メッセージ交換型並列計算機のための並列化コンパイラ TINPAR

平成7年2月15日
三吉 郁夫

次世代のスーパーコンピュータの構成方式として並列計算機が期待されている。特にメッセージ交換型の並列計算機は、通信を行うコードを陽に記述することによってプログラムを高度に最適化することができるため、高い実効性能を達成することが期待できる。しかし現実には、最適なプログラムを作成するためには高度な技術が必要とされるだけでなく、基本的なプログラミング自体も容易であるとはいえない。このことより、簡単なプログラミングによってある程度の性能を引き出せる言語処理系が必須であるといえる。

そこで本研究では、メッセージ交換型並列計算機のための並列化コンパイラ TINPAR の開発を行った。TINPAR は拡張 Tiny language で記述された逐次プログラムを owner computes rule に従って並列化する。これによってユーザは、グローバルメモリ空間を想定して逐次プログラムを書くことができる。

TINPAR による並列化は、単純な並列化手法、高度な最適化手法、および専用の通信ライブラリによって特徴づけられる。すなわち TINPAR は、owner computes rule に従った単純な並列化を行った後、不用なコードの削除や通信コードの移動を行うことによって効率のよいオブジェクトコードを生成する。さらに、動的ベクトル化機能を持つ専用通信ライブラリをリンクすることによって、このオブジェクトコードでは効率のよい通信処理が実現される。

TINPAR はまず、ソースプログラムを効率よく並列化できるように、様々なプログラム変換による前処理を行う。次に TINPAR は owner computes rule に従って並列化を行うが、その結果生成されたコードには冗長な`if`文が数多く含まれており、通信によるレイテンシの隠蔽も行われていない。

そこで続いて TINPAR は、これらの問題点を改善するための最適化を行う。まずランタイムオーバーヘッドを削減するために、1) ループの実行範囲の縮小、2) 恒真/恒偽となる`if`文の削除、3)`if`文を削除するためのループ分割が行われる。これらは並列化の際に挿入された不用なコードを削除するための最適化手法であり、特に 1) と 3) が効果的である。一方`SEND`/`RECV`文をともに含むループのループ分配は、通信レイテンシを隠蔽するための最適化手法であり、通信の最適化に必要不可欠である。

また TINPAR の専用通信ライブラリは、動的ベクトル化機能を提供する。この機能は、通信ランタイムシステムが送信データをバッファリングし、長いメッセージとして送信するものであり、メッセージ送信の回数を大幅に削減する。さらにこの機能を十分に活用するための最適化として、`MSG_BUFFER_FLUSH`文の挿入も行われる。

これらの手法の有効性を示すために、簡単な数値処理プログラムを並列化して性能を評価した。この結果 64 プロセッサの AP1000 を用いて、行列積で 48.2 倍、ガウス消去法で 37.1 倍、ヤコビ法で 79.8 倍、SOR 法で 28.8 倍、FFT で 33.6 倍の加速率が達成された。

TINPAR:
A Parallelizing Compiler for Message-Passing Multiprocessors

February 15, 1995
Ikuo MIYOSHI

Parallel machines will be the next generation supercomputers. In particular, message-passing multiprocessors are expected to achieve high performance, because programs can be optimized with explicitly coded communication. However, not only writing well-tuned programs but also basic programming are not always easy nor practical. Therefore, a parallel language or a parallelizing compiler is strongly required to achieve high performance without complicated programming.

In this research, we have developed TINPAR, a parallelizing compiler for message-passing multiprocessors. TINPAR parallelizes programs written in an extended version of Tiny language, allowing users to write sequential programs with global address space.

The major features of TINPAR are simple parallelization policy, powerful optimization techniques, and the special communication library for the compiler. TINPAR parallelizes sequential programs following the owner computes rule, then it eliminates unnecessary codes and reschedules communication codes to derive efficient object codes. The communication library, which has a capability of *dynamic message vectorization*, is linked with the object codes and handles message passing efficiently.

The first step of TINPAR is program transformation with various techniques to make a program efficiently parallelized. Then, a parallelized code is obtained applying the owner computes rule. The code, however, has many redundant testing statements and too tight communication ones with long latency.

These inefficient statements are removed or modified by the next optimization step. *Narrowing loop bounds*, *eliminating unnecessary* `if` *statements*, and *loop splitting for more elimination of* `if` *statements* eliminate redundant codes generated during the parallelization. Among them, the first and the third one play important roles. On the other hand, *distribution of loops having* `SEND/RECV` *statements* is indispensable for hiding communication latency.

The communication library has a capability of dynamic message vectorization. With this technique, the runtime communication system buffers short messages and combines them into a long message to hide startup cost of message passing. *Insertion of* `MSG_BUFFER_FLUSH` *statements* is important optimization to utilize this capability.

We evaluated performance of various scientific programs to show the effectiveness of these optimization and communication techniques. As a result, TINPAR achieved 48.2 times speedup for 512×512 matrix multiplication, 37.1 times for 1024×1024 matrix Gaussian elimination, 79.8 times for 512×512 Jacobi method, 28.8 times for 512×512 SOR method, and 33.6 times for 64k FFT on AP1000 using 64 PEs.

1　はじめに

次世代のスーパーコンピュータの構成方式として並列計算機が期待されている。特にメッセージ交換型の並列計算機は、通信を行うコードを陽に記述することによってプログラムを高度に最適化することができるため、高い実効性能を達成することが期待できる。

しかし現実には、最適なプログラムを作成するためには高度な技術が必要とされるだけでなく、基本的なプログラミング自体も容易であるとはいえない。これらの具体的な理由としては、以下の点があげられる。

- プロセッサ間のデータの受渡しはすべてメッセージ通信で実現しなければならない。

  すなわち単なるデータの参照でさえ、ローカル/リモートのどちらに対するものであるかによって処理が大きく異なる。

- メッセージ通信のために用いられるライブラリでは多種多様な通信機能が提供されている [8]。

  しかしこれらの機能を活用するためにはある程度の知識および技術が必要とされる。

これらはいずれもプログラミングの本質、すなわちいかに計算するかという問題とは別の問題であるため、一般のユーザがこのような計算機を敬遠する大きな要因となる。それゆえこのような計算機が普及するためには、これらの点に配慮する必要がなく、簡単なプログラミングによってある程度の性能を引き出せる言語処理系が必須であるといえる。

そこで本研究では、メッセージ交換型並列計算機のための並列化コンパイラ TINPAR の開発を行った。TINPAR は拡張 Tiny language で記述された逐次プログラムを owner computes rule に従って並列化する。これによってユーザは、グローバルメモリ空間を想定して逐次プログラムを書くことができる。

TINPAR による並列化は、単純な並列化手法、高度な最適化手法、および専用の通信ライブラリによって特徴づけられる。すなわち TINPAR は、owner computes rule に従った単純な並列化を行った後、不用なコードの削除や通信コードの移動を行うことによって効率のよいオブジェクトコードを生成する。さらに、動的ベクトル化機能を持つ専用通信ライブラリをリンクすることによって、このオブジェクトコードでは効率のよい通信処理が実現される。

本稿では、まず 2 章で TINPAR の概要について述べた後、3 章では基本となる並列化の手法について、4 章では最適化のために用いられる種々の手法について、5 章ではオブジェクトコードで効率のよい通信処理を実現するための専用通信ライブラリについて、それぞれ述べる。続いて 6 章では、簡単な数値処理プログラムの並列化に対する有効性、各最適化手法の効果、およびオブジェクトコードの実行時間に対する通信処理の影響について述べる。その後 7 章で関連する研究について述べ、8 章でまとめる。

2　TINPAR の概要

TINPAR(TINy PARallelizer) は、現在我々が開発しているメッセージ交換型並列計算機のための並列化コンパイラである。本章では、TINPAR の設計方針、TINPAR の対象言語である拡張 Tiny language の概要、生成されるオブジェクトコードの形式、および TINPAR の内部構成について述べる。

2.1　設計方針

TINPAR は、並列化コンパイラのための各種技法の研究プラットフォームとして位置づけられる。このため TINPAR の設計方針の選択には、細心の注意が必要となる。本節では、TINPAR の設計に関する基本的な方針について述べる。

並列化の方針　メッセージ交換型の並列プログラムを作成する場合には、効率のよい通信処理を行うことによって、通信によるレイテンシを可能な限り低減することが重要である。一方逐次プログラムを並列化する場合には、必ずしもそのアルゴリズムが並列プログラムのものとして最適であるとは限らないため、アルゴリズム面での見直しが必要となることもある。

そこでこれらの問題を個別に扱うために、TINPAR による並列化は次のような基本方針をとる。

- 効率よく並列化できるはずのプログラムを、効率よく並列化する。
- 並列化しにくいプログラムを、効率よく並列化できるはずのプログラムに変換する。

これらはそれぞれ、強力な最適化手法、および柔軟なプログラム変換手法によって実現できる。またこれらを両立させるためには、「効率よく並列化できるはずのプログラム」の条件を明確にすることが必要である。そこで TINPAR では、並列化の基本戦略や最適化手法の適用条件を極力簡潔明瞭化することを目指す。

並列化の基本戦略　TINPAR は数値処理プログラムを並列化の対象とする。一般に数値処理プログラムの並列化では、データ並列性、すなわち多数のデータに対して同様かつ独立な処理を行う場合に生じる並列性を利用しやすい。

そこで TINPAR では、並列化の基本戦略として owner computes rule を用いることにより、ソースプログラムからデータ並列性を抽出する。

owner computes rule

> 各プロセッサは、それぞれが所有するデータのための演算を担当する。

owner computes rule を並列化の基本戦略として用いる場合には、データを各プロセッサに割り付けることが必要であり、これはデータの分散と呼ばれる。このとき各データを割り付けられたプロセッサは、それぞれのデータの所有者プロセッサと呼ばれる。また配列データを各プロセッサに割り付ける単位に分解することは、データの分割と呼ばれる。

owner computes rule に従って並列化されたプログラムでは、各プロセッサはそれぞれに割り付けられたデータに対する代入文を実行する。このため、プログラムに並列性が存在し、かつ各プロセッサへのデータの分散状態に対して参照の空間的局所性が存在する場合には、owner computes rule は効率のよい並列化戦略となる。また逐次プログラムを並列化する場合には、プログラム中のデータの分散法を指定するだけで並列化戦略を指示できるため、ユーザにとっては直観的で分かりやすいといえる。

対象とする並列計算機　TINPAR がオブジェクトコード生成の対象とする並列計算機のモデルは、一般にメッセージ交換型と呼ばれるものであり、以下のような特徴を持つ。

- MIMD 型の並列計算機である。
- 各プロセッサのメモリ空間は互いに独立しており、プロセッサ間のデータの受渡しはメッセージの送受信によって実現される。
- 送信側プロセッサから送られたメッセージは、明示的な受信動作が行われるまで受信側でバッファリングされる。またこのバッファリング容量は十分大きいものとする。

このため TINPAR のオブジェクトコードは、無限長バッファを持つ通信路が各プロセッサ間に存在することを仮定して動作する。

2.2　拡張 Tiny language

TINPAR のソースプログラムは拡張 Tiny language によって記述される。本節では、拡張 Tiny language の言語仕様の概要、および並列化に関係する機能について述べる。

2.2.1　言語仕様の概要

現在研究されている並列化コンパイラには、Fortran を対象言語としているものが多い。この理由の 1 つには、ユーザに対してプログラミング言語の連続性を保証することがあげられる。しかしこの言語仕様も FORTRAN77 から Fortran90[6] そして HPF(High Performace Fortran)[7] へと検討が加えられ、使用しないことが推奨されている機能も存在する。また並列化コンパイラのための技法を研究するにあたっては、この言語仕様に対して完全に準拠する必要もない。

これらの点を踏まえ、TINPAR の対象言語には拡張 Tiny language が採用されている。拡張 Tiny language は、ループ再構成ツール Tiny[14] のために設計された逐次型言語 Tiny language を拡張したものであり、Fortran に類似した簡潔な構文規則を持つ。なお我々が行った拡張としては、データ分割ディレクティブ、`while` 構文、スコープ規則の導入などがある。

拡張 Tiny language の言語仕様の概要を以下に示す。

- 扱えるデータは、定数およびスカラ/配列変数である。
- 扱えるデータ型は、整数型および浮動小数点型である。
- 変数には、入れ子型静的スコープ規則が適用される。
- 配列変数のデータ分割法は、その宣言時に静的に指定する。
- 実行文として以下のものをもつ。
  - 代入文
  - `if` ∼ `then` ∼ `else` ∼ `endif` 構文
  - `for` ∼ `endfor` 構文
  - `while` ∼ `endwhile` 構文
- 多数の算術関数が組み込み関数として扱われる。

ここに示されているように拡張 Tiny language の言語仕様は最低限かつ必要十分なものであり、コンパイラによる解析にとって有利となるように配慮されている。

例えば拡張 Tiny language の`for` ループは、ループの実行開始時にループ回数が確定しなければならない。すなわちループ内からの飛び出しや、ループ中でのループ回数の変更を認めない。このような制約は、従来よりベクトル化コンパイラによる最適化に対して設けられており、TINPAR による並列化においても最適化を容易にする。なおこれらの制約によって`for` 文で実現不可能であるループ、例えば不定回数ループなどは、`while` 文を用いて記述できる。

またソースプログラム中にユーザ定義関数/手続きの呼び出しが存在すると、コンパイラによる解析は一般に複雑なものとなり、最適化が困難となる。例えばこのような関数の引数中に配列変数が含まれている場合には、その配列変数のデータ分散の状態は呼出し時の実引数に依存し、一意に求まるとは限らない。このためデータの分散状態に強く依存するような最適化は、このような場合に適用が困難となる。このような場合に対する従来の手法としては、関数/手続き間に跨るデータフロー解析によって実引数となりうる変数の組をコンパイル時に求め、適切に最適化された関数/手続きを複数用意する手法がある [11]。しかしこのような手法では、非常に複雑な解析が必要とされる。

そこで拡張 Tiny language ではユーザ定義による関数/手続きの呼び出し機構を設けず、このような機能は関数/手続きのインライン展開によって実現する。こうすることによって、コンパイラによる解析の負担を大幅に軽減することができる。なお関数/手続きのインライン展開を行うためには変数に対する入れ子型の静的スコープ規則が必要となるため、拡張 Tiny language はこれを採用している。またこのようなインライン展開によるオブジェクトコード量の増加は、ソースプログラム中で性能に与える影響の大きい部分を対象とする場合には、問題にならない。ただし関数/手続きの再帰呼び出しを含むプログラムは、並列化の対象外とする。

