OpenMP型並列プログラミングモデルのクラスタ計算環境への適用

上羽琢磨
新實治男

あらまし

本研究では，OpenMP型並列プログラミングモデルのクラスタ計算環境への適用化研究を行った．OpenMPは，共有メモリ型並列処理システム用のAPIとして普及しつつあり，分散メモリ型並列処理システム用のMPIに比較して，プログラミングのコストも安価である．そこで，このOpenMPを用いて記述された並列プログラムを入力とし，MPIプログラムへの変換を行うトランスレータの開発を行うことで，新たな並列プログラミングの支援方式の提案を行っている．

本トランスレータでは，入力のOpenMPプログラムにおける「sharedデータ」を，MPIプログラムにおける「privateデータ」へ変換するための一定の規則を定義した．そして，データの依存性解析の結果から，必要なMPI通信関数の挿入を行い，さらに，配列データなどに対しての通信命令を削減するなどの最適化を行うことで，より並列効果の高いMPIプログラムの生成を図った．

各種ベンチマークプログラムにより，本トランスレータを用いた並列実行方式の性能を評価した結果，局所性の高いOpenMPプログラムを入力コードとした場合には，出力コードとして得られたMPIプログラムにおいても，十分な並列処理効果がみられた．また，クラスタシステムにおけるOpenMPプログラムの実行環境の1つである，Omni/SCASH[注1)]と比較しても，総じて高い速度向上効果を得ることができた．これらのことから，本トランスレータを用いた新たな並列実行方式の有効性を示すことができた．

1　はじめに

近年，マイクロプロセッサやネットワーク技術の高性能化によって，汎用PCを相互接続したクラスタシステムが並列処理のプラットフォームとして普及してきており，並列処理システムを比較的安価に構築できるようになった．

一般にクラスタシステムのような分散メモリ型並列処理システムでは，プログラムを並列実行するためには，MPIのようなメッセージ通信ライブラリを用いて，プロセス間のメッセージ通信命令をプログラムに明示的に挿入する必要がある．しかし，ユーザにとって，このメッセージ通信命令の挿入は，他の並列処理用APIと比較しても負担が大きく，正しく動作する並列

プログラムを開発するだけでも相当の時間を要する．さらに，その並列プログラムをより並列処理効果の高いものに最適化するためには，データの分散配置の方法や依存関係を考慮して，それに適したMPI関数の選択と利用方法を考える必要があり，プログラム開発にはさらに多くの時間を必要とする．また，自動並列化コンパイラなどを用いるアプローチも考えられるものの，やはり最適なMPIプログラムを生成するためには，並列性の抽出やデータ依存性の解析などに洗練された手法を採用する必要があり，MPI関数利用の最適化技術も含め，その実現には大きな困難が予想される．

そこで本研究では，共有メモリ型並列処理システム用のAPIとして近年利用の広がりを見せているOpenMPに着目した．OpenMPはMPIと比較してプログラミングコストも安価であり，並列プログラミングにおける標準的なパラダイムとなる可能性を有している．本研究では，OpenMPを用いて並列性を記述したプログラムを入力とし，これを分散メモリ型並列処理システム用のMPIプログラムへ変換を行うトランスレータの開発を行う．この並列実行方式により，ユーザの負担の軽減を図りながら，クラスタシステムの持つ潜在的な並列実行性能を最大限に引き出すことを目指す．また本方式により，OpenMPで記述された並列プログラムは，共有メモリ型と分散メモリ型の両方の並列処理システムにおける共通のプログラミングパラダイムとなり，並列処理の適用範囲の拡大に寄与するものと期待できる．

2　並列処理用API

本章では，並列処理システム上で並列実行を行う際に必要となる並列処理用APIの代表的なものについて，それぞれの特徴と互いの相違点について述べる．

2.1　並列処理用API

並列処理システム上でプログラムを並列実行するためには，OpenMPやMPIなどの並列処理用APIを利用したプログラミングが一般的である．これらの並列処理用APIは，並列実行のシステムモデルの違いにより，以下の2種類に分類される．

⑴　共有メモリ型並列システム

共有メモリ型並列システムにおいての並列実行では，主にp-threadやSolaris-threadなどのライブラリ，あるいは，OpenMPなどが利用されている．これらは，プログラミング手法により，さらに2種類に分類することが出来る．

OpenMPを用いた並列プログラミングは，既存のプログラムにコンパイラ指示文を挿入するモデルである．従って，並列プログラムにおける並列実行箇所は構造的に記述される．このことから，逐次プログラムに対して大規模な変更を必要とせず，プログラミングコストはそれほど大きくない．

一方，p-threadやSolaris-threadは，並列性を非構造的に記述するため，OpenMPと比較した場合，ユーザの負担が大きい．また，コンパイラの最適化という視点から見ても，プログラム構造の解析が困難であり，コンパイラの負担も同様に大きい．

⑵ 分散メモリ型並列システム

クラスタシステムのような分散メモリ型並列システムでは，MPIやPVM，またHPFなどを利用するのが一般的である．近年では，実行効率という点から，MPIを用いて並列処理を行うユーザが最も多い．MPIではプロセス間のデータ共有や同期をメッセージ通信で行うために，明示的に処理の負荷分散方法や通信命令の付加をユーザに要求する分，実行効率を考慮したプログラミングが可能となっている．

MPIはさまざまな通信方式を支援している．基本となる1対1通信を行う命令にも，通信モードのセマンティックスの違いにより，standardモード，readyモード，synchronousモードの３方式（通信モード）があり，さらにそれぞれのモードにブロッキング通信とノンブロッキング通信の２方式（通信方式）が用意されている．このように，通信モードや通信方式が豊富でユーザは柔軟に選択できるという一方で，それぞれのモードや方式の区別，さらに最適である関数の選定など，ユーザの負担も大きい．

2.2 OpenMPとMPIの相違点

本節では，2.1で示したOpenMPとMPIのプログラミングモデルの比較を行う．両者間の大きな相違点として以下の2項目を挙げることができる．

（ｉ）並列実行環境

（ii）プログラミングコスト

まず，（ｉ）の「並列実行環境の違い」に起因するプログラミングモデルや並列実行モデルの相違点について述べる．

1. OpenMPは，fork-joinモデルであり，parallel構文を用いて並列実行する部分を明示的に示し，その動的な有効範囲でのみ，masterスレッドとmasterスレッドによって生成された複数のスレッドが並列実行を行う．並列実行の出口部分で各スレッドがjoinされ，その後はmasterスレッドのみで実行される．一方，MPIでは，すべてのプロセスが実行開始と共に生成され，実行終了までプロセス数が変化することはない．
2. OpenMPでは，プログラム中に定義されているデータはデフォルトではsharedとして全スレッドで共有されるため，データの値の更新の際にもスレッド間の通信は必要としない．各スレッドが固有のデータを保持したい場合は，コンパイラへの指示節として**private**と指定する必要がある．一方，MPIでは，分散メモリ型システムを対象としているため，定義されるデータは各プロセス固有のprivateデータとして確保される．共有したいデータの値の更新や同期を行いたい場合は，**MPI_Send**や**MPI_Recv**といった通信用関数を用いる

必要がある．

3. OpenMPでは，sharedデータに対しての排他制御が主なオーバヘッドと考えられる．これに対してMPIでは，プロセス間通信による通信オーバヘッドが性能低下の主な原因として挙げられる．従って，MPIプログラムでは無駄な通信をいかに減らし，効率の良い通信ができるかが非常に重要である．

一方，（ii）のプログラミングコストに関しては，円周率πを次の定積分，

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

により求めるプログラムを例として述べることにする．このプログラムの逐次版を図2.1，OpenMP指示文を挿入した並列プログラムを図2.2，そして，MPIライブラリを利用する並列プログラムを図2.3にそれぞれ示す[1]．また，図2.2，2.3に関しては，逐次版に対して記述を追加した部分を青色，変更した部分を赤色で示す．

図2.1のプログラムは，7行目から10行目のforループで**x**の閉区間[0,1]を**num_steps**個に分割し，その区間毎にできる微小台形の面積を累積加算することでπの近似値を求めるものである．このうち，7行目から始まるforループの部分をOpenMPとMPIのそれぞれで並列化することにする．

```
 1  static long num_steps = 100000;
 2  double step;

 3  void main( ){
 4    int i;
 5    double x, pi, sum=0.0;

 6    step = 1.0/(double)num_steps;

 7    for ( i = 1; i <= num_steps ; i++ ){
 8      x = ( i - 0.5 )*step;
 9      sum += 4.0 / ( 1.0 + x*x );
10    }
11    pi = step *sum;
12  }
```

