平成13年度　　修士論文

メタスレッドを導入した
分散処理環境VIOS IVの開発

指導教官
岩田　彰　　教 授
松尾　啓志　助 教 授

名古屋工業大学大学院　電気情報工学専攻
平成12年度入学　12417043番

川 脇　智 英

目 次

第 1 章　はじめに **1**

第 2 章　VIOS システム構成 **4**

第 3 章　VIOS プログラミングモデル **6**

3.1　実行フロープログラム 6

3.1.1　データの細分割転送 8

3.2　モジュールプログラム 11

3.2.1　ワーキングセット 11

3.2.2　並列処理構文 12

3.2.3　多次元キャッシュ領域の一括更新命令 14

3.2.4　依存型リダクション 15

第 4 章　I/E 法を用いた並列処理構文の拡張 **18**

4.1　Inspector/Executor 方式 20

4.2　提案手法 21

4.2.1　手法の流れ 22

4.2.2　Inspector 部生成法 24

4.2.3　特徴のまとめ 26

4.3　拡張並列構文 28

4.3.1　検査対象外指定構文 28

第 5 章　評 価 **30**

5.1　拡張並列構文 30

5.1.1 アルゴリズム記述能力 30
5.1.2 実処理に対する記述能力 32
5.1.3 計算による通信の隠蔽効果 37
5.2 分散処理環境 VIOS 41
5.2.1 アルゴリズム記述能力 41
5.2.2 クラスタ環境下における速度評価 42

第 6 章 関連研究 **45**

第 7 章 まとめ **47**

謝辞 **48**

参考文献 **49**

Appendix A VIOS 実行フロー記述言語 VPE-i 仕様 **52**
A.1 設計方針 52
A.2 予約語 53
A.3 利用可能な型 53
A.4 通信環境設定用構文 54
A.4.1 プラグマによる指定 54
A.4.2 Set 手続きによる指定 55
A.5 その他 55
A.5.1 並列モジュール実行後の変数の扱い 55
A.5.2 VPE-p への手続きの追加方式 56

Appendix B 並列処理モジュール記述言語 VPE-p 仕様 **57**
B.1 はじめに 57
B.2 モジュールプログラムファイル構成 57
B.3 モジュールプログラミング言語 VPE-p 58
B.3.1 拡張構文 58

B.3.2 VIOS 変数 60
B.3.3 特殊関数 60
B.3.4 全体に関する制限 64
B.4 その他 65
B.4.1 細分割実行の効果に関して 65
B.4.2 モジュールファイルのコンパイル 65

第1章

はじめに

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．

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)[2] 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[3]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[4]．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[6]を，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的とし，次の4つの要因を総合的に取り入れた環境の構築を目指す．

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4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．

- リダクション命令を拡張した，依存型リダクションの導入．
- 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．

以降の章では，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 における基本プログラミングモデルと，データ並列処理全般を対象とするために拡張した，幾つかの命令について示す．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拡張並列構文について示す．そして 5 章でそれまでの章において述べた拡張を利用することによる，VIOS のアルゴリズム記述能力，およびクラスタ環境下における速度向上率の両面に対して評価を行う．最後に 6 章において関連研究について述べ，7 章でまとめる．

第2章

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たデータ分散並列処理環境であり，図 2.1 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．

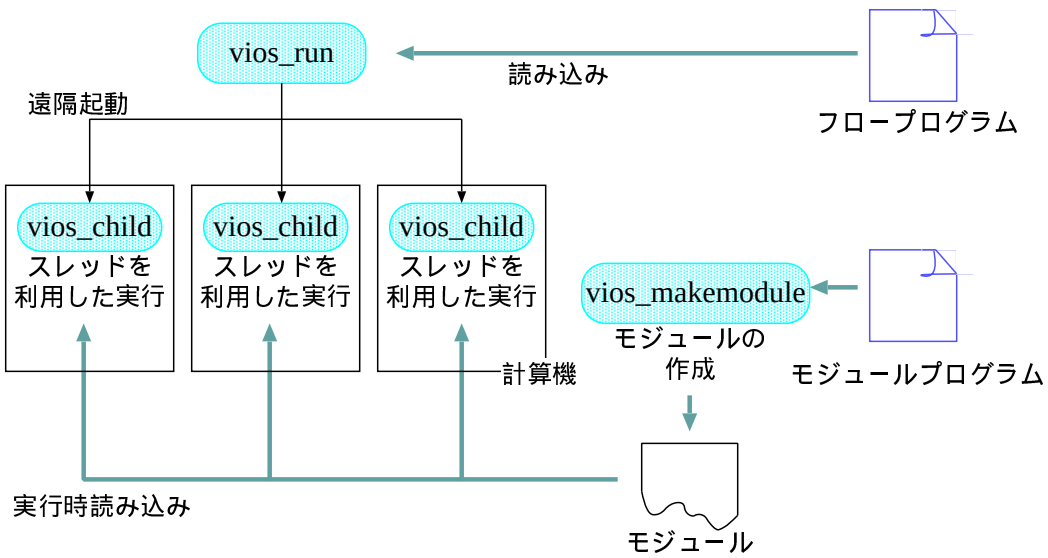


図 2.1: VIOS システム構成

以下，これらプロセスについての説明する．

- vios_run は，実行フロープログラム (3.1 節参照) 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と

なる．

- vios_child は，各計算機上に分散配置され，並列処理の本体となるモジュールプログラム (3.2 節参照) を実行するサーバプロセスとなる．また，モジュール実行は，主演算実行部，通信処理部，それぞれについてマルチスレッドを用いて実行される．

- vs_makemodule は，VPE-p (3.2 節参照) により記述された並列実行モジュールを，VIOS 実行部が読み込む動的モジュールへと変換するためのスクリプトとなる．スクリプト内部では，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++コンパイラの呼び出しなどを行う．

以上のプロセスが，ネットワーク上の各計算機において通信を伴いながら処理を行うことにより，VIOS は並列処理を行う．

第3章

VIOSプログラミングモデル

この章では，並列処理環境 VIOS による並列処理プログラミングについて説明する．VIOS による並列処理プログラミングは，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．また両プログラムの関係は，図 2.1 に示す通りである．

以降では，まず実行フロープログラミングについての説明を行う．

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，これら逐次処理部は，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の必要ない，環境への柔軟な対応を可能としている．

図 3.1 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．

実行フローでは，まず図 3.1 (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，データ分割転送 (3.1.1 節参照) を利用するための指示も行う．

そして通信環境の設定の次に，図 3.1 (3a) (3b) に示すように，大域データの宣言

```
// 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
#pragma divide_num 2                   ← ( 2 )
#pragma connect_start

/* 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 3.1: メインフロー記述言語 VPE-i による記述例

や，初期化，読み込み部の記述を行う．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．

以上の初期化処理が終了した後，並列処理モジュールの実行命令を発行する (図 3.1 (5)) ．この並列モジュールの実行は，“module” 命令により行う．またその前に，各

データに対し，並列処理ワーキングセット (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 3.1 (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 に対して並列処理を行う場合，各ワーキングセットは半径 1 の周辺参照を行うことを指示している．

以上のように，実行フロープログラムは，複数の並列処理部を含めた，ある一連の処理の流れを記述するプログラムとなる．

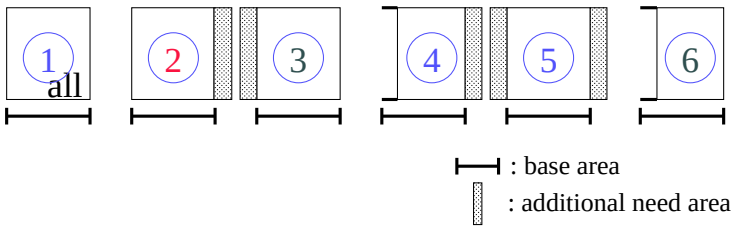
以降では，実行フロー部において指定することが可能な，VIOS における特徴的な機能について説明する．

3.1.1 データの細分割転送

データ並列処理では，処理開始時などにおいて，大規模なデータ通信が避けられない場合がある．そのような場合， 領域を計算機台数よりさらに細かく分割し，小刻みに転送・計算を繰り返すことにより，計算と通信のオーバーラップ，各計算機に対する処理時間の均等化を図ることが多い (以降では，この機能を細分割転送と定義する) .

しかし，画像や空間情報などの並列処理に対し細分割転送を適用する場合，次のような問題を考慮する必要が生じる．それは，上記のような処理では，データは単純に分割配置されるだけでは無く，各計算機は他計算機が管理する領域をキャッシュ領域として一定量保持することが多いため[7)8)9)]，細分割転送時にもキャッシュ領域を考慮しなければならない問題である．そして，プログラマは図 3.2 上のような場合分けを考慮する必要があり，このことは大きな負荷となる．キャッシュ領域については別領域として持つ，という手法も考えられるが，処理の際，全てのセルの処理に条件式として上乗せされるため実行効率が良くない．また，データの分割転送に関しては，文献[10)] などによりすでに定式化されているが，キャッシュ領域も合わせて考慮したものは，我々が調べた限りでは現状見受けられない．

そこでまず，1 次元データ分割に対して，キャッシュ領域を考慮した分割領域のパターンの解析を行った (図 3.2 下) . そしてその結果を元に，複数次元データに対しても，以下に示す手法を用いることにより，任意の次元のデータに対する細分割転送に対応することを可能とした．



host_num / divide_num	1	2	over 3
1	1	id 0: 2 / id 1: 3	id 0: 2 / ···· : 5* / id last: 3
2	id 0: 2 / id last: 6	2 4 / 5 6	2 4 / 5 4 / 5 6
over 3	2 4* 6	2 4* 4 / 5 4* 6	2 4*4 / 54*4 / 5 4* 6

図 3.2: 1 次元データに対する，キャッシュ領域付き細分割転送時の領域パターン (上) と，分割数・計算機台数による発生パターン (下)

1. 各次元の軸に対し，個別に分割パターンテーブルを作成する．各テーブルには，データ開始アドレス，およびデータ長を格納する．
2. まず各分割領域にインデックスを付ける．ただし，このインデックス番号は，図 3.3 に示すように，C 言語における配列のインデックスと同様の順に従うものとする．このように割り振られたインデックスの元で，分割領域 id に対し，式 3.1 ~ 3.3 に従い，各次元のインデックスを取得する．

$$x_{table}[id \ mod \ x_{div}] \tag{3.1}$$

$$y_{table}[(id \ mod \ (x_{div} \times y_{div})) \ / \ x_{div}] \tag{3.2}$$

$$x_{table}[id \ / \ (x_{div} \times y_{div})] \tag{3.3}$$

このように定式化することによりまた，各分割領域個別に範囲情報を取得することが可能となる．このことは，各分割領域を個別に扱うことを可能とし，通信時における並列転送や，負荷分散に利用することも可能とする．

この手法を利用し，VIOS では，以下の様に宣言することにより，データの細分割転送をシステム側により自動的に行う (図 3.1 (2)) .

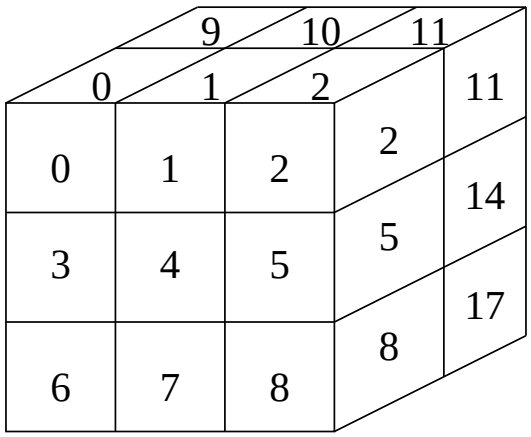


図 3.3: 分割領域と対応する id (3 * 3 * 2 の 18 領域に分割した場合)

