インスペクタ／エグゼキュータ方式によるメタスレッド実行方式

川 脇 智 英[†]　松 尾 啓 志[†]

大規模シミュレーション，自動並列性抽出等においては，大量の同時処理可能単位が発生する場合がある．しかしそれら全てを独立，かつ並列に処理することは，資源／速度／制御の問題より現実的ではない．また並列処理のための制御部を記述することはユーザにとって容易ではない．そこで本論文では，並列処理ソースコードを Inspector/Executor 方式を基礎とした手法を用いて変換することにより，高い記述性を保ちつつ効率的な纏め上げ実行を行う方式を提案する．さらに本手法では，通信遅延の影響が大きな環境において，Inspector 部の事前実行機能を利用することにより，計算による通信の隠蔽効果を図ることが可能となる．

The Meta-Thread Execution by using Inspector/Executor Scheme

Tomohide KAWAWAKI[†] and Hiroshi MATSUO[†]

In large-scale simulation, automatic parallelism extraction, there might be an occurence of large amount of units which could be processed concurrently. However, processing all of the units independently and parallely is not realistic due to the problem of resources/velocity/control. Moreover, it is not easy for the user to describe the control part for the purpose. This paper proposes a new method basically used Inspector/Executor Scheme in parallel language to perform collection efficiently while the high level of descriptiveness is maintained. Moreover, in an environment with large influence of communication delay, the using of prior execution function of the inspector session permits the system to plan the hiding effect of the communication by calculation.

1. まえがき

大規模シミュレーション，マルチエージェントシステム，自動並列性抽出などの分野では，大量の同時処理可能単位（以下セルとする）を扱う場合がある．また記述量の軽減，負荷分散，汎用性を上げることを目的とした細粒度レベルでのプログラミングスタイルに対する研究も行われている [1)2)]．しかし，生成可能な全セルを，実際に独立かつ並列に処理することは，資源/速度/制御の観点から現実的では無く，何らかの手法を用いたセル間の順序付け，あるいは纏め上げを行う必要がある（図 1 参照）．またセル間に依存関係が存在するならば，適切な順で処理を行うことは，実行効率に大きな影響を与えるだけでなく，デッドロックを回避するためにも重要な問題となる．

このような問題への対応には，しばしばスレッドによる実装が利用される．スレッドはプロセスの生成に比べ，高速かつ低資源な生成・実行が可能である．しかし大量のセルが存在する場合，全てのセルをスレッドにより生成することもまた困難となる．

本論文では，このような問題に対し，並列言語処理レベルに Inspector/Executor 方式を組み込んだ手法を提案する．本手法は，ループによる高速な連続処理を基礎とする．そして，ソースコードより一定の規則に従う検査部の生成，事前実行を行うことにより，即時実行が困難な処理を先送りにし，デッドロック発生問題を回避する．また記述面においても，個々のセル間に存在する依存関係を考慮する必要の無い，より抽象度の高い記述を行うことを可能とする．さらに処理先送りの際に，通信要求の纏め上げを行うことにより，通信遅延の大きな環境（例えばネットワーククラスタ）においては，計算による通信遅延の隠蔽効果が得られ，性能向上を図ることが出来る．

2. Inspector/Executor 方式

本手法の基礎となる Inspector/Executor 方式（以下 I/E 方式）について簡単に説明する．I/E 方式は，自動並列性抽出の分野において，動的に決定する配列添字アクセスパターンを解析するために提案された枠

