並列化コンパイラTINPARにおけるスカラ変数処理

後藤 慎也, 前山 浩二, 三吉 郁夫*, 窪田 昌史,
森 眞一郎, 中島 浩, 富田 眞治

京都大学 工学部　*富士通 (株)

内容梗概

本稿では，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 におけるスカラ変数処理手法について述べる。TINPAR ではスカラ変数は全プロセッサで重複して配置されるため，単純に owner computes rule を適用する場合，全プロセッサでスカラ変数に対する同一の処理が行われる。本手法では，データフロー解析を用いて，不必要なスカラ変数への代入文，通信文を検出し，それらを削除することによって性能の向上を図る。本手法の効果を LIINPACK ベンチマークによって評価したところ，本手法を適用しない場合と比べて 45.7 倍の性能向上が得られた。

Scalar Variable Management in A Parallelizing Compiler TINPAR

Shin-ya GOTO, Koji MAEYAMA, Ikuo MIYOSHI*, Atsushi KUBOTA,
Shin-ichiro MORI, Hiroshi NAKASHIMA, Shinji TOMITA

Faculty of Engineering, Kyoto University　*Fujitsu Ltd.

Abstract

This paper presents a scheme of scalar variable management in TINPAR; a parallelizing compiler for message-passing multiprocessors. In TINPAR scalar variables are mapped onto every processor. Thus with the owner computes rule the same statements related to the variables are executed on each processor. We propose an optimization technique which removes needless statements for assignment or communication with deta-flow analysis. The effect of this technique was evaluated with LINPACK Benchmark Program, and we could obtain the speed up of 45.7 times comparing with the case without the optimization.

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(TINy PARallelizer) を開発している [1, 2]。TINPAR は拡張 Tiny Language により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。拡張 Tiny Language とは，ループ再構成ツール Tiny[3] のために設計された Tiny Language にデータ分割ディレクティブなどを加えたものである。

配列データは各プロセッサに分散配置され，プロセッサ間にまたがるデータ参照は send/recv 文といった通信コードの挿入によって実現される。

ところで，TINPAR の並列化戦略では，スカラ変数は全プロセッサが所有するものとしている。そのため，分割されたデータからスカラ変数への代入などにおいて不要なブロードキャストコードが挿入されてしまうことがある。このような不要な通信は性能を低下させる大きな要因となる。

そこで本稿では，並列化により全プロセッサに対し配置されたスカラ変数に対する不要な通信を削除する最適化手法を提案する。一般の並列化コンパイラでは，不要な通信が発生しないよう，最初に逐次コードに対しデータ依存解析を行い，その情報を用いて最適な並列化を行うという手法を用いる。本手法はこの方法とは逆に，最初に並列化を行った後，コードの解析によって最適なスカラ変数の配置を決定する手法である。

以下第 2 節において並列化コンパイラにおけるスカラ処理について従来提案されている手法及び，我々が提案する手法を概説し，続いて第 3 節では TINPAR におけるスカラ変数処理手法について述べる。その後第 4 節で本手法の評価を行い，第 5 節でまとめる。

2 並列化コンパイラにおける変数処理

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は盛んに研究されている。一般に数値処理プログラムを並列化するコンパイラはプログラムのデータ並列性を利用し，SPMD(Single Program Multiple Data stream) コードを出力することが多い。特に，データを多数のプロセッサに分割し，各プロセッサは分割配置されたデータのための演算を担当するという，owner computes rule と呼ばれる手法が広く用いられている。配列データは，逐次プログラム上で指定された分割手法に従ってプロセッサ間に分散させる。代表的な分割手法としてはブロック分割，サイクリック分割などがあげられる。

```
/* a(i), b(i) は分割されている */
t = b(0)              ··· S1
t = t*t               ··· S2
for i=1, n-1 do
   a(i) = t           ··· S3
endfor
```

