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

後藤 慎也

平成7年2月 提出

内容概況

次世代のスーパーコンピュータとして並列計算機には大きな期待が寄せられている。しかし，並列計算機のためのプログラミングは一般に複雑であるため，逐次言語によって記述されたプログラムを自動的に並列化する並列化コンパイラの実現が求められている。

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。TINPAR は拡張 Tiny Language で記述された逐次プログラムに対し owner computes rule に従った並列化を行う。

しかし TINPAR の並列化戦略では，不必要な送信コードが挿入されてしまうことがある。一般にメッセージ交換型並列計算機は通信にかかるコストが大きく，不必要な通信コードの挿入は性能を大きく低下させる原因となる。したがって不要な通信はできる限り除去されなければならない。

本研究では，データフロー解析にもとづいて不要な通信コードを削除する最適化手法を提案する。本手法では，プログラム中の変数定義の有効性について実行の流れに従って解析し，その結果定義された値が参照されないと判明した場合，その定義文を削除の候補とする。削除の候補となる定義文が単純な代入文であった場合には無条件で削除する。削除の候補が受信文 (RECV 文) であった場合には，それに対応する送信文 (SEND 文) も同時に削除する。この際，RECV 文と SEND 文が1対1に対応している場合には無条件に削除してよいが，それ以外の場合には，当該 SEND 文に対応するすべての RECV 文が削除可能な場合にのみ，これらの文を削除する。これによって SEND/RECV の対応関係を崩さないようにする。

本手法の適用例としては，不要なテンポラリ変数の削除がある。また本手法は，ループ最適化手法の一つであるスカラエクスパンションの代用として，ループの最適化に適用することができる。

本研究では TINPAR に対して本最適化手法の実装を行い，部分的ピボッティング付きガウス消去法のプログラムを用いて並列計算機 AP1000 上で本最適化手法の評価を行った。当該プログラムに対して本手法を適用しないで並列化すると，不要な放送が多数挿入される。そのためプロセッサ数64台での実行性能は逐次実行よりも低いものとなる。これに対して本最適化手法を適用して並列化したプログラムでは，これらの放送は全て削除された。そのため大幅に速度を改善することができ，プロセッサ数64台，行列サイズ 1000×1000 の場合，本最適化を適用しない場合と比べて49.7倍の性能向上が得られた。

An Optimization Technique for Interprocessor Communication on a Parallelizing Compiler

GOTO Shin-ya

Abstract

The parallel computer is expected as the next generation supercomputer. However developing programs for parallel computers is generally difficult. For this reason, there is requirement for parallelizing compilers which translate sequential programs to parallel programs for multiprocessors.

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.

However, unnecessary communication codes may be inserted under the parallelizing strategy of TINPAR. Such needless codes should be removed because high communication cost of message-passing multiprocessors has a bad influence on the execution performance.

This paper presents an optimization technique which removes needless communication codes using dataflow analysis. In this technique, the validness of a variable definition is analized along the execution flow. When the defined value is not used, the definition is useless and regarded as a remove candidate. When the candidate is a simple substitution code, it is removed. But when the candidate is a message-receiving code(RECV), both the RECV and the message-sending code(SEND) must be removed together. In the case when the SEND and RECV have one-to-one correspondence, they can be easily removed, but in the other cases the communication codes can be removed only when all the related RECV codes are useless.

This technique can be applied to remove needless temporary variables. And it can also be applied to optimize loops instead of the scalar expansion, the optimization method used on other compilers.

The technique described in this paper has been implemented to TINPAR. The effect of this optimization technique was evaluated using the program of Gaussian elimination method with pertial pivoting. When we run this program without the optimization, it provides lower performance on 64 processors than the performance of sequential program. This is caused by insertions of many needless broadcast codes. On the other hand, in the parallel program optimized with this technique, all needless broadcast codes were removed. Therefore on 64 processors and with matrix size 1000×1000, by using this technique, we could obtain a speed up of 49.7 times comparing with the same program without using this technique.

1 はじめに

シミュレーションや数値解析などの大規模科学技術演算に用いられるスーパーコンピュータに対し，複雑かつ大規模な演算をより高速に行うためにさらなる演算処理能力の向上が要求されている。そのような中で並列計算機は，現在主流のベクトル計算機に代わる次世代のスーパーコンピュータとして大きな期待が寄せられている。

しかし，並列計算機のためのプログラムの開発には，通信・同期処理などの高度かつ複雑なプログラミング技術が必要となるため，プログラマにとって大きな負担となっているのが現状である。そのため，既存の言語で記述された逐次プログラムを自動的に並列計算機用のプログラムに変換する処理系の開発が望まれている。

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 を開発している [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。

しかし，TINPAR の並列化戦略では，全く参照されることのないデータを送受信してしまう等，不必要な通信コードが挿入されてしまう場合がある。メッセージ交換型並列計算機では一般にメッセージ通信のコストが大きいことを考えると，不要な通信は性能を低下させる大きな要因となる。性能の低下を防ぐためには，このような不要な通信の除去を行うべきである。

そこで本論文では，データフロー解析を用いて不要なメッセージ通信コードを削除する最適化手法を提案する。

以下第 2 章で現在開発中の並列化コンパイラ TINPAR について概説し，続いて第 3 章では本研究で提案する最適化手法について述べる。その後第 4 章で本最適化手法の評価を行い，第 5 章で研究成果のまとめを行う。

2 並列化コンパイラ TINPAR

2.1 TINPAR の概要

TINPAR 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である。TINPAR は owner computes rule を用いて逐次プログラムを並列化する。これによりユーザ記述した単一メモリ空間を想定した逐次プログラムを並列計算機上で実行することが可能となる。

2.1.1 TINPAR の入出力

TINPAR は拡張 Tiny Language によって記述された逐次プログラムを並列化する。拡張 Tiny Language とは，ループ再構成ツール Tiny[3] のための言語 Tiny Language を拡張したプログラム記述言語である。言語拡張としては `while` 構文，データ分割ディレクティブ，スコープルールなどの追加があげられる [2]。

一方，TINPAR の目的コードは C 言語で記述された SPMD(Single Program Multiple Data stream) プログラムである。このため並列計算機で実行するプログラムは，このコードを対象並列計算機用の C コンパイラによってコンパイルし専用通信ライブラリとリンクすることで得られる。このように TINPAR をプリコンパイラとして設計することで，対象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。また，対象計算機の変更も比較的容易となる。

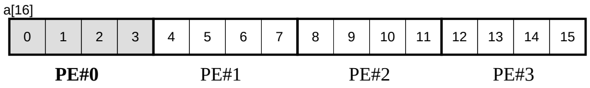
2.1.2 TINPAR の内部構成

TINPAR は以下の 6 つのモジュールから構成される。

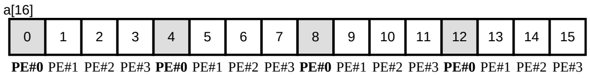
1. 構文解析部—ソースプログラムの構文解析を行い，処理系の内部表現に展開する。
2. 前処理部—逐次プログラムにはそのままでは並列化に適さない部分が存在することがある。前処理部ではこのような部分に対してプログラム変換を行うことにより並列化に適した形に変形する。例として総和演算に対するイディオム変換などがあげられる。
3. 並列化部—owner computes rule に従ってソースプログラムを並列化する。具体的にはデータを所有するプロセッサ (所有者プロセッサ) を判定するための `if` 文や，プロセッサ間のデータ参照を行うための通信コードの挿入を行う。また，この時点では分割されたデータに対して分割前のデータ全体が格納できる大きさのメモリ領域を確保する。このため最適化時には分割前の単純なアドレッシングを使用することができ，データ依存関係解析が容易になる。なお，owner computes rule および並列化手法の詳細については 2.1.3節で述べる。
4. 最適化部—並列化部で用いられる並列化手法は極めて単純なため，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。これらの具体的な内容については 2.2節で述べる。
5. 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。
6. コード生成部—処理系の内部表現から C 言語で記述された目的コードを出力する。

