アルゴリズム 60－8
（1998．1．22）

SPMDプログラムを生成するWork-Time C処理系の実現

岸部 祥典　小河原 徹　藤本 典幸　萩原 兼一

大阪大学 大学院基礎工学研究科 情報数理系専攻

我々は並列プログラミング言語としてワーク・タイムモデルに基づく並列言語Work-Time C (WTC)を提案している．WTCはグローバルなアドレス空間を持ち，プロセッサという概念はないため高い抽象度で並列アルゴリズムが記述可能である．本稿ではWTCで記述されたプログラムを，メッセージ通信ライブラリMPIの呼び出しを含むC言語のSPMD (Single Program Multiple Data)プログラムに変換するトランスレータについて述べる．効率の良い並列プログラムを生成するために，本WTC処理系で行っている通信最適化について説明する．また，WTCでは関数の並列呼び出しや並列再帰などを記述できるが，それらをどのように処理しているのかについて述べる．

Implementation of A Work-Time C Compiler to Generate SPMD Programs

Yoshinori Kishibe　Toru Kogawara　Noriyuki Fujimoto　Ken-ichi Hagihara

Department of Informatics and Mathematical Science,
Graduate School of Engineering Science, Osaka University

We propose Work-Time C (WTC) as a parallel programming language based on Work-Time model. In WTC we can describe parallel programs without consideration for the number of available processors and interprocessor communication. WTC programs allow functions to be called recursively in parallel. So we can use WTC programs to implement parallel algorithms at a natural and high level. In this paper we describe how to translate a WTC program to the corresponding SPMD-type C program with MPI calls. We also describe communication optimization to generate high performance parallel programs.

1. はじめに

分散メモリマシン用の並列プログラムを記述する方法として従来からよく用いられてきたものは，逐次言語にメッセージ通信ライブラリ呼び出しを挿入するという方法であった．この方法では，プログラマは利用できるプロセッサ数やデータを各プロセッサにどのように分散させるのかを考えなければならない．さらに，プロセッサ間の通信や同期を明示的に記述しなければならない．このため，プログラマへの負担が大きいものになってしまう．その負担を軽減するためにHigh Performance Fortran[5], Dataparallel C[4], NCX[8]のようなデータ並列言語が提案されている．しかし，依然としてプロセッサ数やデータの分散を考慮しなければならない．

これに対して，プロセッサ数やデータの分散などの物理的詳細を記述せず，並列プログラムを論理的に高水準な観点から記述できるモデルとして，ワーク・タイムモデル[6]が知られている．我々はこのワーク・タイムモデルに基づくWork-Time C言語[3]（以下WTC言語）を提案している．WTC言語では，プロセッサという概念はなく，データの分散もプログラマが考慮する必要はない．このため，プログラマは，プロセッサ間の通信や同期を記述する必要はなく，並列アルゴリズムの本質的な部分のみを記述すればよい．並列アルゴリズムを学習する上で，WTC言語は並列プログラミング言語として適している．WTC言語で記述されたプログラムは，逐次ワークステーション上で動作するシミュレータとデバッガを用いることにより，シミュレート実

行とデバッグが可能である[3].

本稿では, WTC 言語で記述されたプログラムを, 実行時ライブラリ呼び出しを挿入した C 言語の SPMD (Single Program Multiple Data)プログラムに変換する WTC トランスレータについて述べる. 本 WTC 処理系は, このトランスレータと実行時にプロセッサ間通信やプロセッサの並列実行を制御する実行時ライブラリからなる. 生成した C 言語プログラムは対象とする並列計算機でコンパイルされ, 実行時ライブラリがリンクされる. 実行時ライブラリは, メッセージ通信ライブラリとして MPI[7]を採用しているため, MPI をサポートしている並列計算機で生成した C 言語プログラムを実行することが可能である.

