2007－HPC－112 (6)
2007／9／9

# ストリーム数とリユースを考慮したループ分配方式

本川 敬子† 橋本 博幸† 伊藤 信一‡ 久島 伊知郎†

† (株) 日立製作所システム開発研究所
‡ (株) 日立製作所ソフトウェア事業部

ループ分配は、ループ本体を複数の部分に分割し、各部分を別ループとして実行するループ変換手法である。本稿で提案するループ分配方式は、巨大ループを対象に適切なループサイズとなるよう分割を行う。ループサイズの基準として、レジスタ数、命令数の他に、ハードウェアプリフェッチ機構を意識してループ内のストリーム数を一定以下にすることを特徴とする。また、配列参照のリユース解析により同じ配列ストリームが別ループに分割されることによるデータ局所性低下を防ぐ。本ループ分配方式をコンパイラに実装し、日立スーパーテクニカルサーバ SR11000 モデル J1 上で評価した結果、NPB2.3 では SP で 22%、姫野ベンチでは 59%の性能向上が得られた。

# Loop Distribution considering Number of Streams and Data Reuse

Keiko Motokawa†, Hiroyuki Hashimoto†, Shinichi Itou‡ and Ichiro Kyushima†

†Systems Development Laboratory, Hitachi, Ltd.
‡Software Division, Hitachi, Ltd.

Loop distribution is a loop transformation technique, which splits a loop body into multiple parts and executes them in separate loops. In this paper we propose a loop distribution method for a large body size loop to split it into appropriate size parts. To determine appropriate size, we consider the number of streams in loop to exploit hardware prefetching mechanism, in addition to the number of registers and instructions. We also consider data locality by reuse analysis of array references. We implemented this loop distribution method in our compiler, and evaluate the performance on HITACHI SR11000 super technical server model J1. Our results show that it improves the performance by 22% for SP in NPB2.3, and 59% for Himeno benchmark.

## 1．はじめに

ループ分配とは、ループ本体を複数の部分に分割し、それぞれの部分をループとして実行するループ変換手法である。この変換手法は古くから知られており、様々な目的で適用される。ベクトルプロセッサやマルチスレッド向けには、並列化を阻害するループ運搬依存を持つ部分とそうでない部分にループを分割して並列化可能なループを生成することにより、並列性を高める[1][2]。多重ループに対しては、最内側ループの本体以外に実行文を含むようなループネストを分配して完全ネストループを生成することにより、ループ交換など他のループ変換を適用する機会を増す[2]。また、巨大ループに対しては、ループを分割して各ループ本体を小さくすることにより、ループコードが命令キャッシュに収まることによる効果や、１つのループ内でアクセスするデータ数を減らしてレジスタスピルを削減するといった効果が知られている[2]。

本稿で述べるループ分配方式は、巨大ループを小さくすることによる実行性能の向上を目的とする。プロセッサの資源を考慮したループサイズにするという観点から、使用レジスタ数の考慮によるレジスタスピルの削減に加えて、プリフェッチストリーム数に着目する。

近年の高性能プロセッサでは、ハードウェアがメモリアクセスのパターンを監視し、連続的に参照されるデータストリームを検出して自動的にプリフェッチを行う、ハードウェアプリフェッチ機構が主流となっている。プロセッサの監視対象のストリーム数には制約があるため、ループ内のストリーム数がハードウェアプリフェッチ機構の対象ストリーム数以下となるようにループを分配することにより、プリフェッチの効果向上を狙う。

巨大ループの分配では、同じ配列を参照する異なる文をループ分配後に別々のループで実行する場合などに、データ局所性を低下させることも知られている[2]。本稿で述べる方式では、配列参照のリユースを考慮に入れ、データ局所性を低下させないようにループ分配を行う。

本稿では、巨大ループを対象として、プロセッサの資源やメモリ参照のデータ局所性を考慮したループ分配方式を提案する。また、コンパイラへの実装および評価結果について述べる。

## 2．ループ分配方式

### 2．1　ループ分配の概要

ループ分配によるループ変換とその効果について、例を用いて簡単に説明する。本稿でのループ分配は、最内側ループのみを対象とし、巨大ループを分割してループ本体を小さくする。

