成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.51 No.1 (2014)pp.35-42

再帰的なデータ構造を扱うＣ言語プログラムのための データ依存解析手法の提案

武市　和真[*1]，甲斐　宗徳[*2]

A Method of Data Dependency Analysis for C Program with Recursive Data Structures

Kazuma TAKECHI[*1], Munenori KAI[*2]

ABSTRACT：Recently, the necessity for parallel programming has been increased with the rapid spread of multicore/multiprocessor systems. However, it is difficult for a programmer to create a highly-effective and high-performance parallel program. So, we are developing the automatic translator from C programs to parallel programs using MPI(Message Passing Interface). In our conventional automatic parallelism analysis of C program with pointer variables, it was able to analyze the data dependencies of only the pointer variables declared explicitly in the code. In this research, we have first applied the Shape analysis to C programs with pointer variables for getting to know the form of the recursive data structures which will be allocated and constructed dynamically at the time of execution. Then, using the result of the analysis, we can analyze the data dependencies of recursive data structures, such as linked list or binary tree structures, and detect more parallelism from C program.

Keywords：parallel processing, parallelism analysis, automatic program translator, recursive data structures

(Received March 31, 2014)

１．はじめに

近年のマルチコア化により並列コンピューティングの必要性が高まっている。並列コンピューティングでは複数のプロセッサやコアを効率よく協調させる並列プログラムであれば十分な計算処理性能を得られるが，そうでなければ逐次処理に比べて処理性能を低下させてしまう可能性がある。そのため並列コンピューティングで高い処理性能を得るには，逐次プログラミングの知識に加えて並列プログラミングの知識が必要となる。これは開発者にとって負担が大きくなると考えられる。

このような問題を解決するための手段として逐次プログラムを自動で並列プログラムに変換する自動並列化トランスレータの利用が望まれる。何故なら，逐次プログラムの自動並列化が可能になれば既存の逐次プログラムを手軽に利用できると考えられるからである。

また，Ｃ言語ではポインタを使用した自由な記述が可能であり，並列性の解析が困難である。そこで本研究ではこのポインタに注目し並列性の解析を行う。

２．Ｃ言語自動並列化トランスレータの概要

Ｃ言語自動並列化トランスレータでは中間データ構造を生成する。そしてＣ言語で記述された逐次実行可能なソースプログラムを読み込み，構文木構造を解析し抽象構文木として中間データ構造に保存する。この中間データ構造に対してプログラム内に存在する並列性を抽出し，中間データ構造に並列化のための情報を保存して抽象構文木の更新を行っていく。最後に中間データ構造の抽象構文木と並列化のための情報を使い並列プログラムの出力を行う。また，並列効果を高めるために並列性を抽出した後に，ループの分割や実行時間を用いたスケジューリングなどの最適化処理を行うことで，より並列効果の

[*1]：理工学研究科理工学専攻博士前期学生
[*2]：理工学研究科理工学専攻教授（kai@st.seikei.ac.jp）

高いコードの生成を行う[1]。

本研究では，共有メモリと分散メモリの両方のメモリ方式に対応した並列プログラミングライブラリMPI(Message Passing Interface)を使うことで，大規模な並列実行環境でも利用可能にすることを目標としている。また並列実行可能なプログラムをソースコードレベルで生成することにより機種非依存で，ユーザによる独自のチューニングやコンパイラによる最適化処理を施すことが可能になる[1]。

3．C言語自動並列化トランスレータの処理手順

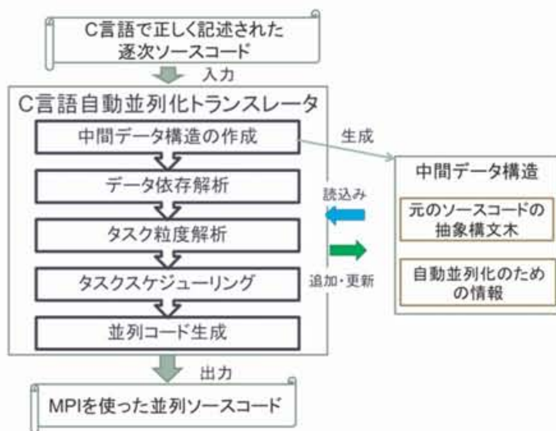

Figure 3.1　C言語自動並列化トランスレータの処理手順

Figure 3.1 はC言語自動並列化トランスレータの処理手順を示したものである。最初に逐次プログラムを読み込み，構文解析により抽象構文木を持った中間データ構造の生成を行う。以降の処理はこの中間データ構造に対して行う。