2. ワーク・タイムモデルに基づく WTC

PRAM(Parallel Random Access Machine)アルゴリズムの性能評価基準として, ワークとタイム[6]が重要である. この評価基準が明確になるプログラム記述モデルとしてワーク・タイムモデルが知られている. このモデルは, 図 1に示すように各時刻に並列に実行したい命令を記述するモデルであり, ある時刻に含まれる命令がすべて並列に実行されたのちに, 次の時刻の命令が実行される. このモデルでは, プロセッサという概念を取り去り, 論理的な共有メモリ空間を仮定しているために通信という概念もない. これらの特徴により, プログラマはプロセッサ数やデータの分散, プロセッサ間の通信など

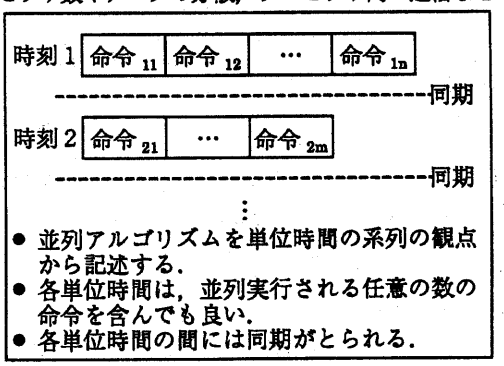


図 1 ワーク・タイムモデル

```
(1) double a[5];
(2)
(3) par i = 1 to 3 do {
(4)   a[i] = a[i-1] + 1;
(5)   b[i] = a[i+1];
(6) }
```

図 2 並列構文 par の記述例

表 1 配列の値の変化

	a[0]	a[1]	a[2]	a[3]	a[4]
実行前	10	20	30	40	50
実行後	10	11	21	31	50

	b[0]	b[1]	b[2]	b[3]	b[4]
実行前	0	0	0	0	0
実行後	0	21	31	50	0

を気にする必要はなく, 並列に実行したい命令のみを記述すれば良いため, 高い抽象度で並列プログラムを記述することができる.

WTC 言語はこのモデルに基づく並列処理を記述するために, 主に並列構文 par を C 言語に追加したものである. この par 文中に記述された文は全て並列に実行される. また, プログラムの各ステップの間には同期がある. さらに, 同一ステップにおいても, 並列に実行される代入文では, 右辺の評価と左辺への代入の間に同期が存在する. 図 2のプログラムコードを例に説明すると, 行(3)の代入文は後者の同期のために, par 文のインデックス変数 i がとる値の全範囲にわたって右辺の評価を並列に行った後, 左辺への代入が並列に行われる. 行(4)の代入文は前者の同期のために, 行(3)の代入文の実行が全て終わったあとに実行される. 以上より, 図 2のプログラムコードを実行すると 配列 a, b の値は表 1のように変化する.

par 文をネストして記述することが可能であり, 並列再帰も記述できる.

3. WTC トランスレータの概要

WTC トランスレータは WTC プログラムと, プロセッサ数やデータの分散などプログラマによる指示を記述したトランスレータ指示ファイルを入力する. WTC トランスレータは WTC プログラムに含まれる par 文を, トランスレータ指示ファイルをもとに, エミュレーションループ[4]に変換し, これに実行時ライブラリ呼び出しを挿入した C 言語の SPMD プログラムに変換する. 図 3に本処理系の概要を示す.

3.1. プロセッサ数

WTC 言語にはプロセッサという概念はないため, 利用するプロセッサ数をトランスレータ指示ファイルに記述し, WTC トランスレータに入力する. 便宜上, プロセッサ数を以下の表記を用いて表す.

```
processors("プロセッサ数")
```

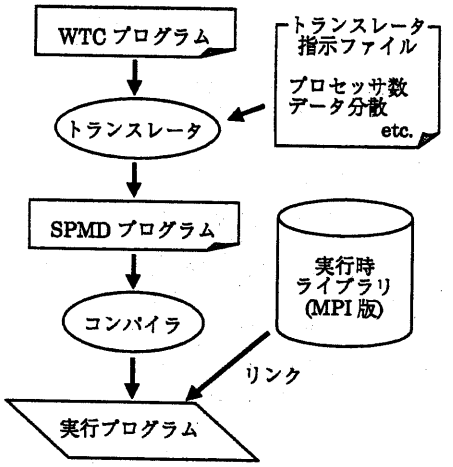


図 3 WTC 処理系の概要

3.2. データの分散

プロセッサと同様に，WTC 言語にはデータの分散という概念はないので，トランスレータがデータの分散を決定する．データの分散の決定方法は，後述のブロック分割を基本にした各プロセッサへのデータの均等な分散である．並列プログラムの性能はデータの分散に大きく依存しているため，プログラマがデータの分散を指示したい場合もある．この場合，データの分散をトランスレータ指示ファイルに記述することによりデータの分散を指定することができる．