本稿では、ストリーム数の削減を分配の主目的とする。ループ分配によるストリーム数の削減例を例1に示す。

［例1：ストリーム数の削減］

（変換前）

```
do i=1,N
  x(i) = a(i)+b(i)+c(i)+d(i)+e(i)+f(i)
enddo
```

（変換後）

```
do i=1,N
  x(i) = a(i)+b(i)+c(i)+d(i)
enddo
do i=1,N
  x(i) = x(i)+e(i)+f(i)
enddo
```

本例では、元のループのロード対象ストリームは配列aからfの6ストリームであるのに対して、分配後は第一ループが4ストリーム、第二ループが3ストリームである。ハードウェアプリフェッチ機構の対象ストリーム数が4の場合、元のループではプリフェッチ対象とならないストリームが生じる可能性があるのに対して、分配後は全ストリームに対してハードウェアプリフェッチの効果が期待できる。尚、本例のようにループ内に文が1つしかない場合には、本変換のように文を分割してループ分配を適用する。

例2にループ分配によりデータ局所性が低下する例を示す。本例では第一文と第三文で同じ配列a1, a2を参照しており、第三文ではこれらをレジスタ上で再参照できる。変換後はこれらが別ループに属するため、第二ループで再ロードが必要となり、性能が低下する可能性がある。

［例2：分配によるデータ局所性の低下］

（変換前）

```
do i=1,N
   a(i) = a1(i)+a2(i)+...
   b(i) = b1(i)+b2(i)+...
   c(i) = a1(i)+a2(i)+...
enddo
```

（変換前）

```
do i=1,N
   a(i) = a1(i)+a2(i)+...
   b(i) = b1(i)+b2(i)+...
enddo
do i=1,N
   c(i) = a1(i)+a2(i)+...
enddo
```

本稿では、このような配列データのリユースを考慮し、同じデータがなるべく同じループに属するような分配を行う。例えば、例3に示すように文を並び替えてから分配することにより、配列a1とa2を参照する第一文と第三文が第一ループに、元のループの第二文が第二ループに属するように分配を行う。

［例3：リユースを考慮した文の並び替え］

（文の並び替え後）

```
do i=1,N
   a(i) = a1(i)+a2(i)+...
   c(i) = a1(i)+a2(i)+...
   b(i) = b1(i)+b2(i)+...
enddo
```

（ループ分配後）

```
do i=1,N
   a(i) = a1(i)+a2(i)+...
   c(i) = a1(i)+a2(i)+...
enddo
do i=1,N
   b(i) = b1(i)+b2(i)+...
enddo
```

例4にスカラ拡張を伴うループ分配の例を示す。本例では変換前のループ内では第一文で定義したスカラ変数vを第二文で使用する依存関係があるため、このままではループ分配できない。そこで、スカラ変数vをワーク配列Wの参照に置換することにより、ループ分配を適用する。元のループと比べると配列Wのストア・ロードが増加する。

［例4：スカラ拡張を伴う例］

（変換前）

```
do i=1,N
   v=a(i)+b(i)+...
   c(i)=...v...
enddo
```

（変換後）

```
do i=1,N
   W(i)=a(i)+b(i)+...
enddo
do i=1,N
   c(i)=...W(i)...
enddo
```

ループ分配方法の決定にあたっては、主に以下の事項を考慮する。

(a) サイズ

ループ本体中の使用レジスタ数、命令数、ストリーム数が、分配後に基準値以下となるように分配点を決める。レジスタ数はプロセッサのレジスタ数以下となるように、ストリーム数はハードウェアプリフェッチ機構の対象ストリーム数以下となるようにする。

(b) リユース
同じ配列ストリームをアクセスする命令同士はなるべく同じループにする。
(c) スカラ拡張
スカラ依存が分配点を跨ると配列のストア・ロードが発生するコストを考慮する。