中間データ構造生成後は中間データ構造に対して並列化可能箇所を解析するためにデータ依存解析を行う。初期段階ではタスク粒度がステートメント単位であるためタスク数が多く，後述するタスクスケジューリングに時間がかかってしまう。そのためタスク粒度解析を行うことでタスク数を適切に減少させる。タスク粒度解析を行った後は処理をまとめることにより通信回数を減らすなど，実行時間を短縮するための変換を行うためにタスクスケジューリングを行い，各プロセッサに対してタスクを適切に割り当てる。ここではタスク粒度解析を行う前の段階なのでタスクとはステートメントのことを指す。

以上の解析結果の情報を持った中間データ構造を基にコード生成を行うことで並列プログラムが生成される。

4．データ依存解析

データ依存関係が存在するタスクは処理する順番が変わってしまうと結果も変わってしまうため，逐次処理を行わなくてはならない。このデータ依存関係を解析するためにそれぞれのタスク内でメモリロケーションのアクセス属性を解析する。メモリロケーションのアクセス属性には読み込み(read)，書き込み(write)があり，この組み合わせから 3 種類のデータ依存関係を解析することができる。Figure 4.1 ではいずれも変数Xによるデータ依存関係が発生しているため逐次実行する必要がある。

Figure 4.1　データ依存関係の種類

逆にこれらのようなデータ依存関係が無ければタスク同士は並列性があるとみなすことが出来る。

5．ポインタ解析

しかし，4 の方法だけではポインタを用いたプログラムに対するデータ依存解析を行えない。Figure 5.1 はFigure 4.1 のXの部分をポインタ変数が指しているメモリロケーションへのアクセスに書き換えたものである。

この場合，ポインタ変数PとQが同じロケーションを指しているならばそれぞれのタスク間にデータ依存関係が成り立つ。しかし，ポインタPとQが同じロケーションを指していなければそれぞれのタスク間にデータ依存関係は発生しない。

このようにポインタは様々なロケーションにアクセスする可能性があり，プログラムの実行時まではポインタがどのロケーションにアクセスするかが不明の場合がある。そのため，単にコードに記述されている変数を静的に読んでデータ依存関係を調べることはできない[2]。以上の理由から，ポインタ解析を行わない従来のデータ依存解析では，実行結果を正しく得るためにポインタ変数にアクセスするステートメントは必ず逐次処理を行うものと判定する。

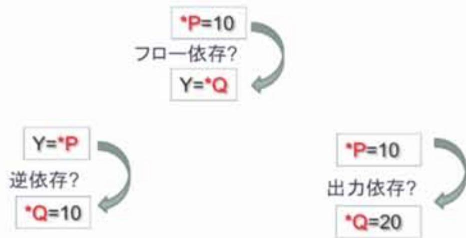

Figure 5.1　ポインタがあった場合

この問題に対し，points-to解析を行い，ポインタが指しているロケーションを明らかにすることにより，新たな並列性の解析が可能になる。

5. 1　従来のポインタ解析

従来のポインタ解析ではSSA(静的単一代入)形式を使ってポインタ解析を行っていた[4]。この方式ではポインタが代入される度に新しいポインタを定義しそれぞれにどの変数を指すのかと言う情報を付与して解析する。Figure 5.2 は実際にSSA形式に書き換えた例である。

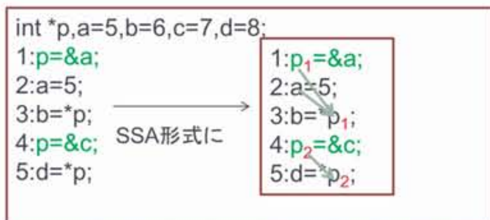

Figure 5.2　SSA形式への書き換えとデータ依存解析

こうしてSSA形式に書き換えたそれぞれのポインタ変数にそれぞれの指し先の変数を保存する。Figure 5.2 の例の場合，p1 は変数a，p2 は変数cを指しているという情報を保存する。1 行目でp1 がwriteされ 3 行目で使われているためにこれらのデータ依存関係を認識する。同様に 4 行目と 5 行目のデータ依存関係を認識する。そしてp1 はaを指しているので 2 行目と 3 行目のデータ依存関係を認識する。書き換え後，図中右側の矢印で示されるようなデータ依存関係が認識されることになる。

