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

三吉 郁夫, 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, moris, nakasima, tomita}@kuis.kyoto-u.ac.jp

内容梗概

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を開発している。本処理系では owner comuptes rule を用いて並列化し、必要な通信コードの挿入や効率の良いローカルメモリの割り当てを行う。このためユーザはグローバルメモリ空間を想定して逐次型プログラムを書くことができる。本稿では本処理系の構成、並列化の際に用いる最適化手法、および実際に本処理系によって並列化されたプログラムの性能評価について述べる。本処理系を用いて Livermore Fortran Kernel No.1 を並列化した結果、64 プロセッサで 42.6 倍の加速率が達成された。

A Parallelizing Compiler for Message-Passing Multiprocessor

Ikuo Miyoshi, Shin-ichiro Mori, Hiroshi Nakashima, Shinji Tomita

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

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

Abstract

We are implementing a parallelizing compiler for message-passing multiprocessors. This compiler can parallelize sequential programs with insertion of communication codes and efficient allocation of local memory using owner computes rule. It allows users to write sequential programs using virtual global memory space. In this paper, we describe the overview of the compiler, the used optimization techniques, and the evaluation of the program parallelized with it. This compiler achieved 42.6 times speedup of Livermore Fortran Kernel No.1 with 64 processors.

1　はじめに

現在並列計算機の開発が盛んに行われているが、これらの並列計算機は共有メモリ型とメッセージ交換型に大きく分けられる。共有メモリ型並列計算機は共有メモリプログラミングモデルを提供できる点でメッセージ交換型に比べて有利である。しかしスケーラビリティの面ではメッセージ交換型が有利であるため、この種の並列計算機でグローバルメモリ空間を扱えるプログラミングモデルをサポートすることが望まれる。

そこで本研究では、メッセージ交換型並列計算機上でグローバルメモリ空間を提供するプログラミングモデルを実現するための並列化コンパイラを開発している。本処理系の要件は、共有メモリ型並列計算機用のコンパイラと比較すると以下のとおりである。

- 通信コードの挿入とその最適化

  メッセージ交換型では他のプロセッサのメモリに直接アクセスすることができないため、そのようなアクセスにはメッセージ通信を行う通信コードを挿入しなければならない。またリモートデータをキャッシングするハードウェアを持たないため、リモートデータへのアクセスが減るようにソフトウェアで最適化しなければならない。

- 効率の良いメモリ割り当て

  並列計算機の特徴の 1 つに広大なメモリ空間を利用できる点があげられる。しかしメッセージ交換型ではローカルメモリが完全に独立しているため、データの分割に対してそれぞれのプロセッサで効率良くメモリ割り当てを行わねばならない。

本処理系はこれらを実現するために、処理全体を前処理、並列化、最適化およびメモリ割り当ての 4 つに分け、それぞれを独立させて単純かつ一般的な手法を用いる。これによって本処理系では、アルゴリズムの変更は並列化前、通信の最適化は並列化の前後両方、並列化に密着した最適化は並列化の直後、メモリの割り当てはコード生成の直前というように、適切なタイミングでそれぞれの処理を行うことが可能となっている。

以下本稿では、処理系の概要、用いる最適化手法、本処理系で並列化されたプログラムの性能評価およびまとめと今後の課題について述べる。

2　処理系の概要

2.1　ソースプログラムの記述言語

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[3] そして High Performace Fortran[4] 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。一方コンパイラにとっては、Fortran の言語仕様の固さが種々の解析を行う上で利点となっている。

そこで本処理系では、Fortran の言語仕様に独自の検討を加えた上で、これに類似した構文を持つ新たなソースプログラム記述言語を定義する。これによって本処理系は、ユーザのプログラミングスタイルの連続性と処理系による高い解析能力を両立させる。

本処理系で定義するソースプログラム記述言語は、ループ再構成ツール Tiny[5] で用いられている Tiny language の構文規則をもとに定義したものである。ただし処理系の内部表現は独自のものである。概要を以下に示す。

- 扱えるデータは定数とスカラ/配列変数である。
- 扱えるデータ型は整数型と浮動小数点型である。
- 制御構文には以下のものがある。
  - `if` 条件式 `then` ～ `else` ～ `endif` 構文
  - `for` ～ `endfor` 構文
  - `while` ～ `endwhile` 構文
- 実行文には以下のものがある。
  - 代入文
  - `send` 文
  - `recv` 文
- ユーザ定義による純関数を今後サポートする予定である。

またデータ分割を指定するためのディレクティブもサポートする予定であるが、現時点ではその仕様が確定していないため、処理系の起動時パラメータとして与えている。なお現在可能なデータ分割は、単一次元方向に対するサイクリックまたはブロック分割である。

