社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－ARC－172 (9)
2007－HPC－109 (9)
2007／3／1

# 分散メモリ向けデータ並列言語OpenMPDの設計と実装

李 珍泌[†] 佐藤三久[††] 朴 泰祐[††]

分散メモリ型並列システムの標準的なプログラミングモデルであるMPIは複雑な仕様を持ち，プログラマに大きな負担となる．分散メモリ並列システムに対し指示文を加えることによりデータ並列プログラムを記述するプログラミングモデルを提案し，その処理系OpenMPDを実装した．プログラミングモデルの設計では，MPIおよびOpenMPとの混在を考慮した．いくつかのアプリケーションにおいては，指示文の記述によって逐次コードの変更をほとんど行わず，簡便に並列化を行うことができた．性能評価の結果，指示文によるわずかな記述により良好な性能が得られることを示した．

# Design and Implementation of OpenMPD parallel programming language for distributed memory

JINPIL LEE,[†] MITSUHISA SATO[††] and TAISUKE BOKU [††]

MPI is a de facto standard for parallel programming on a distributed memory system. However parallel programming with MPI is often time consuming and difficult. In this research, we propose a simple programming model, named OpenMPD for a distributed memory system with some directives and implemented the compiler. OpenMPD provides directives for data parallelization, which allow incremental parallelization for a serial code. It allows the user to use MPI and OpenMP for complicated parallel programs. The result of evaluation shows that OpenMPD achieves good performace only by adding a few directives.

## 1. はじめに

今日では，高性能並列計算のプラットフォームとして，PCクラスタを始めとする分散メモリ型のシステムが多く普及している．分散メモリ上の並列プログラミングにおいては，MPI(Message Passing Interface)[2]が，標準的なプログラミング環境として用いられている．MPIでは，プロセッサ間のデータの送受信をプログラマが明示的に記述することによって並列プログラミングを行う．細かい最適化による効率の良い並列プログラムを書くことができるが，プログラミングが難しくなり，並列システムの利用コストを増大させることになっている．これまで，分散メモリ向けの並列プログラミング言語として，HPFをはじめとする様々なプログラミング言語が提案されてきたが，普及にいたっていない．一方，共有メモリ型計算機では，OpenMP[1]が用いられるようになってきた．OpenMPは指示文によって並列化する部分を記述し，処理系が並列化を行うため，並列化が容易となる利点がある．しかし，OpenMPは基本的には共有メモリモデルに対するものであり，分散メモリに対してはソフトウエア分散共有共有メモリシステムを用いた実装Omni/SCASHなどがあるものの性能面で問題がある．今後，クラスタにおいてもノードのマルチコア化が進み，並列プラットフォームが複雑になるため，並列プログラミング環境の重要性は増している．

本稿では，分散メモリ型並列システムに対して，既存の逐次プログラムをベースに指示文を用いて並列化を行うための簡便なプログラミングモデルを提案し，その処理系OpenMPDについて述べる．我々の対象とするアプリケーションは並列処理により高性能計算が必要な科学技術計算である．多くの科学技術計算アプリケーションでは，配列などのデータに対し同一の処理を行うデータ並列が主な処理になる．MPIを用いて並列化した場合，分散された配列の袖領域の交換など典型的な処理が並列化の主な作業になる．OpenMPDでは，既存コードの変更をなるべく行わずに，これらの典型的な処理を指示文で行うことができる．従来，MPIを用いて行われてきた分散メモリ型のシステムでの並列プログラミングを簡単に行うために，簡便なプログラミングモデルを提供し，プログラマの負担を減らすことを目的とする．

これまで，分散メモリ向けに多くの並列プログラミングモデルが提案されてきた．最近では，CAF(Co-Array Fortran)[6]やUPC(Unified Parallel C)[5]のよ

† 筑波大学 第三学群 情報学類
College of Information Science, University of Tsukuba
†† 筑波大学 大学院 システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

