

ループ並列化の為のリタイミングによるループ最適化

東京工業大学
大学院情報理工学研究科
数理・計算科学専攻

五十嵐 巳玲一
(09M37040)

平成22年度 修士論文

指導教官 佐々 政孝 教授

平成23年1月28日

概要

現在、計算機はマルチコア、メニーコア化が進んでおり家庭用の PC でさえデュアルコア以上が一般的となっている。しかし、実際には並列プログラムを人の手で書くことは難しく、特に依存関係の入り組んでいる配列を含むループを並列化することは困難なことである。

ループ内での処理はプログラムの実行時間の大部分、これを並列化することができれば、実行時間の大幅な改善が見込まれることになる。

以上のことから本研究ではループ内の処理の並列度の向上をコンパイラに行わせることを目標とする。ループ内の各繰り返しを飛び越しての依存に注目し、その依存関係を解消することで並列化をねらう。依存関係を解消する手法として、データ依存を計算しループ内の配列が関係する処理の順番を変更する処理を行う。この手法を本稿ではリタイミングと呼ぶ。

リタイミングを使ったデータ依存の解消手法は Liu らのアルゴリズムを採用した。しかし、Liu らは、ループ内の処理後の評価を並列度が上がったかどうかで判断しており、PC 上での実行速度実験は行っていない。本研究ではコンパイラ上で自動並列化したのちに PC 上での実行速度を測定しこの手法が有効であるか検証した。

実装は COINS の HIR 上で行い C プログラムから最適化されたループを含むソースコードを自動出力する事に成功した。そのソースコードから実行コードを生成し実験をおこない、プログラムや適応するコア数によって効果があげられた。逆に効果が見られない場合もあった為、今後の課題としたいと思う。

目次

| | | |
|-------|----------------|----|
| 第1章 | はじめに | 6 |
| 1.1 | 背景 | 6 |
| 1.2 | 本研究の概要 | 6 |
| 1.3 | 構成 | 6 |
| 第2章 | 準備 | 8 |
| 2.1 | ループ | 8 |
| 2.1.1 | ループ制御変数 | 8 |
| 2.1.2 | 動的インスタンス | 8 |
| 2.2 | ループ並列化 | 8 |
| 2.3 | 制御フローグラフ | 9 |
| 2.4 | 逆転制御フローグラフ | 10 |
| 2.5 | 支配木 | 10 |
| 2.6 | 逆支配木 | 11 |
| 2.7 | 支配境界・逆支配境界 | 12 |
| 2.7.1 | 支配境界の定義 | 12 |
| 2.7.2 | 支配境界を求める例 | 12 |
| 2.7.3 | 支配境界を求めるアルゴリズム | 13 |
| 2.7.4 | 逆支配境界 | 13 |
| 2.8 | 制御依存グラフ | 13 |
| 2.9 | データ依存関係 | 14 |
| 2.9.1 | 概要 | 14 |
| 2.9.2 | データ依存グラフ | 14 |
| 2.10 | プログラム依存グラフ | 15 |
| 第3章 | データ依存とその移動 | 18 |
| 3.1 | 配列とデータ依存 | 18 |
| 3.2 | 依存グラフ | 19 |
| 3.3 | 隣接行列 | 20 |
| 3.3.1 | 無効グラフ | 20 |
| 3.3.2 | 重みつき有効グラフ | 20 |
| 3.4 | 依存移動 | 21 |
| 3.4.1 | 依存移動の例 | 21 |
| 3.5 | OpenMP | 23 |

| | | |
|------------|------------------------------------|-----------|
| 第4章 | リタイミングによるループ最適化 | 24 |
| 4.1 | リタイミング | 24 |
| 4.2 | アルゴリズム | 25 |
| 4.2.1 | 強連結成分 (SCC) | 25 |
| 4.2.2 | DAG | 26 |
| 4.2.3 | CGSC | 29 |
| 4.2.4 | ループ逆変換 | 30 |
| 第5章 | 並列化コンパイラ向け共通インフラストラクチャCOINS | 31 |
| 5.1 | COINS の目的 | 31 |
| 5.2 | COINS の構成 | 31 |
| 5.3 | 中間表現のレベル | 31 |
| 5.4 | 高水準中間言語 HIR について | 33 |
| 5.4.1 | 概要 | 33 |
| 5.4.2 | 具体表現 | 33 |
| 第6章 | 実験と評価 | 35 |
| 6.1 | 実装 | 35 |
| 6.1.1 | 対象となるループ | 35 |
| 6.1.2 | HIR 上での実装 | 35 |
| 6.2 | 実験 | 36 |
| 6.3 | 考察 | 39 |
| 6.3.1 | ループのサイズについて | 39 |
| 6.3.2 | アルゴリズムの欠点 | 39 |
| 6.3.3 | 実験の方法 | 40 |
| 6.4 | 今後の課題 | 40 |
| 6.4.1 | 対応できないループについての取り組み | 40 |
| 第7章 | 関連研究 | 43 |
| 7.1 | Liu らの研究 | 43 |
| 7.2 | COINS でのループ並列化 | 43 |
| 第8章 | まとめ | 44 |
| | 参考文献 | 45 |

目 次

| | | |
|------|-----------------------------------|----|
| 2.1 | 制御フローグラフの例 | 9 |
| 2.2 | 逆転制御フローグラフの例 | 10 |
| 2.3 | 支配木の例 | 11 |
| 2.4 | 逆支配木の例 | 11 |
| 2.5 | 制御依存と支配境界の関係 | 14 |
| 2.6 | 制御依存グラフの例 | 15 |
| 2.7 | 制御フローグラフの例 | 15 |
| 2.8 | データ依存グラフの例 | 16 |
| 2.9 | 文を単位としたプログラム依存グラフの例 | 17 |
| 3.1 | 配列を対象としたデータ依存グラフの例 | 18 |
| 3.2 | 重みつき依存グラフの例 | 19 |
| 3.3 | 無効グラフとその隣接行列の例 | 20 |
| 3.4 | 重みつき有効グラフとその隣接行列の例 | 20 |
| 3.5 | 依存移動の例 | 21 |
| 3.6 | 制御変数毎の依存 | 22 |
| 4.1 | リタイミング関数の例 | 24 |
| 4.2 | 強連結成分の例 | 25 |
| 4.3 | SCC によってグラフのタイプを分ける | 26 |
| 4.4 | リタイミングの制約 ($k = 3$) | 26 |
| 4.5 | 新ノード追加と Bellman-Ford アルゴリズム適用 | 27 |
| 4.6 | リタイミング関数の決定と適用 | 28 |
| 4.7 | リタイミング関数の値を標準化 | 28 |
| 4.8 | サイクル内の重みの操作と固定 | 29 |
| 4.9 | DAG アルゴリズムの適用 | 29 |
| 4.10 | リタイミング関数の標準化 | 30 |
| 5.1 | 並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図 | 32 |
| 5.2 | プログラム 5.1 の HIR | 34 |
| 6.1 | プログラムの例 | 35 |
| 6.2 | 図 6.1 の最適化後のソース | 36 |
| 6.3 | プログラム bre について | 37 |
| 6.4 | プログラム reo について | 38 |

| | | |
|------|--------------------------|----|
| 6.5 | プログラム shr について | 38 |
| 6.6 | バリア同期のイメージ | 39 |
| 6.7 | for 文の明示的な分割 | 40 |
| 6.8 | 今回対象外のループの例 1 | 41 |
| 6.9 | ループ最適化への対処 | 41 |
| 6.10 | 今回対象外のループの例 2 | 42 |

第1章 はじめに

1.1 背景

現在、計算機はマルチコア、メニーコア化が進んでおり、それに伴い並列プログラミングに注目が集まっている。プログラムの実行時間の多くはループでの処理で費やされており、ループ内を並列化することで実行時間の大幅な改善が見込める。しかし、配列の複雑な依存関係により並列化が困難であることが現状である。そこで本研究では配列の依存関係に注目し、依存関係を解消してやることにより並列度を向上し、実行時間の改善を見込めるようにすることができた。

1.2 本研究の概要

本研究では、Liuらによるリタイミング [6] という考え方をを用いてループ内の並列度をあげる実装を行い、対象とするプログラムであるC言語のプログラムに対してループ並列化を適用できるようにした。また、リタイミングによるループ最適化の効果を検証した。

本研究の特徴は、プログラムのループの繰り返しを飛び越したデータ依存に注目し、できるだけそのような依存関係を少なくすることによってループの並列度を上げる事である。また、Liuらは計算機上での実行速度実験を行っておらず、その実験を行うことでLiuらの手法の効果の検証を行った。

実装はCOINS[3]のHIR上でいCプログラムから最適化されたループを含むソースコードを自動出力する事に成功した。そのソースコードから実行コードを生成し実験を行ったが大きな実行速度の向上は見られなかった。この原因としては、対象としたループの範囲が狭い事、Liuらのアルゴリズムでは完全にループ外への依存を取り除けない事があげられる。

1.3 構成

本論文の構成を以下に示す。

- 第2章 本論文での説明に必要な予備知識の準備
- 第3章 ループ並列化の妨げとなる配列のデータ依存について
- 第4章 ループ並列化とそのためのリタイミング処理について

- 第5章 並列化コンパ入れ向け共通インフラストラクチャCoins について
- 第6章 実験とその結果、検証
- 第7章 関連研究
- 第8章 まとめ

第2章 準備

2.1 ループ

2.1.1 ループ制御変数

ループ内での定義が1箇所、定義が $i = i + c$ または $i = i - c$ の形をとるような変数のことをループの帰納変数 (induction variable) とよぶ。[1, 2, 7, 9]。ただし、 c はループ不変である。本論文では、さらに以下の3つの条件を加えたものを満たす変数 i をループ制御変数 (loop control variable) と呼ぶ。

- c の値が定数
- ループの繰返し条件の式に変数 i が含まれる
- 変数 i の初期値が一つに定まる

例えば、プログラム 2.1 の変数 i は、初期値が1でループ内で1ずつ値が変化していき、ループの繰返し条件式 $i < 100$ に含まれている。よって変数 i は、このループのループ制御変数である。

プログラム 2.1 (ループの例)

```
1: for ( $i = 1$ ;  $i < 100$ ;  $i = i + 1$ ) {  
2:    $a[i] = 1$ ;  
3: }
```

2.1.2 動的インスタンス

ある文が何度も実行される時、その文を動的インスタンスの実行とよぶ。それぞれの動的インスタンスは、繰返しベクトル $\vec{i} = (i_1, i_2, \dots)$ で定義される。ただし、 i_k は k 番目のループ制御変数とする。

2.2 ループ並列化

ループ並列化とはループの各繰返し (イタレーション) を並列に実行するよう変換することを言う。本論文では、並列実行しやすいようにループを変換しており、このことをループの最適化と呼ぶこととする。

2.3 制御フローグラフ

制御フローグラフ (control flow graph) とは、プログラムの制御の流れをグラフで表したものである [1, 5, 7]。制御フローグラフは、プログラム中の分岐も合流もない部分 (これを基本ブロックという) をノードとして、それらの間を分岐や合流を表す有向辺で結んだ有向グラフである、制御フローグラフに含まれるノードの集合を N_{CFG} 、辺の集合を $E_{CFG} = \{(n, m) \mid n, m \in N_{CFG} \text{ かつ 制御が } n \text{ から } m \text{ に移る}\}$ としたとき、その制御フローグラフは両者の組 $CFG = \langle N_{CFG}, E_{CFG} \rangle$ で表すことにする。