## 2.2 処理方式

本節ではループ分配の処理方式について述べる。
我々のコンパイラでは、ソース上の各文を1つの木構造で表現した中間コードを用い、ソースレベル最適化として、マルチスレッド向けの並列化処理を行った後、ループネスト最適化を中心とするプログラムの構造変換を伴う最適化と、式の移動などの伝統的最適化を行う。その後、ターゲットプロセッサの命令にほぼ1対1に対応する命令レベル中間コードへの変換を行い、レジスタ割付けや命令スケジューリング等の最適化を実施する。
ループ分配は、ループネスト最適化内で実施される。ループ分配と逆の変換を行うループ融合は、ループネスト最適化に先立って実施される。
ループ分配では、各最内側ループに対して以下の手順で処理を行う。
(1) 文の分割
文のサイズが基準値を超えるような巨大文を複数の文に分割しておく。文分割の処理については2.5節で詳しく述べる。
(2) 依存グラフの作成
ループ本体の各文に対応するノード（文ノード）を生成し、ループ先頭から順にS1, S2, ... のようにラベル付けする。各文の組の依存関係を調べ、ノード間にデータ依存のエッジを張る。
(3) 文のグループ化
依存グラフの文ノードを分配グループと呼ばれるグループに分類し、分配グループ集合を生成する。
全ての文ノードは、いずれか1個の分配グループに属する。分配グループGiは、

Gi = <..., Sj, Sk, ....> （j < k とする）

のように文ノードのインデックスが昇順となるような列から成る。また、グループの組

Gi = < Sm, ... >, Gj = <Sn, ...>,

に対して、「m < n ならば i < j」が成り立つようにする。即ち、分配グループのインデックスの順序と各グループの先頭文ノードのインデックスの順序は一致する。
分配グループ集合は、

< ..., Gj, Gk, ... > （j < k とする）

のような、インデックスが昇順となるような分配グループの列と、分配グループ間の依存エッジ集合、および、分配グループ間のスカラエッジ集合から成る。分配グループ間の依存エッジGi→Gjは、Giに属する文SiとGjに属する文Sjの間に文ノード間の依存エッジSi→Sjが存在することを示す。スカラエッジGi→Gjは、スカラ拡張対象変数vがGiに属する文で定義され、その値がGjに属する文で使用されることを表し、Gi中のどのスカラ定義に対する依存かを識別するための定義点情報を保持している。
各分配グループは分配後の各ループの本体に対応し、分配グループ列は分配後のループの順に対応する。各分配グループ内の文ノードの順は、分配後の各ループ本体の文の順に対応する。
分配グループ集合の決定方法については、次節で詳しく説明する。
(4) 文の並び替え
分配グループ列の先頭グループから順に、グループに属する文ノードの列に従って文を並べる。
例えば、分配グループの列が

G1=<S1, S3, S6>, G2=<S2>, G3=<S4, S5>

ならば、ループ内の文を

S1, S3, S6, S2, S4, S5

の順に並べ替える。
(5) スカラ拡張
スカラ拡張に先立ち、2.6節で述べるスカラ拡張対象変数を削減する最適化を適用し、分配グループ間のスカラエッジを更新する。その後、分配グループ間のスカラエッジに対し、スカラ拡張の変換を行う。スカラエッジに登録されているスカラ変数vの定義点をW[i]のようなワーク配列の参照で置換し、その定義が到達するループ内の使用も同様にW[i]で置換する。
(6) ループの分配変換
ループをコピーして分配グループの数のループを生成し、ループ本体が各分配グループの文集合に対応するようなコードを生成する。

## 2.3 文ノードのグループ化

本節では、前節の処理方式(3)で述べた文ノードのグループ化、即ち、依存グラフの文ノードを分配グループに分類する方式の詳細を説明する。
基本的な処理方針は、初めに(1)で依存によるグループ化を行い、最小単位のグループを生成する。その後(2)(3)(4)では、複数のグループを併合し、グループをだんだん大きくしていく。
(1) 依存によるグループ化
依存グラフ中の各文ノードを、分配グループに登録する。この際、2ノード間にループ分配を妨げる依存があれば、これらのノードを同じ分配グループに登録する。
ループ分配を妨げるデータ依存は、上向きの依存である。但し、スカラ拡張の対象となる変数の依存は、ループ分配を妨げる依存には含めない。
本グループ化に伴う文の並び替えにより、依存関係が不正になる場合があるので、次のように補正する。
S2→S3およびS3→S1のデータ依存があり、S3→S1は上向きの依存でループ分配を妨げるので、G1=<S1, S3>, G2=<S2>, ... のようにグループ化さ

れたとする。S2→S3 は元々下向きの依存であったが、グループ化により実行順序が逆転して上向きの依存になり、不正となる。

このようにグループ間の上向きの依存が残っている場合、本来下向きの依存がグループ化により不正に上向きになったものなので、依存先のグループに依存元のグループを併合する。

