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 —

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
〒 606-01 京都市 左京区 吉田本町

E-mail: {miyoshi, maeyama, sigoto, moris, nakasima, tomita}@kuis.kyoto-u.ac.jp

内容梗概

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。このとき TINPAR は、不用なコードの削除や通信の最適化を行うことにより、効率のよいオブジェクトコードを生成する。本稿では、TINPAR で用いられているこれらの最適化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することによってその有効性を示す。この結果 64 プロセッサの AP1000 を用いて、行列積で 42.3 倍、ガウス消去法で 29.6 倍、SOR 法で 27.6 倍の加速率が達成された。

TINPAR:
A Parallelizing Compiler for Message-Passing Multiprocessors
— Optimization Techniques and Performance Evaluation —

Ikuo Miyoshi, Koji Maeyama, Shin-ya Goto,
Shin-ichiro Mori, Hiroshi Nakashima, Shinji Tomita

Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606-01 Japan

E-mail: {miyoshi, maeyama, sigoto, moris, nakasima, tomita}@kuis.kyoto-u.ac.jp

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 order to derive efficient codes, it eliminates unnecessary codes and optimizes interprocessor communication. In this paper, we describe optimization techniques and show performance evaluation results for generated codes. TINPAR achieved 42.3 times speedup for 512×512 matrix multiplication, 29.6 times for 1024×1024 matrix Gaussian elimination, and 27.6 times for 1024×1024 SOR method on the 64 PE AP1000.

1 はじめに

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]。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。

TINPAR による並列化処理は以下の 3 ステップで構成される。

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の最適化を行う。

このようにして TINPAR 内部の処理の独立性を高めることにより、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、これらのうちの最適化のステップで用いられる種々の最適化手法について述べるとともに、各最適化手法の効果および簡単な数値処理プログラムの並列化に対する有効性について述べる。

2 TINPAR の概要

本章では、次章以降の準備として TINPAR の概要について述べる。

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[2] 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 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。この専用通信ライブラリは、効率のよい通信を実現するために動的ベクトル化の機能を持つ。動的ベクトル化機能とは、ランタイムシステム内にデータ送信用のバッファを設け、実行時に短いメッセージを長いメッセージにまとめて送信する機能であり、通信コストを低減することができる。

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、スカラエクスパンジョン、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。

   (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良くない。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。具体的な手法については次章で述べる。

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。

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 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、およびそれらの適用順序について述べる。

3.1 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、ランタイムオーバヘッドを削減するための最適化手法について述べる。

3.1.1 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 1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台数に反比例して低くなる。

そこで本最適化では、ループの実行範囲をあらかじめ狭めることによって、そのように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、または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 $index \times a + b$ 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 $BlockSize$、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。

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

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

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

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

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

   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。

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

```
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 1: ループの実行範囲の縮小

さらに、

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

とすると、この等式を満たす $index$ は、

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

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

で与えられる。

ここで $0 \leq \alpha' < Stride$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、$Stride$、$BlockSize$、$Offset1$、$Offset2$ 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 $BlockSize \leq \alpha < Stride$であるから、

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

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、$Stride$、$BlockSize$ 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 $Offset1$ と $Offset2$ の組から定まるα'の範囲の和集合を求め、$Offset1' \leq \alpha' < Offset2'$となる $Offset1'$と $Offset2'$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、

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

   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。

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

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 $Offset1'$と $Offset2'$を用いて以下のようになる。

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

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。

3.1.2 恒真/恒偽となるif 文の削除

ループの実行範囲の縮小を行った結果、図 2上のように条件式が恒真または恒偽となるif 文が生じる場合がある。

そこで本最適化では、コンパイル時にif 文の条件式を評価することによって、そのような不必要な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。

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、$Stride$、$BlockSize$ 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。

```
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 2: 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 3: if 文を削除するためのループ分割

3.1.3 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 3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、$Stride$、$BlockSize$ 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 $Offset1$ と $Offset2$ の組から定まるα'の範囲を昇順にソートし、$Offset1'' \leq \alpha' < Offset2''$となる $Offset1''$と $Offset2''$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 $Offset1''$ の値がすべて等しい場合には一番小さい $Offset2''$、それ以外の場合には二番目に小さい $Offset1''$と一番小さい $Offset2''$のうちで小さい方とする (図 4 参照)。