しかし，従来のC言語自動並列化トランスレータではポインタ変数の指し先として変数のみを扱っていた。これだけではポインタ解析を行って並列性を見いだせるプログラムが動的なメモリ確保を行わないものに限られてしまう。

例えばFigure 5.3 のような他のオブジェクトを指すポインタを持つ再帰的な構造体があり，この構造体を使ったFigure 5.4 のようなプログラムを実行した場合，リスト構造が構成される。従来のC言語自動並列化トランスレータではFigure 5.4 に対するデータ構造の依存関係の解析が不可能だった。

```
struct list {
        int i;
        struct list *nxt;
};
```

Figure 5.3　リスト構造に使う構造体

```
1:p=(struct   list*)malloc(sizeof(struct list));
2:x=p;
3:for(i=0;i<N;i++)
{
        4:temp=(struct   list*)malloc(sizeof(struct list));
        5:x->nxt=temp;
        6:x=x->nxt;
}
7:x->nxt=0;
```

Figure 5.4　リスト構造の作成

5. 2　Shape解析

Figure 5.5 のようなプログラムでポインタがアクセスしているロケーションを解析するための解析手法としてShape解析が挙げられる[4]。Shape解析とはプログラムを実行せずに動的に割り当てられたデータの特性を解析する技術である。Figure 5.5 のようなリスト構造や二分木等の動的メモリ確保により作成される再帰的なデータ構造の形状を解析する。これによりポインタがどのロケーションを指しているのかを明らかにし，新しい並列性の検出が可能になる。

Figure 5.5　再帰的なデータ構造

5. 2. 1　Shapeグラフとノード

再帰的なデータ構造を表現するためにFigure 5.6 のようなShapeグラフを作成する。Shapeグラフ内の円は動的に確保した領域を示すノードであり，矢印はポインタの指し先，つまりロケーションのリンクである。

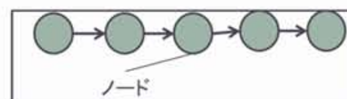

Figure 5.6　Shapeグラフ

5. 2. 1. 1　要約(summarize)と実体化(materialize)

再帰的なデータ構造は実行時までどのような規模になるかは不明である。このような問題に対して要約

(summarize)や実体化(materialize)を行うことであらゆる規模の再帰的なデータ構造に対して対応できるようにする。Figure 5.7 は要約の一例である。この図の上部に描かれているShapeグラフの四角で囲まれた部分はどのポインタにも指されていない。よってこれらのノードに対して要約を行うことで一つにまとめたノードで表現した。

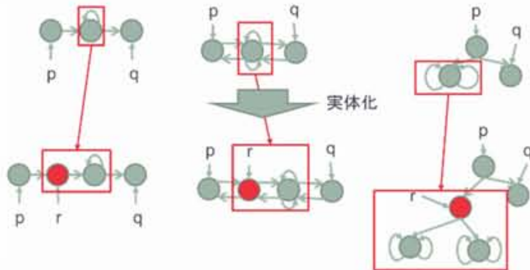

Figure 5.7　要約の様子

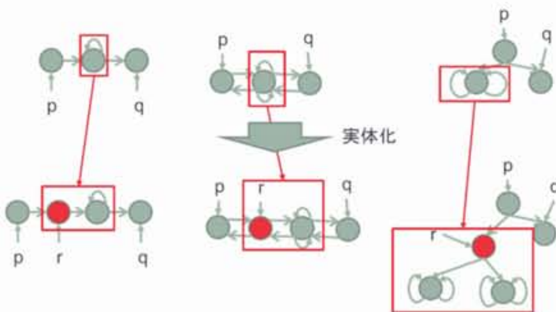

Figure 5.8　実体化の様子

逆にShape解析で要約されたノードを使用する際は実体化を行う。Figure 5.8 は実体化の一例である。この時r=p->nxtというようなpが指しているノードのメンバを代入するような式によってポインタを更新した場合，要約されたノードを指そうとする。しかし，要約されたノードはFigure 5.7 から分かるように実際には複数のノードである。そのため，このように要約されたノードを指そうとするようなタスクを解析する場合は実体化を行い，要約されたノードから一つのノードを取り出す。