(2) リユースによるグループ化

ループ内の配列参照をリユースにより分類する。2 つの異なる配列要素参照に対して、それらが同一配列の参照でかつ添え字差が一定値以下の場合、それらの参照は、同一配列グループに属するとする。

分配グループの組(Gi,Gj)に対して、Gi に属する参照と Gj に属する参照を共に含む配列グループの数を(Gi,Gj)のキャッシュリユース指数とする。また、Gi と Gj に含まれる同一参照（同配列で同一添え字の参照）の数を(Gi,Gj)のレジスタリユース指数とする。これら両指数の和を(Gi,Gj)のリユース指数とする。

図 2.1 の例では、ループ中の配列参照が 3 個の配列グループに分類され、各参照は図のように分配グループ G1,G2,G3 に属している。(G1,G2)に対するリユース指数は、配列 a と配列 b に対してキャッシュリユース指数が 2、a(i)に対するレジスタリユース指数が 1 で、リユース指数は 3 となる。(G1,G3)のリユース指数は b(i+1)のキャッシュおよびレジスタリユースにより 2 となる。(G2,G3)については、b のキャッシュリユースにより 1 である。

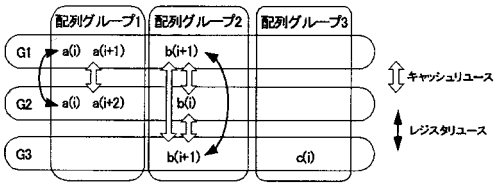


図 2.1 リユースの解析

リユース指数の高い分配グループの組(Gi,Gj)から順に併合を試みる。併合可能な条件は、Gi と Gj の併合後のサイズを見積もり、サイズが基準値以下であることとする。

Gi と Gj (i < j とする)を併合するために、間にある他の分配グループ Gk (i<k<j)も併合が必要な場合がある。Gk の併合が必要なのは、Gk から Gj へ到達する依存パスが存在する場合である。例えば、G2→G3, G3→G5 という依存が存在する場合、G2→G3→G5 という依存パスが存在するので、(G1,G5)を併合する際には(G1,G2,G3,G5)の 4 個の分配グループを併合する。

サイズ見積もり時には、併合対象となる間の分配グループも含めて見積もり、基準を満たせば、これらの分配グループ全てを併合する。

(3) スカラ拡張削減のためのグループ化

分配グループの間にスカラ変数のフロー依存があるとき、グループ間にスカラ依存のエッジを張る。異なるスカラ変数に対しては異なるスカラ依存エッジを張る。分配グループの組(Gi,Gj)間のスカラ依存エッジの数を(Gi,Gj)のスカラ指数とする。

スカラ指数の高い分配グループの組から順に併合を試みる。併合可能な条件は、Gi と Gj の併合後のサイズを見積もり、サイズが基準値以下であることとする。サイズ見積もり時には、(2)と同様に併合対象となる間の分配グループも考慮する。

(4) サイズによるグループ化

サイズが基準を超えない範囲で、隣接グループの組を併合していく。隣接グループの併合では文の並び替えは起こらないので、依存条件などは考慮する必要がない。

併合後のサイズがなるべく均等になるように、次の手順により、サイズが小さいグループから順に併合する。まず各分配グループのサイズを見積もり、最小サイズのグループを取り出す。このグループの前後のグループの小さい方を併合対象に選ぶ。併合サイズを見積もり、基準以下ならば併合する。以上の処理を併合できなくなるまで繰り返す。

## 2.4 グループ化の適用例

前節のアルゴリズムの適用例を示す。

図 2.2(a)にソースプログラムの例を、(b)に依存グラフを示す。依存によるグループ化において、分配を阻害する依存は S3→S1 の逆依存であるから、S1 と S3 を同じグループに登録する。分配グループは次のようになる。

G1=<S1,S3>, G2=<S2>, G3=<S4>, G4=<S5>

次にリユースによるグループ併合を試みる。S3 と S5 の間で配列 e と f のリユースがあるので、(G1,G4)の併合を試みる。間のグループ G2, G3 と G4 の間に依存はないので、併合対象は(G1,G4)のみである。併合後のサイズを見積もり、基準を満たしているとすると、これらのグループを併合する。併合後の分配グループ集合は次のようになる。