図2.1 円周率πを求める逐次プログラム

図2.2のOpenMPプログラムでは，9行目で**omp_set_num_threads**関数を用いてスレッド数を**NUM_THREADS**(この場合は，4スレッド)に設定している．そして，10行目の**omp parallel for**により，その対象となるループをWork-sharing型で並列実行することを指示し，さらにデータ**sum**に対してリダクション演算を行うこと，データ**x**が各スレッドでprivateであることを，それぞれ指示節として記述している．各スレッドに対してのループのスケジューリング方法は，**schedule**節を用いて指示がない場合は，OpenMPコンパイラがコンパイル時に割り当てる（Omni/OpenMPコンパイラ[2]では，ブロック分割として割り当てる）．このように，

```
1  #include < omp.h >
2  static long num_steps = 100000;
3  double step;
4  #define NUM_THREADS 4
5  void main( ){
6    int i;
7    double x, pi, sum=0.0;

8    step = 1.0/(double)num_steps;
9    omp_set_num_threads ( NUM_THREADS );
10 #pragma omp parallel for reduction ( + : sum ) private ( x )
11   for ( i = 1; i <= num_steps ; i++ ) {
12     x = ( i - 0.5 )*step;
13     sum += 4.0 / ( 1.0 + x*x );
14   }
15   pi = step *sum;
16 }
```

図2.2 円周率πを求めるOpenMPプログラム

```
1   #include " mpi.h "
2   static long num_steps = 100000;
3   double step;

4   void main( ){
5     int i;
6     double x, pi, sum, sum_local;
7     int my_rank, p;
8     int my_lw, my_hi, sched_blk;

9     MPI_Init ( &argc, &argv );
10    MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );
11    MPI_Comm_size ( MPI_COMM_WORLD, &p );

12    step = 1.0/(double)num_steps;

13    sched_blk = num_steps / p;
14    my_lw = sched_blk * my_rank + 1;
15    my_hi = sched_blk + my_lw;
16    for ( i = my_lw ; i <= my_hi ; i++ ){
17      x = ( i - 0.5 )*step;
18      sum_local += 4.0 / ( 1.0 + x*x );
19    }
20    MPI_Reduce( &sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM,
                  root, MPI_COMM_WORLD );
21    pi = step *sum;
22    MPI_Finalize( );
23  }
```

図2.3 円周率πを求めるMPIプログラム

OpenMPはユーザに対しては並列化を行いたい対象ブロックの先頭に**omp parallel**構文を挿入することのみを要求する．従って，ユーザの負担は逐次プログラムと比較しても，それほど大きくないといえる．

次に，図2.3のMPIプログラムに注目する．まず，MPIの初期化を行うために，**MPI_Init**を呼び出す（9行目）．その後，**MPI_Comm_rank**，**MPI_Comm_size**を呼び出し，コミュニケータ**MPI_COMM_WORLD**内の各プロセスのランク番号と全プロセス数を**my_rank**，**p**に代入している（10，11行目）．そして，並列化の対象となるforループの直前に各プロセスへの分散方法を記述している．MPIではOpenMPコンパイラとは異なり，処理の分散方法や各プロセ

スへの割当て方法をコンパイラが自動的には行わないため，ユーザが自ら記述しなければならない．そして，対象ループの終了後，各ループで累積加算されたデータ**sum_local**に対して**MPI_Reduce**関数を用い，全プロセスによるリダクション演算を行っている．最後に，MPIでの並列実行の終了を示す**MPI_Finalize**を呼び出している．

ここで，OpenMPプログラムとMPIプログラムを比較すると，OpenMPプログラムは逐次プログラムに4行付け加えるだけでよいのに対して，MPIプログラムは，11行の追加を必要としている．また，単なる追加だけではなく，各プロセスでのforループの初期値（**my_lw**）と終了条件の値 (**my_hi**) の設定，さらに**MPI_Reduce**関数の制約により，データ**sum**を各プロセスの一時的な格納場所となるデータ**sum_local**へと変更することも必要である．このように，ユーザへの負担，すなわちプログラミングコストとしての両者の比較においては，明らかにOpenMPのほうが小さい．また，MPIを理解し，逐次プログラムを並列版のプログラムに書き換えるためには，ある程度の時間が必要であることが判る．さらに，速度向上のための最適化を行うとなれば，それ以上の時間が必要となり，ユーザがMPIプログラムを正しく，かつ効率よく実行させることには多大の努力が必要であると考えられる．

次に，図2.1，図2.2のそれぞれのプログラムから図2.3のMPIプログラムを生成するためには，どのような処理が必要か述べる．

(1) データの依存関係の解析

図2.1：宣言されている全てのデータ**num_steps**，**step**，**i**，**x**，**pi**，**sum**の依存性を解析する必要がある．

num_steps → 依存関係なし．

step → 6行目と8行目．

i → 依存関係なし．

x → 8行目と9行目．

pi → 依存関係なし．

sum → 9行目と11行目．

これらにより，データ**step**，**x**，**sum**にステートメントレベルで依存関係があることが判る．また，7行目から始まるforループのイタレーション間のデータの依存性解析も行う必要がある．この場合，データ**sum**に対してイタレーション間の依存性があることがわかる．そうすると，この**sum**に対してどのような依存性があるのかという点についてさらに解析する必要がある．

図2.2：sharedデータ**num_steps**，**step**，**i**，**pi**，**sum**の依存性を解析する必要がある．

num_steps → 依存関係なし．

step → 8行目と12行目．

i → 依存関係なし．

pi → 依存関係なし.

sum → 13行目と15行目.

従って，sharedデータ**step**，**sum**にステートメント間で依存関係があることがわかる．こちらの場合は，図2.1に対しての解析では必要であった「forループのイタレーション間におけるデータの依存性解析」を行う必要はない．また，リダクション演算として**sum**に対する加算（+）が指定されているため，図2.1に対して行われるような「**sum**に対しての依存性の解析」を行う必要もない.

⑵ 並列化ブロックの検出

図2.1：⑴のデータの依存性解析の結果により，7行目から始まるforループを並列化ブロックとしようとすると，データ**sum**をforループ終了後にリダクション演算を行うことで並列化が可能であることを検出する必要がある.

図2.2：10行目のOpenMP指示文を抽出することで，11行目から始まるforループを並列化ブロックとして容易に特定することができる.

⑶ プロセススケジューリング

図2.1：制御変数の初期値，条件値，ステップ値を用いて，各プロセスにループのイタレーションを分散させる.

図2.2：図2.1と同様の方法で行う.

⑷ 通信命令の挿入

図2.1：⑵で求めた**sum**に対してのリダクション演算を行う命令を挿入する.

図2.2：10行目の**reduction**指示節により，**sum**に対してのリダクション演算命令を挿入すればよいことがわかる.

図2.1と図2.2に対する解析の処理量について比較を行うと，図2.2に対しての方が明らかに処理量は小さい．特に，図2.1に対する分析の際に必要となる⑴の「ループイタレーション間での依存性の解析」と，⑵の「データ**sum**に対してリダクション演算命令を挿入することで，7行目から始まるforループを並列化ブロックとして検出する」といった解析は，コンパイラなどにとって負担が大きい．また，場合によっては検出できない場合も考えられ．結果として出力コードの実行性能の低下の原因となることも考えられる．したがって，MPIプログラムの生成をより簡単に行うためには，逐次プログラムを入力とするよりも，OpenMPプログラムを入力とする方が適当であり，かつ，変換後のMPIプログラムの実行効率も良いものが得られると期待できる.

3 トランスレータ

本章では，本研究で開発を行ったトランスレータの並列化方針と実装方法の概要について述

べる．

3.1 トランスレータの並列化方針

本節では，OpenMPとMPIのプログラミングモデルの相違点を考慮して，目的コード生成における並列化の方針について述べる．

3.1.1 目的コードの並列実行モデルの決定

OpenMPは，fork-joinモデルであるため，Work-sharingを指示する以前，または，Work-sharing構文に組み合わせる形式で**parallel**指示文を記述して，並列リージョンの開始を指示する（並列実行するためのスレッドを生成する）必要がある．これに対してMPIプログラムでは，実行開始時に1度だけプロセスが生成されるモデルであり，動的なプロセス生成は許していない[注2)]．このことから，並列実行モデルとして，次の2通りの方法が考えられる．

1. MPIプログラムにおいても，OpenMPで指示されている並列リージョン以外（すなわち，逐次リージョン）はrootプロセスのみが実行する．
2. 逐次リージョンにおいても全プロセスが並行して同じ記述文を実行する．

このうち，1の手法のほうが，rootプロセス以外のプロセスが無駄な実行をせず，プログラムの実行としては簡略である．しかし，並列リージョンの開始地点で，逐次リージョンで生成されたsharedデータの値を全プロセスに通知する必要がある．一方，2の手法は，一見すると，rootプロセス以外のプロセスについては無駄な処理にみえるが，1の手法の場合に必要な並列リージョン開始直前の通信が必要ないことが判る．従って，本研究では，並列実行モデルとして2のモデルを用いることにする．ただし，入出力に関しては，rootプロセスのみで実行することとする．

3.1.2 並列化ブロックの検出

本トランスレータでは，原則としてOpenMPプログラム中にWork-sharing構文として明示的に指示されたブロックのみ並列化の対象とする．

OpenMPがWork-sharing構文として定義しているのは，次の3つの構文である．

1. **for**指示文
2. **sections**指示文
3. **single**指示文

このうち，3の**single**指示文は，対応する構造ブロックを１つのスレッドのみが実行することを指示する構文であるため，MPIプログラムにおいても並列化する必要はない．従って，並列化の対象は**for**指示文と**sections**指示文のみとなる．さらに，OpenMP API開発の背景，また，OpenMPプログラム中の並列化頻度の高いブロックという点を考慮すると**for**構文の使用頻度

