B-011

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-

Development of Automatic Translator from C Programs to Parallel Programs Using MPI
- Implementation of Parallel Structure Analysis Based on Syntax Tree and Dynamic Execution Control -

武市　和真†　遠山　純也†　小林　裕昌†　甲斐　宗徳†
Kazuma Takechi　Sumiya Tohyama　Hiromasa Kobayashi　Munenori Kai

1　はじめに

近年のマルチコア化により並列コンピューティングの必要性が高まっている。しかし、並列コンピューティングでは複数のプロセッサやコアを効率よく協調させなければ高い計算処理性能を実現することができず、逆に逐次処理に比べて処理性能を低下させてしまう可能性がある。そのため並列コンピューティングで高い処理性能を得るには逐次プログラミングの知識に加えて、並列プログラミングの知識が必要となり、開発者にとって負担が大きくなると考えられる。

このような問題を解決するための手段として逐次プログラムを自動で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利用が望まれる。逐次プログラムを自動で並列化することができれば既存の逐次プログラムを手軽に利用できると考えられる。

本研究では多数ある並列プログラミングのライブラリの中でも、多くのプラットフォーム、共有メモリと分散メモリの両方に対応した、MPI(Message Passing Interface)[1]を利用して、コード生成を行う。分散メモリ環境で実行できるコードを出力することで、より大規模な並列環境で利用できるようにしている。また並列プログラムのコードを出力することで、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になる。また並列化を行う際に構文木をベースとした中間データ構造を使うことにより並列化処理の手順の見直しや新しい並列化処理を加えるなどの柔軟な拡張を行えるようにしている。さらに、並列実行時間を短縮するには抽出した並列タスクの実行時間を基にしたタスクスケジューリングが重要であると考えられるが並列タスクの実行時間を静的に定めることは非常に困難なため、今回は実行時に並列タスクのプロセッサ(またはコア)への割当てを行う動的実行制御を実装した。

2　C 言語自動並列化トランスレータの概要

本研究での C 言語自動並列化トランスレータは中間データ構造を構築しこれに対して解析を行い、並列プログラムを出力することで並列化を行う。

中間データ構造の構築を行う前にまず C 言語で正しく記述された逐次プログラムに対してプリプロセス展開を行いライブラリやマクロの展開を行う。次に構文解析を行い、図 1 のような構文木を作成する。この構文木に対してデータ依存解析を行うことで図 2 で示すような中間データ構造を生成する。点線で囲まれているのが元の構文木であり、実線で囲まれているのが作成される中間データ構造である。この中間データ構造の各ノードは本来の構文木構造としての意味に加えて並列化する際に使う情報を持っている。