```
#pragma divide_num num
```

なお，*num* には 1 計算機あたりの分割数を指定する．

図 3.1 の例では，各計算機に対し，分割数を 2 と指示している．この指示により VIOS では，それぞれの引数に対して，上記手法を元にキャッシュ領域や重複転送を考慮したデータの分割・転送を行う．また各計算機は，細分割データ 1 ブロックを受け取るごとに，非同期に受信ブロックに対してモジュール実行を行う．

3.2 モジュールプログラム

次に，モジュールプログラムについての説明を行う．

モジュールプログラムは，VIOS により実行され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 を利用し，その記述は行う．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，などの制約は存在するが，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．

VPE-p を利用して記述されたプログラムは，実行時メインフロー部より，以下のように宣言されたモジュール構文が呼び出されることにより実行される．

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 (VIOS 変数という形をとる) 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拡張構文を用い，並列処理の記述を行う．

それでは次に，この並列モジュールを記述するための基本概念となるワーキングセットについて説明する．またワーキングセットは，VIOS における並列処理実行に対する基本構成にもなっている．

3.2.1 ワーキングセット

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[11] の場合，母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，それぞれ無限に生成可能な論理スレッドと 1 対 1 に対応させ並列処理を行うモデルを採用している．また 1 ワーキングセットを構成するデータ構造は，3.2.2 章で述べる並列構文，およびメインフロー部でのセット関数により様々なタイプから選択することが可能である．

このようにして定義したワーキングセットに対し，以降に示す構文を用いて処理を記述することが，VIOS における基本的な並列処理スタイルとなる．

3.2.2 並列処理構文

VIOS では，任意のワーキングセットに対する処理を，以下に示す parallel 構文を利用し記述することにより，全ワーキングセットに対する処理を並列に行うプログラミング，および実行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 */ }
```

まずこの parallel 構文に対し，データ処理全般，特に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新たに行った．

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

以降では，この並列処理構文についての説明を順に行う．

まず引数について，< および > に囲まれた，第 1 引数となる 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す．これは，あるデータ “ブロック” を 1 ワーキングセットとして扱う場合に利用する．

次に，(および) に囲まれた，第 2 引数となる $index$ 変数は，並列処理中，各ワーキングセットが自身を特定するための変数名を指定するもので，任意の名前 (事前に宣言されている必要はない) が利用可能である．

また これら “引数の数” は，モジュールが複数持つ部分データ中の，どのデータを元として並列処理を行うかの指示となる．なお，モジュール中に同じ次元数のデータ

が複数存在する場合は，モジュールの最初の引数の情報に従う．

次に，これらの引数についての具体例を示す．

$parallel$<2,3>(x,y){ .... } と記述した場合を考える．この場合は，各引数は 2 つづつなので，並列処理は 2 次元の部分データに対し行われる．また，第 1 引数より，部分データは，2x3 のブロックを 1 ワーキングセットして並列実行され，さらに第 2 引数より，各ワーキングセットが自身を特定するために利用する変数名は x, および y となる．

以上に示した，新しい parallel 構文を用い，VIOS モジュールを記述した具体例を，図 3.4 に示す．なお，このプログラムは 5.1.2 節における評価の際に用いるものであり，アルゴリズムの詳細についてはそちらで示す．

```
//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 3.4: Jacobi 処理に対する VIOS モジュール

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．この際，第 1 引数である <> は省略されているが，この場合 $< 1, 1 >$，つまり 1 要素に対して 1 ワーキングセットが割り当てられるものとなる．

また，parallel 構文中において，各ワーキングセットは，従来の VIOS[6] と同様に，“[][]” アクセス方式を取ることにより，相対的な位置関係を利用して，他のワーキングセットが保有するデータ領域を参照することが可能となる（図中（2）参照）．

3.2.3 多次元キャッシュ領域の一括更新命令

次に，本 VIOS より新たに追加された拡張命令の 1 つである，多次元キャッシュ領域の一括更新命令について説明する．

Jacobi 法，電磁界シミュレーションなどの処理[9] 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（図 3.5）．このような処理の場合，1 時間ステップごとに，全てのデータの更新・並列処理の反復，を空間が収束するまで続けるため，他計算機の管理するデータのキャッシュは，1 時間ステップごとに無効なものとなる．

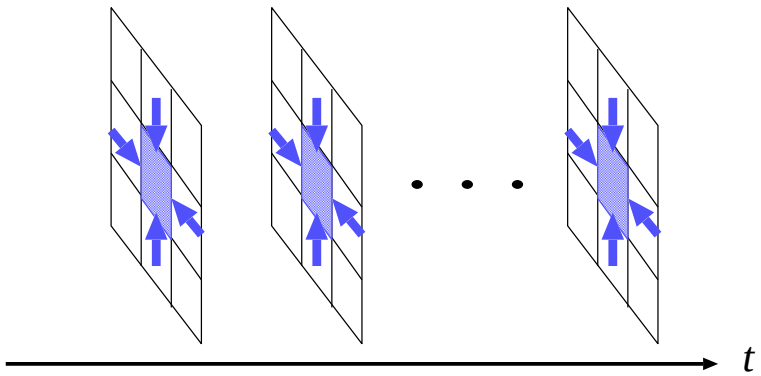


図 3.5: Jacobi 処理におけるデータの更新

従って毎ステップごとに，キャッシュ領域を更新する必要があるが，各キャッシュ領域の位置と，実データを保持する計算機の関係は，データ形式，キャッシュの定義，使用する計算機台数などの実行時パラメータに依存し複雑に変化する．

例えば，3 次元データ，キャッシュ領域は周囲 6 方向，計算機台数 4 台，に実行時パラメータを限定した場合でも，各計算機それぞれを場合分けした上で，3 つのデータ分割境界面に対するアドレスおよび範囲計算，通信処理，同期処理を行うための記述が必要となる．さらに上記の各パラメータに対し汎用性を考慮するならば，プログラ

マにかかる負荷は非常に大きなものとなる．

そこで VIOS では，これら実行時に決定される情報に基づいたアドレス演算と通信処理を外側には 1 命令として行える様に実装し，以下に示すキャッシュ領域の一括更新命令として新たに導入する．

vsSyncCache(*VIOS* 変数);

図 3.4 (3) に使用例を示す．この命令により，プログラマは，計算機台数などの各種パラメータに依存しない，汎用性の高いプログラムを容易に記述することが可能となる．特に，通常扱うデータの次元数が高くなるにつれ複雑化する境界通信の隠蔽は，空間処理に対する記述性を大幅に高くするものでとなる．また，メインフロー部により指定される，実行時パラメータから独立した記述であることは，それらの変更に対しての再コンパイルを要求しないというメリットも存在する．

3.2.4 依存型リダクション

次に，VIOS が提供する，記述性を高めるための特殊命令について説明する．

各計算機により実行される，モジュール中での管理外変数へのアクセスは，各プロセッサ間の通信遅延が膨大なネットワーク分散環境においてはシステムのボトルネックになる．また，画像処理，認識アルゴリズムなどで利用される大域変数には，各計算機ごとに，その複製を局所変数とみなして実行し，それぞれの結果を結合することにより処理を進めることができる，すなわち大域変数へのアクセスが結合則により成り立つものも多い．

そこで VIOS では，このような特性を利用し，各計算機ごとに指定された大域変数にの複製を持ち，ユーザが明示的に複製の値の統合処理を指定することにより，大域変数へのアクセスの高速化を実現する，いわゆるリダクション処理[12] を導入している．

しかし，テンプレートマッチングのようなボーティング処理，また粒子シミュレーション[9] などでは，各計算機共通の大域変数に代入される値の最大 (最小) 値ではなく，その値を最大 (最小) にするパラメータが必要となることがある．このような，最大値 (最小値) に付属した値，を取得することは，結合演算のみしか定義されていな

いリダクションでは行うことが出来ない．そのため通常，このような機能を実装するためには，プログラマによる新たな大域変数の宣言，排他制御の挿入，など煩雑な処理の記述が必要となる．

そこで VIOS では新たに，リダクション処理における結合ルールに，以下のような依存型ルールを導入した．

vsUnion(*VIOS* 変数, vs{MIN|MAX}_DEPEND_ON_FIRST);

第 1 引数でリダクション対象となる大域変数を，第 2 引数でその結合ルールを指定する．従来，第 2 引数には，vsMAX, vsMIN, vsADD などの予約語を利用した結合則の成り立つルールのみであったのに対し，予約語 vs{MIN|MAX}_DEPEND_ON_FIRST を新たに追加する．なお，予約語 vs{MIN|MAX}_DEPEND_ON_FIRST を指定するリダクション処理では，対象となる *VIOS* 変数は，配列変数に限定される．

実際に行われる処理では，まず最初の要素に対し，通常のリダクション処理が行われる．そして，第 2 以降の要素に関しては，第 1 引数を採用したワーキングセットの値が採用される．

図 3.6 を用いて，さらに依存型リダクションの説明を行う．

図 3.6 は，テンプレートマッチング処理を，VIOS により実装したものである．モジュール template_matching は，実行フローより，探索画像 data，テンプレート画像 temp，結果を格納するための配列 result を引き継いでいる (図中 (1)) . また図中 (2) より開始される，*parallel* 構文では，3.2.2 節において示した通り，部分 2 次元配列 data の 4×4 の領域を 1 ワーキングセットとして並列処理が行われる．なお，このアルゴリズムの詳細は，5.1.2 節における評価の際に述べるが，おおよその処理としては，探索画像の各ポイントを中心にテンプレートマッチングが行われる．そしてその誤差と大域変数 result の先頭要素を比較し，より小さければ誤差，自ワーキングセットの id (=探索画像中の位置) を新たに result に挿入し，処理は終了される．なお，ここまでの処理は，各計算機独立の行われる (result は各計算機が複製を持つことになる)．

そして最後に，各計算機間においても，依存型リダクションを利用することにより，最終的に result の先頭要素には，マッチング結果において最も小さな誤差の値が，2,3

```
vs_module
template_matching(iImage data, iImage temp, ivalue result)    ← ( 1 )
{
    int o_width = orig_width(temp)/2;
    int o_heigh = orig_heigh(temp)/2;

    // 全比較が可能な領域以外はマッチングを行わない
    vsSetBoundary2D(max(o_width, o_heigh));

    parallel<4,4>(x,y) {                                        ← ( 2 )
        int sum = 0;
        for(int j=-o_heigh; j<o_heigh; j++)
            for(int i= -o_width; i<o_width; i++)
                sum += abs(data[+i][+j] \
                            - temp[[i+o_width]][[j+o_heigh]]);

