ハイパフォーマンス
コンピューティング 90−11
（２００２．５．２７）

並列化コンパイラ HPF/ES の不規則問題向け機能

村井 均[†1] 阿南統久[†2] 林 康晴[†3]
末広謙二[†3] 妹尾義樹[†3]
奥田洋司[†4] 横川三津夫[†5]

地球シミュレータ上の HPF コンパイラ HPF/ES に、不規則問題向け機能である HALO を実装した。HALO とは不規則な配列参照と通信を効率良く実行するための機能であり、HALO を用いれば不規則問題を容易に効率良く並列化することが可能である。本稿では、HALO の利用法と実装方式を説明するとともに、地球シミュレータ上で行った評価の結果を示す。HALO を用いて並列化した有限要素法のベンチマークプログラムは、地球シミュレータ上で従来の 10 倍以上の性能を示した。

Features of the HPF/ES parallelizing compiler for irregular problems

Hitoshi Murai ,[†1] Norihisa Anan ,[†2] Yasuharu Hayashi ,[†3]
Kenji Suehiro ,[†3] Yoshiki Seo ,[†3] Hiroshi Okuda [†4]
and Mitsuo Yokokawa[†5]

We implemented a feature for irregular problems, called *HALO*, into the HPF/ES compiler on the Earth Simulator. HALO enhances irregular access and communication of an array, and makes it possible to write efficient parallel programs of irregular problems easily. This paper describes the usage and implementation of HALO and shows its evaluation results on the Earth Simulator. A Benchmark program of the finite element method parallelized with HALO achieved an over 10 times faster execution than the one parallelized without HALO on Earth Simulator.

1. はじめに

地球シミュレータ[4] では、並列プログラミングの手段として、Message Passing Interface (MPI) とともに High Performance Fortran (HPF) を提供する。HPF は Fortran の拡張として定義された並列言語であり、MPI に比べプログラミングが容易であることが特長であるが、コンパイラの能力の制限から、これまで HPF による並列プログラムの性能は MPI には及ばない場合が多かった。最近では優れた HPF コンパイラも現れ、さらに HPF2.0 およびその公認拡張[1] や HPF/JA 言語仕様[2] で加えられた機能を利用すれば十分な性能を得ることができるようになっている[3]。

しかし、有限要素法を始めとする不規則問題に対してはそれらの機能も十分ではなく、多くの HPF コンパイラは不規則問題をうまく並列化できない。不規則問題への対応は HPF の主要な課題である。

これに対し、地球シミュレータ上の HPF コンパイラ HPF/ES は、不規則問題向けに HALO と呼ぶ機能を提供している。HALO とは不規則問題におけるシャドウ (隣接プロセッサ間通信のためのバッファ領域) であり、HALO を用いることにより不規則なパターンの配列アクセスと通信を高速に実行することができる。

本稿は、HALO の利用法と実装方式を説明するとともに、HALO を用いて並列化した有限要素法のベンチマークプログラムを地球シミュレータ上で評価することによって、不規則問題に対する HPF/ES の適用可能性を評価することを目的とする。

以下、2 章で従来手法について述べ、3 章で HALO の概要、4 章で HALO の利用法と実装方式を説明する。5 章では評価結果を示し、6 章で本稿を総括する。

†1 地球シミュレータセンター
Earth Simulator Center
†2 横浜国立大学大学院工学研究科
Yokohama National University
†3 NEC ソリューションズ 第一コンピュータソフトウェア事業部
1st Computers Software Division, NEC Solutions
†4 東京大学大学院工学系研究科 システム量子工学専攻
University of Tokyo
†5 日本原子力研究所
Japan Atomic Energy Research Institute

2. 従来手法

有限要素法に代表される、間接参照により配列が不規則なパターンでアクセスされる問題を**不規則問題**と呼ぶ。不規則問題の並列化を考えるとき、データの分散と通信の 2 つが問題になる。

配列が不規則にアクセスされることから、BLOCK や GEN_BLOCK, CYCLIC といった通常の分散や整列では、十分なローカリティを持つマッピングを記述できない。そこで、あらかじめ、あるプロセッサがアクセスする配列要素が連続した要素番号を持つように並べ替え (リオーダリング) を行っておき、適切な GEN_BLOCK 分散を指定することによって、データのローカリティを実現することができる。