5. 3　Shape解析の実装

5. 3. 1　ノードの実装

従来のＣ言語自動並列化トランスレータではポインタの指し先として変数しか存在していなかった。そのためShape解析でノードを定義し動的に確保した領域を表現するために抽象化したロケーションを定義して，ロケーションを継承する形でノードを定義した。

5. 3. 2　リンク情報

points-to解析でポインタとロケーションの関係を参照するためにリンク情報を保存する。このリンク情報には，どのポインタに指されているか，ポインタの指し先，ノードの場合は要約を行っているのかの情報が含まれている。従来の研究ではSSA形式で新しい変数を定義し，それぞれの変数にこのリンク情報に相当する情報を保存していたが今回はロケーションをキー，リンク情報を値としたmapに保存する。これでShape解析後のpoints-to解析で同じロケーションをキーとしてステートメントごとに異なるリンク情報にアクセスすることが可能になる。

5. 3. 3　ステートメントの追跡とShapeグラフの保存

このようなロケーションをキー，リンク情報を値としたmapをShapeグラフとして扱い，ステートメントごとに保存してpoints-to解析で使用する。中間データ構造からステートメントの追跡を行い，ポインタに関連するステートメントならShapeグラフの更新を行う。

5. 3. 4　ループでのShapeグラフの更新

ループ文に対してステートメントの追跡とShapeグラフの更新を行う際は，イテレーションの最後で要約を行った後に終了条件を満たすか，前イテレーションの形状と比較しShapeグラフの変化を見込めなくなるまでイテレーションの頭に戻り追跡を続行する。このようにShapeグラフの変化を見込めなくなることを安定化と呼ぶことにした。Figure 5.4 のプログラムに対するステートメントの追跡とShapeグラフの更新を例にして説明する。

5. 3. 4. 1　要約の様子

Figure 5.9 はFigure 5.4 の 1 周目のShapeグラフの変化である。このように 4 行目でノードの作成，5 行目でノードのリンク，6 行目でポインタの更新を行ってShapeグラフの更新を行っていく。

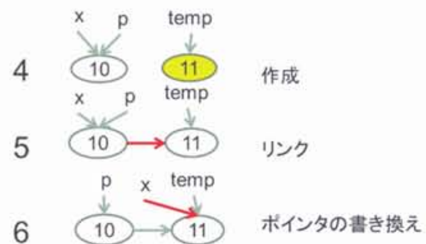

Figure 5.9　Figure 5.4 の 1 周目でのShapeグラフの変化

このループのステートメントの追跡とShapeグラフの更新を行うことで 3 周目と 4 周目のステートメントの最後でFigure 5.10 のようなShapeグラフが得られる。

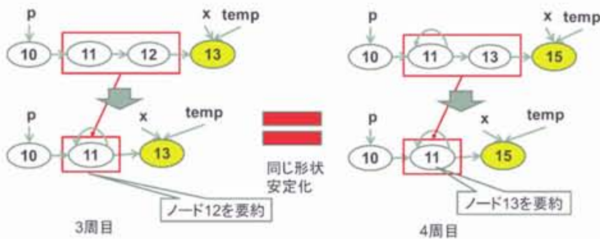

Figure 5.10　Figure 5.4 の 3 周目と 4 周目のShapeグラフ

3 周目の最後のステートメントのShapeグラフではノード 11 とノード 12 がどのポインタ変数にも指されておらず連続している。よってこれらのノードは要約を行い一つのノードで表現する。要約を行う際に，どのノードを要約したのかを保存しpoints-to解析で同じロケーションである可能性のあるノードを解析する。4 周目も同様にノード 11 とノード 13 を要約する。

3 周目と 4 周目で同じ形状であることを解析するのでこのループでのステートメントの追跡を終了する。またFigure 5.10 はあくまでもループの 3 周目以降のShapeグラフである。しかし，実行時にループが 3 周以上するとは限らず 0~2 周目の場合もある。そのため，Figure 5.11 のような 0~2 周目で得られたShapeグラフに対しても以降のステートメントの追跡でShape解析を行う。

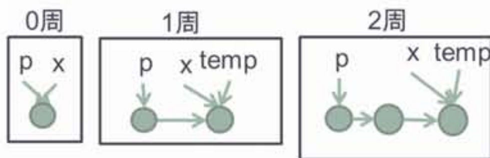

Figure 5.11　0～2 周目のShapeグラフ

5. 3. 4. 2　実体化の様子

Figure 5.4 のプログラムの続きであるFigure 5.12 のプログラムに対するステートメントの追跡とShapeグラフの更新を例にして安定化するまでの実体化の様子を説明する。

```
max=0;
for(x=p;x!=0;x=x->nxt){
        if(max<x->i)
                max=x->i;
}
```