2.2　オブジェクトプログラムの形式

本処理系は owner computes rule[1] を用い、与えられたデータ分割にしたがって並列化された、SPMD 形式のオブジェクトプログラムを出力する。このオブジェクトプログラムは C 言語で記述されるが、これには以下の利点がある。

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

  ゆえに処理系は並列化に関する処理に専念すればよい。

- 対象とする並列計算機の変更が比較的容易である。

  出力されたオブジェクトプログラムは、対象とする並列計算機上の C コンパイラでコンパイルされ、専用通信ライブラリとリンクされる。そのため対象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よい。

専用通信ライブラリのインターフェースは処理系側で定めることにより、オブジェクトプログラムでの効率の良

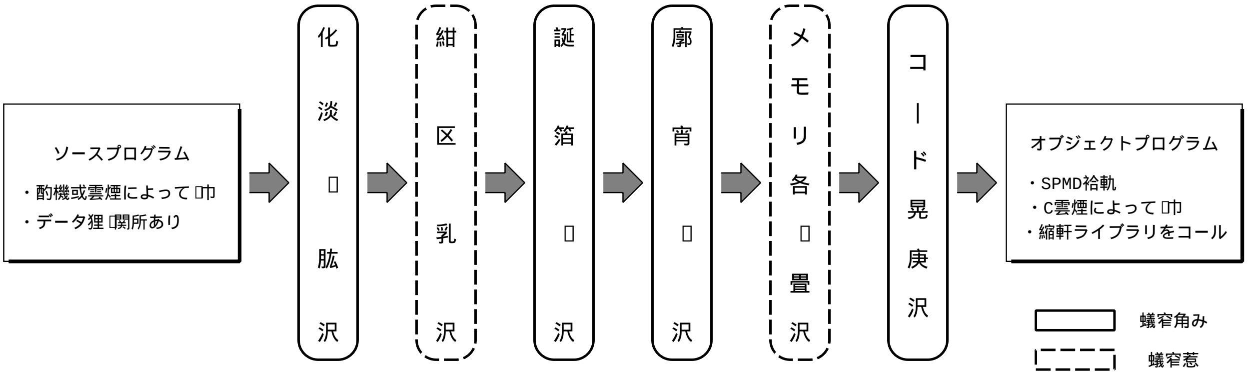

図 1: 処理系の概要

い通信を可能とする。なお現時点のインターフェースでは 1 対 1 通信とバリア同期のみを定めているが、今後はメッセージの動的ベクトル化機能や放送通信機能もサポートする予定である。

2.3 処理系の構成

本処理系は以下の 6 つのモジュールから構成される (図 1参照)。

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

   ソースプログラム中にそのままでは並列化に適さない部分が存在する場合がある。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。現在以下の変換を予定している。

   - スカラエクスパンジョン

     ループ内で同一の変数を一時変数として用いていることによってデータに逆依存関係が生じ、並列化の妨げとなる場合がある。しかしこのような場合には、この一時変数を配列化して逆依存関係を除去することにより、容易に並列化することが可能となる [6]。

   - イディオム変換

     総和演算などを行う逐次プログラムは、一般に並列化に適さない形で書かれることが多い。しかしこれらの演算も本質的には並列実行が可能である。そこでこれらの演算を行うコード、すなわちイディオムをソースプログラムから抽出し、並列化に適したアルゴリズムに変換することによって効率良く並列化することが可能である [2]。

3. 並列化部

   owner computes rule を用い、与えられたデータ分割にしたが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

   分割されて各プロセッサに割り付けら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

   分割されて各プロセッサに割り付けら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` および `endif` 文を挿入する。

   また分割されたデータのメモリ割り当てには、各プロセッサに無限のローカルメモリを仮定し、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。これによって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以下のような最適化を行う。

   (a) 通信コードの移動

   データ転送によるレイテンシを隠蔽するためには、`send` 文の実行から対応する `recv` 文の実行までの時間的距離を長くとる必要がある。そのためには `send` 文をプログラムの上流に、`recv` 文を下流に移動すればよい。これに関する具体的な方法については次章で述べる。

   (b) 無用なコードの削除

   並列化部における並列化の処理は、`if` 文などを挿入することにより、並列化前に比べて

```
const n = 65536
real q, r, t
real x(0:n - 1)
real y(0:n - 1)
real zx(0:n - 1)

for k = 0, n - 12 do
 x(k) = q + y(k) * (r * zx(k + 10) + t * zx(k + 11))
endfor
```

