成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.53 No.1 (2016)pp.17-22

C言語自動並列化トランスレータのための静的実行制御方式に基づく並列コード生成機構の実装とチューニングツールの試作

阿加井　星[*1]，甲斐 宗徳[*2]

Implementation of Parallel Code Generator under Static Execution Control and Proposal of Performance Tuning Tool for Automatic Parallelizing Translator for C Programs

Sei AKAI[*1], Munenori KAI[*2]

ABSTRACT：We have been studying an automatic parallelizing translator for sequential C programs with MPI, which is utilized by users without the knowledge for parallel programming languages and parallel computer architectures. In our parallelizing translator for C programs in the past, dynamic task execution control has been adopted for generated parallel programs, because it was difficult to analyze the costs of tasks, or execution time before actual execution. On the other hand, the researches on task scheduling technologies and task cost analysis has been evolved concurrently by another research groups in our laboratory and they are cooperating with the automatic parallelizing translator, so in this paper we report the implementation of generating parallel program with static task execution control and a tuning mechanism to reconfigure parallel programs by rescheduling with actual task execution time as task costs.

Keywords：Parallelizing translator, Code generator, Parallel processing, Task cost tuning

(Received June 30, 2016)

1．はじめに

近年のコンピュータの高速化については、物理的な問題点からその限界が指摘されてきている。その解決策として、マルチコアやマルチプロセッサによる並列処理を行うことで処理時間を短縮する方法がある。このような背景から、プログラムの並列化の必要性が高まっている。しかし、並列処理効果の高い並列プログラムを作成することは、逐次プログラムの開発では考慮しなかった新しい知識と手間が要求され、開発者の負担になるという問題がある。

逐次コードを並列コードに変換する際に、逐次コード内部の分化された処理群をどのように複数のプロセッサに割り当てて実行させるかという割り当て問題が生じる。その際、プログラムの実行ごとに処理コストが変化するコストや、実行前からそのコストを正確に測定することが難しい処理が存在している。そのため以前までは動的実行制御方式によって並列プログラムを動作させていたが、当研究室で開発された通信遅延を考慮したスケジューラの搭載やタスクの処理コスト算出方法の見直しなどが行われたので今年度は静的実行制御方式に基づく並列コード生成機構の実装を行った。

2．C言語自動並列化トランスレータ

当研究室で開発中のC言語自動並列化トランスレータ[1]は、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、プログラム内に存在する並列性を自動抽出し、MPIによる並列実行用コードを埋め込むことで自動的に並列化コードを出力することを目的としている。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の解析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、より並列効果の高い

*1：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)

コードを生成する。

以上のような、プログラムの実行性能向上を実現するC言語自動並列化トランスレータを目指している。

2．1　C言語自動並列化トランスレータの構造

初期作業として入力された逐次プログラムから中間データ構造を作成したのち、それに対する、並列性解析を行うことで並列性を抽出する[2]。この中間データ構造には、元のソースコードと等価であるタスクの木構造、また並列化において使用される様々な情報が格納される。

その後、タスクの実行時間と依存関係を考慮し、タスクの適切な粒度を求めるタスク粒度解析を行う。現在のトランスレータでは、内部でタスクスケジューリングを実行しタスクに対してプロセッサの割り当てを静的に行う。タスク粒度解析とは、タスクスケジューリングの効率を上げるために必要な作業となる。

最終段階では、各解析・変換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。詳しい説明は 3 章で行う。図 2.1 は並列化トランスレータの処理手順を表わしている。

