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

三吉 郁夫, 前山 浩二, 後藤 慎也, 森 眞一郎, 中島 浩, 富田 眞治
京都大学 工学部

概 要

現在我々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 を開発中である。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。本稿では、TINPAR の構成、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、および最適化に用いられている種々の手法について述べるとともに、簡単な数値処理プログラムを並列化した結果を用いてその有効性を示す。

Abstract

We are developing TINPAR, a parallelizing compiler for message-passing multiprocessors. TINPAR parallelizes sequential programs written in an extended version of Tiny language following the owner computes rule. In this paper, we describe the overview of TINPAR, an efficient communication library, and optimization techniques, and show performance evaluation results for generated codes.

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要であるといえる。

現在我々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 を開発中である [1, 2]。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。

このような方針をとることによって並列化作業全体の見通しがよくなるため、プログラマだけでなく、支援ソフトウェアにとっても有益であると考えられる。そこで TINPAR による並列化処理は、具体的には以下の 3 ステップから構成される。

1. 前処理 ... 並列化に適した形にプログラム変換する。
2. 並列化 ... owner computes rule に従って単純な並列化を行う。
3. 最適化 ... 不用なコードの削除や通信コードの移動を行う。

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。

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、4 章では最適化のステップで用いられる種々の手法について、それぞれ述べる。続いて 5 章では、各最適化手法の効果、簡単な数値処理プログラムの並列化に対する有効性、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 6 章で関連する研究について述べ、7 章でまとめる。

2 TINPAR の概要

本章では、次章以降の準備として TINPAR の入出力および構成について述べる。

2.1 TINPAR の入出力

TINPAR は拡張 Tiny language で記述された逐次プログラムを並列化する。Tiny language とはループ再構成ツール Tiny[3] のために設計された言語であり、これに我々はデータ分割ディレクティブ、`while` 構文およびスコープ規則を追加した。なお現在指定できるデータ分割は、単一次元方向に対するブロック、サイクリックおよびブロックサイクリック分割である。

TINPAR: A Parallelizing Compiler for Message-Passing Multiprocessors
Ikuo Miyoshi, Koji Maeyama, Shin-ya Goto, Shin-ichiro Mori, Hiroshi Nakashima, and Shinji Tomita
Faculty of Engineering, Kyoto University

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。これはさらにコンパイルされ、専用通信ライブラリとリンクされて実行形式となる。こうすることにより、1) 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、2) 対象並列計算機の変更が比較的容易である、などの利点が生じる。

2.2 TINPARの構成

TINPARは以下の6つのモジュールから構成される。

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

   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/利用などが挙げられる。

3. 並列化部

   owner computes ruleに従ってソースプログラムを並列化する。具体的には以下の処理を行う (図1参照)。

   (a) `if` 文の挿入

   分割されたデータに対する代入は、当該データの所有者プロセッサのみが行う。そこでこのようなデータに対する代入文には、所有者プロセッサを判定するための`if`および`endif`文を前後に挿入する。

   (b) `SEND`/`RECV` 文の挿入

   分割されたデータに対する参照は、当該データの所有者プロセッサと参照者プロセッサの間のメッセージ通信で実現される。そこでこのようなデータに対する参照には、所有者プロセッサによる送信のための`SEND`文、参照者プロセッサによる受信のための`RECV`文、および両者を判定するための`if`/`endif`文を挿入する。

   またこの時点では、分割されたデータに対して分割前のデータ全体を格納できる大きさのメモリ領域を確保する。この結果、分割されたデータ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。

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。

5. メモリ再割当部

   分割されたデータ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、分割されたデータ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2参照)。

```
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)

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 1: 並列化

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

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
                            ⇓
real a(0:⌈n ÷ GET_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 2: 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によるオブジェクトコードを出力する。

3 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、およびそれを用いた効率のよい通信処理の実現法について述べる。

3.1 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。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、または一定距離毎にあるならば、

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 [4]。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。

3.2 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。