2.1.3 並列化手法

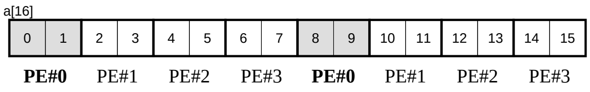
TINPAR による並列化は owner computes rule をその基本戦略としている。owner computes rule とは，『各プロセッサはそれぞれが所有するデータのための演算を担当する』という規則である。さらにこのような規則に従って並列化を行う場合，データを分割し，各プロセッサに対して分配することが必要となる。現在 TINPAR では，単一次元方向に対する任意ブロック長のブロックサイクリック分割を扱うことができる (図 1)。



(a) ブロック長=配列要素数/プロセッサ数 (=ブロック分割)



(b) ブロック長=1 (=サイクリック分割)



(c) ブロック長=2

図 1: ブロックサイクリック分割 (プロセッサ数 4 の時)

このため TINPAR によって並列化されたプログラムでは，各プロセッサは自分の所有するデータのための演算を行う際，自分が所有しないデータに対する参照が起こるとデータの所有者プロセッサとの間でメッセージ通信を行わなければならない。

ここで図 2のプログラムを用いてこの並列化手法を説明する。図 2では配列 `a()`, `b()` に対して異なるデータ分割法が施されている。owner computes rule に従うと，図 2(a)5 行目の代入文は `b(i)` の所有者プロセッサによって実行されなければならない。そのためこの代入文の前後には，所有者プロセッサを

判定するための `if` 文および `endif` 文が挿入される (図 2(b)9 および 12 行目)。一方，右辺のデータ `a(i)` の所有者プロセッサは，左辺のデータ `b(i)` の所有者プロセッサと同一であるとは限らない。そのため `b(i)` の所有者プロセッサが `a(i)` の値を参照するためには，メッセージ通信によってその所有者プロセッサから値を受け取る必要がある。そこでメッセージ受信のための `RECV` 文がこの代入文の直前に挿入される (図 2(b)10 行目)。また，`a(i)` の所有者プロセッサは `a(i)` の値を送信しなければならないため，送信を行う `SEND` 文および `a(i)` の所有者プロセッサを判定するための `if` 文が挿入される (図 2(b)5〜7 行目)。このようにして図 2(b) のように並列化されたプログラムが得られる。

以上まとめると，並列化部では具体的に次のような処理が行われる。

`if` 文の挿入 owner computes rule に従った並列化を行う場合，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，そのようなデータに対する代入文の前後には，所有者プロセッサを判定するための `if` 文および `endif` 文を挿入する。

`SEND`/`RECV` 文の挿入 他プロセッサの所有するデータに対する参照は，その所有者プロセッサとのメッセージ通信によって行われる。そこでそ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` 文を挿入する。さらに `SEND` 文の前後には参照されるデータの所有者プロセッサを判定するための `if` 文を挿入する。

2.2 最適化手法

2.1.3節で述べた並列化手法によって生成されるコードには無駄が多く性能もよくない。そこでこのように並列化されたコードに対し，実行時のオーバヘッドの削減，通信によるレイテンシの低減を図る最適化を施す。また，本研究で提案するデータフロー解析にもとづいて不要な通信命令を削除する最適化手法については第 3 章で述べる。

2.2.1 ランタイムオーバヘッドの削減

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` 文などの，ランタイムオーバヘッドを増加させる文が多数含まれる。そこでこれらのうちの不要なものを削除しランタイムオーバヘッドを削減する。具体的には以下の最適化が行われる。

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(a) のようなループが多数あらわれる。図 3(a) のループは内部で所有者プロセッサの判定を行っており，この条件が成立しない時には無意味なイタレーションを実行することになる。そこでこの条件が成立する範囲にループの実行範囲をあらかじめ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 (図 3(b))。

恒真/恒偽となる `if` 文の削除 ループの実行範囲の縮小により，ループの全イタレーションにおいて条件式が常に真または偽となる `if` 文が生ずることがある (図 4(a))。そこでコンパイル時に `if` 文の条件式を静的評価することで，このように恒真/恒偽となる無意味な `if` 文の削除を行う (図 4(b))。

`if` 文を削除するためのループ分割 ループ中に成立条件の異なる複数の `if` 文が存在する場合，そのままでは恒真/恒偽となる `if` 文の削除を行うことはできない (図 5(a))。しかし，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で条件式を恒真/恒偽に導くことができる。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 (図 5(b))。

```
1  real a(0:n-1:*) /* a(i) はブロック分割 */
2  real b(0:n-1:1) /* b(i) はサイクリック分割 */
3
4  for i = 0, n-1 do
5    b(i) = a(i)
6  endfor
```

(a) 並列化前

```
1  real a(0:n-1:*) /* a(i) はブロック分割 */
2  real b(0:n-1:1) /* b(i) はサイクリック分割 */
3
4  for i = 0, n-1 do
5    if owner( a(i) ) == GET_CELL_ID() then /* if 文の挿入*/
6      SEND( owner( b(i) ), a(i) )        /* SEND 文の挿入*/
7    endif
8
9    if owner( b(i) ) == GET_CELL_ID() then /* if 文の挿入*/
10     RECV( owner( a(i) ), tmp_a )       /* RECV 文の挿入*/
11     b(i) = tmp_a
12   endif
13
14 endfor
```