2.2.2　並列化に関係する機能

Tiny language に対して我々が行った最も重要な言語拡張として、並列化に関係する次のようなディレクティブの追加がある。

- processors ディレクティブ
  並列度、すなわちオブジェクトコードの実行に用いるプロセッサ数を指定する。
- データ分割ディレクティブ
  各プロセッサに対する配列変数の分散法を指定する。

TINPAR は owner computes rule に従って並列化を行うため、これらのディレクティブを用いるだけでソースプログラムの並列化戦略を指示することができる。この

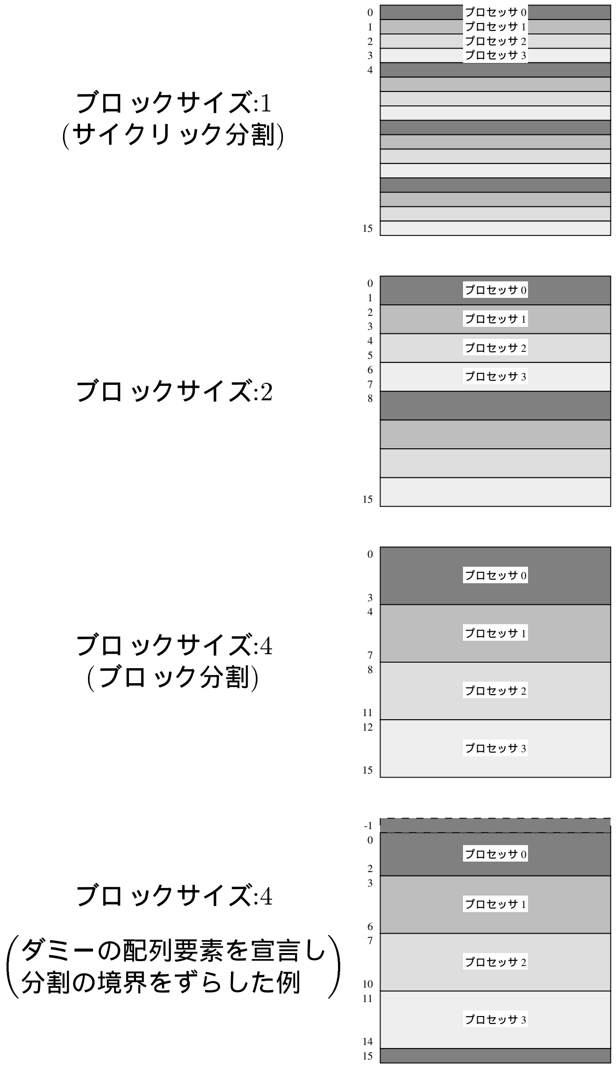

図 1: TINPARで扱えるデータ分割法
(要素数:16、プロセッサ数:4)

ときこれらのディレクティブが指定されたソースプログラムとオブジェクトコードの振舞いとの間には次のような関係が保証される。

- データ分割が施されている配列変数の各要素に対する代入文は、その要素の所有者プロセッサだけで実行される。
- 並列化されたオブジェクトコードの実行結果は、ソースプログラムを逐次プログラムとして実行して得られる結果と等しい。

このため適切にデータを分散することによって、効率のよいオブジェクトコードを生成させることが可能である。

現在TINPARで扱えるデータ分割法[1]は、単一次元方向に対する、任意ブロックサイズのブロックサイクリック分割である(図1参照)。ブロックサイクリック分割は、ブロックサイズを1とした場合にはサイクリック分割に相当し、「分割された次元の配列要素数/プロセッサ数」とした場合にはブロック分割に相当するため、これらを一般化したものであるといえる。またダミーの配列要素を宣言することにより、分割の境界を前後にずらすことも可能である。分割の方向が単一次元に限られている点を除けば、これらはHPFで規定されているデータ分割法を実現するのに十分である。なおスカラ変数に対しては、各プロセッサがそのコピーを持ち、それらすべてが所有者プロセッサであるとみなされる。

一方データの再分散機能は、拡張Tiny languageでは提供されない。データの再分散とは配列変数のデータ分散の状態を動的に変更することであるが、前項の関数/手続き呼び出しの問題と同様に、これもコンパイラによる最適化にとって障害となる。データの再分散機能の利点はプログラマの負担の軽減、およびデータの格納領域の効率化であるが、これらについてはプログラム変換、および動的メモリ管理のための言語拡張によって達成可能である。

2.3 オブジェクトコードの形式

TINPARのオブジェクトコードはC言語によるSPMDプログラムである。

SPMDプログラムとは、並列計算機のためのプログラム形式の1つであり、すべてのプロセッサ上で同一のプログラムが実行されることを前提に書かれたものである。このようなプログラムでは、実行するプロセッサのプロセッサ番号によってその振舞いが変化する。TINPARのように数値処理プログラムをowner computes ruleに従って並列化する場合、すなわちデータ並列性を抽出する場合には、各プロセッサの処理内容に本質的な差が生じることは少ないため、SPMD形式のオブジェクトコードを生成するのが妥当である。

一方C言語によるオブジェクトコードを生成することには、次のような利点がある。

- 対象並列計算機のプロセッサに依存した最適化をCコンパイラに任せることができる。

  対象並列計算機のプロセッサアーキテクチャに強く依存するような低レベルの最適化、例えば機械命令のリスケジューリングやレジスタアロケーションなどは、すべてCコンパイラに任せることができる[17]。このためTINPARは、通信のリスケジューリングや不用な通信の削除などの、並列化に密接した、より上位の最適化に専念することができる。

- 対象並列計算機の変更が比較的容易である。

  C言語によるプログラムは一般に移植性が高い。またTINPARでは専用の通信ライブラリの仕様を定めており、オブジェクトコード中の通信処理はこれによって実現される。このため対象並列計算機を変更する場合にも、専用通信ライブラリを用意するだけで対応できる。

なおオブジェクトコードにリンクされてすべての通信処理を行うTINPARの専用通信ライブラリについては、5章で詳しく述べる。

2.4 内部構成

TINPARはその設計方針にもとづき、以下のように独立した6つのモジュールから構成される。

---

[1] 正確にはデータ分割およびデータ分散の両方を合わせたものであるが、本稿では慣用に従ってデータ分割法と呼ぶ。

```
processors n_cell
real sum
real a(0:n - 1:*)
/*  a() はブロック分割 */

sum = 0.0
for i = 0, n - 1 do
  sum = sum + a(i)
endfor
```

⇓

```
processors n_cell
real sum
real a(0:n - 1:*)
real local_sum(0:n_cell - 1:1)
/*  local_sum() はサイクリック分割 */

/*  部分和を格納する配列local_sum() を初期化する。 */
for cell_id = 0, n_cell - 1 do
  local_sum(cell_id) = 0.0
endfor
/*  部分和を求める。 */
for i = 0, n - 1 do
  local_sum(OWNER(a(i))) \
    = local_sum(OWNER(a(i))) + a(i)
endfor
sum = 0.0
/*  総和を求める。 */
for cell_id = 0, n_cell - 1 do
  sum = sum + local_sum(cell_id)
endfor
```

図 2: 前処理 1: 総和演算のイディオム変換

```
real a(0:n - 1:*)
real b(0:n - 1:*)
real x(0:n - 1, 0:n - 1:*)
/*  a()、b() は1次元め、c() は2次元めでブロック分割 */

for i = 0, n - 1 do
  for j = 0, n - 1 do
    x(i, j) = a(i) * b(j)
  endfor
endfor
```

⇓

```
real a(0:n - 1:*)
real b(0:n - 1:*)
real x(0:n - 1, 0:n - 1:*)
real local_a

for i = 0, n - 1 do
  /*  変数local_a にローカルコピーを作成する。 */
  local_a = a(i)
  /*  ローカルコピーを用いて計算を行う。 */
  for j = 0, n - 1 do
    x(i, j) = local_a * b(j)
  endfor
endfor
```

図 3: 前処理 2: リモートデータに対するローカルコピーの作成/利用

```
real a(0:n - 1:*)
real b(0:n - 1:1)
/*  a() はブロック分割、b() はサイクリック分割 */

for i = 0, n - 1 do
  b(i) = a(i)
endfor
```

⇓

```
real a(0:n - 1:*)
real b(0:n - 1:1)
real tmp_a

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

図 4: 並列化

1. 構文解析部

   ソースプログラムを構文解析し、処理系の内部表現に展開する。

2. 前処理部

   先に述べたように、逐次プログラムにはそのままでは並列化に適さない部分が存在することがある。そこで本モジュールでは、このような部分に対してプログラム変換を行って、並列化に適した形に変形する。この変換の例としては、総和演算のイディオム変換、リモートデータに対する一時的なローカルコピーの作成/利用などがあげられる (図2、3参照)。

3. 並列化部

   本モジュールでは、owner computes rule に従ってソースプログラムを並列化する。具体的には、所有者プロセッサを判定するための`if`文、およびリモートデータアクセスのための`SEND`/`RECV`文が挿入される (図4参照)。

   なおこの時点では、データ分割が施された配列に対しても配列全体を格納できる大きさのメモリ領域を各プロセッサで確保する。この結果、このような配列に対するアクセスは分割前と同様のアドレッシングで可能となるが、実際に使用されるメモリ領域は各プロセッサに割り付けられたデータを格納する部分のみとなる。

   これらの具体的な処理内容については3章で述べる。

4. 最適化部

   並列化部で用いている並列化手法は極めて単純であり、コード的に無駄が多く効率もよくない。そこで本モジュールでは、不用な`if`文の削除や`SEND`文のリスケジューリングなどの最適化を行う。なおこれらの最適化は、それぞれ独立したサブモジュールによって個別に実現される。

   これらの具体的な処理内容については4章で述べる。

5. メモリ再割り当て部

   データ分割が施された配列に対する並列化部のメモリ割り当て手法は、最適化部によるデータ依存関係解析を容易にするが、確保されるものの全く使用されないメモリ領域が存在することになって無駄も多い。そこで本モジュールでは、このような配列に対するメモリ割り当て手法の改善およびそれに伴うアドレッシングの修正を行う (図5参照)。

   これらの具体的な処理内容については3章で述べる。

```
processros n_cell
real a(0:n - 1:*)

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
```

⇓

```
processros n_cell
real a(0:⌈n / n_cell⌉ - 1)

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(LOCAL_INDEX(a(i))) = ...
  endif
endfor
```

図 5: メモリの再割り当て

6. コード生成部

   処理系の内部表現から C 言語によるオブジェクトコードを生成する。

このような構成をとることによって TINPAR 内部の処理の独立性が高まり、1) 実装が容易になる、2) 用いた手法の評価が行える、3) 一般的かつ包括的な手法の採用および評価ができる、などの利点が生じる。

3 並列化手法

強力な最適化手法の適用を前提に、TINPAR では owner computes rule にもとづく単純な並列化手法を採用している。本章では、並列化部およびメモリ再割り当て部で用いられる並列化/メモリ割り当ての手法について述べる。

3.1 データの分散

owner computes rule に従って逐次プログラムを並列化する場合には、各プロセッサに対してデータが分散されていることが前提となる。現在 TINPAR では、単一次元方向に対する、任意ブロックサイズのブロックサイクリック分割を扱うことができる。

TINPAR による並列化では、データ分割が施された配列に対するメモリ割り当てを 2 段階に分けて行う (図 6、7参照)。すなわち分割された次元の i_k番めの配列要素は、並列化部によって一旦 OWNER(i_k) 番プロセッサの i_k番要素に割り当てられ、その後メモリ再割り当て部によって LOCAL_INDEX(i_k) 番要素に割り当て直される。ここで当該配列は k次元めにブロックサイクリック分割が施されており、当該配列要素の所有者プロセッサ番号、および所有者プロセッサ上でのローカルな配列要素番号は、以下の関数 OWNER()、LOCAL_INDEX() で与えられる。

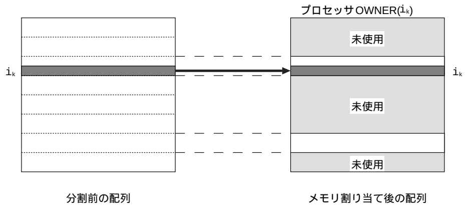

図 6: 並列化部によるメモリ割り当て

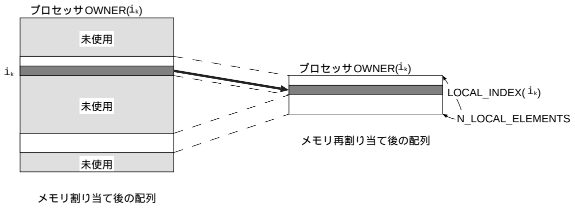

図 7: メモリ再割り当て部によるメモリ割り当て

$$
\begin{aligned}
&\mathrm{OWNER}(\text{分割された次元の添字の値 } i_k) \\
&\quad = \lfloor (\text{分割された次元の添字の値 } i_k \\
&\qquad\quad - \text{分割開始位置のオフセット値}) \\
&\qquad\quad \div \text{ブロックサイズ} \rfloor \\
&\qquad\quad \bmod \text{プロセッサ数}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{LOCAL\_INDEX}(\text{分割された次元の添字の値 } i_k) \\
&\quad = \lfloor (\text{分割された次元の添字の値 } i_k \\
&\qquad\quad - \text{分割開始位置のオフセット値}) \\
&\qquad\quad \div (\text{ブロックサイズ} \times \text{プロセッサ数}) \rfloor \\
&\qquad\quad \times \text{ブロックサイズ} \\
&\qquad\quad + (\text{分割された次元の添字の値 } i_k \\
&\qquad\qquad - \text{分割開始位置のオフセット値}) \\
&\qquad\qquad \bmod \text{ブロックサイズ}
\end{aligned}
$$

一方データ分割が施された k次元めに関して、各プロセッサが所有する配列要素数 N_LOCAL_ELEMENTS は次の式で与えられる。

$$
\begin{aligned}
&\mathrm{N\_LOCAL\_ELEMENTS} \\
&\quad = \lceil \text{分割された次元の配列要素数} \div \text{プロセッサ数} \rceil
\end{aligned}
$$

ただし分割開始位置のオフセット値は分割された次元の先頭配列要素番号と等価である。また $0 \leq$ プロセッサ番号 $<$ プロセッサ数とする。

このようにメモリ割り当てを 2 段階で行うことにより、並列化部の生成するコードは、1) 最適化部によるデータ依存関係解析が容易である、2) メモリの利用効率が悪い、という性格のものとなる。

なお具体的な処理としては、並列化部ではデータ分割が施された配列の要素に対するアドレッシングを実質上修正する必要がなく、この処理はすべてメモリ再割り当て部で行われる。

3.2 演算の分割とプロセッサへの割り付け

データの分散が行われると、owner computes rule に従ってソースプログラム中の演算を分割し、各プロセッサに割り付けることが可能となる。これらの処理は各文を単位として行うことができる。具体的な処理内容は以下のとおりである (図 8参照)。