Figure 5.12　リスト構造から最大値を求めるプログラム

Figure 5.13 はFigure 5.12 で 1 周目終了時の再代入式に対するステートメントの追跡を行った際のShapeグラフの変化である。再代入式はx=x->nxtであるので，xを上書きし，元々xが指していたノード 10 のメンバnxtで指しているノードであるノード 11 を指すようにする。しかし，5.2.1.1 でも記述したように要約されたノードは複数のノードをまとめたものであるので，要約されたノードから一つのノードを取り出す。つまり 2 個以上であるノード 11 に対して実体化を行い一つのノード 12 を取り出す。そしてポインタ変数xはこのノード 12 を指すようにする。

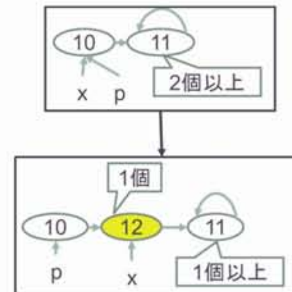

Figure 5.13　Figure 5.12 での 1 周目の再代入式での Shapeグラフの変化

こうして一度実体化してノードを一つ取り出したノードは 2 個以上をまとめている状態から 1 個以上をまとめているノードとなる。以降，ノード 11 の実体化を行う際はノード 11 が要約されていない場合と要約されている場合，つまりノード 11 が 1 個である場合と 2 個以上の状態と場合分けを行ってShapeグラフの更新を行っていくことになる。その様子をFigure 5.14 で示す。

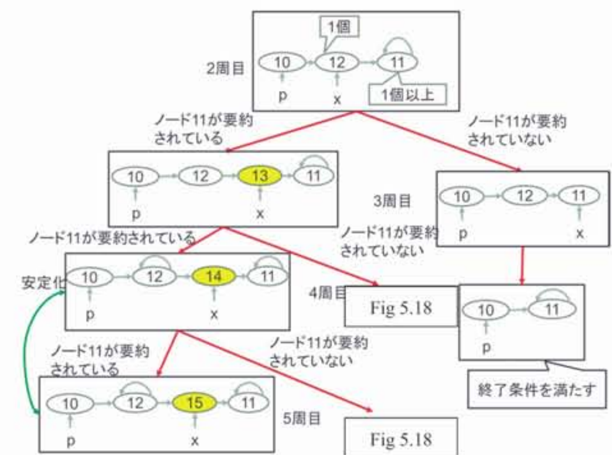

Figure 5.14　Figure 5.12 での 2 周目以降のShapeグラフの変化

ノード 11 を 3 周目で要約されていないものとして解析する。つまり，リスト構造の要素が 3 つだった場合はxが全ての要素を確認した後に終了条件を満たしてループでのステートメントの追跡を終了する。ノード 11 が要約されているものとしてShapeグラフの更新を続けていった場合，5 周目で 4 周目と同じ形状のShapeグラフとなり安定化したと解析しこのループでのステートメントの

追跡を終了する。

またFigure 5.15は5周目での要約前のShapeグラフである。ノード14及びノード15は両方共ノード11から実体化したものである。実体化は複数にまとめられたノードから一つのノードを取り出すため違うIDのノードでも同じロケーションである可能性がある。そのためFigure 5.15のように違うロケーションであることが明らかである場合はpoints-to解析でポインタがそれぞれ違うロケーションを指していることを示すためにノード14とノード15が違うロケーションであると言う情報を付与しておく。

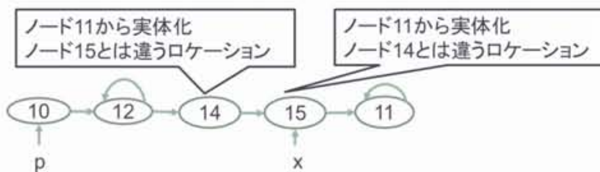

Figure 5.15　Figure 5.12 の追跡 5 周目での要約前の Shapeグラフ

また4周目及び5周目でノード11を単体として実体化した場合，Figure 5.16のようにShapeグラフが変化し，リスト構造の全ての要素を確認した上で終了条件を満たす。