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

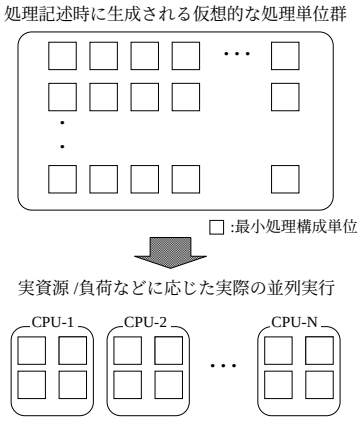

図 1 記述単位と実行単位
Fig. 1 A description unit and an execution unit

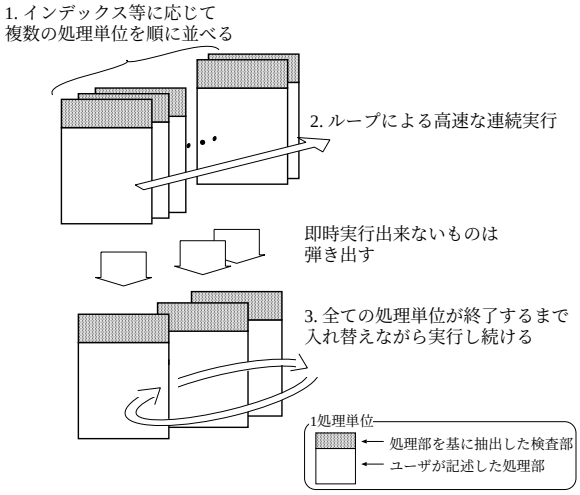

図 2 提案手法の実行方式
Fig. 2 The execution of proposed method

組みであり，現在この枠組みを基とした様々な手法が提案されている [3)4)]．I/E 方式ではまず準備として，ソースコードより Inspector と呼ばれる添字検査部を生成する．そして実行時，実処理部となる Executor 部に先立ち，Inspector 部による検査を行い，得られた情報を並列実行に役立てる．検査は主としてループ実行により発生する，アクセスパターンに対して行なわれる．そのため，パターンのテーブル化・解析がなされ，特定の枠に当てはまるアクセスパターンならば必要なデータ交換，ループ並び替え等が行われる．またいずれにも当てはまらない場合は，別に用意された逐次処理用コードの実行が行われる．

3. 提 案 手 法

本手法では，ユーザに求める記述は，あくまで 1 セルに関するものにとどめ，並列実行に伴う実環境用の制御は内部処理レベルで行うことを目指す．そのために，2 章で示した I/E 方式の枠組みを取り入れる．また，一般的な I/E 方式においては，ループ処理全体の実行に先立ち Inspector 部が実行されることに対し，本手法では 1 セル毎の実行に先立ち Inspector 部の実行を行う．さらに実行部を 2 段に分割する．以上により，デッドロック問題回避，記述における高い抽象度，同期処理挿入の軽減等を可能とする．またパターン解析部や解析のためのテーブルを作成する必要は無く，これに伴う負荷を負うことは無い．

3.1 処 理 方 式

本手法の具体的な処理方式を示す．

準備： コンパイル時，ソース中の並列実行部より 3.2 節に示す手法を元に Inspector 部を抽出する．

ここで，抽出した Inspector 部は，次のように扱う．

- 並列実行部開始前では無く，各セルの実行直前部に挿入する．
- 各セルの実行毎に呼び出され，即座に実行することが可能か否かを調べる．検査は，必要なデータがローカル計算機内に存在するか，受信待ちでブロックする必要があるか，等の情報に基づく．

以上の Inspector 部の挿入を基とし，以降の step を実行する．

step1： 全てのセル（Inspector 部＋ Executor 部）を，インデックス等の単純な情報により並べ，ループに纏め上げて高速に連続実行する．

step2： ループ実行される各セルに対し，Inspector 部終了の段階で即時実行不可能と判明した場合，対象セルの実行を止め，セルを実行キューへ入れる．

step3： 全てのセルに対して一度目の実行，すなわちループ処理が終了した後，次はキューに入れられたセルに対して，同様の処理をキューが空になるまで繰り返す．

本手法の概念図を図 2 に示す．

3.2 Inspector 部

3.2.1 生 成 方 式

Inspector 部はセルの実行に際して，即時実行不可能とする要因の検出を目的として生成される．この際オーバーヘッドの削減，および実行の意味を保持するために以下のルールに従い実行部より抽出される．

検査対象となるソースの各文に対し，

(1) 局所変数アクセスのみで構成される文は，そのまま Inspector 部に加える．

(2) 大域変数への書き込み部を含む文は，書き込み部分を削除し，残りを Inspector 部に加える．
(3) 大域変数からの読み出し部を伴う文は，読み出しの前に検査を行うように書き換えたものを Inspector 部に加える．ただし，検査の必要が無いことがユーザに指定されているものは，そのまま Inspector 部に加える．
(4) ユーザ指定の特定の展開法が記述されている場合はその記述に従う．