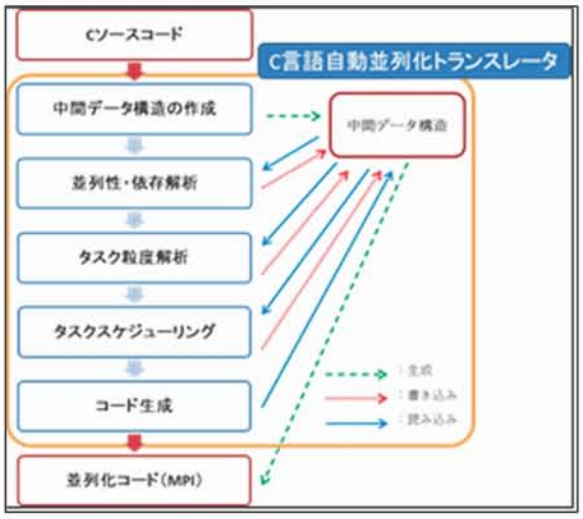

図 2．1　C言語自動並列化トランスレータ処理手順

2．2　タスクグラフについて

本トランスレータに読み込まれたソースコードはまず、タスクと呼ばれるコードセグメントに分解される。また、タスク間には依存関係が存在し、これらの先行・後続関係をエッジで表し、タスク同士をつないだグラフをタスクグラフと呼ぶ。また、ステートメントレベルのタスク以外にも、ブロックスコープ単位のタスク、制御構造単位でタスクとなるifタスク、forタスクや、main関数などを示すfunctionタスクといったタスクをMacroタスクとして定義する。図 2.2 はタスクグラフの一例を示す。

2．3　実行制御について

通常、逐次プログラムはプログラム内の命令を上から順に実行していく。しかし、並列プログラムの場合タスクを各プロセッサに分散させるために何らかの制御機構を実装する必要がある。

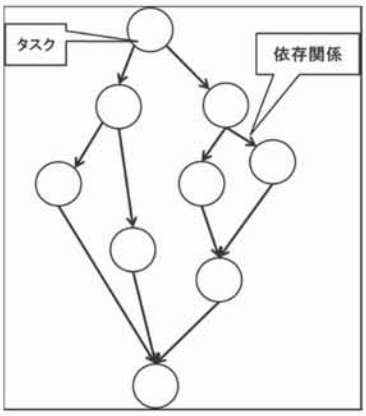

図 2．2　タスクグラフの一例

2．3．1　動的実行制御の問題点

以前のトランスレータにおいて実行制御方式は動的実行制御方式を選択していた。これは、実際に実行するまで実行時間などの実行時情報を解析することは困難であったためである。しかし、動的実行制御方式には以下のような問題点が存在していた。

①タスクの処理順序が実行ごとに異なるので、処理効率が不安定である

②マスタースレーブ方式による並列性の低下

③タスクを関数化して各プロセッサに処理をさせる際の関数呼び出しのオーバーヘッド

上記のような問題点を解決するために、本年度は静的実行制御方式による実行制御を導入する。

2．3．2　静的実行制御

動的実行制御に対して静的実行制御とは、静的スケジューリングによってプログラムの実行前にタスクスケジューリングを行い実行プロセッサへのタスク割り当てが終了している実行形態のことである。動的実行制御と異なり、実行前から実行プロセスが一意に割り当てが終了しているため、実行時の情報を取得するための処理が不必要であり、処理のオーバーヘッドが削減できることが上げられる。

静的実行制御は元の逐次実行プログラムの解析を行うことによって実現できる。依存性・並列性解析、タスク粒度解析など各種解析の結果を用いてより効率の良い並列実行となるように処理を割り振る。

3．並列コード生成

並列性・依存性解析が終了した時点では、一部を除き、タスクの初期粒度はステートメントレベル、すなわち細粒度である。従ってステートメント数が多いソースコードが対象の場合、タスク数は大きくなる。するとタスクスケジューリングが組合せ最適化問題であるため、その求解時間はタスク数の増加に対して指数関数的に増大する。このため、ステートメント数が多いソースコードが対象である場合、無駄な並列性を省き、タスクを適切な粒度にまとめる作業は必須となる。この、適切な粒度にまとめ上げられたタスクのことを「融合化タスク」と呼び、タスク粒度解析以降のタスクスケジューリング、コード生成の際には融合化タスクを1タスクとして扱う。