データの分散は配列の各次元ごとにサイクリック分割[1]，ブロック分割[1]，ブロック・サイクリック分割[1]が指定可能である．データの分散は以下のように指定する．

```
distribute "配列名"("分割1","分割2",...)
```

各次元の分割の指定は cyclic("*block size*")で行う．"*block size*"が 1 のときはサイクリック分割となり，それ以外のときはブロックまたはブロック・サイクリック分割となる．分割の指定が "*" のときは分割しないことを表す．説明の都合上，ブロック分割の場合は，分割の指定に"block"を用いることにする．

図 4に 16×16 の二次元配列 a, b におけるデータ分散の例を示す．

3.3. 命令のプロセッサへの割り当て

プログラム中の代入文をどのプロセッサで実行するかは，owner computes rule [2](以下 OC 規則)に基づいて行う．OC 規則は，代入文の左辺値を持つプロセッサにその計算を割り当てるものである．

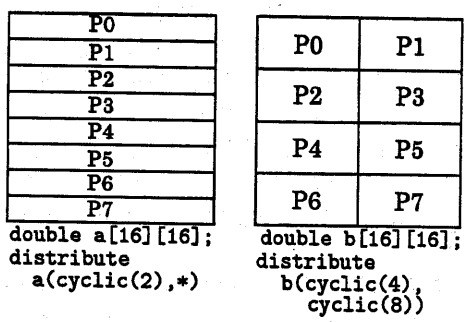


図 4 データ分散の例

4. プログラム変換の基本方針

WTC プログラム中の文は，文法と実行時のコンテクストから以下の 4 つのクラスに分類する．

- S1: 文法的に見て par 文外に記述されている文で，その文を含む関数が逐次実行される．
- S2: 文法的に見て par 文中に記述されている文で，その文を含む関数が逐次実行される．
- S3: 文法的に見て par 文外に記述されている文で，その文を含む関数が並列実行される．
- S4: 文法的に見て par 文中に記述されている文で，その文を含む関数が並列実行される．

ただし，本トランスレータでは逐次ループの並列化は行わない．

4.1. クラス S1 に属する文の変換

その文を実行するのに必要なデータに対して，計算マッピングとデータ分散からデータの転送を行う通信コードを挿入する．

図 5にこのクラスの文の変換例を示す．ただし，プログラムリスト中で行番号がない行はトランスレータ指示ファイル中に記述されているものである．

```
     processor(4);
     distribute a(block)
(1)  double a[16];
(2)  a[0] = a[0] + a[15];
     (a) WTC プログラム
(1)  double a[16];
(2)  if (PID == 3) send(a[15],P0);
(3)  if (PID == 0) {
(4)    recv(a[15],P3);
(5)    a[0] = a[0] + a[15];
(6)  }
     (b) SPMD プログラム
```

図 5 クラス S1 に属する文の変換例

4.2. クラスS2に属する文の変換

WTC言語では，並列に実行される代入文において，右辺の評価と左辺への代入の間に同期がとられる．そこで，par文中の代入文は一時的な配列を用いて，右辺の評価を並列に行い，その後に左辺へ代入するように変換を行う．図6のWTCプログラムコードに対して，この変換を行った結果を図7に示す．