(b) 並列化後

図 2: TINPAR による並列化

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

(a) 最適化適用前

```
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
```

(b) 最適化適用後

図 3: ループの実行範囲の縮小

```
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
```

(a) 最適化適用前

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

(b) 最適化適用後

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
```

(a) 最適化適用前

```
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
```

(b) 最適化適用後

図 5: `if` 文を削除するためのループ分割

2.2.2 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受信するための待ちによって生ずる。このようなレイテンシの低減を図り，プロセッサの利用率を高める。具体的には以下の最適化が行われる。

`SEND` 文のリスケジューリング 送信されたデータが使用されるまで受信側でバッファリングされるメッセージ交換型並列計算機では，送信側は使用されるよりも早い時期にデータを送ることができる。そこで `SEND` 文をリスケジューリングし，送信処理をできるだけ先行させことで，通信レイテンシを低減することができる。

`SEND`/`RECV` 文をともに含むループの分配 `SEND` 文と `RECV` 文を同時に含む図 6(a) のようなループが得られた場合，送信から受信までの時間的距離が非常に短く，通信レイテンシが生ずる可能性が高いと予想される。そこでこのようなループを送信と受信の 2 つのループに分配することにより，送信から受信までの時間的距離を十分長くすることができ通信レイテンシを隠蔽することができる (図 6(b))。

`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 [4]。動的ベクトル化機能とは，通信ランタイムシステムによって複数の送信データをバッファリングし，まとめて一つのメッセージとして送信する機能である。この機能を用いるとメッセージ送信の起動回数が減少するため効率のよい通信が可能となる。しかし送信バッファを適切にフラッシュしなければ，未送信のデータがバッファに残ったままとなり十分な効果が得られない場合がある。そこで送信バッファをフラッシュするための `MSG_BUFFER_FLUSH` 文を適切な位置に挿入し，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。

3 データフロー解析にもとづく通信最適化手法

3.1 概要

TINPAR は owner computes rule にもとづいた並列化戦略をとる。しかし，この戦略は 2.1.3節で述べられているように単純なものであるため，効率よく通信を行うコードが常に生成されるとは限らない。

このような例としてはスカラ変数に対する代入文があげられる。TINPAR による並列化では，スカラ変数は全プロセッサが所有者となるように配置される。このためデータ分割を施された配列からスカラ変数への代入には，前者の所有者プロセッサが全プロセッサに対して放送を行うコードが挿入される。ところが，このように全プロセッサに値が放送されるにもかかわらず，実際には一部のプロセッサ上でしかその値を必要としない場合がある。そのような場合には，残りのプロセッサへの送信は不要であるため，このような不要なデータの送受信を削除すれば性能を向上させることが可能である。

本最適化手法は，一度も参照されない不要データの送受信を，データフロー解析により発見し削除することによって，オブジェクトコードの最適化を図る。

データフロー解析によって不要なコードを削除する最適化は既存のコンパイラで広く用いられている手法である [5, 6]。これはプログラム中の変数定義の有効性を実行の流れに従って解析することによって判断し，値が一度も参照されないような定義の場合にはそれを無効であるとして削除する手法である。このデータフロー解析を TINPAR の並列化したコードに適用した場合，単なる不要な代入文ばかりでなく不要なデータ送受信までも発見することができる。そこで代入文とともに不要な通信文も削除すれば，無駄な通信のない効率のよいプログラムが生成される。すなわち本最適化手法は，一般のコンパイラで用いられるデータフロー解析による無用文削除の手法を並列化コンパイラに応用することによって通信の最適化を行うものであるといえる。

3.2 適用例

本最適化手法が適用できる例について，以下に 2 つのプログラムを示し説明する。

テンポラリ変数の削除 図 7は配列 `a()` の全ての要素の絶対値を求めるプログラムである。また，これを TINPAR によって並列化すると図 8のプログラムが得られる。ただし配列 `a()` はサイクリック分割されているものとする (図 9)。なお，第 2 章で述べた最適化手法は全て適用されている。また，`if` 文の条件節内での配列要素 `a(i)` の参照はテンポラリ変数 `tmp` を介して行われる (図 8: 8, 16 行目)。

図 8では，ループの実行範囲の縮小や恒真/恒偽となる `if` 文の削除，`if` 文を削除するためのループ分割といった最適化手法の適用により，`i` のループは `a(i)` が所有されているか否かによって 2 つのループに分割されている。また owner computes rule に従って並列化されているため，`a(i)` に対する代入文は `a(i)` が所有されている場合に対応する前半のループにのみ存在し，後半のループからは削除されている。

しかし，`if` 文による条件判定に使用されるテンポラリ変数 `tmp` はスカラ変数であり，所有者は全プロセッサであるため，`tmp` に対する `a(i)` の値の受信は後半部に残されたままである (図 8: 16 行目)。そのため全プロセッサに対する `a(i)` の値の放送が行われ，図 10(a) のようにほぼ逐次的にしか実行されず効率が悪い。

ところが，変数 `tmp` に対してデータフロー解析を行うと，16 行目で受信した値はのちに参照されないことがわかるため，`RECV` 文とそれに対応する `SEND` 文は不要であるとして削除できる。これにより通信を全く必要としない図 11のプログラムが得られる。このプログラムでは，図 10(b) に示されるように各プロセッサは完全に並列に動作する。

スカラエクスパンションの代用 図 12に示すプログラムは Livermore Kernel No.10[7, 8] である。これは $i=\{0,...,n-1\}$，$j=\{4,...,13\}$ について差分予測子 (difference predictor)

$$PX(i,j)=\begin{cases} CX(i,4), & j=4,\\ CX(i,4)-\displaystyle\sum_{k=4}^{j-1}PX(i,k), & j>4\end{cases}$$

を求めるプログラムである。このループにおいて変数 ar, br, cr 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 12をこのまま並列化した場合，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，逐次実行と同じ順序で実行しなければならない。そのためこのループはほぼ逐次的に実行される。

このようなプログラムを並列化するための従来の最適化手法としてはスカラエクスパンションがある [9, 10]。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。これによりループ運搬逆依存関係が解消されるため，各イタレーションが並列に動作できるようになる。例えば図 12にスカラエクスパンションを適用したプログラムは図 13のようになる。しかし，スカラエクスパンションを適用するためには，前処理部でソースプログラムの解析を行い，さらに図 13のようなプログラム変換を行う必要がある。また，スカラ変数をベクトル変数に拡張することによる余分なメモリ消費も問題となる。

しかし，本最適化手法はこのような場合にも効果的な最適化を行うことができる。図 14は図 12を TINPAR により並列

```
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
```

(a) 最適化適用前

```
/*送信のためのループ*/
for i = 0, n-1 do
  if owner( a(i) ) == GET_CELL_ID() then
    SEND( owner( b(i)), a(i))
  endif
endfor

/*受信のためのループ*/
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
```

(b) 最適化適用後

図 6: SEND/RECV 文をともに含むループの分配

```
const n=1048576
real a(0:n-1)             /*配列 A の宣言*/
for i=0, n-1 do
  if( a(i) < 0.0) then    /*条件判断*/
    a(i) = -1.0 * a(i)
  endif
endfor
```

図 7: 配列要素の絶対値を求めるプログラム

```
for i_outer = ... do

 /*所有する a(i) に対する処理*/
 for i ∈ {i|owner(a(i))==GET_CELL_ID(), 0≤i ≤n-1} do
  /* a(i) を送信 */
  BROADCAST_REAL(a(i))
  tmp = a(i);
  if ( tmp < 0.0) then
   a(i) =  - 1.0 * a(i)
  endif
 endfor

