B-001

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と並列化チューニングツールの実装

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

近藤 竜也† 阿加井 星† 小倉 健太郎† 甲斐 宗徳†

Tatsuya Kondo Sei Akai Kentaro Ogura Munenori Kai

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。そこで筆者らは C 言語自動並列化トランスレータ[1]の開発を進めている。

プログラムの持つ並列性を自動検出し最小実行時間で並列処理するには、タスクスケジューリングに基づく静的な実行制御が必要である。並列化トランスレータでは、ステートメントレベルのタスク初期粒度から自動的に粗粒度方向へタスク粒度を調整する。これにより、細粒度タスクの並列処理で起こるオーバーヘッドを小さくし、またタスクの総数を減らすことでタスクスケジューリングにかかる時間を削減することが可能となる。本稿では推定タスクコストとタスクスケジューリング結果により静的実行制御方式に基づく MPI(Message Passing Interface)を用いた並列プログラムを自動生成する機構を実装した。また、推定タスクコストには精度を上げる余地があるため、推定タスクコストと実並列実行からの実コストを比較し、再度チューニングするツールも併せて実装した。

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並列化トランスレータは、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、プログラムに内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並列実行可能なコードへ変換し、出力する。

2.1. トランスレータ処理手順

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。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、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。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構文木構造、また並列化において使用されるデータ依存関係など様々な情報が格納される。

中盤の作業として、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。並列化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。一般的に細粒度タスクにかかる実行時間よりも通信にかかる処理時間の方が大きいという場合が多い。このような無駄な通信を削除するために、タスク粒度解析ではタスクの粒度をステートメントレベルである初期粒度から粗粒度へ調整する。またタスクの総数を減らすことでタスクスケジューリングにかかる時間を削減することが可能となる。

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自動生成し出力する。

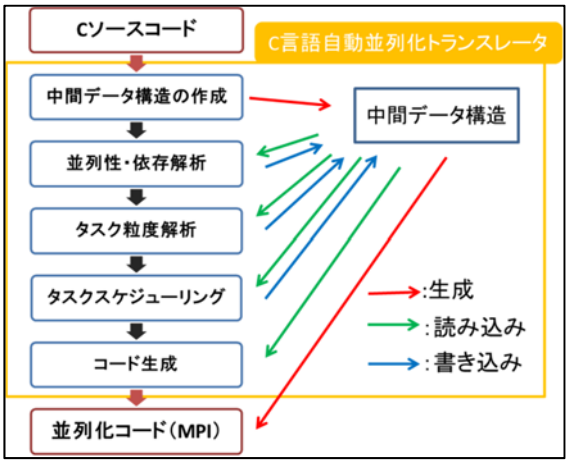

図 2.1 トランスレータ処理手順

2.2. タスクグラフについて

並列化トランスレータに読み込まれたソースコードは、解析器によって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す。依存関係のある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、ブロックスコープと制御フロー文を持つ if タスク、for タスクや、関数を示す function タスクといったタスクをマクロ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. タスク粒度解析

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

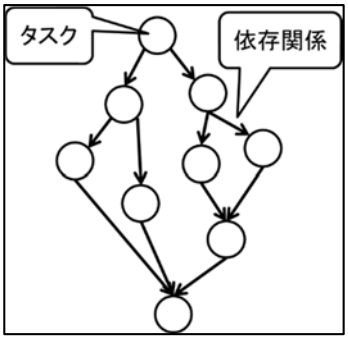

図 2.2　タスクグラフの例

本トランスレータでは、依存・並列性解析が行われた時点でタスクの初期粒度はステートメントレベルである。つまり、存在するタスク数は元のソースコードのステートメント数に相当する。そのため大規模なソースコードになるほどタスク数は増え、タスクスケジューリングを行う際に、その求解時間はタスク数に対して指数関数的に増大する。また、初期粒度のタスクグラフにおいてはタスク間の通信回数も増大し、並列プログラム実行の際に大きなオーバーヘッドとなる。そのためタスクの総数を減らす必要があり、そのための手段としてタスク融合手法が挙げられる。

2.3.1.　タスク融合

前述の通り、タスクスケジューリングの求解時間の削減、並列プログラム実行時のオーバーヘッドの削減を目的とし、タスクを統合することをタスク融合と呼ぶ。融合したタスクは単一プロセッサで逐次実行され、実行途中で他のタスクとの通信は発生しない。並列化トランスレータでは Parallel タスク融合と Sequential タスク融合が実装されている。どちらのタスク融合においても、タスク融合によりクリティカルパスが変化してしまう場合は、最適スケジューリングが不可能となってしまうため融合は行わない。