が高く，重要であると考えることができる．従って，本研究では，1の**for**指示文を対象とした構造ブロックの並列化を中心としてトランスレータの開発を行うことにする．2の**sections**指示文は，各スレッドが実行するブロックを**section**指示節によって明示的に記述するため，それぞれの**section**指示節とその動的有効範囲内のブロックを並列化の対象とする．

3.1.3 並列化ブロックのスケジューリング

本節では，**for**指示文と**sections**指示文で指定した構造ブロックのプロセススケジューリングについて述べる．

まず，**for**指示文とそのブロックについて考える．OpenMPでは，**for**指示文の指示節として**schedule**節を指定することができ，ユーザは**schedule**節により，各スレッドへのスケジューリングを行う．**schedule**節として利用できるのは以下のいずれかである．

- **schedule** (**static ,** *chunk_size*)
- **schedule** (**dynamic ,** *chunk_size*)
- **schedule** (**guided,** *chunk_size*)
- **schedule** (**runtime**)

一方，MPIプログラムにおけるスケジューリング方法は，ブロック分割，サイクリック分割，ブロックサイクリック分割が一般的であり，本研究では，原則としてブロック分割で割り当てることにする．ただし，**schedule**節で分割方法を**static**，または，**dynamic**で指定し，さらに*chunk_size*が1のとき，つまり，**schedule** (**static,** 1)，**schedule** (**dynamic,** 1)の場合のみサイクリック分割で割り当てる．

もう一方の**sections**指示文の場合，それぞれの**section**指示節の動的有効範囲ブロックを，rootプロセスから順に割り当て，並列実行を行う．

3.2 トランスレータの概要

本トランスレータは，OpenMP指示文が挿入されたCプログラムを入力コードとし，目的コードとしてMPIの通信命令が挿入されたCプログラムを出力する．フェーズ構成は図3.1のようになる．字句解析は，字句解析生成系であるlexを利用し，構文解析は構文解析生成系でLR構文解析ルーチンを生成するyaccを利用した．

構文解析後，並列性に関する記述を抽出するための構造ブロック「**並列性記述ブロック**」（**Parallelism Description Block**，以下では**PDB**と略記する）に分割する．この**PDB**に，タスク間のデータの依存関係を示したエッジを付加して表現したグラフを「**データフローグラフ**」（**Data Flow Graph**，以下では，**DFG**と略記する）とし，これを中間表現とする．その**DFG**をもとに，データフロー方程式を解き，sharedデータの依存性解析を行う．この依存性解析の結果，**PDB**間でsharedデータの値の更新が必要な場合にはMPI通信命令の挿入を行う．その

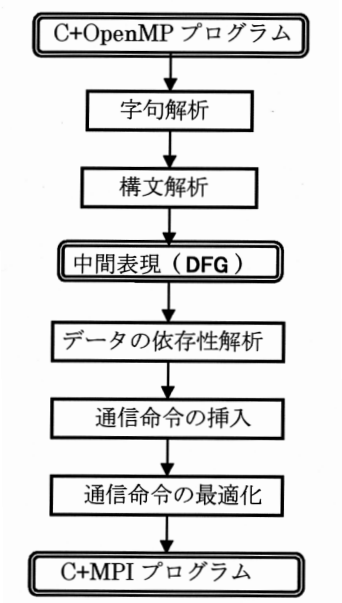

図3.1 トランスレータのフェーズ構成

際に，通信命令がさらに最適化できる場合は最適化を行い，最終的にMPIプログラムとして出力する．

次に，OpenMPプログラムからMPIプログラムへの変換を行う際に，必要となるsharedデータの依存性解析について述べる．

3.2.1 sharedデータの依存性解析

OpenMPプログラム中で宣言される各データの依存性解析は，基本的にsharedで確保されているデータについてのみ行えばよい．プログラム上で明示的にprivateとして確保されたデータに関してはMPIでも同様に扱うことができ，特別な変換処理を行う必要はない．

3.2.1.1 並列性記述ブロックとデータフローグラフ

構文解析後，入力プログラムを「**並列性記述ブロック**」**PDB**に分割する[3][4]．入力プログラムは，

⑴ OpenMP指示文が動的有効範囲外，すなわち，逐次実行する範囲．

⑵ OpenMP指示文の動的有効範囲．

の2種類に分類することができる．まず，⑴の範囲に対する**PDB**の定義は，

⒜ 連続する代入文ブロック．

⒝ forループやdo-while文などの繰り返し文ブロック．

⒞ if文などの分岐を伴うブロック．

(d) サブルーチンブロック．

とする．一方，(2)の範囲に対する**PDB**の定義は，以下の4つに分類される．

(e) 並列リージョン開始を示し，スレッドの生成を行う**omp parallel**指示文で，かつ，その指示文が並列Work-sharing組合せ構文で記述されていないブロック．

(f) **omp for**や**omp master**などの指示文とそのOpenMP指示文の動的有効範囲内に記述されているブロック．または，**omp parallel**指示文で，それが並列Work-sharing組合せ構文である場合の指示文とその動的有効範囲内に記述されているブロック．

(g) (e)に該当する**omp parallel**指示文の動的有効範囲内で，かつ(f)に適合しない代入文，分岐命令，または繰り返し文ブロック．ただし，それがforループブロックで，さらにそのブロック内に(f)で定義された**PDB**を含んでいた場合は，forループの入口から(f)で定義された**PDB**までに記述されているブロック．

(h) (e)に該当する**PDB**の動的有効範囲の出口を示すブロック．

このうち，(e)のブロックは**PDB**中では，**parallel** (*parallel_instruction_number*)として表現する．また，データスコープ属性指示節が記述されている場合は，その情報も付加する．また，(g)によって定義されているブロックに現れるループ本体の出口を示すトークンを**PDB**では，**for** (*for_construct_number*)として表現する．そして，(h)によって定義された**PDB**は**parallel end**として表現する．

この(a)〜(h)の定義をもとに入力プログラムを**PDB**列に分割し，また，各**PDB**内のステー

```
 1  lap_main()
 2  {
 3     int i, x, y;
 4     double sum;

 5  #pragma omp parallel private ( i, x, y )
 6  {
 7     for(i = 0; i < N; i++){
 8  #pragma omp for
 9       for(x = 1; x <= XSIZE; x++){
10         for(y = 1; y <= YSIZE; y++){
11           uu[x][y] = u[x][y];
12         }
13       }

14  #pragma omp for
15       for(x = 1; x <= XSIZE; x++){
16         for(y = 1; y <= YSIZE; y++){
17           u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1]
                      + uu[x][y+1]) / 4.0;
18         }
19       }
20     }
21  }

22     sum = 0.0;
23  #pragma omp parallel for reduction(+:sum) private( y )
24     for(x = 1; x <= XSIZE; x++){
25       for(y = 1; y <= YSIZE; y++){
26        sum += (uu[x][y]-u[x][y]);
27       }
28     }
29  }
```

図3.2 laplaceのOpenMPによる並列化部分

トメントに対して，記述順にp1から始まるステートメント番号を付加して表す．

次に，中間表現となる「**データフローグラフ**」**DFG**を定義する．**DFG**は，**PDB**間に制御フローをエッジとして付加したグラフである．エッジの張り方は，上記の**PDB**の定義方法により，それぞれの**PDB**からその直後の**PDB**に対して張るだけでよいこととなる．例えば，上記(g)に該当するforループのような繰返し構文の場合は，ループ本体の出口から構文の出口**for**(*for_construct_number*)に対してエッジを張ることで，ループ入口に対しての逆向きのエッジは張る必要はない．

ここで，2次元ラプラス方程式の境界値問題をヤコビ法により求めるプログラム（以後，*laplace*とする）を例として，**DFG**について述べる．この*laplace*においてOpenMPによる並列化が可能な部分のみを抜粋したものを図3.2に示す．ここでは，挿入したOpenMP指示文を青色で示している．この図3.2のプログラムを解析して**PDB**に分割し，それにエッジを張って**DFG**表現としたものが図3.3である．