 /*所有しない a(i) に対する処理*/
 for i∈ {i|owner(a(i))!=GET_CELL_ID(),0≤i ≤n-1} do
  /* a(i) を受信 */
  RECV_REAL(owner( a(i)), tmp)
  /* if~は消去された */
 endfor
endfor
```

図 8: コンパイラ出力結果 (概略)

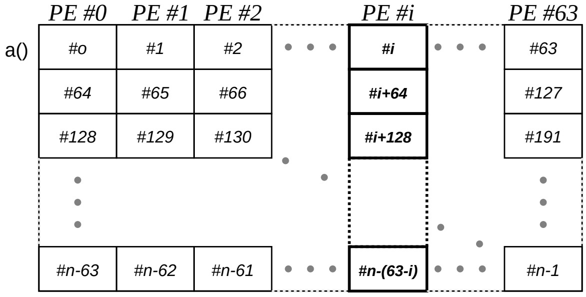
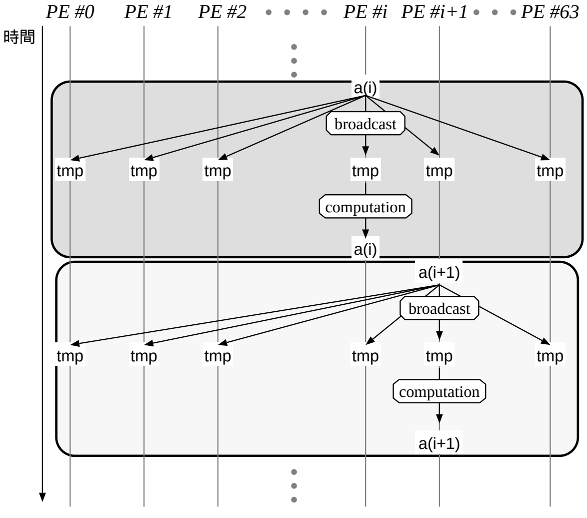
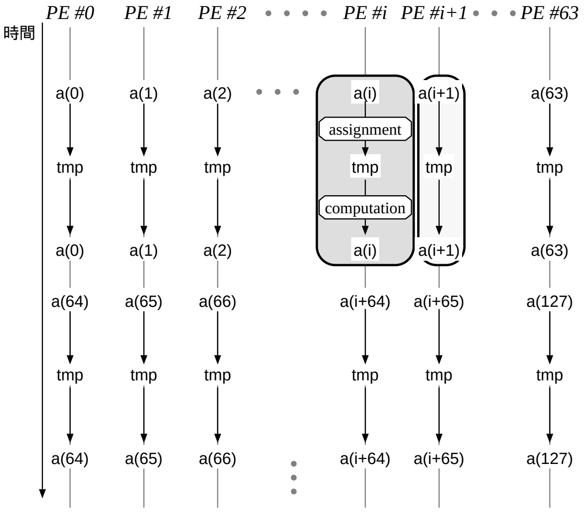


図 9: 配列 a() のデータ分割方法



(a)本最適化手法適用前



(b)本最適化手法適用後

図 10: 計算の流れ (絶対値を求めるプログラム)

```
for i_outer = ... do

 /*所有する a(i) に対する処理*/
 for i ∈ {i|owner(a(i))==GET_CELL_ID(), 0≤i ≤n-1} do
   tmp = a(i)
   if ((tmp < 0.0)) then
     a(i) = - 1.0 * a(i)
   endif
 endfor

  /*所有しない a(i) に対する処理は削除された*/
endfor
```

図 11: 最適化されたコード

化した出力である。スカラ変数 ar, br, cr の所有者は全プロセッサであるため，これらの変数に対する値の代入は，配列要素の所有者プロセッサから全プロセッサへの放送により行われる (5~10 行目)。しかし，これらの変数に対しデータフロー解析を行うと，受信側のプロセッサはこれらの変数の値を参照しておらず，非所有者プロセッサへの ar, br, cr の放送は削除できることがわかる。この結果これらの通信コードを削除すると図 15のように通信が全く行われないプログラムが得られる。

3.3 最適化手順の詳細

本最適化手法はデータフロー解析，無用文の削除の 2 つの処理から構成される。データフロー解析では，定義文 (代入文や RECV 文) によって定義された値に対する参照の有無について解析を行い，参照がない場合には当該定義文を削除の候補とする。無用文の削除では，データフロー解析により削除の候補とされた定義文の削除を行うが，定義文が RECV 文の場合には送信/受信の対応関係を崩さないよう適切に削除する。

以下にデータフロー解析，無用文の削除の詳細について述べる。

3.3.1 データフロー解析

代入文や RECV 文により定義された値に対し，その後の実行で値が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であるとして削除の候補とする。

解析の対象とするプログラム 制御フローを単純化しデータフロー解析を容易にするため，本データフロー解析では，第 2 章で述べた TINPAR による最適化手法が既に適用されたコードを前提とする。

解析の対象となる変数 本データフロー解析では，現在のところスカラ変数のみを対象とし，配列変数に対しては解析の対象としない。

スカラ変数は最終的な結果を格納するものと，テンポラリ変数として一時的な値を格納するものの 2 種類に分けることができる。このうち，最終値格納用の変数に対する定義は削除できないため，データフロー解析の対象とはしない。

データフロー解析の具体的な対象は以下に述べる変数である。

1. コンパイラにより挿入されたスカラ変数
   TINPAR では並列化の段階で一時的な値を格納するためいくつかの変数を挿入する。図 2の例では，他プロセッサからの代入時に RECV 文 (図 2(b)10 行目) と代入文 (図 2(b)11 行目) の間で一時的に値を格納するために a_tmp が使用されている。このようにコンパイラが独自に挿入した変数は，プログラム内で一時的にしか用いられず最終値を保持しておく必要もないため，削除を行ってもかまわない。そこでこのような変数は全て解析の対象として扱う。
2. スコープ内で宣言されたローカルなスカラ変数
   拡張 Tiny Language は静的スコープルールにもとづいたブロック構造を持ち，スコープ内で宣言された変数はローカル変数として扱う。さらに TINPAR では，最終値格納用の変数はグローバル変数として，一時格納用の変数はローカル変数として宣言されることを前提としている。このように宣言方法によって変数の使用目的を明確にすることは最適化を行う上でも妥当なことであるといえる。
   ローカル変数の生存範囲は変数を宣言したスコープ内のみであり，スコープ外から参照されることはない。したがって，スコープ内の解析により不要であると判断されたローカル変数は無条件かつ安全に削除することができる。また，解析はスコープ内を対象とするだけでよく，解析時間を短縮することも可能となる。なお，1 で述べたテンポラリ変数としてもローカル変数が用いられる。

```
/*px, cx は 1 次元目を分割 */
real cx(0:n - 1:*, 0:4),px(0:n - 1:*, 0:13)
real ar,br,cr /*テンポラリ変数として使用 (スカラ変数)*/
for i = 0, n - 1 do
  ar = cx(i, 4)
  br = ar - px(i, 4)
  px(i, 4) = ar
  cr = br - px(i, 5)
  px(i, 5) = br
  ar = cr - px(i, 6)
  px(i, 6) = cr
  br = ar - px(i, 7)
  px(i, 7) = ar
  cr = br - px(i, 8)
  px(i, 8) = br
  ar = cr - px(i, 9)
  px(i, 9) = cr
  br = ar - px(i, 10)
  px(i, 10) = ar
  cr = br - px(i, 11)
  px(i, 11) = br
  px(i, 13) = cr - px(i, 12)
  px(i, 12) = cr
endfor
```

図 12: Livermore Kernel No.10

```
/*px, cx は 1 次元目を分割 */
real cx(0:n - 1, 0:4),px(0:n - 1, 0:13)
/*テンポラリ変数として使用*/
/*スカラエクスパンション適用 (テンポラリの配列化)*/
real ar(0:n-1),br(0:n-1),cr(0:n-1)
for i = 0, n - 1 do
  ar(i) = cx(i, 4)
  br(i) = ar(i) - px(i, 4)
  px(i, 4) = ar(i)
  cr(i) = br(i) - px(i, 5)
  px(i, 5) = br(i)
  ar(i) = cr(i) - px(i, 6)
  px(i, 6) = cr(i)
  br(i) = ar(i) - px(i, 7)
  px(i, 7) = ar(i)
  cr(i) = br(i) - px(i, 8)
  px(i, 8) = br(i)
  ar(i) = cr(i) - px(i, 9)
  px(i, 9) = cr(i)
  br(i) = ar(i) - px(i, 10)
  px(i, 10) = ar(i)
  cr(i) = br(i) - px(i, 11)
  px(i, 11) = br(i)
  px(i, 13) = cr(i) - px(i, 12)
  px(i, 12) = cr(i)
endfor
```

図 13: スカラエクスパンション (Livermore Kernel No.10)

