成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.50 No.1 (2013)pp.37-44

C言語自動並列化のための並列構造解析と動的実行制御の実現

遠山　純也[*1]，甲斐　宗徳[*2]

Implementation of Parallelism Analysis and Dynamic Parallel Execution Control for C Programs

Sumiya TOHYAMA[*1], Munenori KAI[*2]

ABSTRACT : In recent years, since it seems that speed up of single processor or single core has been going up to the utmost physical limit, parallel processing by multicore or multiprocessor becomes the mainstream to make the processing speed much higher. On the other hand, developing the effectively parallelized programs is very difficult for software developers. So, it is expected that automatic parallelization of existing sequential programs is accomplished. In this paper, we propose a parallelism analysis method to find parallel structures from original sequential programs, and implement its dynamic parallel execution method for automatically parallelized C programs.

Keywords : Parallel processing, Automatic parallelization, Dynamic execution control

(Received April 8, 2013)

1．はじめに

近年のコンピュータにおいて，物理的な問題から高速化の限界が指摘されてきている。その解決策として，マルチコアや複数のプロセッサによる並列処理を行うことで処理スピードを上げる方法がある。しかし，並列処理による処理効果の高い並列実行可能なプログラムを作成することはプログラム開発者にとって困難なものである。そのため，どのような人にも容易に並列効果の高い並列実行可能なプログラムを作成できるようにすることが望まれている。そこで本研究では逐次のユーザプログラムを読み込み，元のプログラムの実行結果と変わらず実行速度の高い並列プログラムに自動的に書き換える“C言語自動並列化トランスレータ”の完成を目指し開発を行っている。本論文では特に並列実行できるプログラム構造を特定し並列実行できる形式に変更する並列構造解析と，解析された並列性に応じて，複数のプロセッサに対するそれらの処理の割り振りを実行時に決定する動的実行制御を実現したので報告する。

*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)

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とは，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抽出し，変換を行うことで並列実行可能なコードを出力する。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コードを生成する。

本研究では分散メモリ環境で実行できるコードを出力することで，より大規模な並列実行可能な環境で利用できるようにしている。また並列実行可能なプログラムを出力することで，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となる。

3．C言語自動並列化トランスレータ処理手順

3.1　ソースプログラムの読み込み，中間データ構造の作成

各解析処理では共通の中間データ構造を解析し解析結果を更新することにより，処理を進める。そのため最初の処理として，ソースプログラムを読み込み，中間デー

タ構造を構築する必要がある。ここでは今後の解析ステップで必要となる情報を保存する必要があるため，元のソースコードの情報や，変数の情報，プログラムの制御構造などを中間データ構造として保存する。ソースプログラムから構文木を作成することで，読み込んだソースプログラムを完全に保存することが出来る。この構文木に変更を加えることでプログラムの変更を行うこともできる。また，構文木だけでなく変数の情報やプログラムの制御構造などの情報を保存し，後の解析処理で利用する。

3.2　並列性の抽出

実行結果を変えることなく並列に処理するためには，並列に実行することが可能な部分を探す必要がある。それぞれの処理に依存関係がある場合，そのままでは並列に実行することができない(Figure 1)。並列実行には変数を適切に通信しなくてはならない(フロー依存)。そのため，並列実行する処理内で必要となる変数に関する情報を解析する必要がある。また，処理を実行する際に順番を入れ替えることができない場合がある。並列実行を行うことで処理を実行する順番が入れ替わることがある。そのため，並列実行する際に出力などの処理の順番を変えることのできない処理は逐次に実行しなくてはならない(逆・出力依存)。また，並列実行できないプログラムを変換し，並列実行できる形式に変換することで並列性を抽出することも行う。