図 2: Livermore Fortran Kernel No.1

プログラムの総演算量を増加させている。そこで並列化による高い性能向上を得るためには、この演算量の増加を最小限に抑えることも重要である。これに関する具体的な方法についても次章で述べる。

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、アドレス変換が不要で効率も良い反面、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

   並列化されたプログラムを処理系の内部表現からC言語によるオブジェクトプログラムに変換し出力する。

3 最適化手法

本章ではLivermore Fortran Kernel No.1を例に、本処理系の最適化手法について述べる。

3.1 準備

用いるソースプログラムを図2に示す。このプログラムのカーネルループには依存関係が存在しないが、データ分割を行って並列化した場合には他のプロセッサの持つデータへのアクセスが必要となる。

これに対して64台による並列化を行った結果が図3である。このとき配列 `x()`, `y()`, `zx()` に用いたデータ分割はブロック分割である。

ここで `SEND()`/`RECV()` は通信ライブラリを用いてデータの送受信を行うマクロであり、`OWNER()` は分割されたデータの所有者プロセッサ番号を返すマクロである。なお `OWNER` マクロは以下のように定義されており、任意ブロックサイズで任意位置から分割を開始する、単一次元方向のサイクリック分割を表現することができる。またブロック分割は、ブロックサイズを⌈分割する次元の配列要素数 ÷ プロセッサ数⌉とした場合に相当する。

```
#define OWNER(NCell, BlockSize, Offset, Index) \
 ((((Index) - (Offset)) / (BlockSize)) % (NCell))
```

`NCell:`	分割に用いるプロセッサ数
`BlockSize:`	分割のブロックサイズ
`Offset:`	分割開始位置のオフセット値
`Index:`	分割された次元の配列インデックス

```
MAIN() {
#define n 65536
 static REAL q;
 static REAL r;
 static REAL t;
 static REAL x[65536];
 static REAL y[65536];
 static REAL zx[65536];
 static REAL tmp_zx_1;
 static REAL tmp_zx_2;
 { /* start of for block */
  INTEGER k;
  for (k = 0; k <= n - 12; k ++) {
   if (OWNER(64, 1024, 0, k + 10) == GETCID()) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 10], REAL);
   } /* endif */
   if (OWNER(64, 1024, 0, k + 11) == GETCID()) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 11], REAL);
   } /* endif */
   if (OWNER(64, 1024, 0, k) == GETCID()) {
    RECV(OWNER(64, 1024, 0, k + 10), tmp_zx_1, REAL);
    RECV(OWNER(64, 1024, 0, k + 11), tmp_zx_2, REAL);
    x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
   } /* endif */
  } /* endfor */
 } /* end of for block */
}
```

図 3: 並列化されたプログラム

図3では並列化の処理により、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` 文と、それらの参照に対応する `SEND`/`RECV` マクロが挿入されている。

3.2 ローカルデータを参照する SEND/RECV マクロの置き換え

図3のプログラムでは、分割されたデータの参照はすべて `SEND`/`RECV` マクロを用いて行われる。しかし所有者プロセッサが参照者プロセッサと同一のプロセッサである場合、すなわちローカルデータを参照する場合には、これらのマクロを用いずに直接参照した方がよい。そこで `SEND`/`RECV` マクロの前後に所有者プロセッサと参照者プロセッサが異なるプロセッサであるかを判定する `if` 文を挿入し、さらに `RECV` マクロ側ではそれらが同一プロセッサである場合に単なる代入操作を行うように変換する。図4にこの最適化の結果を示す。

3.3 SEND/RECV マクロをともに含むループのループ分配

図4のプログラムでは最外側の `for` ループの各イタレーションで `SEND`/`RECV` マクロが実行される。このような場合にはデータ送信の実行からデータ受信の実行までの時間的距離が短く、データ転送待ちによるレイテンシが生じやすい。そこで `SEND`/`RECV` マクロをともに含むループを最後の `SEND` マクロの直後でループ分配し、このようなレイテンシを抑える。図5にこの最適化の結果を示す。