基本ブロック (basic block) は、文 (statement) の列で、その間には分岐も合流もない。基本ブロックの先頭には一般に合流があり。最後の文からは分岐がある。基本ブロック中の文は先頭から最後まで一直線に実行される。

制御フローグラフにおいて、ノード X からノード Y に向かって有向辺が引かれていることを、 $X \rightarrow Y$ と表記する。ここで、 X は、 Y の先行ノード (predecessor node)、 Y は X の、後続ノード (successor node) という。 X の先行ノードの集合を $Pred(X)$ 、後続ノードの集合を $Succ(X)$ と表す。また、先行ノードが複数あるノードを 結合ノード (join node)、後続ノードが複数あるノードを 分岐ノード (branch node) という。

図 2.1 に制御フローグラフの例を示す。プログラム例の各文の右側に付けられた括弧内の数字は、その文が含まれる基本ブロックの番号を表す。この例において、 $Succ(3) = \{4, 5\}$ 、 $Pred(3) = \{1\}$ である。

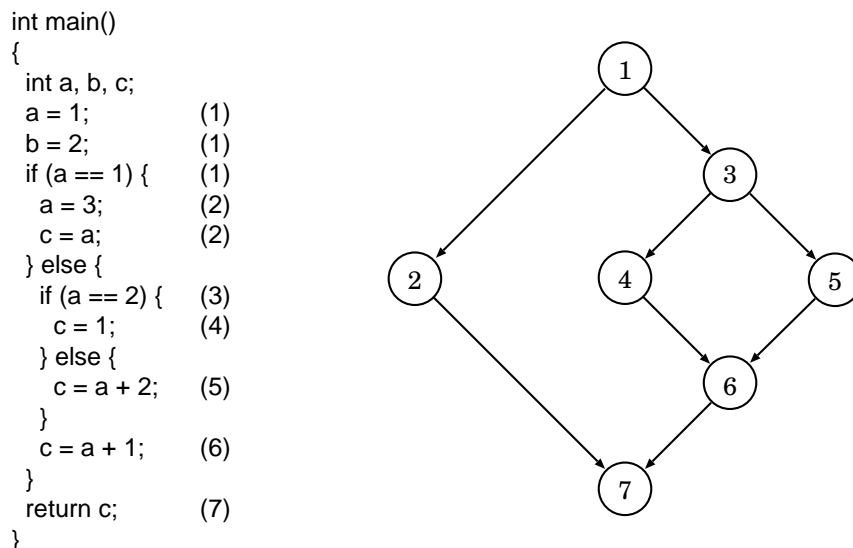


図 2.1: 制御フローグラフの例

2.4 逆転制御フローグラフ

逆転制御フローグラフ (reverse control flow graph) とは、制御フローグラフにおける有向辺を逆向きにした有向グラフである [1, 5, 7]。

制御フローグラフ $CFG = \langle N_{CFG}, E_{CFG} \rangle$ の逆転フローグラフは、辺の集合を $E_{RCFG} = \{(m, n) \mid n, m \in N_{CFG} \text{ かつ 制御が } n \text{ から } m \text{ へ移る}\}$ としたとき、逆転制御フローグラフ $RCFG = \langle N_{CFG}, E_{RCFG} \rangle$ で表すことにする。

図 2.2 に、図 2.1 の制御フローグラフの逆転制御フローグラフを示す。図 2.1 と同様にプログラム例の各文の右側に付けられた括弧内の数字は、その文が含まれる基本ブロックの番号を表す。

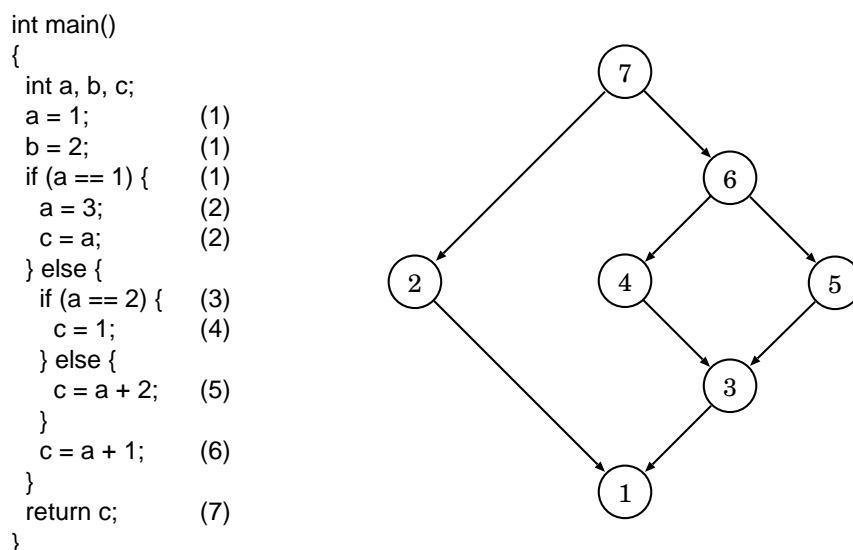


図 2.2: 逆転制御フローグラフの例

2.5 支配木

X と Y を制御フローグラフのノードとしたとき、制御フローグラフの入口ノードから Y に至る全ての路 (path) に X が現れるとき、 X は Y を支配する (dominate) という [1, 5, 7]。支配関係は反射的 (reflexive) であり、また、推移的 (transitive) でもある。よってノード X は自分自身を支配する。また、 X が Y を支配し、 Y が Z を支配するならば、 X は Z を支配する。

X が Y を支配し、かつ $X \neq Y$ のとき、 X は Y を厳密に支配する (strictly dominate) という。 X が Y を厳密に支配し、 X から Y への路に X 以外に Y を厳密に支配するノードがないとき、 X は Y を直接支配する といい、 $X = IDOM(Y)$ と表す。

ノード間の直接支配の関係を辺で表した木構造を 支配木 (dominator tree) という。 X の親 (parent) を $parent(X)$ 、子の集合を $Children(X)$ と表す。

図 2.1 の支配木を図 2.3 に示す。この例において、 $parent(3) = 1$ 、 $Children(3) = \{4, 5, 6\}$ である。

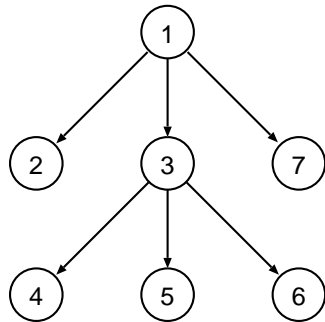


図 2.3: 支配木の例

2.6 逆支配木

X と Y を制御フローグラフのノードとしたとき、制御フローグラフのノード Y からプログラムの出口に至る全ての路 (path) に必ず X が現れるとき、 X は Y を後支配する (postdominate) という。支配関係は反射的 (reflexive) であり、また、推移的 (transitive) でもある。よってノード X は自分自身を後支配する。また、 X が Y を後支配し、 Y が Z を後支配するならば、 X は Z を後支配する。

X が Y を後支配し、かつ $X \neq Y$ のとき、 X は Y を厳密に後支配する (strictly postdominate) という。 X が Y を厳密に後支配し、 X から Y への路に X 以外に Y を厳密に後支配するノードがないとき、 X は Y を直接後支配するという。

ノード間の直接後支配の関係を辺で表した木構造を逆支配木 (postdominator tree) という [1, 5, 7]。

図 2.2 の逆支配木を図 2.4 に示す。

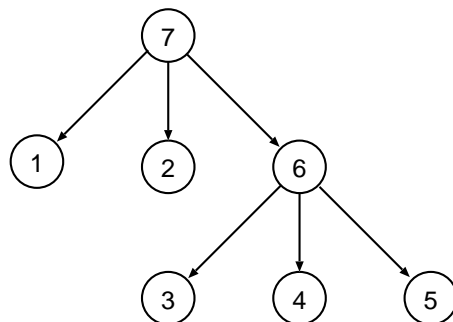


図 2.4: 逆支配木の例

2.7 支配境界・逆支配境界

制御フローグラフ上で、ノード X からグラフを辿っていき、初めて X の支配から外れたノードの集合のことを、 X の 支配境界(dominance frontier) という [1, 5, 7]。

2.7.1 支配境界の定義

支配境界の定義は次の通りである。

定義 1 (支配境界) 制御フローグラフのノード X の支配境界 とは、次の二つの条件を満たすノード Y の集合である。

1. X は Y の先行ノードの集合 $Pred(Y)$ のうちの少なくとも一つのノードを支配する。
2. X は Y を厳密に支配しない。

ノード X の支配境界 (*dominance frontier*) $DF(X)$ は次の式で定義される。

$DF(X) = \{Y | U \in Pred(Y) \text{ が存在し, } X \text{ は } U \text{ を支配し, } X \text{ は } Y \text{ を厳密な支配はしない}\}$

2.7.2 支配境界を求める例

例として図 2.1 と図 2.3 の支配境界をもとめてみると、以下のようになる。

$$DF(1) = \{\}$$

$$DF(2) = \{7\}$$

$$DF(3) = \{7\}$$

$$DF(4) = \{6\}$$

$$DF(5) = \{6\}$$

$$DF(6) = \{7\}$$

$$DF(7) = \{\}$$

2.7.3 支配境界を求めるアルゴリズム

支配木の各ノードの支配境界を求めるアルゴリズム。 X は支配木の節点とする。

2.7.4 逆支配境界

逆支配境界は、支配境界を求めるアルゴリズムを用いて、逆転制御フローグラフの支配木に適用すれば良い。図 2.2 と図 2.4 を用いて逆支配境界を求めてみる。PDF(X) を X の逆支配境界とする。

$$\text{PDF}(1) = \{\}$$

$$\text{PDF}(2) = \{1\}$$

$$\text{PDF}(3) = \{1\}$$

$$\text{PDF}(4) = \{3\}$$

$$\text{PDF}(5) = \{3\}$$

$$\text{PDF}(6) = \{1\}$$

$$\text{PDF}(7) = \{\}$$

2.8 制御依存グラフ

この節で制御依存グラフの説明をする [1, 5, 7]。

定義 2 (制御依存) 制御フローグラフの 2 つの節点 X 、 Y について、次の 2 つが成り立つとき Y は X に制御依存 (*control dependent*) するという。

1. X から Y への空でない路があり、 Y はその路の X より後のすべての節点を後支配する。
2. Y は X を厳密に後支配はしない。

制御依存グラフ (control dependence graph) は基本ブロックを節点とし、基本ブロック間の制御依存関係をグラフ化したものである。制御依存グラフに含まれる節点を N_{CDG} 、辺の集合を $E_{CDG} = \{CD(m, n) \mid m, n \in N_{CDG} \text{ かつ } n \text{ は } m \text{ に制御依存する}\}$ としたとき、その制御依存グラフは、両者の組 $CDG = \langle N_{CDG}, E_{CDG} \rangle$ で表すことにする。

Z が X に制御依存するとは、定義を言い換えれば、 X からのある辺を辿ると、その後必ず Z を通るが、別の辺を通れば Z は通らないことを表す。すなわち、 X でのある分岐を選択したら必ず Z を通ることになる。この関係を図で表すと、図 2.5 (b) のようになる。その図の矢印を逆向きにすると Exit が入口の役割をし、後支配関係は支配関係になる。それが図 2.5 の (c) である。この (c) と支配境界の関係を比べると次のことがわかる。