うな既存のプログラミング言語を拡張した分散メモリ向けの並列言語が注目されている．これらのような並列言語は多様な機能を提供する反面，仕様が複雑となっている．抽象化された並列化の記述は並列化の大部分を処理系によって行うようにするため，実装が見えにくくなり，プログラマが並列化のチューニングを行うことが困難になりがちである．OpenMPD の設計に際しては，直観的で簡便なプログラミングモデルを提供することを心がけた．我々は，既に同じようなコンセプトに基づくプログラミングモデル OpenMPI[10] を提案した．OpenMPD はこれを検討・発展させ，実装したものである．

まず，次章にて OpenMPD の概要について述べ，3 章では実装した処理系によるコード変換について説明する．4 章では幾つかのアプリケーションを OpenMPD で記述し，並列化されたプログラムの性能評価を行う．5 章では OpenMPD の持つ限界と問題点を挙げ，最後にまとめと今後の課題を述べる．

## 2. 分散メモリ向けデータ並列言語 OpenMPD の概要

OpenMPD は MPI によってデータ並列を行うときに頻繁に使われる手法を機能単位でまとめ，指示文として提供している．プログラマはこれらの機能が必要な場合に MPI 関数の代わりに指示文を記述することでプログラミングのコストを抑えることができる．OpenMPD は MPI プログラミングを簡単に記述する手段を提供することで分散メモリ型並列システムに対し簡単なプログラミングモデルを提供する．

### 2.1 OpenMPD の設計方針とプログラミングモデル

以下に，OpenMPD の設計方針／目標を挙げる:

- 明確なプログラミングモデルを提供して，プログラマが動作を十分に理解し，性能チューニングができるような仕様にする．
- 主に，典型的なデータ並列プログラムの並列化を支援する．
- 指示文により並列化を行い，なるべく既存の逐次コードの変更を行わなくても良いようにする．
- MPI との混在するプログラム，あるいはノード内の OpenMP による並列化と組み合わせることができる柔軟性を提供する．

OpenMPD は自動並列化を行う言語ではない．データの依存性の分析や解決をプログラマに任せ，並列化に必要な指示文が明示的に記述されることを前提とする．OpenMPD の指示文は処理系に並列化に必要な情報を提供するためのものであり，処理系は指示文を並列コードに変換するたけである．そうすることで並列化されたプログラムの性能が処理系に依存されないようにする．

OpenMPD の実行モデルは，SPMD(Single Program Multiple Data) モデルである．各プロセッサは，同じプログラムを main プログラムからほぼ同じ順序で実行し，異なるデータを操作することを基本とする．大域宣言されたデータは，各プロセッサに重複して割り当てられる．分割して処理される配列に関しては，指示文の記述によってプロセッサに割り当てられる領域を指示する．

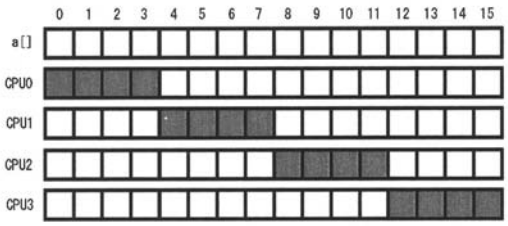


**図 1** 配列の割り当て

図 1 に 1 次元配列 a を 4 個のプロセッサに割り当てる様子を示す．各プロセッサは元の逐次コードの配列宣言に変更を加えず，そのまま宣言する．各プロセッサがブロック分割により異なる領域を処理するように制限することで配列を割り当てる (図の灰色の領域がプロセッサによって処理される)．割り当てられていない領域 (白い領域) を参照する場合，その領域が割り当てられたプロセッサと同期を行ってデータの整合性を確保しなければならない．このような場合，プログラムの実行コードの中に同期・通信を行う操作を指示文として明示的に記述する．同期の操作には以下のものがある．