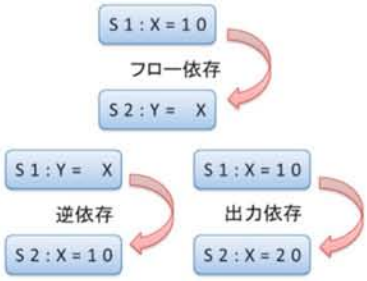

Figure 1　依存の種類

3.3　最適化

元のソースプログラムを並列に実行するだけでは高い並列効果は望むことができない。通信のためのオーバーヘッドや他プロセスの処理を待つことで発生する遅延，並列実行する処理における実行時間の違いなどの問題がある。それらの問題点を解消するために最適化処理を施す必要がある。それぞれの処理の大きさを平滑化することや，処理をまとめることで通信を減らすことなど，実行時により高い性能を出すための変換を行う。

3.4　コード生成

実行できる形式で出力するためには正しい文法であり，正しい処理手順にそって実行できる形式で出力する必要がある。中間データ構造から実行できる形式へと変換し，並列実行を行うための処理を加えて出力する必要がある。

4．従来の研究との比較

4.1　中間データ構造の構築

従来までの研究では中間データ構造が解析を行うごとに構造が変換されていたため，解析ごとに独自のデータ構造を解析していた。そのため類似したプログラムが多数あり保守性や拡張性が乏しい状態となっていた。そのため，本研究では統一された中間データ構造を作成するとことから始めた。統一された中間データ構造を作成することにより，処理間での解析結果の確認や，処理手順の見直し，新たな処理を加えることなど柔軟な拡張を行うことができるようになる。また出力時にのみ専用の構造に変換することにより複数の出力形式で出力することができるようにする。そうすることにより，より効率の良いコードを出力することや実行結果の比較など研究開発が容易になる（Figure 2)。

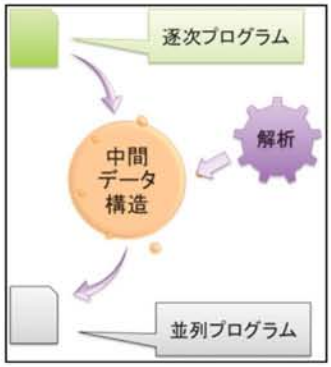

Figure 2　中間データ構造

4.2　並列実行モデルの改良

従来の研究では並列実行可能な処理の割り当てを静的に行っていた。解析結果をもとに処理ごとに実行するプロセッサを決め，プログラム中に埋め込むことで並列実行を行っていた。この方法は余分な処理手順などを含まず高速に実行することができるが，この手法では実行する処理の大きさや，並列実行するプロセスの状態など，プログラムの解析では知ることのできない実行時の情報などにより正確な割り当てを行うことができない。そのため，きちんと処理を均等に分割することができず待ち時間などの遅延のため効率よく実行することができない。これを解決するために，実行時に並列実行を行う処理を

各プロセスに対し再割当てを行うことで高速化を図ることが出来るように変更した（Figure 3）。

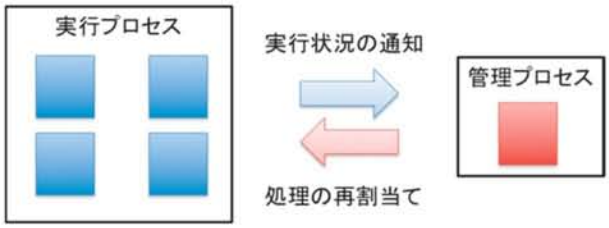

Figure 3　管理プロセスのタスク処理再割当て

5．並列化処理

5.1　中間データ構造の作成

中間データ構造の作成では，まずプリプロセス展開を行うことでincludeされる標準ライブラリの検査を行うことやソースプログラムをC言語のソースファイルとして構文解析を行うための変換を行う。標準ライブラリの検査は必要なライブラリがincludeされているか，ライブラリを追加した際に宣言が重複しないかを調べる。プリプロセス展開が行われたプログラムに対し構文解析を行う。構文解析ではまず字句解析を行い入力されるプログラムを構成する字句を抽出する。抽出された字句がどのような意味を持つ構文規則で構成されているのかを構文解析を行うことで判定する。構文解析を行うことで各構文規則を表す構文木を作成し中間データ構造へ保存する。この構文木は入力されるプログラム構造を保存しているため，この構文木に対し解析することで元のプログラム構造に対応した解析処理を行うことが可能となる。

