B-013

C 言語自動並列化トランスレータの開発
－構文木による並列構造解析へのループリストラクチャリング適用と OpenMP の活用－
Development of Automatic Translator from C Programs to Parallel Programs Using MPI
- Applying the Loop Restructuring to Parallel Structure Analysis Based on Syntax Tree and Exploiting OpenMP for the Parallel Loops -

小倉 健太郎†　**阿加井 星**†　**甲斐 宗徳**†
Kentaro Ogura　Sei Akai　Munenori Kai

１．はじめに

近年のプロセッサのマルチコア化により並列コンピューティング環境が身近になっているが、並列プログラミングには逐次プログラミングに加えて専用の知識が必要となり、ソフトウェア開発者の負担となっている。この問題の解決策の１つとして筆者らは、C 言語で記述された逐次プログラムを自動で MPI（Message Passing Interface）を利用した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開発を進めている。

筆者らが開発中の C 言語自動並列化トランスレータは、読み込んだ逐次プログラムから構文解析により構文木構造を作成し、それに対して並列性解析やタスク粒度解析の結果を反映するよう、構文木のままで必要な変換をしていく。必要な解析がすべて終了したら、構文木構造から MPI を用いた並列コードを生成することができる。

一般に言われているプログラムにおける実行時間のほとんどを占めるのはループ処理であるということから、自動並列化で処理時間を短縮するために、本稿ではループに焦点を絞っている。外見上並列性が無いように見えるループでもリストラクチャリングにより並列性を引き出すことができるループも存在することから、構文木構造を用いてイテレーション内外の並列性を解析し、その結果に対してループリストラクチャリング手法を適用していく。また、検出された DOALL 可能なループに対しては、マルチコアプロセッサで高速化可能な OpenMP による並列化を施すことも試みたので報告する。

2．ループの並列性

ループを並列化して処理時間を短縮するためには、イテレーション内外のデータ依存関係が無い状態にする必要がある。イテレーション内のデータ依存関係を解消することにより、ループ一回の中での並列処理が可能となる。また、イテレーション間のデータ依存関係を解消することで、ループの回数を分割することによる並列処理が可能となる。連続するイテレーションが部分的にオーバラップして実行されるループは DOACROSS ループと呼ばれ、すべてのイテレーションが完全にオーバラップして実行できるループは DOALL ループと呼ばれている。並列処理効果の最も高いループは DOALL ループである。連続するイテレーション間のデータ依存関係がそれぞれのイテレーション内のどの位置のステートメントにあるかで、オーバラップして実行できる範囲が定まる。すなわちオーバラップが 0%であればそれは逐次処理しかできないループであることを意味し、100%オーバラップできるのであればそのループは DOALL ループであることを意味する。

†成蹊大学理工学研究科理工学専攻　Graduate School of Science and Technology, Seikei University

3．ループリストラクチャリングの概要

前章で述べたように、イテレーション間にデータ依存関係がある場合、DOALL 処理ができないので並列処理は不可能である。しかし、中にはループを再構築することでイテレーション間のデータ依存関係を無くし、並列性を引き出すことができるようなものも存在する。ループリストラクチャリングには様々な手法が提案されているが、その主な手法として

・スカラエクスパンション
・ループディストリビューション
・ステートメント並べ替え
・ノードスプリッティング

などが挙げられる。本稿では、これらの変換手法を for ループのイテレーション内及びイテレーション間のデータ依存関係に基づき、行っていく。

3.1. スカラエクスパンション

スカラ変数が計算の中間結果を一時的に保存する変数として使用されている場合に適用される手法である。このようなスカラ変数はイテレーション間にデータ依存関係を生み出し、並列化を妨げる原因となる。そこで、そのスカラ変数をベクトル化した変数に置き換えることでイテレーション間依存を解消し、並列性を抽出する。以下にスカラエクスパンション適用可能な例とその適用後のコードを示す。

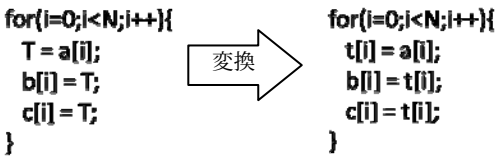

図 1　スカラエクスパンション

3.2. ステートメント並べ替え

前方へのステートメント間依存が発生している場合に後方の依存へと変えることが可能になる手法である。例えば図 2 のような for 文を考える。

```
for ( i=0 ; i<N ; i++ ){
  a[i] =  b[i] + c[i] ;          //S1
  d[i] = a[i+1] + b[i] ;         //S2
}
```