Z が X に制御依存することは、矢印を逆にしたグラフで $X \in DF(Z)$ であることと同じである。

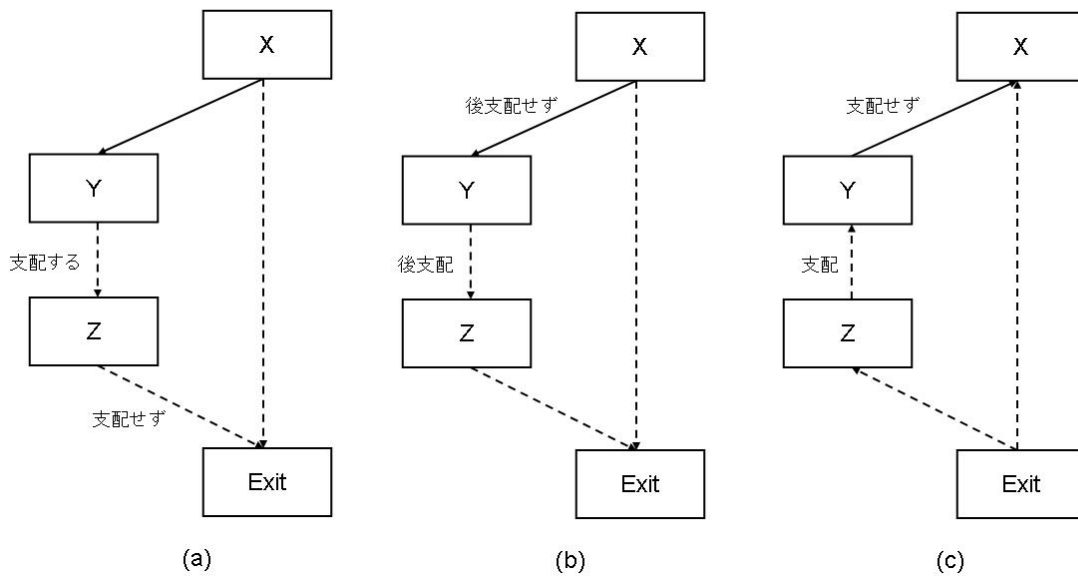


図 2.5: 制御依存と支配境界の関係

以上より、制御依存を計算するためには支配境界を求めるアルゴリズムを逆転制御フローグラフおよびその支配木に適用すればよい。

以下に、図 2.1 の制御フローグラフを用いて制御依存グラフの例をあげる。例は図 2.6 である。2.7.4 節の例で求めた逆支配境界より、次の様な制御依存関係が得られる。ここで、 $X \rightarrow Y$ は、Y が X に制御依存することを示す。

- 1 \rightarrow 2, 3, 6
- 3 \rightarrow 4, 5

2.9 データ依存関係

2.9.1 概要

文 s から文 t に対してデータ依存関係 $DD(s, t)$ があるとは、ある変数 w が存在して、文 s において定義された変数 w が、変数 w を使用している文 t に到達する場合をいう [1, 5, 7]。変数 $w \in Use(t)$ に関してデータ依存関係があることを明示する場合には、 $DD_w(s, t)$ と表すこととする。ここでいう到達するとはフローグラフ上で文 s から t に至る実行系列が存在し、その実行系列上では変数 w が再定義されないことをいう。

2.9.2 データ依存グラフ

データ依存関係をグラフにしたものをデータ依存グラフ (data dependence graph) という。

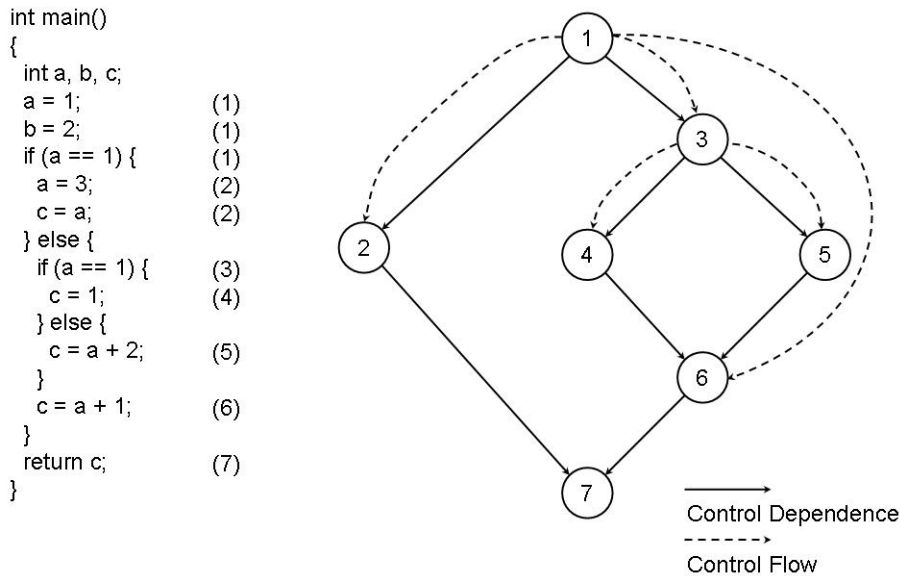


図 2.6: 制御依存グラフの例

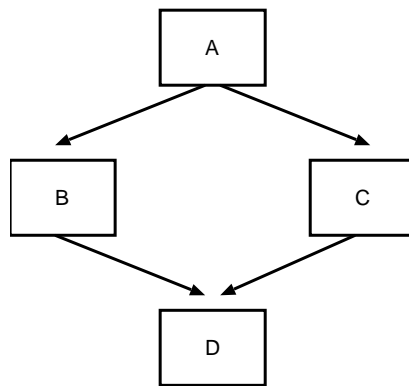


図 2.7: 制御フローグラフの例

データ依存グラフは各文を節点とし、文間のデータ依存関係をグラフ化したものである。データ依存グラフに含まれる節点を N_{DDG} 、辺の集合を $E_{DDG} = \{DD(m, n) | m, n \in N_{DDG}\}$ としたとき、そのデータ依存グラフは、両者の組 $DDG = \langle N_{DDG}, E_{DDG} \rangle$ で表すことにする。本論文で扱うグラフはデータ依存グラフであるので、今後データ依存グラフを単に依存グラフと呼ぶこととする。

以下に、制御依存グラフと同様、図 2.1 を用いたデータ依存グラフの例として図 2.8 をあげる、

2.10 プログラム依存グラフ

前述のとおり、プログラム依存グラフ PDG は、各文間の依存関係を解析することにより求めることができる。依存関係には制御依存関係とデータ依存関係があるので、

```

int main()
{
  int a, b, c;
  a = 1;      (1)
  b = 2;      (2)
  if (a == 1) { (3)
    a = 3;    (4)
    c = a;    (5)
  } else {
    if (a == 2) { (6)
      c = 1;  (7)
    } else {
      c = a + 2; (8)
    }
    c = a + 1; (9)
  }
  return c;   (10)
}

```

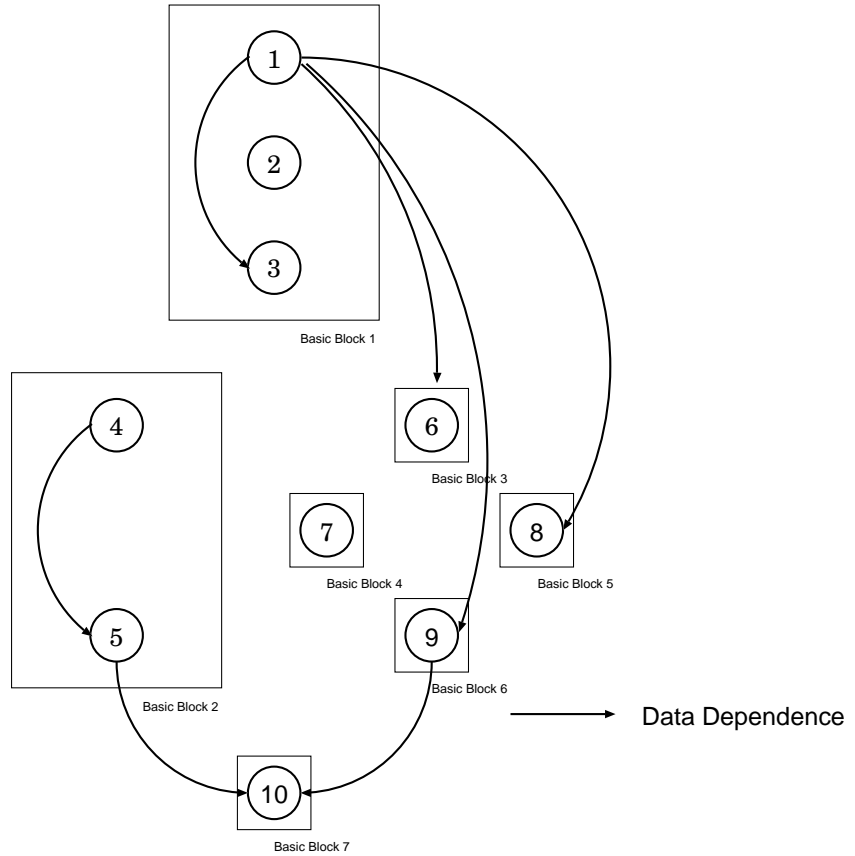


図 2.8: データ依存グラフの例

制御依存グラフとデータ依存グラフを求め、それらをマージしたものがPDGとなる。PDGの節点は文である。辺は文の間の依存関係を表している。プログラム依存グラフの節点を N_{PDG} 、辺の集合を $E_{PDG} = \{CD(m, n) \text{ または } DD(m, n) \mid m, n \in N_{PDG} \text{ かつ } n \text{ は } m \text{ に制御依存またはデータ依存する}\}$ としたとき、そのプログラム依存グラフは、両者の組 $PDG = \langle N_{PDG}, E_{PDG} \rangle$ で表すことにする。

しかし、 CDG の節点である N_{CDG} は基本ブロックを粒度とし、 DDG の節点である N_{DDG} は文を粒度としているため、制御依存グラフとデータ依存グラフを合わせるためにはギャップが生じる。

そこで、 CDG から粒度を文にした CDG_{st} というグラフを作り直すことにする。その後、 CDG_{st} と DDG をマージすることで PDG を作る。基本ブロックは、文の列で、基本ブロック中の文は先頭から最後まで一直線に実行されるため、 CDG から CDG_{st} を作るには CDG の各基本ブロックに対応した文を当てはめれば良いことになる。 CDG 上の辺は、ある基本ブロック X と Y に制御依存関係があることを表すが、 X の文の列の最後は分岐文なので、 CDG_{st} すなわち PDG の CDG 上の辺は、 X の最後の文から Y の各文に制御依存な辺が引かれることになる。

以下、図 2.6 と図 2.8 を合わせて PDG を作る例を示す。図は図 2.9 のようになる。

```

int main()
{
  int a, b, c;
  a = 1;      (1)
  b = 2;      (2)
  if (a == 1) { (3)
    a = 3;    (4)
    c = a;    (5)
  } else {
    if (a == 2) { (6)
      c = 1;  (7)
    } else {
      c = a + 2; (8)
    }
    c = a + 1; (9)
  }
  return c;   (10)
}