(a) 並列化前

```
/* S1 */
if( owner(b(0)) == CELL_ID() ) then
  broadcast( b(0) )
  t = b(0)
else
  recv( owner( b(0) ), t )
endif
/* S2 */
t = t*t
for i = 0, n/N_CELL do
  /* S3 */
  if( owner(a(i)) == CELL_ID() ) then
    a( i ) = t
  endif
endfor
```

(b) 並列化後

図 1: スカラ変数の並列化

2.1 スカラ変数の配置に関する問題

一般にスカラ変数は配列データの場合と異なり，基本的に全プロセッサに重複して配置する戦略がとられることが多い。図 1に例を示す。(a) が逐次コード，(b) が並列化されたコードである。並列化によりスカラ変数 t は全プロセッサにコピーされる。そのため，S_1のように分割された配列からの代入を行う場合，b(0) を所有するプロセッサから全プロセッサへのブロードキャストが挿入される。逆に S_3のように，スカラ変数から分割データへの代入を行う場合，a(i) を所有するプロセッサ内で，t の値の a(i) への代入がローカルに行われる。また，S_2のようなスカラ変数からスカラ変数への代入は，全プロセッサにおいてローカルに代入される。

ところが，スカラ変数を単純に全プロセッサに対して配置すると問題が生ずることがある。例として図 2のループを示す。なお，配列変数 a と b は同じ分割がなされているとする。図 2のループでは，スカラ変数 c は同一イタレーション内ではフロー依存 (ループ独立フロー依存)，前後のイタレーションとの間で逆依存 (ループ運搬逆依存) の関係にある。このため，図 2の並列化の際，これらの変数に対する依存関係を守らなければならな

いため並列実行ができず，このループはほぼ逐次的に実行され，性能が大幅に低下してしまう。

```
for i = 0, n-1 do
  c = b(i)             ··· S_1
  a(i) = a(i) + c      ··· S_2
endfor
```

図 2: ループ独立なテンポラリ変数

2.2 一般的なスカラ最適化手法の手順

2.1節で述べた問題点を解決する手法としてスカラエクスパンション [5, 6] とスカラプライバティゼーション [6] が提案されている。これは，先に述べた 2 つの依存関係のうち，ループ運搬逆依存を解消し並列実行させる手法である。

スカラエクスパンション (scalar expansion) 図 2 のようなループを並列化するために，ベクトル化コンパイラなどではスカラエクスパンションという手法が用いられてきた。これはイタレーション内で一時変数として使用されるスカラ変数を，反復回数分の要素数を持つ配列に拡張し，イタレーション毎に別々の要素を一時変数として用いる。図 2のループに適用したコードを図 3に示す。この処理によりループ運搬逆依存関係が解消されるため，各イタレーションを同時に実行できる。並列化コンパイラにおいても同様にスカラ変数を配列化することで並列化が可能となる。

```
for i = 0, n-1 do
  c_vec(i) = b(i)          /*変数 c を配列に拡張*/
  a(i) = a(i) + c_vec(i)
endfor
```

図 3: スカラエクスパンション

スカラプライバティゼーション (scalar privatization) 並列化コンパイラの場合には，ベクトル化コンパイラのように全イタレーション分の変数に拡張する必要はない。ループ運搬フロー依存がないスカラ変数の場合には，各プロセッサがスカラ変数をローカルなものとみなすことで，プロセッサ間通信をなくすことができる。これはスカラプライバティゼーションと呼ばれる。図 2のコードに対しスカラプライバティゼーションを行ったものを図 4に示す。スカラエクスパンションを並列化コンパイラに応用したものと比較すると，スカラ変数のためのメモリ消費はプロセッサ個数分となるためメモリ効率がよい。

図 2のような逐次プログラムにスカラエクスパンションやスカラプライバティゼーションを適用する場合，以下のような手順が踏まれる。

```
for i = 0, n-1 do
  /*変数 c をプロセッサ固有のものとする*/
  /* (通信を行わない) */
  c_private = b(i)
  a(i) = a(i) + c_private
endfor
```