図 2　ステートメント並べ替え適用可能例

この例において、ステートメント S2 の a[i+1]は次のイテレーションで実行されるステートメント S1 の a[i]に逆依存のデータ依存関係を持っている。すなわちステートメント S1 の a[i]とステートメント S2 の a[i+1]の間に同じ配列に関して前方への依存関係が発生していることになる。この二つのステートメントの間には配列インデックスが異なるので同一のイテレーション内では依存は発生しておらず、二つのステートメントを並べ替えることは可能である。この並べ替えは以下のように行われる。図 2 の for 文は構文解析を用いた中間データ構造作成により、図 3 に示すような構文木構造の解析結果で保存されている。

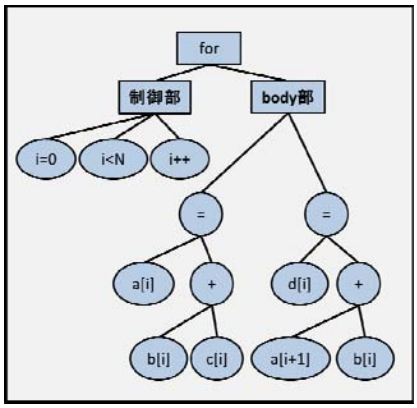

図 3　ステートメント並べ替え適用前

構文木の中で S1 と S2 のステートメントに対応する部分木を検出したら、この 2 つのステートメントの入れ替えを行い、図 4 のように変換する。2 つのステートメントが元の配置のままでは、連続する 2 つのイテレーションをオーバラップして実行することはできないが、入れ替えを行うことにより、オーバラップして実行できるイテレーションの範囲が大きくなる。また元のコードでは同じ配列に関してステートメント S2 からステートメント S1 への前方の依存関係があったのに対し、変換後はステートメント間では後方への依存関係に変更されたことになる。

```
for ( i=0 ; i<N ; i++ ){
  d[i] = a[i+1] + b[i] ;         //S2
  a[i] =  b[i] + c[i] ;          //S1
}
```

図 4　ステートメント並べ替え適用後コード

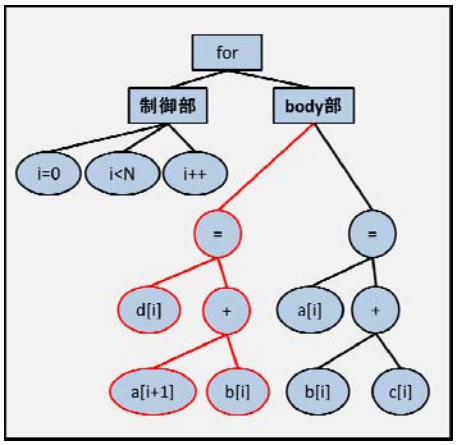

図 5　ステートメント並べ替え適用後

4.3. ノードスプリッティング

ステートメント並べ替えと同様に前方へのステートメント間依存を後方への依存に変える手法であるが、その対象ステートメント間にイテレーション内依存が発生している場合に適用可能な手法である。例えば図 6 のような for 文を考える。

```
for ( i=0 ; i<N ; i++ ){
  a[i] =  b[i] + c[i] ;          //S1
  b[i] = a[i+1] ;                //S2
}
```

図 6　ノードスプリッティング適用前コード

この例において、ステートメント S2 の a[i+1]がステートメント S1 の a[i]に前節の例と同様な依存関係を持っている。また、解析を進めていくとステートメント S1 とステートメント S2 の b[i]との間に逆依存のデータ依存関係を持っている。すなわちステートメント S1 とステートメント S2 の b[i]との間にイテレーション内依存が発生しており、このままステートメントの並べ替えを行ってしまうとイテレーション内の依存関係が崩れてしまい、正しい答えを得ることができなくなってしまう。そこで、並べ替えではない依存関係の解消を行っていく。図 6 の for 文は図 7 に示すような構文木構造で保存されている。

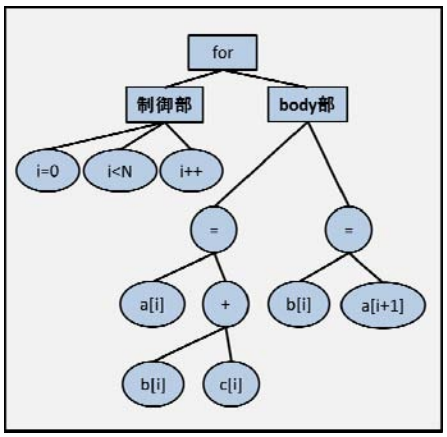

図 7　ノードスプリッティング適用前

構文木から S1 と S2 のステートメントに対応する部分木を検出したら、この例における a[i+1]のような次のイテレーションの変数に依存関係を持つような変数を別に新しく宣言した変数に読み込ませ、新しい変数を用いて処理を行わせるようにする。こうすることで、イテレーション内部の依存関係を崩すことなく各イテレーションをオーバラップして並列処理することが可能な状態となる。