5.2　並列構造解析

依存解析ではタスク内の依存解析を行い，その解析結果を元にタスクグラフを構築することでタスク間の依存関係を解析する。

5.3　タスク内の依存解析

タスク内の依存解析ではタスク内で使用される変数を依存情報と共に保管する。依存情報の作成は式を表す構文木を解析し，式の演算子からオペランドへのアクセスを調べる。依存情報は識別子に対するread，writeの情報や識別子が指し示す変数が保存される（Figure 4）。またポインタ解析を行いポインタ変数の参照先を保存することでプログラム上に現れない依存情報を解析することが可能となる。また複数のタスクが内包される複合タスクの場合，すべての内包されるタスクの依存情報を合成したものとすることで外部から複合タスクを一つのタスクとして見た場合にそのタスクに対する依存情報を正しく保存できるようにした。

ポインタ解析はプログラムの実行前にプログラムソースの解析を行なってポインタ変数がどのロケーションにアクセスするのかを解析するステップである。ポインタ解析を行わない並列性解析の仕様では，安全のためポインタ変数が含まれるステートメントは逐次処理をする。これはポインタ変数が様々なロケーションにアクセスする可能性があるため通常の変数のread/write解析をして依存を調べるだけではデータ依存を見抜けないからである。

そこでポインタ変数がどのロケーションをアクセスするのかを解析するポインタ解析を行い，アクセスロケーションを明確にする。その結果，それまで逐次処理しかできなかった一連のステートメントも並列性が見つかる可能性が出てくる。

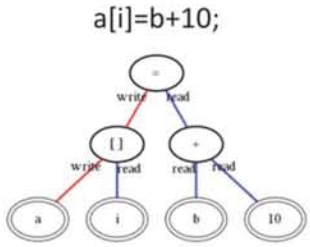

Figure 4　プログラム例に対する依存解析例

5.2.1　タスクグラフの構築

タスク間の依存解析では並列実行を行うのに必要となる各タスク間の通信を解析する。各タスク内の依存情報を使用し，それぞれのタスク内で使用される変数をどのタスクを行ったプロセスから受け取るべきかを解析する。逐次プログラムでの変数には最後に変数に対し代入を行ったステートメントでの値が保存されている。そのため，並列実行を行う場合も同様に，逐次処理で最後に代入を行ったステートメントを表すタスクから値を受け取る必要がある。そのため，それぞれのタスク間にはその変数に対する依存が存在している。タスク間の依存がないタスク同士は通信を必要としないため，それぞれのタスクは並列実行を行うことが出来る。また，各タスクと各依存情報からタスクグラフの構築を行う（Figure 5）。タスクグラフでは各ノードがそれぞれのタスク，各依存情報がエッジとなっている。

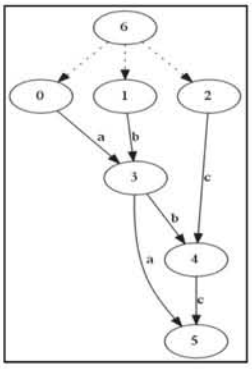

Figure 5　作成されたタスクグラフ

6．最適化

6.1　タスク粒度解析

並列・依存性解析が終了した時点では，タスクの初期粒度はステートメントレベルとしているため，タスクスケジューリングのステップで受け取るタスク数は元のソースコード中のステートメント数に近い数となっている。そのため，ステートメント数が多いソースコードを対象とした場合，タスクスケジューリングに要する時間が指数関数的に増大してしまうという問題がある。