図 4: スカラプライバティゼーション

1. 逐次コードに対して依存解析を行い，対象となる変数を求める。
2. 依存解析により対象変数に “private” マークを付ける。スカラエクスパンションの場合には逐次コードに対して変更を加え配列化する。
3. 並列化を行う。“private” マークの付いた変数のみプライバティゼーションを行う。

このように，エクスパンションやプライバティゼーションは一般に並列化の前にプログラム依存解析を行い，ソースコードに対してプログラム変換を行う必要があるため，前処理の負担やプライバティゼーションを意識した並列化が求められる。

今回われわれが提案する手法は，以上に述べた手法とは異なり，並列化を行う以前にこのようなデータ依存解析は行わない。つまり，並列化の段階ではスカラ変数は全プロセッサに配置されるようなコードを生成する。その後のコード変形やデータフロー解析によって不要な通信命令を発見し削除することにより，最適なスカラ変数配置を行う。

3 TINPAR におけるスカラ変数処理の最適化

3.1 TINPAR の並列化手法

TINPAR による並列化手法は強力な最適化手法の適用を前提とした単純なものであり，効率のよいオブジェクトコードの生成は並列化後の高度な最適化によって実現される。

TINPAR は以下の 6 つのモジュールから構成される。

構文解析部 ソースプログラムの構文解析を行い，処理系の内部表現に展開する。

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。前処理部ではこのような部分に対してプログラム変換を行うことにより並列化に適した形に変形する。例として総和演算に対するイディオム変換などがあげられる。

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。具体的にはデータを所有するプロセッサを判定するための `if` 文 (以下ガードと呼ぶ) や，プロセッサ間のデータ参照を行うための通信コードの挿入を行う。

最適化部 並列化部で用いられる並列化手法は単純なため，生成コードには無駄が多く効率がよくない。そこで最適化部では，不要なコードの削除，通信コードのリスケジューリングといった最適化を行う。本稿で述べるスカラ変数の最適化処理もここで実行される。

メモリ再割当部 分割されたデータに対して，プロセッサが所有するデータ分だけのメモリ領域を確保するように，メモリの再割り当てとアドレッシングの修正を行う。

コード生成部 処理系の内部表現から C 言語で記述された SPMD コードを出力する。

3.2 スカラ変数処理の概要

2.1節で述べたように，並列化部においてスカラ変数は無条件に全プロセッサに対し配置され，配列変数からスカラ変数への代入は全てブロードキャストにより行われる。例えば，図 2のコードをこの並列化部によって並列化すると図 5の 4 行目のようなブロードキャストが挿入されたコードが生成される。

```
 1  for i = 0, n-1 do
 2    /* c = b(i) */
 3    if( owner( b(i) ) == CELL_ID()) then
 4      broadcast( b(i) )            ··· P1
 5    endif
 6    if( owner( b(i) ) != CELL_ID() ) then
 7      recv( owner( b(i) ), b_tmp) ··· P2
 8    else
 9      b_tmp = b(i)                 ··· P3
10    endif
11    c = b_tmp                      ··· P4
12    /* a(i) = a(i) + c */
13    if ( owner( a(i) ) == CELL_ID() ) then
14      a(i) = a(i) + c              ··· P5
15    endif
16  endfor
```

図 5: 並列化部による出力

TINPAR ではこのような単純に並列化されたプログラムに対し，最適化部によるランタイムオーバヘッドの削減や通信レイテンシの低減を図る処理 (コード変形) が行われる。この処理により `i` のループが図 6のような 2 重ループに変換される。これに伴い，`b(i)` の所有者プロセッサが実行するループ (図 6: 3~5 行目，9~13 行目) と，`b(i)` を所有しないプロセッサが実行するループ (図 6: 15~18 行目) に分割される [1]。

```
 1  for i_out = ... do
 2   /* b(i) の所有プロセッサの処理 */
 3   for i∈ {i|owner(b(i))==CELL_ID()} do
 4    broadcast( b(i) )
 5    b_tmp = b(i)
 6    c = b_tmp
 7    a(i) = a(i) + c
 8   endfor
 9   /* b(i) の所有プロセッサ以外の処理 */
10   for i∈ {i|owner(b(i)) != CELL_ID()} do
11    recv( owner( b(i) ), b_tmp)
12    c = b_tmp
13   endfor
14  endfor
```