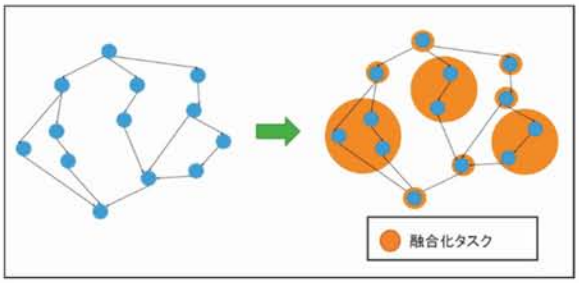

図3．1　融合化タスクイメージ

3．1　出力される並列コードの構造

本トランスレータの解析対象となるのはCプログラムである。Cプログラムは以下の図のような構成でプログラムが成り立つ。

include文
大域変数・関数定義
main文
{
宣言部
処理部
}

図3．2　逐次コードの構造

これに対して、出力される並列コードの構成は以下のようになる。

include文
+MPI用のライブラリを追記
大域変数・関数定義
main文
{
宣言部
+MPI用の各種宣言とMPI関数
処理部←並列化
+MPI関数
}

図3．3　並列コードの構造

追加するライブラリや各種宣言、MPI関数に関してはのちに詳しく解説する。また現在のトランスレータは関数の並列化には対応していない。並列化を行うのはmain文内部の処理に限るものとする。

main文内部の処理部の並列化についてであるが、以下の図のような記述によってそれは達成される。

```
if(rank==0){
    rank0が行う処理、通信
}
if(rank==1){
    rank1が行う処理、通信
}
・・・
```

図3．4　main文内部のタスクの記述

if(rank==x)(x:実行プロセッサ番号)という記述によって実行プロセッサが指定され、内部にそのプロセッサが行うべき処理を記述することによって並列処理を実現している。トランスレータが並列コードを出力する時点で、トランスレータ内部では依存/並列性解析・タスク粒度解析などの処理が行われ、複数のタスクが統合された状態になっている。この統合されたタスクを融合化タスクと定義し、出力においてはこの融合化タスクが細粒度の構造として扱われる。

3．2　出力される並列コードに埋め込むMPI通信命令

先行タスクと後続タスクを実行するプロセッサが異なる場合、それらのタスクは通信を行う必要がある。また逆に、先行タスクと後続タスクを実行するプロセッサが等しい場合にはデータの送受信の必要は無い。

本トランスレータによって行われるデータの送受信は、全てブロッキング通信とする。MPIによるデータ送受信の規格を以下の図に記す。

```
MPI_send ( 受信するデータの先頭アドレス,
           受信するデータの個数,
           受信するデータのデータ型,
           送信元プロセッサのランク,
           タグ,
           コミュニケータ )
```

図 3．5　MPIによる送信命令の規格

```
MPI_recv ( 受信するデータの先頭アドレス,
           受信するデータの個数,
           受信するデータのデータ型,
           送信元プロセッサのランク,
           タグ,
           コミュニケータ,
           送信プロセスの情報 )
```

図 3．6　MPIによる受信命令の規格

送受信するデータの先頭アドレス、個数、データ型などの情報はタスクグラフを作成したときの情報、構文木構造との照らし合わせによって取得する。

送受信するプロセッサのランクは、タスクグラフとスケジューリング結果とを参照することによってわかる。これによりタスク番号から、そのタスク番号が実行するのに必要なタスク番号の情報と、タスクが完了した際にその情報を送信すべきタスク番号を同時に得ることが可能である。

3．3　並列コード生成

コード生成の流れについて以下で見ていく。

・ヘッダファイル

MPIによる並列化コードにはプログラムのヘッダ部分に

#include<mpi.h>

が必要になる。これは、逐次コードから保存されたヘッダファイルを出力する際に追加で自動出力する。

・コマンドライン引数取得部

MPIによる並列プログラムは実行時にインライン引数によって実行時の実行プロセス数を指定しなければならない。よって以下のような記述が必要になる。

int main(int argc , char *argv[])