次に，データ分散と計算マッピングに基づき，各プロセッサが担当すべきエミュレーションループの範囲を求めるコードを生成する．そして，各文の実行に必要なデータの転送を行う通信コードを挿入したエミュレーションループに変換する．図7のWTCプログラムに対してこの変換を行い，生成されたSPMDプログラムを図8に示す．

par文中の関数呼び出しは，その関数をpar文中にインライン展開したように実行される．そのため，並列に呼び出された関数のインスタンス間で，その関数内の各文の実行は同期をとらなければならない．図9を用いて説明すると，WTCプログラムにおいて関数gが並列に実行される．gの複数のインスタンスの処理をSPMDプログラムで実現するためには，複数の呼び出しを処理できるように引数をベクトル化する（図9(b)の行(3)～(6)参照）．関数が返り値をもつ場合は並列度の分だけ返り値が存在するため，その返り値を別の領域に確保しておかなければならない(図9(b)の行(2)参照)．そのために，関数の引数に返り値の領域へのポインタを追加し，その領域へ返り値を格納するようにする（図9(b)の行(15)，(18)参照）．

```
(1) double a[16], b[16];
(2)
(3) par i = 0 to 15 do {
(4)   a[i] = b[i] * 2;
(5)   b[i] = b[i] + 1;
(6) }
```

図6 WTCプログラムコード

```
(1) double a[16], b[16];
(2)
(3) par i = 0 to 15 do
(4)   tmp1[i] = b[i] * 2;
(5) par i = 0 to 15 do
(6)   a[i] = tmp1[i];
(7) par i = 0 to 15 do
(8)   tmp2[i] = b[i] + 1;
(9) par i = 0 to 15 do
(10)  b[i] = tmp2[i];
```

図7 一時的な配列を用いた代入文の分割

```
(1) double a[16], b[16];
(2)
(3) (各PEが担当する範囲を求める)
(4) for ( 各PEが担当する範囲 ) {
(5)     (b[i]の転送);
(6)     tmp1[i] = b[i] * 2;
(7) }
(8) for ( 各PEが担当する範囲 )
(9)     a[i] = tmp1[i];
(10) (各PEが担当する範囲を求める)
(11) for ( 各PEが担当する範囲 ) {
(12)    tmp2[i] = b[i] + 1;
(13) }
(14) for ( 各PEが担当する範囲 )
(15)    b[i] = tmp2[i];
```

図8 エミュレーションループへの変換例

```
(1) int a[16];
(2) void f(void) {
(3)   par i = 0 to 15 do
(4)     a[i] = a[i] + g(i);
(5) }
(6) int g(int x) {
(7)   return(h(x));
(8) }
```

(a) WTCプログラム

```
(1) int a[16];
(2) void f(void) {
(3)   int buf[16];  /* 返り値領域 */
(4)   int arg[16];
(5)   arg[0] = 0;  /* 引数のベクトル化 */
(6)   arg[1] = 1;
(7)    …
(8)   g(buf,arg);
(9)   (各PEが担当する範囲を求める)
(10)  for (各PEが担当する範囲) {
(11)    tmp[i] = a[i] + buf[i];
(12)  }
(13)  for (各PEが担当する範囲)
(14)    a[i] = tmp[i];
(15) }
(16) void g(int *buf, int *x) {
(17)   (実行時に各PEが担当する範囲を求める)
(18)   for (各PEが担当する範囲)
(19)     buf[i] = h(x[i]);
(20) }
```

(b) SPMDプログラム

図9 関数の並列呼び出し

また，分散メモリマシン上での実行において，この並列に呼び出される関数内で通信が必要な場合，その通信に関わるプロセッサは，その通信を行うためにその関数を実行していなければならない．そのようなプロセッサ集合を一般にコンパイル時に求め

ることができない．従って，並列ループ中の関数呼び出しは，全プロセッサでその関数を実行するように変換する（図 9(b)の行(7)参照）．

4.3. クラス S3 に属する文の変換

文法的には逐次実行のように記述されている文でも，並列に呼ばれる関数内では全ての文が並列に実行される．従って，4.1節の変換に加えて，並列に呼び出された関数の数に対応する数の文を実行するエミュレーションループを追加する．このエミュレーションループの範囲は並列に呼び出された関数の数に依存するため，プログラムの並列度を管理している実行時ライブラリの関数を呼び出すことによって各プロセッサが担当する範囲を求める（図 9(b)の行(16)〜(18)参照）．