(1) は動的に決まるアクセス対象を特定するために必要なルールであり，(2) は実行中の意味の変化を避けるために必要なルールとなる．

3.2.2 オーバーヘッドの削減

Inspector 部導入に伴うオーバーヘッドを削減するために，ルール (3) を適用する．このルールは，データ分散並列処理の特性を使用し，大域変数アクセスについて差別化を行うものである．データ分散並列処理では，各セルは自身が管理するデータ領域を持つ．そこで，異なるデータ配置に対して，異なる関数を経由しアクセスをすることにより効率化を図る．なお，このルールは，ユーザに処理に対する知識を要求するため，付加的なルールとして定義する．

(4) も同様にオーバーヘッド削減のためのルールである．ユーザは I/E が必ず連続して実行される，という本手法の特性を利用して，Executor 部では Inspector 部で実行した処理を行わない，という制御を行うことが可能となる．

3.2.3 生 成 例

図 3，4 に，抽出対象プログラムと Inspector 部の抽出結果をそれぞれ示す．

図 3 において，データ配列 middle および ans は各計算機に分割配置された配列であり，他の変数は各セルが個別に利用する局所変数である．また関数 r() は，ユーザが記述する，ローカルアクセスの可否検査を伴う読み出し関数，focus_*() は検査を行わない読み出し関数である．図 4 で生成されている関数 ro() は，指定データへのアクセスを検査する関数であり，アクセス不可能なら，そのデータを持つ計算機へ取得要求を送信する． また関数 dummy_w() は構文の意味を変化させないために，トランスレータにより暗黙的に挿入されるダミー関数である．

3.3 計算による通信の隠蔽効果

本手法では，各セルレベルでの Inspector 部，および実行部を 2 段階に分けることを利用して，通信遅延の大きな環境下において，データの集合先読みを合わせて行うことが可能である．即ち本手法は計算による通信の隠蔽効果を併有していると言える．

```
1  for(int i=1;i<middle->focus_r(); i++) {
2    if(col-i>=0) {
3      val = sqrt(pow(i,2) +
4             pow(middle->r(row,col-i),2));
5      if(val<ans->focus_r())
6        ans->focus_w() = val;
7    }
8  }
```

図 3 抽出対象プログラム
Fig. 3 Target of extract program

```
1  for(int i=1;i<middle->focus_r(); i++) {
2    if(col-i>=0) {{middle->ro(row,col-i);}
3      if(val<ans->focus_r()){ans->dummy_w();}
4    }
5  }
```

図 4 Inspector 部の抽出結果
Fig. 4 Result of extract inspector

3.4 問 題 点

本手法を導入する並列処理言語では，以下の問題を考慮する必要がある．

- 各セル毎に Inspector 部を導入するため，計算オーバーヘッドが大きくなる．
- 動的な依存関係が多段に渡る場合（他のセルに依存する値に依存するアクセス，等），検査／先送りによる問題点の解消は，1 段ずつとなる．

しかしながら，前者は先に提案したルールにより，後者はプログラムする際，並列化ブロックを複数に分けることにより，対応することが出来るものと考える．

4. 評 価

本手法の評価に際し，分散画像処理環境 VIOS[2] におけるデータ並列型言語 VPE-p を元に実装を行った．分散画像処理環境 VIOS は，細粒度並列処理モデルによる記述を基とする，データ分散型並列処理記述言語 VPE-p，およびその実行環境などからなるシステムである．VPE-p 言語によるプログラミングでは，ユーザまず，データ（＝処理）の最小構成単位を pixel, box 等の指示子を用いて宣言する．そして parallel{} 構文を利用し，各セルに対する処理の記述を行う．この VPE-p を基礎とし，本手法を C++プログラムへのトランスレータとして実装した．大域変数の扱いにはテンプレート機能を利用し，新たな型として実装した．また通信機構，および自動データ分割機構も同様に，VIOS から簡易化して流用している．トランスレータは記述されたソースコードを元に，Inspector 部の抽出，特定クラス関数の置き換え等を行い，結果を C++

コードの中間ファイルを出力する．そして C++コンパイラを通し実行ファイルを生成する．

4.1 記述性の確認

簡単な依存関係を持つ並列処理を例に，本手法の記述性を確認する．使用する処理は，1 次元の配列各要素（セル）に対し，ある同一の処理を並列に行うものである．またそれぞれのセルは，図 5 に示す依存関係が解消されるまで実行出来ない．処理可能領域は，動的に決定する開始点を中心とし，依存関係の解消に伴い左右のセルへ順に広がる．