なおこの最適化では、ループ分配によってデータのアクセス順序を変更するため、これによってデータの依存関係が変化しないことが最適化適用の条件となる。

```
{ /* start of for block */
 INTEGER k;
 for (k = 0; k <= n - 12; k ++) {
  if (OWNER(64, 1024, 0, k + 10) == GETCID()) {
   if (OWNER(64, 1024, 0, k) != GETCID()) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 10], REAL);
   } /* endif */
  } /* endif */
  if (OWNER(64, 1024, 0, k + 11) == GETCID()) {
   if (OWNER(64, 1024, 0, k) != GETCID()) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 11], REAL);
   } /* endif */
  } /* endif */
  if (OWNER(64, 1024, 0, k) == GETCID()) {
   if (OWNER(64, 1024, 0, k + 10) != GETCID()) {
    RECV(OWNER(64, 1024, 0, k + 10), tmp_zx_1, REAL);
   } else {
    tmp_zx_1 = zx[k + 10];
   } /* endif */
   if (OWNER(64, 1024, 0, k + 11) != GETCID()) {
    RECV(OWNER(64, 1024, 0, k + 11), tmp_zx_2, REAL);
   } else {
    tmp_zx_2 = zx[k + 11];
   } /* endif */
   x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
  } /* endif */
 } /* endfor */
} /* end of for block */
```

図 4: SEND/RECV マクロの置き換え

```
{ /* start of for block */
 INTEGER k;
 for (k = 0; k <= n - 12; k ++) {
  if (OWNER(64, 1024, 0, k + 10) == GETCID()) {
   if (OWNER(64, 1024, 0, k) != GETCID()) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 10], REAL);
   } /* endif */
  } /* endif */
  if (OWNER(64, 1024, 0, k + 11) == GETCID()) {
   if (OWNER(64, 1024, 0, k) != GETCID()) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 11], REAL);
   } /* endif */
  } /* endif */
 } /* endfor */
} /* end of for block */
{ /* start of for block */
 INTEGER k;
 for (k = 0; k <= n - 12; k ++) {
  if (OWNER(64, 1024, 0, k) == GETCID()) {
   if (OWNER(64, 1024, 0, k + 10) != GETCID()) {
    RECV(OWNER(64, 1024, 0, k + 10), tmp_zx_1, REAL);
   } else {
    tmp_zx_1 = zx[k + 10];
   } /* endif */
   if (OWNER(64, 1024, 0, k + 11) != GETCID()) {
    RECV(OWNER(64, 1024, 0, k + 11), tmp_zx_2, REAL);
   } else {
    tmp_zx_2 = zx[k + 11];
   } /* endif */
   x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
  } /* endif */
 } /* endfor */
} /* end of for block */
```

図 5: SEND/RECV マクロを含むループのループ分配

3.4 ループの実行範囲の縮小

図5のプログラムの 2 つの `for` ループでは、`if` 文によるガードのために実行文を全く実行しないイタレーションが存在する。このような場合にはループの実行範囲をあらかじめ狭め、空回りとなるイタレーションを削除する。用いる最適化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. ループ内の各実行文の実行条件を求める。

   実行文とは、代入文および `SEND/RECV` マクロを指す。実行文の実行条件とは、当該実行文が実行されるための条件を指し、この最適化においては当該ループのループインデックス値の範囲で表す。

   具体的な手法としては、

   $$\begin{aligned}&\texttt{OWNER(NCell, BlockSize, Offset,}\\ &\qquad \text{当該ループのループインデックスの 1 次線形式)}\\ &\texttt{[=!]=} \ \ \text{セル}\, ID\end{aligned}$$

   の形をした `if` 文の条件式を認識し、等式または不等式が成立して `then` 節が実行されるときのループインデックス値の範囲を求める。ただし `NCell`, `BlockSize`, `Offset` は定数とする。

   以下では等式を解く場合について述べる。この等式はループインデックスの 1 次線形式を f_{index} で表して `OWNER` マクロを展開すると、

   $$\begin{aligned}&\lfloor (f_{index} - \texttt{Offset}) \div \texttt{BlockSize} \rfloor \bmod \texttt{NCell}\\ &\quad = \ \ \text{セル}\, ID\end{aligned}$$

   となる。このとき $0 \leq \text{セル}\, ID < \texttt{NCell}$ ならば任意の整数 n に対して、

   $$\begin{aligned}&\lfloor (f_{index} - \texttt{Offset}) \div \texttt{BlockSize} \rfloor\\ &\quad = \ \ \text{セル}\, ID + \texttt{NCell} \times n\end{aligned}$$

   が成り立ち、任意の数$\alpha (0 \leq \alpha < \texttt{BlockSize})$ に対して、

   $$\begin{aligned}&f_{index} - \texttt{Offset}\\ &\quad = \ \ (\text{セル}\, ID + \texttt{NCell} \times n) \times \texttt{BlockSize} + \alpha\end{aligned}$$