出力部がmain文の“()”を検出した際に自動的に出力する。

・各種宣言とMPI関数

MPIによる並列プログラムを自動生成するために必要な変数とMPI関数を出力する。

```
int    MPI_size , MPI_rank ;
MPI_Status    status;
MPI_Init ( &argc , &argv ) ;
MPI_Comm_size      (      MPI_COMM_WORLD      ,
&MPI_size );
MPI_Comm_rank      (      MPI_COMM_WORLD      ,
&MPI_rank ) ;
```

図 3．7　自動生成する宣言とMPI関数

上記の宣言とMPI関数は上記のものをこのまま出力される並列プログラム内に出力するものとする。変数名などはこちらで一意に与えたものである。

・main文内部のタスク

各タスクを出力する際には、まず自身を実行するプロセッサ番号を出力する。その後、受信命令が必要ならば受信命令を生成したのち、各タスクの処理内容を記述する。その後送信命令が必要ならば送信命令を出力し、一つのタスクの出力は終了となる。この繰り返しにより、すべてのタスクを出力する。

・MPI_Finalize ( );の挿入

“ MPI_Finalize ( ); ”の呼び出しによって、MPIを終了する。

main文の” return 0; “の前に自動生成する

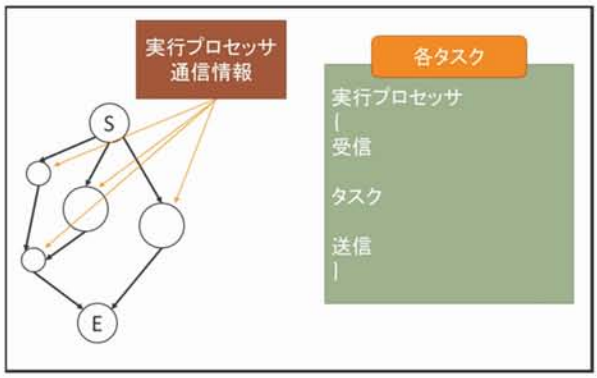

図 3．8　main文内部のタスク

4．タスクのコストチューニングツール

4．1　静的実行制御でのタスクコスト解析の問題点

本トランスレータにおいてタスクコストとは、タスクの持つ静的な情報から概算されるタスクの実行時間のことを言う。このタスクコストは、タスクの粒度を決定するための指標であり、タスク粒度の最適化、タスクスケ

ジューリングなどで用いられる。このタスクコストは並列プログラムを効率よく実行するために高い精度が求められる。現在のトランスレータによる実行時間解析は、静的メトリクスに基づく実行時間解析を行っている。これは、ソースコード全体を見た時にそのタスクが全体のどれくらいの割合を担っているか、ということをタスクの形状から概算しようとするものであるが、現状で繰り返し回数が不定であるループや、実行環境に繰り返し回数が左右されるタスクのような、実行してからでないとそのコストが一意に定まらないタスク群に対しては未だに高い精度でタスクコストが得られているとはいいがたい。

その、決して高い精度で得られているとは言えないタスクコストを使用してタスク粒度解析やタスクスケジューリングが行われていて、どの程度のオーバーヘッドが出ているのかわからない、といった問題が発生している。

そこで今年度は実際にトランスレータによる変換で得られた並列プログラムを実行し、その実行によって得られた実実行時間と概算されたタスクコストの比較表示、また得られた実実行時間をコストに並列プログラムを生成する機構を作成する。

4．2　チューニングツールの概要

中間データ構造上に、トランスレータによる解析結果のタスクコストを埋め込む。また、算出されたタスクコストを用いて出力された並列プログラムに時間計測用の関数を埋め込んだものを実行し、得られた実実行時間を同様に中間データ構造上に埋め込む。これによってトランスレータによる実行時間解析と実実行時間の比較が容易になる。また、得られた実実行時間を用いてスケジューリングを行えるようにトランスレータを拡張した。