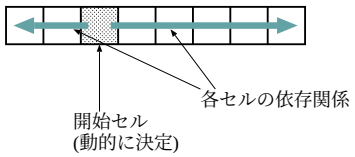

図 5 記述性評価サンプル（1 次元配列データに対して）
Fig. 5 Sample of descriptive evaluation(for 1-D array)

この処理を，複数の計算機を利用し並列に処理する場合，通常，図 6（*）に示す様に開始点の場合分けをする制御構造を含め，記述を行う必要がある．このように，単純な例に対しても，実行時情報（この例では開始点がどの計算機内に存在するか）を考慮した記述は非常に煩雑なものとなる．実行時情報がより複雑になった場合（この例に対してでは，開始点が 2 点以上存在する，配列の次元数が高くなる，等），その記述がさらに困難となることは明らかである．

一般的な依存関係パターンを持つ処理に対しては，ライブラリやテンプレートを用いて対応することが可能である．しかし，通常このような問題は汎用例として定義することが難しく，対応は困難となる．

この問題に対し，本手法を導入することを前提として記述したものを図 7 に示す．図に示される通り，本手法を用いることにより，実環境に依存した制御部を記述する必要が無く，コードは非常に簡潔なものとなる．なおコード中で利用している関数 wait() は，指定セルの実行が終了するまで待つ関数である．

また負荷分散，不確定な近傍定義，などの理由から，実行時に通信を一括処理することが難しい処理が存在する [5)6)7)]．このような場合，その事情を考慮した上でのアルゴリズムの変更が望まれる．アルゴリズムの変更は最適な実行を保証するが，高いプログラミング技術を要求し，かつ処理対象やシステムに関する汎用性を低くする．本手法はこのような問題に対しても対応することが可能となる．これに関しては 4.2.2 節の評価に際して言及する．

```
if(isMyArea(start_point)) {            //(*)
    exec_process(start_point);
    for(int row=start_point+1;
              row < MyMaxRow; row++) {
        exec_process(row);
    }
    for(int row=start_point-1;
             MyMinRow <= row; row--) {
        exec_process(row);
    }
} else
if(isUpperArea(start_point)) {         //(*)
    wait(MyMaxRow+1);
    for(int row = MyMaxRow;
             MyMinRow <= row; row--) {
        exec_process(row);
    }
} else {                               //(*)
    wait(MyMinRow-1);
    for(int row = MyMinRow;
              row < MyMaxRow; row++) {
        exec_process(row);
    }
}
```

図 6 通常手法で記述したコード
Fig. 6 Code of descripted by usually method

```
parallel{
    if(start_point != row) {
        if(start_point  < row) {
            flag->wait(row+1);
        } else {
            flag->wait(row-1);
        }
    }
    exec_process(row);
}
```

図 7 本手法を用いて記述したコード
Fig. 7 Code of descripted by proposed method

4.2 実装評価

本手法の有効性を確認するために，3 つの異なる特性を持つアルゴリズムの記述を行い，評価した．なお全てのアルゴリズムは，界面通信の統合など，効果的と思われる代替手法が存在するものとし，比較対象プログラムは通常考えられる最適化を手動で行うものとした．またデッドロックが発生しうる処理に対して，従来手法では実装するアルゴリズムに特化した手法で記述を行うか，またはスレッド等により実行単位ごとに処理しなければならず，これによる記述面，または実行面での負荷が大きいことは明らかである．

1 つ目のアルゴリズムとして，Laplace 法を利用し

た 2 次元温度弛緩シミュレーションを用いた（以下アルゴリズム 1 と略す）．この処理は，各セルの近傍定義を知ることにより，静的に最適な通信が記述可能な処理である．さらに 1 セル当りの毎ステップ計算量は比較的小さく，提案手法にあまり向かない処理であると考えられる．なお比較用プログラムは，データ分割界面に対し一括通信を明示的に記述するものを用いた．

2 つ目のアルゴリズムとして，セル・オートマトンの 1 つであるガス拡散シミュレーション [7] を用いた（以下アルゴリズム 2 と略す）．この処理は，各セルの近傍定義が状態毎に変化するため，静的に最適な通信を決定することは困難な処理である．しかし近傍数は少ないため，使用する可能性のある近傍全てを一括通信することにより対応できる．またアルゴリズム 1 と比較し，1 セル当りの計算量は若干大きい．比較用プログラムは，使用する可能性のある領域の一括通信を明示的に記述した．