```

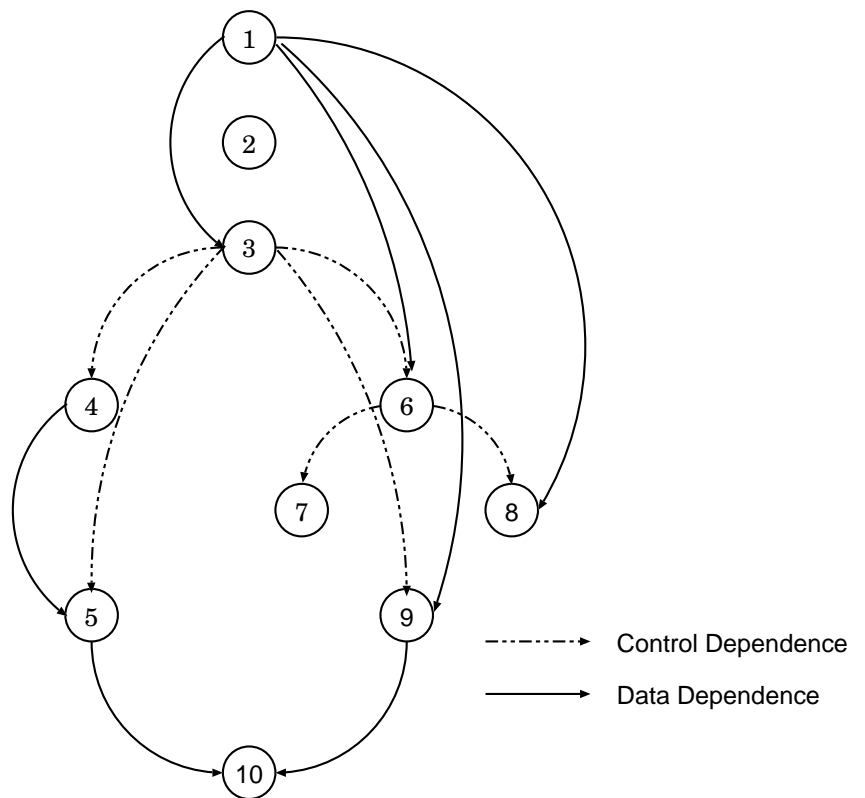


図 2.9: 文を単位としたプログラム依存グラフの例

第3章 データ依存とその移動

この章では、ループ並列化の妨げとなるデータ依存関係とその移動について詳しく説明する。

3.1 配列とデータ依存

本論文で扱うデータ依存の定義は前述のとおりである。この時並列度を向上するために妨げとなる依存は図 3.1 の実線で表した依存である。配列の添え字を見るとループを繰り返し越しての依存が発生していることが分かる。この時、ループの繰り返し越しての依存がどのくらい離れているかを依存距離と呼ぶことにする。以下、ループ繰り返し依存の定義である。

- **intra-iteration dependence**
依存距離が0であるデータ依存。ループを繰り返し越さずに同じイタレーション内で発生する依存関係。
- **inter-iteration dependence**
依存距離が0より大きいデータ依存。ループを繰り返し越して異なるイタレーションで発生する依存関係。この依存によって並列度の向上が妨げられている。

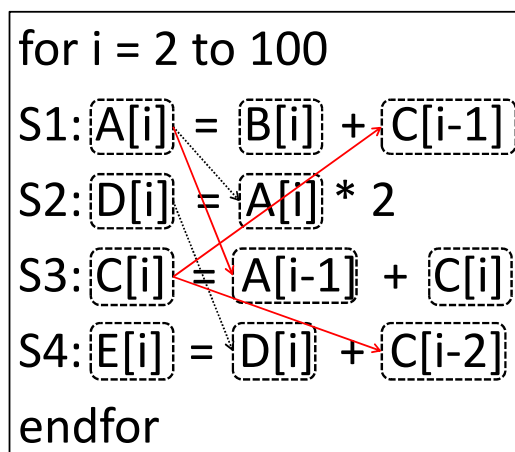


図 3.1: 配列を対象としたデータ依存グラフの例

ループの並列度を上げる為には、inter-iteration dependence を減らす事が重要である。inter-iteration dependence を減らす手法として、本研究ではループ内の処理の順番変更を行っている。これは、各イタレーションを飛び出しての依存をなるべく同じイタレーション内で実行するように最適化する処理である。このような処理を依存移動という。

3.2 依存グラフ

依存移動を説明する前に本研究で扱う重みつきデータ依存グラフについて説明する。以下、重みつきデータ依存グラフの定義である。

- 依存グラフ $G = (V, E, w)$
- ステートメントの集合 V
- 辺の集合 $E = \{(u, v) : u \rightarrow v \in V\}$
 (u, v) にデータ依存が発生している。
- 重み w
 ここで $w(u, v)$ という記述は辺 (u, v) 間の依存距離を表す。

このグラフを図で表したものが図 3.2 である。

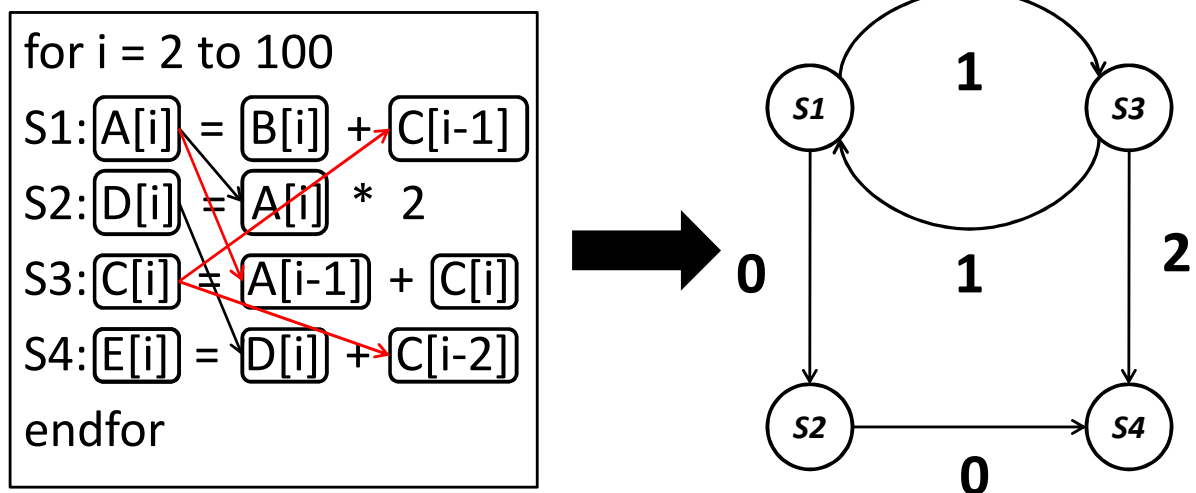


図 3.2: 重みつき依存グラフの例

3.3 隣接行列

今回グラフを表現するにあたって隣接行列といわれる手法を採用したので説明する。隣接行列とはグラフを実装する際の手法であり、グラフの接続関係を行列によって表す。以下、使用する行列を M とし、ノード i と j の関係を $M[i][j]$ によって表す。

3.3.1 無効グラフ

ノード i と j に辺がある場合、 $M[i][j]$ と $M[j][i]$ の値を 1 とし、辺が無い場合は 0 とする (図 3.3)。

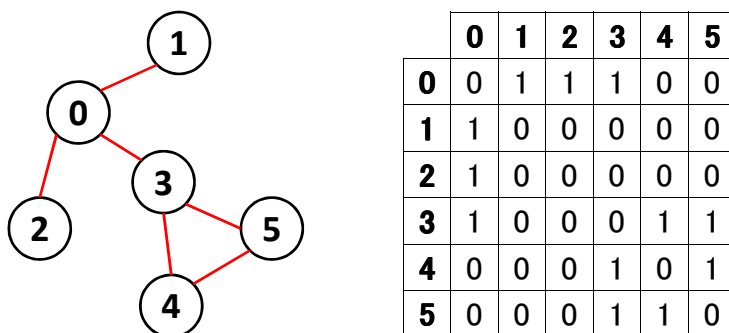


図 3.3: 無効グラフとその隣接行列の例

3.3.2 重みつき有効グラフ

今回採用する隣接行列がこれである。ノード i からノード j に向かって重み w の辺がある場合、 $M[i][j]$ の値を w とし、辺が無い場合は非常に大きな値とする (図 3.4)

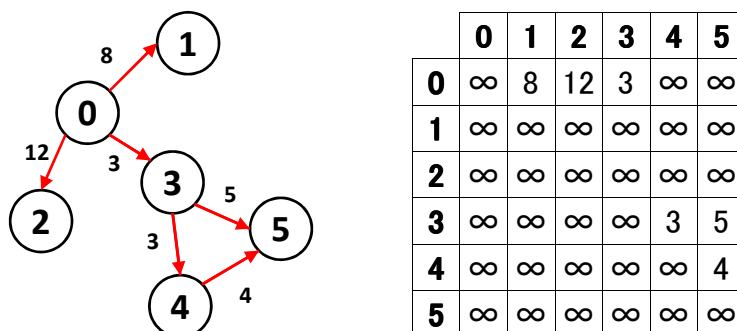


図 3.4: 重みつき有効グラフとその隣接行列の例

3.4 依存移動

3.4.1 依存移動の例

例として、図 3.1 で与えられたループの依存を依存移動によって解消する。基本的な手順は、まず、ループを解りやすいように依存グラフ化する。この後に依存関係を解消するために依存移動をする。依存移動をする際の目標は、重みが非ゼロである辺の中で重みが最小である辺の重みを最大化する事である。この計算アルゴリズムは次章で詳しく紹介する。例えば図 3.1 の重み $w(S1, S3) = 1$ を辺 $(S3, S1)$ と $(S3, S4)$ の重みに移動してやる (図 3.5)。

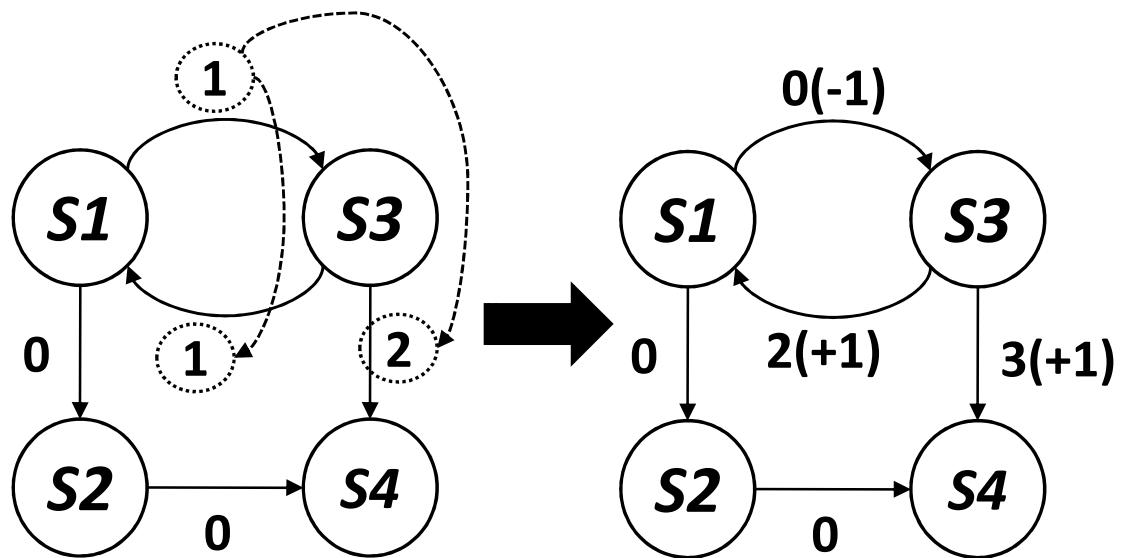


図 3.5: 依存移動の例

さらに、変化した依存グラフをループに戻すと、

```

/*Prologue*/
S3:C[2] = A[1] + C[2]

/*New loop kernel*/
for i = 2 to 99
S1:A[i] = B[i] + C[i-1]
S2:D[i] = A[i] * 2
S3:C[i+1] = A[i] + C[i+1]
S4:E[i] = D[i] + C[i-2]

/*Epilogue*/
S1:A[100] = B[100] + C[99]
S2:D[100] = A[100] * 2
S3:E[100] = D[100] + C[98]