- sleeve 領域の同期
  配列のデータを並列処理する場合には，隣接した要素に依存する計算を行う場合がある．その例を図 2 に示す．白い領域が各プロセッサによって並列に処理される．各要素を処理するとき，隣接する要素が参照されるとする．その場合，割り当てられた領域の境界の要素を処理するために，灰色の領域が参照される (以後，この領域を sleeve 領域と呼ぶことにする)．しかし，この領域は他のプロセッサによって処理されるため，自身のプロセッサでは有効な値を持たない．この操作では，灰色の領域が実際割り当てられているプロセッサからデータを取得する (sleeve 領域の同期を行う)．
- 配列の集約
  並列に処理した配列は割り当てられた領域しか有効な値を持たない．配列の全ての値が必要になる場合，他のプロセッサから配列のデータを集めなければならない．以後，この作業を配列の集約と呼ぶことにする．
- 変数の同期

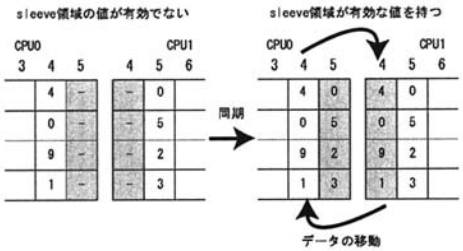


**図 2** sleeve 領域の同期

同期の元になる値を持つプロセッサが他の全てのプロセッサに変数の値を転送する．一つのプログラムが各プロセッサにコピーされ実行されるため，同じ名前の変数でもプロセッサによって異なる値を持つ可能性がある．特定のプロセッサの値を他のプロセッサで使うためにはプロセッサの間で通信を行い，同期を行わなければならない．

- リダクション操作
  各プロセッサが持つ値を全て足し合わせるか，最高値を求めるなどのリダクション演算が必要になる場合もある．変数の同期と同様，リダクション演算にもプロセッサ間の通信が必要である．各プロセッサが他のプロセッサの値と自身の値を交換し合い，計算を行う．

プロセッサ間の並列実行は，OpenMP と同様に for 指示文で行うことができる．各イタレーションは各プロセッサに分配されて行われる．ループ内において，配列の要素を処理する場合には配列の担当する要素について計算を行わなければならない．for 指示文では配列要素に整合して (affinity) ループを割り当てる機能を持つ．

次の節で OpenMPD の指示文の記述例を示す．

### 2.2 プログラム例

OpenMPD は逐次コードに指示文を記述することで並列化を行う．図 3 に OpenMPD の指示文の記述による並列化の例を示す．

図 3 はデータ並列の典型的な手法を指示文によって記述したものである．指示文 distvar を記述することで配列 array を 2 次元方向に分割し，各プロセッサに割り当てる．配列を並列に処理するために指示文 for を 2 次元要素を処理する外側の for 文に対して記述している．affinity に配列の名前を記述して，distvar による配列の割り当てに整合してループを割り当てる．最後に，指示文 gather を用いて配列を集約することを記述している．

もし内側の for 文に対して指示文 for を記述した場合でも OpenMPD の処理系はコードを生成する．しかし，distvar による配列の割り当てと整合性が取れていないため，プログラムは間違った結果を返す．このようにプログラマは並列化が正しく行われるように

```
int array[10][10];

#pragma ompd distvar(var = array;dim = 2)

main(){
  int i,j;

  #pragma ompd for affinity(array)
  for(i = 0; i < 10; i++)
    for(j = 0; j < 10; j++)
      array[i][j] = func(i, j);

  #pragma ompd gather(var = array)
}
```

**図 3** OpenMPD による並列化の例

指示文を記述する責任を持つ．

### 2.3 OpenMPD の指示文

以下に OpenMPD が提供する指示文の記述形式を説明する．

#### 2.3.1 distvar 指示文

#pragma ompd distvar(var=*list*; dim=*size1*; sleeve=*size2*)

distvar は配列を各プロセッサに割り当てる方法を記述する．var，dim，sleeve に値を代入することで割り当ての仕方を記述する．var は対象の配列のリストである．dim には分割する次元を設定する (メモリ上で連続して配置される次元を 1 とする)．sleeve で sleeve 領域の大きさを設定する (図 2 の場合，大きさは 1 である)．dim と sleeve は省略することができる．デフォルトの値は dim が 1，sleeve が 0 である (この場合，各プロセッサは sleeve 領域を持たない)．OpenMPD は大域の配列に関するデータ並列を支援し，distvar の対象とする配列は大域に宣言されたものだけとする．distvar 自身も対象の配列が全て宣言された後に大域宣言として記述する．

