計算機アーキテクチャ　147−8
ハイパフォーマンスコンピューティング　89−8
（2002．3．7）

分散処理環境 VIOS IV の開発

川脇智英† 松尾啓志†

画像処理やシミュレーションなどのデータ依存型処理を，クラスタ環境下において分散・並列処理する場合，適切なデータ配置の決定，およびそれら配置の元での通信の記述が重要な問題となる．そこで，本論文では，クラスタ環境下において，大規模なデータ並列処理を行う分散処理環境 VIOS を提案する．今回新たに，(1) 複数の次元データに対する，統一的な並列処理構文の提供，(2) データキャッシュ領域の更新命令，(3) 処理単位間の依存関係を記述する必要の無い並列処理構文，(4) システム側による通信処理のまとめあげ，通信と計算のオーバーラップ機能，を導入することにより，データ処理一般に対する，より抽象度の高いプログラミング環境を提供することを目指した．更に，VIOS を使用し実際に複数の処理に対して実装を行い，記述能力，および C 言語により記述した場合との比較による処理能力の有効性を確認した．

Distributed Processing Environment VIOS IV

Tomohide KAWAWAKI† and Hiroshi MATSUO†

Recently, in cluster environment, distributed and parallel processing for data-dependent process, such as image processing, a simulation have been reasonable. In this case, description of data distribution, data allocation, and communication become very important problem. This paper proposes the distributed processing environment VIOS which performs large-scale data parallel processing in cluster environment. The new features of the proposed method are : (1) the same parallel processing syntax for multi-dimension data based, (2) cache reflesh instruction, (3) the parallel processing syntax that need not to consider about the dependence between each parallel processing unit, (4) auto communication packing and overlap communication and calculation by the system, aimed at providing the high abstract programming environment for the general parallel data processing. Furthermore, we described some processes using VIOS, and evaluate the description capability and the throughput in comparison with the case that was described by the C language.

1. まえがき

近年，計算機の高速化に伴い，従来困難であった様々な問題を 1 台の計算機でも解くことが可能となっている．しかし，構造解析，電磁界シミュレーションなどの大規模データ処理を行う問題では，大量のメモリ領域を利用する反復計算などが行われ，依然として計算量が大きく，メモリ資源の問題などからも並列処理が望まれるものが多い．また計算機の低価格化に伴い，少数の CPU により構成される SMP 環境や，一般 PC をベースとしたネットワーククラスタを利用する，並列処理環境の構築がより容易なものとなっている．しかし，同期，データ分配，負荷分散など，並列処理を記述するためには様々な問題を考慮する必要がある．そのため，SMP やクラスタ環境を対象とする様々な手法が，従来より提案されている．

† 名古屋工業大学電気情報工学科
Nagoya Institute of Technology, Electrical and Computer Engeneering

計算機間の通信のために，メッセージパッシングを利用するためのライブラリとしては，MPI フォーラムにより規格化された，API 仕様，MPI[1] などが一般的に用いられている．この API に準拠したライブラリ MPICH などを利用することにより，比較的容易に分散処理を実現可能となるため，多くの利用例がある．しかし MPI は，より実用性を上げるため，比較的低水準のライブラリとして実装されており，利用には分散処理の十分な知識を必要とする．

より抽象度を上げ，分散メモリ環境における行列演算を目的としたライブラリとして，BLACS(Basic Linear Algebra Communication Subprograms) がある．BLACS は，MPI などの上に構築されており，そのためプログラマは MPI の API を直接操作する必要がない．またプロセッサのグリッド管理，行列レベルでのデータの送受信，など行う機能を提供しており，より高いレベルでのプログラミングを可能としている．