```

のようになる。新しいループは Prologue という前処理部分、新しいループボディ、Epilogue という後処理部分で構成される。この時依存関係を調べてみると、S3 から S1、S4 へのデータ依存以外はすべて intra-iteration dependence になっている事がわかる。さらに、制御変数事に並べてみる (図 3.6)。下記の図では i が偶数と奇数で 2 つに分けられ、それぞれを別の CPU に割り当てる事で並列実行する事が出来る。よって並列度が 2 であることが分かる。

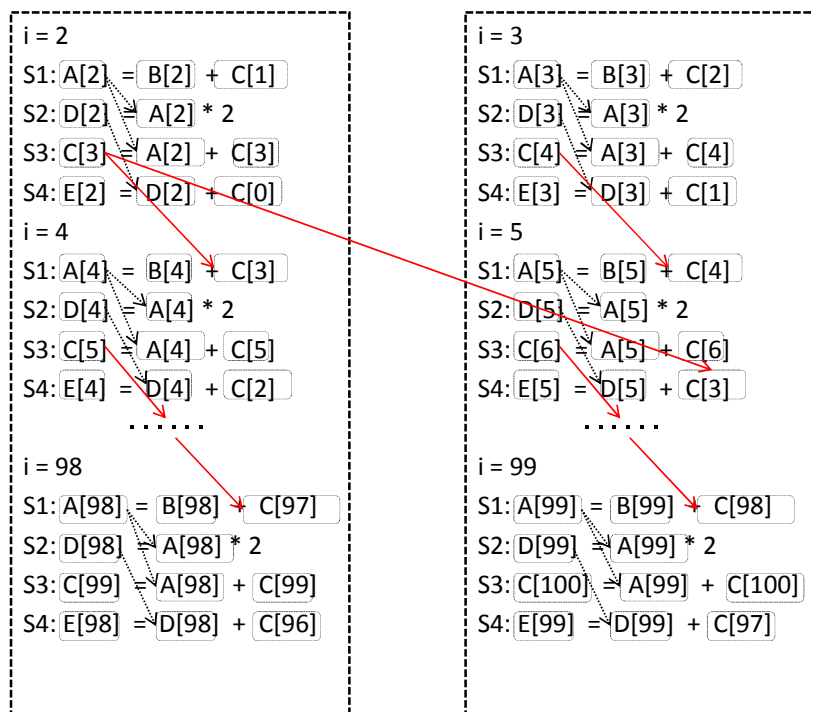


図 3.6: 制御変数毎の依存

3.5 OpenMP

本研究では、ループ並列化処理を OpenMP を用いて行った。OpenMP(Open Multi-Processing)とは、スレッド処理を行うための API であり、おもに共有メモリ型並列計算機で使用される [11]。OpenMp は使用できない環境では無視されるディレクティブを挿入することで並列化を行う。そのため並列環境と非並列環境でほぼ同じソースコードを使用できるという利点がある。ただし、非並列環境では実行できないコードを書くこともできてしまうため注意が必要である。

また OpenMP とは別に、MPI と呼ばれる規格も存在する。MPI(Message Passing Interface) は分散メモリ型計算機で使用され、メッセージの交換を明示的にプログラム中に記述しなければならない [12]。そのため、OpenMP とはちがひ MPI のライブラリが備わっていない非並列環境で並列計算プログラムを実行することは一般的に難しいとされる。

現在 OpenMP は、FORTRAN と C/C++ について標準化が行われている。

第4章 リタイミングによるループ最適化

この章ではリタイミングによる依存移動のアルゴリズムを、具体例を二つ挙げ詳しく説明していく。

4.1 リタイミング

リタイミングを以下で定義する。依存グラフと共に表すと図 4.1 である。

ループ $G = (V, E, w)$ が与えられたとき G のリタイミング r とは V のノード毎にマップされた整数の関数 $r(v)$ である。 $r(v)$ の値はノードに入ってくる辺から出て行く辺へと依存移動した重みの値である。また、リタイミング r に対して $G_r = \langle V, E, w_r \rangle$ をリタイミング r を作用したグラフとする。この時、 $w_r(u, v) = w(u, v) + r(u) - r(v)$ 、 $(u, v) \in E$ in G_r とする。

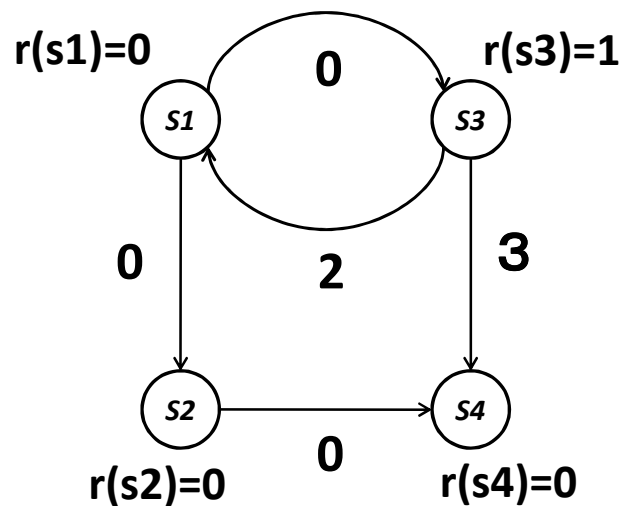


図 4.1: リタイミング関数の例

4.2 アルゴリズム

リタイミングを計算し、依存移動を行うアルゴリズムを説明する。このとき、ループは依存グラフに変換されているものとする。

4.2.1 強連結成分 (SCC)

強連結成分とは、ある頂点からすべての他の頂点にたどり着けるパスが存在する有向グラフのことである。具体的には次のようなものを言う。破線で囲まれた頂点の集合が1つの強連結成分となる。

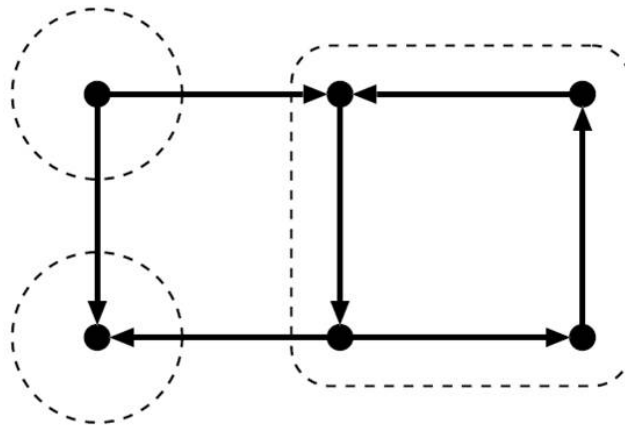


図 4.2: 強連結成分の例

Step1: グラフを SCC に分割しサイクルがあるかどうか判定する (図 4.3)。この時、各 SCC が一つのノードの場合と、一つの SCC がサイクルになっている場合で後に適用するアルゴリズムが違って来る。また、本研究ではグラフにした時サイクルが複数ある場合のループは対象外としている。

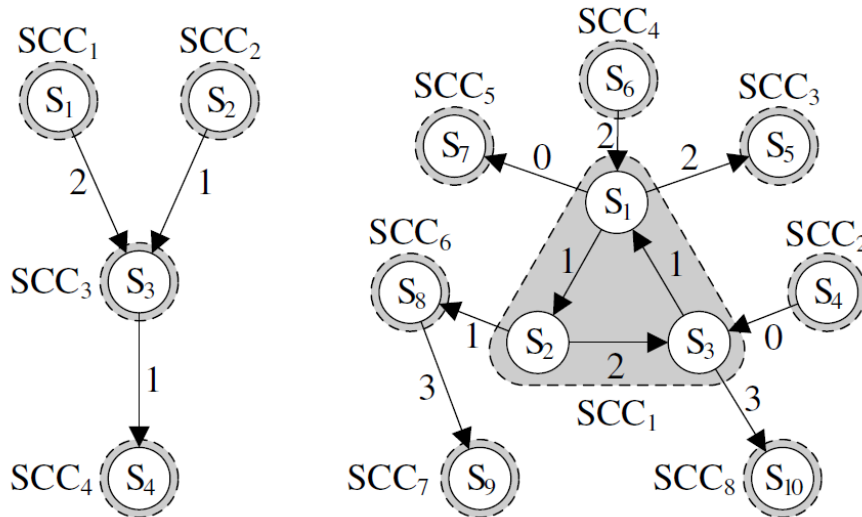


図 4.3: SCC によってグラフのタイプを分ける

4.2.2 DAG

全ての SCC が一つのノードから成るグラフを無閉路有向グラフ (DAG:Directed Acyclic Graph) という。以下、DAG に適用する依存移動のアルゴリズムである。

Step1: 全ての SCC が一つのノードで成る場合、まずある正の整数 δ を設定する。対象となるループは、この δ を並列度とするループに変換される。さらに、この δ を使い各ノードに対してリタイミングの制約 $r(u) - r(v) \leq w(u, v) - \delta$ を設けてやる (図 4.4)。

$$\begin{aligned} r(S_3) - r(S_1) &\leq -1 \\ r(S_3) - r(S_2) &\leq -2 \\ r(S_4) - r(S_3) &\leq -2 \end{aligned}$$

図 4.4: リタイミングの制約 ($\delta = 3$)

Step2: リタイミングの制約をもとに各辺の重みを変更する。さらに、各ノードに新たなノード S_0 を接続する。新たなノードから延びる辺の重みは0とする。この新しくできたグラフに対して、Bellman-Fordの探索アルゴリズム [10] を適用し S_0 から各ノードへの最短経路を導く (図 4.5)。

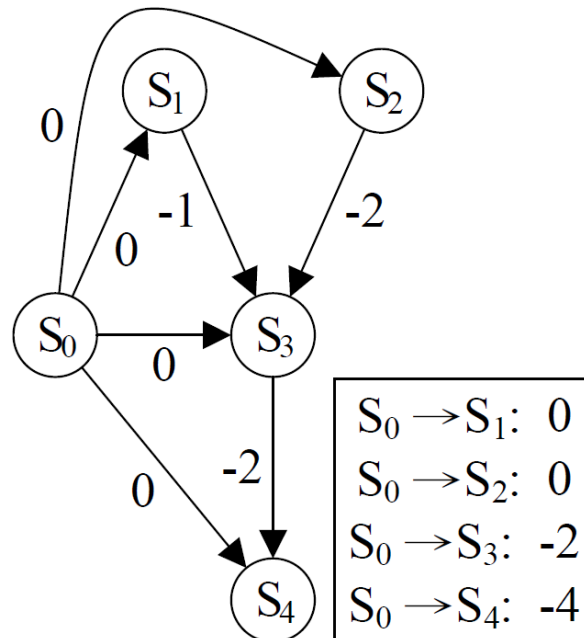


図 4.5: 新ノード追加と Bellman-Ford アルゴリズム適用

Bellman-Ford のアルゴリズム [10] 重みつき有向グラフ G が負の閉路を持たなければ、以下の方法で $s, t \in G$ 間の最短経路が求められる。全てのノードが SCC の場合にこのアルゴリズムを適用するが、この時主な実行時間はここで消費することとなる。Bellman-Ford のアルゴリズムは $O(n^3)$ であることが知られており、本研究のアルゴリズムも $O(n^3)$ となる。