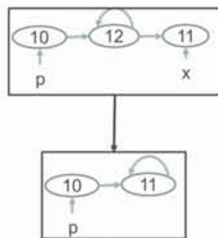

Figure 5.16　3 周目以降でノード 11 が要約されていない場合

作成する部分が実行時に0~1周でのリスト構造，つまりリスト構造の要素数が1~2個だった場合のShapeグラフに対してステートメントの追跡を行った結果はFigure 5.17となる。つまり，いずれの場合もリスト構造の全ての要素を確認することが分かる。

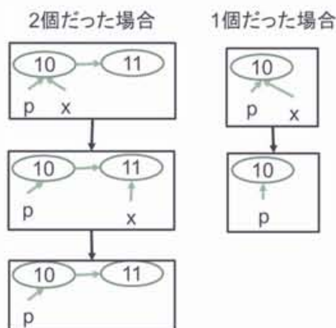

Figure 5.17　リスト構造が 1 個か 2 個だった場合

6．並列性の解析

以上のようにShape解析を行い中間データ構造にそれぞれのタスクでのShapeグラフを明らかにし保存する。これによりpoints-to解析でもこの情報を参照できるようになる。つまり，Shapeグラフの情報を参照しポインタがどのロケーションを指しているのかの情報が参照可能になり，この情報と4で述べたようなデータ依存解析と組み合わせることによって再帰的なデータ構造に対してもデータ依存解析が可能になる。

Figure 5.12の4周目及び5周目でのShapeグラフから二つのイテレーションの最後のShapeグラフは同じ形状ではあるが4周目でxが指しているノードのIDは14であり，5周目で指しているノードはノード14であることが分かる。これらはFigure 5.15で解析した情報から異なるロケーションを指していることが分かる。つまり，3周目以降，データ依存解析時にxが指しているロケーションがイテレーションごとに異なる事を解析できた。

これはスカラエクスパンションの特殊ケースである。つまりFigure 6.1のようにリスト構造をプロセッサごとに区切って並列処理でそれぞれのリスト構造で最大値を求めた後にそれぞれのプロセッサでの最大値の中からさらに最大値を求めることで逐次実行と同じ結果を出すことが可能である。

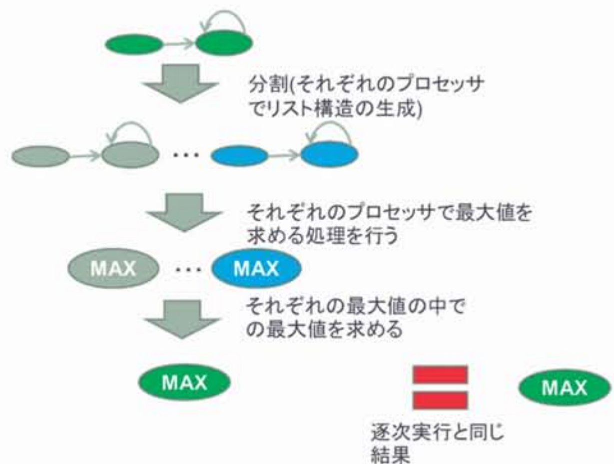

Figure 6.1　Figure 5.12 の並列化案

Figure 5.12を並列化したものがFigure 6.2である。それぞれのプロセスで元のリスト構造の要素数をプロセス数で割った数だけのリスト構造を作成しデータを入力していたものとする。それからそれぞれのプロセスで最大値を求めた後にMPI_Reduce命令で全てのプロセスの最大値の中での最大値を求めることが出来る。

```
//元のコードと同じでそれぞれのプロセスで最大値を求める
for(x=p;x!=0;x=x->nxt){
        if(max<x->i)
                max=x->i;
}
//MASTERに計算結果を集める
MPI_Reduce(&max,&recv,1,MPI_INT,MPI_MAX,MASTER,MPI_COMM_WORLD);
```

Figure 6.2　Figure 5.12 の並列化

7．hmmer（エイトマー）に対するShape解析

本研究の評価として使った逐次プログラムはhmmerである。hmmerはSPEC CPU2006[6]のベンチマーク一つであり遺伝子配列の検索を行う。Figure 7.1 はhmmerの一部を書き換えたものでありリストold_tableを逆順にしたリスト，new_tableが得られる。

```
new_table=malloc(sizeof(遺伝子のリスト));
optr=old_table;
while (optr != 0) {
        nptr = new_table;
        new_table = optr;
        optr = optr->nxt;
        new_table->nxt = nptr;
}
```