   が成り立つ。ここで $f_{index} - \texttt{Offset}$ も当該ループインデックスの 1 次線形式となるから、この等式が成り立つときに当該ループインデックスがとりうる値の範囲を求めることができる。すなわちこの等式を満たすループインデックス値に対して、

   $$\begin{aligned}&\frac{(\text{セル}\, ID + \texttt{NCell} \times n) \times \texttt{BlockSize} + \mathit{offset1}}{f_{index}\text{の 1 次の係数}}\\ &\quad \leq \ \ \text{等式を満たすループインデックス値}\\ &\quad \leq \ \ \frac{(\text{セル}\, ID + \texttt{NCell} \times n) \times \texttt{BlockSize} + \mathit{offset2}}{f_{index}\text{の 1 次の係数}}\end{aligned}$$

   を満たす整数 n が必ず存在するような整数定数 $\mathit{offset1}$, $\mathit{offset2}$ を求めることができる。ただし、$0 \leq \mathit{offset1} \leq \mathit{offset2} \leq \texttt{NCell} \times \texttt{BlockSize} - 1$ とする。

   これにより当該ループを長さ $\texttt{NCell} \times \texttt{BlockSize}$ でストリップマイニングした場合、当該 `if` 文に関して内側ループの実行範囲を $\mathit{offset1}$ から $\mathit{offset2}$ の範囲に縮小できることがわかる。

   なお、不等式の場合も同様に求めることができる。

2. 求めた実行条件の和集合を求める。

   ループ内の各実行文について `NCell`、`BlockSize`、および f_{index}の 1 次の係数の値がすべて等しければ $\mathit{offset1}$ の最小値 $\mathit{offset1}_{min}$および $\mathit{offset2}$ の最

大値 $\mathit{offset2}_{max}$ を求め、さもなくば終了する。この結果得られた $\mathit{offset1}_{min}, \mathit{offset2}_{max}$ により、ループ内の少なくとも 1 つの実行文が実行されるループインデクス値をすべて含む最小かつ連続なループの実行範囲を表すことができる。

3. 当該ループをストリップマイニングする。

   当該ループを長さ $\mathit{stride} = \texttt{NCell} \times \texttt{BlockSize}$ でストリップマイニングする。このとき外側ループの初期値、終了値および増分値を以下のように修正する。

   $$\begin{aligned}&\text{外側ループの初期値}\\&\quad= \left\lfloor\frac{\text{もとの初期値} \times f_{index}\text{の } 1 \text{ 次の係数}}{\mathit{stride}} - 1\right\rfloor\\&\quad\times \mathit{stride} + \texttt{BlockSize} \times \text{セル}\mathit{ID}\\&\text{外側ループの終了値}\\&\quad= \left\lceil\frac{\text{もとの終了値} \times f_{index}\text{の } 1 \text{ 次の係数}}{\mathit{stride}}\right\rceil\\&\quad\times \mathit{stride} - 1\\&\text{外側ループの増分値} \quad = \quad \mathit{stride}\end{aligned}$$

   この修正によって外側ループのループインデックスは、stride の倍数に $\texttt{BlockSize} \times$ セル ID を加えた値をとり、かつすべてのセル ID に対してもとのループの実行範囲を含む最小の範囲で変化する。

4. 内側ループの実行範囲を修正する。

   内側ループの初期値と終了値を以下のように修正する。ただし外側ループのループインデックスを INDEX とする。

   $$\begin{aligned}&\text{内側ループの初期値}\\&\quad= \max(\text{もとの初期値},\\&\qquad\left\lceil\frac{\mathit{INDEX} + \mathit{offset1}_{min}}{f_{index}\text{の } 1 \text{ 次の係数}}\right\rceil)\\&\text{内側ループの終了値}\\&\quad= \min(\text{もとの終了値},\\&\qquad\left\lfloor\frac{\mathit{INDEX} + \mathit{offset2}_{max}}{f_{index}\text{の } 1 \text{ 次の係数}}\right\rfloor)\end{aligned}$$

   この修正により、ループの実行範囲は 2. で求めた範囲となる。なお $\max()$ および $\min()$ は、もとの初期値および終了値をまたぐ外側ループのイタレーションも統一的に扱うために用いられている。

図 6にこの最適化の結果を示す。図 5と比較すると、ループインデックス `k` のループが両方とも `outer_k` と `k` のループにストリップマイニングされており、例えば最初の 2 重ループでは、$\texttt{NCell} = 64, \texttt{BlockSize} = 1024$, f_{index}の 1 次の係数 $= 1, \mathit{stride} = 65536, \mathit{offset1}_{min} = 65525, \mathit{offset2}_{max} = 65535$ である。