また並列処理のための専用言語として，京都大学で開発された NCX[2] がある．NCX は，仮想プロセッ

サと，その結合を示すフィールドの概念をベースとすることにより，従来の C 言語との互換性を維持，汎用性，また大規模並列処理への対応を考慮している．

しかしこれら手法では，依然としてプログラマは，計算機 ID・グループなどの形で通信に関する配慮を必要とする．また分散メモリ環境においては考慮する必要が大きいキャッシュ領域に関する配慮，および通信遅延の隠蔽は別の枠組みとして記述する必要がある．

またデータ並列処理の特徴の 1 つでもある，大規模な並列性を有する問題に対する手法も多数提案されている．多田らは longjmp 命令を用いることにより，汎用性が高く，軽量な疑似スレッドライブラリを提供している [6]．また StackThreads[5] では，再帰並列などに見られる，タスク間の指向性を利用し，大規模並列処理に対応している．

しかし，これら手法は，基本的に共有メモリモデルを元に考案されており，またデータ並列処理特有の特徴を活かすものではない．

そこで，本論文では，画像処理専用環境であった VIOS を改良し，データ処理一般に対する，通信環境の考慮を要求しない，より抽象度の高いプログラミング環境の提供を目指す．そのために，次の 4 つの要因を総合的に取り入れた環境の構築を行う．

(1) 複数の次元のデータに対して，統一的な並列処理構文を提供する
(2) より高いレベルでの通信指示命令を用意する
(3) 並列処理時における，各処理単位間の依存関係に対する記述を要求しない
(4) 通信処理のまとめあげや，計算と通信のオーバーラップを自動的に行う

具体的には，以下の 3 つの拡張を行った．

- 3 次元データの扱いも含め，複数次元，複数データを持つ処理に対する並列化を，統一的に記述できるよう並列構文の拡張，
- 複数の次元のデータに対する，キャッシュ領域の一括更新命令の導入
- VIOS プログラミングにより生成される，細粒度論理スレッド（3.2.1 節参照）間の依存関係，および分散メモリ環境に起因する，処理の開始可能時間の差異を吸収する並列処理構文の導入

本論文では，2 章でまず VIOS の基本的なシステム構成について示し，3 章で VIOS におけるプログラミングモデルとその拡張について示す．さらに 4 章において，新たに導入した論理スレッド間の依存関係を考慮する並列処理構文について示す．5 章で記述性・速度の両面に対して評価を行い，6 章でまとめる．

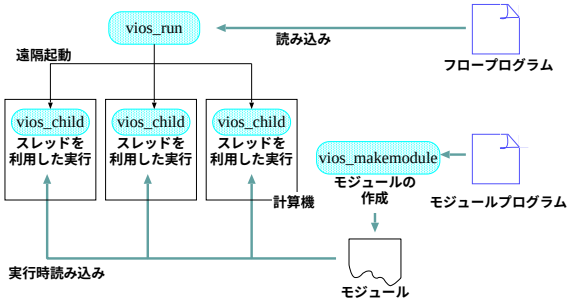

図 1 VIOS システム構成
Fig. 1 VIOS system construction

2. VIOS システム構成

まず VIOS の基本的なシステム構成について述べる．

VIOS はネットワーククラスタ環境上で実行されることを想定し，図 1 に示すように，“vios_run”，“vios_child”，“vios_makemodule” の 3 種類のプロセスにより構成される．

- vios_run は，実行フロープログラムを入力とし，各クラスタ計算機上でのモジュール実行の依頼，共有データの管理などを行うメインプロセス．
- vios_child は，各計算機上に分散配置され，ローダブルモジュールの実行を行うサーバプロセス．
- vios_makemodule は，VIOS で実際に並列実行を行う，動的モジュールを生成するためのスクリプト．VIOS モジュールコードから C++コードへのトランスレータ，および C++コンパイラを呼び出す．