        if(result[[0]] > sum) {                                 ← ( 3 )
            result[[0]] = sum;
            result[[1]] = l2g_x(data, x); // アドレス変換関数
            result[[2]] = l2g_y(data, y); // アドレス変換関数
        }
    }
    // 各計算機の結果を統一する
    vsUnion(result, vsMIN_DEPEND_ON_FIRST);                     ← ( 4 )
}
```

図 3.6: テンプレートマッチング処理に対する VIOS モジュール

番目の要素にはその座標が代入される (図中 (4)) . 通常 , この記述は煩雑なものとなるが , 図中 (4) に示すように , 依存型リダクションを使用することで , 特定のパラメータを取得する問題を効率よく記述することが可能となる .

第4章

I/E法を用いた並列処理構文の拡張

この章では，ワーキングセット間に依存関係が存在する場合に対応するための，並列処理構文の拡張について述べる．

3.2.1 節で述べたように，VIOS では，実行時に膨大な数として生成されるワーキングセットを論理スレッドとして実行する．このように，理論上多くのスレッドを生成することは，記述の抽象化，コーディング負荷軽減を図れるのみでなく，文献[13] により，“並列性能の観点から，分散メモリ計算機上でのデータ分割のブロック幅と，ブロック化アルゴリズムのブロック幅は切り離して並列アルゴリズムは設計されるべき” と指摘されている通り，さらには負荷分散を行うためにも，有効な手段と言える．しかし，生成される論理スレッド全てを，実際に独立，かつ並列に処理することは，資源/速度/制御の観点からも現実的でなく，事実上何らかの手法を用いた実行順序の決定や，纏め上げての処理が必要となる．そしてこの際，各論理スレッド間にデータ・制御依存が存在するならば，適切な手法用いた対応を行わなければ，深刻な速度低下だけでなく，デッドロックを引き起こす可能性もある．

そこで，このような問題に対しては，様々な手法が従来より提案されている[5)4)14)]．

多田[4)] らは，longjmp 命令を用いることにより，コンテクスト切り替えにより発生するページ切り替えを抑制し，プログラム実行の一時中断，再開を扱うことの出来る，疑似スレッドを利用する手法を提案している．また Cilk[14)]，および StackThreads[5)] では，再帰並列の特性を活かし，関数スタックにコンテクストを保存することをベース

とした手法を提案している．

しかし，これら手法は，スレッドの軽量な生成，実行に焦点を当てているものであり，中断・切り替えに関してはプログラマの明示的な記述を要求する．このことは，記述することに対する負担のみでなく，動的に決定するデータ分配方や論理スレッドの依存関係による実行の中断，再開には対応することが難しい．また，longjmp におけるコンテクスト保存領域や関数スタックは，1 スレッド中での利用変数の多いデータ並列言語では，依然として大量のメモリ領域を保存する．

従来の VIOS[6] では，ユーザが生成した大量の論理スレッドは，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しているが，この手法は画像処理特性を考慮したものであるため，並列処理一般を考慮する場合には，上記のように論理スレッドの中断と再開が問題となる．

本システムで扱う，データ並列処理における論理スレッドの特徴をまとめると，次のようになる．

1. 依存関係が単純に実行時生成順とならない場合がある．
2. その生成数は，一般にデータの大きさに比例するため，擬似的にでも全てをスレッドとして扱うことは，切り替え速度，コンテキスト保存領域などの問題より，依然として難しい．特にデータ並列処理の場合，コンテクスト保存領域は大きくなる傾向がある．
3. SIMD 型の実行形式を取る
4. 1 つの論理スレッドが，多数の論理スレッドの生成データに依存することがある．
5. 分散メモリ環境下を想定するため，通信遅延も依存関係に含める必要がある．

そこでこの問題を考慮するために，Inspector/Executor 手法[15] (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提案，導入する．

4.1 Inspector/Executor 方式

本手法において基礎とする，Inspector/Executor 方式（以下 I/E 方式）についてまず簡単に説明する．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[15)16)17)]．

一般的な I/E 方式では，まず準備として，ソースより 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，アクセスパターンの検出を目的とする．アクセスパターンのテーブル化・解析を行い，特定の枠に当てはまるアクセスパターンならば，必要なデータのプリフェッチ・ループの並列処理等が行われる．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．

I/E 法の実行フローを図 4.1 に示す．

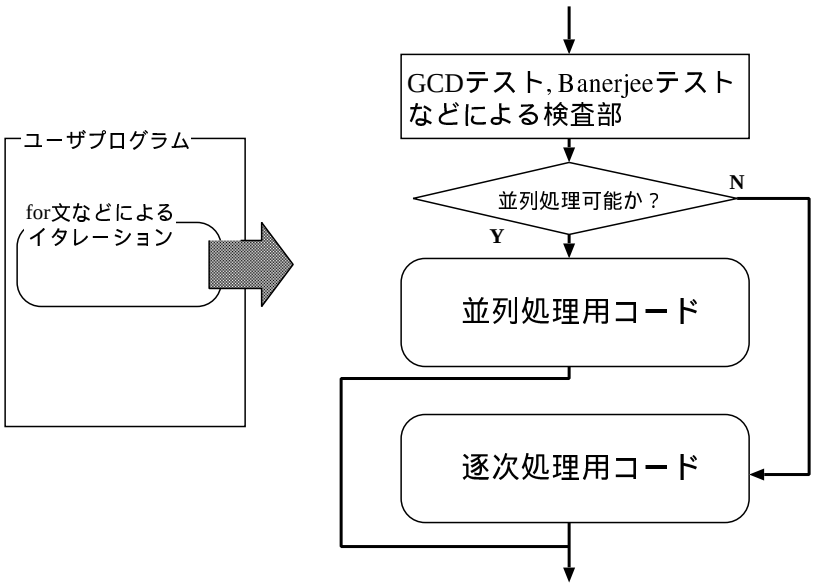


図 4.1: 一般的な I/E 法

4.2 提案手法

VIOS では，上記問題を解消することに加え，ユーザに求める記述は，あくまで 1 ワーキングセット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，4.1 節で述べた I/E 方式の枠組みを取り入れる．しかし，本手法は，以下の点において一般的な I/E 法と異なる．

- 通常，I/E 法では，ループ処理を対象とし，かつ注目ループ全体に対する Inspector 部を用意することに対し，本手法では各ワーキングセットに対し，Inspector 部を用意する．
- 各ワーキングセットレベルでの I/E 方式による判定に加え，全ワーキングセットレベルでのスケジューリング部を用意する．

以下では，これら違いを含めて，具体的な提案方式について説明を行う．

VIOS によるワーキングセットは，個々が独立した並列処理を行う事を前提とする．実際，幾つかの処理[7)8)] に対しては，ワーキングセットを固定的な順で処理するようにすると，深刻な速度低下やデッドロックを招く．すなわち，ワーキングセット群の並列処理はループ処理の並列化ではない．そのため，一般 I/E 法のように，記述されたものをそのまま実行することで逐次処理とすることも出来ない．さらに，先に述べたように，全ての論理スレッドの生成，およびコンテクストの保存も現実的ではない．

そこで，本手法では，個々のワーキングセットレベルで Inspector 部を付加し，実行時，その時点で中断されることなく最後まで実行出来るかを検査する．すなわち，その時点で中断，コンテクストの切り替えが必要となる論理スレッドは始めから実行を開始しない．そして，実行出来ない，と判断された論理スレッドの先送りを行うため，全体のレベルでのスケジューリング部を用意する．

このような方式をとることにより，本手法はまた次のような特徴も持つ．

まず，Inspector 部により，ワーキングセット全体に対し検査を行うため，検査結果として，実行をブロックする要因全ての情報を得ることができる．よって，例えばブロック要因がネットワーク外データの参照である場合，検査時に全ての要素に対する

受信要求を行うことにより，ブロック要因の解消を纏めて行うことができる．さらに，ブロック要因の数を利用してスケジューリングを行うことも可能となる．

また，一般 I/E 方式で行う，flow, anti, output 等のパターン解析部や解析のためのテーブルを作成は行わないため，これら検査を可能とするために通常付加される，記述上の制限も緩和される．

4.2.1 手法の流れ

本手法の具体的な処理の流れを図 4.2 に示す．また，以下では具体的な実装方式について述べる．

準備： コンパイル時，ソース中の並列実行部より，まず，各ワーキングセットに対して検査を行うための Inspector 部を，4.2.2 節に示す方式により抽出する．抽出した Inspector 部は，以下のように処理される．

- 並列実行部開始前では無く，各ワーキングセットの実行直前部に挿入する．
- 各ワーキングセットの実行毎に呼び出され，現時点で実行することが可能か否かを調べる．検査は，注目ワーキングセットが実行される際，必要な資源が全てローカルにそろっているか，他のワーキングセットからの受信待ちはあるか，などの情報に基づく．

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

step1： 全てのワーキングセット (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 (メタスレッド実行)．

step2： ループ実行される各ワーキングセットに対し，Inspector 部終了の段階で即時実行不可能と判明した場合，対象ワーキングセットは実行を開始せず，ワーキングセットの ID を実行キューへ入れる．なおこの際，必要となる値の要求など，先に行うことが可能な処理は実行しておく．

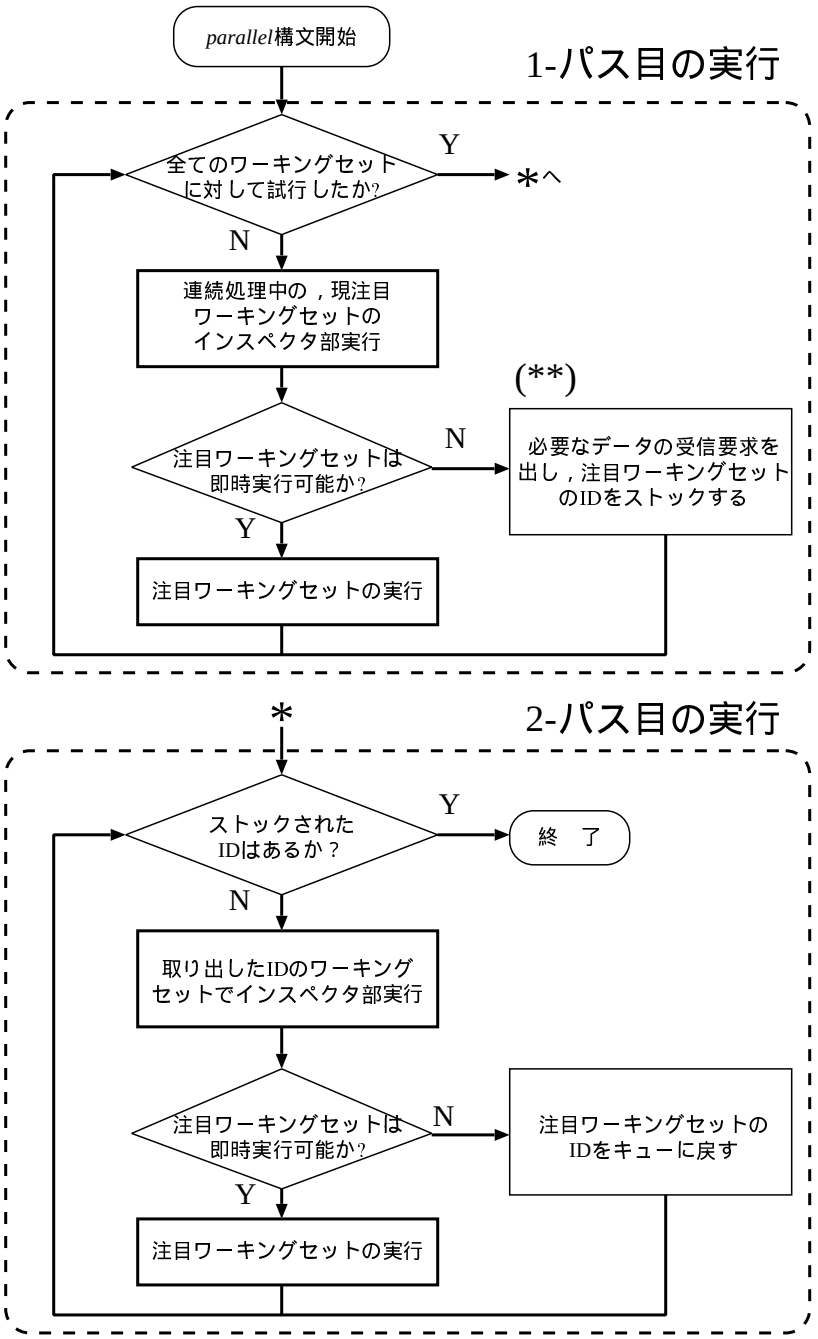


図 4.2: 提案手法の実行方式

step3： 全てのワーキングセットに対して一度目の検査，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ワーキングセットに対して，同様の処理を，それぞれ Inspector 部の最初から，キューが空になるまで繰り返す．

次に，本手法で用いる Inspector 部の具体的な生成ルールについて説明する．

4.2.2 Inspector 部生成法

生成方式

Inspector 部はワーキングセットの実行に際して，即時実行不可能とする要因の検出を目的として生成する．またこの際，Inspector 部実行による負荷の削減，および実行の意味の変更 (2 重実行など) を防ぐために，以下のルールに従う．なお，これらルールは全て，各ワーキングセットが共有する大域変数へのアクセスに対しての定義となっているが，各ワーキングセット間における通信処理に対しても同様のルールが定義される．

並列処理構文内の各文に対して，

1. 局所変数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を含む文は，その部分を削除した残りの部分を Inspector 部に加える．
3. 大域変数からの読み出しを伴う文は，
   (a) 自身が保有することが明らかな要素に対しては，局所変数と同様に扱う．
   (b) そうでないならば，Inspector 部の対応する部分に検査処理付きのデータアクセス関数を加える．
4. プログラマ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．

各ルールについて (1) は動的に決定するアクセス対象を特定するために必要な，最低限，かつ 2 重実行による意味変化の無い演算であり (2) はワーキングセットの実行意味の変化を避けるために必要なルールとなる．

(3) に関して，このルールはオーバーヘッド削減のために，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においては，各計算機は自身が管理するデータ領域を通常持つ．また VIOS における相対アクセス方式により，このような差別化は比較的容易に判定することができる．そこで，このような領域に対する検査を間引き，局所変数と同様の扱いとする．

(4) も同様にオーバーヘッド削減のためのルールである．このルールを利用する構文として，4.3.1 節に示す検査対象外指定構文が上げられる．また，現在は導入されていないが，I/E が必ず連続して実行されることを利用し，Executor 部では Inspector 部で実行した処理を行わない，といった制御を行う構文も導入する予定である．

生成例

本方式に基づいて実装を行った VPE-p により，実際に抽出された Inspector 部の例を幾つか示す．図 4.3 は，先ほどの図 3.4 の並列構文に対して抽出を行った例である．

記述された構文：