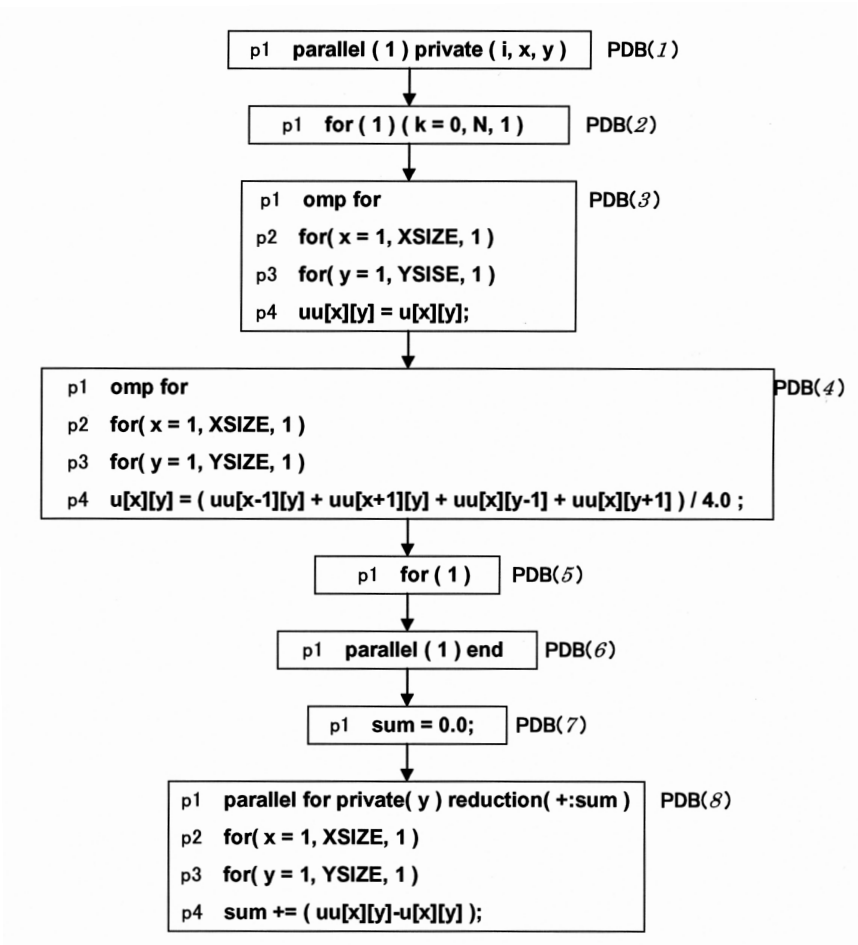

図3.3　*laplace*を入力した場合の中間表現（**DFG**）

まず，プログラムの構成要素を上記の定義より，**PDB**に分割する．定義(e)より，5行目の**omp parallel**指示文が，**parallel (1)**という表現で**PDB**として定義される．この**PDB**には，指示節である**private (i, x, y)**という情報も付加される．また，定義(f)より，8行目から13行目，14行目から19行目，また，23行目から27行目までの**omp for**指示文，または**omp parallel for**指示文とその動的有効範囲内のブロックを１つの**PDB**として定義する．7行目のforループの入口を示す記述は定義(g)より，**for (1)**として定義される．19行目と20行目の“**}**”は，定義(g)，定義(h)より，それぞれ，**for (1)**と**parallel (1) end**として定義される．22行目は，定義(a)により，１つの**PDB**として定義される．これらの**PDB**に制御フローをエッジとして付加して表現したグラフが図3.3である．

同図を見れば判るように，上記の**PDB**の定義方法により，自身より前で定義されている**PDB**に戻るような逆向きのエッジは必要なく，自身の直後の**PDB**に対してのみにエッジを張るだけでよい．以上の手法により，トランスレータは**PDB**間の制御の流れが把握できる状態となる．

次節では，図3.3のような**DFG**で表現されたグラフをもとに，**PDB**間におけるsharedデータの依存性解析をデータフロー方程式によって求める方法について述べる．

3.2.1.2 データフロー方程式

ここでは，sharedデータの依存性解析の際に用いるデータフロー方程式について述べる．まず，この方程式で利用される集合について定義する．以下では，**PDB**毎に並列性記述ブロック番号*PDB_number*を割り当て，**PDB** (*PDB_number*)で表現する．また，ステートメントpでsharedデータ*x*の値が生成，または，更新されているときにp (gen(*x*))とし，また，sharedデータ*x*の値が参照されているときはp (ref (*x*))と記述する [5] [6]．

GEN (**PDB** (*PDB_number*)) = { (*x*, *n*) | *n*は*PDB_number*,
*x*は，p (gen(*x*)) ∈ **PDB** (*PDB_number*)なるpが存在し，また，p以降，**PDB** (*PDB_number*)の出口までにp’(gen(*x*)) ∈ **PDB** (*PDB_number*)なるp’が存在しないときのデータ*x* }

REF (**PDB** (*PDB_number*)) = { (*x*, *n*) | *n*はPDB_number,
*x*は，p(ref(*x*)) ∈ **PDB** (*PDB_number*)で，**PDB** (*PDB_number*)の入口からpまでの間にp’ (gen(*x*)) ∈ PDB (*PDB_number*)なるp’が存在しないときのデータ*x* }

これらの定義により，ある**PDB** (*PDB_number*)に到達するデータとそのデータが生成された**PDB**番号の集合**REACH** (**PDB** (*PDB_number*))は，以下のデータフロー方程式を解くことで求めることができる．ただし，**GEN** (**PDB** (*PDB_number*))，**REF** (**PDB** (*PDB_number*))において，集合要素[*x*]を示す場合は“ →*data* ”，集合要素[*n*]を示す場合は，“ →*num* ”で表すことにする．また，ある**PDB** (*PDB_number*)の１つ前の**PDB**を**PDB** (*p_PDB_number*)と表現する．

$$\begin{aligned}&\textbf{REACH}\ (\textbf{PDB}\ (\mathit{PDB\_number}\ ))\ \cup\\ &\qquad = \textbf{GEN}\ (\textbf{PDB}\ (\mathit{p\_PDB\_number}))\\ &\qquad\quad \{\ (x, n\ )|\ \ (x, n\ ) \in \textbf{REACH}\ (\textbf{PDB}\ (\mathit{p\_PDB\_number}))\\ &\qquad\qquad \text{かつ，}\ x \notin \{\textbf{REF}\ (\textbf{PDB}\ (\mathit{p\_PDB\_number}))\rightarrow \mathit{data}\}\}\end{aligned}$$

このようにして求められた**REACH** (**PDB** (*PDB_number*))をもとに，sharedデータの依存性解析を行う．

ある**PDB**（ここでは，説明の都合上*PDB_number* = aとして，**PDB**(a)で表現する）について，**REACH** (**PDB** (a))→*data*と**REF** (**PDB** (a))→*data*が一致した場合に，**REACH** (**PDB** (a))の要素*PDB_number*（ここでは*PDB_number* = bとする）が示している並列性記述ブロック**PDB** (b)と**PDB** (a)の間でsharedデータに関して依存関係があることが判る．このsharedデータの依存性を示し，集合要素[*x*，*n*]からなる集合を**SH_DATA_DEPEND** (**PDB** (*PDB_number*))とすると，

$$\begin{aligned}&\textbf{SH\_DATA\_DEPEND}\ (\textbf{PDB}\ (\mathit{PDB\_number}\ ))\rightarrow \mathit{data}\\ &\qquad = (\textbf{REACH}\ (\textbf{PDB}\ (\mathit{PDB\_number}))\rightarrow \mathit{data}\ )\\ &\qquad\quad \cap\ (\textbf{REF}\ (\textbf{PDB}\ (\mathit{PDB\_number}))\rightarrow \mathit{data}\ )\end{aligned}$$

が空集合でない場合，

$$\begin{aligned}&\textbf{SH\_DATA\_DEPEND}\ (\textbf{PDB}\ (\mathit{PDB\_number}))\rightarrow \mathit{num}\\ &\qquad = \textbf{REACH}\ (\textbf{PDB}\ (\mathit{PDB\_number}))\rightarrow \mathit{num}\end{aligned}$$

で表すことができる．

図3.2の*laplace*を入力とした場合のsharedデータの依存性解析表を表3.1に示す．この表より，**PDB** (*4*)で，**SH_DATA_DEPEND** (**PDB** (*4*)) = {(*uu*, *3*)}となり，**PDB** (*3*)との間でデータ*uu*に関して依存性があることが判る．同様に，**PDB** (*5*)の**SH_DATA_DEPEND** (**PDB** (*5*))={(*u*, *4*)}より，**PDB** (*4*)と**PDB** (*5*)でデータ*u*に関して依存性があることが判る．なお，**PDB** (*5*)で求める集合の対象は，**for_construct_number**が同じであるforループ入口が示されてい

る**PDB** (*2*)から，**PDB** (*5*－*2*)である**PDB** (*3*)までの解析を行うことで，sharedデータの依存関係が把握できる．つまり，**PDB** (*n*)で**for (for_construct_number) end**が示されている場合，**for_construct_number**と同じ値を保持するforループ入口を示している**PDB**から，**PDB** (*n*－*2*)までの**GEN**および**REF**を，**PDB** (*n*)でのそれぞれに置き換えることができ，これより，**SH_DATA_DEPEND** (**PDB** (*n*))を求めればよい．**PDB** (*7*)は逐次リージョンであるため，依存性解析は行われないため，**GEN**，**REF**，**SH_DATA_DEPEND**は空集合φとなる．そして，**PDB** (*8*)では，**REACH** (**PDB** (*8*))と**REF** (**PDB** (*8*))により{(*u*, *4*)，(*uu*, *5*)}に対して依存性があることが判り，**SH_DATA_DEPEND** (**PDB** (*8*)) = {(*u*, *4*)，(*uu*, *5*)}となる．