```
for i_outer = ... do

 /* 所有する cx(i,*), px(i,*) に対する処理 */
 for i∈ {i|owner(cx(i))==GET_CELL_ID(),0≤i ≤n-1} do
  BROADCAST_REAL( cx(i, 4) ) /* tmp_a に送信 */
  BROADCAST_REAL( px(i, 4) ) /* tmp_b に送信 */
  BROADCAST_REAL( px(i, 5) ) /* tmp_c に送信 */
  BROADCAST_REAL( px(i, 6) ) /* tmp_d に送信 */
     ⋮
  BROADCAST_REAL( px(i, 11) ) /* tmp_i に送信 */
  ar = cx(i, 4)
  br = ar - px(i, 4)
  px(i, 4) = ar
  cr = br - px(i, 5)
     ⋮
  px(i, 13) = cr - px(i, 12)
  px(i, 12) = cr
 endfor

 /* 所有しない cx(i,*), px(i,*) に対する処理 */
 for i∈ {i|owner(cx(i))!=GET_CELL_ID(),0≤i ≤n-1} do
  RECV_REAL(owner(cx(i, 4)), tmp_a)
  ar = tmp_a       /* ar = cx(i, 4) */
  RECV_REAL(owner(px(i, 4)), tmp_b)
  br = ar - tmp_b  /* br = ar - px(i, 4) */
  RECV_REAL(owner(px(i, 5)), tmp_c)
  cr = br - tmp_c  /* cr = br - px(i, 5) */
  RECV_REAL(owner(px(i, 6)), tmp_d)
  ar = cr - tmp_d  /* ar = cr - px(i, 6) */
     ⋮
  RECV_REAL(owner(px(i, 11)), tmp_i)
  cr = br - tmp_i  /* cr = br - px(i, 11) */
 endofor
endfor
```

図 14: 本最適化適用前 (Livermore Kernel No.10)

```
for i_outer = ... do

  /* 所有する cx(i,*), px(i,*) に対する処理 */
  for i ∈
{i|owner(cx(i,*))==GET_CELL_ID(), 0≤i ≤n-1} do
    ar = cx(i, 4)
    br = ar - px(i, 4)
    px(i, 4) = ar
    cr = br - px(i, 5)
    px(i, 5) = br
    ar = cr - px(i, 6)
    px(i, 6) = cr
    br = ar - px(i, 7)
    px(i, 7) = ar
    cr = br - px(i, 8)
    px(i, 8) = br
    ar = cr - px(i, 9)
    px(i, 9) = cr
    br = ar - px(i, 10)
    px(i, 10) = ar
    cr = br - px(i, 11)
    px(i, 11) = br
    px(i, 13) = cr - px(i, 12)
    px(i, 12) = cr
  endfor