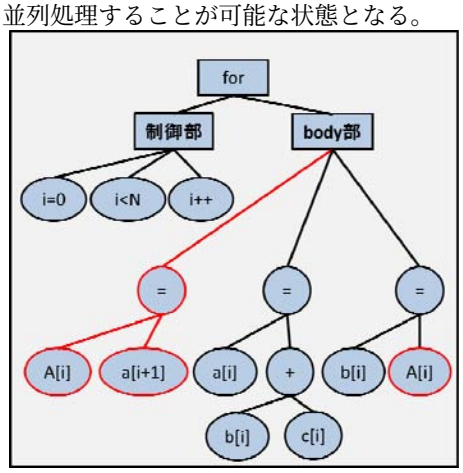

図 8　ノードスプリッティング適用後

4.4. ループディストリビューション

ステートメント間に前方へのステートメント間依存が無く、後方へのステートメント間依存が存在する場合にループを複数に分割する手法である。図 9 のようなコードを考える。

```
for ( i=0 ; i<N ; i++ ){
  a[i+1] =  b[i] + c[i] ;
  d[i] = a[i] + b[i] ;
}
```

図 9　ループディストリビューション適用前

この例において、ステートメント S1 の a[i+1]は次のイテレーションで実行されるステートメント S2 の a[i]に後方へのフロー依存が発生している。つまりステートメント S1 の a[i+1]とステートメント S2 の a[i]との間に後方へのステートメント間依存が発生していると言える。この二つのステートメントの間には前方へのステートメント間依存が無いため、ステートメント S1 とステートメント S2 の間でループを分割することで、ステートメント間依存を解消することが可能である。この分割の流れは次のように行われる。まず、図 9 の for 文は図 10 に示すような構文木で保存されている。

構文木の中から S1 と S2 のステートメントに対応する部分木を検出したら、それぞれのステートメントから 2 つのループを生成することで、図 11 のようにループを分割する。これにより生成された 2 つのループはそれぞれ DOALL 可能なループとなり、並列処理が可能となる。

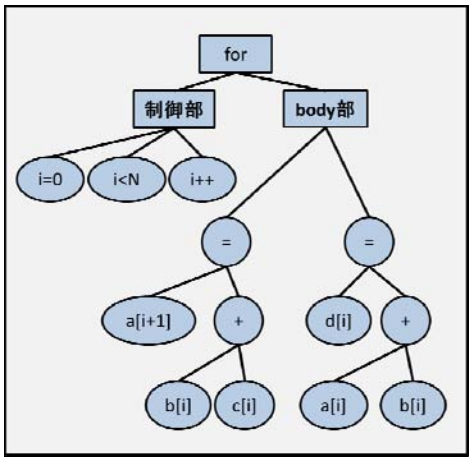

図 10　ループディストリビューション適用前

```
for ( i=0 ; i<N ; i++ ){
  a[i+1] =  b[i] + c[i] ;
}
for ( i=0 ; i<N ; i++ ){
  d[i] = a[i] + b[i] ;
}
```

図 11　ループディストリビューション適用後コード

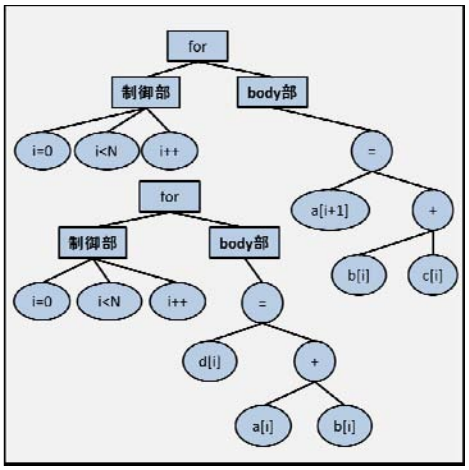

図 12　ループディストリビューション適用後

以上で述べたループリストラクチャリング手法を適用していくことで、ループの並列性を引き出していく。

5．評価

本稿で取り上げたループリストラクチャリングの各手法を簡単なコードに適用し、並列化可能なループに変換されるかどうかを確認した。そのサンプルコードを図 13 に示す。図 13 の for 文を解析することにより、イテレーション内外に下記の依存関係を発見することができる。

まずイテレーション内の依存関係は、以下の 3 つである。

S1 と S2 の間にスカラ変数 T に関してフロー依存
S2 と S4 の間にベクトル変数 c に関してフロー依存
S3 と S5 の間にベクトル変数 a に関してフロー依存

またイテレーション間の依存関係は以下の 3 つである。

S4 と S3 の間にベクトル変数 b に関してフロー依存
S2 と S1 の間にスカラ変数 T に関して逆依存
S5 と S3 の間にベクトル変数 a に関して逆依存

```
for ( i=0 ; i<N ; i++ ){
 T = 5 ;              //S1
 c[i] = T + 1 ;       //S2
 a[i] = b[i] ;        //S3
 b[i+1] = c[i] ;      //S4
 d[i] = a[i] + a[i+1] ;  //S5
}
```