表3.1 sharedデータの依存性解析表

PDB(*PDB_number*)	GEN	REF	REACH	SH_DATA_DEPEND
PDB (*1*)	φ	φ	φ	φ
PDB (*2*)	φ	φ	φ	φ
PDB (*3*)	(*uu, 3*)	(*u, 3*)	φ	φ
PDB (*4*)	(*u, 4*)	(*uu, 4*)	(*uu, 3*)	(*uu*, 3)
PDB (*5*)	(*uu, 5*)	(*u, 5*)	(*u, 4*)	(*u, 4*)
PDB (*6*)	φ	φ	(*u, 4*) (*uu, 5*)	φ
PDB (*7*)	φ	φ	(*u, 4*) (*uu, 5*)	φ
PDB (*8*)	(*sum , 8*)	(*sum, 8*) (*uu, 8*) (*u, 8*)	(*u, 4*) (*uu, 5*)	(*u, 4*) (*uu, 5*)

以上でsharedデータの依存性解析は完了する．次に，sharedデータに関して依存性があった場合，MPIプログラムでのデータの更新方法について述べる．

3.2.2 通信命令の挿入と最適化

前節のsharedデータの依存性解析により，sharedデータがどの**PDB**どうしの間で依存性があるかが解析される．前述のとおり，目的コードとなるMPIプログラムでは，プロセス間で共有されるsharedデータは存在せず，全てのデータがプロセス固有のprivate変数として定義される．従って，MPIプログラムでは，sharedデータの更新が必要となった場合，プロセス間通信により，通知する必要がある．本節では，この通信命令の挿入方法とその最適化について述べる．

3.2.2.1 通信命令の挿入

OpenMPプログラムでsharedとして扱われるデータは，配列（またはポインタ）とスカラ変

数の2種類に大別することができ，さらに，大規模な処理を行う数値計算のような分野では，これらのデータの値が**omp for**指示文による並列実行中にどのような形式で更新されるかは推測が可能な場合が多い．配列の場合は，イタレーションごとに異なった要素に対しての代入，またはリダクション操作が最も頻繁に利用されている．また，スカラ変数の場合は，そのデータに対してのリダクション操作が最も頻繁に利用されている．従って，これらの場合の変換規則を定義すれば，本トランスレータは比較的広範囲の入力に対して適用できると考えられる．

まず，配列で宣言されたデータの変換規則を定義する．例えば，図3.2の**PDB** (*3*)と**PDB** (*4*)間で依存関係がある**uu[XSIZE+2][YSIZE+2]**を例として述べることにする．**PDB** (*3*)のp2のforループが，p1の指示文により各スレッドでWork-sharingされる．スレッドへのスケジューリングは，**schedule**節による指示がないため，ブロック分割として割り当てられている．そして，p4で*uu*のそれぞれの配列要素の値が生成されている．このとき各スレッドが生成する配列要素を，次の**PDB** (*4*)を実行する前に他のプロセスにブロードキャストして通知する必要がある．本トランスレータでは，**MPI_Bcast**関数を挿入することで，他の全プロセスに対して自プロセスが生成したデータの更新を通知する．

次に，sharedデータがスカラ変数であり，それに対してリダクション操作が行われている場合について考える．ここで，新たに，**omp for**指示文の**reduction**節で指定されたデータとその**PDB**の番号を要素とする集合を示す**OMP_REDUCT_DECL** (**PDB** (*PDB_number*))，また，ある同一ステートメントで値の参照と生成が行われるデータとその**PDB**番号を要素とする集合を示す**REDUCT_OP_DATA** (**PDB** (*PDB_number*))という2つのリダクション操作が行われる場合のデータ集合を定義する．

OMP_REDUCT_DECL (**PDB** (*PDB_number*))
= { (*x, n*) | *n*は*PDB_number*，*x*は**PDB** (*PDB_number*)のp1において**reduction**節で指定されているsharedデータ}

REDUCT_OP_DATA (**PDB** (*PDB_number*))
= { (*x, n*) | *n*は*PDB_number*,
*x*はp (gen (*x*)) ∈ **PDB** (*PDB_number*),
かつ，p (ref (*x*)) ∈ **PDB** (*PDB_number*) なるpが存在するときのsharedデータ}

これらのデータ集合をデータフロー方程式を用いて，「これら２つのデータ集合の一方，または両方で宣言されたデータから，その前の**PDB**ですでに宣言されているデータを除いたもの」を示す**MPI_REDUCT_DATA** (**PDB** (*PDB_number*))を求める．そして，最終的に「前の**PDB**の**MPI_REDUCT_DATA**で宣言され，かつ，当**PDB**で参照されるデータから，前の

PDBの**MPI_RDCT_COM_RQ**で宣言されているデータを除き，残ったデータに対してMPIプログラムでリダクション操作が必要である」ことを示す**MPI_RDCT_COM_RQ** (**PDB** (*PDB_number*))を求める．

MPI_REDUCT_DATA (**PDB** (*PDB_number*))
= (**OMP_REDUCT_DECL** (**PDB** (*PDB_number*))
∪ **REDUCT_OP_DATA** (**PDB** (*PDB_number*)))
− **MPI_REDUCT_DATA** (**PDB**(*p_PDB_number*))

MPI_RDCT_COM_RQ (**PDB**(*PDB_number*))
= (**MPI_REDUCT_DATA** (**PDB** (*p_PDB_number*))
∩ **REF** (**PDB** (*PDB_number*)))
− **MPI_RDCT_COM_RQ** (**PDB** (*p_PDB_number*))

この**MPI_RDCT_COM_RQ** (**PDB** (*PDB_number*))を求め，該当したデータに対してリダクション演算命令を挿入することで，入力コードと等価の出力コードの生成が可能となる．本トランスレータでは，出力コードに**MPI_Reduce**，または**MPI_Allreduce**関数を挿入することで実現する．

3.2.2.2 通信命令の最適化

前節で基本的な通信命令の挿入方法について述べた．しかし，より並列効果が得られるプログラムの生成には，実行には必要ない命令を削り，通信オーバヘッドの削減が必要となる．本節では，通信命令の最適化が可能である場合の最適化手法について述べる．

⑴ 配列データの場合

前節で述べた通信命令**MPI_Bcast**関数の挿入は，sharedデータが配列の場合において，各プロセスが自分で生成した配列要素を，全てのプロセスに送信するという方式であった．この方法では，「全プロセス」に対して送信する方式であるため，そのデータを必要としないプロセスに対しても通信が行われる．このように，実行に必要ない通信も行われ，それが通信オーバヘッドとなることが考えられる．従って，本節では以下の2つの条件，

1. sharedデータが配列である．
2. 通信命令の挿入対象となる2つの**PDB**のプロセススケジューリング方法が同じである．

に該当する場合の最適化について述べる．

最適化手法として，図3.3の**PDB** (*3*)と**PDB** (*4*)間で依存関係がある**uu** [**XSIZE + 2**] [**YSIZE + 2**]を例として述べることにし，また，ここでは**XSIZE** = **YSIZE** = 8，プロセス数は4

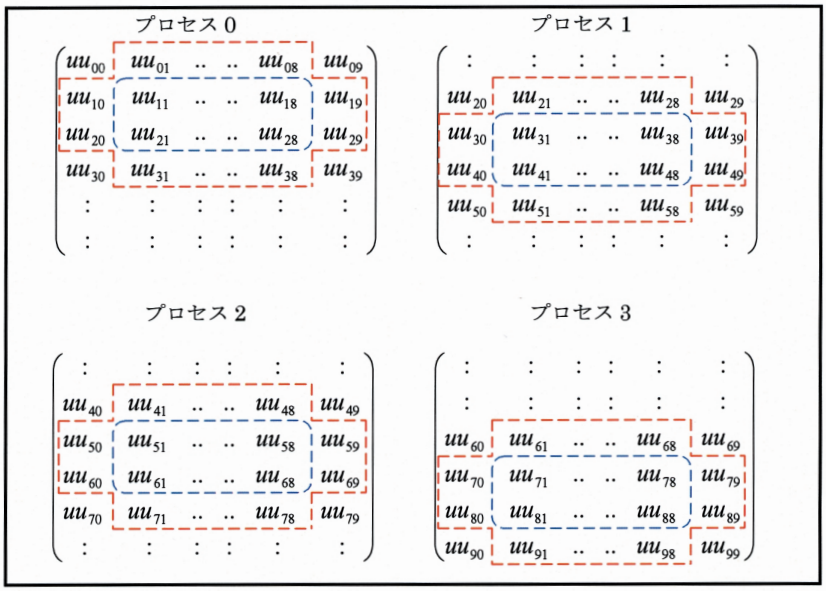

図3.4 各プロセスの**PDB** (*3*)の*uu*の更新要素と**PDB** (*4*)の参照要素

とする．図3.4では，**PDB** (*3*)で各プロセスが更新する*uu*の要素を青色で示し，**PDB** (*4*)で各プロセスが参照する要素を赤色で示している．