```
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} do¹
  if OWNER(a(i)) == GET_CELL_ID() then
    a(i) = ...
  endif
endfor
```

図 3: ループの実行範囲の縮小

4 最適化手法

TINPAR は C 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について述べる。なおアルゴリズムの詳細については文献 [2] に述べられている。

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、分割されたデータ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 3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。具体的には、まず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。その後当該ループをストリップマイニングし[2]、求めた条件に従って内側ループの実行範囲を可能な限り縮小する。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、分割されたデータ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 4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

---

[1] 実際には、ストリップマイニングされた 2 重ループとなる。

[2] ブロックおよびサイクリック分割はブロックサイクリック分割の特殊な場合とみなせるため、こうすることによってすべての分割を同様に扱える [2]。

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

図 4: 恒真/恒偽となるif 文の削除

```
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 5: if 文を削除するためのループ分割

削除する。具体的には、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とループインデックスのとりうる値を比較し、条件式が恒真/恒偽であるかどうかを調べる。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 5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。具体的には、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、実行されるべき文が共通となるループインデックス値の範囲毎に当該ループを分割する。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 6上のような場合には、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 または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。

```
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 6: 不用なSEND/RECV 文の削除

```
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 7: SEND 文のリスケジューリング

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 7参照)。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。

```
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 8: `SEND`/`RECV` 文をともに含むループのループ分配

```
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 9: `MSG_BUFFER_FLUSH` 文の挿入

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 8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 8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。

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、先に述べたように動的ベクトル化機能を持つ。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。

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 9参照)。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。

- 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。ここで最適化の適用順序は以下のとおりである。
  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
- 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`-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 -LSEND` を指定した[3]。
- 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 10: 性能評価の条件

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

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
```

図 11: 最適化の効果の評価に用いるプログラム

5 性能評価

以上で述べた専用通信ライブラリ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、個々の最適化手法の効果、簡単な数値処理プログラムの並列化に対する有効性、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[5] を用いた。評価の条件を図 10に示す。

5.1 各最適化手法の効果

まず最初に個々の最適化手法の効果を示す。評価には図 11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 12に示す。ここで最適化レベルは図 10の番号を用いて示されている。

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

---

[3] ただし行列積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 -NOLSEND` を指定した。

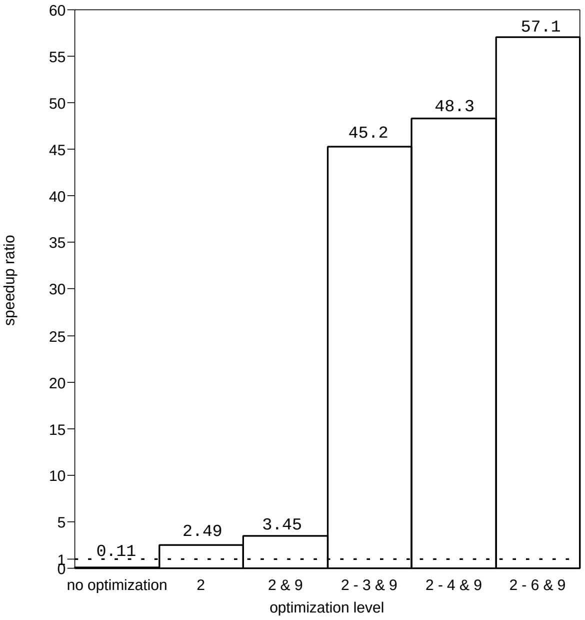

図 12: 各最適化手法の効果

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 0.99 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 57.1 倍であったが、この最適化を外した場合には 31.1 倍であった。このときプロセッサの平均稼働率は 84.5%から 55.7%へと低下していた[4]。

なお、評価プログラムが短かったため、`SEND`文のリスケジューリングによる効果は現れなかった。また、削除の対象となるような不用な`SEND`/`RECV`文も生成されなかった。

5.2 数値処理プログラムによる評価

次に TINPAR による並列化の有効性を示すために、行列積、ガウス消去法および SOR 法のプログラムを並列化した場合の台数効果を測定した。なおこれらのプログラムには、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/利用などの、前処理部で行われるべきプログラム変換が人手で施されている。

5.2.1 行列積

評価には図 13のプログラムを用いた。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。また、配列`local_b()`はリモートデータ`b()`の一時的なローカルコピーを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の台数効果を図 14に示す。プロセッサ数を増加させると 1 プロセッサ当たりの演算量が減少して並列化効率が低下するが、まずまずの性能が得られているといえる。