```
real a(0:n - 1:*)
/*  a() はブロック分割 */

for i = 0, n - 1 do
  a(i) = 0.0
endfor
```

⇓

```
real a(0:n - 1:*)

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = 0.0
  endif
endfor
```

図 8: 演算の分割とプロセッサへの割り付け

- 変数宣言文

  並列化部では、データ分割が施された配列の要素に対するアドレッシングを変更しないため、配列の大きさの宣言自体も変更しない。この結果メモリ再割り当て部では、そのような配列の分割された次元の要素数を N_LOCAL_ELEMENTS に縮小する必要がある。

- 代入文

  左辺値すなわち代入先の変数が、データ分割が施された配列の要素である場合には、その代入文は当該データの所有者プロセッサのみによって実行されなければならない。そこでこのような代入文には、次のような条件式を持つ`if/endif` 文を前後に挿入する。

  ```
  OWNER(左辺値) == GET_CELL_ID()
  ```

  ここで`ONWER()` 関数は指定されたデータの所有者プロセッサ番号を返し、`GET_CELL_ID()` 関数は自分のプロセッサ番号を返す。

  なお右辺にデータ分割を施された配列の要素が現れる場合にはメッセージ通信のためのコードを挿入する必要があるが、これについては次節で述べる。

- `for`、`if`、および`while` 構文

  全プロセッサで実行されるため、変更を受けない。

3.3 通信コードの挿入

TINPAR による並列化では、リモートデータアクセス、すなわち他のプロセッサに割り付けられているデータの参照は、メッセージ通信によって実現される。このためデータ分割を施された配列の要素を代入文の右辺で参照している場合には、以下のような処理が行われる (図9参照)。

1. 新たなスカラ変数を宣言する。

   参照者プロセッサによって受信される当該データの値は、一時的にこのスカラ変数に格納される。

```
processors n_cell
real a(0:n - 1:*)
real b(0:n - 1)
/*  a() はブロック分割、b() は分割なし */

for i = 0, n - 1 do
  b(i) = a(i)
endfor
```

⇓

```
processors n_cell
real a(0:n - 1:*)
real b(0:n - 1)
real tmp_a

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    for cell_id = 0, プロセッサ数 - 1 do
      SEND(cell_id, a(i))
    enddo
  endif
  RECV(OWNER(a(i)), tmp_a)
  b(i) = tmp_a
endfor
```

図 9: 通信コードの挿入

2. `SEND` 文および`if/endif` 文を挿入する。

   当該データを送信するための`SEND` 文、およびその所有者プロセッサを判定するための`if/endif` 文を当該代入文の前に挿入する。

   - 代入文が特定のプロセッサで実行される場合

     ```
     if OWNER(当該データ) == GET_CELL_ID() then
       SEND(OWNER(左辺値), 当該データ)
     endif
     ```

   - 代入文が全プロセッサで実行される場合

     ```
     if OWNER(当該データ) == GET_CELL_ID() then
       for cell_id = 0, プロセッサ数 - 1 do
         SEND(cell_id, 当該データ)
       endfor
     endif
     ```

3. `RECV` 文を挿入する。

   当該データを受信するために、当該代入文の直前に次のような`RECV` 文を挿入する。

   ```
   RECV(OWNER(当該データ), 1. で宣言したスカラ変数)
   ```

   なお当該代入文が特定のプロセッサで実行される場合に挿入される、左辺値の所有者プロセッサを判定するための`if` 文は、この`RECV` 文の直前に挿入される (3.2節参照)。

4. 当該代入文の右辺中の参照を置き換える。

   当該データの参照を 1. で宣言したスカラ変数への参照に置き換える。

なおデータ分割が施されている配列の要素の参照は、あらかじめプログラム変換を施すことにより、代入文の右辺にのみ現れるものとする。

4 最適化手法

TINPAR は C 言語によるオブジェクトコードを生成する。このため対象並列計算機に依存した低レベルの最適化は C コンパイラに任せることが可能である。そこで TINPAR の最適化部では、並列化に密接した最適化のみを行う。本章では、TINPAR で用いられている種々の最適化手法、およびそれらの適用順序について述べる。

4.1 ランタイムオーバーヘッドの削減

TINPAR の並列化戦略は、owner computes rule に従った単純なものである。このため並列化部の生成するコードには、データ分割が施された配列の要素の所有者プロセッサを判定するための`if` 文のような、ランタイムオーバーヘッドを増加させる文が多数挿入されている。本節では、これらのうちの不必要なものを削除し、ランタイムオーバーヘッドを削減するための最適化手法について述べる。

4.1.1 ループの実行範囲の縮小

並列化部の生成するコードには図 10上のような`for` ループが多数現れる。このようなループでは、`if` 文の条件式が成立しない場合に無意味なイタレーションを実行することになる。そのうえ多くの場合には、この条件式の成立する確率は用いるプロセッサ数に反比例して低くなる。

そこで本最適化では、そのような条件式が成立する範囲にループの実行範囲を狭めることによって、空回りとなるイタレーションを削除する。具体的なアルゴリズムは以下のとおりである。なお対象ループの増分値は 1 とする。

1. ループ内の各文が実行されるべきループインデックス値の条件を求める。

   ループ内の各文について、その文を囲む`if` 文の条件式を解析し、その文が実行されるべきループインデックス値の条件を求める。具体的には次のような等式または不等式を対象として、解となるループインデックス値の条件を式変形によって求める。

   ```
   OWNER(x(..., index * a + b, ...)) \
             {==|!=} GET_CELL_ID()
   ```

   ただし配列`x()` は単一次元方向にブロックサイクリック分割されており、当該ループのループインデックスを index とすると、その次元の添字が $\mathit{index}\times a+b$ で表されるものとする。また、a、b の値および配列`x()` の分割はループ内で不変であり、$a\neq 0$ であるとする。

   以下では等式を解く場合について述べる。配列`x()` に対する分割のパラメータを、プロセッサ数 NCell、ブロックサイズ $\mathit{BlockSize}$、分割開始位置のオフセット値 Offset で表し、`GET_CELL_ID()` 関数の返り値を CellID で表すと、次の等式が得られる。

   $$\lfloor\frac{(\mathit{index}\times a+b)-\mathit{Offset}}{\mathit{BlockSize}}\rfloor \bmod \mathit{NCell}=\mathit{CellID}$$

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor

                    ⇓

for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
```

図 10: ループの実行範囲の縮小

   このとき $0\leq \mathit{CellID}<\mathit{NCell}$ であるから、整数 n に対して、

   $$\lfloor\frac{(\mathit{index}\times a+b)-\mathit{Offset}}{\mathit{BlockSize}}\rfloor=\mathit{CellID}+\mathit{NCell}\times n$$

   が成り立ち、実数$\alpha(0\leq\alpha<\mathit{BlockSize})$ に対して、

   $$\begin{aligned}&(\mathit{index}\times a+b)-\mathit{Offset}\\ &\quad=\ (\mathit{CellID}+\mathit{NCell}\times n)\times\mathit{BlockSize}+\alpha \qquad (1)\end{aligned}$$

   が成り立つ。この結果、当該ループのループインデックスに関する次の等式が得られる。

   $$\begin{aligned}&\mathit{index}\\ &\quad=\ \frac{(\mathit{CellID}+\mathit{NCell}\times n)\times\mathit{BlockSize}}{a}\\ &\qquad+\frac{\alpha+\mathit{Offset}-b}{a}\end{aligned}$$

   さらに、

   $$\begin{aligned}\mathit{Stride}&=\mathit{NCell}\times\mathit{BlockSize}\\ \mathit{Offset1}&=(\mathit{Offset}-b)\bmod\mathit{Stride}\\ \mathit{Offset2}&=\mathit{Offset1}+\mathit{BlockSize}\end{aligned}$$

   とすると、この等式を満たす index は、

   $$\mathit{index}=\frac{\mathit{BlockSize}\times\mathit{CellID}+\mathit{Stride}\times n+\alpha'}{a}$$

   $$0\leq\alpha'<\mathit{Offset2}-\mathit{Stride}$$
   $$\text{または }\mathit{Offset1}\leq\alpha'<\min(\mathit{Stride},\mathit{Offset2})$$

   で与えられる。

   ここで $0\leq\alpha'<\mathit{Stride}$ であるから、n とα'の値の組は index の値と 1 対 1 対応する。従って、Stride、$\mathit{BlockSize}$、$\mathit{Offset1}$、$\mathit{Offset2}$ および a を用いて、各文が実行されるべきループインデックス値の条件を一意に表すことが可能である。

   一方不等式の場合には、式 (1) で $\mathit{BlockSize}\leq\alpha<\mathit{Stride}$ であるから、

   $$\begin{aligned}\mathit{Offset1}&=(\mathit{Offset}-b+\mathit{BlockSize})\bmod\mathit{Stride}\\ \mathit{Offset2}&=\mathit{Offset1}+\mathit{Stride}-\mathit{BlockSize}\end{aligned}$$

   とすることによって同様に表すことができる。

2. 求めたループインデックス値の条件の和集合を求める。

   ループ内のすべての文について、Stride、$\mathit{BlockSize}$ および a の値が共通であれば、ループインデックス値の条件の和集合を求めることができ、本最適化が適用可能である。適用できる場合には、各々の $\mathit{Offset1}$ と $\mathit{Offset2}$ の組から定まるα'の範囲の和集合を求め、$\mathit{Offset1}' \leq \alpha' < \mathit{Offset2}'$となる$\mathit{Offset1}'$と $\mathit{Offset2}'$の組の昇順リストで表す。

3. 当該ループをストリップマイニングする。

   求めた index の条件は、

$$\begin{aligned} \mathit{index} &= \frac{\mathit{index}' + \alpha'}{a} \\ \mathit{index}' &= \mathit{BlockSize} \times \mathit{CellID} + \mathit{Stride} \times n \end{aligned}$$

   で与えられる。そこで当該ループをストリップマイニングし、外側ループのループインデックスでindex'を与える。このとき外側ループの初期値、終了値および増分値は以下のようになる。

$$\begin{aligned} \text{初期値} &= \mathit{BlockSize} \times \mathit{CellID} \\ &\quad + \mathit{Stride} \times \lfloor \frac{\text{もとの初期値} \times a}{\mathit{Stride}} - 1 \rfloor \\ \text{終了値} &= \text{もとの終了値} \times a \\ \text{増分値} &= \mathit{Stride} \end{aligned}$$

4. 内側ループの実行範囲を縮小する。

   求めた昇順リストの要素数だけ内側ループをコピーする。このとき各々の初期値、終了値および増分値は、リストの順に $\mathit{Offset1}'$と $\mathit{Offset2}'$を用いて以下のようになる。

$$\begin{aligned} \text{初期値} &= \max(\text{もとの初期値}, \\ &\qquad \lceil \frac{\mathit{index}' + \mathit{Offset1}'}{a} \rceil) \\ \text{終了値} &= \min(\text{もとの終了値}, \\ &\qquad \lceil \frac{\mathit{index}' + \mathit{Offset2}'}{a} \rceil - 1) \\ \text{増分値} &= 1 \end{aligned}$$

これにより、空回りとなる不用なイタレーションは実行されなくなる。

4.1.2 恒真/恒偽となる`if` 文の削除

並列化部の生成するコードに含まれる、データ分割が施された配列の要素の所有者プロセッサを判定するための`if` 文の条件式には、除算や剰余算といった重い演算が含まれる。一方ループの実行範囲の縮小を行った結果、そのような条件式が恒真または恒偽となる場合がある (図 11上参照)。

そこで本最適化では、コンパイル時に`if` 文の条件式を静的評価することによって、このような不必要な`if` 文を削除する。具体的なアルゴリズムは以下のとおりである。

1. `if` 文の条件式を成立させるループインデックス値の条件を求める。

   前項と同様にしてループインデックス値の条件を求める。

```
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
         0 ≤ i ≤ n - 1} do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
```

$$\Downarrow$$

```
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
         0 ≤ i ≤ n - 1} do
  a(i) = ...
endfor
```

図 11: 恒真/恒偽となる`if` 文の削除

2. ループインデックスのとりうる値の範囲を求める。

   ループインデックスのとりうる値の範囲を対応する`for` 文より求める。

3. 後者と前者の積および差集合を求める。

   両者について、Stride、$\mathit{BlockSize}$ および a の値が共通であれば、これらの積および差集合を求めることができ、本最適化が適用可能である。ただし A、Bの差集合とは、集合 A と Bに対し、A に属し Bに属さない要素全体からなる集合を指す。

4. 積集合が空であれば、`else` 節を残して`if` 文を削除する。

   積集合が空であれば当該条件式は恒偽である。

5. 差集合が空であれば、`then` 節を残して`if` 文を削除する。

   差集合が空であれば当該条件式は恒真である。

これにより、実行時の不必要な条件判定を削除することができる。

4.1.3 `if` 文を削除するためのループ分割

恒真/恒偽となる`if` 文の削除は、図 12上のような場合には当然適用できない。しかしこのような場合にも、ループインデックスのとりうる値の範囲を分割することによって、`if` 文の条件式を恒真または恒偽とすることができる。

そこで本最適化では、`if` 文の条件式が恒真または恒偽となるようにループ分割を施すことによって、`if` 文の削除の最適化の適用を可能にする。具体的なアルゴリズムは以下のとおりである。

1. ループ内の各文が実行されるべきループインデックス値の条件を求める。

   前々項と同様にしてループインデックス値の条件を求める。

2. 求めたループインデックス値の条件を昇順にソートする。

   ループ内のすべての文について、Stride、$\mathit{BlockSize}$ および a の値が共通であれば、ループインデックス値の条件をソートすることができ、本最適化が適用可能である。適用できる場合には、各々の $\mathit{Offset1}$ と $\mathit{Offset2}$ の組から定まるα'の範囲を昇順にソート