4.4. クラス S4 に属する文の変換

4.2節の par 文に対する変換だけでなく，4.3節と同様に並列に呼び出された関数の数に対応する数の文を実行するようにループを追加する．

5. エミュレーションループ最適化

性能の良い SPMD プログラムを得るためには，エミュレーションループにおけるオーバヘッドを削減することが重要になる．

5.1. 一時的な配列の削除

par 文中の代入文は，図 8のように変換される．しかし，ループの各イタレーション間でデータ依存がない場合には，右辺の評価のあと直ちに代入をしても正しい結果を得る．このような場合，一時的な配列を用いる必要はない．その結果，右辺を評価した結果を一時的な配列に代入をするエミュレーションループを削除することができる．

図 8に対してこの最適化を適用したものを図 10に示す．

```
(1)  double a[16], b[16];
(2)
(3)  (各PEが担当する範囲を求める)
(4)  for ( 各PEが担当する範囲 ) {
(5)      (b[i]の転送);
(6)      a[i] = b[i] * 2;
(7)  }
(8)
(9)  (各PEが担当する範囲を求める)
(10) for ( 各PEが担当する範囲 ) {
(11)     b[i] = b[i] + 1;
(12) }
```

図 10 エミュレーションループ削除適用例

5.2. エミュレーションループの統合

二つのエミュレーションループが隣接する場合，両ループでデータ依存がなく，ループの範囲が同じである場合は二つのループを統合することにより，ループの初期化（各プロセッサの担当範囲を求める）やループの終了判定，ループ制御変数の更新にかかるオーバヘッドを削減することができる．

図 10においてループの範囲が同じであると仮定した場合，この最適化を適用した例を図 11に示す．

```
(1)  double a[16], b[16];
(2)
(3)  (各PEが担当する範囲を求める)
(4)  for ( 各PEが担当する範囲 ) {
(5)      (b[i]の転送);
(6)      a[i] = b[i] * 2;
(7)      b[i] = b[i] + 1;
(8)  }
```

図 11 エミュレーションループ統合適用例

5.3. par 文の逐次化

par 文はエミュレーションループを用いて変換される．しかし，並列プログラムによっては，利用できるプロセッサ数より並列度が大きい場合がある．このような場合は，ネストしている内側の par 文のいくつかに対して，逐次ループに変換することによりエミュレーションループに変換した際のオーバヘッドが生じないようにしている．

6. 通信最適化

性能のよい並列プログラムを生成する上で，通信のオーバヘッドを削減することが重要となる．本 WTC 処理系では冗長通信の削除，通信の一括化を行い，通信回数を減らすことにより通信のオーバヘッドを削減している．

6.1. 冗長通信の削除 [1]

他のプロセッサがもつある変数の値を異なる時刻に参照する場合，その値が最初に参照した値から変更されないのであれば，最初に受信したデータを再利用することが可能である．この場合，最初に参照されるときにデータの転送を行い，それ以降の参照は受信したバッファを参照するようにし，データの再転送を行わない．例えば，ループ中において参照される変数で，各イタレーションでその値が変更されない場合は，そのデータの転送をループの外側に追い出すことができる．その結果そのデータに対する転送はループの前に一度すればよいことになる．

図 12はこのような参照を含む例とループ追い出

```
(1) double a[16], b[16];
    processors(4)
    distribute a(block), b(block)
(2) for (i=4*PID; i<4*(PID+1);     {
(3)   if (PID==0) send(b[0],allPE);
(4)   else recv(b[0],P0);
(5)   a[i] = a[i] + b[0];
(6) }
          ⇓
(1) double a[16], b[16];
(2)
(3) if (PID==0) send(b[0],allPE);
(4) else recv(b[0],P0);
(5) for (i=PID*4; i<(PID+1)*4; i++)
(6)   a[i] = a[i] + b[0];
```

図 12 ループ追い出し適用例

しを適用した結果である.

6.2. メッセージの集成 [1]

一般に，データのサイズが同じであれば，通信にかかる時間とメモリコピーにかかる時間を比較すると，通信にかかる時間のほうが大きい．そこで，同一プロセッサ間で複数のデータを転送する場合は，複数のデータをいったんバッファに格納し，格納したデータを一括して転送する.