図 6: 最適化部によるコード変形

TINPAR におけるスカラ変数処理の最適化は，種々の最適化手法によって図 6のようなコード変形が行われたものに対して行う。このような最適化処理によってループの所有者プロセッサの処理が分割されると，プロセッサ毎のデータの流れの解析が容易になるためである。

例えば，図 6のコードで `b(i)` を所有しないプロセッサの処理について考える。12 行目で `c` に対して代入された値は，これ以後の処理について絶対に使用されないということがデータフロー解析により判断できる。したがってこの `c` に対する `b_tmp` への代入は削除できる。さらに，この `b_tmp` への代入に対する通信 (`broadcast`–`recv`) も不要であることがわかり，通信コードも削除することができる。このようにして図 7のコードが生成される。

```
1  for i_out = ... do /*受信 */
2   /* b(i) の所有プロセッサの処理 */
3   for i∈ {i|owner(b(i))==CELL_ID()} do
4    b_tmp = b(i)
5    c = b_tmp
6    a[i] = a(i) + c
7   endfor
8  endfor
```

図 7: 本手法適用後のコード

3.3 処理手順

TINPAR の並列化部では，ソースコードに記述されたデータの分割指定に従った単純な並列化が行われるため，スカラ変数は全プロセッサが所有するものとされる。

並列化部により並列化されたコードにはガードが多数含まれており，コードの解析を行うことは困難である。そこで，最適化部によるコード変形の一つとしてガードの統合が行われる。この処理は本スカラ変数最適化処理の重要な前処理となっている。

ガードの統合によってプログラムの見通しが良

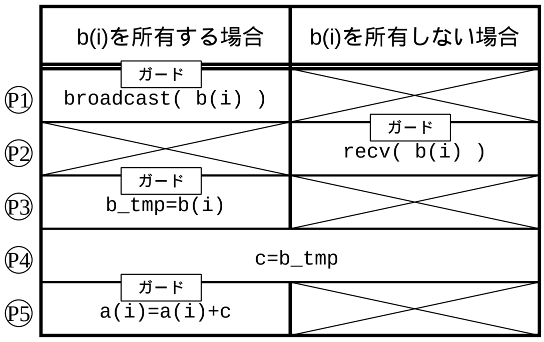


図 8: ガード統合前の各イタレーション

くなった時点でスカラ変数に対するデータフロー解析が行われる。その後，解析によって検出された不要な通信の削除を行う。こうしてスカラ変数処理の最適化がなされる。

以下にガードの統合，データフロー解析，不要な通信の削除についてその手順を述べる。

3.3.1 ガードの統合

並列化されたコードにはガードが多数存在する (図 5: 3,6,13 行目)。このままではコードの見通しが悪いためガードを減らして解析を容易にする必要がある。図 5の各イタレーションの動作を，分割された変数 `b(i)` を所有する場合と所有しない場合に分けて図示すると図 8が得られる。図のように，特定のプロセッサ上でのみ実行させる文の直前にはガードが置かれている。

そこでループを，`b(i)` を所有している時と，所有しない時という 2 つのループのイタレーションの集合に分割することによりイタレーション内部のガードを削除する。このループの分割は，分割データのインデックスを持つループを 2 重ループに変形することによって行う [1]。

この変形によって図 5のコードは図 6となる。図 6の各イタレーションにおける命令の実行を図示すると図 9のようになり，イタレーション内部からガードが削除され，イタレーションの入口に統合されていることがわかる。