#### 2.3.2 for 指示文

#pragma ompd for affinity(*var*) reduction({*op*:*list*})

for は for 文を並列に実行することを記述する指示文である．対象とする for 文の直前に挿入する．affinity に for 文の中で並列処理される配列の名前を記述する．処理系は各プロセッサが対象配列の領域を均等に処理し，割り当ていない領域にアクセスしないようにループの反復回数を割り当てる．affinity が省略されている場合，処理系はループの反復を各プロセッサに均等に割り当てる．この場合，配列が割り当てられていない領域の要素を参照する可能性があるため，計算の正しさは保障されない．reduction は for 文の中でリダクション演算が行われている場合に記述する (そうでない場合は省略可能)．*op* にリダクション演算子を，*list* に結果を収納する変数のリストを設定する．並列化された for 文で行われるリダクション演算は各プロセッサで不完全な値を持つため，このように指示文を

記述することで値を完全なものにしなければならない.

#### 2.3.3 sync_sleeve 指示文

#pragma ompd sync_sleeve(var=*list*)

sync_sleeve は sleeve 領域の同期を記述する指示文である. 同期が必要で, C 言語の文 (*statement*) が書ける所なら, どこにでも挿入することができる. var は同期を行う配列のリストである. distvar によって記述された配列の中で, sleeve の値が 1 以上のもの (sleeve 領域を持つもの) でなければならない.

#### 2.3.4 gather 指示文

#pragma ompd gather(var=*list*)

gather は配列の集約を記述する指示文である. 使い方は sync_sleeve と同様であるが, 対象となる配列は sleeve 領域を持たないものでも構わない. gather の記述によって全てのプロセッサで配列が集約され, 全ての範囲において有効な値を持つ.

#### 2.3.5 sync_var 指示文

#pragma ompd sync_var(var=*list*;master=*num*)

sync_var は変数の同期を記述する指示文である. var には同期を行う変数のリストが設定される. master には同期の元になる値を持つプロセッサ番号が設定される. 0 以上の整数で, 全てのプロセッサ数より大きい値でも構わない (single の説明を参照). master は省略可能であり, デフォルトの値は 0 である.

#### 2.3.6 reduction 指示文

#pragma ompd reduction({*op*:*list*})

reduction はプロセッサ間で行われるリダクション演算を記述する指示文である. 使い方や利用できる演算子は for と同じである. 文 (*statement*) が書ける場所ならどこにも挿入することができる.

#### 2.3.7 single 指示文

#pragma ompd single(master=*num*)

single は指示文に続く文を特定のプロセッサだけで実行するようにする指示文である. 対象となる文の直前に記述する. 文は master に設定された番号を持つプロセッサだけで実行される. 複数の異なる文を異なるプロセッサ実行することでタスク並列化を行うこともできる. master は 0 以上の整数で全プロセッサ数より大きい値でも構わない. 省略可能であり, デフォルトの値は 0 である. single の後に同じ値の master を用いた sync_var を記述することよって実行結果の同期を行うことができる.

### 2.4 OpenMPD のライブラリ関数

OpenMPD は MPI と同様, SPMD 実行モデルで実行されるため, 全てのプロセッサの数と自分のプロセッサ番号が必要になる場合がある. 実行環境に関する情報を取得するため, OpenMPD は以下の関数をプログラマに提供する.

- int ompd_get_num_nodes(void);
  ompd_get_num_nodes は全体のプロセッサ数を返す関数である. 戻り値は 1 以上の整数である.
- int ompd_get_node_num(void);
  ompd_get_node_num は自分のプロセッサ番号を返す関数である. 戻り値は 0 から (全プロセッサ数-1) の間の整数である.