- n を G のノード数とする。
- 高々 i 本の辺を用いる $v - t$ 間の最小コストを $OPT(i, v)$ とする。
- 漸化式 $OPT(i, v) = \min(OPT(i - 1, v), \min_{w \in V}(OPT(i - 1, w) + c_{vw}))$ を与える。ただし $(c_{vw}$ は v, w 間のコストである。
- 配列 $M[0 \dots n - 1, V]$ をあたえ、 $M[0, t] = 0$ とし t 以外の各点 $v \in V$ で $M[0, v] = \infty$ とする。
- 各 $i = 1, \dots, n - 1$ と $v \in V$ について上記の漸化式を計算する。
- $M[n - 1, s]$ を返す。

Step3 S_0 から各ノードへの最短経路を通った時の重みを各ノードのリタイミング関数とし、それを適用する。この時新たな重みを $w_r(u, v) = w(u, v) + r(u) - r(v)$ で再定義してやる。(図 4.6)

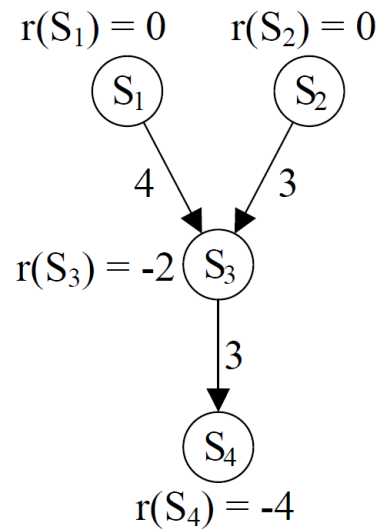


図 4.6: リタイミング関数の決定と適用

Step4 各ノードのリタイミング関数のうち負のものがあれば、その中で最小のリタイミング関数の値を加算してやる事で 0 にする。また、その分他のリタイミング関数の値にも同じ数を足してやり最終的なリタイミング関数とする (図 4.7)。

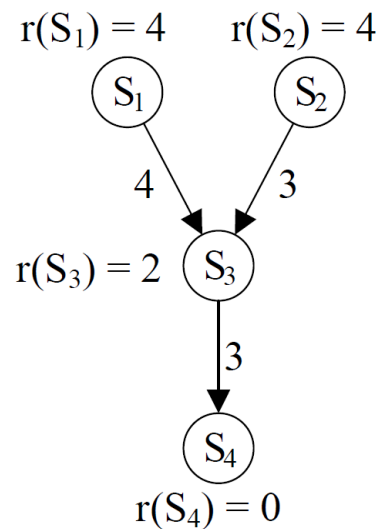


図 4.7: リタイミング関数の値を標準化

4.2.3 CGSC

一つのSCCがサイクルとなっているグラフを本論文ではCGSC(Cyclic Graph with Single Cycle)と呼ぶこととする。以下は、CGSCにリタイミングを適用するアルゴリズムである。また具体例として図4.3での右のグラフを挙げる。

Step1: θ をサイクルを構成する辺の重みの合計とし、サイクル内で一番大きな重みを持つ辺に依存を移動させる。この際にサイクル内のリタイミング関数と重みを固定する(図4.8)。ここで、サイクル内の重みを固定するのは、後にDAG用のアルゴリズムを呼び出しているためである。DAG用のアルゴリズムでBellman-Fordのアルゴリズムを適用しているため、負の経路を持ってないからである。

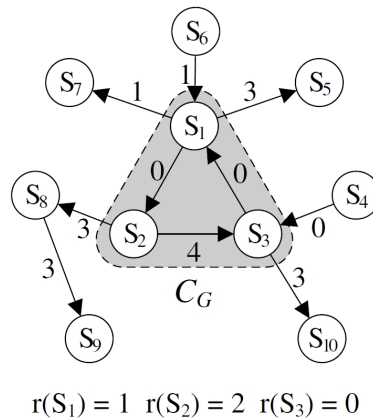


図 4.8: サイクル内の重みの操作と固定

Step2: サイクル内の重みを固定したままグラフにDAG用のアルゴリズムを適用し、リタイミング関数を計算する。この時、 θ の値として $\theta = 4$ を渡してやる(図4.9)。この事で、CGSCはサイクルの重みを並列度としたループに変換されることが分かる。

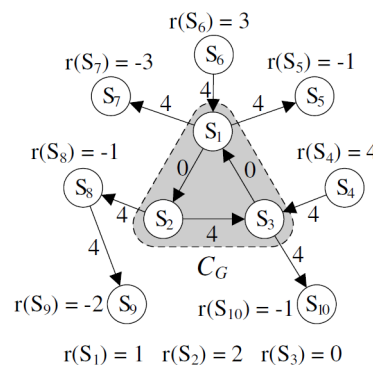


図 4.9: DAG アルゴリズムの適用

Step3: DAG用のアルゴリズムと同じようにリタイミング関数の値を標準化し、最終的なリタイミング関数を得る (図 4.10)。

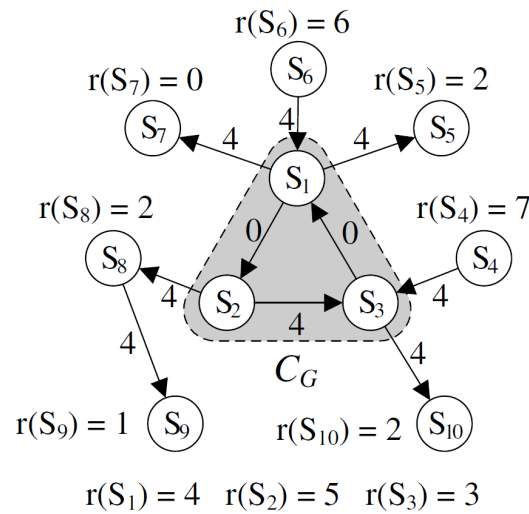


図 4.10: リタイミング関数の標準化

4.2.4 ループ逆変換

上記のアルゴリズム等で得られた、新しいグラフとリタイミング関数により再度ループを構築するアルゴリズムについて説明する。 M を最適化前のループ制御変数の最大値、 L を最小値とする。 r_{max} をリタイミング関数の最大値とする。

Step1: Prologue の生成 新しいループの前処理部を生成する。最適化後のグラフ G_r の各ノード u について、ループ制御変数を i とした時、 L から $L + r(u) - 1$ について、 i での処理を Prologue での処理とする。

Step2: 新しいループボディの生成 新しいループボディを生成する。リタイミングの依存移動によるループ制御変数の増分を記述する。

Step3: Epilogue の生成 新しいループの後処理部を生成する。最適化後のグラフ G_r の各ノード u について、ループ制御変数を i とした時、 1 から $r_{max} - r(u)$ について、 $M - i + 1$ での処理を Epilogue での処理とする。

第5章 並列化コンパイラ向け共通インフラストラクチャCOINS

5.1 COINSの目的

COINS (COmpiler INfraStructure) はコンパイラを構成する基本機能のモジュールを備え、それらの組み合わせを変えたり、一部のモジュールを新たに開発するだけで、新しいコンパイラを実現することができるようなコンパイラ・インフラストラクチャである [3, 8]。それらの仕様もプログラムもすべて公開され、自由に使えるようになっており、それによって、多くの人に使ってもらえる共通のインフラストラクチャとなることを目的としている。

5.2 COINSの構成

一般にコンパイラはフロントエンド (front end) とバックエンド (back end) から構成される。フロントエンドは原始プログラム (source program) を中間コード (intermediate code) と呼ばれる内部形式に変換する。バックエンドは中間コードを計算機の機械コードに変換する。フロントエンドはさらに字句解析器 (lexical analyzer)、構文解析器 (syntax analyzer)、意味解析器 (semantic analyzer) に分けられる。バックエンドは最適化器 (optimizer) とコード生成器 (code generator) に分けられる。これらの各部分はコンパイラのフェーズと呼ばれる。

COINS の概念図を図 5.1 に示す。COINS では、複数の入力言語、複数の目的機種に対応する 2 つの中間表現がある。入力言語の論理構造に近いレベルの中間表現を高水準中間表現 (high-level intermediate representation, HIR) と呼び、機械語に近いレベルの中間表現を低水準中間表現 (low-level intermediate representation, LIR) と呼ぶ。

5.3 中間表現のレベル

プログラムの解析や変換では、手続き単位やループ単位で並列性を認識するとか、ループ変換やインライン展開など、大きいかたまりを単位として扱う場合がある。また、並列化などの変換をした結果をまたソースプログラムの形で出力する場合もある。このような場合には、入力プログラムの構造を反映できる中間表現が求められる。一方、コンパイラでは、レジスタ割付けや命令スケジューリングなど、細か

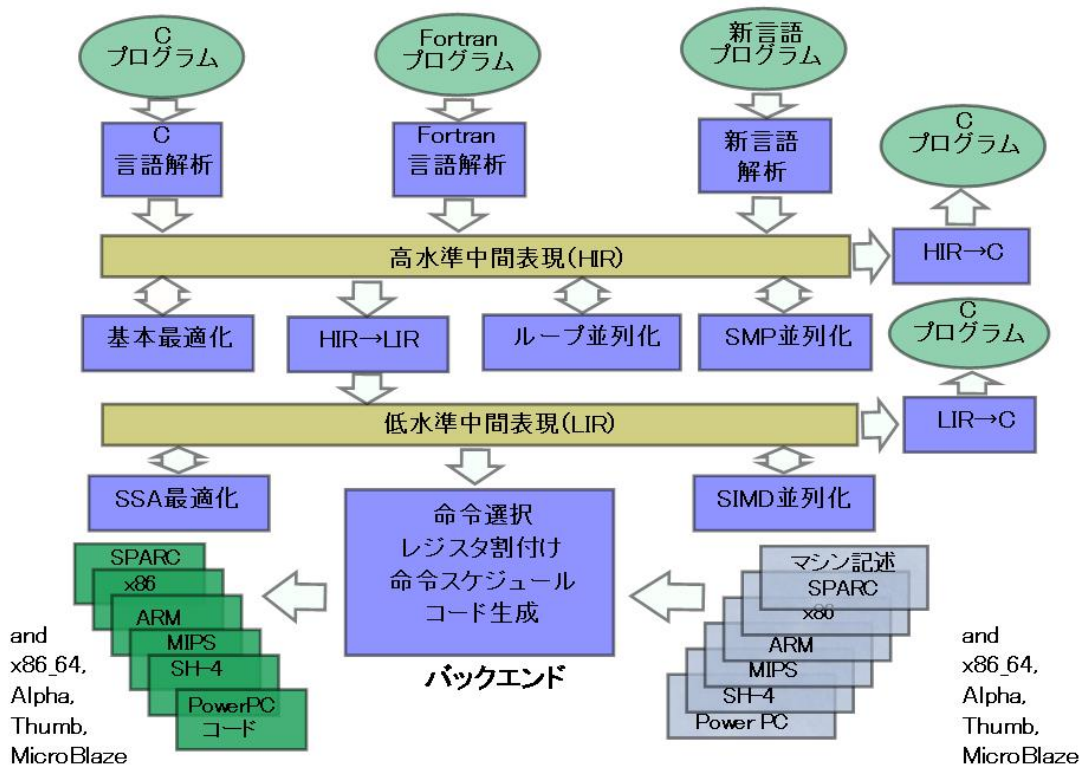


図 5.1: 並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図

いレベルでの処理も要求され、そのような場合は機械命令に近い中間表現が求められる。そこで、COINS の中間表現は前述のように、入力プログラムの構造を反映できる高水準中間表現と、計算機による処理の特性を反映できる低水準中間表現の 2 つが設定されている。各中間表現の具体的利用形態としては、次のようなものが想定されている。