6.3. メッセージのベクトル化 [1]

ループの各イタレーションでデータの転送が必要な場合，各イタレーションで通信を行うことになる．しかし，参照されるデータが連続領域に格納されている場合は，各イタレーションでデータを転送するのではなく，転送されるデータの範囲を求め，ループの前にそのデータをベクトルとして一括転送する.

図 13にメッセージのベクトル化を適用可能な例とベクトル化を行った結果を示す．ただし，配列 a の添字が“i:j”である場合は，a[i]から a[j]までの全ての要素を表すものとする.

6.4. メッセージの融合 [1]

メッセージのベクトル化などを行うことにより，二つのデータ転送に対して転送されるデータの範囲が重複するような場合がある．このような時は，二つの転送を一つのデータ転送に融合することにより，重複したデータの転送を行わないようにする.

7. 関数の並列呼び出しにおける最適化

並列に関数が呼び出される場合，4節のように，引数のベクトル化と返り値のための領域確保などを行わなければならなかった．これらは，並列に呼び

```
(1) double a[16][16], b[16][16];
    processors(4)
    distribute
       a(block,*), b(*,block)
(2) for (i=0; i<15; i++) {
(3)   if (PID==0) send(a[0][i], P1);
(4)   if (PID==1) recv(a[0][i], P0);
(5)   if (PID==1) b[i][0] = a[0][i];
(6) }
          ⇓
(1) if(PID==0) send(a[0][0:15],P1);
(2) if(PID==1) recv(a[0][0:15],P0);
(3) for (i=0; i<15; i++)
(4)   if(PID==1) b[i][0] = a[0][i];
```

図 13 メッセージのベクトル化

出される関数のインスタンス間にデータ依存がある場合や，データの転送が必要である場合を考慮しているためである.

7.1. 並列な関数呼び出しの逐次化

並列に呼び出される関数 f 内で，さらに別の関数を呼び出すことがない場合は，f のインスタンス間にデータ依存がなく，データの転送も必要なければ，f は並列に呼び出されたように処理する必要はない．そこで，f を逐次的に呼び出すことによって，引数のベクトル化と返り値のための領域確保を削除することができる．さらに，関数呼び出しを逐次化することにより，4.3節の関数を並列に実行する際に必要だったエミュレーションループも削除することができる.

図 14に関数の逐次化を適用可能な例とそれを行った結果を示す.

```
(1) int a[16];
    distribute a(block)
(2) void f(void) {
(3)   par i = 0 to 15 do
(4)     a[i] = g(i);
(5) }
(6) int g(int x) {
(7)   return(x*x);
(8) }
    (a) WTC プログラム
(1) int a[16];
(2) void f(void) {
(3)   for (i=4*PID; i<4*(PID+1); i++)
(4)     a[i] = g(i);
(5) }
(6) int g(int x) {
(7)   return(x*x);
(8) }
    (b) SPMD プログラム
```

図 14 関数呼び出しの逐次化

並列に呼び出される関数内で，さらに別の関数を呼び出す場合でも，その呼び出す関数が逐次実行可能であれば，関数の逐次化を行うことができる．

7.2. 並列再帰

並列再帰は，並列に関数が呼び出される場合の一種であるが，関数が呼び出される回数，つまり再帰の深さが分からないという問題点がある．この場合，並列再帰の途中で通信が必要になることを考えて，全プロセッサで再帰処理を行わなければならず，SPMD プログラムの性能が低下する．

しかし，並列再帰を含むプログラムでは，ある再帰の深さ d に対して，深さ d + i (i>0)で処理されるデータの集合は，深さ d で処理されるデータの部分集合であることが多い．この条件が成り立つ関数であることがトランスレータ指示ファイルに記述されているならば，計算マッピングである OC 規則を緩和し，各プロセッサへ再帰呼び出しそのものを割り当てるという方法をとる．