3.3.2 データフロー解析

スカラ変数からの代入は，代入文または `broadcast`/`recv` 文により行われる。そこで，コード中の代入文や `recv` 文により定義された値 (左辺値) に対し，その後の実行で値が実際に参照されるかどうかの解析を行う。もし参照がない場合にはその定義は無効なため削除の候補としてよい。以降，代入文と `recv` 文を定義文と呼ぶ。

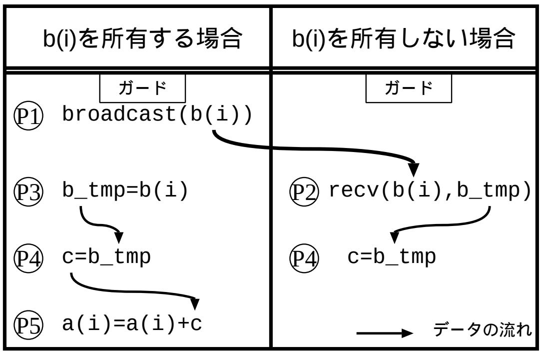


図 9: ガード統合後の各イタレーション

解析の対象となる変数　データフロー解析において対象となる変数はスコープ内で宣言されたスカラ変数とする。

具体的には，ソースプログラムに存在するスカラ変数やコンパイラにより挿入されたスカラ変数が解析の対象となる。前者の例としては図 5の変数 `c` があげられる。後者は，TINPAR により並列化の段階で一時的な値を格納するため挿入された変数で，図 5の 9 行目の `b_tmp` がこれに相当する。

なお，グローバル変数は解析対象としない。

解析の手順　解析は，対象となる変数のそれぞれについて次のような手順で実行される。

1. 定義文以降の文について，解析対象の変数についての参照 (代入文の右辺・`send` 文・`broadcast` 文) または定義 (代入文の左辺・`recv` 文) があるか調べる。
2. 解析中の文が以下の制御文であるならばそれぞれの処理を行う
   - `if` 文: 条件式の内容にかかわらず `then` 節，`else` 節内の文を共に解析する
   - `for` 文: ループの評価式にかかわらずループ内部・ループ以降の文を共に解析する
3. 解析対象となる変数の定義文の集合のうち，以下の条件を満たす定義文を削除の候補とする
   - 定義文の含まれるスコープ中で，その後実行されうるすべての文でその定義が参照も送信もされない
   - 定義文の含まれるスコープ中で，その後実行されうるすべての文で参照されないまま再定義される

3.3.3　不要な通信の削除

データフロー解析により削除の候補とされた代入文や通信コードの削除を行う。代入文の削除は無条件に行うことができるが，通信文の削除は送信と受信の対応関係を崩さないようにしなければならない。

スカラ変数から配列データの代入時に発生する通信は m 個の `broadcast` 文と n 個の `recv` 文の組となる $(m \geq 1, n \geq 1)$。

ここで $n = 1$ の場合，その `recv` 文が削除可能であれば送信文は無条件で削除できる (図 6: 16 行目)。$n \geq 2$ の場合でも，全ての `recv` 文が削除可能な場合には単純に全 `broadcast` 文の削除を行ってよい。ところが，n 個の `recv` 文のうちの一部しか削除可能でなかった場合には問題が生ずる場合がある。これについて以下に述べる。

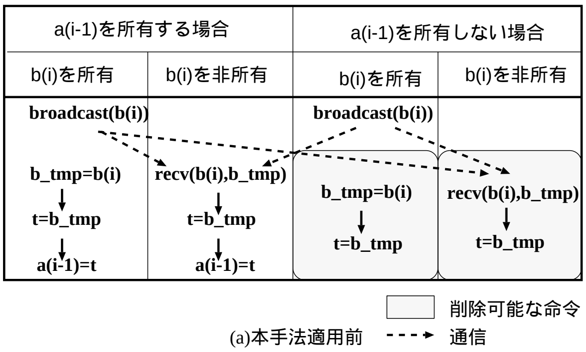
通信コードの削除時に生ずる問題　通信コードを削除する際に注意が必要となる例を図 10に示す。ここで配列の分割方法は図 2と同じであるとする。