- 高水準中間表現 (HIR) 向き処理
 - 逐次実行型プログラムから並列実行型プログラムへの変換
 - ループ解析 (配列要素の依存関係、並列実行性抽出、他)
 - ループ変換 (ループ展開、タイリング、他)
 - インライン展開
 - ポインタ解析
 - 各種プログラミングツールの開発
- 低水準中間表現 (LIR) 向き処理
 - レジスタ割付け
 - 詳細レベルの (データのアクセスパスを考慮した) 最適化
 - 命令スケジューリング
 - 命令レベル並列化 (マルチメディア処理向き並列化、VLIW 並列化、他)
 - 計算機資源 (レジスタ、演算器等) に合わせた最適化、並列化

ピープホール最適化

本論文の目的はループ並列化 (逐次実行型プログラムから並列実行型プログラムへの変換) であるので、実装は HIR 上で行った。次節で更に詳しく HIR について説明する。

5.4 高水準中間言語 HIR について

本節では、本研究で使用する中間表現である HIR について説明する。

5.4.1 概要

HIR は一般的な手続き型言語において共通的な概念を抽出し、特定の言語に依存しないように表現された抽象構文木である。HIR 自身は Java 言語で実装される。

5.4.2 具体表現

HIR をテキストで表すとき、節は

(オペレータ 型 第1子 第2子 … 第n子)

葉は

< 種別 記号 型 >

と表す。HIR ではコンパイル単位全体を1つの木として表現する。その中には一般に、複数の副プログラム定義が含まれる。例えば次のプログラムがあったとする。

プログラム 5.1 (入力プログラム)

```
int fact(int p) {  
    if (p <= 1)  
        return 1;  
    else  
        return p * fact(p - 1);  
}
```

このプログラムから生成される HIR は図 5.2 のようになる。

```

(prog
  <null 0 void>
  <nullNode>
  (subpDef void
    <subp <SUBP < int > false false int> fact>
  <null void>
    (labeldSt void
      (list <labelDef _lab1>)
      <block void
        (if void
          (cmpLe bool <var int p> <const int 1>)
          (labeldSt int
            (list <labelDef _lab3>)
            (return int <const int 1>))
          (labeledSt int
            (list <labelDef _lab4>)
            (return int
              (mult int
                <var int p>
                (call int
                  (addr <PTR <SUBP < int > false false int>>
                    <subp <SUBP < int > false false int> fact>)
                  (list
                    (sub int <var int p> <const int 1>))))))))
            (labeledSt void
              (list <labelDef _lab5>)
              <null void>))))))
  )

```

図 5.2: プログラム 5.1 の HIR

第6章 実験と評価

6.1 実装

6.1.1 対象となるループ

今回、対象としたループの条件は以下である。

- ループの種類は for 文とする。
- ループ内の構成が一つの基本ブロックとなっている。つまり、ループ外飛び出しや分岐が無い。
- 2個以上のループの重なりが無い、つまりネストしていない。
- グラフ化したときに、サイクルが2つ以上できない。

6.1.2 HIR 上での実装

本手法で、ループの最適化が出来ているかどうか確かめるために、Cプログラムから COINS 上で HIR に変換し、HIR 上で最適化を行った後に C プログラムへの変換を行っている。最適化を確認した後に、またコンパイラを通して機械コードを生成している。例えば図 6.1 の for 文を入力とした場合、図 6.2 のようなソースが生成される。

```
for(i = 1; i < 100000; i++){  
a[i] = c[i-1];  
b[i+3] = a[i-1] + 2;  
c[i] = b[i-1];  
}
```

図 6.1: プログラムの例

```

b[(hir__ADD((hir_t_int )2,(hir_t_int )3))] =
(hir__ADD(a[(hir__SUB((hir_t_int )2,(hir_t_int )1))],(hir_t_int )2));
b[(hir__ADD((hir_t_int )1,(hir_t_int )3))] =
(hir__ADD(a[(hir__SUB((hir_t_int )1,(hir_t_int )1))],(hir_t_int )2));
a[(hir_t_int )1] =
c[(hir__SUB((hir_t_int )1,(hir_t_int )1))];
for ( i = (hir_t_int )1;hir__LT(i,(hir_t_int )99998);
i = (hir__ADD(i,(hir_t_int )1))
{
    {
        c[(hir__ADD(i,(hir_t_int )0))] =
b[(hir__SUB((hir__ADD(i,(hir_t_int )0)),(hir_t_int )1))];
        a[(hir__ADD(i,(hir_t_int )1))] =
c[(hir__SUB((hir__ADD(i,(hir_t_int )1)),(hir_t_int )1))];
        b[(hir__ADD((hir__ADD(i,(hir_t_int )2)),(hir_t_int )3))] =
(hir__ADD(a[(hir__SUB((hir__ADD(i,(hir_t_int )2)),
(hir_t_int )1))],(hir_t_int )2));
    }
}
_lab4:;
c[(hir_t_int )99998] = b[(hir__SUB((hir_t_int )99998,(hir_t_int )1))];
c[(hir_t_int )99999] = b[(hir__SUB((hir_t_int )99999,(hir_t_int )1))];
a[(hir_t_int )99999] = c[(hir__SUB((hir_t_int )99999,(hir_t_int )1))];

```

図 6.2: 図 6.1 の最適化後のソース

6.2 実験

テストプログラムは Liu らの用いたテストプログラム [6] を使用した。また、実験の方法として、マルチコアの計算機上で、本手法を用いた場合と用いずにコンパイルした場合とで比べた。使用したコンパイラは gcc である。gcc でコンパイルする際に openMP のプラグマを挿入し openMP による並列実行をおこなった。実行時間の 10 回分の平均値を取った。実験環境は表 6.2 に示す。

| | |
|---------|---------------------|
| プロセッサ種別 | Intel Xenon 2.93GHz |
| コア数 | 最大 8 コア |
| メモリ | 54Gbyte |

表 6.1: 実験環境

結果は表 6.2 である。グラフは図 6.3、6.4、6.5 にしめす。

| コア数 | 1 | 2 | 4 | 8 |
|----------|--------|--------|--------|--------|
| bre | 2.918s | 2.798s | 2.728s | 2.869s |
| bre 最適化後 | 2.896s | 2.800s | 2.846s | 2.862s |
| reo | 2.868s | 2.881s | 2.862s | 3.016s |
| reo 最適化後 | 2.910s | 2.855s | 2.884s | 2.861s |
| shr | 2.949s | 2.886s | 2.855s | 2.896s |
| shr 最適化後 | 2.949s | 2.910s | 2.872s | 2.896s |