この計算マッピングの変更は以下のように行う．ここでは，ある再帰の深さにおけるプログラムの並列度を変更の基準とし，この並列度はプロセッサ数を超えるまで更新され，いったん超えると再帰呼び出しが終了するまで更新されないものとする．

```
(1) void f(int x) {
(2)   if (x > 1) {
(3)     . . .
(4)     par i=1 to 2 do
(5)       f(x-i);
(6)   }
(7) }
    (a) WTC プログラム

(1) void f(int *x) {
(2)   b[i]=(x[i]>1); /* 0<=i<=並列度-1 */
(3)   . . .
(4)   if (並列度 < プロセッサ数) {
(5)     if(b[0]||b[1]||..|| b[並列度-1]) {
(6)       (引数のベクトル化);
(7)       f(arg);
(8)     }
(9)   } else {
(10)    (データの再分散・複製を行う)
(11)    (各 PE が担当する範囲を求める)
(12)    for (各 PE が担当する範囲)
(13)      (引数のベクトル化);
(14)      if (b[i])
(15)        (ローカルに f を再帰呼び出し)
(16)  }
(17) }
    (b) SPMD プログラム
```

図 15 並列再帰の変換例

- 並列度がプロセッサ数より小さいとき
  再帰呼び出しを各プロセッサに割り当てるとアイドル状態のプロセッサができるので，4.3 節のように全プロセッサで処理を行う．
- 並列度がプロセッサ数より大きいとき
  各プロセッサのローカルメモリに，それ以降の再帰処理をするのに必要なデータが存在するようにデータの再分散・複製を行う．以降の処理は各プロセッサで独立に行う．

図 15に並列再帰を含むプログラムの変換例を示す．

8. 評価

本処理系の評価を「ガウスの消去法（ピボット選択なし）」と「行列積」のプログラムを用いて，並列計算機 AP1000 上で行った．図 16はガウスの消去法の，図 17は行列積のプログラムコードである．問題サイズは，それぞれ最適化なしのときが N=256，最適化ありのときが N=1024 である．評価は以下に示す台数効果を用いる．

$$\text{台数効果} = \frac{\text{逐次プログラムを 1PE で実行した時間}}{\text{生成した SPMD プログラムの実行時間}}$$

図 18の(a)に最適化を行わない場合，(b)に最適化を行った場合の結果を示す．最適化を行わない場

```
(1) double a[N][N+1];
    processor(P)
    distribute a(block,*)
(2) for (k=0; k<N; k++) {
(3)   if (PID == k/(N/P)) {
(4)     tmp_akk = a[k][k];
(5)     for (j=k; j<=N; j++)
(6)       a[k][j] /= tmp_akk;
(7)   }
(8)   for(i=N/P*PID;i<N/P*(PID+1);i++)
(9)     for (j=k; j<=N; j++) {
(10)      (a[k][j]の転送)
(11)      if (i != k) {
(12)        tmp_aik = a[i][k];
(13)        a[i][j] -= tmp_aik * a[k][j];
(14)      }
(15)    }
(16) }
              (a) 最適化なし

(8)   (a[k][k:N]の転送)
(9)   for(i=N/P*PID;i<N/P*(PID+1);i++)
(10)    if (i != k) {
(11)      tmp_aik = a[i][k];
(12)      for (j=k; j<=N; j++)
(13)        a[i][j] -= tmp_aik * a[k][j];
(14)    }
              (b) 最適化あり
```

図 16 「ガウスの消去法」のプログラムコード

```
(1) double a[N][N], b[N][N], c[N][N];
    processor(P)
    distribute
      a(block,*), b(block,*), c(block,*)
(2) for (k=0; k<N; k++)
(3)   for (i=N/P*PID; i<N/P*(PID+1); i++)
(4)     for (j=0; j<N; j++) {
(5)       (b[k][j]の転送)
(6)       c[i][j] += a[i][k] * b[k][j];
(7)     }
```

(a) 最適化なし

```
(2) for (k=0; k<N; k++) {
(3)   (b[k][0:N-1]の転送)
(4)   for (i=N/P*PID; i<N/P*(PID+1); i++)
(5)     for (j=0; j<N; j++)
(6)       c[i][j] += a[i][k] * b[k][j];
(7) }
```