```
shadow[][] = (data[-1][] + data[][-1] + \
              data[+1][] + data[][+1]) / 4;
```

プリプロセッサによって変形された後：

```
data->vs_ck(x-1,y), data->vs_ck(x,y-1), \
data->vs_ck(x+1,y), data->vs_ck(x,y+1);
```

図 4.3: 生成例その 1 Jacobi 処理に対して

なお，VPE-p は，C++コードへのプリプロセッサとして機能し，VPE-p 中の VIOS 変数とその読み書きは，クラスとそのメンバ関数により実装される．図 4.3 の例では，大域変数である data へのアクセスは，全て検査のためのメンバ関数 “vs_ck” により置き換えられ，shadow への代入演算，および無効となる各演算は削除される．

次により複雑な処理に対する抽出例を図 4.4 に示す．この例では，大域変数は base, middle, ans の変数となってるが，VIOS アクセス方式 “[][]” により，保持していることが既知となるアクセスは，メンバ関数 “vs_rdc”, および “vs_wtc” に置き換えられ，局所変数と同様の扱いとなっている．また図中 (1) に示すように，大域変数アクセスを含む構文は，最低限度の検査構文へと修正されている．

記述された構文：

```
if(!base[][]) {
    for(int i=1; i < middle[][]; i++) {
        if(l2g_y(base,y)-i >= 0) {
            val = sqrt(pow(i,2) + pow(middle[][-i],2));
            if(val < ans[][]) {
                ans[][] = val;
            }
        }
    }
}
```

プリプロセッサによって変形された後：

```
if(!base->vs_rdc(x,y) ) {
    for(int i=1; i < middle->vs_rdc(x,y) ; i++) {
        if(l2g_y(base,y)-i >= 0) {
            middle->vs_ck (x,y-i);                ← ( 1 )
            if(val < ans->vs_rdc(x,y))
                 ans->vs_wtc(x,y) = val;
        }
    }
}
```

図 4.4: 生成例その 2 より複雑な演算に対して

4.2.3 特徴のまとめ

VIOS に本手法を用いた構文を導入することにより，以下のような効果が期待できる．

- 各ワーキングセットに依存関係が存在する処理に対しても，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く，メモリ領域の消費を最小限に抑えることが可能となる．

- 1 ワーキングセット内に存在する，複数の依存関係をまとめて解消することが可能となる．
- 先送りする処理が複数たまるまでデータ受信要求の発行を遅らせることにより，通信のまとめあげ効果，また要求中に他のワーキングセットの計算を並進することが可能であることより，計算による通信の隠蔽効果を期待することが出来る．(図 4.2 中 (**))

しかしまた，本手法を導入することにより，以下の問題を考慮する必要がある．

- 各ワーキングセット毎に 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 (他のワーキングセット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．

まず Inspector 部の実行負荷に関して，この負荷は，対象とする処理の性質によっては非常に大きなものになり得ると思われる．しかし，本手法では疑似スレッドを生成する場合における，無駄な再開と切り替えが無いこと，および 4.2.2 節で述べた負荷軽減手法，4.3.1 節で導入する検査対象外指定構文などにより対応可能であると考える．

また依存関係の解消問題は，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できる．

4.3 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．

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

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．

parallel_ie 構文は，本章で示した I/E 方式を用いてプログラムコードの展開を行うため，ワーキングセットが以下の様な特徴を持つ場合に利用効果をあげることが出来る．

- 各ワーキングセットの処理間に依存関係があり，またその依存方向がループ，再帰処理のような単純な生成順とならない．
- 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ため，静的なキャッシュ領域として定義することが難しい．

本構文を利用した例を図 4.5 に示す．この例は，5.1.2 節における評価の際に利用する処理の一部となる．詳細な説明は 5.1.2 節で行うが，大域変数 middle の参照範囲が動的に決定する処理となる．従って，従来手法による実装では，参照範囲する可能性のある範囲の取得，取得する値を持つ計算機の特定，および各計算機ことのデータ取得処理の記述が必要となるが，図 4.5 に示すように，プログラマは動的な情報を考慮する必要なく，処理を記述することが可能となっている．

4.3.1 検査対象外指定構文

本手法における Inspector 部負荷軽減のために，以下のようなオプション命令をさらに追加する．

no_inspect{ この部分に記述された処理は，Inspector 部に反映されない }

この命令は，プログラマが処理が実際に実行される環境を把握している場合，その情報を元に最適化を行うための命令となる．このことは，本手法の主旨である，実行

```
vs_module
euclidean_map(iImage base, iImage middle)
{
   // 初期化処理，および中間地図作成処理が入る
    parallel_ie(x,y) {
        if(!base[][]) {
            // この middle[][] の値により参照領域が変化する
            for(int i=1; i < middle[][]; i++) {
                if(l2g_y(base,y)-i >= 0) {
                 // middle[][-i] が計算機内か否かを考慮する必要無し
                    val = sqrt(pow(i,2) + pow(middle[][-i],2));
                    if(val < ans[][]) {
                        ans[][] = val;
                    }
                }
                if(l2g_y(base,y)+i < orig_heigh(base)) {
                 // middle[][+i] が計算機内か否かを考慮する必要無し
                    val = sqrt(pow(i,2) + pow(middle[][+i],2));
                    if(val < ans[][]) {
                        ans[][] = val;
                    }
                }
            }
        }
    }
    // 以下処理が続く
```

図 4.5: ユークリッド地図作成モジュールの一部

時環境などから独立した，抽象度の高いプログラムを可能とすることから厳密には反する．しかし，例えば並列処理部中の局所演算の内，大域変数へのアクセスに影響しない演算を把握することは，あくまで 1 ワーキングセット内の処理として考慮することが可能であり，処理全体のパフォーマンスも考慮した上，付加的な命令ととして追加したものである．

第5章

評 価

VIOS の有効性を確認するため，いくつかの処理の実装と評価を行った．実装に関して，言語処理部は 2 章で述べたように，VPE-p から C++プログラムへのトランスレータとして実装したが，その際，VPE-p における大域変数は，C++ 言語におけるテンプレート機能を利用し新たな型へと変換した．また，文献[18] により紹介されている，C++テンプレートの静的展開を利用した高速化を合わせて利用した．

5.1 拡張並列構文

まず拡張並列構文を利用する，VIOS におけるアルゴリズム記述能力に関する評価を行った．

5.1.1 アルゴリズム記述能力

図 5.1 に示す，簡単な依存関係を持つデータへの処理を元に，拡張並列構文に対する予備的な評価を行う．この例は，1 次元データに対する処理を表しており，処理は，実行時動的に決まる開始点（図中 start point）を中心とし，左右それぞれの要素が，矢印の方向に対し，順に処理可能となるものとする．このように，処理可能領域が “伝搬” する処理は粒子シミュレーション[9] やボロノイ図作成処理[11] など，データ並列処理によく見られるパターンの 1 つである．

この例をデータ並列処理する場合，一般に必要となる記述を図 5.2 に示す．

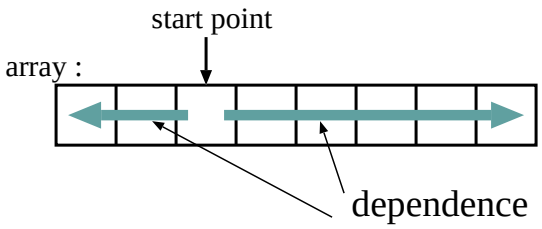


図 5.1: アルゴリズム記述能力の評価サンプル

図中の各関数について，exec_process() は各要素が行う処理を，isMyArea(id) ，および isUpperArea(id) は，各計算機において，自計算機が保有している配列部分要素は id を含むか否か，id より大きな値を持つか，を返す関数であり，MyMinRow，MyMaxRow は，自計算機が保有する部分配列の最大，最小値をそれぞれ示す．

図中の (*) に示す様に，通常の記述では，開始点を含む部分データが，どの計算機に存在するか，により場合分けをする制御構造をプログラマが記述する必要がある．そしてこの，実行時のデータ分割方式，計算機台数などのパラメータに依存する煩雑な記述は，扱うデータの次元数や処理の複雑さが増すに従い，より複雑で冗長なものとなる．またこのような問題は，一般的な形式として定義することが難しく，ライブラリやテンプレートを用いて対応することも困難となる．

それに対し，この例を VIOS の拡張並列構文を用いて記述したものを図 5.3 に示す．

図中の関数 comm(id) は，処理中のワーキングセットが，ワーキングセット id よりデータ受信を行うブロック関数を表す．このように，拡張並列構文を用いたコードでは，プログラマに要求するものは，あくまでワーキングセットレベルでの依存関係となり，これらはアルゴリズムを設計した段階で自明な情報のみである．このように，VIOS では，実行時のデータ分散や，デッドロックを回避するための処理方向，など実行時のパラメータを意識する必要のない，非常に簡潔な記述が可能となる．

以上より，拡張並列処理構文を用いることにより，実行時に決まる各パラメータから独立した，より抽象度の高いプログラミングが可能であることが示された．