  /* 所有しない cx(i,*), px(i,*) に対する処理 */
  /* ループも含めて削除された */
endfor
```

図 15: 本最適化適用後 (Livermore Kernel No.10)

解析の手順 データフロー解析は，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. 削除の対象となる変数に対する代入文 (a=b など) や RECV 文を検索しリストを作成する。

2. 求められたリストから最初の定義文を取り出し，削除対象となる変数を求め，3 の処理を行う。

3. 現在注目している定義文以降の文について，削除の対象となる変数に対して参照や定義が行われているかを調べる。具体的には定義文の次の文を開始点として以下の処理を行う。

   (a) 解析中の文が代入文または SEND/RECV 文であった場合には，対象変数に対する定義/参照の有無により以下に述べる処理を行う。

   - 参照または送信がある場合 (例: `b[i] = a` や `SEND(a)`)
     現在注目している定義は有効であるとして，現在の削除対象である変数に対する解析を終了する。
   - 定義または受信がある場合 (例: `a = 3` や `RECV(a)`)
     現在注目している定義は無効である (参照されない) として，現在の削除対象である変数に対する解析を打ち切る。
   - 参照と定義の両方がある場合 (例: `a = a + 1`)
     時間的には参照が先であるため，参照時と同様の処理を行う。つまり，現在注目している定義は有効であるとして，現在の削除対象である変数に対する解析を終了する。
   - 参照も定義もない場合: 次に実行される文について 3 の処理を行う。

   (b) 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:
     if 文の条件式で解析対象の変数に対する参照が行われていた場合には，定義は有効であるとして解析を終了する。
     if 文は条件式の値により then 節または else 節を実行するが，本解析では条件式の値にかかわらず then 節と else 節の両方の内部について解析を行う (図 16)。具体的には，then 節，else 節内の先頭の文以降について 3 の処理を続ける。解析により両節ともに無効であった場合のみ endif 文の次の文から 3 の解析を続ける。
   - else 文:
     then 節の内部から解析を開始した場合には else 節内の解析は行う必要がない (図 17)。そのためこのような状況で else 文が現れた場合は endif 文の次の文から 3 の処理を続ける。
   - endif 文:
     endif 文の次の文から 3 の処理を続ける。
   - for/while 文:
     for/while 文の評価式において，解析の対象となる変数に対する参照がある場合には，定義は有効であるとして解析を終了する。
     for/while ループは評価式の値により，ループ本体またはループの後続文を実行する。本解析では評価式の値にかかわらず両方の解析を行う (図 18)。具体的には，for/ while 文の次の文および endfor/endwhile 文の次の文から 3 の処理を続ける。

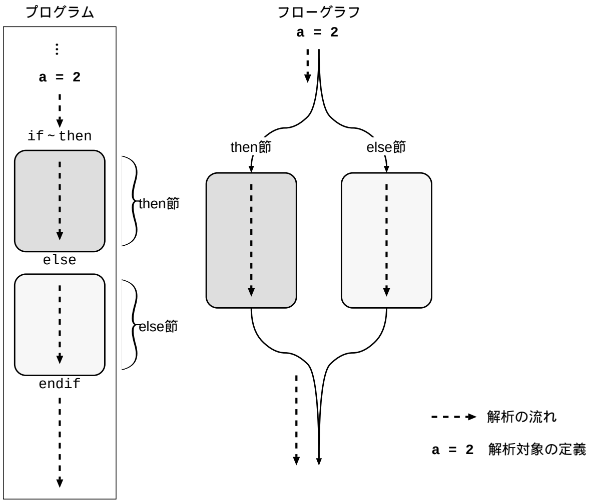


図 16: `if` 文のデータフロー解析

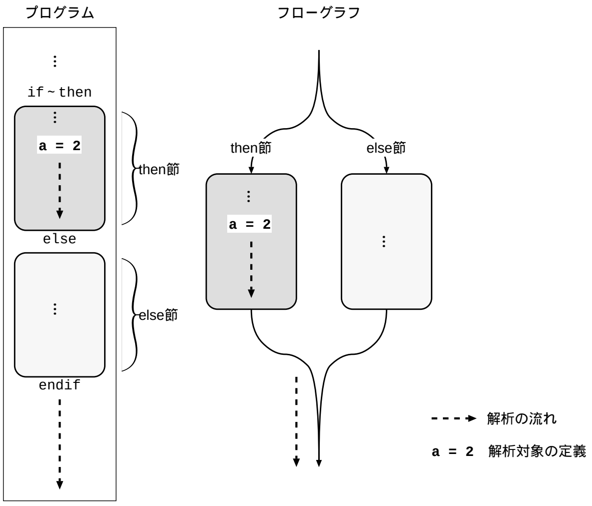


図 17: `then` 節内部からのデータフロー解析

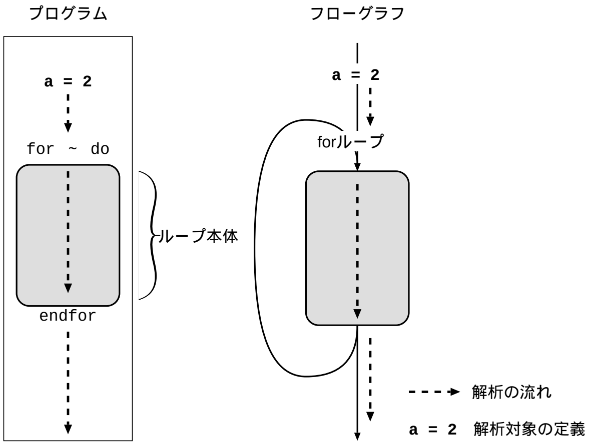


図 18: `for(while)` ループのデータフロー解析

- `endfor/endwhile` 文:
  `for/while` ループ内から解析を開始し `endfor/endwhile` 文に達した場合，評価式の値によりループ内の最初の文またはループに後続する文が次に実行される。本解析では評価式の値にかかわらず両方の解析を行う (図 19)。具体的には，`for/while` 文の次の文および `endfor/ endwhile` の次の文から 3 の処理を続ける。

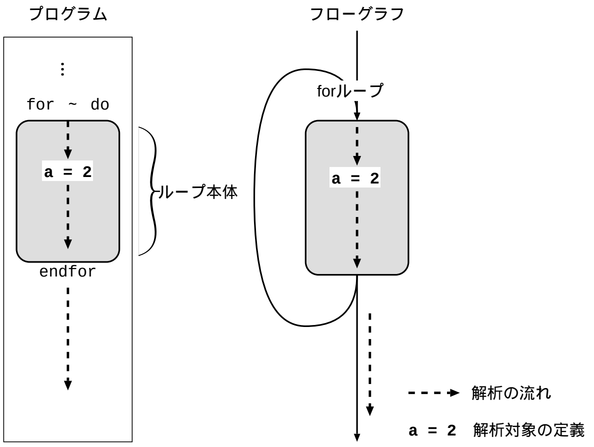


図 19: `for(while)` ループ内部からのデータフロー解析

4. 解析により，対象変数について次の条件のどちらかが成立したならば，その変数は削除可能と判断して削除の候補とする。
   - 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は無効である (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無効な状態で再定義される。
5. 1 で求めたリストが空であれば処理を終了する。そうでなければリストから次の定義文を取り出し 3 の処理を行う。

3.3.2 不要文の削除

本処理では，データフロー解析により削除の候補とされた定義文の削除を行う。ただし `RECV` 文が削除候補となった場合には対応する `SEND` 文とともに削除を行うが，このとき送信/受信の対応関係が崩れないようにしなければならない。

削除の方針 単純な代入文については，一般のコンパイラの場合と同様に無条件で削除することができる。

ところが `RECV` 文の場合はプロセッサ間通信によって値を得ているため，対応する `SEND` 文の削除も同時に行う必要がある。ただし `SEND/RECV` 文の対応関係はコンパイラが把握しているため，対応する `SEND` 文は簡単に求めることができる。

ところが，単純に `SEND/RECV` 文を削除しただけでは問題が生ずることがある。例えば図 20では，1 つの `BROADCAST` 文 (5 行目) に対して 2 つの `RECV` 文が存在している (13, 17 行目)。13 行目の `RECV` 文は解析により削除可能であると判断される。しかし，単純に 13 行目のみを削除したり，13 行目と 5 行目を削除すると，送受信間の対応が崩れてしまう。すなわちこのように複数の `SEND` 文や `RECV` 文が存在する場合には単純な削除を行うことはできない。そこで本手法ではこの点を考慮し，対応関係にある `SEND` 文と `RECV` 文が複数個存在した場合には，

全ての RECV 文が削除可能である時のみ SEND/RECV 文の削除を行う。

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。すなわち，削除された定義文において参照されていた変数の値が，削除によって参照されなくなったために，この変数の定義文も不要なものとなることがある。そこでこのような変数も削除されるように，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データフロー解析を行い，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 SEND/RECV 文の対応関係を崩さないようにしなければならない。そこで関係する SEND/RECV 文全てが削除することができる場合のみ SEND/RECV 文を削除する。具体的には以下の処理を行う。

  1. 削除候補の RECV 文に対応する SEND 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 文のうち不要な文の集合 Xを求める。
  4. $X = R$ならば Sおよび Rに含まれる文を削除する。

4　性能評価

4.1　評価の目的

本研究では実際に TINPAR に対して本最適化手法の実装を行った。このため現在，TINPAR は本手法の適用を行った目的プログラムを自動的に生成することが可能である。

そこで，本最適化手法の適用を行って並列化されたプログラムと，本手法を適用しないで並列化されたプログラムの実行性能を比較し，本手法による性能向上が実際に得られているか評価を行う。評価は並列計算機 AP1000 を使用し，それぞれのプログラムの加速率を測定することによって行った。

4.2　評価用プログラム

4.2.1　概要

本評価には，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

$$Ax = b$$

の解を求めるプログラムを使用した。なおこのプログラムは，線型代数や正方行列を扱うための LINPACK パッケージ [11] 中のサブルーチンを使用したものであり，一般的には LINPACK ベンチマークプログラム [12] として知られている。

本評価プログラムの構造は図 21に示すように 2 つのサブルーチン dgefa, dgesl からなる。

dgefa 部分的ピボッティング付き LU 分解を行う。$O(n^3)$ の浮動小数点演算を必要とする。

```
for i_outer = ... do

  /*所有する a(i) に対する処理*/
  for i∈ {i|owner(a(i))==GET_CELL_ID(),0≤i ≤n-1} do
    BROADCAST_REAL( a(i) )  /*他プロセッサに放送*/
  endfor

