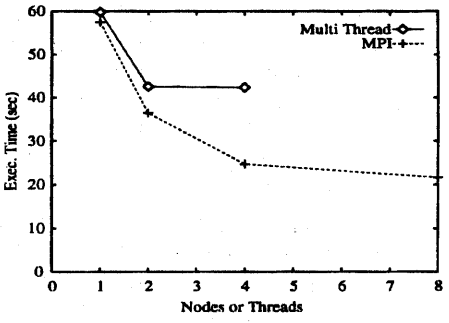

図 3 CG カーネル (Class A) の実行時間 (分散および共有)

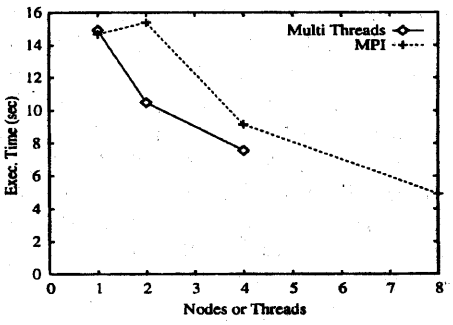

図 4 radix ソートの実行時間 (分散および共有)

それぞれの並列化効率を表 5に示す.

CG カーネルを共有プログラミングで実装した場合, 分散プログラミングで実装した場合に比べて並列化効率が低いことが分かる. 特にスレッド数が 4 になるとその差が顕著に現れる. これは, Pentium Pro のバスの性能がネックとなっているためと考えられる. radix ソートの場合は分散プログラムにおいてノード数が 2 の場合に並列化効率がかなり低い値を示しているが,

表5　CG カーネルおよび radix ソートの並列化効率

スレッド数/ノード数	2	4	8
CG カーネル (共有)	0.70	0.35	
CG カーネル (分散)	0.79	0.58	0.33
radix ソート (共有)	0.71	0.49	
radix ソート (分散)	0.48	0.40	0.37

これは2ノードの場合には各ノードが保持するデータ量2M個であり，*complete exchange* で交換するデータ量が約1M個 (4MB) となるため，この部分の処理に時間がかかってしまうためと考えられる．

図5および図6に，CG カーネルおよび radix ソートを共有/分散融合プログラミングにより実装した結果を示す．ノード数が1，2，4，8台の4通りに関し，各ノード数ごとにスレッド数を1，2，4の3通りに変化させて実行時間を計測した．ノード数とスレッド数の積が動作するスレッド数となる．CG カーネルのサイズは Class A，radix ソートのデータ数は4M個である．

CG カーネルおよび radix ソートに関して，各ノード

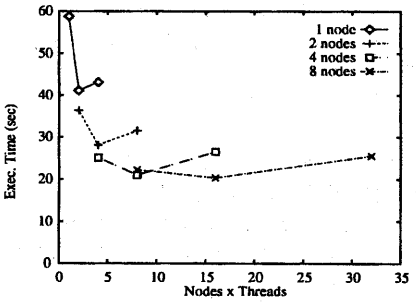

図5　CG カーネルの実行時間 (共有/分散融合)

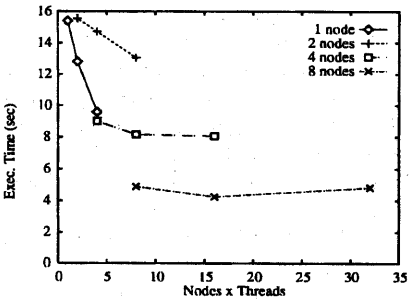

図6　radix ソートの実行時間 (共有/分散融合)

数ごとにスレッド数を変化させた場合の並列化効率を図7および図8に示す．

CG カーネルと radix ソートのいずれにおいてもス

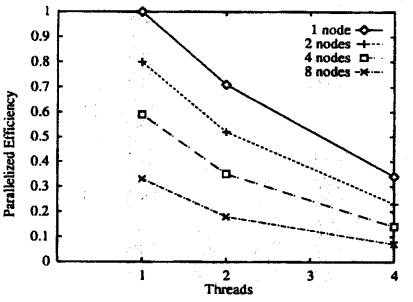

図7　CG カーネルにおける共有/分散融合プログラムの並列化効率

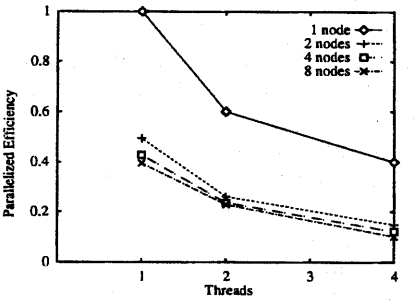

図8　radix ソートにおける共有/分散融合プログラムの並列化効率)

レッド数が増えると並列化効率が低下してしまう．これは，前述のようにいずれの場合においてもスレッド数が増えるとバスの性能がネックになってしまうことが原因と考えられる．