不規則なアクセスに伴って発生する不規則な通信を扱う方法として、Inspector/Executor 法が提案されている[5]。これは、当該ループの最初の実行において通信スケジュールを生成しながら計算を行い (Inspector 実行)、次回以降には先に生成したスケジュールによって通信を行う (Executor 実行) という方法である。Inspector 実行の効率が非常に悪いために、同一の通信スケジュールを利用する Executor 実行を十分な回数だけ繰り返して Inspector 実行のコストを償却できないと高性能を達成できないことが問題である。

3. HALO の概要

通常のシャドウが、規則問題において隣接プロセッサに割り付けられた分割境界上のデータを一時的に保持するバッファ領域として使われるのに対し[1]、HALO は不規則問題において同様の役割を果たすものである[6]。HALO を用いることによって、任意の配列要素を「シャドウ」として指定することが可能になる。

有限要素法における HALO の例を図 1 に示す。有限要素法では、要素と節点を各プロセッサに分割するとき、要素とそれに付随する節点が同一プロセッサ上にある必要がある。したがって、分割境界上にある節点は、それをアクセスする複数のプロセッサに割付けられるが、そのうちのいずれか一つのプロセッサだけが当該節点の**実体**を保持し、それ以外は *HALO* を保持する。図 1 では、分割境界上の節点 n6 を、プロセッサ 1 に実体として、プロセッサ 2 と 3 に HALO として割り付けている。

ユーザの立場からは、HALO を宣言することは、その配列のメモリ割り付けと通信に対し特別な最適化を適用するようコンパイラに要求することを意味する。

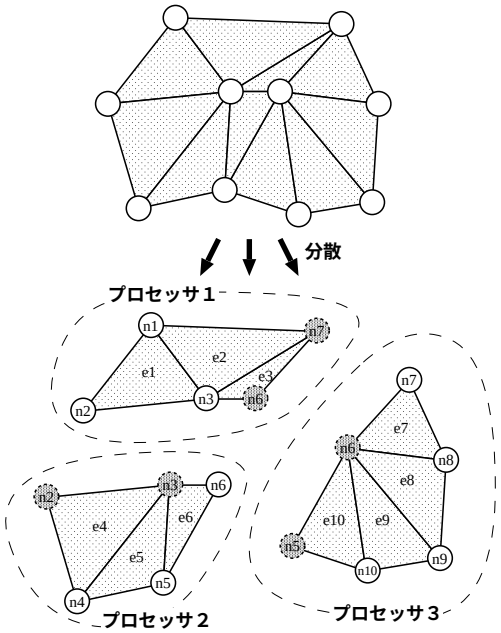

図 1 HALO の例 (三角形は要素を、白丸は節点の実体、グレイの丸は節点の HALO を表す)

4. HALO の利用法と実装方式

以下では、配列 `n` と `e` には、図 1 の状態を実現するような GEN_BLOCK 分散と HALO が指定されているものとする。

4.1 HALO の宣言

ある配列に HALO を宣言するには、HPF2.0 公認拡張の `SHADOW` 指示文の拡張形式を用いて次のように記述する。HPF/ES の独自拡張である `RESHADOW` 指示文を使えば、HALO を動的に変更することも可能である。

```
!HPF$ SHADOW (halo) :: n
```

ここで、`SHADOW` 指示文に指定する `halo` は、以下のように宣言される一次元の構造型配列である。`halo` のサイズは、プロセッサ数に一致させる。

```
type shadow_type
 integer, pointer :: index(:)
end type shadow_type
type(shadow_type), allocatable :: halo(:)
```

`halo(i)%index` には、プロセッサ `i` が HALO として保持すべき配列要素の番号を設定する。

実行時に、各プロセッサは、自分が保持する配列の実体および HALO に対する領域を割り付ける。このとき、実体の直後に連続して HALO が位置するよう

に割り付けを行う。これにより、実体と HALO を同じ方法で (同じインデックス配列による間接参照で) 透過的にアクセスすることが可能になる。

図 1 の例では、配列 n に対し halo は以下のように設定され、このときの各プロセッサにおける配列 n のメモリイメージは図 2 のようになる。

```
halo(1)%index = (/6, 7/)
halo(2)%index = (/2, 3/)
halo(3)%index = (/5, 6/)
```