```
// ここまでの処理により，start_point が決定する

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 5.2: 通常手法で記述したコード

5.1.2 実処理に対する記述能力

次に，実際のデータ並列処理において，基本的なアルゴリズムとして利用される，3つの異なる特性を持つ処理に対して，VPE-p による記述と，その評価を行った．なお今回対象とした処理は，いずれも全体の挙動を視野に入れることにより，比較的効率的な実装が行える処理を対象とした．これは，上記のような代替え法が存在しない処

```
// ここまでの処理により，start_point が決定する

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

図 5.3: 拡張並列構文を用いて記述したコード

理に対しては，全てのワーキングセットに関して，詳細な実行を制御するか，全てのワーキングセットを真にスレッドとして生成する，またはI/E 法のように何か投機的なスケジューリング法を導入するしかなく，いずれの場合もコーディング負荷，実行効率ともに比較できるものにならないと判断したことによる．

評価アルゴリズム

以下に，評価に利用するアルゴリズムを示す．

処理 **1** 2 次元空間に対するJacobi 処理．本処理は，空間中の各粒子が，周囲の粒子より現在の温度を取得し，その値を用いて次の自温度を決定する処理である（図 3.4 参照）．そのため，各粒子は毎時間ステップにおいて，特定の周辺粒子と温度情報の交換を行う．本処理を空間分割により並列に処理する場合，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ため，分割された空間の界面において毎ステップ通信が発生する．すなわち，本処理は各粒子の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．また 1 セル当りの毎ステップ計算量は比較的小さい．

処理 **2** 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[7]．本処理は処理 1 と同様に，各粒子が周辺粒子と状態を交換し合い，次の自状態を決定する処理であるが，図 5.4 に示すように，情報を交換する周辺の定義が現状態により変化する．そのため，並列処理の反復のたびに必要となるキャッシュ領域変化する．よって静的に最適な通信を決定することは困難となる．しかし，近傍の数は少ないため，使用する可能性のある情報を全て通信することにより対応することもできる．また処理 1 と比較して，1 セル当りの計算量はより大きなものとなる．

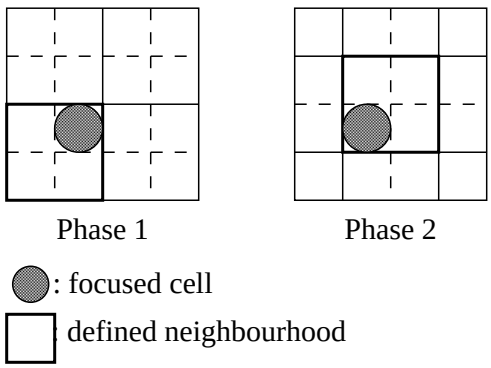


図 5.4: ガス拡散シミュレーションにおける Margolus 近傍定義

処理 **3** 3 つ目の処理は，Hirata と Kato のアルゴリズム[8] を使用した，2 次元ユークリッド地図作成処理．本処理は，空間中に複数の物体が存在する環境下において，各々の地点から最も近い物体までのユークリッド距離地図を作成するものである．Hirata と Kato のアルゴリズムは，データ並列処理において，オーダーが最も小さくなるアルゴリズムとなる．各セルはまず，自身の左右を探索し最も近い物体までの距離を中間地図に記録する．次に，上下に位置するセルに対応する中間地図から値を参照し，その値をもとに，三角測量を用いて自身に最も近い物体までの距離を求める (図 5.5 参照)．本処理では，動的に生成される中間地図の値に依存して，上下に参照する範囲が変化する，すなわちワーキングセットが利用するキャッシュ領域が動的に変化する．そのため，通信が必要な周辺を，“近傍” として静的に決定することが難しい．従って，より最適な並列実行を行う

ためには，アルゴリズムそのものを変更する必要がある[8]．また 1 セル当りの計算量も三角測量などを行うため比較的大きい．

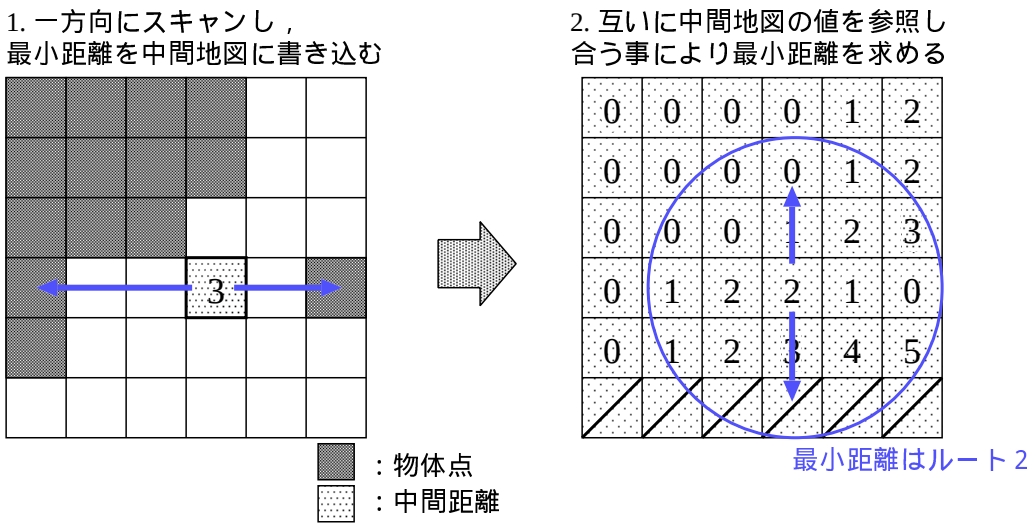


図 5.5: ユークリッド地図作成処理

これらの処理を，VPE-p により記述し，VIOS 上に実装した．また比較実装は，MPI[1] などの一般的な通信ライブラリを用いることを想定した．そのため，処理 1，2 は，データ分割境界面に対する集合通信部を，処理 3 に対しては，必要となる可能性のある計算機外データの範囲取得，および集合通信部を追加し，それぞれ分割数に合わせ最適化をしたものを，および逐次処理用として別に最適実装したものを，それぞれ作成した．

アルゴリズム記述能力

いずれの処理に対しても，本構文を用いて記述を行った場合，キャッシュ領域の更新処理などの記述は必要なく（parallel_ie 構文により，外部データの参照は自動的に要求され，要求したワーキングセットは，自動的に先送りされる），また計算機台数などの実行時パラメータに関する情報も与える必要なく記述することができた．そのため，記述量は逐次実行に対するものと同等となった（ただし，メインフロー文における並列処理指示文は必要となる）．なお，処理 3 に対する記述の一部は，図 4.5 においてすでに示したようになる．

これに対し，比較用に記述した通常実装では，いずれも全体の実行を視野に入れ，分割方式や分割部等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，考慮する必要があった．結果，モジュールプログラムのソースサイズは，逐次処理に対する記述に比べ，数 10%から，通信制御コードを含めるならば数倍まで増加した．

また特に，処理 2,3 に対して，分散メモリ環境下においては，処理 2,3 のように，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，分割境界面のデータを処理するワーキングセットを別領域として定義し，処理を分ける必要がある．さらに処理 3 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．

このような処理に対しても，拡張並列構文を利用したプログラムでは，$\mathrm{parallel_ie}$ 構文を用いた以外は変更の必要なく，逐次処理と同様の記述のみにより，集合通信，計算による通信の隠蔽効果を期待することが可能であった．なお，通信と計算のオーバーラップ，ならびに通信の纏め上げ効果の詳細は，次節により評価する．

以上より，実処理の基本となるアルゴリズムに対しても，拡張並列構文には高い記述能力があることが確認出来た．

5.1.3 計算による通信の隠蔽効果

4.2.3 節により示した，拡張並列処理構文における，計算による通信の隠蔽効果の確認とその評価を行った．なお評価には，5.1.2 節により定義した処理 1 から 3，比較実装 1 から 3，逐次処理用実装 1 から 3 を利用した．

評価環境

評価は，CPU Athron(1.2GHz)，Memory 512 Mbyte，OS Solaris8 の計算機を，100 BaseTX 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．また処理対象データは，処理 1,2,3 ともに，$1024\times1024, 2048\times2048, 4096\times4096$ の 3 つの 2 次元空間を用いた．なお，処理 3 における物体点の配置は，今回の評価では負荷分散は考慮しないため，1/20 の確率でランダムに均等配置したデータを利用した．

評価結果

まず拡張並列構文における，Inspector 部導入のために速度低下について評価を行う．VIOS により実装したものを 1 台の計算機により実行し，逐次処理用に実装したものと比較した速度低下率を表 5.1 に示す．なお，この表は，2048×2048 のデータ空間に対する実行結果を元に作成したものであり，速度低下の要因としては，Inspector 部の実行負荷，および並列処理のために必要なデータ分割処理，同期処理，通信処理すべてを含んでいる．

表 5.1: 拡張並列構文おける逐次処理からの速度低下率

	処理 1	処理 2	処理 3
速度低下率 (%)	88	41	5

結果，速度低下率は処理に依存し，88%から 5%までの広い値を取ることになった．これは，処理 1 では，処理そのものの計算量が小さいく，プログラムに追加される Inspector 部，すなわち Executor 部から引き継が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，処理 2,3 で

は，Executor 部に存在する浮動小数点演算の多くが Inspector 部には存在しないため，全実行に対する Inspector 部の実行コストが小さくなることによるものである．

以上より本方式は，適用するワーキングセットが，外部大域変数アクセスを多用した，複雑な演算を主として構成されている場合に，より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評価を行う．図 5.6 から図 5.8 に実行結果を示す．それぞれ左図が比較用実装の評価結果であり，右図が拡張並列構文を元に実装を行った結果である．また横軸は計算機台数，縦軸は逐次用実装を元とした，台数効果を示す．

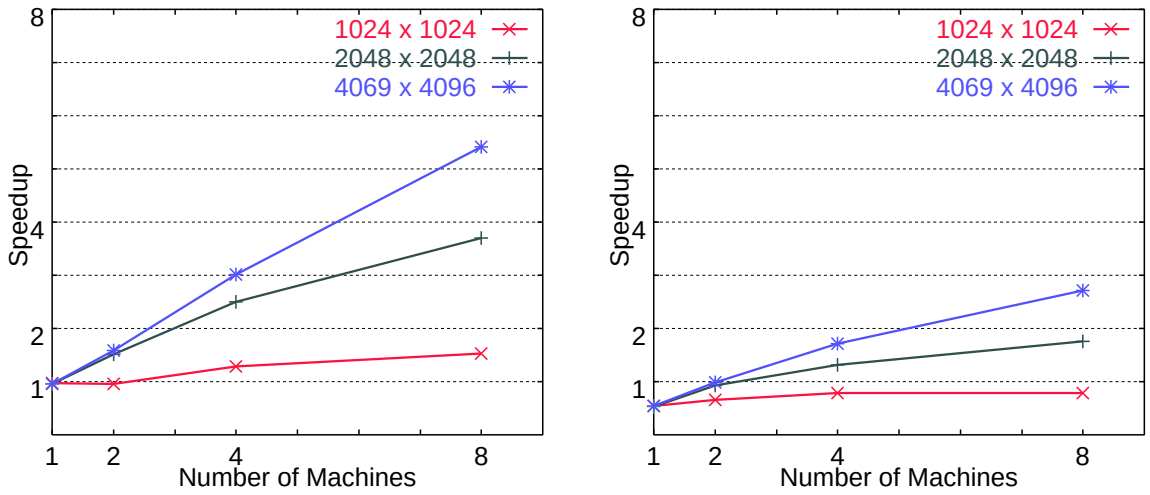


図 5.6: 処理 1 における速度向上率

まず処理 1 に対して，本方式は，比較プログラムと通信量については同等であるが，通信処理に前後した同期処理が必要でない，という利点が存在する．しかし評価結果は，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を示すものとなった．これは，上記のように，1 ワーキングセットの処理そのものが単純であるため，Inspector 部による演算負荷が総実行時間に影響を及ぼしたものである．

次に処理 2 では，上記の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．またさらに，Inspector 部を実行することによる演算負荷も処理 1 に比べ相対的に低い．結果，比較実装に比べ，1024×1024 の空間では 1.4 倍程度の性能向上が，2048×2048 の空間でもほぼ同等の性能となった．しかしながら 4096×4096 の空間で

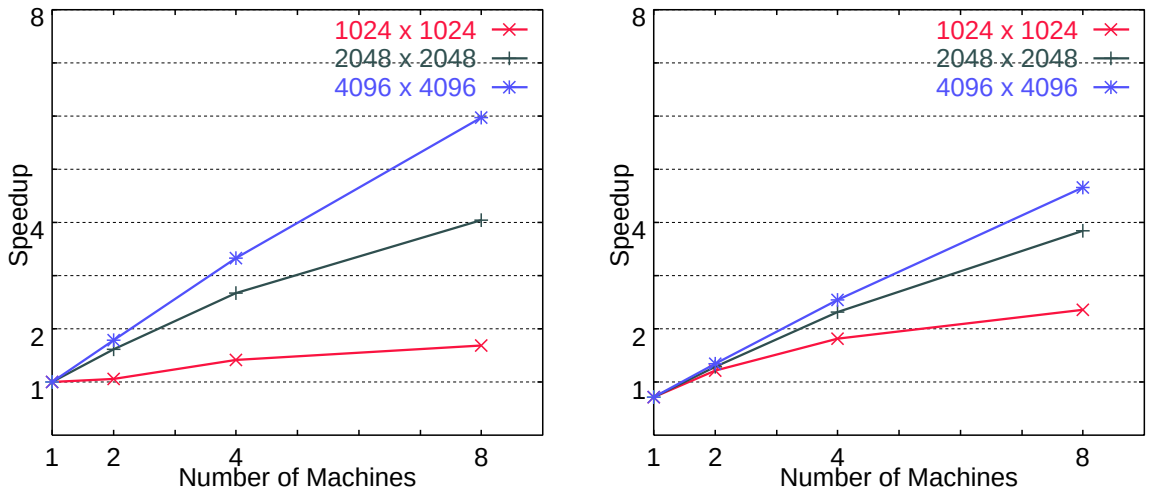


図 5.7: 処理 2 における速度向上率

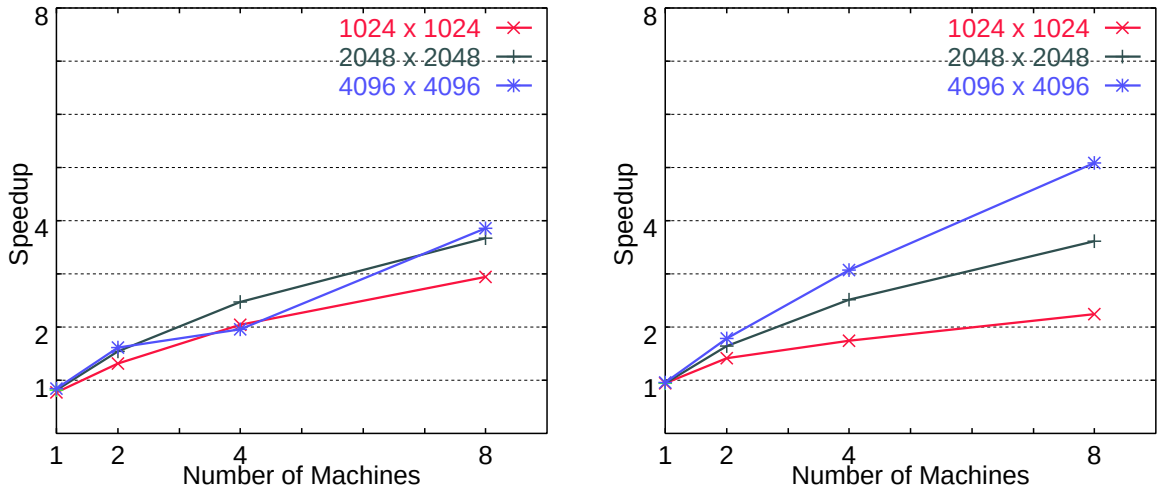


図 5.8: 処理 3 における速度向上率

は，比較実装の方が良い結果となった．このことは，本処理が，データサイズを n^2 とした場合，演算量は n^2 に，通信量は n に比例する処理となるため，空間サイズの増加に伴い通信処理時間の影響が下がり，演算量に比例する Inspector 部の追加負荷が無視できなくなったためである．

そして処理 3 では，本方式が順調な速度向上を見せることに対し，比較実装では空間サイズの増加に伴い速度向上率が低下する．これは，本処理が通信量の静的な決定が困難なことに起因し，処理 2 とは逆に，比較実装における外部参照領域の計測部分

の処理が無視できない計算量になるためのものである．

以上の結果より，本手法における，計算による通信の隠蔽効果は有効に機能することが示された．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ことも合わせて示された．

5.2 分散処理環境VIOS

VIOSの全ての機能を利用した場合のアルゴリズム記述能力と，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5つの処理について実装を行い，この評価を行った．

なお，処理5は前節における処理3と同様のものであるが，5.2.1節における評価に用いるため，もう一度扱う．

処理**1** 2次元画像に対するテンプレートマッチング処理．探索画像は分割，テンプレート画像は全ての実行部が重複して，それぞれ保有することにより，ほぼ完全に並列処理可能であるが，3.2.4節で述べたように処理後の結果統合が問題となる．そこで，依存型リダクションを利用してこの記述を行った．

処理**2** 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．5.1.3節の結果に従い，3.2.3節で述べた，キャッシュ一括更新命令を利用して記述を行った．

処理**3** 3次元空間に対するJacibi処理，処理2の3次元空間に対するものであり，処理2と同様に，キャッシュ一括更新命令を利用して記述を行った．