前節の**MPI_Bcast**関数を用いた更新処理では，例えば，rootプロセスでは，**uu** [**1,1**]から**uu** [**2,9**] までの計18要素を全プロセスに送信していた．しかし，図3.4をみると，プロセス0が値を更新した要素を**PDB** (*4*)で参照するプロセスはプロセス1のみ，プロセス1，2の更新要素はそれぞれの前後のプロセス，さらに，プロセス3はプロセス2のみに参照されるだけである．従って，MPIプログラムにおいての**MPI_Bcast**による通信には，無駄な通信も含まれており，通信命令の最適化を行う必要がある．この場合における最適化として，**PDB** (*4*)で各プロセスに参照される配列要素を計算することで，**PDB** (*3*)のどのプロセスで更新された要素を必要とするのかを解析し，その結果より，必要な要素の通信を行うといった方法が考えられる．本研究では，**MPI_Isend**，**MPI_Irecv**関数のノンブロッキングモードでの通信を行うことで，無駄な通信を削減する方法を用いる．詳しくは，3.3.5.2で述べる．

⑵ **firstprivate**，**lastprivate**で指定されたデータのブロードキャスト

firstprivate，**lastprivate**節で指定されたデータは，それぞれ対象ループの前後でrootプロセス，または最終プロセスの値をブロードキャストする必要がある．この方法では，指定されたデータの数の回数分ブロードキャストしなければならず，性能低下の原因となることが考えられる．そこで本研究では，データをブロードキャストする前に，**MPI_Pack**関数を用いて複数のデータをpackし，その後に，1回のブロードキャストによる通信を行う方法を用いることにした．

4 性能評価

本章では，本研究で開発したトランスレータの性能評価を行う．評価方法として，4.1節ではラプラス方程式の陽解法プログラム(*laplace*)を入力とした場合に出力するMPIプログラムの実行により，並列処理効果の検証を行う．4.2節では，NASA Ames Research Centerで開発されたNAS Parallel Benchmarks[7]（以下，NPBと略記する）の実行結果を用いて，RWCPで開発されたOmni/OpenMPコンパイラとソフトウェア分散共有メモリシステムSCASHを用いて実行した結果[2][8]との比較を行う．また，4.2節では，これらの比較だけでなく，NPBとして公式に配布されているMPIプログラムを用いた実行結果とも比較を行うことにする．

本トランスレータを用いた実行方式と，RWCPが開発したOmni/SCASHを利用した実行方式の並列実行環境としては，表4.1に示す構成のノード8台で構築された小規模なクラスタシステムを用いた．また，MPIプログラムの実行時に必要となるMPIライブラリは，MPICH-1.2.5.2を利用した．

表4.1 各ノードの構成

ノード名	calp41~calp48
Processor	Intel Pentium4 / 2.0GHz
Memory	768MB(RIMM)
NIC	100Mbps Ethernet on Board (Intel PRO/100 VM Network Connection)
OS	Linux 2.4.20

4.1 ラプラス方程式の陽解法プログラムによる評価

3章で例として示した2次元ラプラス方程式の境界値問題をヤコビ法を用いて求解するプログラム(*laplace*)を実際に本トランスレータでMPIプログラムに変換し，そのプログラムを実行した結果を示す．並列化を行う部分は，図3.2で示した箇所の計算のみとした．また，2次元格子上の隣接する4点の平均を繰り返し求める際の反復回数は21回と固定し，対象となる行列サイズを変化させて評価を行った．変換に伴う主な変更箇所は，図3.3の**PDB** (*3*)，**PDB** (*4*)間のデータ*uu*に対しての更新処理，また，**PDB** (*8*)の終了後の*sum*に対するリダクション処理の2箇所である．**PDB** (*5*)の*uu*，**PDB** (*8*)の*u*，*uu*に関しては，スケジューリング方法が同じであり，かつ，前方の**PDB**で他のプロセスが生成したデータをその**PDB**で参照しないため，更新処理は必要としない．従って，プロセス数をp (ただし，pは2^k)とすると，通信回数Cは，

$$C = \{(p-1)\times 2\}\times 21 + (2p-2)$$

となる．$(p-1)\times 2$は**PDB** (*3*)，**PDB** (*4*)間の**MPI_Isend**，**MPI_Irecv**による通信回数である．21は反復回数，残る部分は**PDB** (*8*)終了後の*sum*に対しての**MPI_Allreduce**による通信回数

を表している．例えば，p=8の場合，C = 308回の通信回数となり，それぞれの通信におけるデータサイズは**PDB** (*3*)，**PDB** (*4*)間の更新処理ではuuの列数となる．例えば，1,000×1,000の行列の場合では（double型）×1,000となり，8,000byteのデータが1回の通信で送受信される．また，**PDB** (*8*)後のリダクション処理ではdouble型1個，すなわち，8byteのデータが通信される．

次に，実際に変換後のMPIプログラムを実行して得られた速度向上率を図4.1に示す．この図では，入力プログラムとなるOpenMPプログラムの並列効果がどれほど得られるのかを理解するために，入力プログラムをSMP（Sun Fire V880：8プロセッサから成るSMPシステム，表4.2参照）で実行した結果を示している．ここでは，両者の実行環境が異なるため，比較は速度向上率のみで行うこととする．実行プロセス数は1，2，4，8，行列サイズは2,002×2,002，3,002×3,002，4,002×4,002，5,002×5,002の場合を示す．

表4.2　Sun Fire V880の構成

Processor	UltraSPARCⅢ /800MHz×8
Cache	1次−64KBデータ，32KBインストラクション 2次−8MB
Memory	16GByte SDRAM
OS	Solaris 9
Cコンパイラ (オプション)	Sun Forte7 コンパイラ（ −xopenmp ）

結果は，図4.1に示すように，行列の規模が大きくなるにつれて理想値に迫っていることが判る．これは，行列サイズの増加に伴い，演算命令回数が増加する一方で，**MPI_Isend**，**MPI_Irecv**，**MPI_Allreduce**関数を用いた通信の回数は一定であり，通信サイズも性能に影響を及ぼさない程度の増加に留まっているためである．おのおのについて注目すると，8プロセスで5,002×5,002の行列を実行した場合，実行時の通信回数は，前述のとおり308回，また，1回の通信で40,000byte（**PDB** (*3*)，**PDB** (*4*)間），もしくは8byte（**PDB** (*8*)終了後）のサイズのデータが転送されている．これは，実行環境のネットワーク100Base-TXの有効バンド幅より小さい量であるため，データサイズが原因となる通信レイテンシはそれほど大きくないと考えられる．その結果，速度向上率は6.75となり，ある程度の並列効果がみられる結果となった．また，入力としたOpenMPプログラムのSMP上での実行結果と比較しても，SMPでは2,002×2,002の行列による実行が最も良い値となり，その後，行列サイズが大きくなるにつれて並列化効率が低下する結果となったのに対して，本トランスレータを用いた並列実行は，行列サイズの増加に伴って並列化効果が向上する結果となった．以上の結果より，**MPI_Isend**，**MPI_Irecv**，また**MPI_Allreduce**関数による通信回数がおよそ300回，また通信サイズが40,000byte程度の場合，そして，演算命令数が十分にある場合，本トランスレータを用いての

並列実行は有効であると考えられる．

4.2 NAS Parallel Benchmarksによる性能評価

本節では，NASA Ames Research Centerで開発されたNAS Parallel Benchmarks (NPB)をRWCPがFortranからCへの変換，さらにそのCプログラムに対してOpenMP指示文を用いて並列化を行った「OpenMP C versions of NPB2.3」[7]を用いる．これにより，RWCPが開発したOmni/OpenMPコンパイラとソフトウェア分散共有メモリシステムSCASHを用いた並列実行結果[8]との比較を行う．また，公式に配布されているNPB（以下，NPB公式版と記述する）の実行結果との比較も行い，本トランスレータの評価を行う．今回は，*EP*，*CG*，*IS*の3種類のベンチマークプログラムを用いることにした．ただし，NPB公式版による実行は，*IS*以外はFortran + MPIで記述されており，さらに，実行環境が本学のハイテクリサーチ・ラボに導入されているクラスタシステム（以下，HRクラスタとする）であるため，本トランスレータとの性能の比較は，速度向上率による性能評価のみとする．

HRクラスタは，100ノード，200CPUからなる中規模のPCクラスタである[注3)]．各ノードの構成を，表4.3に示す．また，ネットワークは，VIAとTCP/IPの2種類の接続から選択することができる．本研究では，できる限り平等な条件のもとで比較を行うため，TCP/IP接続の実行結果により性能の評価を行うことにする．また，MPI通信ライブラリはMPI/PRO-1.5を用いて実行した．

表4.3 HRクラスタの構成

ノード名	hr001~hr060（60台）	hr061~hr100（40台）
Processor	Intel Pentium Ⅲ / 750MHz	Intel Pentium Ⅲ / 866MHz
Memory	512MB	756MB
HDD	9.2GB × 2	9.2GB × 1
OS	RedHat Linux 6.2	RedHat Linux 6.2
Cコンパイラ	gcc version2.4	gcc version2.4

4.3.1 *EP* (the Embarrasingly Parallel) による性能評価

*EP*は，モンテカルロ法などで用いられる乱数生成プログラムで，乱数の種を求める関数と乱数列を求める関数を呼び出した後，得られた乱数列から乱数のペアを求め，それがどの領域に属するかを集計するものである．