$(Offset1'', Offset2'')$ 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 4: if 文を削除するためのループ分割の例

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。

3.2 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。

3.2.1 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5参照)。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。

3.2.2 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 6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 6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ループ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。具体的には、SEND およびRECV 文を含むループを最後のSEND 文の直後で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。

3.2.3 MSG_BUFFER_FLUSH 文の挿入

TINPAR の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、ランタイムシステムが短いメッセージ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、通信コストを低減することができる。

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

図 5: 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 6: 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
MSG_BUFFER_FLUSH_ALL()
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 7: MSG_BUFFER_FLUSH 文の挿入

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、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7参照)。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。

3.3 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 8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。

- SEND 文のリスケジューリング

  ループ分配やif 文の削除によって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。

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. SEND 文のリスケジューリング
8. MSG_BUFFER_FLUSH 文の挿入

図 8: 最適化の適用順序

```
const n = 1024
real a(0:n - 1:*, 0:n - 1)
real b(0:n - 1:*, 0:n - 1)
/* a()、b() は 1 次元でブロック分割 */

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 9: 最適化の効果の評価に用いるプログラム

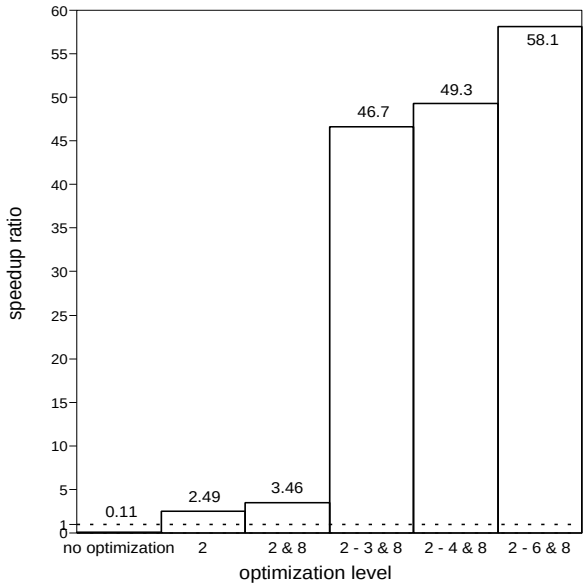

図 10: 各最適化手法の効果

4 性能評価

前章で述べた最適化手法は既に TINPAR に実装されている。そこで本章では、これらの最適化手法の個々の効果、および簡単な数値処理プログラムを並列化した場合の台数効果について述べる。

本章の評価には AP1000[3] を用いた。評価の条件は以下のとおりである。

- 評価には自動的に並列化された TINPAR の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-O4、セルライブラリにはlib.fast を指定した。
- TINPAR の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コールを用い、実行時オプションには-R 0 -LSEND を指定した[1]。
- 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。

4.1 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 9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 10に示す。ここで最適化レベルは図 8の番号を用いて示している。

図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、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

[1]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。

ても加速率は 1.01 であった。またMSG_BUFFER_FLUSH 文の挿入を行わなかった場合、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 58.1 倍であったが、この最適化を外した場合には 31.4 倍であった。このときプロセッサの平均稼働率は、88.3%から 63.0%へと低下していた[2]。

なお、評価プログラムが短かったため、SEND 文のリスケージュリングによる効果は現れなかった。

4.2 数値処理プログラムによる評価

次に本稿で述べた最適化手法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。

4.2.1 行列積

評価には図 11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表 1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。

なおプログラムの前半でグローバルなb() からローカルなlocal_b() へのコピーを行っているが、これは同じ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して性能を引き出すために必須である。今後このような最適化は、ソースプログラムのプログラム変換として前処理部で自動的に行う予定である。

```
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 11: 行列積のプログラム

表 1: 行列積の台数効果

プロセッサ数	2	4	8	16	32	64
加速率	1.91	3.77	7.35	13.9	25.2	42.3

4.2.2 ガウス消去法

評価には図 12のプログラムを用いた。ここで配列a() は 2 次元め、local_a() は 1 次元めでサイクリック分割されている。また、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表 2に示す。プロセッサ数に関わらず、ある程度の性能が得られているといえる。