図 10のプログラムは図 2と異なり，S'_2においてスカラ変数 `c` は `a(i-1)` に対して代入されるため，`a(i-1)` を所有するかどうかによって S'_2の実行の有無が決まる。そのため，このループでは，`b(i)` の所有者判定と `a(i-1)` の所有者判定の 2 種類のガードがある。ガードの統合を行った時にそれぞれのイタレーション集合において実行される文は図 11(a) のようになる。さらにデータフロー解析を行うと，`a(i-1)` を所有しないプロセッサの代入文および `recv` 文は不要であることがわかる。そこでこれらの文の削除を行うが，`recv` 文に対応する `broadcast` 文は `a(i-1)` の所有者に対しても必要なデータを送信しているため，無条件に `recv` 文および `broadcast` 文の削除をすることはできない。そこで `b(i)` の送信先を `a(i-1)` を所有するプロセッサのみに変更を行う。これによって図 11(b) のような，不要な通信・代入が削除されたコードを得ることができる。

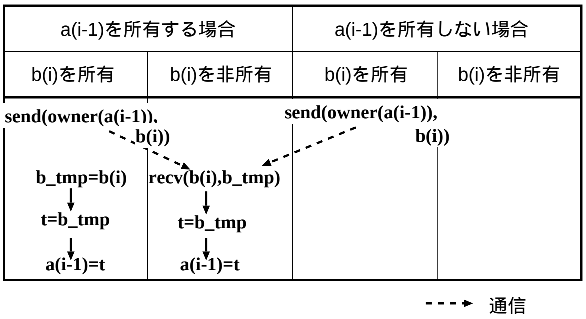
```
for i=1, n-1 do
  c = b(i)                ··· S'1
  a(i-1)= a(i-1) + c      ··· S'2
endfor
```

図 10: 削除において注意が必要となる例

なお，本削除では `broadcast`–`recv` 通信の削除の他に `send`–`recv` 通信の削除も行うことができる。ただし削除が可能となるのは全ての `recv` 文が削除可能となる時のみである。



(a)本手法適用前



(b)本手法適用後

図 11: 図 10 の各イタレーションにおける処理

また，削除によって空ループや空の `if` 文が生じた場合にはこれらも同時に削除する。さらに，不要な定義文の削除により連鎖的に他の定義文が不要となる場合があるため，削除された文で参照されていた全ての変数に対してさらに解析を行い，不要ならば削除を行う。これにより不要な定義を全て削除することができる。

削除の手順　以下に削除の具体的手順について述べる。ただし，普通の代入文と `recv` 文では処理が異なるため，それぞれの場合について手順を示す。

- 定義文が代入文である場合:

  代入文の場合は，データの所有者プロセッサのみが関わるため，削除が他のプロセッサに与える影響は考えなくてよい。そのためこのような定義文は無条件に削除される。

- 定義文が `recv` 文である場合:

  プロセッサ間のメッセージ通信を考慮する必要がある。また，削除によって通信の対応関係を崩さないようにしなければならない。具体的には以下の処理が行われる。

  1. 削除候補の `recv` 文に対応する `send` (`broadcast`) 文の集合 Sを求める

2. Sに対応する `recv` 文の集合 Rを求める
3. R中の全ての `recv` 文のうち必要な文の集合 Xを求める
4. $X = \phi$(全通信が不要) ならばSおよびRを削除する
5. S中の全ての文が `broadcast` である場合，Xに対してのみ送信するような `send` 文に置きかえる

4 性能評価

本スカラ変数処理手法による実際のプログラムでの性能向上の評価を行う。評価は並列計算機 AP1000 を使用し，本手法によって最適化が図られているものといないものについて，加速率を測定することによって行った。