3.5 恒真/恒偽となる if 文の削除

図 6のプログラムでは条件式の値が恒真または恒偽である `if` 文が存在する。このような場合には静的に条件式を評価しておくことができる。そこで各 `if` 文について前節と同様に当該条件式が成立するループインデックス値の範囲を求め、ループインデックスのとりうる値の範囲とこれとの積および差集合[1]を計算する。このとき積集合が空であれば恒偽であり、差集合が空であれば恒真である。図 7にこの最適化の結果を示す。図 6と比較すると、2、3、4、5 番目の `if` 文が削除されている。

```
{ /* start of for block */
 INTEGER outer_k;
 for (outer_k = -65536 + 1024 * GETCID();
      outer_k <= 65535; outer_k += 65536) {
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k + 65525);
        k <= MIN(65524, outer_k + 65535); k ++) {
    if (OWNER(64, 1024, 0, k + 10) == GETCID()) {
     if (OWNER(64, 1024, 0, k) != GETCID()) {
      SEND(OWNER(64, 1024, 0, k), zx[k + 10], REAL);
     } /* endif */
    } /* endif */
    if (OWNER(64, 1024, 0, k + 11) == GETCID()) {
     if (OWNER(64, 1024, 0, k) != GETCID()) {
      SEND(OWNER(64, 1024, 0, k), zx[k + 11], REAL);
     } /* endif */
    } /* endif */
   } /* endfor */
  } /* end of for block */
 } /* endfor */
} /* end of for block */
{ /* start of for block */
 INTEGER outer_k;
 for (outer_k = -65536 + 1024 * GETCID();
      outer_k <= 65535; outer_k += 65536) {
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k);
        k <= MIN(65524, outer_k + 1023); k ++) {
    if (OWNER(64, 1024, 0, k) == GETCID()) {
     if (OWNER(64, 1024, 0, k + 10) != GETCID()) {
      RECV(OWNER(64, 1024, 0, k + 10), tmp_zx_1, REAL);
     } else {
      tmp_zx_1 = zx[k + 10];
     } /* endif */
     if (OWNER(64, 1024, 0, k + 11) != GETCID()) {
      RECV(OWNER(64, 1024, 0, k + 11), tmp_zx_2, REAL);
     } else {
      tmp_zx_2 = zx[k + 11];
     } /* endif */
     x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
    } /* endif */
   } /* endfor */
  } /* end of for block */
 } /* endfor */
} /* end of for block */
```

図 6: ループの実行範囲の縮小

3.6 if 文を削除するためのループ分割

図 7のプログラムはループを展開することによってさらに前節の最適化を行える。この最適化はオブジェクトプログラムサイズを大きくさせるものの、実行時の `if` 文の条件式判定をさらに減らすことができる。この最適化ではまず、当該ループ内の各 `if` 文について前々節と同様に条件式が成立するループインデックス値の範囲を求める。このとき `NCell`, `BlockSize`, f_{index}の 1 次の係数の値がすべて等しければ、ループ分割によって `if` 文を削除できる。具体的には $\mathit{offset1}$ の値がすべて等しければ $\mathit{offset1}$ から $\mathit{offset2}$ の最小値まで、等しくなければ $\mathit{offset1}$ の最小値から 2 番目に小さい $\mathit{offset1} - 1$ まで、

---

[1] 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 A、B の差集合という。

```
{ /* start of for block */
 INTEGER outer_k;
 for (outer_k = -65536 + 1024 * GETCID() * 1024;
      outer_k <= 65535; outer_k += 65536) {
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k + 65525);
        k <= MIN(65524, outer_k + 65535)); k ++) {
    if (OWNER(64, 1024, 0, k + 10) == GETCID()) {
     SEND(OWNER(64, 1024, 0, k), zx[k + 10], REAL);
    } /* endif */
    SEND(OWNER(64, 1024, 0, k), zx[k + 11], REAL);
   } /* endfor */
  } /* end of for block */
 } /* endfor */
} /* end of for block */
{ /* start of for block */
 INTEGER outer_k;
 for (outer_k = -65536 + 1024 * GETCID();
      outer_k <= 65535; outer_k += 65536) {
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k);
        k <= MIN(65524, outer_k + 1023); k ++) {
    if (OWNER(64, 1024, 0, k + 10) != GETCID()) {
     RECV(OWNER(64, 1024, 0, k + 10), tmp_zx_1, REAL);
    } else {
     tmp_zx_1 = zx[k + 10];
    } /* endif */
    if (OWNER(64, 1024, 0, k + 11) != GETCID()) {
     RECV(OWNER(64, 1024, 0, k + 11), tmp_zx_2, REAL);
    } else {
     tmp_zx_2 = zx[k + 11];
    } /* endif */
    x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
   } /* endfor */
  } /* end of for block */
 } /* endfor */
} /* end of for block */
```