表 6.2: 実験結果

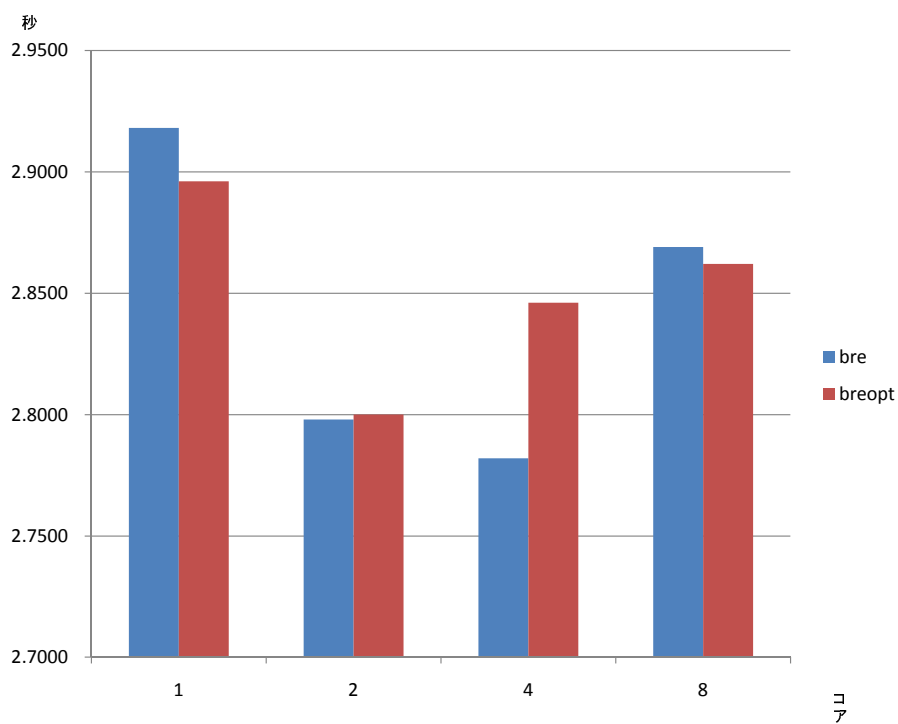


図 6.3: プログラム bre について

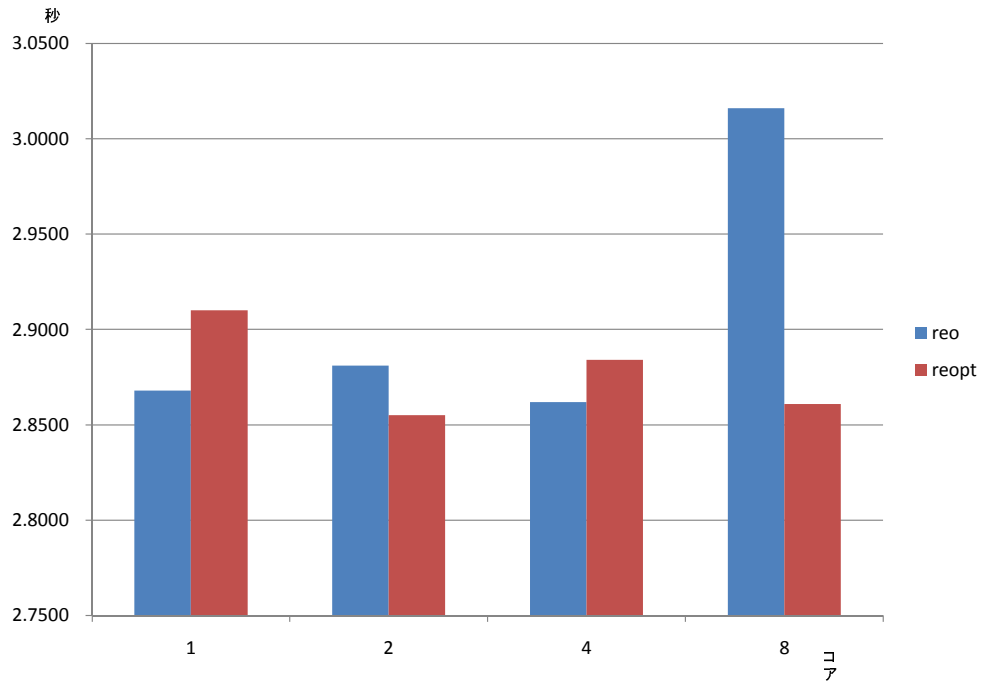


図 6.4: プログラム reo について

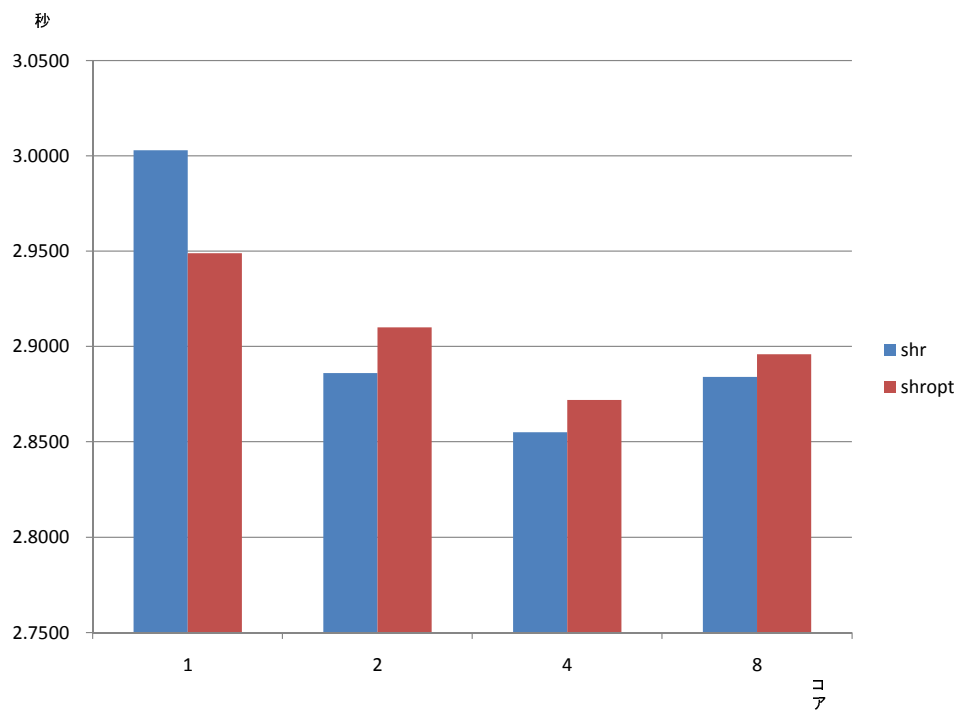


図 6.5: プログラム shr について

6.3 考察

本手法を適用した場合の実験結果が思うようになかった。以下で原因を考察していく。

6.3.1 ループのサイズについて

今回実験に用いたループは1000回の繰り返しを行っている。また、プログラム自体もかなり簡単なもので、時間がかかる処理はループ以外では行っていなかった。よって、サイズが小さいために10回の平均をとってもさほど大きな差は出なかったのではないかと考える。

6.3.2 アルゴリズムの欠点

本手法では、リタイミングによる依存移動により並列度は上がっているものの、完全に inter-iteration dependence を解消しているわけではない。よって制御変数によっては、同期が必要な場合がある。そのためにうまく並列度の向上を生かせなかったのではないかと考える。

バリア同期 バリア同期とは OpenMP ではプログラム中にディレクティブを挿入することで明示的に同期処理を行うことができる。バリア同期とはプログラム中にバリアを張ってやることにやり、すべてのスレッドがそのバリアを通るまで処理を待たせることができる。本手法ではバリアを張ることまではできなかったが今後の課題として挙げておきたい。バリア同期のイメージは以下に示す (図 6.6)

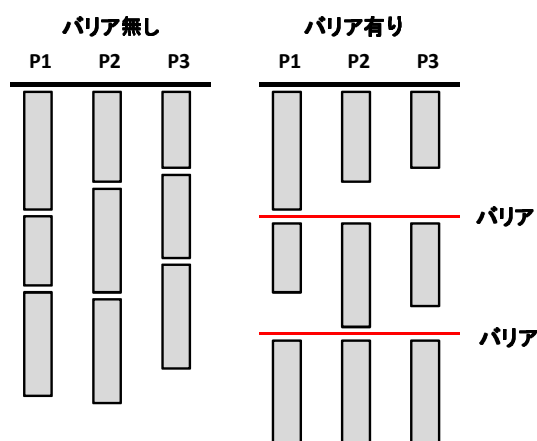


図 6.6: バリア同期のイメージ

6.3.3 実験の方法

今回の実験では OpenMP で並列化指示を出すことにより並列化を試みたが、本手法による変形ではスレッド化がうまくいっていない可能性がある。プログラム内にさらなる OpenMP の指示分を追加してやることにより並列化がうまくいくものと考えられる。例えば本手法で上がった並列度によりループ制御変数毎に CPU を明示的に割り当てれば実行コードの実行速度向上が期待できる。図 6.7 に for 文の明示的な分割を示す。

```
for i = 2,4,6,...
S1: A[i] = B[i] + C[i-1]
S2: D[i] = A[i] * 2
S3: C[i] = A[i-1] + C[i]
S4: E[i] = D[i] + C[i-2]
endfor

for i = 3,5,7,...
S1: A[i] = B[i] + C[i-1]
S2: D[i] = A[i] * 2
S3: C[i] = A[i-1] + C[i]
S4: E[i] = D[i] + C[i-2]
endfor
```

図 6.7: for 文の明示的な分割

6.4 今後の課題

6.4.1 対応できないループについての取り組み

ネストしたループはもちろんの事、並列対象のループを増やしていく事で結果の向上が望めるであろう。今回、対象外であったループで特徴的なものを説明する。

一つの辺に依存が2つ以上ある場合 図 6.8 をご覧いただきたい。

左がループの本体で右が依存グラフ化したものである。ノード S1 から S3 へとのびる依存が2種類あり重みが定まらずループ最適化が出来ない状態である。この場合は一時変数を用い、ノードを増やし、ループを変化してやる事でループ最適化が可能になる (図 6.9)。

```

for l = 2 to 100
S1: A[i] = B[i] + C[i-1]
S2: D[i] = A[i]*2
S3: C[i] = A[i] + A[i+1]
S4: E[i] = D[i] + C[i-2]
endfor

```

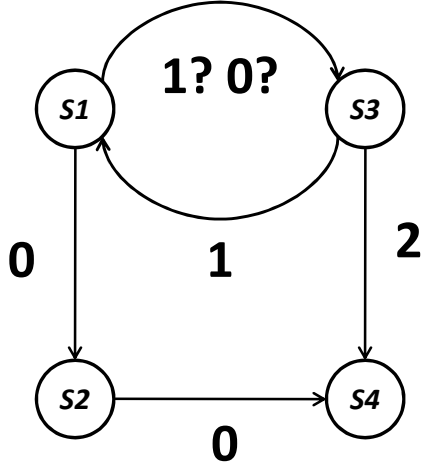


図 6.8: 今回対象外のループの例 1

```

for l = 2 to 100
S1: A[i] = B[i] + C[i-1]
S2: D[i] = A[i]*2
S3: temp[i] = A[i]
S3': C[i] = temp[i]+A[i+1]
S4: E[i] = D[i] + C[i-2]
endfor

```

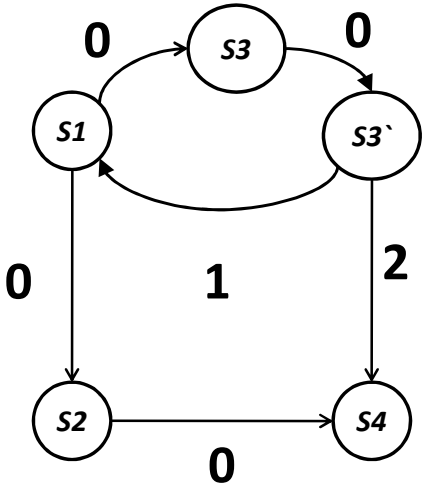


図 6.9: ループ最適化への対処

配列添え字の制御変数への処理が加算、もしくは減算以外 以下の図 6.10 の S3 については対処法が発見できなかった。配列の添え字の制御変数への処理が乗算となっており、依存距離が一定に定まらなくなっており、今回使ったアルゴリズムではループ最適化できない。

```

for l = 2 to 100
S1: A[i] = B[i] + C[i-1]
S2: D[i] = A[i]*2
S3: C[i] = A[i*2]
S4: E[i] = D[i] + C[i-2]
endfor

```

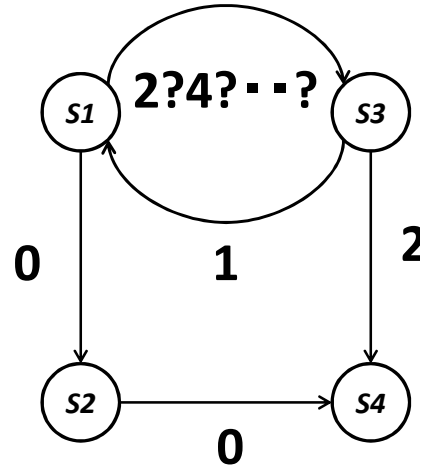


図 6.10: 今回対象外のループの例 2

第7章 関連研究

現在、ループを対象とした並列化の研究は多く行われている。本手法に関連の深い研究をいくつか紹介する。

7.1 Liuらの研究

今回のアルゴリズムは彼らが提案したものである。[6] 本研究では実装していない、マルチサイクルのグラフも最適化の対象としている。しかし、評価の指標はどのくらい並列度が向上したかであり、計算機上での実行速度実験などは行っていない。本論文の実験の結果から考察すると、計算機上でパフォーマンスを発揮するにはいろいろな制約が付きまといそうである。

7.2 COINSでのループ並列化

COINSではループ並列化でも本論文のようなデータ依存を計算している。しかし対象とするループはDOALL型のループで繰り越し依存がないものであるため、かなり条件が限定されてしまう。[4] この研究では、OpenMPのコードを出力させて共有メモリ型並列計算機で実験を行い結果を残している。

第8章 まとめ

本研究では、ループ内の配列の依存関係を解消する事で並列度が向上するアルゴリズムを紹介し、その効果の検証を行った。COINS上で実装を行い実験を行ったが、効果はさほど出なかったのが残念である。原因としては、ループ外への依存をどうしても解消しきれないために計算機上でデータの同期を必要としてしまうことがあげられる。また、今回対象としていないループにこのアルゴリズムを適用することにより効果の改善が見込めると考えている。

謝辞

本研究を進めるにあたり多大なるご指導ご鞭撻をいただいた，東京工業大学数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します。

また，COINS のループ並列化、データ依存の計算について詳しくご教授いただきました電気通信大学の渡辺担教授、拓殖大学の岩澤京子教授にも深く感謝の意を表します。

また様々な面でサポートをいただきました佐々研究室の皆様に深くお礼申し上げます。

参考文献

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools 2nd ed.* Addison Wesley, 2006.
- [2] Andrew W. Appel. *Modern Compiler Implementation in Java second edition.* Cambridge University Press, 2002.
- [3] COINS Project. COINS home page. <http://www.coins-project.org/> .
- [4] COINS Project. COINS home page. ループ並列化. <http://www.coins-project.org/COINSdoc/hirOpt/parallel-frame.html> .
- [5] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.
- [6] Duo Liu, Meng Wang, Minyi Guo, Jingling Xue. *Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism.* Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp.67-76, 2009.
- [7] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [8] 中田育男. 渡邊坦. 佐々政孝. 滝本宗宏. コンパイラの基盤技術と実践-コンパイラ・インフラストラクチャ COINS を用いて-. 朝倉書店, 2008.
- [9] 佐々政孝. プログラミング言語処理系. 岩波書店, 1989.
- [10] 浅野孝夫, 浅野泰仁, 小野孝男, 平田富夫. アルゴリズムデザイン. 共立出版, 2008.
- [11] OpenMP.org. OpenMP home page. <http://openmp.org/>
- [12] Message Passing Interface Forum. MPI forum home page. <http://www.mpi-forum.org/>