1.　Parallel タスク融合

Parallel タスク融合は、同時に並列実行可能なタスク群の中でタスクの和が最小となる 2 つのタスクの融合を指す。ただし、そのタスク群のタスク数が与えられたプロセッサ数未満になる場合は、生成された並列プログラム実行時における最大並列度が低下してしまうので融合の対象としないものとする。図 2.3 に Parallel タスク融合対象の検出例を示す。黄色の円で囲まれた 2 つのタスクが Parallel タスク融合可能なタスクを示している。

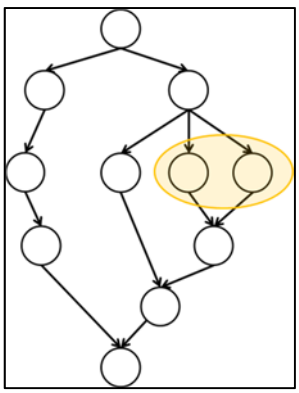

図 2.3　Parallel タスク融合対象の例

2.　Sequential タスク融合

Sequential タスク融合は、先行タスク、または後続タスクとの依存関係が大きいと考えられる 2 つのタスクを融合することである。タスク間の通信で送受信されるタスクの個数を依存強度と呼ぶ。コストと依存強度の比率が大きい場合、それら 2 つのタスク間の依存関係が大きいと考えられる。ただし、タスクを融合することでタスクの依存関係にループが発生してしまう場合は融合の対象としないものとする。図 2.4 に Sequential タスク融合対象の検出例を示す。緑色の円で囲まれた 2 つのタスクが Sequential タスク融合可能なタスクを示している。

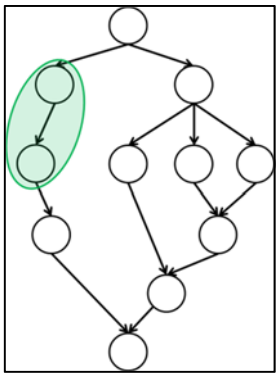

図 2.4　Sequential タスク融合対象の例

2.4. 実行制御について

2.4.1. 動的実行制御

以前の並列化トランスレータにおいて実行制御方式はマスタースレーブ方式による動的実行制御を選択していた。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が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ため、処理効率が不安定
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、タスクコストなどの実行時情報の推定の改良が行われた。そのため、本年度は推定したタスクコストとタスクスケジューリングを組み合わせる静的実行制御方式による実行制御を導入した。

2.4.2. 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の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る。そのため、処理のオーバーヘッドの削減が期待できる。ただし、並列処理効率を良くするためにはタスクスケジューリングの精度を良くする必要がある。

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。データ依存解析、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. 並列化情報統合手法

タスクスケジューリングをする上で初期はステートメントレベルとしているタスク粒度を最適な粒度へ調節する作業は必須となる。タスクの粒度を調節する際にタスクの融合やループリストラクチャリングなどの粒度の変更に対応して粒度情報の保持や更新をする必要がある。そのためのデータ構造として、並列化情報統合手法を実装した。

今までの並列化トランスレータが利用していた解析処理手法では、マクロタスク（内部にタスクグラフを持つタスク）の解析を行う際に、木構造を上から順に走査してマクロタスク内の各タスクにアクセスして情報を取得していた。

今回実装した手法では、統合した情報をマクロタスク自体に持たせてそこにアクセスすることで情報を取得できるようにした。これにより、余分な走査を省くことができ解析処理にかかるコストを削減することが可能になる。

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。

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。状態は、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、関数の実行を行うための exec がある。

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方の処理ができない状態のことである。

制御依存とは、データ依存が無くても処理を評価したのちに実行する必要が生じる状態のことである。例として、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場合が挙げられる。

依存情報を一つにまとめて保存するために、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。並列化情報を一つの場所にまとめるにあたって、配列のように複数の値を持った変数や、構造体・ポインタといった一つの変数についてアクセスするオブジェクトが複数生成される可能性のあるものが存在する。これらの変数は、解析を進めるうえで解析方法やメモリの管理の面でネックとなる。そこで、本研究ではこれらの変数に対する解析の方法について改良を施した。

配列変数については、配列全体をひとつの変数として解析を行っていたものを、要素単位まで掘り下げて依存関係の情報を取得できるようにした。

構造体については、元に宣言された変数のタスクに情報をもたせ、その構造体のオブジェクトが出現したときには元の変数へアクセスさせることで情報の重複を防ぐ。

ポインタ変数に関してはアドレスの指す先のタスクに情報を持たせて置き、そのアドレスを指そうとする変数が現れたときにはそのタスクの情報にアクセスさせることで情報の重複を防ぐ。