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

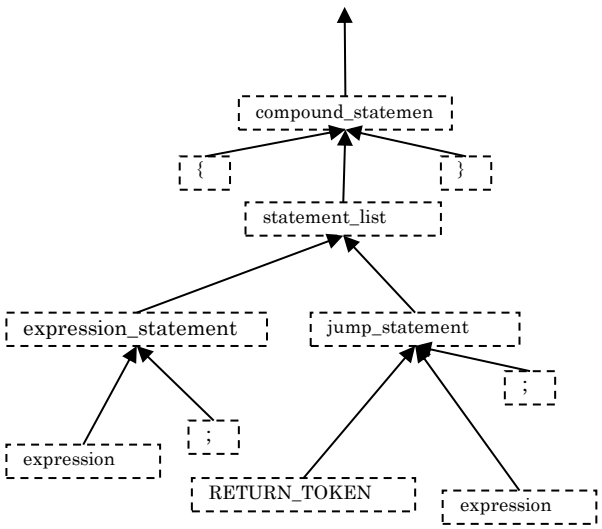

図 1　通常の構文木

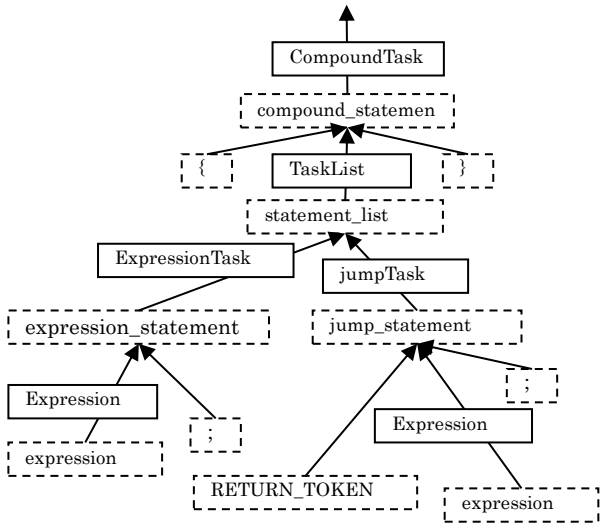

図 2　中間データ構造

この中間データ構造のタスクに対してデータ依存関係の解析を行い並列化可能である箇所を解析する。データ依存関係の解析が終了した時点ではタスクはステートメント単位であるのでこのままタスクスケジューリングを行うとタスク数が多く、組み合わせが膨大となってしまう。そのため複数のタスクを一つのタスクと見なすタスク融合によりタスク数を減少させる。融合したタスクのデータ依存関係を使うことで中間データ構造は同時にタスクグラフとしても使えるようになる。また、データ依存関係を解析するだけでなくループリストラクチャリングを行い新たな並列性を抽出することも行う。

各種並列性解析を通じて並列化情報を付加し、最終的に得られた中間データ構造から抽出される並列タスクを最小並列処理時間で完了するためには、タスクスケジューリ

Copyright © 2013 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

ングが必要となる。しかし、それには各並列タスクの処理時間やデータ依存関係にあるデータ通信にかかる時間を出来る限り正確に知る必要があるが、現段階ではそれは容易ではない。そこで、その代わりに各並列タスクの処理時間と必要な通信時間について適当なコストを用いることにした。そのコストをもとに厳密なスタティックタスクスケジューリングを行なっても実際の処理時間とはずれが生じるため、今回は動的な実行制御を行う。並列タスクのコストを基に初期分散を決めておき、並列実行の進行に従って並列タスクの再割当てを動的に決めていく方法である。

3　中間データ構造を使った並列化処理

前章で述べたように並列化のための各種解析は中間データ構造に対して行われ、中間データ構造に解析結果を追加していく。ここではその詳細について述べる。

3.1　データ依存解析

データ依存解析ではステートメントごとにデータ依存関係の解析を行う。

3.1.1 変数のアクセス属性

プログラム内での変数には read、write、read-write と言う 3 つのアクセス属性が存在している。read は変数の値を変更することが無く、参照するだけである。write は変数に代入されている値を上書きし、read-write はそれら両方を行うアクセス属性である。これらは図 2 のタスクを表すノード内を解析していき、式の演算子からオペランドへのアクセスを調べ中間データ構造内で変数として解析されたノードに対してアクセス属性の情報を付与する。

3.1.2 タスク間のデータ依存解析

タスク内の変数のアクセス属性を使用しタスク間のデータ依存関係を解析する。これを解析することでタスク間の並列性を抽出できる。タスク間のデータ依存関係にはフロー依存、出力依存、逆依存の三種類がある。フロー依存は最後に変数に対し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場合にそれぞれのタスク同士で生じるデータ依存関係である。出力依存と逆依存は処理を実行する際に順番を入れ替えることが出来ないために生じるデータ依存関係である。出力依存は先行ステートメントで write されている変数が後続ステートメントで write されている場合に発生する。逆依存は先行ステートメントで read されている変数が後続ステートメントで write されている場合に発生する。

解析結果を図 2 の各タスクを表すノードに対して先行タスクと後続タスクの情報を付与することでデータ依存関係を表現出来る。これらの情報をつなぎ合わせることで依存情報をエッジとしたタスクグラフと見なすこともできる。

3.2　ポインタ解析

プログラム中にポインタがある場合、3.1.2 の方法だけでは解析が行えない。何故ならポインタ変数が他の変数等、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データ依存解析だけではデータ依存関係を見抜けないからである。そのためポインタ解析を行わない並列性解析の仕様ではポインタが使われるプログラムに対しては安全のため並列化処理を行うことが出来ない。ポインタ解析を行うことでポインタ変数の参照先の情報を解析し中間データ構造のポインタに保存していくことでプログラム上に現れないデータ依存関係を解析することが可能になる。