G1=<S1,S3,S5>, G2=<S2>, G3=<S4>

次にスカラ拡張を考慮したグループ併合を試みる。S3 の属する G1 と S4 の属する G3 を併合すると併合後のグループの文集合は<S1,S3,S4,S5>となる。この場合のサイズを見積もり、基準を満たしている場合は併合する。ここではサイズが基準値を超えているものと仮定し、スカラ拡張を考慮したグループ併合は行わないことにする。

最後にサイズによるグループ併合を試みる。G2=<S2>と G3=<S4>の併合後のサイズを見積もり、基準を満たしているとすると、これらを併合する。併合後の分配グループ集合は次のようになる。

G1=<S1,S3,S5>, G2=<S2,S4>

以上のグループ化の過程を図 2.2(c)に示す。この分配結果に従いループ分配の変換を行うと、変換後のループは図 2.2(d)のようになる。v の参照はスカラ拡張によりワーク配列の参照 t(i)に置換され

ている。

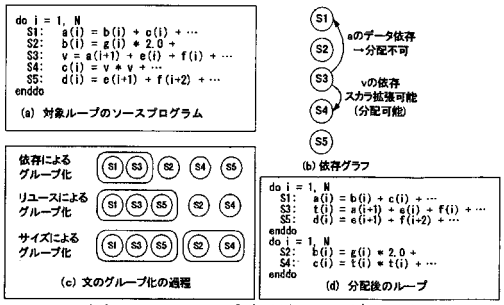


図 2.2 ループ分配処理の適用例

## 2.5 文の分割

文分割では、文のサイズを見積もり、基準値を超えている文を処理対象とする。一文の中間語は木構造で表現されているため、これを辿りサイズが基準を超える点で部分式を切り離す。

図 2.3 に文分割の例を示す。(a)において、E1, E2, ...は式を表すものとする。(c)において、右辺の中間コードを親から子へと辿り、初めに①の加算ノードにおいて本ノードをルートとする部分木全体のサイズを見積もり、基準値を超えているとすると、次に1子のノード②をルートとする部分木（図の点線部）のサイズを見積もる。サイズが基準値以下であればこの部分木を切り出し、元の文の前に挿入する。切り出した部分は左辺のa(i)で置換する。この結果、分割後のコードは(b)のようになる。残りの文 ’ a(i)=a(i)+(E5*E6)’ のサイズがまだ大きい場合には、この文に対して処理を続け、さらに分割することができる。

上で述べた方式では、オペランドの順序を入れ替えることが出来ないため、分配効果が十分に得られない場合がある。例えば、右辺がE1+E2+E3+E4の文を分割する場合、オペランド2個程度が適正サイズであるとすると、E1+E2とE3+E4の組に分割することになり、E1とE3の間にリユースがあっても同一ループで実行することができない。

このようなケースに対処するため、文分割において、各オペランドが別の文となるよう細かく分割し、一時変数にその結果を保持する変換を導入した。本変換は、対象文の右辺が図2.4(a)のような加算式に場合に限定して適用している。図 2.4(b)に文分割後のループを示す。本ループに対して分配を適用し、リユースを考慮した分配グループ併合によりE1とE3が同じループにグループ化された場合、文の並び替えによって(c)のようなコードに変換される。これに対して分配変換を行うと、(d)のようなコードが生成される。

このように予め各オペランドに対して一時変数を導入して文分割する方法では、ループ分配により多くのスカラ拡張が発生する危険性がある。そこでスカラ拡張を削減する最適化を行う。これについては次節で述べる。

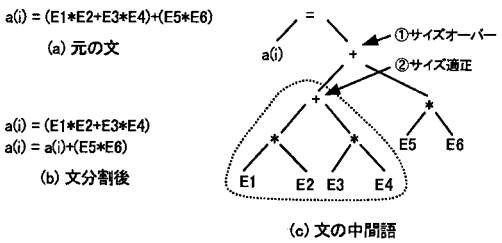


図 2.3 文分割の例

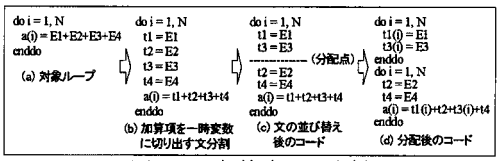


図 2.4 加算式の文分割

## 2.6 スカラ拡張の削減最適化

スカラ拡張の対象変数削減の例を図 2.5 に示す。(a)ではv1とv2の2変数がスカラ拡張対象である。これらの変数の使用が v1*v2 のみであるとすると、’ v1*v2’ を v1, v2 の定義直後に引き上げ、その式の結果を保持する一時変数をスカラ拡張対象とすることにより、スカラ拡張を削減できる。変換後のコードは(b)のようになり、変数 t がスカラ拡張対象である。