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。そのための新しいフォーマットを作成し、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。今回定めたフォーマットでは、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、必要な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を図 3.1 に示す。

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 3.1　変数の依存情報クラス

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。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。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。

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。

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、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。

3.2.2. スケジューリング結果の保存手法

依存情報の保存手法と同様に、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。保存する情報を図 3.2 に示す。

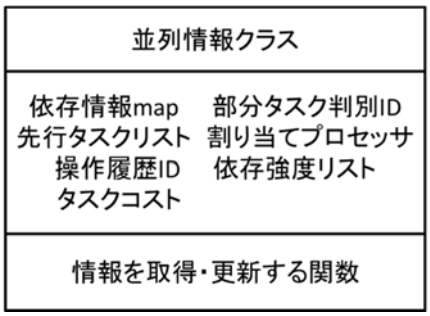

図 3.2　タスクの並列情報保存クラス

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（部分的に最適化をするために検出したタスク群）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ッサ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。

4. タスクグラフに対する操作履歴

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では、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。

自動で並列化を行うにあたり、行ったタスク融合などの操作が並列処理時間の改悪につながってしまう場合がある。その場合、結果が理想のものとは異なってしまうため、その操作まで処理手順を遡り、処理を取り消したり別の操作へ変更する必要がある。

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。

4.1. 操作履歴を保存する操作

タスクへの操作履歴を保存する操作は以下の 3 操作である。

1. 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に対する融合（Sequential 融合）や、並列関係にあるタスクに対してタスクの融合（Parallel 融合）を行う可能性がある。タスク融合の例を図 4.1 に示す。

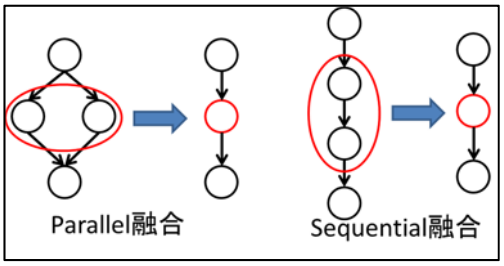

図 4.1　タスク融合

2. 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。タスク分割の例を図 4.2 に示す。

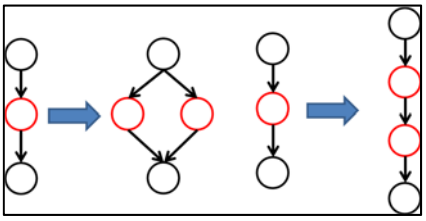

図 4.2　タスク分割

3. 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。

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。タスク複製の例を図 4.3 に示す。

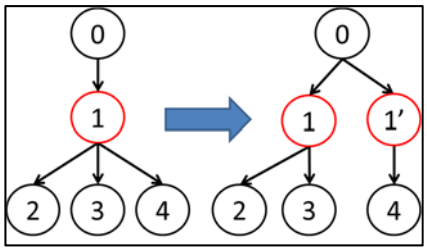

図 4.3　タスク複製

4.2. 操作履歴を保存する方法

操作履歴は、操作が行われる度に順番に保存していく。保存形式にはハッシュマップを使用し、キーには操作 ID、値に操作情報を保存する。操作情報を図 4.4 に示す。

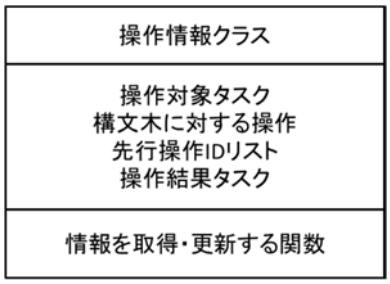

図 4.4　操作履歴に保存する情報

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作られたタスクの情報を保存する。

4.3. 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操作 ID を指定する。指定した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。図 4.5 と図 4.6 に操作履歴の出力形式と例を示す。

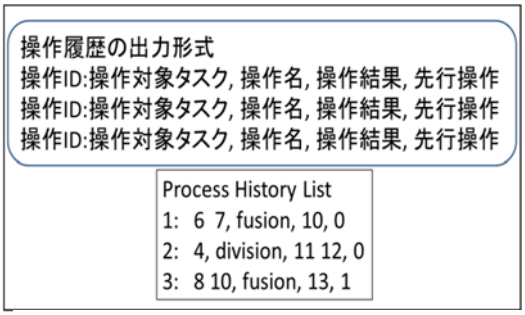

図 4.5　操作履歴出力形式と例

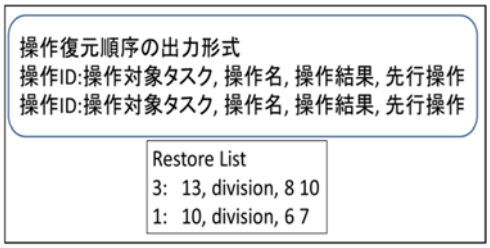

図 4.6　操作復元順序出力形式と例

5. 並列コード生成機構

本トランスレータは解析や出力に独自の中間データ構造を用いており、中間データ構造が保持している構文木構造からは元の逐次プログラムと等価なものを出力することが可能になっている。トランスレータの並列コード生成機構はその全体の構文木を再帰的にたどりながら、中間データ構造上に保存されている並列化情報と照らし合わせて並列実行に必要な情報を追加出力していく。