(b) 最適化あり

図 17 「行列積」のプログラムコード

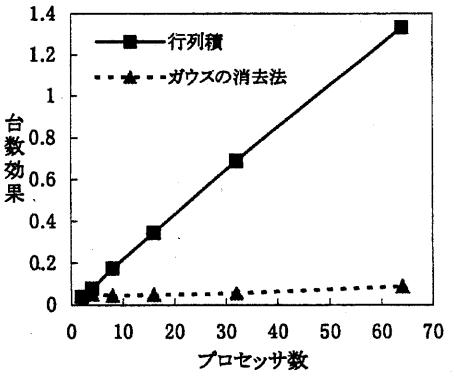
合は，通信回数が非常に多いため台数効果が得られなかった．しかし，最適化を行う場合は，いずれのプログラムに対しても，メッセージのベクトル化と冗長通信の削除により通信回数が大幅に減少し，台数効果が得られた．

9. まとめ

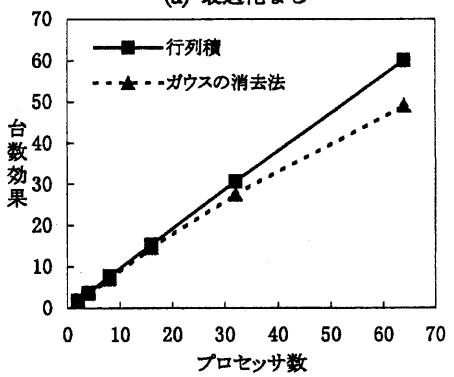
ワーク・タイムモデルに基づく WTC 言語で記述されたプログラムを SPMD プログラムに変換するトランスレータについて述べた．WTC は高い抽象度で並列アルゴリズムを記述可能である．本トランスレータは，WTC 言語にはないデータ分散や計算マッピングを補完し，メッセージ通信ライブラリとして MPI を用いることにより移植性の高いプログラムを生成することができる．AP1000 上でいくつかのプログラムに対して評価を行い，通信最適化の効果を示した．また，関数の並列呼び出し，並列再帰の処理について述べた．

7節で説明した関数のインスタンス間でのデータ転送の必要性をどのように判定するのか，またその必要性がある場合はどのように処理すれば，プログラムの性能をよくすることができるのかが，今後の課題として挙げられる．

謝辞 本研究は一部文部省科研費(平成９年度基盤研究(C)(2)09680336)，並列・分散処理研究推進機構の補助による．並列計算機 AP1000 を利用させて頂いた富士通(株)に感謝する．



(a) 最適化なし



(b) 最適化あり

図 18 評価プログラムの実行結果

参 考 文 献

[1] Bacon, F. D., Graham, L. S., and SHARP, J. O.: "Compiler Transformations for High-performance Computing", ACM Computing Surveys, Vol.26, No.4, pp.345–420(1994).

[2] Callahan, D. and Kennedy, K.: "Compiling programs for Distributed Memory Multiprocessors", The Journal of Supercomputing, pp.171– 207(1988).

[3] 藤本典幸, 乾和弘, 前田昌也, 柘植宗俊, 萩原兼一："ワーク・タイムモデルに基づく並列プログラミング言語 Work-Time C の提案と EWS 用コンパイラの実装", 日本ソフトウェア科学会第 13 回大会論文集, pp.205–208 (1996).

[4] Hatcher, P.J. and Quinn, M.J.: "Data-Parallel Programming on MIMD Computers", The MIT Press (1991).

[5] High Performance Fortran Forum: "High Performance Fortran Language Specification Version 1.1", http://www.crpc.rice.edu/HPFF/hpf1/hpf-v11/hpf-report.html (1994).

[6] JáJá, J.: "An Introduction to Parallel Algorithms", *Addison-Wesley Publishing Company* (1992).

[7] Message Passing Interface Forum: "MPI: A Message-Passing Interface Standard Version 1.1", http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html (1995).

[8] 湯浅太一, 貴島寿郎, 小西浩："データ並列計算のための拡張 C 言語 NCX", 信学会論文誌 D-I, J78-D-1, 巻２号, pp.200–209 (1993)