従来の研究では，ステートメントごとのコスト（変数の個数）と依存強度（通信によって渡さなければならない変数の個数）を考慮することによってタスク融合を行い，タスク数を減少する解析手法を試作した。その結果，全体のタスク数を減少させ，タスクスケジューリングに要する時間を短縮することに成功した。

本研究では，プロセッサ数に基づく最大並列度を考慮するために，同時に並列実行可能なタスク同士の融合（Parallelタスク融合。以下，P融合。）はタスクのコストを使用して融合を行う。また，タスクグラフにおける同一エッジ上のタスク同士の融合（Sequentialタスク融合。以下，S融合。）についてはタスクの依存強度を使用して融合を行う。以上の２つの融合を組み合わせた手法を提案する。この提案によって，これまで融合の対象としていなかったタスクについても融合を可能にすることができ，タスク数の更なる減少が期待できる。

6.2　ループリストラクチャリング

一般的に，プログラムの実行時間のほとんどはループ中で行われる何らかの処理によって消費されている。したがって，自動並列化に大きく影響しているのは主にループであると考えられる。

従来の研究では，元のC言語で記述されたソースコード中のfor文に対して，C言語自動並列化トランスレータの並列性解析部で，並列性解析を行い，解析の結果によってループの再構築が元のソースコードに対して行われていた。本研究では，新しく解析用の「構文木構造」がつくられた。本研究では，ループリストラクチャリングの対象をfor文とする。一見並列性の見えないfor文に対して構文木構造を利用して並列性解析を行い，その解析の結果ループの再構築が可能ならばループリストラクチャリングという手法を用いて，構文木構造に対してループの再構築を行う。構文木構造を用いてのループの並列性解析，ループリストラクチャリングの実装，ループの再構築に関しての構文木構造の変更を行うことで，最終的に並列化コード生成後，その並列化コードを実行する際の実行時間の短縮を狙う。

6.3　通信の最適化

逐次で書かれたコードを実行時間短縮のために自動並列化を行う場合，自動抽出されたタスクがプロセッサに割り当てられる。各プロセッサはそのタスク処理を行うために必要な情報の受け渡し，即ち通信を行わなければならない。この通信はMPI（Message Passing Interface）という並列化ライブラリを用いて行う。先行，後続関係にある２つのタスク間でデータ依存関係により送受信しなければならないデータが複数ある場合，これまでの研究ではこの情報の受け渡しを１対１プロセッサ間通信を複数回行うことで行っていた。また，この１対１通信では通信を行うプロセッサの分だけセットアップ（通信経路確保）を必要とする。

本研究では，より実行時間の高速化を図るために，プロセッサ間の通信部分に着目し，MPIのライブラリに用意されている，MPI_Bcastにも適応させることを目的とする。また複数データの送受信を行わなくてはならない場合も構造体を用いて１つにまとめブロードキャスト通信によって一括でデータの送受信を完了させることが可能となる。しかし全ての１対１通信をブロードキャストに置き換えるのではなく１対１通信とブロードキャストの使い分けを行うこととする。このブロードキャスト命令を利用することにより通信時間の短縮のメリットが見込め，最終的にはプログラム実行時間の短縮につながると考えられる。

7．実行制御

7.1　処理の関数化

並列実行を行うためには各プロセスに任意の処理を実行させる必要がある。しかし，他のプロセスに任意の処

理を実行させることは難しい。それを実現させるために並列化トランスレータでは各処理を関数にして切り出す（Figure 6）。

関数に切り出した処理を実行させる際には依存する変数の値が必要となる。しかし，元のプログラムから切りだされているため局所変数を使用することができない。そのためグローバル変数を使用し，各処理間で変数の受け渡しを行う。また，異なるプロセスで実行された処理に依存する変数をある場合，通信を行うことで必要な変数を受け取ることが出来る。そのため，各関数は必要な変数を受け取る受信，実際の処理を行う実行，代入された値を次の依存する処理に送る送信の３つの処理から構成される。