3. VIOS プログラミングモデル

VIOS による並列プログラミングは，実行フロープログラム部，モジュール部の 2 つのブロックに分けられる．そこでまず実行フロープログラミングについて説明する．

3.1 実行フロープログラム

実行フロープログラムは，並列処理に使用する計算機の登録，処理データの読み込み，データの分割配置方式の指定，モジュール実行命令など，並列処理に伴う逐次処理を記述するものである．また通常，これら処理は，種々のパラメータや処理対象などにより変更されることが多い．そこで VIOS では，C ベースのインタープリタ言語 VPE-i を提供し，これによりコンパイルを必要としない，プログラム中のパラメータ変更などへの柔軟な対応を可能とする．

図 2 に，この VPE-i を利用したプログラム例を示す．以下，このプログラムを用いて，VIOS における実行フローの記述法を簡単に示す．

```
// 通信処理初期化 -------------
#pragma host_name host1                ← (1)
#pragma host_name host2
#pragma host_name host3
#pragma host_name host4
#pragma use_intercomm          // 相互通信を利用する
#pragma divide_num 2                   ← (2)
#pragma connect_start          // 接続を開始する

/* データの初期化 **********************
int w,h, pic;                          ← (3a)
w   = 1024;
h   = w;
pic = w/3*2+1;
fImage data(w,h);
int i,j;
for(j=w/3; j<pic; j=j+1) {
    for(i=w/3; i<pic; i=i+1) {
        data(i,j) = 8;
    }
}
************************************/

imgLoad("../image/jacobi.pgm", data);  ← (3b)
set(data, cache, 1);                   ← (4)
module(jacobi, data);                  ← (5)

// 以下処理が続く
```

図 2 メインフロー記述言語 VPE-i による記述例
Fig. 2 An example of main-flow programming by VPE-i

実行フローでは，まず図 2 (1) に示すよう，#pragma 構文を利用し，計算機の登録・通信方式など，使用する通信環境の設定を行う．またここで，大規模なデータ転送が必要な場合に発生する，処理開始時間の遅延を回避するためのデータ分割転送の指示も行う．VIOS では，以下の様に宣言することにより，データの分割転送を行う（図 2（2））．

#pragma divide_num *num*

num は 1 計算機あたりの分割数を指示する．図 2 の例では，各計算機に対し，分割数を 2 と指示している．この指示を受け VIOS は，全ての引数に対しキャッシュ領域や重複転送を考慮した後，データを分割・転送する．また各計算機は，1 データブロックを受取り次第，非同期に受信ブロックに対するモジュール実行を行う．

通信環境の設定後，図 2（3a），(3b）に示すように，大域データの宣言や，初期化，読み込みを記述する．これは C に準拠した文法により記述することが可能である．

並列モジュールの実行は，“module” 命令により行い（図 2（5））, またその際，各データにおけるワーキングセット（3.2.1 節参照）の参照キャッシュ半径を設定することが可能である（図 2（4））．この例では，set 関数，およびキーワード “cache” を用いて，変数 data に対する並列処理の際，各ワーキングセットは半径 1 の周辺参照を行うことを示している．

3.2 モジュールプログラム

モジュールプログラムは，VIOS により実行する並列処理部を記述するものであり，VIOS モジュールプログラミング言語 VPE-p により行う．VPE-p は，一般 C 言語に共有変数型と並列構文を追加した形をとる．そのため，モジュールを記述する際には，MT-Safe な関数を利用する必要はあるが，並列構文中以外の部分において，C 言語に対する制約はほぼ無い．

vs_module *”module_name”*(*vios_value*, ...) { /* 並列モジュールの記述部 */ }

並列モジュールは，予約語 “vs_module” を用いて宣言する．またその引数は，実行フローより自動分割され引き渡される，1,2,3 次元の部分配列である VIOS 変数となる．このモジュール構文内に，C 関数，および以下に述べる並列構文を用い，処理を記述する．

3.2.1 ワーキングセット

まず，VIOS により並列処理を記述する上での基本概念について述べる．データ並列処理は一般に，ある特定データの小領域（pixel など）単位に対して並列性を有することが多い．例えば，Laplacian フィルタの場合，処理対象画像の各 pixel，ボロノイ分割の場合，戊点を格納した配列の各要素，単位で並列処理することが可能となる．