ここでは，以下の3方式の実行により比較を行う．

- 本トランスレータによって出力されるMPIプログラムを8ノードのPCクラスタで実行したもの．
- RWCPが開発した*EP*のOpenMPプログラムをOmni/SCASHクラスタシステムで実行した

もの．

- NPB公式版をHRクラスタで実行したもの．

実行はクラスA（乱数生成数：536,870,912）で行い，その結果を図4.2に示す．Omni/ SCASHクラスタシステムを用いた実行結果を**Omni/SCASH**，NPB公式版の実行結果を**NPB-official**，そして，本トランスレータを用いた実行結果を**omp2mpi**で表す．

まず，Omni/SCASHとの比較を行う．**Omni/SCASH**が8プロセスで6.96倍であったのに対して，**omp2mpi**は，7.91倍となり，1.19倍程度**omp2mpi**の方が良い結果を得ることができた．Omni/SCASHは，共有メモリ領域にあるデータの一貫性制御機構として，共有メモリ領域内の各ページを管理するための「ホームノードを配置」しており，あるノードがアクセスするページのホームノードが他ノードであった場合，「ページフォルト」が起こる．このページフォルトにより，対象ページはホームノードからページコピーされ，このときデータの転送が必要となる．**omp2mpi**の方が良い値を得ることができたのは，Omni/SCASHのこのページフォルトに伴うオーバヘッドによる性能低下が原因として考えられる．

次に，**omp2mpi**の並列化効果について注目する．2，4，8プロセスの速度向上率はそれぞれ，1.97，3.95，7.9となり，ほぼ理想といえる結果を得ることができた．また，NPB-officialとの比較を行っても，**NPB-official**が8プロセスで7.71倍となり，本トランスレータの方が若干ながら良い性能値を得ることができた．**omp2mpi**，**NPB-official**は，ともに通信回数，データサイズが同じであるにも関わらず，**omp2mpi**が**NPB-official**を上回る結果となった．この理由として考えられる点を以下に示す．

- 試行回数の不足による誤差．
- MPI通信ライブラリであるMPICH，MPI/PROの実行効率による差．
- プログラミング言語としてのCとFortranのソフトウェアオーバヘッドの差．

並列実行プログラムのアルゴリズムは，両者とも同じとなり，良い性能値を得ることができた．従って，本トランスレータは，sharedデータの更新処理やリダクション処理の割合が小さい場合，効率の良いMPIプログラムを生成することが可能であり，有効な手段であると考えられる．また，Omni/SCASHでは，プログラム実行中のデータアクセスパターンによって，並列処理効果が著しく低下したのに対して，本トランスレータによる実行は，*EP*のように局所性が高いプログラムの場合，データへのアクセスパターンが問題とはならないことが判った．

4.3.2 *CG* (Conjugate Gradient) による性能評価

*CG*は，疎行列の最小固有値を共役勾配法によって近似的に解くプログラムである．出力プログラムは，OpenMPプログラムの**for**指示文により13箇所のforループを並列化した．評価に用いたクラスAの場合，MPIプログラムでは，sharedデータに対するリダクション処理，または，更新処理のために，**MPI_Allreduce**関数，**MPI_Bcast**関数が必要となり，それぞれ812

回，390回呼び出される．このように，プログラム実行中は絶えず通信が行われるため，通信性能が結果に反映されると考えられる．また，**MPI_Allreduce**関数により転送される通信サイズはdouble型の8byte，**MPI_Bcast**関数の場合は（自プロセスが担当したループ回数）×（double型）であり，クラスAでは｛(14,000/プロセス数)×8｝byteとなる．従って，プロセス数が増加するにつれ，通信サイズは減少することが判る．

次に，本トランスレータの出力プログラム，Omni/SCASH版を実行して得られた結果，そして，NPB-officialによる実行結果を図4.3に示す．**Omni/SCASH**，**omp2mpi**は，並列効果がみられない結果となり，4プロセス以降は速度向上率が1以下となった．しかし，良い性能値が得られていない両者間に差がみられる．2プロセスの速度向上率は，**omp2mpi**が1.03倍と逐次処理の場合における性能値に対してわずかながら上回っている一方で，**Omni/SCASH**は0.37倍と逐次処理の半分にも達していない．そして，8プロセスの実行結果に注目すると，**omp2mpi**は0.716倍，**Omni/SCASH**は0.163倍と，**omp2mpi**の方がおよそ4.39倍，良い値を得ることができた．

NPB-officialの結果との比較を行うと，**omp2mpi**の速度向上率が1.03，0.83，0.71であるのに対して，**NPB-official**は，1.43，2.11，1.75という値となり，この結果より，更なる最適化が必要であることが判った．本トランスレータでは，入力となるOpenMPプログラムで並列化を指示している部分を，必ず，MPIプログラムにおいても並列化部分として出力している．従って，並列ブロック中で行う演算回数に対して通信の割合が極端に高い場合においても並列化を行っており，これが通信オーバヘッドによる性能低下につながり，良い値を得ることができなかったと考えられる．

4.3.3 *IS* (Integer Sort) による性能評価

*IS*は，大規模な整数ソートのベンチマークプログラムである．クラスAでは，8,388,608個の整数に対してのソートが15回反復される．変換後のMPIプログラムでは，プログラム開始地点で行われるrootプロセスで生成された乱数のブロードキャスト通信，そして，各プロセスがソートを行った後に自分の担当したデータをrootプロセスに集める通信が行われる．これらの通信はそれぞれ**MPI_Bcast**関数，**MPI_Isend**関数，そして**MPI_Reduce**関数を用いて実現している．**MPI_Bcast**関数によるデータ個数と**MPI_Reduce**関数によるリダクション処理の対象となるデータ個数は，ともに8,388,608個であり，**MPI_Isend**関数では，各プロセスが（8,388,608/プロセス数）×（int型のサイズ）byteのデータをrootプロセスに送信する．プログラム全体の通信回数は17回となる．

次に，この出力プログラムを実行した結果とOmni/SCASHで実行した結果を図4.4に示す．結果をみると，8プロセス実行における速度向上率は**omp2mpi**が0.727倍，**Omni/SCASH**では0.85倍となっており，両者とも並列処理効果がみられない結果となった．これは，実行中に

発生する**MPI_Bcast**，**MPI_Reduce**関数による集団通信の回数がプロセス数の増加に伴って多くなり，通信サイズが大きいことも実行に影響している．データサイズが大きいことにより，**MPI_Reduce**関数によるリダクション演算回数も8,388,608回となり，これが通信オーバヘッドとなっていることが考えられる．

次に，NPB-officialによる実行結果との比較を行う．**omp2mpi**では，8プロセスの実行で，速度向上率が0.72であったのに対して，**NPB-official**では，0.90となり，NPB-officialが3つの実行方法のなかで最も良い値を得る結果となった．これは，上記の**MPI_Reduce**関数による通信を防ぐために独自の関数を用意して，通信回数の削減を行うことで速度の低下を回避しているためである．さらに，手続き間解析を行うことで，通信命令の削減も行っている点も挙げることができる．この2項目に関しては，本トランスレータでは現段階でまだ解析／実装できていない．今後，このような最適化機能を装備していくことで，両者の実行結果にみられる「差」をより縮めることができると考えられる．