```
const n = 512
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 */

for i = 0, n - 1 do
  for j = 0, n - 1 do
    /*  リモートデータのローカルコピーを作成する。 */
    local_b(i, j) = b(i, j)
  endfor
endfor
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 13: 行列積のプログラム

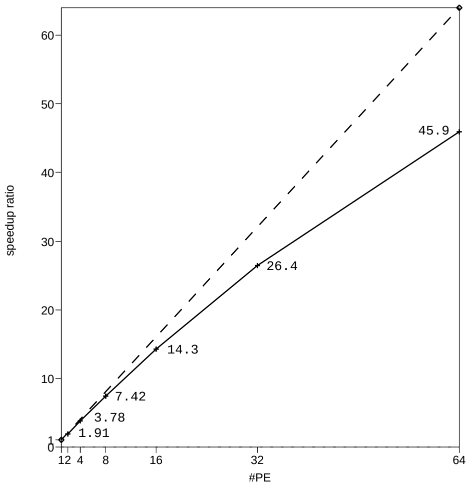

図 14: 行列積の台数効果

5.2.2 ガウス消去法

評価には図 15のプログラムを用いた。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[5]。また配列`local_idamax()`および`local_dmax()`は、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ために導入した配列である。一方配列`local_a()`は、各プロセッサで分解列の一時的なローカルコピーを格納するための配列である。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。

このプログラムを並列化した場合の台数効果を図 16に示す。プロセッサ数を増加させると 1 プロセッサ当たりの演算量が減少して並列化効率が低下するが、まずまずの性能が得られているといえる。

5.2.3 SOR 法

評価には図 17のプログラムを用いた。ここで配列`a()`は 1 次元めでブロック分割されている。また`outer_j`のループは、タイリング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。

---

[4] トレース情報を用いて計算されるため、加速率の比と完全には一致していない。

[5] 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。

```
const n = 1024
real a(0:n - 1:1, 0:n - 1)
integer ipvt(0:n - 1:1)
integer info
temporary integer local_idamax(0:GET_N_CELL() - 1:1)
temporary real local_dmax(0:GET_N_CELL() - 1:1)
temporary real local_a(0:GET_N_CELL() - 1:1, 0:n - 1)
temporary integer l
temporary real abs_a_k_i, dmax, pivot, t
/*  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)), GET_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 15: ガウス消去法のプログラム[6]

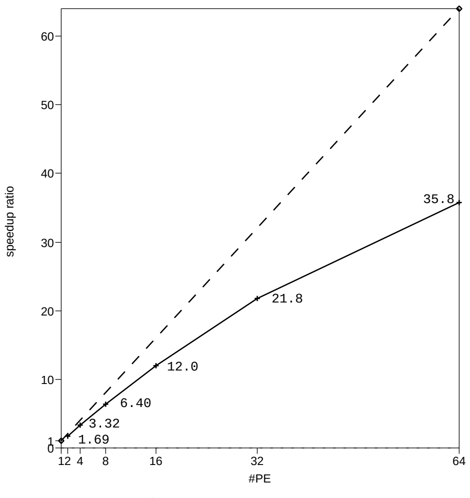

図 16: ガウス消去法の台数効果

```
const n = 1024
const width = 6
real omega
real a(0:n - 1:*, 0:n - 1)
/*  a()は1次元めでブロック分割 */

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

図 17: SOR法のプログラム

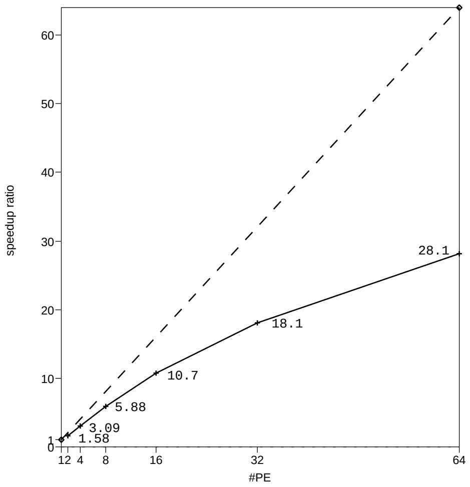

図 18: SOR法の台数効果

このプログラムを並列化した場合の台数効果を図18に示す。プロセッサ数を増加させると1回の通信当たりの演算量が減少して並列化効率が低下するが、ある程度の性能が得られているといえる。