なおこのプログラムでは、他プロセッサへの分解列の受渡しを明示的にスケジューリングしているが、これはプログラムのクリティカルパスを短くして性能を引き出すために必須である。今後このような最適化を自動化するためには、並列化されたプログラムのどの部分の実行がクリティカルパス上に存在するかを推定することが必要となる。

```
const n = 1024
real a(0:n - 1, 0:n - 1:1)
real local_a(0:GET_N_CELL() - 1:1, 0:n - 1)
/*  a() は 2 次元め、local_a() は 1 次元めでサイクリック分割 */

for k = 0, n - 1 do
  for i = k + 1, n - 1 do
    a(i, k) = a(i, k) / a(k, k)
  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(i, k + 1)), GET_N_CELL() - 1 do
    for i = k + 1, n - 1 do
      local_a(cid, i) = a(i, k)
    endfor
  endfor
  for cid = 0, OWNER(a(i, k + 1)) - 1 do
    for i = k + 1, n - 1 do
      local_a(cid, i) = a(i, k)
    endfor
  endfor
  for i = k + 1, n - 1 do
    for j = k + 1, n - 1 do
      a(i, j) = a(i, j) \
                - local_a(OWNER(a(i, j)), i) * a(k, j)
    endfor
  endfor
endfor
```

図 12: ガウス消去法のプログラム

表 2: ガウス消去法の台数効果

プロセッサ数	2	4	8	16	32	64
加速率	1.20	2.37	4.61	8.76	16.4	29.6

4.2.3 SOR 法

評価には図 13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。

このプログラムを並列化した場合の台数効果を表 3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。

なおこのプログラムでは、ループ運搬依存によって生じる通信の発生回数を減らすために、タイリング技法によってイタレーションの実行をスケジューリングしている。今後このような最適化を自動化するためには、プログラムからループ運搬依存を検出し、イタレーション内の演算量から適切なタイルサイズを推定することが必要となる。

[2] 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。

```
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 13: SOR 法のプログラム

表 3: SOR 法の台数効果

プロセッサ数	2	4	8	16	32	64
加速率	1.54	3.08	5.85	10.6	17.9	27.6

5 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 SUPERB[4] や Fortran D[6] が挙げられる。

SUPERB では、ソースプログラムの各文に mask を適用することで並列化を行っている。文献 [5] では、ループの実行範囲の縮小および恒真/恒偽となるif 文の削除の最適化について述べられているが、ブロック分割の場合しか考慮されておらず、サイクリックおよびブロックサイクリック分割の場合については触れられていない。

Fortran D では、イタレーションセットの概念を用いて演算の分割と通信コードの挿入を行っている。文献 [7] では、ブロックサイクリック分割されている配列に対する演算の分割や通信コードの生成法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、被分割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っている。

VPP Fortran[8] では、インデックスパーティションにもとづく並列化を行っている。文献 [8] ではノンブロッキング送信機能、文献 [9] ではストライドデータ転送機構を活用するコードの生成について述べられている。現在 TINPAR ではこのような高度な通信機能を用いていないが、これらの機能を専用通信ライブラリに採り入れ、これらを活用するような最適化を行うことによって、さらなる性能向上を期待することができる。

6 まとめ

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 は不用なコードの削除や通信の最適化を行うことにより、効率のよいオブジェクトコードを生成する。

本稿では、これらの最適化手法について述べ、その効果を評価した。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 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。

さらに本稿では、これらの最適化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 42.3 倍、ガウス消去法で 29.6 倍、SOR 法で 27.6 倍の加速率が達成された。

謝辞

日頃御討論頂く富田研究室の諸氏に感謝致します。また、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。なお本研究の一部は、文部省科学研究費補助金 (重点領域研究 (1) 課題番号 04235103　超並列ハードウェア ・アーキテクチャの研究」) による。

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

[3] 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.

[4] H. P. Zima, H-J. Bast, and M. Gerndt: SUPERB: A tool for semi-automatic MIMD/SIMD parallelization, Parallel Computing vol.6 no.1, pp.1–18, 1988.

[5] M. Gerndt: Work Distribution in Parallel Programs for Distributed Memory Multiprocessors, Proc. of 1991 Int'l Conf. on Supercomputing, pp.96–104, 1991.

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

[8] 岩下 英俊, 進藤 達也, 岡田 信: VPP Fortran: 分散メモリ型並列計算機言語, JSPP'94 論文集, pp.153–160, 1994.

[9] 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.