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
  b(i) = ...
endfor
                                ⇓
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  a(i) = ...
  b(i) = ...
endfor
for i ∈ {i|OWNER(a(i)) != GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  b(i) = ...
endfor
```

図 12: `if` 文を削除するためのループ分割

$(\mathit{Offset1}'', \mathit{Offset2}'')$ のリストが、

- $\{(0,2),(0,4),(0,8)\}$ の場合
  2 を境界としてループ分割する。
- $\{(0,2),(0,4),(4,8)\}$ の場合
  2 を境界としてループ分割する。
- $\{(0,2),(0,4),(1,8)\}$ の場合
  1 を境界としてループ分割する。

図 13: `if` 文を削除するためのループ分割の例

し、$\mathit{Offset1}'' \leq \alpha' < \mathit{Offset2}''$となる $\mathit{Offset1}''$と$\mathit{Offset2}''$の組の昇順リストで表す。

3. 当該ループを分割する。

   求めた昇順リストの要素が複数あれば、当該ループを 2 つのループに分割する。このとき分割の境界となるループインデックス値は、リスト中の $\mathit{Offset1}''$の値がすべて等しい場合には 1 番小さい $\mathit{Offset2}''$、それ以外の場合には 2 番めに小さい $\mathit{Offset1}''$と 1 番小さい $\mathit{Offset2}''$のうちで小さい方とする (図 13 参照)。

4. 分割されてできた後半のループに対し、以上の手順を繰り返す。

これにより、並列化部で挿入されたほとんどの`if` 文を削除することができる。

4.1.4 不用な`SEND`/`RECV` 文の削除

メッセージ交換型並列計算機では通信処理にかかるコストが一般に大きいため、不用な通信は可能な限り削除しなければならない。例えば図 14上[2]のような場合には、もし変数`tmp_a` の値が前半のループ以外で参照されなければ、`BROADCAST`/`RECV` 文は不用な通信を行っていることになる。

そこで本最適化では、データフロー解析を行うことによって、不用な通信コードを削除する。具体的には、不用な代入操作となる`RECV` 文をデータフロー解析によって検出し、対応する`SEND`/`BROADCAST` 文とともに削除する。これにより、不用な通信を削除することができる。なお具体的なアルゴリズムについては、文献 [4] に述べられている。

```
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  BROADCAST(a(i))
  tmp_a = a(i)
  if tmp_a < 0 then
    a(i) = - tmp_a
  endif
endfor
for i ∈ {i|OWNER(a(i)) != GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  RECV(OWNER(a(i)), tmp_a)
endfor
                                ⇓
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), \
          0 ≤ i ≤ n - 1} do
  tmp_a = a(i)
  if tmp_a < 0 then
    a(i) = - tmp_a
  endif
endfor
```

図 14: 不用な`SEND`/`RECV` 文の削除

4.2 通信レイテンシの低減

通信レイテンシは、演算に必要なデータを他のプロセッサから受信するための待ちによって生じる。本節では、そのようなレイテンシを低減し、プロセッサの利用率を高めるための最適化手法について述べる。

4.2.1 `SEND` 文のリスケジューリング

メッセージ交換型並列計算機では、送信側プロセッサから送られたデータは、実際に使用されるまで受信側でバッファリングされる。このため送信側プロセッサは、使用されるよりも早い時期にあらかじめデータを送っておくことができる。

そこで本最適化では、`SEND` 文をリスケジューリングすることによって、データ送信を可能な限り先行させる。具体的には、`SEND` 文、および`SEND` 文を含むが`RECV` 文を含まない`if`/`for`/`while` ブロックを、プログラムの上流方向に移動する (図 15参照)。これにより、通信レイテンシをある程度抑えることができる。

4.2.2 `SEND`/`RECV` 文をともに含むループのループ分配

`SEND` 文のリスケジューリングは、制御およびデータ依存関係による制約を受けるため、通信レイテンシを完全に隠蔽するには十分でない。例えば図 16上のような場合には、データの送信から受信までの時間的距離は非常に短く、通信レイテンシが生じると予想される。しかしこのコードに図 16下のようなループ分配を行うと、送信から受信までの時間的距離は前半ループの実行時間程度となって十分長く、レイテンシはほとんど隠蔽される。

---

[2] 配列`a()` の各要素に対してその絶対値を求めるプログラムを並列化したもの。

```
for i = 0, n - 1 do
  a(i) = ...
  if OWNER(b(i)) == GET_CELL_ID() then
    SEND(OWNER(c(i)), b(i))
  endif
  if OWNER(c(i)) == GET_CELL_ID() then
    RECV(OWNER(b(i)), tmp_b)
    c(i) = tmp_b
  endif
endfor
```

⇓

```
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    SEND(OWNER(c(i)), b(i))
  endif
  a(i) = ...
  if OWNER(c(i)) == GET_CELL_ID() then
    RECV(OWNER(b(i)), tmp_b)
    c(i) = tmp_b
  endif
endfor
```

図 15: SEND文のリスケジューリング

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

⇓

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
endfor
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

図 16: SEND/RECV文をともに含むループのループ分配

そこで本最適化では、SENDおよびRECV文をともに含むループをSEND文のみを含むループとRECV文のみを含むループに分配することによって、データ送受信間の時間的距離を拡大する。これにより、通信レイテンシを大幅に削減することができる。

4.2.3 MSG_BUFFER_FLUSH文の挿入

オブジェクトコードにリンクされる専用通信ライブラリは、動的ベクトル化機能を持つ。この機能は、通信ランタイムシステムが送信データをバッファリングし、長いメッセージとして送信するものである。これにより、オブジェクトコードの通信処理は柔軟かつ効率よく実現される (詳細は5章参照)。しかしこのようなバッファリングを行う場合には、そのバッファを適切にフラッシュしなければ、その利点を十分に生かせないことがある。

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
endfor
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

⇓

```
for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
endfor
ALL_MSG_BUFFER_FLUSH()
for i = 0, n - 1 do
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

図 17: MSG_BUFFER_FLUSH文の挿入

そこで本最適化では、連続するSEND文の直後にMSG_BUFFER_FLUSH文を挿入することによって、通信ランタイムシステムの送信バッファを適切にフラッシュさせる (図17参照)。これにより、送信バッファに未送信のデータが残ることによる性能低下を防ぐことができる。

4.3 最適化の適用順序

一般に最適化の適用順序はその効果を左右する。TINPARでは、本章で述べた最適化を図18の順序で適用する。この適用順序は、以下の点を考慮することによって定められている。

- ループの実行範囲の縮小

  ループボディ内の文の数が少ないほど適用しやすいため、ループ分配を行った後の方が適用できる可能性が大きい。

- 恒真/恒偽となるif文の削除

  これに先だってループの実行範囲の縮小やif文を削除するためのループ分割を行わなければならない。

- 不用なSEND/RECV文の削除

  データフロー解析を必要とするため、if文の削除によって制御フローが単純になった後の方が適用できる可能性が大きい。

- SEND文のリスケジューリング

1. `SEND` 文のリスケジューリング
2. `SEND/RECV` 文をともに含むループのループ分配
3. ループの実行範囲の縮小
4. 恒真/恒偽となる`if` 文の削除
5. `if` 文を削除するためのループ分割
6. 恒真/恒偽となる`if` 文の削除
7. 不用な`SEND/RECV` 文の削除
8. `SEND` 文のリスケジューリング
9. `MSG_BUFFER_FLUSH` 文の挿入

図 18: 最適化の適用順序

ループ分配や`if` 文の削除によって制御/データ依存関係による制約が弱まった後に行うと、効果が大きくなる場合がある。

- `SEND/RECV` 文をともに含むループのループ分配

  `SEND` 文のみを含むループと`RECV` 文のみを含むループに分配するため、`SEND` 文をリスケジューリングした後の方が適用できる可能性が大きい。

- `MSG_BUFFER_FLUSH` 文の挿入

  連続した`SEND` 文の直後に`MSG_BUFFER_FLUSH` 文を挿入するため、`SEND` 文のリスケジューリングや`SEND/RECV` 文をともに含むループのループ分配の後で行う方が無駄な挿入を避けられる。

5 専用通信ライブラリ

オブジェクトコード中の通信処理は、専用の通信ライブラリによって効率よく実現される。本章では、この専用通信ライブラリの特徴である動的ベクトル化機能、この機能をベースとした効率のよい通信処理の実現法、および高並列計算機 AP1000 に対する実装の概要について述べる。

5.1 動的ベクトル化機能

メッセージ交換型並列計算機では通信処理の起動コストが一般に大きいため、その起動回数を減らすことが重要となる。このため、メッセージのベクトル化と呼ばれる最適化手法が従来より用いられている [10]。これは送信データをまとめて 1 つの長いメッセージとして送信することにより、通信処理の起動回数を減らす手法である。

メッセージのベクトル化を行う場合、もし送信データが連続領域にあるか、または一定間隔にあるならば、専用ハードウェアによるデータ領域からの直接送信が可能となり、高速な処理が可能である [22]。しかし、送信データの配置が必ずしもこのような条件を満たすとは限らず、満たさない場合には送信データをパッキングする処理が必要となる。さらに、このようなパッキングを行うためには一時作業領域が必要となり、そのためのメモリ管理についても考慮する必要がある。

そこで TINPAR では、専用通信ライブラリによって提供される通信ランタイムシステムによってメッセージのベクトル化を動的に行う。すなわち、TINPAR のオブジェクトコード中では 1 つの`SEND/RECV` 文が 1 データのパッキング/アンパッキングに対応し、通信ランタイムシステムがそれを適切な長さのメッセージとして送受信する。これにより、コンパイル時にはメッセージのベクトル化やそのためのメモリ管理について配慮する必要がないにも関わらず、効率のよい通信処理を実現することができる。さらに、専用ハードウェアによる直接送信が可能である場合を検出して適切なコードを生成することにより、双方の長所を両立させることも可能である。

ただしこのようなバッファリングを行う場合には、そのバッファを適切にフラッシュしなければ、送信バッファに未送信のデータが残ることによって性能が低下することがある。そこで専用通信ライブラリでは、コンパイラによる最適化の手段として、送信バッファの強制フラッシュ機能を提供する (4.2.3項参照)。

5.2 効率のよい通信処理の実現

通信ランタイムシステムによる動的ベクトル化機能をベースとした通信処理は、具体的には以下のように実現できる。

1. 送信データのパッキング

   プロセッサがパッキングを行う。ただし、専用ハードウェアによる直接送信を行う場合には不用である。

2. メッセージの送信

   パッキングされたデータを送信する場合には、パッキングに使用したメモリ領域は送信後に破棄されてもよいため、ノンブロッキング送信が利用可能である。一方専用ハードウェアによる直接送信の場合には、送信完了フラグを用いたノンブロッキング送信が利用可能である。

3. メッセージの受信

   メッセージ到着の割り込み処理によって、通信ランタイムシステム管理下のメモリ領域に直接受信する。

4. 受信データのアンパッキング

   通信ランタイムシステム管理下の受信領域からプロセッサが直接アンパッキングして参照する。ただしオブジェクトコード管理下のメモリ領域にコピーする場合には、専用ハードウェアによってブロック転送を行った方が高速である。

よって理想的な場合、すなわち専用ハードウェアによる直接送信/プロセッサによるアンパッキングの場合には、1) ノンブロッキング送信の起動、2) メッセージ到着割り込みの処理、3) 受信領域の確保、4) 受信処理の起動、5) 参照時のレジスタへのロード、によってリモートデータを参照できる。このような場合には、メッセージ長に比例する処理は、5) を除くとすべて専用ハードウェアで行えるため、通信以外の演算とオーバーラップできる割合が高く効率がよい。

またプロセッサによるパッキング/アンパッキングを行う場合には、それらのコードをインライン展開することによって高速化することが重要となる。

5.3 AP1000 に対する実装

以上で述べた専用通信ライブラリは、高並列計算機 AP1000[21] に対して既に実装されている。本節では、この実装の概要について述べる。

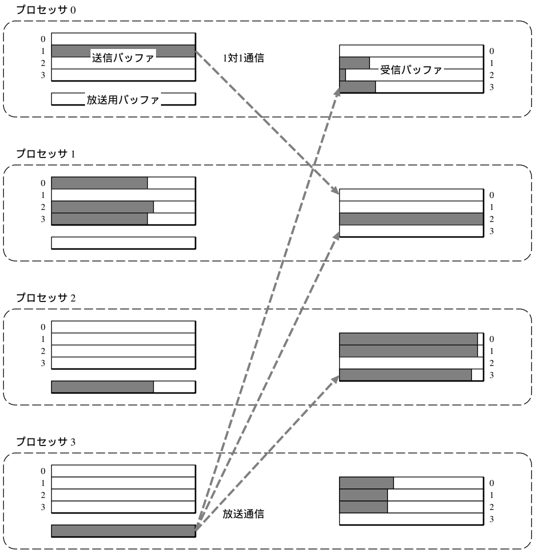

図 19: 送受信バッファの構成

標準通信ライブラリ　AP1000 の標準通信ライブラリでは、高速ネットワークを用いた非同期 1 対 1 通信機能が提供されている。これによって連続領域上のデータに対するブロッキング送受信を行えるため、パッキング/アンパッキングおよび送受信バッファ管理の機能を付加することによって、専用通信ライブラリを実装することができる。

一方放送通信機能としては、1) 高速ネットワークによる同期放送、2) 低速ネットワークによる非同期放送、の 2 つが提供されている、しかし専用通信ライブラリで利用するものとしては、高速かつ非同期のものが望ましい。このため現在のところ、専用通信ライブラリの放送通信機能は高速ネットワークを用いた 1 対 1 通信によってエミュレートしている。

なお標準通信ライブラリでは提供されていないが、1) 専用ハードウェアによる一定間隔データのノンブロッキング送信、2) メッセージ到着の割り込み処理によるユーザ領域への直接受信、がハードウェア的に可能である [22, 23, 24]。