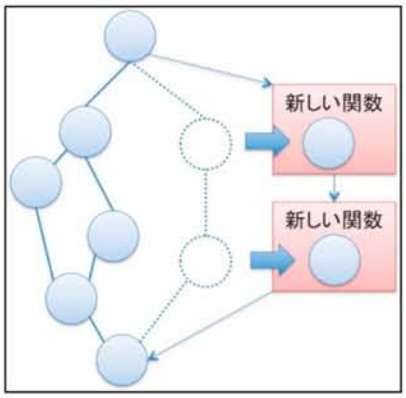

Figure 6　タスクの関数化

切りだされた関数の関数ポインタを配列に保存し関数テーブルを作成する。全プロセスは同一の関数テーブルを持ち，保存された処理を順次実行する。このようにすることで他のプロセスに任意の処理を実行させたい場合，関数テーブルのインデックスを渡すことで任意の処理を行わせることができる（Figure 7）。

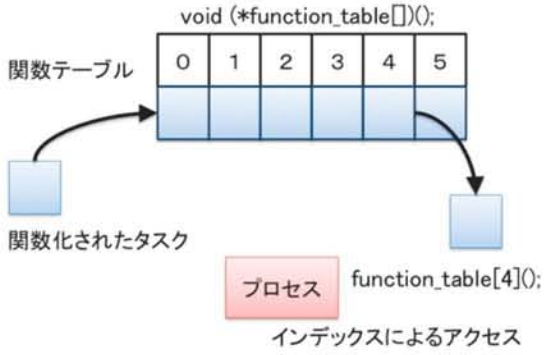

Figure 7　タスクの保存と呼び出し

7.2　処理の初期分散

プログラムを並列実行する際により高速に動作させるためには，適切に処理を並列実行させる必要がある。並列実行させる際に，他のプロセスに任意の処理を実行させるためには関数テーブルのインデックスを通信により渡す必要があるため，通信のオーバーヘッドが発生してしまう。そのため，処理を実行前の段階で分散させることで次に行う処理を決めておくことでより高速に実行できるようにする。ここでは，タスクの依存関係や処理時間を求めた実行時間解析の結果を用いタスクスケジューリングにかけることでより最適な初期分散を求める。初期分散されたタスクを順次処理することで元のプログラムを並列実行することが可能になる。

7.3　タスク管理

初期分散は元のプログラムを静的に解析した結果から求めるため，プログラム解析では実行時のタスクの正確な処理時間を求めることができない。解析時に割り当てられた処理を並列に実行するだけでは高速化が望めない場合がある。そのため，実行時に再割当てを行うことでより高速に実行することが可能になると期待される。各プロセスが割り当てられた処理を実効する際には必要な変数を受け取る必要があり，依存するタスクが終了していなくてはならない。依存するタスクが処理中の場合，他の依存しないタスクを先に実行することや，他のプロセスに割り当てられている実行可能なタスクを先に実行することで処理時間の無駄を削減し高速化を図る。それらの実行時のタスクを管理するために，それぞれ割り当てられたタスクをタスクテーブルとして管理する。

7.3.1　タスクテーブル

タスクテーブルには依存するタスク，実行するプロセス，タスクの実行状態が保存される。依存するタスクは依存する変数をどのタスクを実行したプロセスから受け取る必要があるのかを保管している（Table 1）。実行するプロセスは初期分散で割り当てられたプロセスのIDが最初に保管される。そして，実行時に動的再割当てが行われた際に再割り当てされたプロセスのIDが再登録される。タスクの実行状態はタスクが実行前，実行中，完了のどの状態なのか，実行可能か，依存するタスクが終了していないために実行可能になっていないのかなどの情報を保管する。依存する変数がない場合などは初期状態が実行可能となる。動的再割当てをする際に自身の持つタスクが完了した場合や，依存するタスクが完了していないため実行できない場合などに他のプロセスのタスクを実行する。その際に他のプロセスの実行可能タスクを優先的に実行することですぐに次のタスクを実行できる。