このようなスカラ拡張の対象を削減する最適化を、スカラ拡張変換の直前に実施している。スカラ拡張対象の変数v1, v2に対して本変換を適用する条件は、v1, v2 の定義が共にある分配グループ Gi に属し、別の分配グループ Gj に属する二項演算の 2 個のオペランドが v1, v2 で、v1, v2 はこれ以外に使用点を持たないことである。このとき、この二項演算部分を一時変数 t の参照で置換し、二項演算を一時変数 t へ代入する文を作成して Gi 内の v1, v2 の定義後に挿入する。

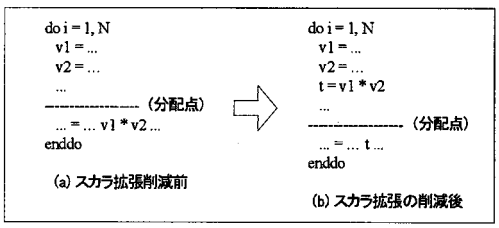


図 2.5 スカラ拡張対象変数の削減

演算の可換性を利用すれば、v1, v2 の使用が直接同一の二項演算のオペランドでない場合にも本最適化が適用できる。例えば、図 2.4(d)のコードで

は t1, t3 の 2 変数がスカラ拡張対象となっているが、加算のオペランドの可換性を利用し、第一ループ内で t1+t3 を計算することにより、スカラ拡張対象を 1 個に削減することができる。

## 3. 評価

前章で述べたループ分配方式を実装し、日立スーパーテクニカルサーバSR11000 モデルJ1 の 1 ノード(16CPU)上で評価した。SR11000 モデルJ1 はCPUに POWER5[1]プロセッサを搭載している。ループサイズの基準としては、POWER5 に合わせてレジスタ数は汎用レジスタ 32、浮動小数点レジスタ 32、プリフェッチストリーム数は 8 とした[3]。

### 3.1 NPB2.3

NPB(NAS Parallel Benchmarks)2.3 の各プログラムに対して、分配が適用されたのは BT, LU, MG, SP の 4 プログラムで、性能に変化があったのは LU と SP であった。LU, SP の性能を図 3.1 に示す。括弧内はクラスを示す。LU ではクラス A は 5%の性能向上が得られたが、クラス C では逆に 5%低下している。SP ではクラス B で 10%、クラス C で 22%の性能向上が得られた。

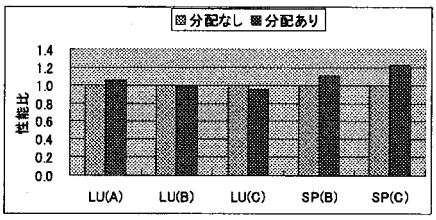


図 3.1 NPB2.3 に対する効果

### 3.2 姫野ベンチ

姫野ベンチ[4]の主要ルーチン jacobi は図 3.2 に示すように 4 個の文（うち 1 個は巨大文）を本体とする 4 重ループである。最内側ループに対して、2.5 節で述べた文分割の適用により巨大文の部分式を別ループとするようなループ分配を適用し、図 3.3 のように変換した。図 3.2 の下線部が図 3.3 の第一ループに属する部分である。

性能を図 3.4 に示す。問題サイズが大きくなるほど分配の効果が向上しており、サイズ XL では 59%の性能向上が得られた。

## 4. おわりに

巨大ループに対するループ分配方式を提案した。レジスタ数、命令数、ストリーム数を基準にループサイズを考慮する他、配列参照のリユースを解析し同じ配列ストリームを分配後もなるべく同じループ内で参照できるようにし、スカラ拡張のコストについても考慮する方式とした。