そこで VIOS では，並列処理の軸とするデータに対し，

(1) 各処理の中心となる注目要素
(2) 並列処理の間，参照のみを行うキャッシュ半径
(3) データ並列の依存関係

の 3 種類の情報を有する，ワーキングセットと呼ぶ単位を，無限に生成可能な論理スレッド上で並列処理する．なおワーキングセットにおける，データ並列性は，3.2.2 章で述べる並列構文により決定することが可能である．

3.2.2 並列処理構文

VIOS では 3.2.1 節で定義したワーキングセットに対して，以下に示す parallel 構文を利用した並列処理記述スタイルを従来より定義している．

parallel{ /* 注目領域に対する処理の記述 */ }

この parallel 構文内において，1 ワーキングセットに対する処理を記述することで，全データへの並列処理

```
//通常の C 関数の定義やヘッダのインクルード可能

vs_module
jacobi(fImage data)
{
    fImage_opt shadow(data);  // テンポラリ変数の作成
    // 径 1 の外周（全領域中での）は処理からはずす
    vsSetBoundary2D(1);
    for(;;) {
        parallel(x,y) {                                  ← (1)
            shadow[][] = (data[-1][] + data[][-1] + \
                          data[+1][] + data[][+1]) / 4;
                                                         ← (2)
        }
        ImgCopy(data, shadow); // 結果を元の変数に戻す
        vsSyncCache(data); // キャッシュ領域の更新 ← (3)
    }
}
```

図 3 Jacobi 処理に対する VIOS モジュール
Fig. 3 VIOS module programing for jacobian method

を行う．

この parallel 構文に対し，複数次元データを一括して扱うため，以下のような拡張を行った．

parallel<*WS* サイズ,...>(*index* 変数, ...){
/* 注目領域に対する処理の記述 */ }

引数について，*WS* サイズは 1 ワーキングセットに含まれる要素の数を示し，*index* 変数は，ワーキングセットが処理中，自身を特定する ID として利用する変数を示す．また引数の数により，モジュールが持つ共有データ（= モジュールの引数）中の，どのデータを元として並列処理を行うかを指定する．

例えば，*parallel*<2,3>(x,y){ .... } と記述した場合，2 次元共有データに対し，2x3 のブロックを 1 ワーキングセットと見なし，並列処理を行うことを示す（変数 x,y により各ワーキングセットは自身を特定する）．

なお，モジュール中に，同一次元のデータが複数存在する場合は，最初のモジュール引数の情報に従う．

parallel 構文を用い，VIOS モジュールを記述した例を，図 3 に示す．なお，このプログラムは 5.1 節における評価の際に用いるものであり，詳細はそちらで示す．

図中（1）において，本章で示す並列処理構文が利用されている．また，(2）に示すよう，並列処理構文中における大域変数アクセスには，注目要素を中心とした相対アドレス指定により周辺データへのアクセスを行うことが可能となる．

3.2.3 多次元キャッシュ領域の一括更新命令

Jacobi 法，電磁界シミュレーション [10] などの処理では，時間軸方向に同期を取りながら，並列処理を反復して行う場合がある．そこで，2,3 次元データ／分割を考慮した，各計算機上が保有するキャッシュ領域を一括して更新するための命令を導入する．

vsSyncCache(*VIOS* 変数);

この命令により，図 3（3）に示すよう，煩雑な通信処理部の記述を大幅に回避した，抽象度の高い記述が可能となる．

4. 並列構文の拡張

従来の VIOS では，ユーザが生成した大量の論理スレッドは，画像処理特性を考慮し，for 文をベースとした連続処理へと置き換えていた（メタスレッド化）．メタスレッド化により粒度を高くすることで，管理資源の節約，高速な連続実行が可能となるが，並列処理一般を考慮する場合，論理スレッド間に依存関係が生じる場合がある．