図 7: 恒真/恒偽となる `if` 文の削除

当該ループをループ分割することを繰り返せばよい。図8にこの最適化を行った後さらに前節の最適化を行った結果を示す。図 7と比較すると、ループインデックス `k` のループが 1 つ目は 1 回、2 つ目は 2 回ループ分割され、かわりにすべての `if` 文が削除されている。

4　評価

本章では前章の最適化の効果を評価するために、本処理系で並列化したオブジェクトプログラムの実行時間を並列計算機 AP1000[7] 上で測定した。

4.1　台数効果

まず最初に並列化による台数効果を測定した。測定条件は以下のとおりである。

- 測定対象には図 2のプログラムを用い、$n = 16k$、$64k$ および $256k$ の 3 通りの場合について測定した。
- 配列 `x()`, `y()`, `zx()` の分割にはブロック分割を指定した。
- 並列化は本処理系で自動的に行い、前章で述べたすべての最適化も行った。
- 本処理系の出力したオブジェクトプログラムの C コンパイルには AP1000 で標準の `cc.cc7` スクリプトを用い、コンパイルオプションには `-O4` および `-CELLLIB lib.fast` を指定した。
- `SEND`/`RECV` マクロには 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 -RB 512` を指定した。
- 台数効果はセルプロセッサによる逐次版の実行時間/並列版の実行時間として求めた。

```
{ /* start of for block */
 INTEGER outer_k;
 for (outer_k = -65536 + 1024 * GETCID();
      outer_k <= 65535; outer_k += 65536) {
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k + 65525);
        k <= MIN(65524, outer_k + 65525); k ++) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 11], REAL);
   } /* endfor */
  } /* end of for block */
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k + 65526);
        k <= MIN(65524, outer_k + 65535); k ++) {
    SEND(OWNER(64, 1024, 0, k), zx[k + 10], REAL);
    SEND(OWNER(64, 1024, 0, k), zx[k + 11], REAL);
   } /* endfor */
  } /* end of for block */
 } /* endfor */
} /* end of for block */
{ /* start of for block */
 INTEGER outer_k;
 for (outer_k = -65536 + 1024 * GETCID();
      outer_k <= 65535; outer_k += 65536) {
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k);
        k <= MIN(65524, outer_k + 1012); k ++) {
    tmp_zx_1 = zx[k + 10];
    tmp_zx_2 = zx[k + 11];
    x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
   } /* endfor */
  } /* end of for block */
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k + 1013);
        k <= MIN(65524, outer_k + 1013); k ++) {
    tmp_zx_1 = zx[k + 10];
    RECV(OWNER(64, 1024, 0, k + 11), tmp_zx_2, REAL);
    x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
   } /* endfor */
  } /* end of for block */
  { /* start of for block */
   INTEGER k;
   for (k = MAX(0, outer_k + 1014);
        k <= MIN(65524, outer_k + 1023); k ++) {
    RECV(OWNER(64, 1024, 0, k + 10), tmp_zx_1, REAL);
    RECV(OWNER(64, 1024, 0, k + 11), tmp_zx_2, REAL);
    x[k] = q + y[k] * (r * tmp_zx_1 + t * tmp_zx_2);
   } /* endfor */
  } /* end of for block */
 } /* endfor */
} /* end of for block */
```

図 8: `if` 文を削除するためのループ分割

図 9に結果を示す。64 プロセッサによる並列化の場合、$n = 256k$ で最高 42.6 倍の加速率を達成している。なお図 8のプログラムは $n = 64k$ として 64 プロセッサで並列化した場合に相当する。

4.2　各最適化の効果

次に各最適化の効果について測定した。測定には図 2のプログラムを 64 プロセッサで並列化したものを用い、最適化の有無を除くと前節と同様の条件で行った。図 10に