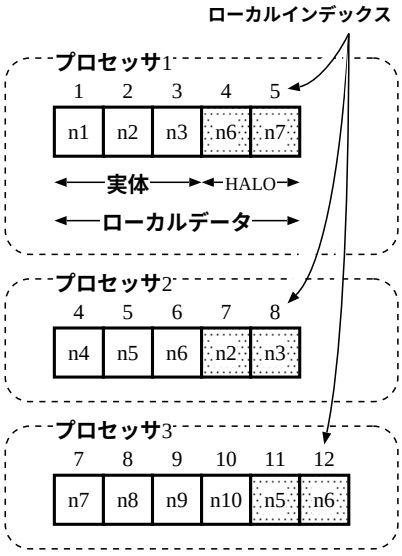

図 2 HALO を持つ配列のメモリイメージ

以下では、実体と HALO を合わせて、ローカルデータと呼ぶ。また、ローカルデータとして保持されている配列要素の、ローカルメモリ上のインデックスをローカルインデックスと呼ぶ。

4.2 HALO のアクセス

本章では、HALO の参照, 更新およびリダクションについて述べる。

参　　照

下のループは、ON 指示文の HOME 節の指定に従い、e(i) を保持するプロセッサが繰り返し i を実行するように並列化される。間接参照される配列 n が分散されているため、通常は効率の悪い通信が生成される。

```
!HPF$ INDEPENDENT
      DO i=1, N
!HPF$  ON HOME( e(i) ), LOCAL(n) BEGIN
       e(i) = n(idx(i))
!HPF$  END ON
      END DO
```

このとき、配列 n の HALO を適切に設定してあれば、繰り返し i を実行するプロセッサが n(idx(i)) を実体または HALO として保持することが保証される。すなわち、あらかじめ HALO を更新してあれば (対応する実体の値を HALO へコピーしてあれば)、このループにおける n の参照はローカルデータのアクセスのみを通じて行うことができ、通信は必要ない。このように、HALO を参照すべきことをコンパイラに指示するには、ON 指示文の LOCAL 節に対象の配列名を指定すればよい。

HALO をアクセスするため、参照 n(idx(i)) の添字式は**ローカルインデックス配列** Lidx に置換される。ローカルインデックス配列はループの繰り返し空間と同じ形状と分散を持つ整数型配列であり、その要素 i には、繰り返し i において参照される配列要素のローカルインデックスを設定する。ローカルインデックス配列は、実行時に当該ループの直前で生成される。

配列 n が図 2 のように割付けられているとき、参照 n(idx(i)) に対して図 3 に示すように Lidx が生成される。例えば、繰り返し i=3 においてプロセッサ 1 が参照する n(7) は、プロセッサ 1 のローカルメモリにおいてローカルインデックス 5 の位置にあるため、Lidx(3) には 5 を設定する。これによって、n(Lidx(3)) はプロセッサ 1 が HALO として保持する n(7) を参照する。

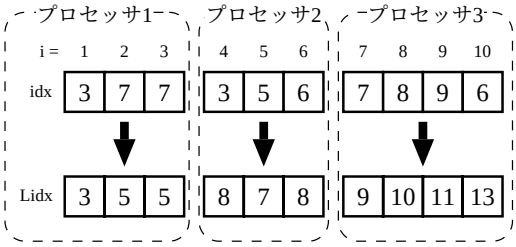

図 3 ローカルインデックス配列

更　　新

HALO を更新するには、通常のシャドウの更新にならって、以下のように HPF/JA 言語仕様の REFLECT 指示文を用いる。HALO の更新の様子を図 4 に示す。

```
!HPFJ REFLECT n
```

HALO を更新する通信は不規則なパターンになり、通常のシャドウを更新する規則的な通信に比べるとコストが大きい。このコストの中で、データの送受信自体とともに、送信元プロセッサや受信先プロセッサ, 送信元アドレス, 受信先アドレス等を計算する処理 (通信スケジュール生成) が大きな割合を占める。