[1] POWER5 は米国 IBM Corp. の商標です。

本ループ分配を適用し SR11000 モデル J1 上で評価した結果、NPB2.3 では SP（クラス C）で 22%、姫野ベンチ(サイズ XL)で 59%の性能向上が得られた。

```
do loop=1,nn
  gosa= 0.0
  do k=2,kmax-1
   do j=2,jmax-1
    do i=2,imax-1
     s0=a(i,j,k,1)*p(i+1,j,k) &
          +a(i,j,k,2)*p(i,j+1,k) &
          +a(i,j,k,3)*p(i,j,k+1) &
          +b(i,j,k,1)*(p(i+1,j+1,k)-p(i+1,j-1,k)-p(i-1,j+1,k)+p(i-1,j-1,k)) &
          +b(i,j,k,2)*(p(i,j+1,k+1)-p(i,j-1,k+1)-p(i,j+1,k-1)+p(i,j-1,k-1)) &
          +b(i,j,k,3)*(p(i+1,j,k+1)-p(i-1,j,k+1)-p(i+1,j,k-1)+p(i-1,j,k-1)) &
          +c(i,j,k,1)*p(i-1,j,k) &
          +c(i,j,k,2)*p(i,j-1,k) &
          +c(i,j,k,3)*p(i,j,k-1)+wrk1(i,j,k)
     ss=(s0*a(i,j,k,4)-p(i,j,k))*bnd(i,j,k)
     GOSA=GOSA+SS*SS
     wrk2(i,j,k)=p(i,j,k)+OMEGA *SS
    enddo
   enddo
  enddo
  p(2:imax-1,2:jmax-1,2:kmax-1)= wrk2(2:imax-1,2:jmax-1,2:kmax-1)
  enddo
```

図 3.2 姫野ベンチ jacobi のソースプログラム

```
do i=2,imax-1
  t1 = a(i,j,k,1)*p(i+1,j,k)
  t2 = a(i,j,k,2)*p(i,j+1,k)
  t3 = t1+t2
  t4 = b(i,j,k,1)*(p(i+1,j+1,k)-p(i+1,j-1,k)-p(i-1,j+1,k)+p(i-1,j-1,k))
  t5 = t3+t4
  t6 = c(i,j,k,1)*p(i-1,j,k)
  t7 = t5+t6
  t8 = c(i,j,k,2)*p(i,j-1,k)
  wl(i) = t7+t8
enddo
do i=2,imax-1
  t9 = a(i,j,k,3)*p(i,j,k+1)
  t10 = b(i,j,k,3)*(p(i+1,j,k+1)-p(i-1,j,k+1)-p(i+1,j,k-1)+p(i-1,j,k-1))
  t11 = t9+t10
  t12 = c(i,j,k,3)*p(i,j,k-1)
  w2(i) = t11+t12
enddo
do i=2,imax-1
  t13 = b(i,j,k,2)*(p(i,j+1,k+1)-p(i,j-1,k+1)-p(i,j+1,k-1)+p(i,j-1,k-1))
  t14 = wrk1(i,j,k)
  w3(i) = wl(i)+w2(i)+t13+t14
enddo
do i=2,imax-1
  ss=(w3(i)*a(i,j,k,4)-p(i,j,k))*bnd(i,j,k)
  GOSA=GOSA+SS*SS
  wrk2(i,j,k)=p(i,j,k)+OMEGA *SS
enddo
```

図 3.3 分配後のコード(jacobi 最内側ループ)

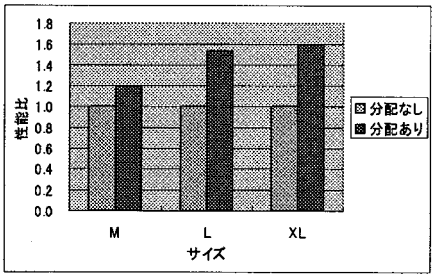


図 3.4 姫野ベンチに対する効果

## 参考文献


[1] Hans Zima and Barbara Chapman, 村岡訳, 「スーパーコンパイラ」, オーム社, 1995.

[2] 中田, 「コンパイラの構成と最適化」, 朝倉書店, 1999.

[3] B. Sinharoy, R. N. Kall, J. M. Tendler, R. J. Eickmeyer and J. B. Joyner, "POWER5 system microarchitecture," IBM J. Res. & Dev. Vol.49, No.4/5, 2005.

[4] http://accc.riken.jp/HPC/HimenoBMT/.