処理**4** 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[7]．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．5.1.3節の結果に従い，3.2.3節で述べた，キャッシュ一括更新命令を利用して記述を行った．

処理**5** ユークリッド地図作成処理[8]．各ワーキングセットのキャッシュ半径が動的に決定する処理．拡張並列構文を利用して記述を行った．

5.2.1 アルゴリズム記述能力

まず処理1，および2について．図3.4，3.6に示すように，簡潔な記述により処理を行うことが可能となった．また，逐次処理用プログラムと比較しても，変更する必要

のある個所は原則的に図 3.4 (3)，図 3.6 (4) に示す 1 命令のみとなり，プログラマは並列処理のアルゴリズムのみの考慮で，通信部を意識すること無く，簡単な記述を行うことが出来た．なお，図 3.4 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も同様に，リダクション命令を利用して記述することが可能であり，その他は逐次処理と同様の記述となった．

次に処理 3 について．処理 3 は処理 2 に対し，扱うデータが 2 次元から 3 次元に変わったものであり，アルゴリズム的にはほとんど変更はない．従って VIOS での実装では，処理対象データタイプ，および内部アルゴリズムの部分の変更のみで対応することが可能であった．このことは，通常 3 次元データに対する並列処理を記述するために必要な負荷 (3.2.3 節参照) を考慮すると，VIOS では非常に低負荷に記述することが可能であったと言える．

最後に処理 4,5 について．処理 4 は，動的に変化するキャッシュ領域のサイズが，ある程度の枠内に収まる処理であるため，利用する可能性のある領域を全て一括更新すべきか，または拡張並列構文を用いて動的な対応を行うべきか，は入力データ，利用計算機数などに依存する問題となる．同様に処理 5 についても，入力データ次第では外部利用領域をキャッシュ領域として定義する手法の方が有効となる．このように，実際に幾つかの実装法に対して試行錯誤を行う必要がある問題は多い．このような問題に対しても，VIOS におけるキャッシュ一括更新命令や拡張並列構文を利用することにより，簡単に試行を行うことができる．実際，処理 4,5 に対して，これら 2 つの実装を切り替えるために必要な記述は，parallel，および parallel_ie 構文の切り替え，およびキャッシュ一括更新命令の有無のみで可能であった．

5.2.2　クラスタ環境下における速度評価

処理 1 から 5 に対し，クラスタ環境下における速度向上率を評価した．なお，評価環境，および利用データは 5.1.3 節と同様である．また，処理 3 の 3 次元データに対しては，一辺 128，256，384 の，それぞれ 3 段階のデータを利用した．

評価結果を図 5.9 に示す．図中，横軸は使用計算機の台数，縦軸は，逐次処理用に別途最適実装したものに対する速度向上率となっている．結果，各処理において，8 台の

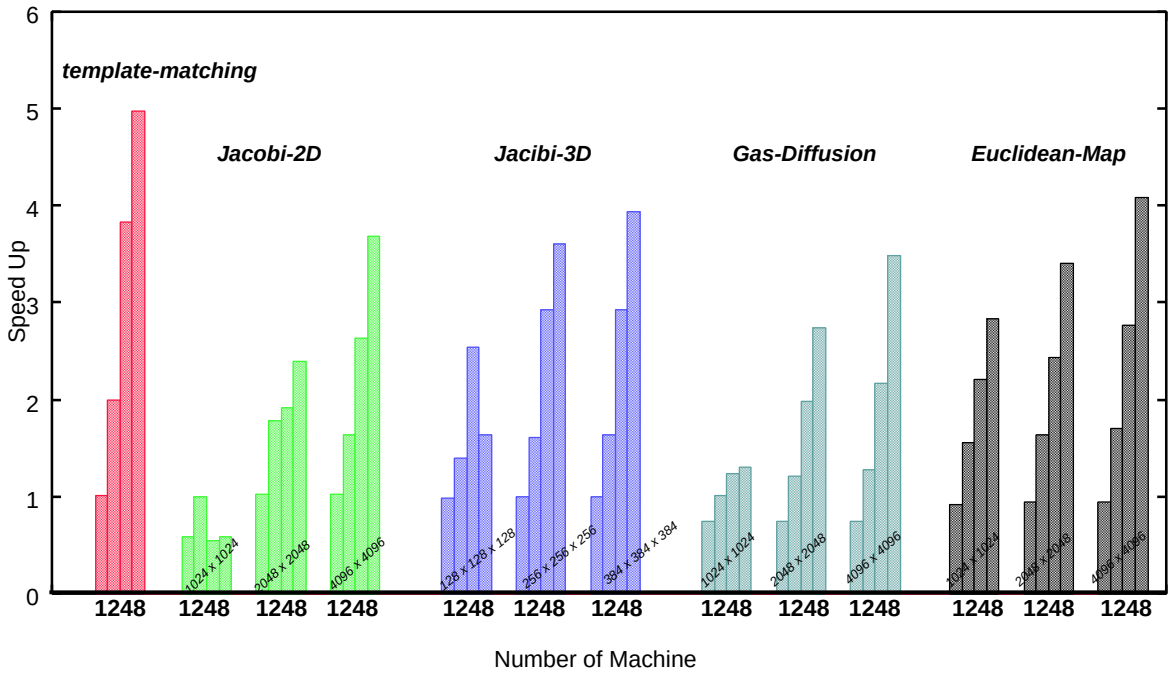


図 5.9: VIOS による実行速度向上率

計算機を使用時では，処理 1 については 5 倍程度，処理 2,3,5 については，4 倍強，処理 4 についても 3.5 倍ほどの速度向上が見られる結果となった．

処理 1 について，この処理は並列性が高いにも関わらず，8 台での並列処理時において速度向上率が低下する結果となった．このことについて調べた結果，図 5.10 に示されるように，今回利用したデータは，テンプレートサイズに対し探索画像のサイズが小さいため，実際には処理を行わない領域のサイズが大きく，各計算機での負荷バランスが不均等になったためであった．またこのことは処理 5 における，物体点のばら

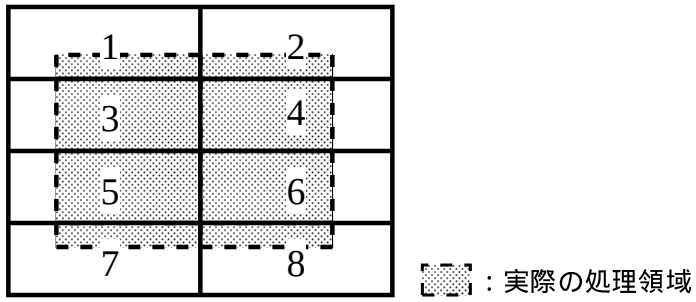


図 5.10: テンプレートマッチングでの有効処理領域

つきに対しても同様であり，処理 5 の速度向上率低下の大きな要因となった．

負荷分散への対応は，データ分散処理における大きな問題であり，VIOS における今後の課題であると考える．

また，処理 2 から 4 における速度向上率の伸び悩みは，5.1.3 節においての評価と同様に，本手法に用いた Jacobi 法が，非常に単純な演算のみにより構成される（各ワーキングセットあたり数回の加算と一回の除算）ためである．よって，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[9] など）ではより高い速度向上が見込まれる．

以上の点を考慮しても，逐次処理用に記述されたプログラムと比較し，VIOS による実行は有効な速度向上が得られる結果となった．

第6章

関連研究

Inspector/Executor 方式を利用し，通信処理を最適に行うための試みとしては，様々な手法が提案されている[15)16)17)] が，通信の “最適化” を目的として導入しているものとして，横田らは[19)] の研究がある．横田らは，ハードウェア支援機能を有効に利用するために，Inspector/Executor 方式を用い，Inspector 部において通信パターンの解析，最適な通信となる組み合わせの探索を行い，場合によっては結果を用いた再コンパイルを行うことにより実行の最適化を行う．このように，解析結果によって実行部の変更を行う手法は，本研究で提案した手法においても，Executor 部の他計算機への移動，などにより導入可能な方式であり，補完的な意味合いを持つ手法であると考える．

また，細粒度レベルでの検査部を用意し，通信の隠蔽効果に用いる研究には Shasta[20)] システムがある．Shasta は，共有メモリ環境用実行コードをクラスタ環境下で実行するために，実行コードを再コンパイルするが，その際各リード / ライトアクセスの前後に，外部アクセス検知用コードを挿入する．また提案手法同様，大域 / 局所変数の差別化による検知コード挿入の削減，複数通信の纏め上げ処理等を導入し，負荷軽減を図っている．しかし，入力として，並列環境用の実行コードがすでに存在することを前提とするため，デッドロック問題は考慮されていない．またセルの実行順序に関する処理は存在しないため，事前処理効果も持たない．

多数の細粒度並列処理部を効率よく実行するための手法は，並列関数型言語や並列論理型言語の分野において従来より提案されている[21)22)23)]．一例として，Threaded Abstract Machine[21)] では，スレッドを保存するコンテクストを，連続的に実行される

と予測されるスレッドグループに対して共通のフレームを割り当てる．これにより，スレッド切り替え時に発生する負荷軽減を図る．しかし，これら分野における研究においては，演算，関数呼び出しレベルでの並列処理に焦点を当てるため，次に処理可能となるスレッドの予測が容易に立つこと，および再帰的な処理を扱うことが多く，LIFO など経験的なスケジューリング方式が適合するため，コンテクスト領域の肥大化に関してはあまり言及されていない．また文献[21] で定義されている通り，スレッドは中断するものとして定義されておらず，これらは別の枠組みであると考えられる．

多数の並列処理実行単位を扱う研究には他に，負荷分散の 1 つの手法である Lazy Task Creation(LTC)[23] がある．LTC では，負荷の静的推定が困難な問題に対する動的負荷分散のために，タスクキューを生成する．そして各々の CPU が，担当するタスクを交換することにより負荷分散を達成する．タスクそのものの移動を扱うため，LTC では通信，データの局所性を考慮する必要のあるデータ並列処理を扱うことは難しいが，本手法で導入することが難しい負荷分散を可能としている．

第7章

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，parallel_ie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．

謝辞

本研究を行うにあたって，多大な御尽力を頂いた岩田彰教授，松尾啓志助教授ならびに高橋直久教授に心から厚く御礼申し上げます．

また，本研究の検討，協力をして頂いた岩田研究室の皆様に感謝致します．

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] 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).

[3] 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).

[4] 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).

[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] 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).

[7] P. Thomas: *Hardware Compilation of Cellular Automata Algorithms*, Oxford University, School of Engineering and Computer Science, Undergrad. project report, (1999).

[8] A. Fujiwara, T. Masuzawa, and H. Fujiwara: *A Parallel Algorithm for the Euclidean Distance Maps*, Technical Report of IPSJ, Vol. 95, No. 10, 1995.

[9] 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).

[10] C.H. Hsu, S.W. Bai, Y.C. Chung and C.S. Yang: *A Generalized Basic-Cycle Calculation Method for Efficient Array Redistribution*, IEEE Trans. Parallel and Distributed Systems, vol.11, no.12,pp. 1201–1216, Dec, 2000.

[11] 渕田孝康, 上原修平, 森邦彦, 村島定行: 離散ボロノイ図における母点の追加と削除のアルゴリズム, 電子情報通信学会論文誌 A, Vol. J84-A, No. 8, pp. 1027–1036 (2001).

[12] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel: *The High Performance Fortran Handbook*, MIT Press, 1993.

[13] 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).

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall: *The Implementation of the Cilk-5 Multithreaded Language*, ACM SIGPLAN Conference on Programming Language (1998).

[15] 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).

[16] 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).

[17] J. Saltz, R. Ponnusamy, S. Sharma: *A Manual for the CHAOS Runtime Library*, University of Maryland, Department of Computer Science and UMIACS Technical Reports CS-TR-3437 and UMIACS-TR-95-34 (1995).

[18] 松田元彦, 石川裕, 佐藤三久: C ++テンプレートを使ったデータ並列ライブラリの効率化手法, 情報処理学会論文誌, Vol. 38, No. 09, pp. 1781–1789 (1997).

[19] 横田大輔, 千葉滋, 板野肯三: 通信機構に合わせた最適化を行う並列化コンパイラ, 情報処理学会論文誌, Vol. 42, No. 4, pp. 860–867 (2001).