4.1 評価プログラム

本手法の評価は LINPACK ベンチマークプログラム [7] を使用した。これは部分的ピボッティング付きガウス消去法を用いて，連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ の解を求めるプログラムである。本評価プログラムの構造は 2 つのサブルーチン `dgefa`, `dgesl` からなり，`dgefa` において部分的ピボッティング付き LU 分解，`dgesl` において上三角行列の後退代入を行う。

本評価においてデータは以下のように分割した。

- `dgefa` では，行列 A のピボット行の選択や交換など列方向のアクセスが多いため，列方向分割を使用した。また負荷分散を考慮してサイクリック分割を行った。

- `dgesl` では，最内ループが行列 A に対して行方向アクセスを行うため行方向のサイクリック分割とした。そのため `dgefa` と `dgesl` の間に A の再配置のためのコードを挿入した。

4.2 評価プログラムにおけるスカラ変数処理

評価プログラムの実行時間の大部分は `dgefa` ルーチン，つまり部分的ピボッティング付き LU 分解が占めている。このルーチンでは，それぞれの分解列についてピボッティングと消去を行う。

この `dgefa` のピボッティングルーチンを図 12 に示す。ここでプログラム中のスカラ変数 `dmax`，`candidata_a` の処理に対し，本稿で提案する最適化手法が適用できる。

```
 1  /*ピボッティングを行う部分 */
 2  for k=0,n-2 do
 3    /*ピボッティングを行う (k 列目で実行)*/
 4    candidate_a = abs( a(k, k) )
 5    l_dmax(k) = candidate_a /*初期値設定*/
 6    for i = k+1, n - 1 do
 7      candidate_a = abs( a(k,i) )     ··· S1
 8      dmax=l_dmax( k )                ··· S2
 9      /*比較*/
10      if (candidate_a >dmax) then     ··· S3
11        l_dmax(k) = candidate_a       ··· S4
12      endif                           ··· S5
13    endfor
14    /*分解行の消去*/
          :
15        :
16  endfor
```

図 12: ピボッティングルーチン

本評価では行列 A を列分割しているため，ピボット列を持つプロセッサのみがピボッティングに関わることとなる。したがってこの部分はピボット列の所有者プロセッサ上での逐次実行となる。また，ピボッティングの間，他のプロセッサはピボット列のデータが送信されるまで受信待ちになる。そのため，逐次実行部分となるピボッティング部分 (図 12) は性能に与える影響が大きい。

このピボッティングの部分 (図 12: 6 ～ 13 行目) について，ピボット列のイタレーションにおける実行命令と，非ピボット列における実行命令を図 13に示す。図より，2 つのスカラ変数 `dmax`，`candidate_a` への代入のために値がブロードキャストされていることがわかる (S_1,S_2)。しかしデータフロー解析を行うと，非ピボット列ではこれらの値は実際に使用されていないことが分かる。そのため，これらの無用な通信を削除することができ，通信を全く必要としないピボッティングを行うことができる (図 14)。

なお，この部分以外では本最適化手法適用の有無による出力コードの差異はみられなかった。

4.3 評価結果

図 15に行列サイズ 1000×1000 の時の本最適化手法適用あり/なしのそれぞれの加速率を示す。加速率は (逐次実行時間/並列実行時間) として求めた。図 15より，本最適化手法の適用の有無により実行性能にかなりの差があることがわかる。特にプロセッサ数が多い場合には，本手法適用なしのプログラムの実行性能は逐次実行よりも低いものとなった。逆に，本最適化を適用したものは良い性能が得られており，プロセッサが 64 台の時には