これらの関数は MPI 関数によって調べた場合と同じ値を返す. MPI 関数のように長い引数を書く必要なく, 簡単に実行環境を調べることができる.

### 2.5 MPI との併用

OpenMPD は OpenMPD の指示文の記述と一緒に MPI 関数を用いて並列化することを許す. 但し, プログラマは OpenMPD で記述した並列化を矛盾しないように MPI プログラミングを行わなければならない. OpenMPD は MPI を用いて通信を行うため, 処理系によって MPI の初期化と終了が行われる. そのため, プログラマが並列化に直接必要な MPI 関数を書くでけで十分である. OpenMPD では記述できな部分を MPI 関数を用いて並列化することで性能のチューニングを行うことができる.

## 3. OpenMPD の処理系

OpenMPD の処理系は Omni OpenMP Compiler[3] をベースに実装した. 指示文から並列化に関する情報を得て逐次コードを並列コードに変換する. 領域割り当てに必要な情報の計算やプロセッサ間通信は MPI 関数で実装した OpenMPD のランタイムライブラリ関数が行う. 処理系が変換する並列コードはランタイムライブラリ関数を利用している. 並列コードとランタイムライブラリを合わせて MPI 並列プログラムが完成する. OpenMPD の処理系がどのようにコード変換を行うかを以下に示す.

図 3 に OpenMPD の処理系によって変換された図 3 の並列コードを示す. __ompd_で始まる変数は並列化に関する情報を収納するために処理系が宣言したものである. __ompd_で始まる関数は OpenMPD のランタイムライブラリ関数である. プロセッサ間の通信や並列化に関する情報の計算が必要となる指示文に対して, 処理系はコードに OpenMPD のランタイム関数を挿入する.

処理系は図 1 のように配列の割り当てを行う. 各プロセッサで割り当てられる範囲の上限と下限を計算し, __ompd_idx_配列の名前_upper と __ompd_idx_配列の名前_lower に収納する.

for 文の並列実行は各ノードで異なる範囲の反復を行うことで実現する. ループ変数が 1 から 100 まで 100 回反復される for 文の場合, 1 から 50 の反復と 51 から 100 までの反復を二つのプロセッサに分けて実行する, という具合である. このように, ループ変数が持つ値を制限して for 文を並列に実行する. affinity が記述された場合は配列の割り当てに整合して, ループ変数の上限と下限を計算する. 図 3 の __ompd_for_ループ変数の名前_upper と __ompd_for_ループ変数の

```
int __ompd_idx_array_lower;
int __ompd_idx_array_upper;
void *__ompd_arg_array_arr_left;
extern void __ompd_init();
extern void __ompd_record_var_without_sync();
extern void __ompd_finalize();
int __ompd_myid;
int __ompd_nproc;
extern int __ompd_get_llimit_eq();
extern int __ompd_get_ulimit_neq();
extern void __ompd_allgather_n_n();
int array[10][10];

int main(argc,argv)int argc;
char **argv;
{
 auto int __ompd_for_i_lower;
 auto int __ompd_for_i_upper;
 auto int i; auto int j;
 __ompd_init(&argc,&argv);
 __ompd_record_var_without_sync(
   &__ompd_idx_array_lower,
   &__ompd_idx_array_upper,
   10,array,10,4,
   &__ompd_arg_array_arr_left);
 __ompd_for_i_lower=
 __ompd_get_llimit_eq(0,__ompd_idx_array_lower);
 __ompd_for_i_upper=
 __ompd_get_ulimit_neq(10,__ompd_idx_array_upper);
 for(i=__ompd_for_i_lower;i<__ompd_for_i_upper;i+=1)
  for(j=0;j<10;j++){
   (*((*((array)+(i)))+(j)))=func(i,j);
  }
 __ompd_allgather_n_n(__ompd_arg_array_arr_left,
   (void *)(array),4,
   __ompd_idx_array_upper-__ompd_idx_array_lower,
   10,10);
 __ompd_finalize();
}
```