図 13　サンプルコード

これらの依存関係はループの並列化を妨げるものである。したがってこのループは DOALL 処理をすることができない。このような DOALL 処理不可能なループに対してループリストラクチャリングの各手法を適用し、並列実行可能なループの構築を試みる。その結果が図 14 になる。

```
for ( i=0 ; i<N ; i++ ){          for ( i=0 ; i<N ; i++ ){
 t[i] = 5 ;                        A[i] = a[i+1] ;
 c[i] = t[i] + 1 ;                 a[i] = b[i] ;
 b[i+1] = c[i] ;                   d[i] = a[i] + A[i] ;
}                                 }
//右のコードに続く
```

図 14　ループリストラクチャリング適用後コード

各手法の適用順序は、他の手法に直接影響されないスカラエクスパンション手法を初めに適用する。その後、ループディストリビューション手法を適用する準備として必要となるノードスプリッティング手法とステートメント並べ替え手法を適用する。最後にループディストリビューション手法を適用する。このように適用することで、DOALL 可能でない for ループ文を、DOALL 可能なループに変換することができた。

2 つに分けられたループ間に依存関係はないので、我々の並列化トランスレータによって自動的に 2 つのプロセッサに各ループが割り当てられた。また、各ループは DOALL 可能なループなので、マルチコアプロセッサであれば MPI だけでなく OpenMP によるコア並列も利用することができる。そこで MPI と OpenMP によるマルチコア/マルチプロセッサ並列を利用したプログラムで並列処理時間の時間の比較を行った。

ループの反復回数を 1000 万回にし、ループリストラクチャリング適用後のコードの並列実行と適用前のコードの逐次実行をそれぞれ 100 回行ったときの平均実行時間を比較した。その結果を図 15 に示す。なお、このグラフにおける MPI+OpenMP(n コア)という記述は MPI により分割した各プロセッサにおいて n コアで並列処理を行うという意味である。

逐次処理と比べてみると、MPI での実行結果が 2 プロセッサで並列処理をしているのにあまり高速化率が上がっていない。これは、スレッド間での通信に時間を割いてしまっているのに加えて、二つに分割したループがそれぞれ等価でないからだと考える。また、OpenMP による実行結果も、並列数にくらべておよそ 5 割から 6 割ほどの高速化の実現となった。

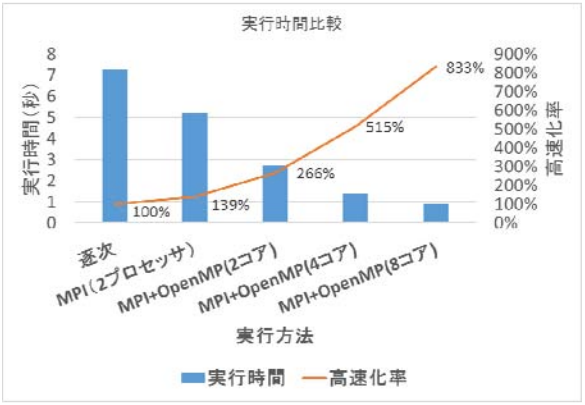

図 15　ループリストラクチャリング適用前後の平均実行時間の比較

以上の結果から、リストラクチャリングすることにより for ループ文の依存を解消し、それを並列化することにより、MPI や OpenMP において並列数と見合っているとは言えないものの、高速化を実現することができたと言える。

6．まとめ

本研究では構文木構造を利用し、構文木構造に対して４つのループリストラクチャリングの実装を行った。実装の手順としては、まずデータ依存解析を行い、データ依存関係があってもループリストラクチャリング手法の適用が可能と判定した場合にループの再構築を行った。

また、ループリストラクチャリングを適用させたコードを MPI と OpenMP を併用して並列処理し、ある程度の高速化を実現させることができた。マルチコアプロセッサが身近になったことから、OpenMP の利用を積極的に行うことが良いと考えられるが、筆者らの自動並列化のアプローチでは、抽出された並列性は構文木構造に組み込まれているので、MPI および OpenMP を利用した並列コード生成が可能になることを示すことができた。

参考文献

[1]美濃本　一浩：“C 言語自動並列化トランスレータの開発”,修士論文,成蹊大学工学部工学研究科情報処理専攻,2005.
[2] 関口俊樹：“C 言語自動並列化トランスレータの開発―ループリストラクチャリングによる並列性の抽出の実装―”卒業論文,成蹊大学ソフトウェア工学研究室,2011.
[3] 山崎 瞳：“C 言語自動並列化トランスレータの開発―構文木構造を用いたループリストラクチャリングの実装―”,卒業論文,成蹊大学ソフトウェア工学研究室,2012.