送受信バッファの管理　現在専用通信ライブラリでは、1 対 1 通信および放送通信のための 2 種類の送信機能が提供されている。このために各プロセッサは、プロセッサ数分の送受信バッファおよび 1 本の放送用バッファを持つ (図 19参照)。ここで受信メッセージは、1 対 1 通信および放送通信のどちらによって送信されたかに関わらず、同一の受信バッファに格納される。さらに、動的ベクトル化機能によって 1 つのメッセージは複数のデータから構成されているため、1 対 1 通信と放送通信を併用する場合にはデータの受信順序を保証する工夫が必要である。そこで専用通信ライブラリの送信関数では、送信バッファおよび放送用バッファのどちらかが必ず空となるように管理することによって、データの追い越しを防いでいる (図 20参照)。

また専用通信ライブラリの受信関数では、該当するメッセージが未到着の場合にはすべての送信バッファおよび放送用バッファを強制的にフラッシュしてから待ち状態に入ることにより、これらのバッファリングが原因でデッドロックに陥ることを防いでいる (図 20参照)。

```
送信 () {
    if (送信バッファが空)
        放送用バッファをフラッシュ;
    送信バッファにデータをパッキング;
    if (送信バッファが満杯)
        送信バッファをフラッシュ;
}

放送 () {
    if (放送用バッファが空)
        送信バッファをフラッシュ;
    放送用バッファにデータをパッキング;
    if (放送用バッファが満杯)
        放送用バッファをフラッシュ;
}

受信 () {
    if (受信バッファが空) {
        if (該当するメッセージが未到着)
            送信バッファおよび放送用バッファをフラッシュ;
        メッセージを受信バッファに受信;
    }
    受信バッファからデータをアンパッキング;
}
```

図 20: 専用通信ライブラリの送受信関数の概要

パッキングとデータ型　現在拡張 Tiny language では、整数型と浮動小数点型の 2 種類のデータを扱える。このため専用通信ライブラリでは、データ長の異なる 2 種類のデータを送受信できなければならない。送受信バッファの利用効率やメッセージの利用密度の面からはこれらを隙間なく並べることが望ましいが、このような処理は性能低下の原因となる。

そこで専用通信ライブラリの内部では、共用体を用いることによってこれらのデータを固定長として扱う。こうすることによってパッキング/アンパッキング処理を高速化することが可能となる。ただし整数型のデータを送受信する場合には効率が低下するが、数値処理プログラムで扱うデータは浮動小数点型のものが支配的であるため、これは問題にならない。

6　性能評価

以上で述べた並列化手法および最適化手法は、既に TINPAR に実装されている。そこで本章では、簡単な数値処理プログラムの並列化に対する有効性、個々の最適化手法の効果、およびオブジェクトコードの実行時間に対する通信処理の影響について述べる。なお評価には AP1000 を用いた。評価の条件を図 21に示す。

6.1　簡単な数値処理プログラムによる評価

まず最初に TINPAR による並列化の有効性を示すために、行列積、ガウス消去法、偏微分方程式の反復解法、および FFT のプログラムを並列化して台数効果を測定した。

- 評価には自動的に並列化されたオブジェクトコードを用いた。

- オブジェクトコードのコンパイルには AP1000 で標準のcc.[hc]c7 スクリプトを用い、C コンパイラには SPARCompiler C 2.0.1、最適化オプションには`-fast -cg87 -O4`、セルライブラリには`lib.fast` を指定した。

- 専用通信ライブラリの実装には AP1000 の標準通信ライブラリを用い、実行時オプションには`-R 0 -FPE 2 f -LSEND -RB 512` を指定した。ただし行列積 1 についてはリングバッファがオーバーフローしたため、`-R 0 -FPE 2 f -NOLSEND` を指定した。

- 加速率は (逐次版の実行時間)/(並列版の実行時間) として求めた。

図 21: 性能評価の条件

```
const n = 512
const n_cell = 64
processors n_cell
real a(0:n - 1:*, 0:n - 1)
real b(0:n - 1:*, 0:n - 1)
real c(0:n - 1:*, 0:n - 1)
real local_b(0:n - 1, 0:n - 1)
/* a()、b()、c() は 1 次元めでブロック分割 */

/* リモートデータのローカルコピーを作成する。 */
for i = 0, n - 1 do
  for j = 0, n - 1 do
    local_b(i, j) = b(i, j)
  endfor
endfor
/* ローカルに計算する。 */
for i = 0, n - 1 do
  for j = 0, n - 1 do
    c(i, j) = 0.0
    for k = 0, n - 1 do
      c(i, j) = c(i, j) + a(i, k) * local_b(k, j)
    endfor
  endfor
endfor
```

図 22: 行列積 1 のプログラム

6.1.1 行列積

評価には、単純なアルゴリズムおよび並列化に適したアルゴリズムによる 2 つのプログラムを用いた。プログラムの概要を以下に述べる。

行列積 1

ijk 型の内積法によって行列積を求めるプログラムである (図 22参照)。ここで配列`a()`、`b()`、`c()` は 1 次元めでブロック分割されている。一方配列`local_b()` は配列`b()` のローカルコピーを一時的に格納するための配列であり、全プロセッサがそのコピーを持つ。

このプログラムを並列化した場合、各プロセッサはまず配列`b()` のローカルコピーを作成し、これを用いて計算を進める。こうすることにより、同一のリモートデータを複数回参照することによって生じる冗長な通信を削除することができる。なおこのような最適化は、将来的には前処理部によって自動化する予定である。

```
const n_cell = 64
const block_size = 8
processors n_cell
real a(0:n_cell - 1:1, 0:block_size - 1, \
       0:n_cell - 1, 0:block_size - 1)
real b(0:n_cell - 1 + 1:1, 0:block_size - 1, \
       0:n_cell - 1, 0:block_size - 1)
real c(0:n_cell - 1:1, 0:block_size - 1, \
       0:n_cell - 1, 0:block_size - 1)
/* 配列はすべて 1 次元めでサイクリック分割 */

for i1 = 0, n_cell - 1 do
  for i2 = 0, block_size - 1 do
    for j1 = 0, n_cell - 1 do
      for j2 = 0, block_size - 1 do
        c(i1, i2, j1, j2) = 0.0
      endfor
    endfor
  endfor
endfor
for phase = 0, n_cell - 1 do
  /* ローカルにもつa()、b() の要素を使って
     対応するc() の要素を計算する。 */
  for i1 = 0, n_cell - 1 - phase do
    for i2 = 0, block_size - 1 do
      for j2 = 0, block_size - 1 do
        for k1 = 0, n_cell - 1 do
          for k2 = 0, block_size - 1 do
            c(i1, i2, i1 + phase, j2) \
              = c(i1, i2, i1 + phase, j2) \
                + a(i1, i2, k1, k2) * b(i1, j2, k1, k2)
          endfor
        endfor
      endfor
    endfor
  endfor
  for i1 = n_cell - phase, n_cell - 1 do
    for i2 = 0, block_size - 1 do
      for j2 = 0, block_size - 1 do
        for k1 = 0, n_cell - 1 do
          for k2 = 0, block_size - 1 do
            c(i1, i2, i1 + phase - n_cell, j2) \
              = c(i1, i2, i1 + phase - n_cell, j2) \
                + a(i1, i2, k1, k2) * b(i1, j2, k1, k2)
          endfor
        endfor
      endfor
    endfor
  endfor
  for i2 = 0, block_size - 1 do
    for j1 = 0, n_cell - 1 do
      for j2 = 0, block_size - 1 do
        b(n_cell - 1 + 1, i2, j1, j2) = b(0, i2, j1, j2)
      endfor
    endfor
  endfor
  /* ローカルにもつb() の要素を隣接プロセッサ間で交換する。 */
  for i1 = 0, n_cell - 1 do
    for i2 = 0, block_size - 1 do
      for j1 = 0, n_cell - 1 do
        for j2 = 0, block_size - 1 do
          b(i1, i2, j1, j2) = b(i1 + 1, i2, j1, j2)
        endfor
      endfor
    endfor
  endfor
endfor
```

図 23: 行列積 2 のプログラム

行列積 2

従来より並列向きであるとされているアルゴリズムによって行列積を求めるプログラムである (図 23参照)。ここで配列`a()`、`b()`、`c()` は 1 次元めでサイクリック分割されている。ただしこれらの配列は、各要素のアドレッシングを簡潔にするために、4 次元配列として宣言されている。また配列`b()` は、1 次元めと 3 次元め、および 2 次元めと 4 次元めが転置された状態で格納される。

このプログラムを並列化した場合、各プロセッサはローカルに持つ配列`a()`、`b()` の要素を使って対応する配列`c()` の要素の計算を行った後、隣接プロセッサ間で配列`b()` の内容を交換する (図 24参

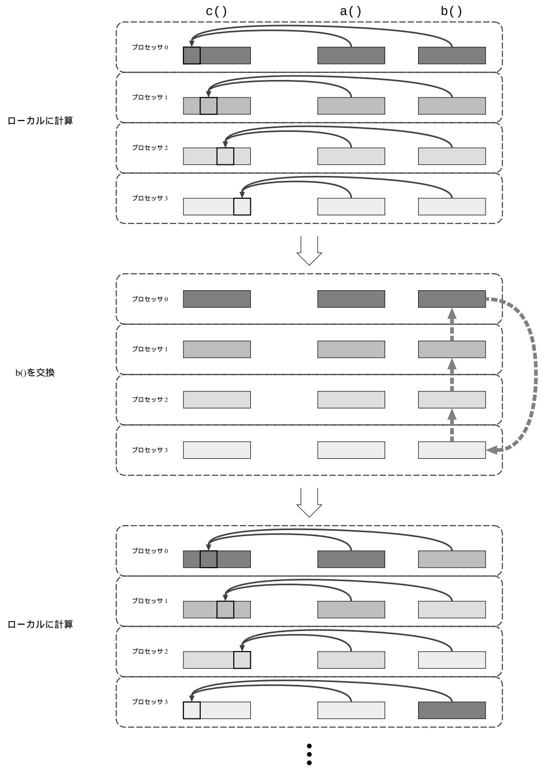

図 24: 行列積 2 の計算方法

照。これを繰り返すことによって、メモリおよびネットワークを効率よく使いながら計算を進めることができる。

これらのプログラムを並列化した場合の台数効果を図25、26に示す[3]。プロセッサ数を増やすと並列化効率[4]が低下するが、どちらも良好な結果が得られているといえる。ここでプロセッサ数を増やしたときの効率の低下は、総通信量の増加が原因である。すなわち問題のサイズをn、プロセッサ数をpとすると、総演算量は$O(n^3)$で一定であるのに対して総通信量は$O(n^2p)$であるために、プロセッサ数が増えるにしたがって演算量に対する通信量の比率が増加し、効率が低下している。

そのうえ行列積 1 では、プログラムの前半で全プロセッサが一斉に配列`b()`の内容を放送する。このため問題のサイズが大きくなると、各プロセッサは一時期に大量のメッセージを受信することになる。この結果問題のサイズをある程度大きくすると、演算対象の配列自体は格納できるにも関わらず、OS のメッセージ受信バッファがオーバーフローしてしまい、実行が不可能となった。

これに対して行列積 2 では、演算と交互かつ複数回にわたって通信を行うために、1 回当たりの通信量が少なくてすみ、問題のサイズをある程度大きくしても実行が可能であった。しかしこの反面、プロセッサ数を増やすと、各プロセッサがローカルにもつデータ量が減少し、通信が行われる間隔が短くなる。64 プロセッサのときの急激な効率の低下は、これによるメッセージ受信待ちのレイテンシの増加が原因と考えられる。

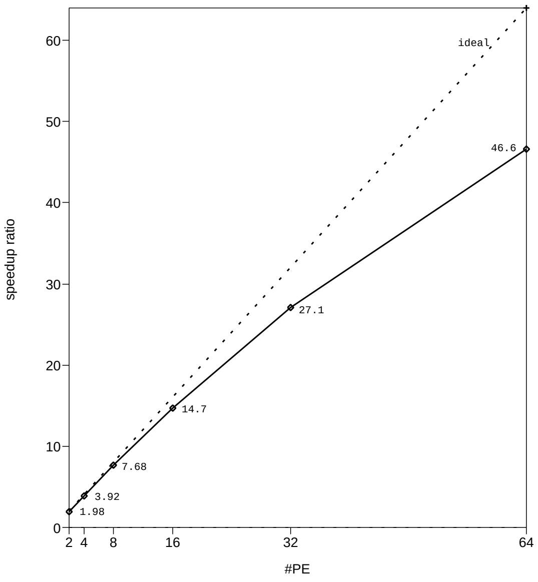

図 25: 行列積 1 の台数効果

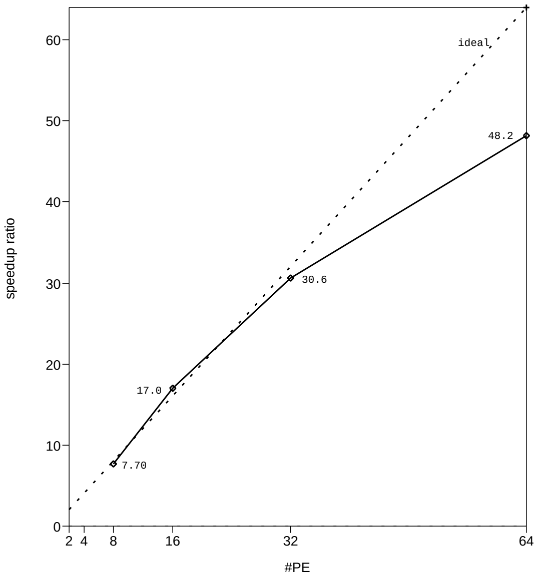

図 26: 行列積 2 の台数効果

なお行列積 2 では、16 プロセッサのときにスーパーリニアの性能[5]が出ているが、これはシステム全体の総キャッシュ容量の増加による効果である。

6.1.2 ガウス消去法

評価には、LINPACK パッケージ [18] の`dgefa`サブルーチンと等価なプログラムを用いた。プログラムの概要を以下に述べる。

ガウス消去法

部分的ピボッティング付きガウス消去法のプログラムである (図 27参照)。ここで配列はすべて 1 次元めでサイクリック分割されている。ただし配列`a()`は転置された状態で格納されるため、実際には列方向の分割となる。

---

[3] 行列積 2 では、8 プロセッサ以上の場合に LSEND モードによる高速な通信を用いることができたため、その結果のみを示している。なお行列積 1 では、リングバッファがオーバーフローするため、LSEND モードによる通信を全く用いることができなかった。