5.3 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バッファのフラッシュを行っている。

評価には前節の3つのプログラムを64プロセッサ用に並列化したものを用いた。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19に示す。

この結果より、通信以外の演算の実行時間はほぼ1となっており、非常に効率がよいといえる。これは最適化によって不用なコードが十分削除されていることを示している。

一方通信ランタイムシステムの実行時間は0.1から0.6であり、あまり効率がよくない。しかしガウス消去法の場合は、通信を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ことによる、送信データパッキングのオーバーヘッドの増大が原因である。またSOR法の場合は、演算量と通信の粒度のバランスがとれてい

---

[6] LINPACKパッケージのdgefaサブルーチンと等価である。

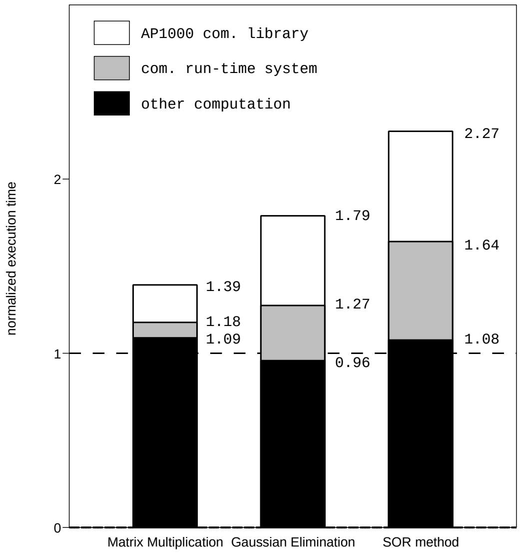

図 19: オブジェクトコードの実行時間の内訳

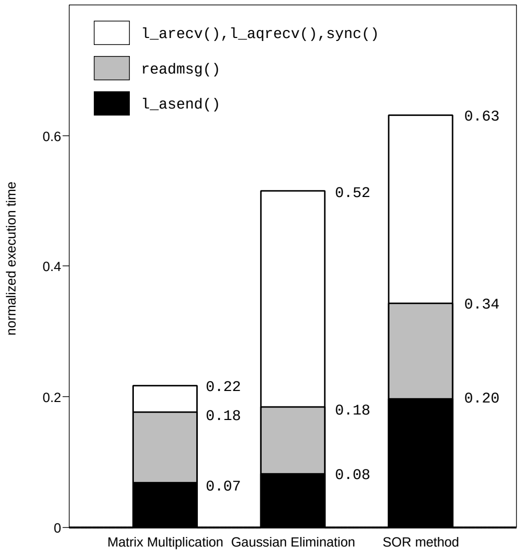

図 20: AP1000 の通信ライブラリの実行時間の内訳

ないために、通信ランタイムシステムのオーバーヘッドが表面化しているものと考えられる。これに対して行列積の場合は、十分効率がよいといえる。

また AP1000 の通信ライブラリの実行時間は 0.2 から 0.6 であり、効率がよいとはいえない。しかし図 20 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く、改善の余地がある。すなわち先に述べたように、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、受信は通信ランタイムシステム内へ直接に行うことができれば、これらの時間をかなり短縮できると考えられる。

6 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[6] が挙げられる。Fortran D 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 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。

また文献 [8] 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、我々の手法と異なり適用範囲に制限がある [2]。

さらに文献 [7] 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。

7 まとめ

本稿では、現在我々が開発中である、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。

また本稿では、TINPAR による並列化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 45.9 倍、ガウス消去法で 35.8 倍、SOR 法で 28.1 倍の加速率が達成された。

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.