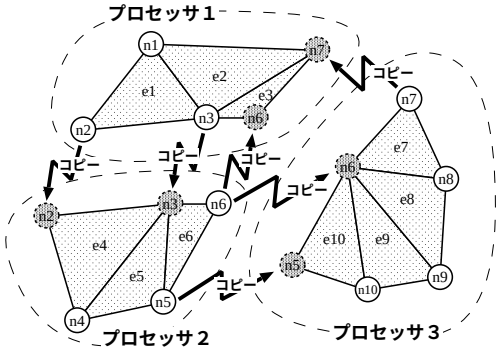

図 4 HALO の更新

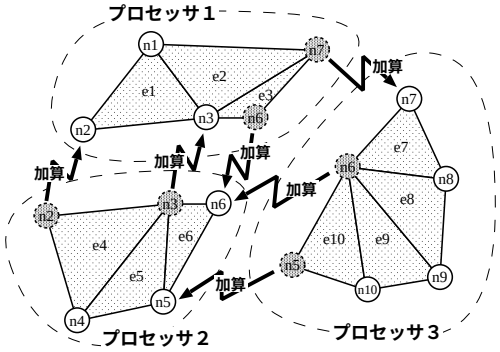

図 5 HALO のリダクション

HALO を更新する通信のスケジュールは、当該配列のマッピングと HALO の宣言によって定まり、配列が再マッピングされたり HALO が再宣言されない限りは変化しない。配列に対する最初の REFLECT 指示文の実行の際に計算された通信スケジュールは、配列記述子の中にキャッシュされ、配列が再マッピングされるか HALO が再宣言されるまで繰り返し利用される。通信スケジュールが再利用されることにより、通信のコストは最小限に抑えられる。

ここで、HALO を更新する通信スケジュールを計算する処理は、Inspector/Executor 法の Inspector 実行に比べると、ずっとコストが小さいことに注意されたい。

リダクション

下のループでは、配列 n に対してリダクション演算 (総和) が行われる。

```
!HPF$ INDEPENDENT, REDUCTION(local:n)
      DO i=1, N
!HPF$  ON HOME( e(i) ) BEGIN
       n(idx(i)) = n(idx(i)) + e(i)
!HPF$  END ON
      END DO
```

n の HALO を適切に設定してあれば、繰り返し i を実行するプロセッサが n(idx(i)) を実体または HALO として保持することが保証される。したがって、通常のリダクション演算と同様に、各プロセッサがローカルデータへのアクセスのみを通じて中間結果 (部分和) を計算した後、その中間結果から最終的な結果を求めるという実装が可能である。最終的な結果を求める集計処理では、各プロセッサの HALO の値を、対応する実体と結合させる (実体へ加算する) 処理が行われる (図 5)。

このような HALO を利用したリダクション演算をコンパイラに指示するには、INDEPENDENT 指示文の REDUCTION 節に、集計種別「LOCAL」とともに対象の配列名を指定すればよい。

ループ直後の集計処理において HALO と実体を結合する通信のパターンは、HALO を更新する通信のパターンの送信側と受信側を逆にしたものに一致する。集計処理の際には、キャッシュされている通信スケジュールを送信側と受信側を逆にして利用する。

4.3 高 速 化

先に述べたように、HALO を持つ配列がアクセスされるとき、元の添字式からローカルインデックス配列が生成される。元の添字式がインデックス配列である場合、これが更新されている可能性を考慮すると、ループが実行されるたびにローカルインデックス配列を生成し直す必要がある。ローカルインデックスの生成は高価な処理であり、頻繁に繰り返されると性能へ及ぼす影響は大きい。しかし、有限要素法の要素と節点の関係を表す配列のように、このインデックス配列は実際にはある一定の期間は更新されないことが多い。

そこで、HPF/ES では、先に生成されたローカルインデックス配列を再利用することを指示する手段を提供する。すなわち、HPF/JA 言語仕様の INDEX_REUSE 指示文 (これは、本来は Inspector/Executor 法のための用意された機能である) をループの直前に指定することによって、ローカルインデックス配列の再利用を指定できる。INDEX_REUSE 指示文の括弧内には再利用できる条件を指定する。

```
!HPFJ INDEX_REUSE (cond) n
      DO ...
```

5. 評　　価

HPF/ES の HALO 機能を利用して並列化したプログラムの性能を地球シミュレータ上で評価した。評価環境は次の通りである。

- OS: ESOS Release 1.1
- HPF コンパイラ: HPF/ES Rev.1.7(585)
- コンパイラオプション:
  `-O2 -Wf"-pvctl vwork=stack"`
- Fortran コンパイラ: FORTRAN90/ES Version 2.0 Rev.250 ES 07