そこでこの問題を考慮する，Inspector/Executor 手法 [7]（以下 I/E 法）の概念を論理スレッドレベルで取り入れたスケジューリング手法を新たに導入する．

4.1 導 入 手 法

VIOS では，I/E 方式を各論理スレッドレベルで導入し，かつ実行レベルを 2 つに分けることにより，中断される可能性のあるスレッドの実行開始を回避する．すなわち，各論理スレッドの実行の前に Inspector 部を実行し，その結果より，検査を行った論理スレッドを今実行開始するか，または後回しにするかを決定する．

導入手法のフローを図 4 に示す．

本手法を用いることにより，

- 各論理スレッドに依存関係が存在する場合でも，メタスレッドの枠組みをそのまま適用することが出来る．
- 実行できない論理スレッドに関しては，実行を開始しない（中断では無い）ため，コンテキストの保存が必要ない．
- Inspector 部において，ネットワーク外データへのアクセスを調べることにより，データの先読み効果，通信のまとめあげ効果，計算による通信の隠蔽効果を期待することが出来る．(図 4 中（**））

以上のような効果が期待できる．

4.2 拡張並列構文

以上に述べた手法を利用し，以下の並列構文を新たに導入した．

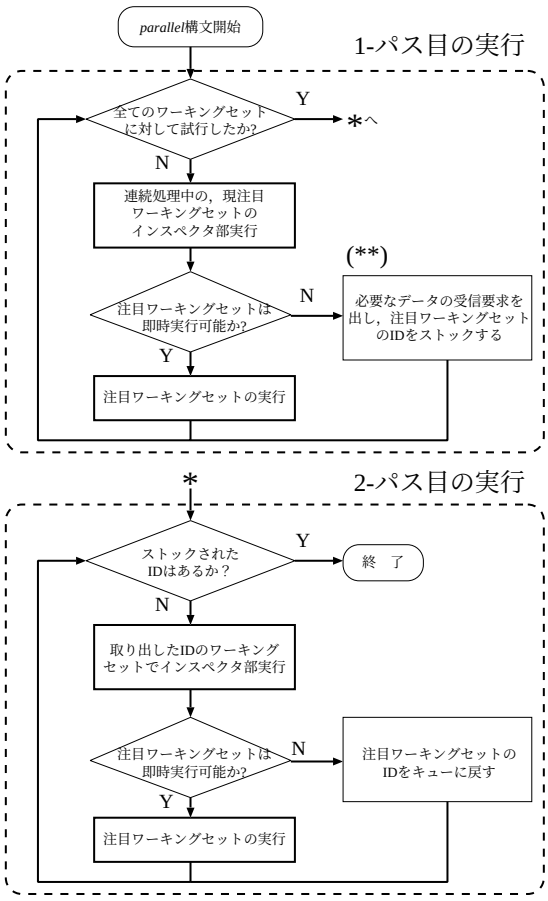

図 4 提案手法の実行方式
Fig. 4 The execution of proposed method

parallel_ie<*WS* サイズ,...>(*index* 変数, ...){
/* 注目領域に対する処理の記述 */ }

parallel_ie 構文は並列処理の際，ワーキングセットが以下の様な特徴を持つ場合に利用する．

- 各ワーキングセットの処理に依存関係があり，またその依存関係がループ・再帰処理とは異なり，生成順の様な単純な形とならない場合．
- ワーキングセットが，他のワーキングセットが保有するデータを参照し，かつその領域サイズが動的に決定する場合．