Table 1 タスク管理表の例

タスクID	プロセス	受信	送信
0	0		0,1
1	0	0	4
2	1	1	2
3	2		3
4	1	2,3	5
5	0	4,5	

7.4　通信の作成

並列プログラムを実行するためには通信を行うプログラムの作成を行う必要がある。通信のプログラムはタスクグラフの全てのノードに対し，それぞれ作成する。タスクグラフ内のエッジは，受信タスク，送信タスク，通信情報の３つの情報を持っているため，それぞれのエッジに対し個別のIDを割り振る。IDを割り振られたエッジを通信の表のインデックスとして使用する。

それぞれの通信には送信元タスクから送られる情報のうちの受信タスクで必要となるデータをコピーする必要があるため，通信プログラム内で通信が完了した後，自身の内部データに変数を退避させるプログラムを作成する必要がある。対比させる変数をデータに登録しておきコードの出力時にプログラムを構築する。

８．実行時再割当て

実行時にタスクの再割当てを行うためには実行時に各プロセスの実行状況を管理する必要がある。そのためには各プロセスは通信を行い実行時の情報を共有しなくてはならない。しかし，通信を行おうとする際に通信を行うプロセスがタスクの実行中では通信を行うことができない。そのため，実行中に各プロセス同士が通信を行うことは現実的ではなくなる。その問題を解決するために各プロセスの実行状況を管理するための管理プロセスを用意し，その他の処理を行うプロセスを実行プロセスとした（Figure 8）。各実行プロセスは処理の実行状況を管理プロセスに通知し，管理プロセスは伝えられた情報を元に再割当てを行い，実行プロセスに再割当てを通知する。このようにすることで，各プロセスが実行状況を送信する際に受信プロセスが処理の実行中でなくなるため，同期が発生しない。

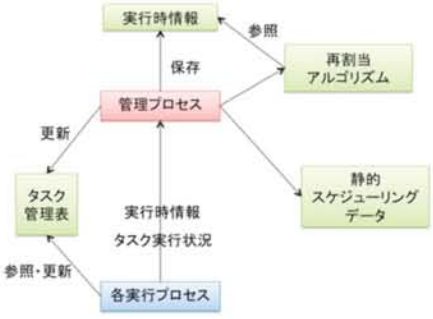

Figure 8　実行時の情報の流れ

９．並列実行可能なプログラムの出力

中間データ構造では並列化された構造になっていないため，中間データ構造を出力用のデータ構造に変換し出力する必要がある。出力用のデータ構造と中間データ構造を分離することにより，一つの出力用のデータ構造から複数の出力プログラムを作成することが出来る。そのため，新しい並列実行のモデルや出力プログラムの改良などの開発を続けていくことが出来る。また，出力するプログラムは元のプログラムから大きく変更されてしまうため改良することが難しくなるという問題点がある。この問題点を解決するためには，元のプログラム構造を大きく壊すことのないよう，大きな変更を加えず，並列実行できる処理を分離することで目標を達成することを目指す。

9.1　実行結果

```
int my_work
(unsigned int time)
{
 return time;
}
int main()
{
 int a, b;
 a = my_work(1);
 b = my_work(2);
 printf("%d¥n", a + b);
 return 0;
}
```

