成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.53 No.1 (2016)pp.23-28

C言語自動並列化トランスレータにおける構文木ベース中間データ構造への並列化情報統合手法の改良

小倉 健太郎[*1]，甲斐 宗徳[*2]

Improvements on Integrating Parallelized Information into Intermediate Data Structure based on Parse Tree for Automatic Parallelizing Translator for C Programs

Kentaro OGURA[*1], Munenori KAI[*2]

ABSTRACT：In our automatic parallelizing translator, an intermediate data structure based on parse tree of the sequential C program is generated. There are some analysis stages in our parallelizing translator, such as data dependency analysis stage, task granularity analysis stage, task scheduling stage, and parallel code generation stage. In this paper, we propose a new integration method of analysis result from each stage into the intermediate data structure based on parse tree for the purpose of translation efficiency. In addition, we report the implementation of the history mechanism of the translate-operations to the intermediate data structure based on parse tree in order to trace backward or forward of the sequence of the operations.

Keywords：Parallelizing translator, Parsing, Code restructuring, Parallelism analysis

(Received June 30, 2016)

1．はじめに

近年ではコンピュータ性能の向上が求められている。しかし、物理的な問題や消費電力の問題からコンピュータ単体での高速化には限界が見えてきている。

そこで、コンピュータ性能の向上の解決策として挙げることができるのが、並列コンピューティングである。これは並列プログラミングを用いて並列処理用のプログラムを作成し、その処理をマルチコアやマルチプロセッサに割り当てることでプログラムを効率よく並列処理させるというものである。

しかし、この並列プログラミングというものはとても難しいものとして知られている。並列化可能か調べる作業が必要となったり、通信を考慮する必要があったり、並列プログラミングの専門的な知識を習得することも必要となるためである。その難しさが並列コンピューティングの普及を妨げる理由となっている。

この問題の解決策として、C言語で記述された逐次実行可能なプログラムを自動的にMPIを用いた並列実行可能なプログラムに変換するトランスレータを利用することが挙げられる。これを用いることにより、自動で並列プログラムを生成することが可能となるため、手軽に並列プログラムを利用することが出来る。本研究では、C言語によって記述された逐次コードを自動的にMPI（Message Passing Interface）コードに変換するC言語自動並列化トランスレータの完成を目指す。

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は、C言語で記述された逐次プログラムをトランスレータに読み込ませることでMPIを用いた並列プログラムに変換されるというものとなっている。その流れを大まかに図にしたものを次に示す。

本トランスレータでは、まず読み込んだC言語のプログラムを中間データ構造として保存する。この中間データ構造は構文木として保存される。保存された中間データ構造に対して並列依存解析を行い、並列性を探し出す。

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

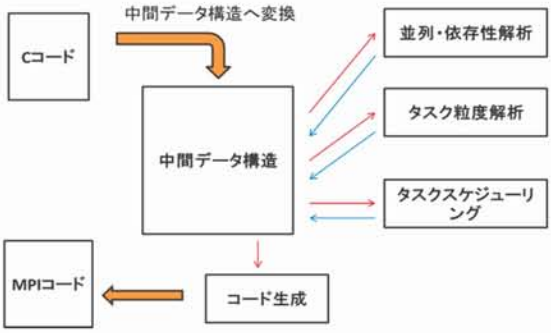

図１　並列化トランスレータ処理手順

タスク粒度解析では、タスクの実行時間や依存関係から適切な粒度を求める。タスクスケジューリングにおいては簡易的なスケジューリングを行い、タスクに対してプロセッサの割り当てを静的に行っている。

最後に、各解析器の処理が完了した中間データ構造から並列プログラムを作成し、出力する。

３．中間データ構造

中間データ構造はトランスレータ内の各解析器で行われる処理の間で受け渡されるデータ構造である。ここには逐次実行可能なC言語プログラムのソースコードを構文木構造に変換し、完全に保存する。この中間データ構造にはほかにも各解析処理の結果や、各解析器がソースプログラムに加えた変更などが保存されている状態が理想と言える。

しかし、現状のトランスレータにおいては、中間データ構造から解析した情報を、中間データ構造とは別に保存してしまっている。その結果、各解析器が中間データ構造を用いて解析をする際に、中間データ構造と解析した情報を照らし合わせながらの解析をする必要があり、とても不便な状態となっている。

そこで、本研究では理想とされてきた中間データ構造の使い方を現実とするために、各解析部で解析した結果を自由に使用することのできるような、並列情報を保存するフォーマットを作成することにする。