5.1 対象プログラム

評価に用いたのは、六面体要素による簡単な有限要素法のベンチマークプログラムである。その大まかな処理の流れは、

(1) 分散と HALO の情報の読み込み
(2) メッシュ構造と初期値の読み込み
(3) 係数行列の生成
(4) CG 法による連立一次方程式の求解

となっており、(4) を実行する手続き CGSOLV が評価対象である。

対象プログラムを HPF 化したときの注意点を以下に示す。

準　　備

プログラムの実行に先立って、METIS[7] 等のメッシュ分割ツールを使って要素と節点の分散を求める。要素と節点にリオーダリングを適用した後、HALO に設定すべき節点の番号を求める。得られた以下の情報をファイルに出力しておく。

- 各プロセッサに割り当てるべき要素および節点の個数 (要素と節点を GEN_BLOCK 分散するときのマッピング配列の値)
- 各プロセッサの HALO の個数と HALO に設定するべき節点の番号

分散と HALO の設定

このプログラムでは、主要データがモジュール中で宣言されている。モジュール中の配列の分散方法は静的に決定されるため、それらのデータを、実行時にファイルから読み込んだ情報に基づいて分散したり HALO を設定したりすることはできない。

そこで、これらの配列には DYNAMIC 属性を付加し、宣言時には適当な分散を指定しておく。用意したファイルから GEN_BLOCK 分散のマッピング配列と HALO の情報を読み込んだ後に、再分散および HALO の再設定によって正しい分散と HALO を設定する。

一般に、DYNAMIC データが現れるとプログラムの性能は低下するが、HPF/ES の独自拡張である ASSERTION 指示文によって各データの分散を明示することでこの性能低下を回避できる。

ループの並列化

手続き CGSOLV の主要ループは以下である。

```
!HPFJ REFLECT v

!HPFJ INDEX_REUSE (.true.) v, vtem
!HPF$ INDEPENDENT, REDUCTION(local:vtem)
      DO nel=1, nelem
!HPF$  ON HOME( dmat(:,:,nel) ), LOCAL(v) BEGIN
!CDIR  nodep
       DO i=1,8
!CDIR   unroll=8
        DO j=1,8
         vtem(nop(i,nel)) = vtem(nop(i,nel))
     &      + dmat(i,j,nel) * v(nop(j,nel),1)
        END DO
       END DO
!HPF$  END ON
      END DO
```

ある要素に付随する 8 つの節点の v の値から、それら 8 つの節点の vtem の値を更新するのが、このループの行う処理である。節点上のデータである v および vtem に HALO が設定されている。ループの直前の REFLECT で v の HALO が更新されループ中でアクセスされる。vtem に対しては、HALO を利用したリダクション演算が行われる。また、ローカルインデックス配列を再利用するために、INDEX_REUSE を指定している。

HALO を利用しない場合、v および vtem に対してループの直前で非分散状態に等しいサイズの作業領域が割り付けられ、それらに対して通信やリダクション演算が起きる。このような処理は、メモリサイズと実行時間の両方の点で、HALO を利用する場合に比べると効率が悪い。

ベクトル化

先のループでは、i ループに nodep 指示行を、j ループに unroll 指示行を指定してベクトル化を行っているが、ベクトル長は高々8 に過ぎないため、ベクトル処理の効率は低い。ループ長が最も長い nel ループは「間接総和」の形になっている。間接総和は粒子コードなどでも頻出するループパターンであり、これをベクトル化するには、足し込み先の配列 (このループでは vtem) に次元を追加した作業配列を利用したり、Retry アルゴリズム[8] を用いたりする必要がある。いずれの方法でも少なからずプログラムの書き換えが必要になり、HALO を利用した並列化の効率を評価するという観点から今回は適用を見合わせた。

5.2 評 価 結 果

150 万節点、140 万要素の問題に対する評価結果を表 1 と図 6 に示す。比較のため、HALO を用いずに並

列化した場合の結果も示した。表の「F90」は FORTRAN90/ES による 1CPU 実行の結果を示す。括弧内は F90 に対する速度向上である。

表 1 評価結果 (実行時間)