Figure 7.1　hmmerの一部

これに対してShape解析を行ったところ，イテレーション内部の 3 行目のタスクでのShapeグラフの更新を行う際に複数のノードとして実体化を続けた場合，Figure 7.2 のように 5 周目と 6 周目のイテレーション終了時に同じ形状となり安定化した。

また，Figure 7.3 のように一回でもIDが 14 のノードを一個のノードとして実体化すればいずれはoptrが 14 に移動し次に 0 を指しステートメントの追跡は終了する。

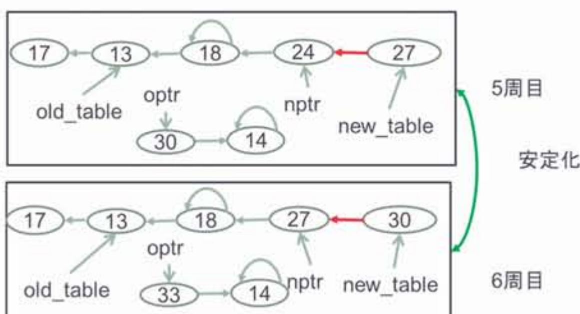

Figure 7.2　Figure 7.1 に対するShape解析の一部

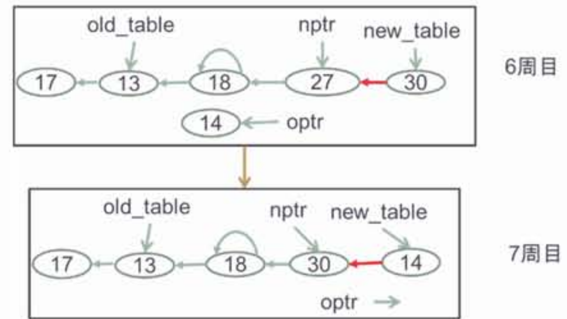

Figure 7.3　6 周目でノード 14 が要約されていないものとして実体化した場合

つまり，Figure 7.1 のプログラム実行時にリスト構造がいくつであったとしてもリスト構造を逆順にすることを解析できた事になる。

このようにベンチマークに対してもShapeグラフを構築しポインタがどのロケーションを指すのかが明らかになりデータ依存解析で使うことが可能になった。

8．終わりに

今回の研究でShapeグラフを構築し再帰的なデータ構造の形状が解析可能になり，その中でポインタがどのロケーションを指しているのか，指す可能性があるのかを解析できるようになった。これによりこれまで不明だった並列性が解析可能になった。そして，リスト構造を分割しそれぞれのプロセッサで処理する例を示した。さらに実際にベンチマークに対して解析したところ，Shape解析をすることが可能であることを確認した。

このようにイテレーションやタスクごとのShapeグラフの変化が明らかになりデータ依存関係が分かるようになったことで人手による並列化は可能になった。今後の課題としては並列コードの自動生成に対応させることである。またインタプロシージャに対応させることでより多くのプログラムに対応させる事ができるようにすることが考えられる。

9．参考文献

[1] Kazuhiro Minomoto :"Implementations of Automatic Translator from C program to Parallel Programs using MPI", 修士論文, 成蹊大学工学部工学研究科情報処理専攻, 2005.

[2] Sumiya Tohyama : "Implementations of Parallelism Analysis and Dynamic Execution Controller for Automatically Parallelizing Sequential C Programs", 修

士論文, 成蹊大学工学部工学研究科情報処理専攻, 2013.

[3] Mase Masayoshi : "Studies on Automatic Parallelization and Low Power Optimization of C Programs on Multicore Processors",早稲田大学大学院基幹理工学理工学専攻博士論文, 2011.

[4] 武市　和真: "C 言語自動並列化トランスレータの開発　ー静的単一代入形式を用いたポインタの依存解析の実装 ー" 卒業論文, 成蹊大学理工学部情報科学科, 2011

[5] Adrian Tineo, et.al : "A New Strategy for Shape Analysis Based on Coexistent Link Sets.", Proceedings of the International Conference ParCo 2005, pp.13-16, 2005

[6] FUJITSU : "WHITE PAPER ベンチマークの概要".2006 http://jp.fujitsu.com/platform/server/primergy/performance/pdf/benchmark-overview-speccpu2006-jp.pdf