図 4　処理系によって変換されたコード

名前_lower が処理系によって計算されるループ変数の上限と下限である．

single による単独実行はプロセッサ番号を獲得し，master の値と同じかを比べて実行するかどうかを決定する if 文を挿入することで実現する．プロセッサ番号は MPI 関数によって獲得されるもので 0 から (プロセッサ数-1) の間の整数である．master の値がプロセッサ数以上である可能性は十分にある．そのような場合でも実行するプロセッサを割り当てるために if 文でプロセッサ番号と比較される値を (master % (全体のプロセッサの数)) にしている．

## 4. 性能評価

ここでは実際のアプリケーションを OpenMPD で並列化し，その性能を評価する．Dennis Cluster の 8 ノード (スレッドを利用しないので 8 プロセッサ) を用いて評価を行った．ノードの構成は表 4 の通りである．

性能評価には NAS Parallel Benchmark[7] の The Embarrassingly Parallel Benchmark(以下 EP) と Conjugate Gradient Benchmark(以下 CG)，姫野ベンチマーク[8] を用いた．OpenMPD は典型的なデータ並列に関する指示文を提供するものであるため，適用範囲は制限される．従って，対象のアプリケーションが OpenMPD で並列化することができるかどうかを判断することは重要である．性能評価に用いたアプリケーションは配列を用いてデータの計算を行っているため，OpenMPD で並列化することができる．

表 1　Dennis Cluster ノードの構成

| 項目 | 名称 |
|---|---|
| CPU | Intel Xeon Processor 2.4Ghz x 2 |
| Memory | 1GB |
| Network | 1000BASE-T |
| MPI Library | LAM 7.0.6 |
| OS | Linux 2.6.9-42.0.3.ELsmp (x86) |

OpenMPD により並列化されたアプリケーションの性能を以下に示す．

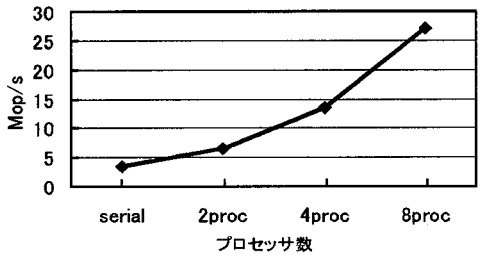


図 5　NPB EP の実行結果 (class A)

図 5 に EP の実行結果を示す．EP の並列化のため，乱数を生成する for 文を並列に実行するように指示文を記述した．for 文の各反復は独立的に実行することができ，プロセッサ間通信はほとんど発生しない．図 5 は EP の特徴をよく表している．並列化のオーバーヘッドが小さいため，プロセッサの数に比例して性能が向上した．このように独立的な処理を行うアプリケーションは OpenMPD を用いて並列化することは MPI プログラムを直接書くより簡単で，理想的な性能向上を得ることができる．

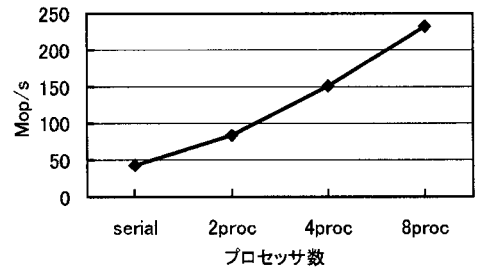


図 6　NPB CG の実行結果 (class B)

図 6 に並列化した CG の実行結果を示す．ベクトル

データを持つ1次元配列を各プロセッサで並列に処理するように指示文を記述し，並列化を行った．EPの場合より性能の向上が低いのはプロセッサ間の通信が並列化のオーバーヘッドになっているためである．並列化したCGは配列の集約とリダクション演算が必要である．そのため，プロセッサ間通信が頻繁に発生する．また，実行時間のほとんどを占めるfor文を並列処理することができたEPはと異なり，CGの場合並列に実行することができないループが存在する．そのような部分は各プロセッサで逐次的に実行される．この二つがCGの並列化のオーバーヘッドとなる．