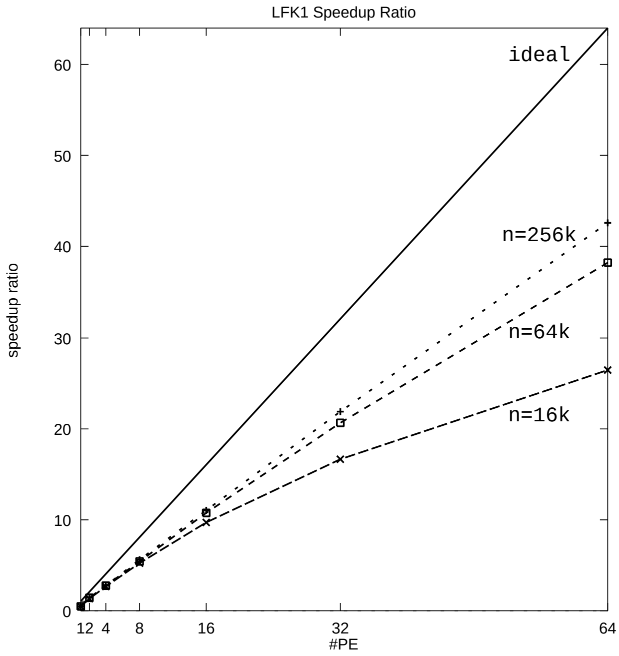

図 9: 並列化による台数効果

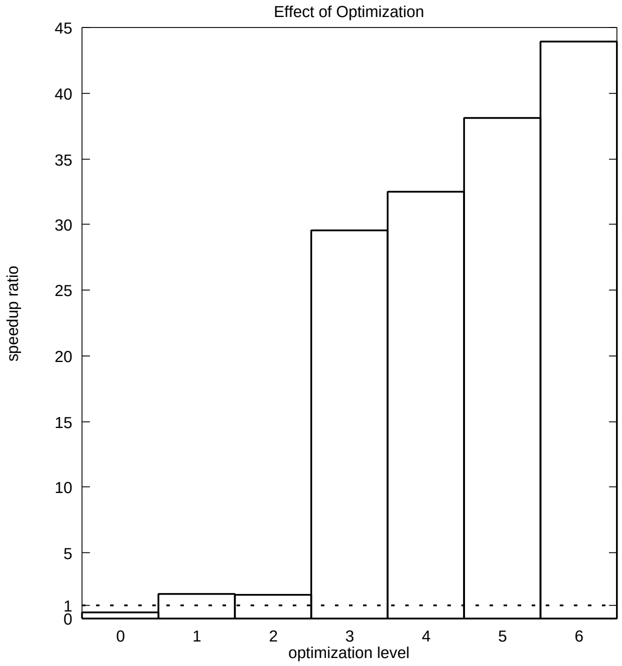

図 10: 各最適化の効果

結果を示す。ここで最適化レベルは表 1のとおりである。

最適化レベルのうち 1. から 5. までは前章で述べたものであり、本処理系に実装済みである。一方最適化レベル 6. で行われる最適化は今後実装予定のものであり、例えば図 8のプログラムにおいて一時変数である `tmp_zx_1` および `tmp_zx_2` への代入操作を削除するものである。

図 10より 3. の最適化が特に効果があることがわかる。これはループの実行範囲を縮小し、実行文の実行がなく結果的に空回りとなるイタレーションを事前に取り除いたことによる効果である。また 4. と 5. の最適化も効果的であるが、これらは `if` 文の条件式を静的に評価したことによる効果である。特に後者は、前処理のループ分割によって一時的にプログラムの総演算量を増加させるものの、結果的には良い効果が得られている。さらに 6. の最適化も効果をあげているが、これは無駄な代入操作を削除してループ内の総実行命令数を削減したことによる効果である。

表 1: 最適化レベルと最適化の内容

0.	並列化のみで全く最適化を行わない
1.	ローカルデータを参照する `SEND/RECV` マクロを代入文に置き換える
2.	1. に加えて `SEND/RECV` マクロをともに含むループをループ分配する
3.	2. に加えてループの実行範囲を縮小する
4.	3. に加えて恒真/恒偽となる `if` 文を削除する
5.	4. に加えて `if` 文を削除できるようにループ分割を行った後、恒真/恒偽となる `if` 文を削除する
6.	5. に加えて一時変数への無駄な代入操作を削除する

なお n = 16k および 256k の場合、最適化レベル 6. でそれぞれ 29.2 倍と 50.3 倍の加速率が得られている。

5 まとめと今後の課題

前章の結果より、現在本処理系では do all 型ループでかつリモートデータへのアクセスが総演算量に比べて少ない問題を、ある程度効率良く並列化できることがわかった。今後はこの種の問題の加速率のさらなる改善とともに、リモートデータへのアクセスが多い問題に対する通信の最適化手法や do serial 型ループを除去するためのイディオム変換手法の検討、メモリ再割当部の実装および各種プログラムでの性能評価を予定している。

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

[2] Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, Proc. of Supercomputing '92, 1992.

[3] Metcalf, M. and Reid, J. 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.

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

[5] Wolfe, M.: Tiny: A Loop Restructuring Research Tool, Oregon Graduate Institute of Science and Technology, 1992.

[6] 笠原 博徳: 並列処理技術, コロナ社, 1991.

[7] 清水 俊幸, 堀江 健志, 石畑 宏明: 高速メッセージハンドリング機構–AP100 における実現–, JSPP'92 論文集, 1992.