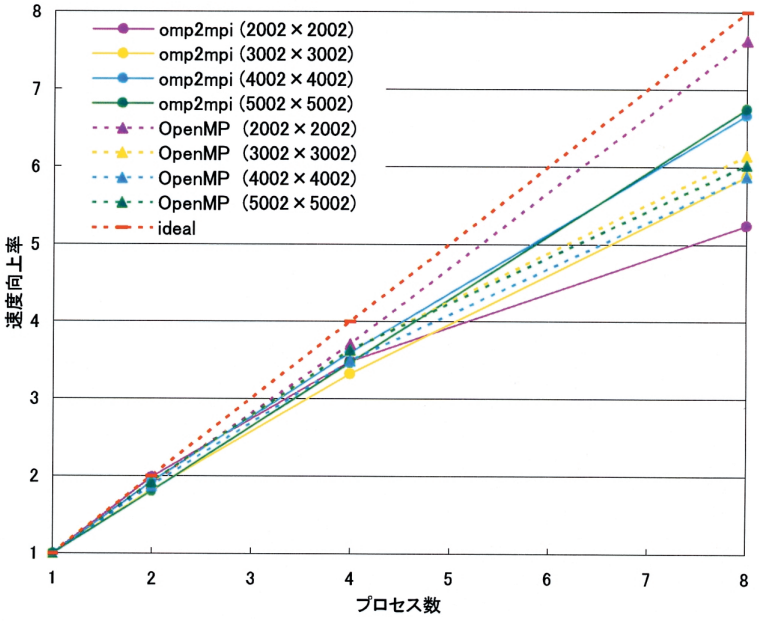

図4.1 *laplace*の速度向上率

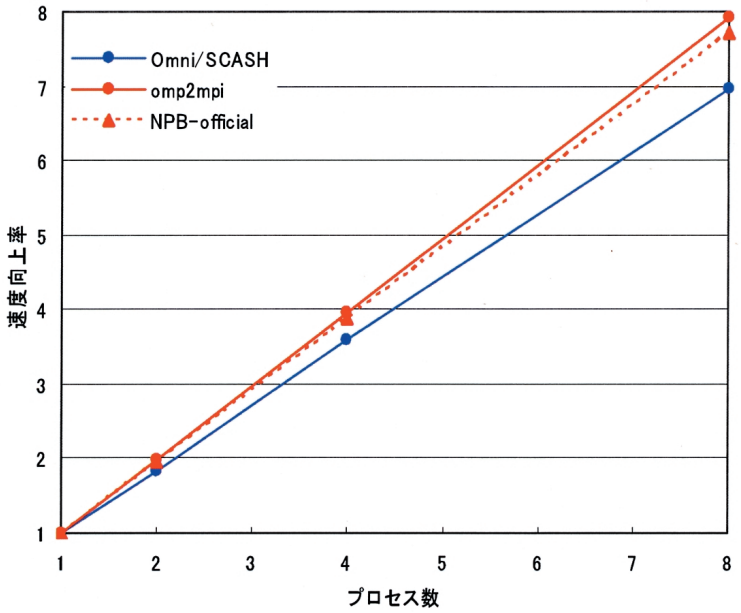

図4.2 *EP*の速度向上率

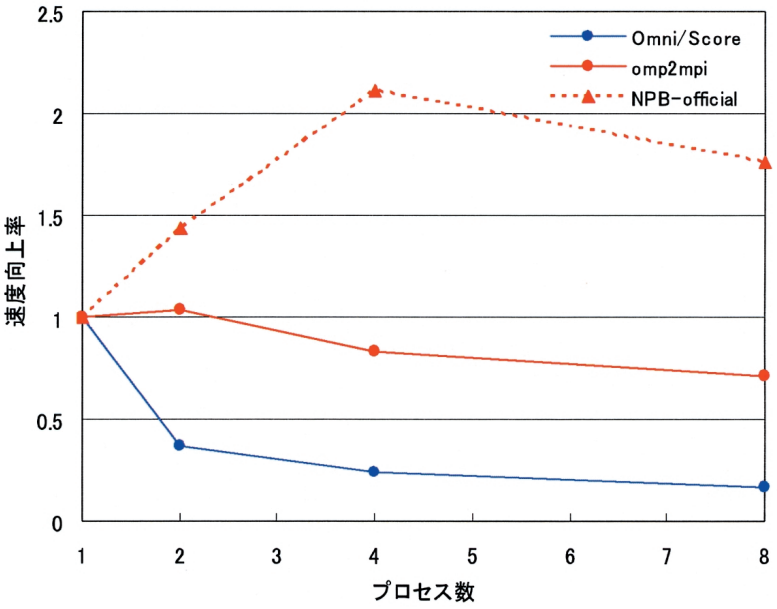

図4.3 *CG*の速度向上率

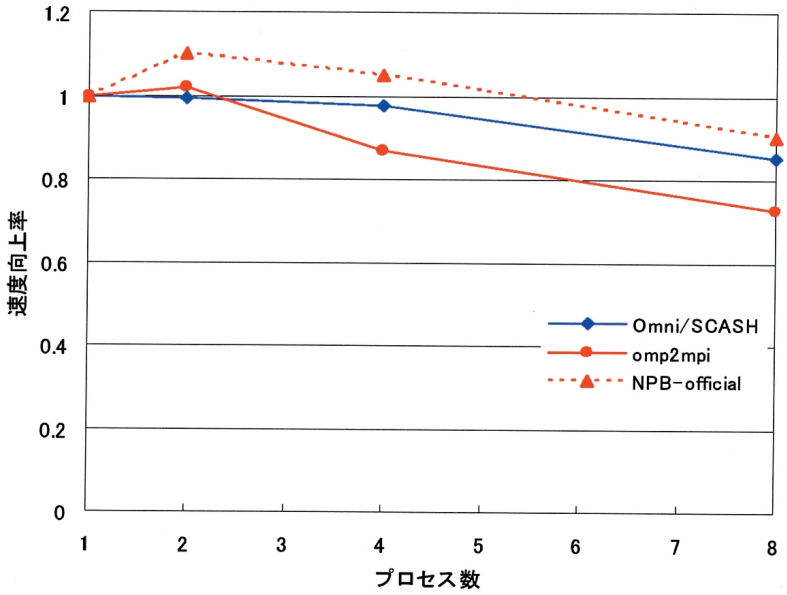

図4.4 *IS*の速度向上率

5 おわりに

本研究では，OpenMPプログラムをクラスタシステムなどの分散メモリ型並列システム上で実行可能なMPIプログラムへ変換を行うトランスレータの開発を行った．*laplace*やNPBの*EP*による実行結果により，局所性の高いOpenMPプログラムを入力コードとした場合，出力コードとして得られたMPIプログラムにおいても，十分な並列処理効果がみられることがわかった．従って，ユーザは局所性の高いOpenMPプログラムを記述することができれば，本トランスレータを用いることで，システム構築が比較的容易な分散メモリ型並列システムにおいても，相当の並列処理効果が期待できるということがいえる．しかし，NPBの*CG*や*IS*などの，もともと並列化による速度向上がみられないOpenMPプログラムの場合，本トランスレータを用いた実行においても**MPI_Allreduce**関数，**MPI_Bcast**関数などの集合通信に起因する通信オーバヘッドにより並列効果が上がらず，逐次実行よりも実行効率が悪い結果となった．Omni/SCASHとの比較では，NPBの*IS*において，本トランスレータを用いた手法の方が良い値を得ることができなかった．しかし，*EP*や*CG*においては，Omni/SCASHよりも速度向上がみられる結果を得ることができ，本トランスレータの有用性を示すことができた．

また，並列実行方式について注目すると，本トランスレータを用いた手法では，OpenMP指示文を挿入したCプログラムを入力プログラムとした．これにより，逐次プログラムを入力とする自動並列化コンパイラと比較しても，並列化ブロックの検出などの過程においてトランス

レータ（コンパイラ）の負担を大幅に削減でき，最適化処理に重点をおくことができると考えられる．また，ユーザが並列処理の動作を把握したい場合，Omni/SCASHを利用するときのOmni/OpenMPコンパイラの中間コードは独自のライブラリを呼び出す形式で表現されているのに対して，本方式ではMPIプログラムであるため，ユーザの出力プログラムに対するreadabilityの高さも長所といえる．そして，SCASHの構築の際に必要となるLinuxカーネルの変更が必要ないことも本研究の特徴であり，これにより，「ユーザの負担の軽減」が期待できる．さらに，MPIライブラリを備えたシステムであればどのようなシステムでも実行できるというscalabilityの高さも本並列実行方式の長所といえる．

今後の課題として，

⑴ **DFG**におけるsharedデータの手続き間解析とその実装．

⑵ 並列化が有効でない箇所を逐次化する場合の構造ブロック判定条件の導出．

⑶ トランスレータの適用範囲の拡大とさらなる最適化．

⑷ 速度向上が期待できるOpenMPプログラム，MPIプログラムを生成するための並列処理システム共通の並列型言語と実行方式の提案．

などが挙げられる．本研究の最終目標ともいえる⑷を実現するためには，共有メモリ型並列処理システムと分散メモリ型並列処理システムのプログラミングモデルの相違点について，さらに詳細な解析を行う必要がある．

参 考 文 献

[1] 新實治男：「OpenMPによる並列プログラミング“入門”」，京都産業大学ハイテクリサーチプロジェクト研究成果中間報告書 pp.99-115，(2003)．

[2] “PC Cluster Consortium,” http://pdswww.rwcp.or.jp/

[3] 福岡岳穂：「OpenMPによる粗粒度タスク並列実行方式」，電気通信大学大学院　修士課程学位論文(2001)．

[4] 佐藤茂久，草野和寛，佐藤三久：「OpenMP並列プログラムのデータフロー解析手法」，情報処理学会，HPC研究会，No.082-013（2000）．

[5] 佐藤茂久，佐藤三久：「OpenMP向けコンパイラ支援ソフトウェアDSMにおける最適化コンパイル手法」，情報処理学会，HPC研究会，Vol.42，No.SIG12-007（2001）．

[6] 中田育男著：「コンパイラの構成と最適化」，朝倉書店（1999）．

[7] “NAS Parallel Benchmarks,” http://www.nas.nasa.gov/NAS/NPB/

[8] 横塚芳明：「Scoreクラスタシステムの環境構築とその性能評価」，京都産業大学特別研究報告書(2003)．

注

1) RWCPが開発したソフトウェア分散共有メモリシステム
2) MPI-2では，動的プロセス生成，Remote Memory Access (RMA),並列IOなどの機能が追加されている．本研究で扱うMPIはMPI-1準拠とする．
3) 本稿では，実行当時の規模，構成要素を記述している．現在では，hr101～hr104までの4ノードが新たに増設されており，104ノード，216CPUのクラスタシステムとなっている．