3 つ目のアルゴリズムとして，Hirata と Kato のアルゴリズム [6] を使用した，2 次元ユークリッド地図法を用いた（以下アルゴリズム 3 と略す）．この処理では，動的に生成される中間地図の値に依存して，各セルが参照する近傍量が変化する．また 1 セル当りの計算量も比較的大きく，提案手法を導入することによる負荷の割合も低い．比較用プログラムは，動的な情報が確定した後，その値の統合・再分割を行い，処理を継続するよう記述した．なお今回の評価では，負荷分散はその評価範囲ではないため，物体点は 1/20 の確率でランダムに均等配置した．

4.2.1 環　　境

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環境下にて行った．

4.2.2 結果および考察

図 8，9 に評価結果を示す．それぞれ左からアルゴリズム 1,2,3 の順に並べたものである．またグラフ中，横軸は計算機台数，縦軸は台数効果を示し，各処理ともに空間の大きさを変化させた 3 つの場合に対して評価を行った．基準となる 1 台による評価結果（図中縦軸単位）は，提案手法部および通信部，領域外判定部等，並列化に関する余分な処理を全て取り除き，CPU1 台での実行用に，別途記述した．

4.2.2.1 記述性に対する考察

本手法を用いて記述した場合，プログラムはいずれの場合も，独立した 1 セルに関する処理を記述する事と，ほぼ同等の記述のみで実装することが可能であった．また 3.2.2 節で示した，オーバーヘッド削減のための拡張記述も，あくまで 1 セルの実行，すなわち 1 フローに関して考慮するものであるため，セル間にわたる相互関係を考慮することよりも容易であると言える．比較用コードでは，いずれも全体の実行フローを視野に入れ，分割方式，分割部などを意識した通信コードを記述することが必要であった．また同期処理を行うタイミングに関しても，より多く考慮する必要があった．結果，ユーザ記述部のソースサイズは，本手法を用いることにより，数 10%から，通信制御コードを含めた場合は数分の 1 まで減少した．

表 1　提案手法および通信処理を加えたことによる速度低下率
Table 1　The speed decreasing rate by the proposal method and communications process

algorithm #	1	2	3
overhead(%)	88	41	5

4.2.2.2 通信の隠蔽効果に対する考察

まず手法導入による，逐次処理用プログラムからの速度低下率について考察する．はじめに，提案手法の導入，および並列処理のために必要な通信処理を付加したことによる，各アルゴリズムの速度低下率を表 1 に示す．なおここでの空間サイズは 2048x2048 セルである．またこの速度低下率には，通信処理を行うための前処理，同期処理等も含まれる．結果，速度低下率は，88%から 5%までの広い値を取ることになった．

これは，アルゴリズム 1 の様に処理そのものの計算量が小さいものでは，Inspector 部としてプログラムに追加されるローカル変数演算，領域判定演算などの割合が，全実行に対して非常に大きなコストになることに対し，アルゴリズム 2，3 では，その多くが生成ルール 3，4（3.2.1 節参照）により Inspector 部には存在しないため，全実行に対する導入コストが小さくなることによると考えられる．この結果より，本手法は，適応範囲が，検査対象を利用した複雑な演算を主としている場合に効率良く働くものと考えられる．

次に計算による通信の隠蔽効果について考察する．アルゴリズム 1 に対して，提案手法は比較プログラムと通信量においては同等であり，さらに通信処理に前後した同期処理が必要ない，という利点が存在する．しかし，演算処理そのものが単純であるため，Inspector 部の導入負荷が，実行時間に影響を及ぼす結果となり，全ての空間サイズにおいて比較プログラムより低い性能となった．それに対しアルゴリズム 2 では，上記同期処理に関する利点に加え，通信量に関する利点が存在する．さらに Inspector 部の導入負荷も低めであるため，1024x1024 のセル空間では 1.4 倍程度の性能向