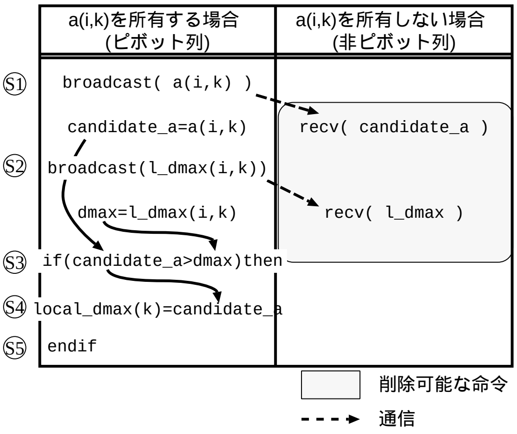


図 13: k列でのピボッティング (本手法適用前)

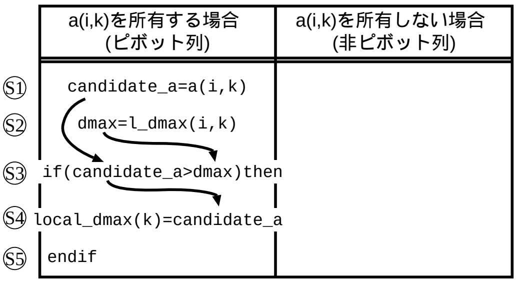


図 14: k列でのピボッティング (本手法適用後)

実に 45.7 倍の性能向上が得られた。

このことより，ピボッティング部分に挿入された不要な放送によるオーバヘッドを本最適化手法の適用により除去することができ，大幅に性能を向上させることができたといえる。

5 結論

本稿では，現在われわれが開発中である並列化コンパイラ TINPAR の，スカラ変数処理手法について述べた。本手法では，並列化時には全プロセッサに対して配置されるスカラ変数に対する不要な通信命令を，コード変換やデータフロー解析を用いて発見し削除する最適化手法である。本手法を利用することにより，並列化の前の依存解析を行う必要がなく，また並列化も単純に行うことができる。

さらに，本手法の効果を LINPACK ベンチマークによって評価した。本手法の適用により，ピボッティング時に現れるスカラ変数に対する不要な通信は削除され，本手法を適用しない場合と比較して 45.7 倍の性能向上が得られた。

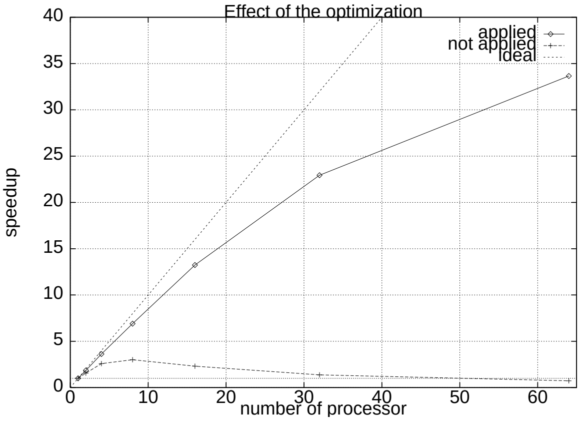


図 15: 評価プログラムに対する本手法の効果

謝辞

日頃より御討論いただく京都大学工学部情報工学教室富田研究室の諸氏に感謝致します。また，並列計算機 AP1000 の実行環境を提供して頂いた (株) 富士通研究所に感謝致します。

参考文献

[1] 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 — 最適化手法と評価 —, 情処研報 94-HPC-54, pp.45-52, 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, JSPP'95 論文集, pp.51-58, 1995

[3] Wolfe,M: *The Tiny Loop Restructuring Research Tool*, Proc. of Supercomputing '89, pp.655-664,1989

[4] Hiranandani,S, Kennedy,K, Tseng,C: *Evaluating Compiler Optimizations for Fortran D*, Journal of Parallel and distributed computing 21, pp.27-45,1994

[5] 笠原 博徳: 並列処理技術 (コロナ社, 1991), pp.122-124

[6] Bacon,D.F, Graham,S.L, Sharp,O.J: *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol.26, No.4, pp.345-420, 1994

[7] Dongarra,J.J: *The LINPACK Benchmark: An Explanation*, Lecture Notes in Computer Science 297, pp. 456-473, 1987