[4] 並列化効率 = 加速率/プロセッサ数である。

[5] 加速率がプロセッサ数を越えること。

```
const n = 1024
const n_cell = 64
processors n_cell
real a(0:n - 1:1, 0:n - 1)
real local_dmax(0:n_cell - 1:1)
real local_a(0:n_cell - 1:1, 0:n - 1)
integer ipvt(0:n - 1:1)
integer local_idamax(0:n_cell - 1:1)
real abs_a_k_i, dmax, pivot, t
integer info, l
/* 配列はすべて 1 次元めでサイクリック分割 */

info = 0
for k = 0, n - 2 do
  /* 分解列を担当するプロセッサがピボット選択を行う。 */
  local_idamax(OWNER(a(k, k))) = k
  local_dmax(OWNER(a(k, k))) = abs(a(k, k))
  for i = k + 1, n - 1 do
    abs_a_k_i = abs(a(k, i))
    dmax = local_dmax(OWNER(a(k, k)))
    if (abs_a_k_i > dmax) then
      local_idamax(OWNER(a(k, k))) = i
      local_dmax(OWNER(a(k, k))) = abs(a(k, i))
    endif
  endfor
  l = local_idamax(OWNER(a(k, k)))
  ipvt(k) = l
  pivot = a(k, l)
  if (pivot != 0) then
    if (l != k) then
      t = a(k, l)
      a(k, l) = a(k, k)
      a(k, k) = t
    endif
    t = -1.0 / a(k, k)
    for i = k + 1, n - 1 do
      a(k, i) = t * a(k, i)
    endfor
    /* 分解列の値を全プロセッサに渡す。
       ただし次の分解列を担当するプロセッサを優先する。 */
    for cid = OWNER(a(k + 1, i)), n_cell - 1 do
      for i = k + 1, n - 1 do
        local_a(cid, i) = a(k, i)
      endfor
    endfor
    for cid = 0, OWNER(a(k + 1, i)) - 1 do
      for i = k + 1, n - 1 do
        local_a(cid, i) = a(k, i)
      endfor
    endfor
    for j = k + 1, n - 1 do
      t = a(j, l)
      if (l != k) then
        a(j, l) = a(j, k)
        a(j, k) = t
      endif
      for i = k + 1, n - 1 do
        a(j, i) \
          = a(j, i) + t * local_a(OWNER(a(j, i)), i)
      endfor
    endfor
  else
    info = k + 1
  endif
endfor
```

図 27: ガウス消去法のプログラム

このプログラムを並列化した場合、変数`abs_a_k_i`、`dmax`および`t`に対する代入文から生成される通信コードは、不用な`SEND`/`RECV`文として削除される。このため、変数`t`に対するスカラエクスパンジョン[6]が不用となるだけでなく、配列`local_idamax()`および`local_dmax()`を導入することにより、分解列を担当するプロセッサがローカルにピボット選択を行うことも可能となっている。なおこの点を考慮した配列`local_idamax()`および`local_dmax()`の導入は、将来的には前処理部によって自動化する予定である。

一方配列`local_a()`は、各プロセッサで分解列のローカルコピーを一時的に格納するための配列である。さらにこの配列に対する代入は明示的にスケジューリングされており、次の分解列を担当するプロセッサのレイテンシが小さくなるように配慮されている。ただしこの問題は、専用通信ライブラリの放送通信機能があまり高速ではないことに起因しており、この点が改善されれば自然に解消される (5章参照)。

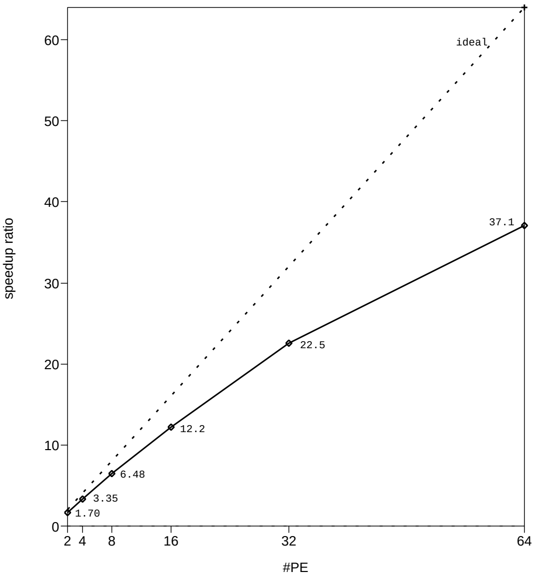

図 28: ガウス消去法の台数効果

このプログラムを並列化した場合の台数効果を図 28 に示す。プロセッサ数を増やすと並列化効率が低下するが、まずまずの結果が得られているといえる。ここでプロセッサ数を増やしたときの効率の低下は、総通信量の増加が原因である。すなわち、総演算量は $O(n^3)$ で一定であるのに対して総通信量は $O(n^2p)$ であるために、プロセッサ数が増えるにしたがって演算量に対する通信量の比率が増加し、効率が低下している。一方プロセッサ数が少ない場合でも効率がよいとはいえないが、これはピボット選択の処理が並列化されず、分解列を所有しているプロセッサによって逐次的に実行されていることが原因である。

6.1.3 偏微分方程式の反復解法

評価には、ヤコビ法および SOR 法のプログラムを用いた。プログラムの概要を以下に述べる。

ヤコビ法

ヤコビ法による楕円型偏微分方程式の求解プログラムである (図 29参照)。ここで配列`a()`、`b()`は 1 次元めでブロック分割されている。

このプログラムを並列化した場合、配列`a()`の要素のうちで分割の境界に接するものの計算には通信が発生するが、それらの計算自体は完全に並列化される。

SOR 法 1

SOR(Successive Over-Relaxation) 法による楕円型偏微分方程式の求解プログラムである (図 30参照)。ここで配列`a()`は 2 次元めでブロック分割されている。

---

[6] スカラ変数に対する逆依存関係を除去するために、当該変数を配列化すること [16]。

```
const n = 512
const n_cell = 64
const n_time = 10
processors n_cell
real a(0:n - 1:*, 0:n - 1)
real b(0:n - 1:*, 0:n - 1)
/*  a()、b() はともに 1 次元めでブロック分割 */

for time = 0, n_time - 1 do
  for i = 1, n - 2 do
    for j = 1, n - 2 do
      a(i, j) = (b(i - 1, j) + b(i, j - 1) \
                 + b(i, j + 1) + b(i + 1, j)) / 4
    endfor
  endfor
  for i = 1, n - 2 do
    for j = 1, n - 2 do
      b(i, j) = a(i, j)
    endfor
  endfor
endfor
```

図 29: ヤコビ法のプログラム

```
const n = 512
const n_cell = 64
const n_time = 10
processors n_cell
real omega
real a(0:n - 1, 0:n - 1:*)
/*  a() は 2 次元めでブロック分割 */

for time = 0, n_time - 1 do
  for i = 1, n - 2 do
    for j = 1, n - 2 do
      a(i, j) \
        = (a(i - 1, j) + a(i, j - 1) \
           + a(i, j + 1) + a(i + 1, j)) * (omega / 4) \
          + a(i, j) * (1 - omega)
    endfor
  endfor
endfor
```

図 30: SOR 法 1 のプログラム

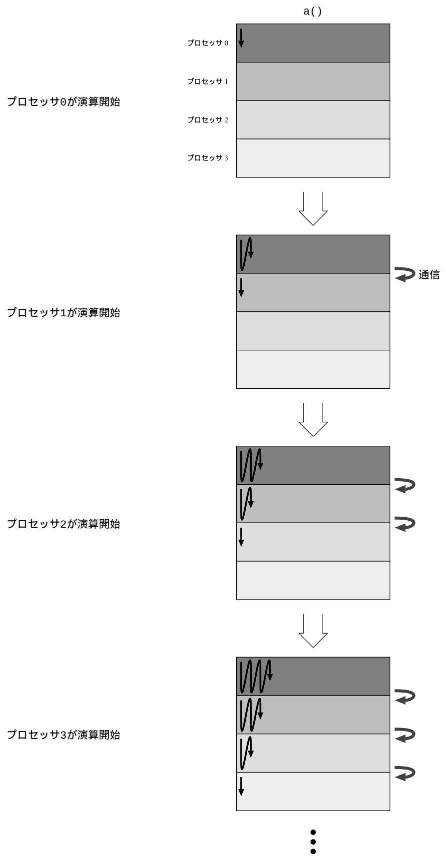

図 31: SOR 法 1 の計算方法

```
const n = 512
const n_cell = 64
const n_time = 10
const tile_width = 16
processors n_cell
real omega
real a(0:n - 1:*, 0:n - 1)
/*  a() は 1 次元めでブロック分割 */

for time = 0, n_time - 1 do
  for outer_i = 0, n - 2, tile_width do
    for i = 1, n - 2 do
      /*  2 次元めには幅tile_width のタイリングを施す。 */
      for j = MAX(1, outer_i), \
              MIN(n - 2, outer_i + tile_width - 1) do
        a(i, j) \
          = (a(i - 1, j) + a(i, j - 1) \
             + a(i, j + 1) + a(i + 1, j)) * (omega / 4) \
            + a(i, j) * (1 - omega)
      endfor
    endfor
  endfor
endfor
```

図 32: SOR 法 2 のプログラム

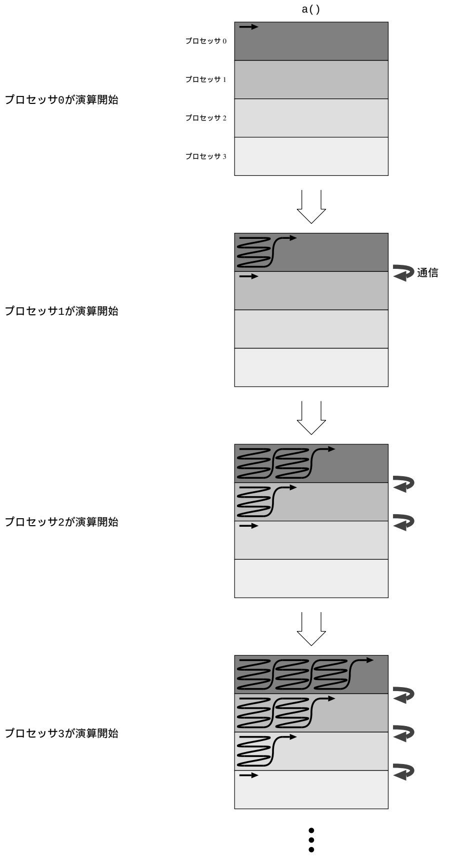

図 33: SOR 法 2 の計算方法

SOR 法では、ヤコビ法と異なり、配列の各要素を計算する 2 重ループの両方にループ運搬のデータ依存関係が生じる。そのためこのプログラムを並列化した場合、これらのループはパイプライン的に実行されることになる (図 31参照)。

SOR 法 2

SOR 法 1 に対してタイリング技法 [15] を用いたプログラムである (図 32参照)。ただし配列`a()` は 1 次元めでブロック分割されている。

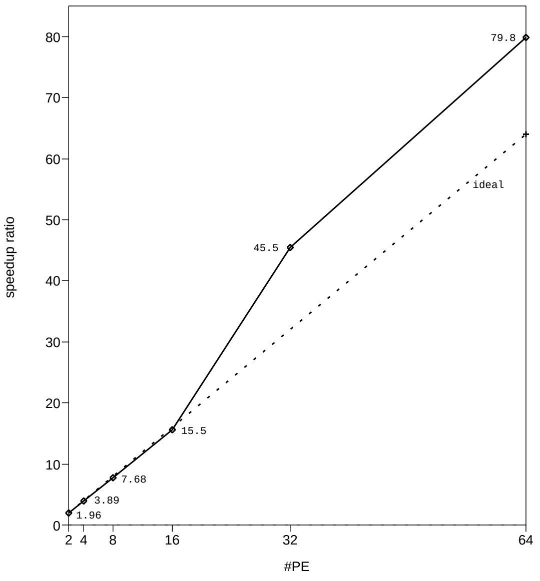

図 34: ヤコビ法の台数効果

タイリング技法とは、先に述べたようなループ運搬のデータ依存関係の結果として頻発する通信の 1 回当たりの演算量を大きくするために、イタレーションの実行をスケジューリングする最適化技法である (図 33参照)。具体的には`outer_j` のループが、この技法を用いるために`j` のループをストリップマイニングして得られた外側ループである。

これらのプログラムを並列化した場合の台数効果を図 34、35、36に示す。ヤコビ法では一部でスーパーリニアの性能が出ているが、プロセッサ数を増やすといずれの場合も並列化効率が低下している。ここでプロセッサ数を増やしたときの効率の低下は、総通信量の増加が原因の 1 つとなっている。すなわち、総演算量は $O(n^2)$ で一定であるのに対して総通信量は $O(np)$ であるために、プロセッサ数が増えるにしたがって演算量に対する通信量の比率が増加し、効率が低下している。一方 3 者間の性能の違いは、通信の粒度の差が原因である。ヤコビ法ではループのイタレーション間にデータ依存関係がないため、必要となるデータをあらかじめ送信しておくことが可能であり、通信の粒度を大きくすることができる。しかし SOR 法 1 および SOR 法 2 ではループ運搬のデータ依存関係が存在するため、更新された最新のデータをループ内で送信しなければならず、通信の粒度を大きくすることができない。しかも後者の場合には、プロセッサ数が増えるにしたがって 1 回の通信当たりの演算量が減少し、効率が急激に低下している。ただし SOR 法 2 を SOR 法 1 と比較すると、タイリング技法を用いることにより、効率の低下がある程度抑えられていることがわかる。