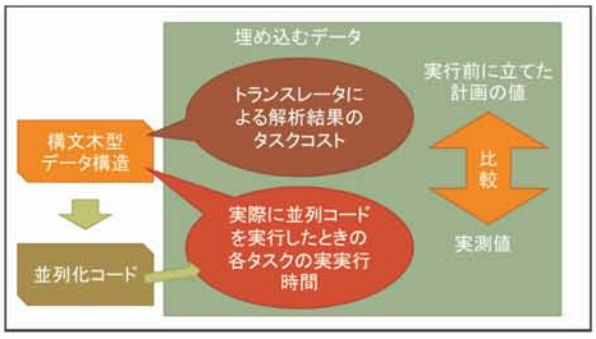

図 4．1　計画値と実測値との比較

5．評価

5．1　並列コード生成結果

以下にトランスレータによる逐次コードから並列コードへの変換結果の一部を示す。

```
if(MPI_rank==0){
MPI_Recv(&omega,1,MPI_FLOAT,3,1,MPI_COMM_WO
RLD, &status);
MPI_Recv(&nn,1,MPI_INT,1,1,MPI_COMM_WORLD,
&status);
gosa=jacobi(nn);
}
if(MPI_rank==2){
cpu1=second();
MPI_Send(&cpu1,1,MPI_DOUBLE,1,1,MPI_COMM_W
ORLD);
MPI_Send(&target,1,MPI_DOUBLE,0,1,MPI_COMM_W
ORLD);
}
if(MPI_rank==1){
MPI_Recv(&cpu1,1,MPI_DOUBLE,2,1,MPI_COMM_W
ORLD, &status);
MPI_Recv(&cpu0,1,MPI_DOUBLE,3,1,MPI_COMM_W
ORLD, &status);
cpu=cpu1-cpu0;
}
```

図 5．1　計画値と実測値との比較

このプログラムを見ると、タスクの割り当てが終了し、また通信が発生していることがわかる。

5．2　コードチューニング結果

今回、求められた実実行時間を用いてスケジューリング・コード生成を行おうと試みたところ、スケジューリングが終わらない、という結果になってしまった。

これは、求められる実実行時間がスケジューラが想定しているよりもはるかに大きな数値であり、計算を開始できなかったためと推測される。解決策としては、スケジューラの改良があげられる。

6．結論

本研究では、静的実行制御方式に基づく並列コード生成機構の実装を行った。以前までは何重にもネストされたループ文や多くの条件分岐があるタスクに対してタス

クコストを設定できないという問題から、動的実行制御方式の並列プログラムを出力していた。しかし、タスクの持つ構造に着目したタスクコスト算出方法が実装されたことからトランスレータ上において当研究室のスケジューリングソフトが実行できるようになり、それに伴って静的実行制御方式の並列プログラムを出力できるよう、出力機構の実装を行った。

また、実実行時間をタスクコストとして再スケジューリング、コード生成を行うことができる機構も作成した。これにより静的実行制御の欠点である実行時間解析の未成熟から発生するオーバーヘッドの減少を行うことができるようになった。しかし、スケジューラの改良を行わなければ実実行時間をコストに計算することが不可能なため、スケジューラの改良が今後の課題であるといえる。

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) 小林　裕昌：”C 言語自動並列化トランスレータの開発 - ソースコード静的メトリクスを利用したタスク粒度解析手法の提案とその評価 - ”, 修士論文, 成蹊大学工学部工学研究科情報処理専攻.2014.
3) 小倉　健太郎:“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” , 修士論文, 成蹊大学工学部工学研究科情報処理専攻.2016.
4) A.V.エイホ,R.セシィ:「コンパイラー原理・技法・ツール. 2」, 翻訳者 : 原田賢一. 全一巻. サイエンス社, 2009.