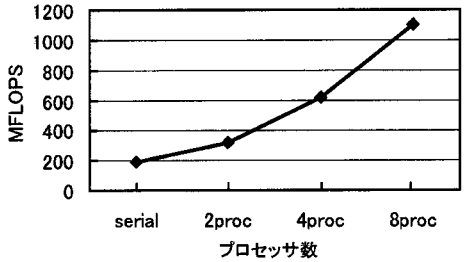


**図7** 姫野ベンチマークの実行結果 (size MIDDLE)

図7に並列化した姫野ベンチマークの実行結果を示す．データを持つ3次元配列を並列に処理するようにOpenMPD指示文を記述した．隣接した要素を参照する計算が行われるため，各プロセッサが大きさ1のsleeve領域を持つようにしている．sleeve領域の要素が繰り返し参照されるため，その同期が並列化のオーバーヘッドとなる．

CGと姫野ベンチマークの場合も，EPより低いというものの，プロセッサの数が増加することによって性能が向上している．実行結果はOpenMPDがこれらのアプリケーションの並列化を行うに十分な機能を提供することを実証している．このように，OpenMPDはデータ並列に関する典型的な並列化を支援するため，限られた機能で多くのアプリケーションの並列化を記述することができる．

## 5. OpenMPDの問題点と課題

OpenMPDが持つ問題点を以下に示す．

- OpenMPDで配列を多次元で分割ことができない．プロセッサを有効に利用するためには多次元分割により，領域をより細かく分割することが望ましい．
- OpenMPDで配列を並列処理するとき，全プロセッサに配列の全ての領域が宣言される．しかし，プロセッサが処理する領域はその中の一部であるため(図1参照)，無駄にメモリを消費することになる．分散メモリ型並列システムのメリットの一つは膨大なデータを各プロセッサに割り当て処理することができることである．現在のOpenMPDの仕様ではこのようなアーキテクチャの特徴を生かすことができない．各配列が割り当てられる部分だけを持つようにして，メモリを効率的に利用するようにしなければならない．
- 並列実行するfor文の中で異なる範囲で割り当てられた配列が参照される場合，affinityの指定が困難である．異なる範囲で割り当てられた配列を一緒のfor文で処理することができるように，データの整合性に関する規則とその実装が必要である．

## 6. まとめ

OpenMPDはMPIプログラミングを簡単に行う手段として，分散メモリ型並列システムに対する簡単なプログラミングモデルを提供する．多くの科学技術計算アプリケーションで用いられるデータ並列の典型的な手法を指示文を用いて記述することができる．

今後，OpenMPDの持つ問題点を解決すると共に，ノード内並列化をOpenMPで記述した場合のスレッドセーフティを確保してハイブリッドなプログラミングモデルを提供することを目指す．

**謝辞** 本研究を行うにあたって多くのアドバイスを頂いた筑波大学HPCS研究室の皆様に厚く御礼申し上げます．

## 参考文献

1) OpenMP.org, http://www.openmp.org
2) The Message Passing Interface (MPI) standard, http://www-unix.mcs.anl.gov/mpi/
3) Omni OpenMP Compiler Project , http://phase.hpcc.jp/Omni/home.ja.html
4) SCASH, http://www.pccluster.org/score/dist/score/html/en/reference/scash/index.html
5) Unified Parallel C, http://upc.gwu.edu/
6) Co-Array Fortran, http://www.co-array.org/
7) NAS Parallel Benchmarks, http://www.nas.nasa.gov/Resources/Software-/npb.html
8) 姫野ベンチマーク, http://accc.riken.jp/HPC/HimenoBMT/
9) Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano and Yohio Tanaka, "Design of OpenMP Compiler for an SMP Cluster", Proc. of 1st European Workshop on OpenMP EWOMP'99, pp. 32-39, 1999.
10) Taisuke Boku, Mitsuhisa Sato, Masazumi Matsubara, Daisuke Takahashi, "OpenMPI - OpenMP like tool for easy programming in MPI", Proc. of 6th European Workshop on OpenMP (EWOMP'04), pp. 83-88, 2004.