なおヤコビ法では、32 および 64 プロセッサのときにスーパーリニアの性能が出ているが、これはシステム全体の総キャッシュ容量の増加による効果である。

6.1.4 FFT

評価には、周波数間引き型 FFT のプログラムを用いた。プログラムの概要を以下に述べる。

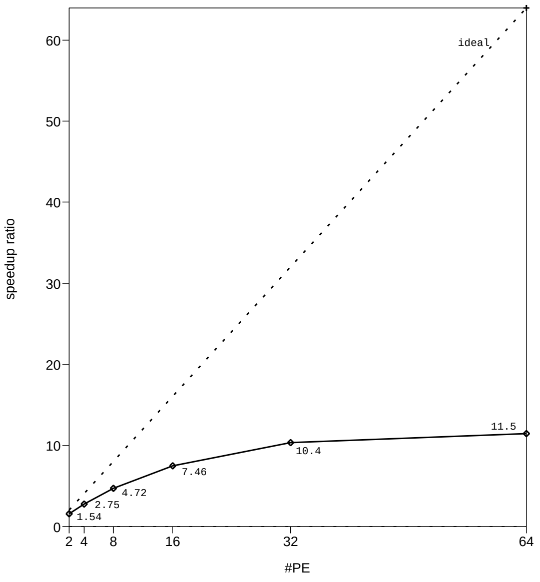

図 35: SOR 法 1 の台数効果

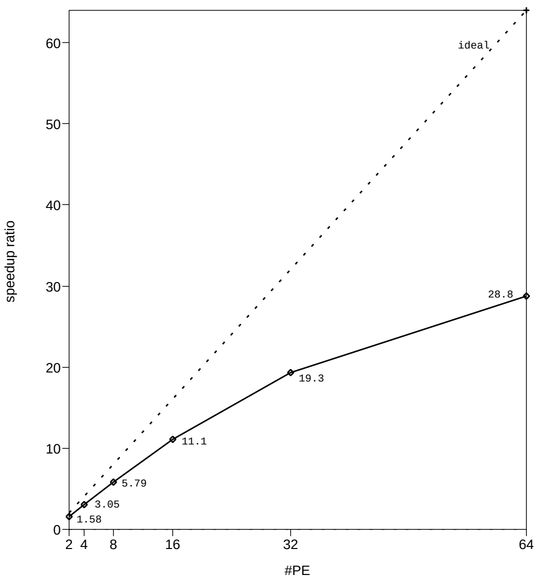

図 36: SOR 法 2 の台数効果

FFT

基数 2 の周波数間引き型 FFT(Fast Fourier Transform) を行うプログラムである (図 37参照)。ここで配列`a1()` は 3 次元め、配列`a2()`、`tmp()` は 1 次元めでブロック分割されている。ただし配列`a1()`、`a2()` は、各要素のアドレッシングを簡潔にするために、4 次元配列として宣言されている。最初の 3 次元はそれぞれ、$2^{\mathrm{log\_n\_cell}}$、$2^{\mathrm{log\_block\_size}}$、$2^{\mathrm{log_n_cell}}$の大きさを持ち、これらを合わせて 1 次元 $2^{\mathrm{log_n_cell}*2+\mathrm{log_block_size}}$点 FFT とみなすことができる。一方最後の次元は、複素数の実部と虚部を格納するために用いられている。なおバタフライ演算で用いられる回転因子の配列`w()` は、あらかじめ計算されているものとする。

一般に基数 2 の FFT では、$2^k(k = 1, 2, \ldots)$ だけ距離の離れた 2 要素間のバタフライ演算を繰り返す。

```
const log_n_cell = 6
const log_block_size = 4
processors n_cell
real a1(0:2 ** log_n_cell - 1, 0:2 ** log_block_size - 1, 0:2 ** log_n_cell - 1:1, 0:1)
real a2(0:2 ** log_n_cell - 1:1, 0:2 ** log_block_size - 1, 0:2 ** log_n_cell - 1, 0:1)
real w(0:2 ** log_n_cell - 1, 0:2 ** log_block_size - 1, 0:2 ** log_n_cell - 1, 0:1)
real tmp(0:2 ** log_n_cell - 1:1, 0:1)
integer distance
/*  a1()は3次元め、a2()およびtmp()は1次元めでサイクリック分割 */

/*  上位log_n_cellビットのFFTをローカルに計算する。 */
for phase = 0, log_n_cell - 1 do
  distance = 2 ** (log_n_cell - phase - 1)
  for outer_i = 0, n_cell - 1, distance * 2 do
    for i = outer_i, outer_i + distance - 1 do
      for j = 0, block_size - 1 do
        for k = 0, n_cell - 1 do
          tmp(k, 0) = a1(i, j, k, 0) - a1(i + distance, j, k, 0)
          tmp(k, 1) = a1(i, j, k, 1) - a1(i + distance, j, k, 1)
          a1(i, j, k, 0) = a1(i, j, k, 0) + a1(i + distance, j, k, 0)
          a1(i, j, k, 1) = a1(i, j, k, 1) + a1(i + distance, j, k, 1)
          a1(i + distance, j, k, 0) \
            = w(i - outer_i + distance, j, k, 0) * tmp(k, 0) - w(i - outer_i + distance, j, k, 1) * tmp(k, 1)
          a1(i + distance, j, k, 1) \
            = w(i - outer_i + distance, j, k, 0) * tmp(k, 1) + w(i - outer_i + distance, j, k, 1) * tmp(k, 0)
        endfor
      endfor
    endfor
  endfor
endfor
/*  真ん中log_block_sizeビットのFFTをローカルに計算する。 */
for phase = log_n_cell, log_n_cell + log_block_size - 1 do
  distance = 2 ** (log_n_cell + log_block_size - phase - 1)
  for i = 0, n_cell - 1 do
    for outer_j = 0, block_size - 1, distance * 2 do
      for j = outer_j, outer_j + distance - 1 do
        for k = 0, n_cell - 1 do
          tmp(k, 0) = a1(i, j, k, 0) - a1(i, j + distance, k, 0)
          tmp(k, 1) = a1(i, j, k, 1) - a1(i, j + distance, k, 1)
          a1(i, j, k, 0) = a1(i, j, k, 0) + a1(i, j + distance, k, 0)
          a1(i, j, k, 1) = a1(i, j, k, 1) + a1(i, j + distance, k, 1)
          a1(i, j + distance, k, 0) \
            = w(0, j - outer_j + distance, k, 0) * tmp(k, 0) - w(0, j - outer_j + distance, k, 1) * tmp(k, 1)
          a1(i, j + distance, k, 1) \
            = w(0, j - outer_j + distance, k, 0) * tmp(k, 1) + w(0, j - outer_j + distance, k, 1) * tmp(k, 0)
        endfor
      endfor
    endfor
  endfor
endfor
/*  データを再分散する。 */
for i = 0, n_cell - 1 do
  for j = 0, block_size - 1 do
    for k = 0, n_cell - 1 do
      a2(i, j, k, 0) = a1(i, j, k, 0)
      a2(i, j, k, 1) = a1(i, j, k, 1)
    endfor
  endfor
endfor
/*  下位log_n_cellビットのFFTをローカルに計算する。 */
for phase = log_n_cell + log_block_size, log_n_cell + log_block_size + log_n_cell - 1 do
  distance = 2 ** (log_n_cell + log_block_size + log_n_cell - phase - 1)
  for i = 0, n_cell - 1 do
    for j = 0, block_size - 1 do
      for outer_k = 0, n_cell - 1, distance * 2 do
        for k = outer_k, outer_k + distance - 1 do
          tmp(i, 0) = a2(i, j, k, 0) - a2(i, j, k + distance, 0)
          tmp(i, 1) = a2(i, j, k, 1) - a2(i, j, k + distance, 1)
          a2(i, j, k, 0) = a2(i, j, k, 0) + a2(i, j, k + distance, 0)
          a2(i, j, k, 1) = a2(i, j, k, 1) + a2(i, j, k + distance, 1)
          a2(i, j, k + distance, 0) \
            = w(0, 0, k - outer_k + distance, 0) * tmp(i, 0) - w(0, 0, k - outer_k + distance, 1) * tmp(i, 1)
          a2(i, j, k + distance, 1) \
            = w(0, 0, k - outer_k + distance, 0) * tmp(i, 1) + w(0, 0, k - outer_k + distance, 1) * tmp(i, 0)
        endfor
      endfor
    endfor
  endfor
endfor
```

図37: FFTのプログラム

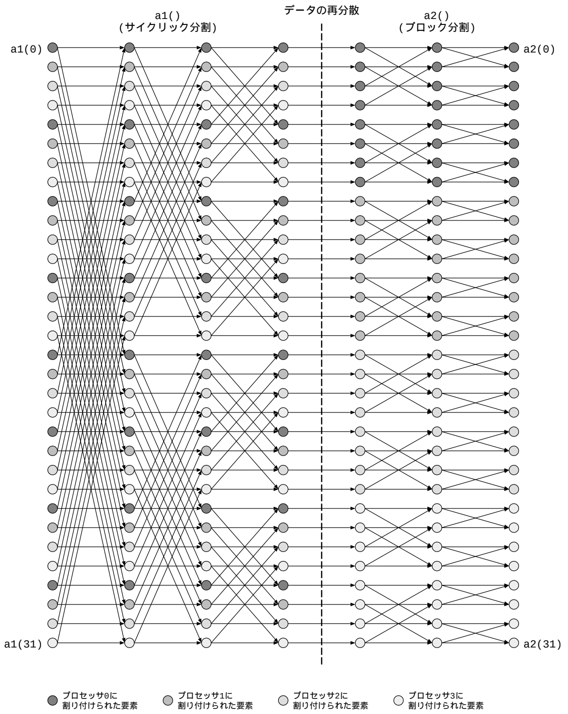

図 38: FFT の計算方法
($\mathtt{log\_n\_cell} = 2$、$\mathtt{log\_block\_size} = 1$)

> ここで演算対象の配列を $2^{\mathtt{log\_n\_cell}}$ プロセッサに対してサイクリック分割した場合には、各プロセッサ上に $2^{\mathtt{log\_n\_cell}}$ おきの要素が割り付けられることにより、距離 $2^k (k \geq \mathtt{log\_n\_celL})$ のバタフライ演算を各プロセッサでローカルに行うことができる。一方演算対象の要素数が $2^{\mathtt{log\_n\_cell}*2}$ 以上であれば、ブロック分割を施すことにより、距離 $2^k (k < \mathtt{log\_n\_celL})$ のバタフライ演算を各プロセッサでローカルに行うことができる。ゆえに要素数 $2^{\mathtt{log\_n\_cell}*2+\mathtt{log\_block\_size}} (\mathtt{log\_block\_size} \geq 0)$ の FFT は、データの再分散を適切に行うことによって効率よく並列化できる (図 38参照)。
>
> そこでこのプログラムでは、この方法にもとづいて最初の 2 つのループではサイクリック分割、最後のループではブロック分割を演算対象の配列に施している。

このプログラムを並列化した場合の台数効果を図 39に示す。プロセッサ数に関わらず、まずまずの結果が得られているといえる。これは、必要となる通信が 1 回のデータ再分散だけであり、プロセッサ数を増やしても総通信量が変化しないためである。ただし総演算量が $O(n \log n)$、総通信量が $O(n)$ であるために、プロセッサ数が少ない場合には、他のプログラムよりも並列化効率が低くなっている。

6.2 各最適化手法の効果

次に個々の最適化手法の効果を示す。評価にはヤコビ法のプログラムを 64 プロセッサ用に並列化したものを用いた。最適化レベルと加速率の関係を図 40に示す。ここで最適化レベルは図 18の番号を用いて示されている。

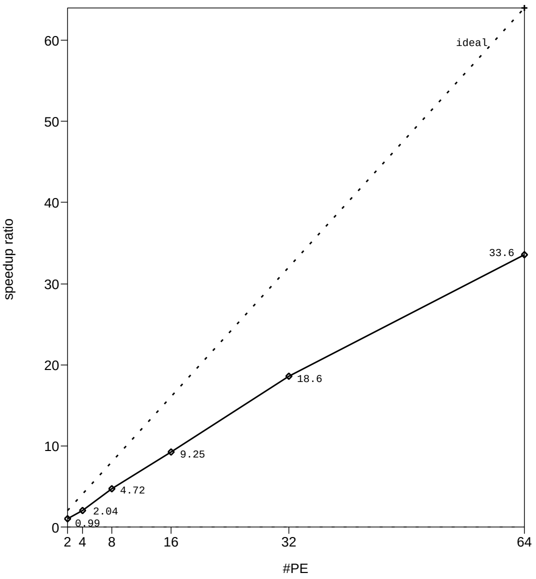

図 39: FFT の台数効果

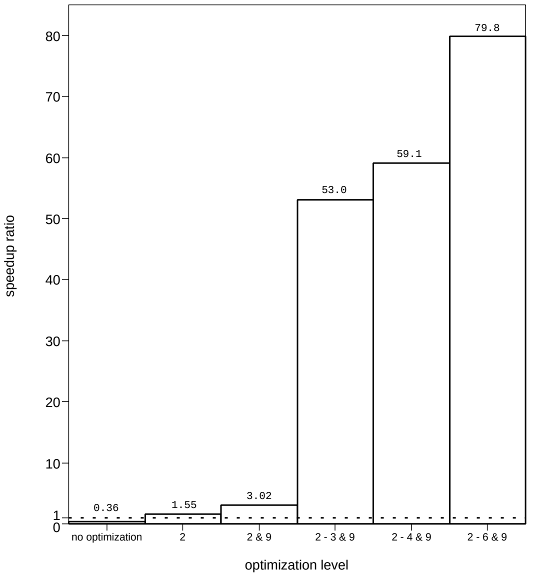

図 40: 各最適化手法の効果

この結果より、ループの実行範囲の縮小が非常に効果的であることがわかる。これは空回りとなる不用なイタレーションを削除したことによる効果である。また `if` 文を削除するためのループ分割も効果的であるが、これはループ分割を行うことによって、並列化の際に挿入された `if` 文をすべて削除したことによる効果である。

一方 `SEND/RECV` 文をともに含むループのループ分配を行わなかった場合には、通信による暗黙の同期によって実行が逐次化されてしまい、たとえその他すべての最適化を行っても加速率は 12.2 倍であった。また `MSG_BUFFER_FLUSH` 文の挿入を行わなかった場合には、送信データのバッファリングが悪影響を及ぼし、他の最適化の効果を打ち消してしまった。例えば、すべての最適化を行った場合には加速率は 79.8 倍であったが、この最適化を外した場合には 41.5 倍であった。このときプロセッサの平均稼働率は 86.1%から 45.8%へと低下していた。

なお `SEND` 文のリスケジューリングは、`SEND/RECV` 文

をともに含むループのループ分配を FFT に適用するために必要となったが、それ以外のプログラムでは効果が現れなかった。ただし効果が現れなかったプログラムのうちでガウス消去法以外のものは、ソースプログラムが短かったためにリスケジューリングが全く行われていなかった。また不用な`SEND`/`RECV` 文の削除の効果については、文献 [4] に述べられている。