Figure 9　入力プログラム例

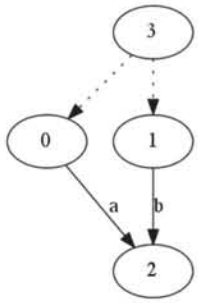

Figure 10　作成されるタスクグラフ

Figure 9 がサンプルとして入力されるファイルになる。このファイルでは２つの変数に初期値を代入し，その合計を出力する簡単な例になっている。この例では処理が少なく並列効果は見込めないが，並列実行が正確に行われていることを示すためにこの例を使用した。このプログラムでは変数aへの代入と，変数bへの代入部分は並列実行可能なる。しかし，合計を出力する箇所では通信を行い，適切な通信を行わなくてはならない。このプログラムを入力し構築されるタスクグラフがFigure 10になる。この図のダミータスク 3 の下にある２つのタスクが変数の代入となっている。この２つのタスクの間にエッジはなく，並列実行することが可能になっていることがわかる。更に下にある２番のタスクが合計を出力するタスクとなっていて，代入を行う２つのタスクから変数を通信しなくては行けないことを示している。

```
int my_work(unsigned int time){
  return time;
}
void acpt_func0(){
  call_recv(0);
  comm.data0.a=my_work(1);
  call_send(0);
}
void acpt_func1(){
  call_recv(1);
  comm.data1.b=my_work(2);
  call_send(1);
}
void acpt_func2(){
  call_recv(2);
  printf("%d¥n",comm.data2.a+comm.data2.b);
  call_send(2);
}
```

Figure 11　出力される関数

```
union Comm{
  struct {
   int a;
  }data0;
  struct {
   int b;
  }data1;
  struct {
   int a;
   int b;
  }data2;
}comm;
```

Figure 12　通信用のデータ構造

Figure 10のタスクグラフから並列プログラムを出力すると，まずそれぞれのタスクが関数化されたFigure 11 のタスクリストが出力される。この関数では使用される局所変数が別の構造体となっている。その構造体を示したものがFigure 12 の構造体である。これは各関数間でのデータの受け渡しや，他のプロセスとの通信に使われるデータ構造になっている。

```
void static_work()
{
  int i;
  for (i = 0; i < TaskTableSize; ++i) {
    if (my_rank == task_table[i][1])
      function_table[i]();
  }
}
```

Figure 13　タスクの実行

Figure 11 のようなタスクリストを実行する部分がFigure 13 で示すタスクの実行プログラムになる。これらのプログラムをコンパイルすることで並列に正しく実行することができた。

9.2　今後の課題とまとめ

本研究では，中間データ構造を作成し，並列性解析を行うことで並列性の抽出が可能となり，並列プログラムを生成することで逐次プログラムと同様の出力を得ることができた。しかし，ここでは最適化処理が行われていないために，多くのプログラムで速度の向上は見られなかった。

そのため，今後の課題は並列実行が可能となったので，より高速に動作するプログラムを出力することになる。そのためには，各種最適化を行うことや，実行制御の改善を行わなくてはならない。最適化や，実行制御の改善のためにはタスクにかかる処理時間の算出や，タスクの粒度を大きくすることで，より効率良く実行を行うことが必要になる。

参考文献

[1] A. V. エイホ，R. セシィ（原田賢一訳）:「コンパイラ―原理・技法・ツール」，サイエンス社，2009

[2] 中田育男:「コンパイラの構成と最適化 第２版」，朝倉書店，2009

[3] ParrTerence（伊藤真浩 訳）:「言語実装パターン　コンパイラ技術によるテキスト処理から言語実装まで」，オライリージャパン，2011

[4] 蒲野茂幸:「SIMD 最適化向けソースコードレベルでのコード変形」，東京工業大学大学院情報理工学研究科修士論文，2008

[5] 山名淳司:「GPC による導出原理の最適化」，早稲田大学理工学研究科修士論文，2005

[6] 松本真樹，片野聡，佐々木敬泰，大野和彦，近藤利夫，中島浩:「ヘテロ型大規模並列環境の階層型タス

クスケジューリングの提案と評価」, 情報処理学会論文誌プログラミング(PRO), 2(1), 1-17, Jan.2009

[7] 美濃本一浩:「C 言語自動並列化トランスレータの開発」, 成蹊大学工学部工学研究科情報処理専攻修士論文, 2005

[8] 國枝義敏, 津田孝夫:「自動ベクトル化コンパイラのための制御関係解析法」, 情報処理学会論文誌, 30(9), 1164-1174, Sep.1989