   ⋮

  /*所有しない a(i) に対する処理*/
  if ... then
    for i = ... do
      RECV_REAL(owner(a(i)),t) /*受信(参照なし)*/
    endfor
  else
    for i = ... do
      RECV_REAL(owner(a(i)),t) /*受信(参照あり)*/
      b(i+1) = t
    endfor
  endif
endfor
```

図 20: 通信コード削除不可能な例

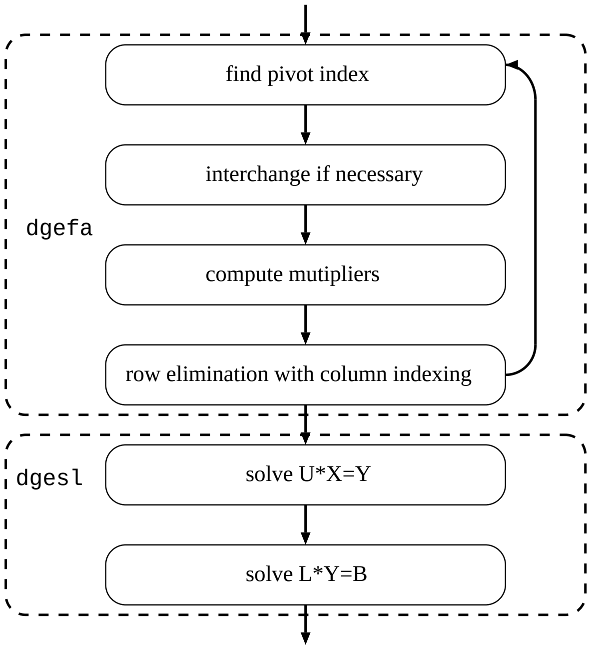


図 21: 本評価プログラムの構造

dgesl dgefa で作成した上三角行列の後退代入によって x を求める。$O(n^2)$ の浮動小数点演算を必要とする。

LINPACK ベンチマークプログラムは FORTRAN 77 で記述されているが，本評価には TINPAR の入力言語である拡張 Tiny Language に変換したプログラムを使用した。

4.2.2 データ分割手法

owner computes rule に従った並列化戦略をとった場合，採用したデータ分割法が性能に大きな影響を及ぼすため，プログラムに対して最も適したデータ分割法を施すことが重要である。

LU 分解を行う dgefa ルーチンでは，行列 A に対してピボット行の選択やピボット行の交換など列方向のアクセスが多い。そのため同一列が同一プロセッサに配置されるように A は列方向分割を採用した。また，分解時の負荷分散を考慮してサイクリック分割を使用する。ベクトル b も同様にサイクリック分割を行った (図 22)。

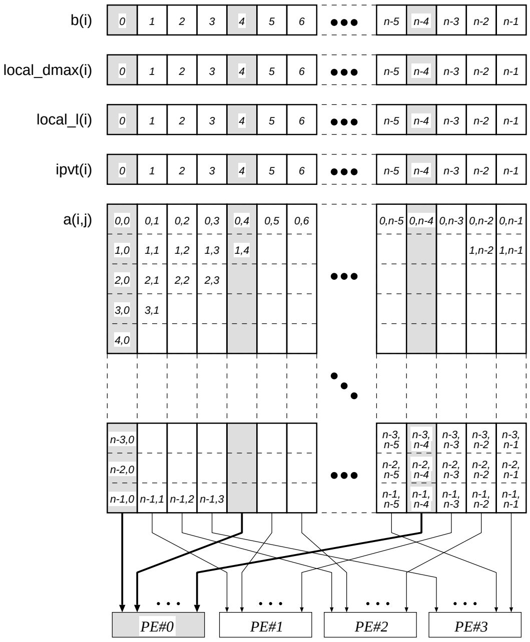


図 22: dgefa で使用するデータ分割方法 (プロセッサ数 4 の場合)

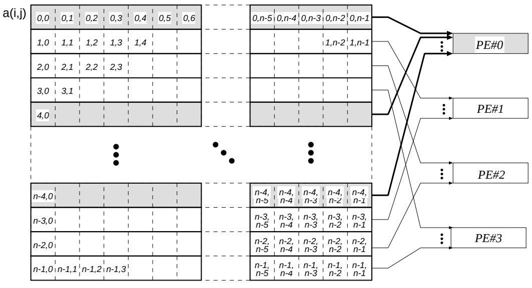


図 23: dgesl で使用するデータ分割方法 (プロセッサ数 4 の場合)

一方，後退代入を行う dgesl ルーチンでは，再内側のループが行列 A に対して行方向のアクセスを行っている。このため同一行の内容は同一プロセッサに配置した場合には通信回数を少なくすることができる。そこで dgesl では行列 A は行方向のサイクリック分割を採用した (図 23)。そのため，dgefa と dgesl の間で A の再配置 (redistribution) のためのコードを挿入した。

また，ピボット行のインデックスを格納する配列 ipvt()，ピボッティング時に分解列でのテンポラリとして用いられる配列 local_dmax()，local_l() に対してもサイクリック分割を適用した (図 22)。

配列変数 local_dmax() は各分解列について最大絶対値を一時的に格納する変数であり，第 k 列のピボッティングでは local_dmax(k) が使用される。この一時変数をスカラ変数としなかった理由は，local_dmax(k) の所有者すなわち行列 A の分解列の所有者プロセッサのみにピボッティングを行わせるためである。また同様の理由で，ピボット行のインデックスを格納する一時変数 local_l() も配列変数となっている。

4.2.3 本最適化手法との親和性

評価プログラム中のピボッティングを行う部分を図 24に示す。ここでは，分解列中で値の絶対値が最大となるような要素を選択する処理を行っている。

先に述べたとおり，dgefa ルーチンにおいて行列 A は列方向分割がなされている。このため同一列の要素は同一プロセッサ上に配置されており，実際のピボッティングは分解列 (第 k 列) の所有者プロセッサによってローカルに行うことが可能である。実際，図 24の 11 行目は local_dmax(k) に対する演算であるため，owner computes rule に従って分解列の要素の所有者プロセッサによって行われる。しかし 10 行目において，スカラ変数 candidate_a と dmax が if 文での判定のためのテンポラリ変数として用いられている。スカラ変数は所有者が全プロセッサであるため，これらに対する代入 (8,9 行目) は全プロセッサへの放送によって行われると考えられる。本来ピボッティングは 1 プロセッサ上で実行可能であるから，このような放送は明らかに不要である。この不要な通信を削除するために，この部分に対して本最適化手法を適用することができると考えられる。

4.3 評価と考察

図 25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 25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 台の時には実に 49.7 倍の性能向上が得られた。

そこで，本最適化手法の適用の有無により実行性能に大きな差が生じた原因について考察を行う。

評価に使用したプログラムの大部分は 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。このルーチンは図 26のような流れをしており，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。本評価では行列 A を列分割しているため，分解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分解列の所有者プロセッサ上での逐次実行となる。そのため，逐次実行部分となるピボッティング部分 (図 24) は性能に与える影響が大きいと考えられる。

このピボッティング部分に対し，本最適化手法を適用したものとしないもののコンパイル出力をそれぞれ図 27，図 28に示す。本最適化手法を適用しない場合，2 つのテンポラリ変数 local_dmax，candidate_a への代入のために値が全プロセッサに放送される (図 29)。すなわち使用されないデータのための通信が多数発生する。一方，本最適化手法を適用した場合，

```
/*ピボッティングを行う部分 */
/* a{ij} は a(j,i) に対応 */
for k=0,n-2 do
  /*ピボッティングを行う*/
  candidate_a = abs( a(k, k) )
  local_dmax(k) = candidate_a /*初期値設定*/
  for i = k+1, n - 1 do
    candidate_a = abs( a(k,i) )
    dmax=local_dmax( k )
    if (candidate_a >dmax) then  /*比較*/
      local_dmax(k) = candidate_a
    endif
  endfor
  /*分解行の消去*/