[20] D. J. Scales, K. Gharachorloo, and C. A. Thekkath: *Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory*, In Proceedings of Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 174–185 (1996).

[21] D.E. Culler, A.Sah, K.E.Schauser, T. Eicken, and J. Wawrzynek: *Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine*, In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 166–175 (1991).

[22] 荒木 拓也, 田中 英彦: Committed-Choice 型言語 Fleng における静的粒度最適化, 情報処理学会論文誌, Vol. 38, No. 9, pp. 1771–1780 (1997).

[23] E. Mohr, D. Kranz, and R. Halstead: *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*, In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, pp. 185–197 (1990).

Appendix A

VIOS実行フロー記述言語 VPE-i仕様

平成 14 年 2 月 7 日現在の VPE-p の仕様を示す．

A.1 設計方針

VPE-i は，メインフロー文において，並列処理に伴う初期化や結果の集計，表示処理を行うことを想定しており，あまり複雑な処理をこなすことを目的としていない．また，インタープリタにおいてはその動作速度が非常に問題となる．そのため，基本構成として大幅に機能削減をした C 言語に，組み込み型の手続きを追加するという形を取る．実際の実装に際しては，

- 基本文法，および各演算子は C 言語に従う．
- 単項演算子，および三項演算子，ポインタは導入しない．
- 戻り値を返す “関数” は導入せず，“手続き” ベースに処理を行う．
- 扱えるデータ型は整数，実数，文字列の 3 種であり，構造体は扱わない．
- モジュール呼び出し，分割方式の指定，などは所定手続きを呼び出すことにより行う．

- 新しい手続きを定義する (A.5.2 参照) 場合は vpe-i を再コンパイルして VIOS の中に “組み込む” .

A.2 予約語

以下に挙げる単語は VPE-i の予約語であり , 変数名などに利用することは制約される .

all, cache, column, depth, division, else, fCube, fImage, float, for, gather, heigh, iCube, if, iImage, imgLoad, imgSave, int, module, pixel, put, RAND_MAX, random, remake, row, set, width, while

A.3 利用可能な型

- iImage 型 : 整数型の 2 次元情報を扱う型であり , 以下の 3 つの方式何れかにより宣言可能 .

  1. iImage 変数名 (*width*, *height*);
  2. imgLoad(“PGM ファイル名”, 変数名);
  3. iImage* 変数名 (*width*, *height*);

  なお , 3 番目の宣言はポインタ変数としての宣言ではなく , 型の後に “ * ” を付けた宣言は , 初期化の時点で領域を確保しない宣言を表す . つまり , 並列モジュールの実行時に , 部分領域の大きさに関する情報のみを各実行計算機にし , 並列モジュール内においてデータを生成することが可能となる .

  また , 変数へのアクセス方式は , data(3,4) のような形で行う .

- fImage 型 : 実数型の 2 次元情報を扱う型であり , 宣言方式は iImage 型と同様となる (ただし , imgLoad による宣言は行うことが出来ない) .

- int 型 : 整数型の 1 次元情報を扱う型 . 通常の C 言語と同様に , 単独の変数 , または [ ] を利用した配列として宣言を行う (ただし 1 次元変数のみ) .

- float 型：実数型の 1 次元情報を扱う型．宣言方式は int と同様となる．
- iCube 型：整数型の 3 次元情報を扱う型であり，以下の 2 つの方式何れかにより宣言可能．
  1. iCube 変数名 (*width*, *height*, *depth*);
  2. iCube* 変数名 (*width*, *height*, *depth*);

  変数へのアクセス方式は，data(3,4, 6) のような形で行う．
- fCube 型：実数型の 3 次元情報を扱う型であり，宣言方式は iCube 型と同様となる．

A.4 通信環境設定用構文

A.4.1 プラグマによる指定

リモートホストの指定や，通信形態の指定は，以下に示す#pragma 構文を利用する．

#pragma host_name *host name*

並列モジュールの実行を行うリモートホストを登録する．*host name* は，ローカル計算機から一意に特定できる名前なら IP でも名前でも利用可能．

#pragma div_num *number*

データ細分割実行 (3.1.1 章参照) を行なう場合，1 以上の数値にセットする．デフォルトの値は 1．

#pragma use_intercomm

リモートホストが相互に通信を行う処理の場合にセットする．通常想定される，キャッシュ領域の参照のみで処理を行うような，完全並列処理ならば設定する必要は無い．

#pragma connect_start

上記指定により設定された構成を基に，ローカルホストとの接続の確立を開始する．接続の確立は，並列モジュールの実行を開始する前に行う必要がある．

A.4.2 Set 手続きによる指定

並列処理に利用する各データの分割方式，キャッシュ領域の指定など，データに対する設定は，以下に示す set 構文を利用する．

set (division, *value_name*, *div_type*);

2,3 次元変数 *value_name* を並列処理する際の，ワーキングセットのタイプを指定する．*div_type* には次のタイプの中から指定する：

{“all”, “pixel”, “column”, “row”, “long,”, “short”}

なお，“long”, “short” はそれぞれ width, heigh のうち，{ 長い | 短い } 一列を 1 ワーキングセットとして定義する．また，デフォルトの指定は “pixel” となる．

set (division, *value_name*, *div_type*);

1 次元配列変数数 *value_name* を並列処理する際の，ワーキングセットのタイプを指定する．*div_type* には次のタイプの中から指定する：

{ “all”, “row”, “pixel”}

また，デフォルトの指定は “pixel” となる．

set (cache, *value_name*, *cache_size*);

変数 *value_name* を並列処理する際，各ワーキングセットが参照するキャッシュ領域の半径を指定する．

A.5 その他

A.5.1 並列モジュール実行後の変数の扱い

1 実行フロープログラムを処理する間，通常複数の並列モジュールを実行することが想定され，また連続的に実行される並列モジュールは同じタイプのワーキングセットに対する処理が行われることが多い．

そこで VPE-p では，実行速度を考慮し，各並列モジュールの実行終了後，メインプロセス (vios_run) に変数の値を暗黙的には戻さず，更新された値は各リモートホストが保持し続ける方針を取る．

そのため，全並列モジュールの実行後，値を統合して保存する場合や，先ほどまでの並列モジュールとは異なるワーキングセットに対して並列処理を行う必要がある場合，以下の命令を使用し，明示的に指定する必要がある．

gather (*value_name*);

並列モジュールの実行により更新された変数 *value_name* を，各リモートホストから回収し，実行フロープログラム中において最新の値にする．

remark (*value_name*);

gather 関数により回収した変数 *value_name* を，以降の並列モジュールにおいて異なるタイプのワーキングセットを用いて並列処理するために，再分割・転送を行う．

A.5.2 VPE-p への手続きの追加方式

VPE-p に新しい手続きを追加するための手順を，参考として以下に示す．

1. vpe-i.l，および lang.h に VPE-p から呼び出す手続き名を登録する．
2. interp.h に実際の処理を行う内部関数名を宣言する．
3. inner_code.cc 内にある vpe-i のコンストラクタへ内部関数を登録する．
4. inner_func.cc に実体（interp.h に宣言した関数）を作成する．
5. vios_run を再構築する．

Appendix B

並列処理モジュール記述言語 VPE-p仕様

平成 14 年 2 月 7 日現在の VPE-p の仕様を示す．

B.1 はじめに

この文章では，C++用ソースコードを出力するプリプロセッサである VPE-p により，制限される事項，汎用性の高いプログラミング環境を提供するための外部プロセスコール，予約語，などを示す．

B.2 モジュールプログラムファイル構成

VIOS では，複数のユーザにより利用されることを想定しているため，各モジュールファイルの統合は行わず，個別に識別するスタイルを取っている．そのため，1VPE-p プログラムファイル辺り，1 つの *module* が記述されていることを想定している．そのため，以下のような制限が存在する．

- コンパイル時指定するファイルに，指定する形式で *vs_module* 構文が 1 つ宣言されていること．

あくまでコンパイル時に指定するファイルに対する制限であり，このメインファイルから複数のファイルをインクルードすることに制限はないが，VPE-p は，インクルー

ドファイルを辿り解析することはしないため，指定するファイル以外（#include などで引用するファイル）では VPE-p の拡張表現，変数，などは利用できない．

B.3 モジュールプログラミング言語 VPE-p

VPE-p は，C 言語をベースとし，これに並列処理構文，部分大域変数およびそのアクセス方式，特殊関数を加えた言語として構成される．そのため，拡張表現を伴う構文（e.x. *parallel* 構文内，iImage 変数からの読み出しなど）以外では，通常 C 言語への制限はない．また，C++言語における，変数の途中宣言，および//によるコメントも使用することができる．

以下に，これら VPE-p における拡張表現ととその制限について示す．

B.3.1 拡張構文

1. ***module*** 構文：

   vios_child により，実行時呼び出される構文であり，並列モジュールにおけるメイン関数となる．

   vs_module 宣言

   ```
   vs_module モジュール名( VIOS型 変数名, ...) {
           /*** 本文 ***/
   }
   ```

   - **vs_module** は typedef された予約型である．
   - 引数は，メインフローから自動分割され渡されるため，{ivalue, fvalue, iImage, fImage, iCube, fCube} のいずれかをとる必要がある．
   - 引数 0 でも問題ないが，その場合 *parallel* 構文は利用できない（vios_child を実行するホスト数での並列処理になる）．

2. 並列処理構文：

   module 構文内では，以下の構文内において，3.2.2 節で定義したベース変数の情

報を元とする，各ワーキングセットの並列処理を行う (詳しくは 4 章を参照) .

(a) 各ワーキングセット間に順序関係の無い場合：

parallel 構文

parallel< *WS* サイズ, ...> (*index* 変数, ...) {
/*** 本文 ***/
}

- 各次元に対応した *parallel* 文を利用するためには，その次元の変数が *module* 構文の引数に有ることが前提となる .
- *module* 引数に ，同次元の変数が複数ある場合は，引数順に最初の変数がベース変数となる .

example:

```
vs_module sample(iImage data1, iImage data2) {
    parallel(lx, ly) {
        if((lx == 0) && (ly == 0)) ....

     } ....
```

この例では，2 次元変数に対する *parallel* 構文を利用している . また，*parallel* 構文内では，*module* sample() の最初の引数である data1 をベース変数とし，各次元の識別変数には lx, ly という名前を利用している .

(b) 各ワーキングセット間に順序関係がある場合
(または通信により各処理単位の処理時間に大きな差が存在する場合):

parallel_ie 構文

parallel_ie< *WS* サイズ, ...>(*index* 変数, ...) {
/*** 本文 ***/
}

- *parallel_ie* 構文内では，変数の宣言を行うことは出来ない .

B.3.2 VIOS 変数

1. モジュール引数として実行フローより引き継がれる変数は，ivalue, fvalue, iImage, fImage, iCube, fCube の何れかの型をとる．なお，各型の扱いは実行モジュールの場合と同様である．
2. 各 VIOS 変数へのアクセス方式は，従来の VIOS[6] と同様に，[ ] 演算子を利用することで，自ワーキングセットからの相対位置で，[[ ]] 演算子を利用することで，全体での絶対位置によるアクセスを行う．
   例えば，data[3][] との指定は，自ワーキングセットから x 軸方向に 3，y 軸方向に 0 移動したポイントの値を指すことになり，data[[3]][[0]] は，変数 data における (3,0) の要素を指すことになる．

- 相対アクセス方式は，その性質上，並列処理構文中のみでの指定となる．
- 全ての型の実体はポインタ形式となっているため，参照渡しのみ可能となる．そのため，実体の複製が必要な場合は，コピー宣言を利用した後，ImgCopy 関数などを利用する必要がある．
- VIOS 変数のアドレス指定の中に，VIOS 変数を利用することは，相対アクセスにおいて 1 段まで，絶対アクセスにおいては使用出来ない．

B.3.3 特殊関数

各ワーキングセットにおいて，境界条件などの例外処理，およびインデックス情報などを得るために使用する関数群となる．

1. アドレス変換関数：
   各計算機が持っている分割データ中での *index* と，統合された全データにおける *index* の変換を行うための関数群となる．アドレスの絶対指定などの場合に利用する．

(a) int l2g_x(*VIOS* 変数, *num*)

VIOS 変数x 軸における，ローカル *index num* を，全体データでの *index* に変換する（local to global）．

(b) int l2g_y(*VIOS* 変数, *num*)

上記関数の y 軸版．vsImage, vsCube 変数のみ利用可能．

(c) int l2g_z(*VIOS* 変数, *num*)

上記関数の z 軸版．vsCube 変数のみ利用可能．

(d) int g2l_x(*VIOS* 変数, *num*)

VIOS 変数x 軸における，グローバル *index num* を，ローカルデータでの *index* に変換する（global to local）．

(e) int g2l_y(*VIOS* 変数, *num*)

上記関数の y 軸版．vsImage, vsCube 変数のみ利用可能．

(f) int g2l_z(*VIOS* 変数, *num*)

上記関数の z 軸版．vsCube 変数のみ利用可能．

2. **VIOS 変数からの情報取得関数：**

VIOS 変数における，幅や高さなどの全体の情報を得るための関数群．

(a) int orig_width(*VIOS* 変数)

VIOS 変数全体における，x 軸方向の最大値を得る．

(b) int orig_heigh(*VIOS* 変数)

VIOS 変数全体における，y 軸方向の最大値を得る．

(c) int orig_depth(*VIOS* 変数)

VIOS 変数全体における，z 軸方向の最大値を得る．

(d) int local_width(*VIOS* 変数)

ローカル計算機が保有する，*VIOS* 変数x 軸方向の最大値を得る (ただしキャッシュ領域を含む) .

(e) int local_heigh(*VIOS* 変数)

ローカル計算機が保有する，*VIOS* 変数y 軸方向の最大値を得る (ただしキャッシュ領域を含む) .

(f) int local_depth(*VIOS* 変数)

ローカル計算機が保有する，*VIOS* 変数z 軸方向の最大値を得る (ただしキャッシュ領域を含む) .

3. **VIOS** 変数のコピー：

型全体のデータを同型式 , 同サイズの変数へコピーするための関数 (ただし , キャッシュ領域のデータも上書きされる) .

(a) void ValCopy(*VIOS* 変数 *1*, *VIOS* 変数 *2*)

vsvalue 型 (ivalue, fvalue) の変数 2 から変数 1 へ全データ領域をコピーする .

(b) void ImgCopy(*VIOS* 変数 *1*, *VIOS* 変数 *2*)

vsImage 型 (iImage, fImage) の変数 2 から変数 1 へ全データ領域をコピーする .

(c) void CubeCopy(*VIOS* 変数 *1*, *VIOS* 変数 *2*)

vsCube 型 (iCube, fCube) の変数 2 から変数 1 へ全データ領域をコピーする .

4. 全体での処理や通信を制御する関数：

特定ワーキングセットに対する処理の抑制や，キャッシュ領域の更新などを行う関数 .

(a) void vsSetBoundary1D(*num*)

1 次元 *parallel* 構文により並列処理を行う際，全体データでの外周 *num* をその処理対象から外します．

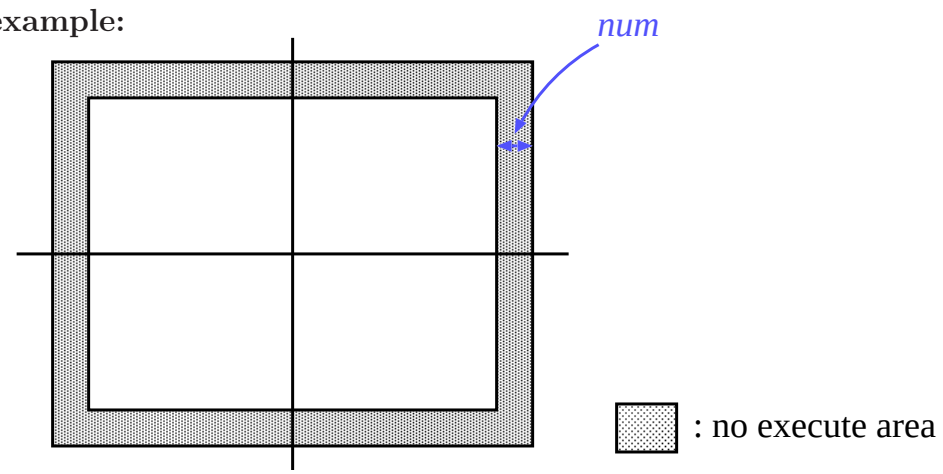
(b) void vsSetBoundary2D(*num*)

上記関数の 2 次元版

(c) void vsSetBoundary1D(*num*)

上記関数の 2 次元版

example:



上記図は，2 次元変数を 4 分割した場合のイメージしている．本来，各計算機が担当する処理領域は，図中心の十文字で区切られた各領域全てであるが，vsSetBoundary により，それぞれ，L 字状に処理を行わない領域が出来ている．

(d) void vsSyncCache(*VIOS* 変数)

VIOS 変数におけるキャッシュ領域を最新のものに更新する．またこの際，全計算機間において同期が取られる．

B.3.4　全体に関する制限

予約語

並列処理部は，外部モジュールとして起動されるため，システム側で利用しているいくつかの変数名，定義語の影響を引き継ぐ．そのため *module* 内において同名の単語を利用することにより，思わぬ誤動作を招く恐れがある．

よって，以下に上げる名前は使用を禁止する．

- 上記拡張変数，および追加関数名
- vios_define.h において定義されている名前
- _vs，および vs で始まる名前
- “module_tab”（グローバル管理クラス名として使用）

実装上の制限

- { ，および } について
  VPE-p では，その性質上，通常の C++プリプロセッサの処理前に処理を行う．また処理速度向上のため厳密な構文解析を行っていない．そのため，{ ，} の対応付けを崩すような#define 定義を利用することは出来ない．
- *module* の引数の数は 256 まで．
- *parallel_ie* を利用する場合は特に，*parallel_ie* 構文内が 65536 文字までに制限される．
- C 言語による記述部に関して
  基本的に MT-Safe な関数ならば利用可能．ただしヘッダをインクルードする必要と，vs_makemodule スクリプトに，ライブラリをリンクするよう指示する必要はある．

B.4　その他

B.4.1　細分割実行の効果に関して

- 細分割実行の効果は，初回時以降の *parallel* 構文には影響がない (すでにデータがあるため)

B.4.2　モジュールファイルのコンパイル

./vs_makemodule 関数名 ファイル名

とすることにより，$VS_LIBRARY_PATH/lib_vs モジュール名.so，または，環境変数$VS_LIBRARY_PATH が NULL の場合は，$HOME/.vios_lib/lib_vs モジュール名.so が生成される．