４．並列化情報統合手法

現状のトランスレータにおいて、各タスクのノード単位の依存情報を取得しようとする場合に、依存のあるノードまで潜っていかないと情報を取得することのできない部分が存在している。そこで、タスクグラフにおいてその中のノード単位まで深く見ることなく大きな部分から様々な解析結果を得られるようにする。そのために、新しくタスクに必要な情報をまとめるフォーマットを作成、適用し解析の精度を上げようというものである。

４．１　過去の並列化情報保存手法

これまでのトランスレータでは、変数やタスクに関する並列・依存情報はリスト構造で保存されており、必要なデータを取得する際に必要なデータまでデータリストの頭から順に見ていく必要があった。これでは解析の効率が悪くなってしまうため、このような状態を解消するためにも、どのようなデータに対しても直接指定して必要なデータを取得できるようなデータ構造を作成する必要がある。

４．２　タスク操作における並列化情報の役割

今回考える並列化情報統合手法の主な役割としては、必要な情報を簡単に取得できるようにする必要がある。過去のトランスレータに見られたような必要な情報を得るためにリストを走査する必要なく、各タスクが必要な情報を直接取得できるようにする。

４．３　並列化情報統合手法

並列・依存情報を使いやすい状態で保存しておくことは、中間データ構造を用いて並列解析を行う上で重要なことである。そこで、今年度の研究として並列化に必要な情報をまとめる新しいフォーマットを作成し、それを用いて解析やスケジューリングを行うことで、より正確な解析が行えるようにするものである。

４．３．１　変数の依存情報保存手法

プログラムの並列化を行う際には、実行する処理がどの変数に依存しているかを正しく判断する必要がある。そのために、各変数が持つ依存について各タスクが自分で情報を所持している状態にし、他に用意したデータリストにアクセスしたりすることなく依存情報を使用できるような構造を作る。

４．３．１．１　変数の依存情報

プログラム内で使用している変数には、様々な状態が存在している。変数に含まれる値を変更することなく値を読み込んで処理に使用するread、変数に代入された値に新たな値を代入するwrite、読み込みの後に続けて書き込みを行うread-write、関数の実行を行うためのexecである。

4．3．1．2　変数間の依存解析

変数間には並列処理を行う際の制限になるものがある。それがデータ依存と制御依存である。

データ依存とは、プログラム中の命令文2つのうち、どちらかの処理を実行しないともう一つの処理ができない状態のことである。

制御依存とは、if文において条件文の処理が完了しないとブロック内の処理を実行するかどうかが決められないように、データ依存が無くとも処理を評価したうえで実行する必要がある状態のことである。

依存情報を一つにまとめて保存する為に、各変数がどの依存を持っているかを解析する必要がある。しかし、変数にも様々な種類があるため、それぞれについて個別に解析する必要がある。ここでは各変数について説明していく。

(1) スカラ変数

スカラ変数とは、その変数の中に数値や文字列のような値を一つだけ保存しておくことのできるものである。各スカラ変数間に依存関係が存在すると並列実行を行うことができない。

(2) 配列変数

配列変数には複数の値をリスト形式で持たせることができる。また、各要素に含むことができるのはスカラ変数のみである。

過去のトランスレータでは、配列はインデックス単位ではなくシンボル単位での解析しか行うことができなかったが、今回インデックス単位でも解析が可能になるように改良した。

(3) 構造体のメンバ変数

構造体のメンバ変数を指定したものの場合は、解析する際に指定変数を別の変数として保存してしまうとメモリ数が増大してしまう恐れがあるため、指定変数ごとに新たに情報を追加するのではなく、元となる構造体のメンバ変数から依存情報を参照することで無駄なメモリの増加を防ぐ。

(4) ポインタ変数

ポインタ変数は、変数や動的に確保された領域に対して解析を行う必要がある。ポインタとして代入される値についてはそれがポインタ変数であるという情報を変数に持たせる。

4．3．1．3　依存情報の保存手法

解析した変数についての依存情報は、中間データ構造内に保存していく。そのために新しくフォーマットを作成し、それに当てはめる形で保存していく。

依存情報の保存の手法として、まず重要となるのが各情報へのアクセスのしやすさである。今回考えるフォーマットでは、各変数に固有のIDを持たせ、そのIDに対応させるようなハッシュテーブルを用いることにした。こうすることで、欲しい情報に直接アクセスすることが可能になる。

ハッシュテーブルの値の部分には変数に関する依存情報を保存するクラスを持たせる。そのクラスに持たせる情報は以下の通りである。

変数の依存情報クラス	
private:	public:
依存先(Task)	依存先タスク取得関数
	依存先タスク変更関数
依存先(Var)	依存先変数取得関数
	依存先変数変更関数
依存情報	依存情報取得関数
	依存情報変更関数