    ⋮

endfor
```

図 24: ピボッティングルーチン

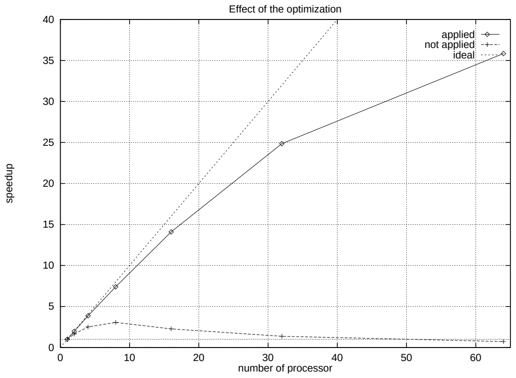


図 25: 本評価プログラムに対する本最適化手法の効果

データフロー解析により local_dmax と candidate_a の値は分解列を所有するプロセッサ以外では使用されないことがわかる。したがって不要な放送は削除され，通信を全く必要とせずにピボッティングを行うことができる。(図 30)。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。

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。

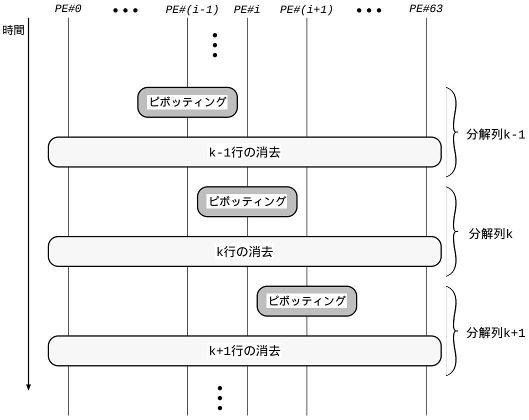


図 26: dgefa ルーチンにおける処理の流れ

5　結論

本研究では，データフロー解析にもとづいて不要な通信コードを削除する最適化手法を提案した。

本最適化手法では，プログラム中の変数定義の有効性について実行の流れに従って解析し，その結果定義された値が参照されないと判明した場合，その定義文を削除する。この時，削除された定義が受信文であったならば対応する送信文も削除する。これにより，目的プログラムからは不要な通信は削除され，効率のよいプログラムの生成が可能となる。

本最適化手法は，一時的に用いられるスカラ変数への不要な代入および通信を削除することができる。さらに本手法はループを並列化するための最適化手法の一つであるスカラエクスパンションの代用としても適用することができる。

本研究ではさらに，提案した最適化手法を TINPAR に実装し，部分的ピボッティング付きガウス消去法を行うプログラムを用いて性能評価を行った。本手法を適用しないで並列化したプログラムでは不要な放送が多数挿入されるため，プロセッサ数 64 台の時での実行性能は逐次実行よりも低いものであった。ところが本最適化手法を適用したプログラムではこのような不要な通信が全て削除されたため，本手法を適用しない場合と比較して 49.7 倍の性能向上が得られた。

謝辞

本研究の機会を与えてくださり，適切な御指導を賜わった富田眞治教授に深甚な謝意を表します。

また，貴重なご助言を頂いた中島浩助教授，森眞一郎助手に深く感謝致します。

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。さらに，共同研究者である三吉郁夫氏，前山浩二氏をはじめとして，日頃ご討論頂いた京都大学工学部情報工学科富田研究室の諸兄に感謝致します。

```
for k_outer = ... do

 /* 所有する a(k,*) に対する処理 */
 for k∈ {k|owner(a(k))==GET_CELL_ID(),0≤i ≤n-2} do
  BROADCAST_REAL( a(k,k) ) /* to tmp_a */
  candidate_a = abs( a(k,k) )
  local_dmax(k) = candidate_a
  for i = k+1, n-1 do
   BROADCAST_REAL( a(k,i) ) /* to tmp_b */
   BROADCAST_REAL(local_dmax(k) ) /* to tmp_c */
   candidate_a = abs( a(k,i) )
   dmax = local_dmax( k )
   if ( candidate_a > dmax ) then
    local_dmax(k) = candidate_a
   endif
  endfor

  /*分解行の消去*/

      ⋮

 endfor

 /* 所有しない a(k,*) に対する処理 */
 for k∈ {k|owner(a(k))!=GET_CELL_ID(),0≤i ≤n-2} do
  RECV_REAL( owner( a(k,*) ), tmp_a)
  candidate_a = abs( tmp_a )
  for i = k+1, n-1 do
   RECV_REAL( owner( a(k,*) ), tmp_b)
   candidate_a = abs( tmp_b )
   RECV_REAL( owner( a(k,*) ), tmp_c)
   dmax = tmp_c
   /*if 文は削除された*/
  endfor

    /*分解行の消去*/

         ⋮

 endfor
endfor
```

図 27: ピボッティングルーチン (本最適化適用なし)

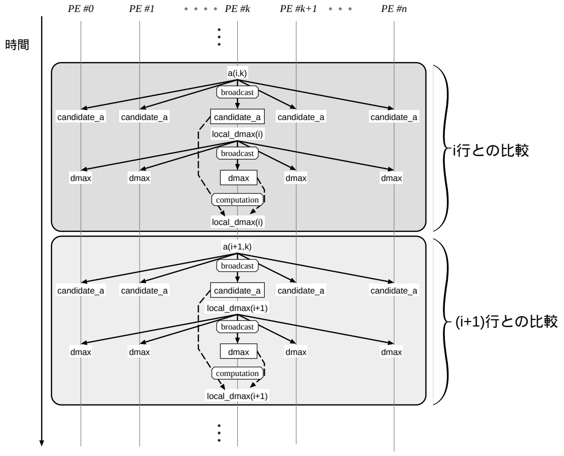


図 29: k列でのピボッティングの流れ (本手法適用前)

```
for k_outer = ... do

 /* 所有する a(k,*) に対する処理 */
 for k∈ {k|owner(a(k))==GET_CELL_ID(),0≤i ≤n-2} do
  candidate_a = abs( a(k,k) )
  local_dmax(k) = candidate_a
  for i = k+1, n-1 do
   candidate_a = abs( a(k,i) )
   dmax=local_dmax( k )
   if( candidate_a > dmax) then
    local_dmax(k) = candidate_a
   endif
  endfor

  /*分解行の消去*/

         ⋮

 endfor
endfor
```

図 28: ピボッティングルーチン (本最適化適用なし)

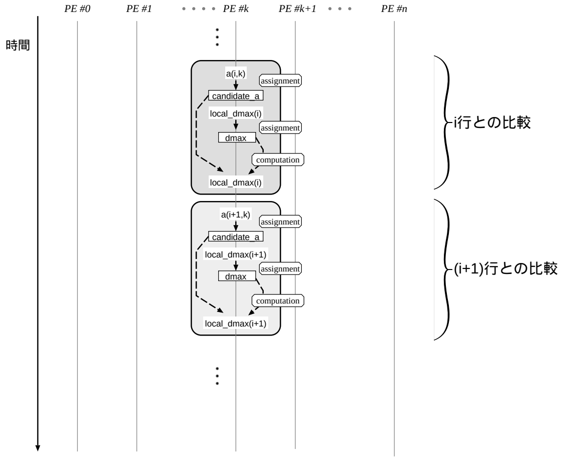


図 30: k列でのピボッティングの流れ (本手法適用後)

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 — 最適化手法と評価 —, 情処研報 94-HPC-54, pp.45-52, 1994

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

[4] 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, 情処学会 第50回全国大会論文集 採録予定, 1995

[5] エイホ, A.V, セシィ, R, ウルマン, J.D 共著; 原田賢一訳: コンパイラII(サイエンス社, 1990), 第10章

[6] 中田 育男: コンパイラ (産業図書, 1981), 第10章

[7] McMahon, F: *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*, Lawrence Livermore National Laboratory Report UCRL-53745, 1986

[8] Feo, J.T: *An analysis of the computational and parallel complexity of the Livermore Loops*, Parallel Computing vol. 7, pp.163-185, 1988

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

[10] 長島 重夫, 田中 義一: スーパコンピュータ (オーム社, 1992), pp.136-140

[11] Dongarra, J. J, Bunch, J. R, Moler, C. B, Stewart, G. W: *LINPACK USERS' GUIDE*, SIAM Pub., 1976

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

[13] 島崎 眞昭: 数値計算におけるベンチマーク, 情報処理Vol31 No.3, pp. 313-320, 1990

[14] 津田 孝夫: 数値処理プログラミング (岩波書店, 1988), 第3章

[15] 梅田 憲: データパラレルアプローチに基づいた分散メモリ型並列計算機用自動並列化コンパイラ, 京都大学大学院工学研究科情報工学専攻 修士論文, 1994

[16] Dowd, K 著; 久良知 真子訳: High Performance Computing(オーム社, 1994), 第10章