なお，パラメータなどに関しては，上記 *parallel* 構文と同様となる．

5. 性 能 評 価

VIOS におけるアルゴリズムの記述能力と，クラスタ環境下における速度向上率を評価するために，以下の 4 つの処理について評価を行った．

処理 1 2 次元空間に対する Jacobi 処理．並列処理を反復して行う処理であり，1 並列処理が終了するたびに，キャッシュ領域の更新が必要となる．

処理 2 3 次元空間に対する Jacibi 処理，上記処理の 3 次元空間に対するもの．

処理 3 セルオートマトンを利用したガス拡散シミュレーション [8]．並列処理の反復のたびに，処理を継続するために必要となるキャッシュ領域が変化する処理．

処理 4 ユークリッド地図作成処理 [9]．各ワーキングセットのキャッシュ半径を動的に決定する処理．

5.1 逐次処理用プログラムと比較した変更点

まず処理 1，および 2 について評価する．VIOS により作成した，処理 1 のためのプログラムを図 3 に示す．なおこのプログラムでは，収束判定部，外周処理部などを除いているが，収束判定部のおいて，VIOS により提供されるリダクション命令 [4] が 1 文追加されることを除き，他の個所は逐次処理の場合と同様に記述可能である．

このように，逐次処理用プログラムと比較し，変更する必要のある個所は図 3（3）により示した，キャッシュ更新命令のみとなり，プログラマは一切煩雑な通信に関する考慮を必要せずに，並列プログラムを記述することが可能となる．

また処理 2 に関しても，処理 1 と，並列構文内における実際の処理部以外は同様の記述のみで作成可能となる．

次に，処理 3，4 について評価を行う．処理 3,4 のように，分散メモリ環境下では，データ依存関係が動的に変化する問題は，データ分割境界面付近の処理を開始するために大きな時間が必要となる．そのため，並列処理による速度向上を得るためには，通常，境界面のワーキングセットを別定義し，処理を分ける必要がある．また処理 4 のように，計算機外データを必要とするワーキングセットすら不定となるような問題に対しては，その記述すら困難なものとなる．これに対し，VIOS を利用したプログラムでは，parallel_ie 構文を用いた以外は変更の必要なく，逐次処理と同様のアルゴリズムで記述することが可能であった．

5.2 クラスタ環境下における性能評価

上記処理の実装を行い，速度向上率を評価した．評価には，CPU:Athron1.2GHz，Memory:512Mbyte，OS: Solaris8 の計算機を，100BaseTX スイッチングハブ (Bay networks Model 28115) で 8 台接続したクラスタシステムを利用した．また各処理は，2 次元データに対する処理は，一辺の長さが 1024，2048，4096 の，3 次元データに対しては 128，256，384 の，それぞれ 3 段階のデータサイズに対して評価した．なお，

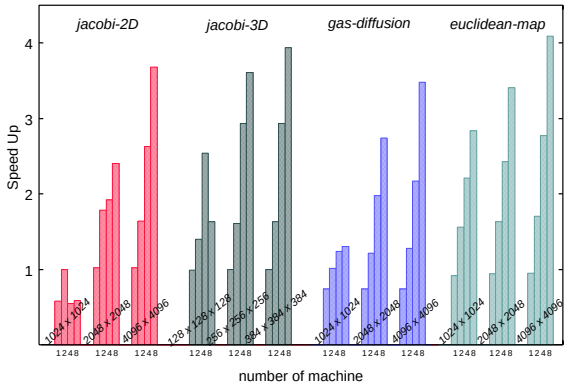

図 5 VIOS による実行速度向上率
Fig. 5 Speed up rate of Execution time for VIOS

処理 4 に関しては，今回の評価では，負荷分散はその評価範囲ではないため，物体点は 1/20 の確率でランダム配置した．評価結果を図 5.2 に示す．

図中，横軸は使用計算機の台数，縦軸は，逐次処理用に最適に実装したものと比較した，本システムでの速度向上率となっている．結果，各処理において，処理 1,2,4 については，8 台の計算機を使用時に 4 倍程度，処理 3 については 3.5 倍程度の速度向上が見られるものとなった．