図２　変数の依存情報保存クラス

依存先(Var)はその変数がどの変数に依存しているかを保存する。

依存先(Task)はその変数が依存している変数がどのタスクであるかを保存する。

コストは各変数が持つ重みの値を保存する。

依存情報はその変数のもつ依存について保存する。

4．3．2　タスクの並列情報保存手法

過去のトランスレータでは、タスクの解析に必要となるエッジや保有するタスクの情報について、それぞれに用意されたテーブルを調べることで取得するという方法をとっていた。今回の研究では、他に用意されたリストを参照するのではなく必要な情報を各タスクが個別に所有するように改良し、解析や操作のための情報を取得するのに手間がかからないようにした。

4．3．2．1　タスクの依存情報

並列性解析ではどのタスクが並列に実行できるかを解析する。タスクの並列実行には変数の通信が必要なため、前項で保存した変数の依存情報を用いて並列性を解析する。

タスクにおける依存情報は、内部の変数における依存情報の種類と向きによって決まる。

4．3．2．2　タスクスケジューリング結果の保存手法

こちらに関しても、各タスクにユニークなIDを持たせ、

そのIDをキーとするハッシュテーブルを用いて各タスクの情報を保存していく。保存する情報は以下の通りである。

並列情報クラス:	
依存情報map	部分タスク判別id
先行タスクリスト	割り当てプロセッサ
タスクコスト	依存強度リスト
操作履歴id	

図３　タスクの並列情報保存クラス

依存情報mapはそのタスクが持つ変数の依存情報保存クラスをまとめたハッシュテーブルである。

先行タスクリストは、そのタスクの先行タスクのリストである。

タスクコストは、そのタスクのコストを計算する関数である。

操作履歴idは後に紹介するタスクグラフ操作履歴機能において、何番目の操作に使われているかという情報である。

部分タスク判別フラグは、そのタスクが部分タスクの一部であるかを判定するためのフラグである。

割り当てプロセッサは、そのタスクがどのプロセスに割り当てられるかの情報である。

依存強度リストは､そのタスクの持つ依存強度である。

５．タスクグラフ操作履歴

トランスレータでは、解析などを行う際にタスクに対して様々な操作をする。これまでのトランスレータにおいて、その操作を行った後は、操作前の状態を保持する機能がなく、一度操作してしまうと元の状態に戻すことが難しいという状態であった。