NoP	HALO あり		HALO なし	
F90	1286.9	(1.00)	—	(—)
1	2663.5	(0.48)	2245.2	(0.57)
2	680.1	(1.89)	711.7	(1.80)
4	350.9	(3.66)	374.1	(3.43)
8	191.6	(6.71)	222.1	(5.79)
16	108.8	(11.8)	197.8	(6.50)
32	67.1	(19.1)	140.6	(9.15)
64	45.2	(28.4)	233.6	(5.50)
128	36.9	(34.8)	644.9	(1.99)
256	43.6	(29.5)	2088.6	(0.61)
512	59.7	(21.5)	N/A	(—)

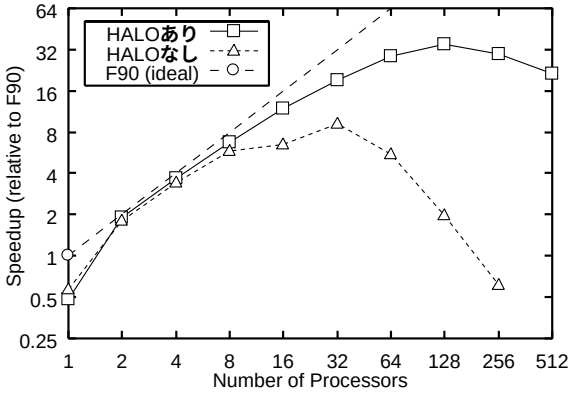

図 6 評価結果 (速度向上)

1〜64 並列は 1 プロセス× n ノード、128 並列は 2 プロセス× 64 ノード、256〜512 並列は 4 プロセス× n ノードで実行した結果である。一ノード内のプロセス数が増えると性能は若干低下するため、128 並列以上の実行時間はやや低目になっている。

表 1 と図 6 より、ある程度のスケーラビリティを 1〜128 並列で達成できていることがわかる。プロセッサ数が小さいときには HALO を用いない場合との性能差は小さいが、プロセッサ数が大きいときには性能差は極めて大きくなり HALO の利用が特に有効になる。

プロセッサ数が増えると、スケーラビリティが低下するのは次のような理由によると考えられる。HALO を利用して有限要素法を並列化するとき、計算量は要素数に、通信量は HALO の個数にそれぞれ比例する。さらに、HALO の個数は要素の分割境界の大きさ (断面積) におおよそ比例する。すなわち、六面体要素を用いるこのプログラムでは、プロセッサあたりの計算量はプロセッサ数 n に反比例し、通信量は n の 2/3 乗に反比例することになり、n が大きくなると相対的に通信の占める割合が大きくなってくる。ベクトル計算機である地球シミュレータでは、この傾向はより顕著であると思われる。

6. おわりに

HPF/ES の不規則問題向け機能である HALO を用いて、有限要素法のベンチマークプログラムを並列化し、地球シミュレータ上で評価を行った。その結果、従来に比べて 10 倍以上の性能が得られ、HALO の有効性が示された。このことから、HPF/ES の HALO を使えば不規則問題も効率良く並列化できることがわかる。今後の課題として、ベクトル性能を改善するコード修正や実アプリケーションでの評価などが挙げられる。

参考文献

1) High Performance Fotran Forum. High Performance Fortran Language Specification. Jan 1997.
2) High Performance Fortran Forum, 財団法人高度情報科学技術研究機構. High Performance Fortran 2.0 公式マニュアル. シュプリンガー・フェアラーク東京, 1999.
3) H. Murai, T. Araki, Y. Hayashi, K. Suehiro, and Y. Seo. Implementation and Evaluation of HPF/SX V2. Concurrency and Computation: Practice and Experience, Wiley & Sons Ltd., (Accepted, to appear in Spring 2002)
4) M. Yokokawa, S. Habata, S. Kawai, H. Ito, K. Tani, and H. Miyoshi. Basic Design og the Earth Simulator. High Performance Computing, LNCS 1615, Springer, pp. 269-280, 1999.
5) J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. Journal of Parallel and Distributed Computing, Vol. 8, No. 4, pp. 303-312, April 1990.
6) S. Benkner. Optimizing Irregular HPF Applications Using Halos. Concurrency: Practice and Experience, pp. 137-155, 2000.
7) G. Karypis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing, Vol. 20, No. 1, 1999.
8) 村井均, 末広謙二, 妹尾義樹. 共有メモリ型ベクトル並列計算機上の高速整数ソーティングアルゴリズム. 情報処理学会論文誌, Vol. 39, No. 6, 1998.