6.3 実行時間に対する通信処理の影響

最後に、通信処理のオーバーヘッドがオブジェクトコードの実行時間に与えている影響を評価するために、AP1000 の標準通信ライブラリの実行時間、通信ランタイムシステムの実行時間、および通信以外の演算の実行時間が総実行時間に占める割合を測定した。ここで AP1000 の標準通信ライブラリは実際のメッセージ送受信を行い、通信ランタイムシステムはデータのパッキング/アンパッキングおよび送信/放送用バッファのフラッシュを行っている (5章参照)。

評価には SOR 法 1 を除くすべてのプログラムを 64 プロセッサ用に並列化したものを用いた[7]。理想的な実行時間、すなわち (逐次版の実行時間)/64 を 1 として正規化した結果を図 41に示す。

まず通信以外の演算の実行時間は最小で 0.57、最大で 1.06 となっており、並列化によるオーバーヘッドは十分小さいといえる。これは最適化によって不用なコードが十分に削除されたことによる効果である。

一方通信ランタイムシステムの実行時間は最小で 0.06、最大で 0.61 となっており、あまり効率がよくない。これは動的ベクトル化機能によるオーバーヘッドが無視できないことを示している。しかしガウス消去法の場合は、分解列の受渡しを効率よくスケジューリングするためにあえて放送通信機能を用いなかったために、送信データのパッキングのオーバーヘッドが増加したことが原因である。一方 SOR 法 2 の場合は、ループ運搬のデータ依存関係が存在することによって中粒度の通信が行われたために、送信バッファのフラッシュのオーバーヘッドが表面化しているものと考えられる。また FFT の場合は、演算量に対する通信量の比率が他のプログラムに比べて大きいために、通信ランタイムシステムのオーバーヘッドも大きく現れているものと考えられる。なお行列積 1 は行列積 2 に比べて効率がよいが、これは放送通信機能を活用したことによる効果である。

また AP1000 の標準通信ライブラリの実行時間は最小で 0.08、最大で 0.59 となっており、改善の余地がみられる。すなわち図 42に示されているように、ブロッキング送信を行う関数`l_asend()`、および OS 領域から通信ランタイムシステム内に受信メッセージをコピーする関数 `readmsg()` がこのうちに占めている割合が少なくない[8]。これらに関しては、先に述べたように、送信はノンブロッキングでかつ OS 領域へのコピーなしで[9]、受信は通信ランタイムシステム内へ直接行うことにより、かなりの時間を短縮できると考えられる。さらに、専用ハードウェアによるデータ領域からの直接送信を行えば、通信ランタイムシステムの実行時間をも大幅に短縮できるはずである。

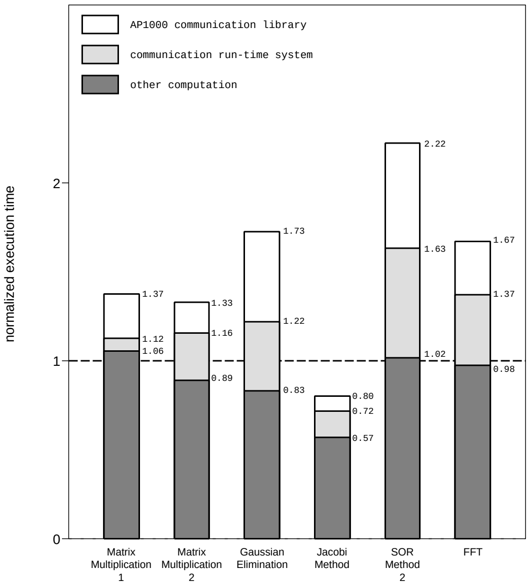

図 41: オブジェクトコードの実行時間の内訳

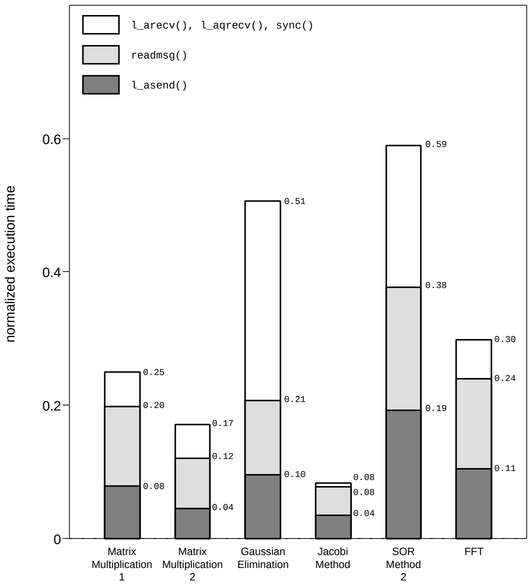

図 42: AP1000 の通信ライブラリの実行時間の内訳

7 関連研究

分散メモリ型並列計算機のための並列化コンパイラの研究は多数行われている。これらのうちで TINPAR と同様に owner computes rule にもとづくものとして Fortran D コンパイラ [10] があげられる。Fortran D コンパイラによる研究はこれらの研究の中でも最も進んでいるものの 1 つであり、HPF の言語仕様も Fortran D の言語仕様 [9] がベースとなっている。

Fortran D コンパイラでは、iteration/index set の概念を用いて演算の分割と通信コードの生成を行っている。Fortran D コンパイラの通信コード生成手法は、コンパイル時の解析にもとづくメッセージのベクトル化を基本としている。一方我々は動的ベクトル化機能を用いることによって、柔軟でかつ効率のよい通信処理を実現している。

---

[7] SOR 法 1 については信頼できるデータが収集できなかった。

[8] `l_arecv()` 関数の実行時間には通信レイテンシも算入されているため、ガウス消去法および SOR 法 2 ではこの割合が高くなっている。

[9] 行列積 1 で用いた NOLSEND モードの通信では、送信メッセージは一旦 OS 領域にコピーされてから送信される。

また文献[13]では、ブロックサイクリック分割された配列に対する演算の分割や通信コードの生成について述べられている。ここで述べられている手法は、ループの実行範囲の縮小のための解析とそれにもとづくコード生成を含んでいるが、分割された次元の添字式のループインデックスに対する係数aが1でない場合の扱いが我々の手法と異なり、適用範囲にも制限がある。

しかしFortran Dコンパイラと TINPARの最大の違いは、その設計方針にある。すなわちFortran Dコンパイラでは完全な自動並列化を目指し、owner computes ruleの緩和、タイリング技法適用の自動化、リダクション演算のアルゴリズム変換などを行っている。一方TINPARでは、これらの処理を並列化に対する前処理として位置づけ、自動または半自動のプログラム変換によって行う。こうすることにより、純粋なコード最適化とアルゴリズムの見直しを分離できるため、柔軟な対応が可能となる。

たとえば文献[12]では本稿と同様なガウス消去法のプログラムを扱っているが、ピボット選択を効率よく行うためにコンパイラがリダクション演算を認識し、owner computes ruleを緩和して並列化を行っている。一方TINPARでは、本稿に示したプログラムを与えると、本稿で述べた一般的かつ包括的な最適化手法のみを用いて効率よく並列化する。これは我々の手法の適用範囲の広さや柔軟性を示しているといえる。

8 おわりに

本稿では、現在我々が開発しているメッセージ交換型並列計算機のための並列化コンパイラTINPARについて述べた。TINPARは拡張Tiny languageで記述された逐次プログラムをowner computes ruleに従って並列化する。このときTINPARは、不用なコードの削除や通信コードの移動を行うことにより、効率のよいオブジェクトコードを生成する。さらにこのオブジェクトコードは、動的ベクトル化機能を持つ専用通信ライブラリを用いることにより、効率のよい通信処理を行う。

TINPARの並列化手法は、強力な最適化手法の適用を前提とした、owner computes ruleにもとづく単純なものである。このため、柔軟なプログラム変換手法による前処理を行うことによって、オブジェクトコードの振舞いを適切に制御することが可能となる。

TINPARの最適化手法は、単純に並列化されたプログラムを高度に最適化する。ループの実行範囲の縮小、恒真/恒偽となる`if`文の削除および`if`文を削除するためのループ分割は、並列化の際に挿入された不用なコードを削除するための最適化手法であり、特に1つめと3つめが効果的であることが実測により示された。一方`SEND`/`RECV`文をともに含むループのループ分配は、通信レイテンシを隠蔽するための最適化手法であり、通信の最適化に必要不可欠であることが示された。また`MSG_BUFFER_FLUSH`文の挿入は、専用通信ライブラリの持つ動的ベクトル化機能を十分に活用するために重要であることも示された。

一方TINPARの専用通信ライブラリは、動的ベクトル化機能を提供する。この機能は、通信ランタイムシステムが送信データをバッファリングし、長いメッセージとして送信することによって、柔軟かつ効率のよい通信処理を実現する。

さらに本稿では、これらの手法の有効性を示すために、簡単な数値処理プログラムを並列化して性能を評価した。この結果64プロセッサのAP1000を用いて、行列積で48.2倍、ガウス消去法で37.1倍、ヤコビ法で79.8倍、SOR法で28.8倍、FFTで33.6倍の加速率が達成された。

しかしこれらの実行時間の内訳を調べてみると、通信処理の占める割合が少なくなかった。すなわち、通信以外の演算については十分に最適化されていたために、通信ランタイムシステムのオーバーヘッドが無視できなくなっていた。しかしこのオーバーヘッドは、専用通信ライブラリの改良および機能拡張によって削減が可能である。またコンパイラ本体も非常に拡張性が高いため、このような専用通信ライブラリの機能拡張に対しても容易に対応可能である。

今後の課題としては、前処理としてのプログラム変換手法の検討があげられる。本稿では2つの具体例をあげたが、この他にも性能向上に直結するものがあると期待できる。

さらに現在本研究と並行してLINPACK Benchmark[19]を用いた性能評価が行われているが、引続きNAS Parallel Benchmarks[20]による性能評価も予定されている。これらの性能評価の作業は、ケーススタディとして今後のTINPARの機能強化に反映される予定である。

謝辞

最後に、本研究の機会を与えて下さった富田 眞治教授、適切な御指導を賜わった中島 浩助教授、数多くの相談に応じて頂いた森 眞一郎助手に深く感謝致します。

また、高並列計算機AP1000の実行環境を御提供頂きました(株)富士通研究所に感謝致します。

さらに、共同研究者である前山 浩二、後藤 慎也両氏をはじめ、日頃御討論頂いた富田研究室の諸氏に感謝致します。

参考文献

[1] 三吉 郁夫, 森 眞一郎, 中島 浩, 富田 眞治: メッセージ交換型並列計算機のための並列化コンパイラ, 情処研報 94-PRG-18, pp.33–40, 1994.

[2] 三吉 郁夫, 前山 浩二, 後藤 慎也, 森 眞一郎, 中島 浩, 富田 眞治: メッセージ交換型並列計算機のための並列化コンパイラTINPAR — 最適化手法と性能評価 —, 情処研報 94-HPC-54, pp.45–52, 1994.

[3] I. Miyoshi, S. Mori, H. Nakashima, and S. Tomita: TINPAR: A Parallelizing Compiler for Message-Passing Multiprocessors, Proc. of the Third Parallel Computing Workshop, Fujitsu Laboratories, pp.P1-H-1–P1-H-9, 1994.

[4] 後藤 慎也: データフロー解析にもとづく並列化コンパイラの通信最適化手法, 特別研究報告書, 京都大学工学部情報工学科, 1995.

[5] 前山 浩二, 後藤 慎也, 三吉 郁夫, 森 眞一郎, 中島 浩, 富田 眞治: 並列化コンパイラTINPARにおける専用通信ライブラリの実装と性能評価, 情報処理学会 第50回全国大会 講演論文集 採録予定, 1995.

[6] M. Metcalf, and J. Reid 著, 西村 恕彦, 和田 英穂, 西村 和夫, 高田 正之 訳: 詳解Fortran90, bit 別冊, 共立出版, 1993.

[7] High Performance Fortran Forum: High Performance Fortran Language Specification Version 1.0, 1993.

[8] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, 1994.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu: Fortran D Language Specification, Technical Report TR90-141, Dept. of Computer Science, Rice University, 1990.

[10] S. Hiranandani, K. Kennedy, and C. Tseng: Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, Proc. of Supercomputing '91, pp.86–100, 1991.

[11] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng: Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines, Proc. of Supercomputing '92, pp.522–534, 1992.

[12] S. Hiranandani, K. Kennedy, and C. Tseng: Preliminary Experiences with the Fortran D Compiler, Proc. of 1993 Int'l Conf. on Supercomputing, pp.338–350, 1993.

[13] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi: Compilation Techniques for Block-Cyclic Distributions, Proc. of 1994 Int'l Conf. on Supercomputing, pp.392–403, 1994.

[14] M. Wolfe: The Tiny Loop Restructuring Research Tool, Proc. of 1991 Int'l Conf. on Parallel Processing vol.II, pp.46–53, 1991.

[15] M. Wolfe: More Iteration Space Tiling, Proc. of Supercomputing '89, pp.655–664, 1989.

[16] 笠原 博徳: 並列処理技術, コロナ社, 1991.

[17] K. Dowd 著; 久良知 真子 訳: High Performance Computing, オーム社, 1994.

[18] J. Dongarra, J. Bunch, C. Moler, and G. Stewart: LINPACK User's Guide, SIAM Publications, 1979.

[19] J. Dongarra: The LINPACK Benchmark: An Explanation, Supercomputing 1st International Conference Proceedings, pp.456–474, 1987.

[20] D. Bailey, J. Barton, T. Lasinski, and H. Simon: The NAS Parallel Benchmarks, Technical Report RNR-91-002 Revision 2, NASA Ames Research Center, 1991.

[21] 石畑 宏明, 稲野 聡, 堀江 健志, 清水 俊幸, 池坂 守夫: 高並列計算機 AP1000 のアーキテクチャ, 信学論 (D-I) vol.J75-D-I no.8, pp.637–645, 1992.

[22] 土肥 実久, 林 憲一, 進藤 達也: ストライドデータ転送機構を用いたコード生成, 情処研報 94-HPC-52, pp.1–6, 1994.

[23] 富士通株式会社システム研究部第 2 研究室: AP1000 動作説明書 第 02 版, 1991.

[24] 富士通株式会社アーキテクチャ研究部第 2 研究室: AP1000 セルプロセッサ論理仕様書 第 04 版, 1992.