そこで、タスクに対する操作の情報を履歴として保存し、後で元の状態に戻したいときに簡単に戻せるようにする。

5．1　操作履歴を保存する操作

ここではタスクの操作の履歴を保存する操作について説明する。

・タスクの融合

2 つのタスクを 1 つにまとめる操作。先行後続関係にあるタスクや、並列関係にあるタスク間においてタスクの融合が可能になる。

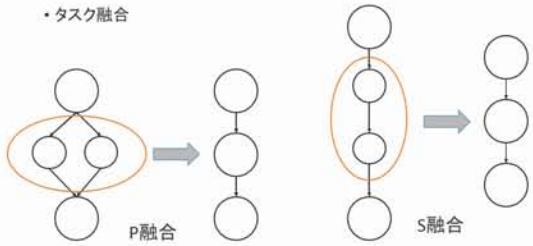

図４　タスク融合

・タスクの分割

1 つのタスクが、分割することで並列可能になると判断される場合、そのタスクを 2 つに分割することができる。

具体的な例として、ループ文を二つに分割するループディストリビューションが挙げられる。

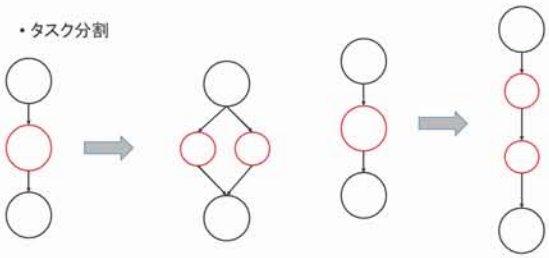

図５　タスク分割

・タスクの複製

タスクを複製することにより複製元タスクの後続エッジを分散させ、後続タスクの処理開始時間を早めることが可能になる場合がある。

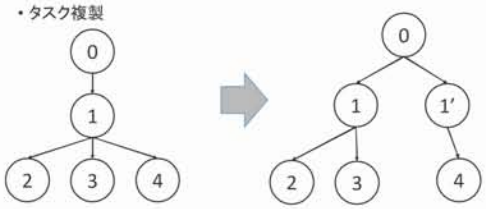

図６　タスク複製

しかし、この操作を行うことにより通信が増えてスケジュール長が伸びてしまうこともあるため、そのような場合には複製は行わないものとする。

以上のような操作が起きたとき、その操作の履歴を取ることとする。

5．2　操作履歴を保存する方法

操作履歴は、操作が行われるごとに順番に保存してい

く。今回も保存形式にハッシュマップを使用し、キーには操作ナンバー、値に操作情報を保存する。操作情報は以下の通りである。

操作情報:
操作対象タスク
構文木に対する操作
先行操作IDリスト
操作結果タスク

図７　操作履歴に残す情報

操作対象タスクには、どのタスクに対して操作を行ったかを保存する。ここでは先に述べた並列化情報統合手法で用いた情報を利用する。

構文木に対する操作は、タスクに対してどのような操作が行われたかを保持する。

先行操作リストは、その操作を行うのに必要な事前操作のIDを保存する。

操　作結果タスクには、そのタスク操作によりどのようなタスクが作られたかの情報を保存する。

5. 3　操作履歴の使用方法

操作履歴機能を使用するには、まず操作履歴の一覧を出力する。その後、やり直したい操作が存在する場合はそのIDを指定することにより、その操作から作られたタスクがその後の操作で使用されているかを検索し、存在した場合はその操作も含めて復元するべき操作を列挙する。

1: 対象, 操作名, 結果, 先行
2: 対象, 操作名, 結果, 先行
3: 対象, 操作名, 結果, 先行

process history List
1: 6 7, fusion, 10, 0
2: 4, division, 11 12, 0
3: 8 10, fusion, 13, 1

図８　操作履歴出力結果

6. 評価

6. 1　変数に関する依存情報の保存と解析

変数間の依存情報が正しく保存されているか確認していく。

今回使用するコードは次のとおりである。

このコードに対して、タスクや変数に解析の際にユニークなIDを持たせる。このようにIDを持ったコードを解析し、その結果を出力したものが次の図になる。

1: 対象, 操作名, 結果
2: 対象, 操作名, 結果

Restore List
3: 13, division, 8 10
1: 10, division, 6 7

図９　操作復元順序

```
struct str{
        int s;
};

int main(){
        int i, a, b, c, x, y, z;
        int *p;
        int arr[10];
        struct str ex;

→つづく
```

```
for(i=0; i<10; i++){
        a++;
        y++;
        a[i]=i+1;
}
c=a+x;
z=17;
p=&z;
ex.s = c;
a=b+x;
 return 0;
}
```

図 10　サンプルコード

```
/*<Val 1100>*/
1 1105 read-write
5 1112 read-write

/*<Task 1104>*/
4 1111 write

/*<Task 1105>*/
0 1100 read
5 1112 read

/*<Task 1106>*/
5 1114 read

/*<Task 1107>*/
3 1109 write

/*<Task 1109>*/
2 1107 read pointer

/*<Task 1111>*/
1 1104 read

/*<Task 1112>*/
0 1100 write
1 1105 write

/*<Task 1114>*/
1 1106 read
```

図 11　依存情報出力結果

Task_ID:4 とTask_ID:8 について見てみる。Task_ID:4 における変数a(Val_ID:1105) とTask_ID:8 における変数a(Val_ID:1112)の間には逆依存の関係があると言える。ここで保存した結果を出力した図を見てみると、二つの変数において、お互いに依存がある相手として認識できていることがわかる。このように、変数に関する依存情報は保存できていると言える。

7．結論

今回の研究内容としては、並列化情報統合手法の改良として、依存情報や並列情報を保存して自由に取り出すことのできるような新しいフォーマットを作成し、解析して得た依存情報を正しく保存したり、取り出すことが可能になった。また、タスクに対する操作の履歴を保存し、後に操作を復元する必要が出た場合にすぐに元の状態をたどれるようにした。

今後の課題として、今回作った並列化情報統合手法を

適用できているのは一部の解析器のみとなっているため、現状で効率の良い解析ができていないと考えられる部分について、今回提案した手法を適用していくことが挙げられる。それにより、今後のトランスレータにおける解析がより良いものになると考えられる。

また、操作履歴に関しては現状では復元に必要な操作のリストを出力するのみとなっており、自動で復元まで行うことのできるところまでは実装できていない。自動修復機能を実装するということが今後の課題であると言える。

参考文献

[1] 遠山　純也：“C 言語自動並列化における並列性解析と動的実行制御”, 修士論文, 成蹊大学工学部工学研究科情報処理専攻.2013.

[2] 武市　和真:“C言語自動並列化トランスレータの開発-再帰的なデータ構造を扱うプログラムのための並列性解析手法-”, 修士論文, 成蹊大学理工学研究科理工学専攻, 2013