6. 考　　察

CG カーネルに関して，8ノードでスレッド数が1，2，4の場合に各処理 (行列とベクタの乗算，ベクタの内積，ベクタの加算など，*complete exchange*) ごとにかかる時間を計測して高い性能が得られない原因を探った．図9に各処理ごとにかかる時間およびその合計時間を示す．*complete exchange* は親スレッド間のみで行なうため，スレッド数には依存せずほぼ一定の値となる．ベクタの内積はスレッド間で *reduction* を行なうためにスレッド数に依存するが，スレッド間の *reduction* は1から 2μ秒程度で行なうことができるため，結果としてはスレッド数に依存せずほぼ一定の値となる．行列の乗算はスレッド数に応じて高い並列化効率を得ることができるはずであるが，実際にはスレッド数が増えてもあまり実行時間は短縮されない．これは前述のようにバスの性能に原因があると考えられる．実行時間の約50%を占める *complete exchange* の時間がスレッド数に依存しないことと，約30%を占

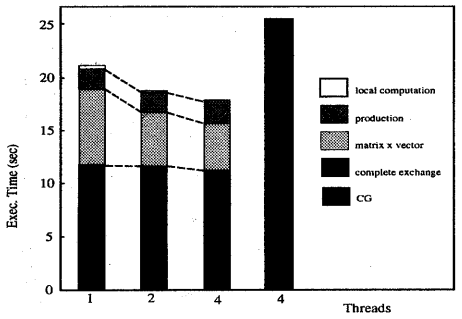

図9 CG カーネルにおける各処理にかかる時間の見積もり

める行列の乗算がバスの性能に抑えられて高い並列化効率を発揮できないことが，全体としてスレッドが増えても並列化効率が高くならない原因であることが分かる．さらに，各処理の時間の合計と図5に示されている実際にかかった時間を比較すると，実際にかかった時間の方が大きくなっていることが分かる．特に4スレッドの場合には各処理の時間の合計の17.83秒に対し，実際には25.51秒と，約43%ほど実際の時間が大きくなっている．これは，実際にすべての処理を行なうとcacheミスが生じてしまうことが原因と推測されるが，行列の非ゼロ要素については局所性は十分高いはずであり，詳細は現在調査中である．

バスの性能がネックになっていることを確認するため，CG カーネルの共有/分散融合プログラムをSPARC Center 2000(20CPU) 上で実行した．mpirunコマンドでMPIを起動する際に，MPIで使用されるすべてのノードはSPARC Center 2000となるように設定し，ノード数は4に固定し，スレッド数を1，2，4に変化させた場合の並列化効率を表6に示す．SPARC

表6 SPARC Center 2000 における CG カーネルの共有/分散融合プログラムの並列化効率

スレッド数 × ノード数	1 × 4	2 × 4	4 × 4
並列化効率	0.73	0.58	0.46

Center 2000 上では4スレッドでもCOMPASSほど並列化効率は低下しないことより，COMPASSではバスの性能がネックになっていることが確認された．

7. 結　　論

本稿では我々が開発したCOMPASSの紹介を行ない，COMPASS上での共有/分散融合プログラミングの方法およびその結果得られた性能に関して報告した．今回題材に用いたCGカーネルおよびradixソートに関しては高い並列化効率を得ることはできなかったが，その原因としては以下のようなことがあげられる．

(1) 複数のスレッドが大量のデータを同時に処理するような場合には，バスの性能がネックとなってしまう．

(2) ノード間の通信は1つのスレッド同士でしか行なうことができないため，ノード間の通信を行なう部分では複数のスレッドを動作させる利点が生じない．特に，ノード間の通信にかかる時間が大きい場合にはこのことは全体の性能に大きな影響を与える．

COMPASSよりもバスの性能が高いノードプロセッサを使用すれば(1)の問題は解決される可能性はあるが，基本的には以下のような性質を持つアプリケーションならばSMPクラスタで高い実行性能を得ることができると考えられる．

(1) データのローカリティが高く (cacheとの相性が良く)，主記憶へのアクセスが集中しない．

(2) ノード間通信は隣接間転送で行なわれ，*complete exchange* のような全対全の通信がない．

(3) 親スレッドがノード間通信を行なっている間に子スレッドが処理を行なうことができる．

(3)に関しては，データ並列プログラムでは困難であるが探索問題のようなプログラムであれば可能であると考えられる．本稿では十分な効果を得る結果については報告できなかったが，現在cacheの影響などについて調査中である．

謝　　辞

本研究にあたり，研究環境を提供して頂いた新情報処理開発機構島田潤一所長に感謝致します．また，クラスタの構築に関してご指導頂いた新情報処理開発機構並列分散ソフトウェア研究室の皆様に感謝致します．

参 考 文 献

1) N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and S. Wen-King, "Myrinet – A Gigabit-per-Second Local-Area Network," IEEE MICRO, Vol. 15, No. 1, pp. 29–36 (1996).

2) http://www.mcs.anl.gov/mpi/index.html

3) 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, S. Weeratunga: "The NAS Parallel Benchmarks", RNR Technical Report, RNR-94-007, NASA Ames Research Center, (1994).

4) 田中良夫，久保田和人，佐藤三久，関口智嗣: "Collective 通信を用いたデータ並列プログラムの性能予測"，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. 97, No. 65, pp. 69–74 (1997).