処理 1,2 について，本評価に用いた Jacobi 法は，計算が非常に単純なものにより構成される（各ワーキングセットあたり数回の加算と一回の除算）．このため，並列処理に伴う同期処理，およびデータ転送処理にかかる時間のためにこのような結果となったものと思われる．同様のアルゴリズムをベースとしたより実用的な処理（電磁界シミュレーション [10] など）ではより高い速度向上が見込まれる．

処理 3,4 について，これらの処理は parallel_ie 構文を用いて記述したものであるが，その速度向上率には差がみられる．これは，処理 3 が，各ワーキングセット辺りの計算量が小さいため，Inspector 部の計算負荷が無視できないことによる．しかし，処理 3 においても，計算量の増加と共に性能向上が確認できる．また処理 4 について，8 台使用時での速度向上率の伸びが低下している問題は，ワーキングセットの計算半径の違いに由来した，各計算機間の負荷の違いによるところが大きい．負荷分散への対応は今後の課題である．

以上のように，逐次処理用に記述されたプログラムと比較しても，VIOS による実行は有効な速度向上が得られると考えられる．

6. ま と め

本論文では，画像処理専用環境であった VIOS を，データ処理一般に対する，抽象度の高いプログラミング環境へとの拡張を行った．そのために，parallel 並列構文の複数次元データへの拡張，分割転送機能，キャッシュ領域の更新命令などの拡張を行った．また，データ分散処理の特徴を活かし，ワーキングセット間の依存関係，データアクセス速度の差より生じる，論理スレッド実行開始可能時間の差を吸収する拡張 parallel 構文を導入し，その記述面，クラスタ環境下における速度面における有効性を確認した．なお，本バージョンの VIOS は，2002 年 3 月に，*http://mars.elcom.nitech.ac.jp/vios* により公開する予定である．

参 考 文 献

1) W. Gropp, E. Lusk, and A. Skjellum: *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, (1999).
2) 湯淺太一, 貴島寿朗, 小西浩: データ並列計算のための拡張 C 言語 NCX, 電子情報通信学会論文誌 D-I, Vol. J78-D-I, No. 2, pp. 200–209 (1995).
3) J. Dongarra and R. C. Whaley: *A User 's Guide to the BLACS v1.1*, UT Tech Report CS-95-281, LAPACK Working Note No. 94, (1995).
4) 松尾啓志, 川脇智英: 分散画像処理環境 VIOS-III, 電子情報通信学会論文誌 D-II, Vol. J84-D-II, No. 6, pp. 955–964 (2001).
5) 田浦健次郎, 米澤明憲: 最小限のコンパイラサポートによる細粒度マルチスレッディング — 効率的なマルチスレッド言語を実装するためのコスト効率の良い方法, 情報処理学会論文誌, Vol. 41, No. 5, pp. 1459–1469 (2000).
6) 多田好克, 寺田実: 移植性・拡張性に優れた C のコルーチンライブラリーの実現法, 電子情報通信学会論文誌,D-I, Vol. J73-D-I, No. 12, pp. 961–970 (1990).
7) L.rauchwerger, N. Amoto, and D. Padua: *Run-Time Methods for Parallelizing Partially Parallel Loops*, Int'l J. Parallel Programming, Vol. 23, no. 6, pp. 537–576 (1995).
8) P. Thomas: *Hardware Compilation of Cellular Automata Algorithms*, Oxford University, School of Engineering and Computer Science, Undergrad. project report, (1999).
9) A. Fujiwara, T. Masuzawa, and H. Fujiwara, *A Parallel Algorithm for the Euclidean Distance Maps*, Technical Report of IPSJ, Vol. 95, No. 10, 1995.
10) C. Guiffaut and K. Mahdjoubi, *A Parallel FDTD Algorithm Using the MPI Library*, IEEE Antennas and Propagation Magazine, Vol. 43, No. 2, pp. 94–103 (2001).