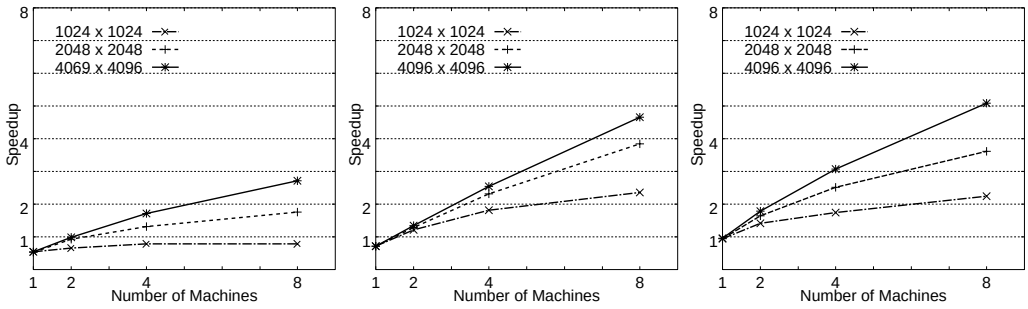

図 8 提案実装の実行結果
Fig. 8 Result of Gas-Diffusion

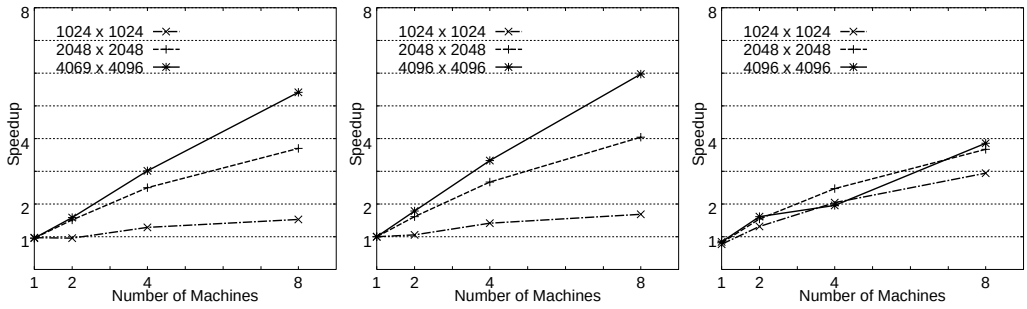

図 9 比較手法の実行結果
Fig. 9 Result of relax method

上が，2048x2048 のセル空間でもほぼ同等の結果が得られた．しかしこの問題においては，通信時間は空間サイズ n^2 に対して n のオーダで影響するが，計算時間は n^2 のオーダで影響する．このため，4096x4096 のセル空間では，Inspector 部の影響が無視出来なくなり，提案手法の速度向上は低くなる，という結果となった．アルゴリズム 3 は，上記利点がさらに大きく影響するため，全ての空間サイズに対して本手法が有効に機能するという結果となった．

以上の結果より，本手法における通信の隠蔽効果の有効性が確認できた．また特に，通信が全実行時間に影響を持ち，かつ計算量がその通信を隠蔽するために十分な量である場合，この効果はより有効に機能すると言える結果が得られた．

5. むすび

本論文では，大規模処理単位生成問題に対して，デッドロック問題を回避し，且つ高い抽象度を保つメタスレッド化手法について提案した．今後の課題としては，さらなるオーバーヘッドの削減，制御構文の拡張，負荷分散の考慮，等について検討している．なお，本手法を導入した新しい分散画像処理環境 VIOS は 2002 年 3 月 http://mars.elcom.nitech.ac.jp/vios にて公開予定である．

参考文献

1) 大山 恵弘, 田浦 健次朗, 遠藤 敏夫, 米澤 明憲: 細粒度スレッド生成をサポートする言語の共有メモリ並列計算機上での実装とその性能評価, *SPA '98 Proceedings* (1998).
2) 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).
3) 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).
4) C.-Z. Xu, V. Chaudhary: *Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences*, IEEE Trans. Parallel and Distributed Systems, Vol. 12, no. 5, pp. 433–450 (2001).
5) 片桐孝洋, 金田康正: 超並列処理に向く効果的な並列固有値計算法, 情報処理学会論文誌, Vol. 41, No. 5, pp. 1558–1566 (2000).
6) A. Fujiwara, T. Masuzawa, and H. Fujiwara, *A Parallel Algorithm for the Euclidean Distance Maps*, Technical Report of IPSJ, Vol. 95, No. 10, 1995.
7) P. Thomas: *Hardware Compilation of Cellular Automata Algorithms*, Oxford University, School of Engineering and Computer Science, Undergrad. project report, (1999).