またタスクコスト算出精度が向上したため、スケジューリング技術と組み合わせて静的実行制御に基づく並列プログラムの自動生成が可能となった。

5.1. 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逐次実行用の C プログラムである。逐次 C プログラムの形式を図 5.1 に示す。

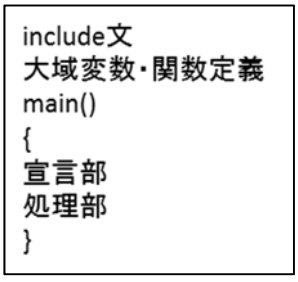

図 5.1　逐次コードの構造

これに対して、出力される並列コードの構造は図 5.2 のようになる。

図 5.2　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解説する。main 関数内部の処理部の並列化は、図 5.3 のような記述によって可能となる。

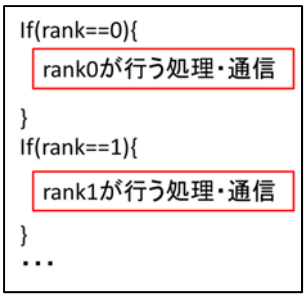

図 5.3　main 関数内部のタスクの記述

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データ依存解析・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 1 つの単位として扱われる。

5.2. 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、MPI によるデータ送受信の規格を利用する。

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、そのタスク番号を実行するため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。

5.3. 並列コード生成

コード生成の流れは以下のとおりである。

- ヘッダファイル

MPI による並列化コード実行に必要な”#include<mpi.h>”を埋め込む。

- コマンドライン引数取得部

MPI による並列プログラムでは実行時にコマンドライン引数によって実行時の実行プロセス数を指定する。

- 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。自動生成する宣言と MPI 宣言を図 5.4 に示す。

```
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 5.4　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。

- main 関数内部のタスク

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プログラム中に埋め込む。（図 5.3）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 を終了する“ MPI_Finalize ( ); “の呼び出しを埋め込む。

6. 並列化におけるチューニング

6.1. 静的実行制御での問題点

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[2]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のかをタスクの記述内容から概算しようとするものである。しかし、現状では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るため、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をする機構を作成した。

6.2. 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。これによって図 6.1 に示すように、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。比較したデータを利用し、タスクコストを調整して再スケジューリングをすることで、より短い実行時間で並列実行できるプログラムの生成が期待できる。

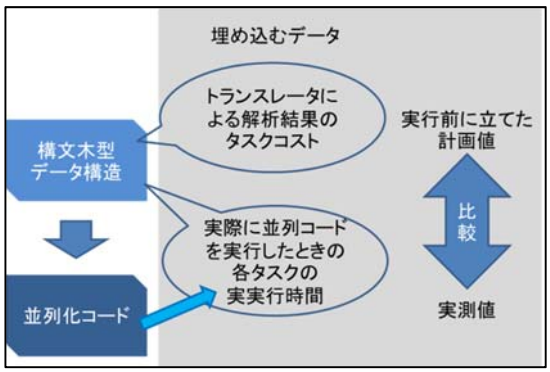

図 6.1　計画値と実行値の比較

7. おわりに

本研究では、静的実行制御に基づく並列プログラム生成のために、並列化情報を保存する新たなデータ構造や、並列プログラム生成機構を開発した。

また、タスクスケジューリングのもととなるタスクコストの推定は現状では見積もりが実行前では不可能な部分もあるため、実際に実行した上でタスクコストをチューニングできるようなツールの開発を行った。

参考文献

[1] 武市 和真, 遠山 純也, 小林 裕昌, 甲斐 宗徳, “C 言語自動並列化トランスレータの開発:構文木をベースとした並列構造解析と動的実行制御の実現”, FIT2013（第 12 回情報科学技術フォーラム）, 第 1 分冊, pp.237-240, 2013

[2] 阿加井 星, 小林 裕昌, 甲斐宗徳, ”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 - ”, FIT2014（第 13 回情報科学技術フォーラム）, 第 1 分冊, pp.133-136, 2014