3.3　タスク融合

データ依存関係の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であるため、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[2]。このままでは、ソースコードのステートメント数の増加に対して、組み合わせが膨大になりタスクスケジューリングにかかる時間が指数関数的に増大していくという問題がある。

そこで本研究ではプロセス数に基づく最大並列度を考慮するためにステートメントに対して変数の個数をコストとして考慮した。このコストを使用してタスク融合を行う。また、タスクグラフにおける同一エッジ上のタスクについては依存強度が強いものに対してタスク融合を行う。以上の 2 つのタスク融合を組み合わせた手法によりタスク数を減少させタスクスケジューリングにかかる時間の短縮化を行う。融合したタスクは図 2 の中間データ構造の CompoundTask(複数のタスクの集まり)として一つにまとめる。

3.4　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内で行われている何らかの処理によって消費されている。従って、C 言語自動並列化トランスレータの性能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。

そこで本研究では一見並列性が見えないループに対し、ループリストラクチャリングを適用し並列性を高める変換を行う。具体的には 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、スカラエクスパンションやループディストリビューションを行う。これらの変換は中間データ構造の ForTask ノードの分割やその上位のタスクノードへの並列情報の付加により対応する。

4　実行制御

プログラムを並列実行するためには融合されたタスクを各プロセスに割当てる必要がある。本研究におけるプロセスへの割当ては大きく分けて 2 つある。初期分散と実行時再割当てである。初期分散は元のプログラムを静的に解析し求めたタスクのコストやデータ依存関係から割当てを行う。しかし、静的な解析ではタスクの正確な実行時間を解析することは困難であるため高速化が望めない場合がある。そこで本研究ではプログラムの実行時にタスクをプロセスに再割当てする動的実行制御を行うことにした。

4.1　タスクの関数化

動的実行制御を行うためには複数のプロセスそれぞれに任意のタスクを実行させる必要がある。そのために本研究では中間データ構造の各タスクを関数の形にして切り出す。

関数に切り出したタスクを実行させる際にはデータ依存関係のある変数の値を必要とするため、各関数は必要な変数を受け取る受信、元のタスクの処理内容、書き換えた変数に依存する後続タスクへのデータ送信の 3 つの処理から構成される。

切り出された関数を関数ポインタの配列に保存し図 3 のような関数テーブルを構築する。全プロセスは同一の関数テーブルを持ち、保存された関数を順次実行する。これにより各プロセスに関数テーブルのインデックスを渡すことで任意のタスクを実行させることが可能になる。

Copyright © 2013 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

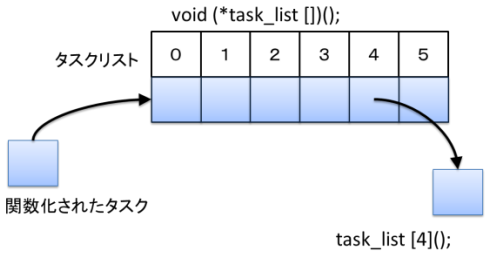

図 3　関数テーブル

4.2　タスクの初期分散

プログラムをより高速に実行させるためには適切にタスクを並列実行させる必要がある。他のプロセスに任意のタスク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ので、タスクを実行前の段階で分散させることで次に行うタスクを決めておく。ここではタスクのデータ依存関係を用いて初期分散を行う。

4.3　タスク管理

各プロセスが割当てられたタスクを実行する際には必要な変数を受け取る必要があり、データ依存するタスクが終了していなくてはならない。データ依存関係を満たしたタスクが実行中であり終了を待機している場合、他の依存しないタスクを先に実行することで待機時間を減らし稼働率を上げることで実行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のタスクに割当てられたプロセスを表 1 のようなタスクテーブルとして管理する。タスクテーブルには各タスクが依存する変数をどの ID のタスクを実行したプロセスから受け取る必要があるかを保管している。そして、実行時に再割当てが行われた際に、タスクに再割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、完了のうち、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かなどの情報を補完する。依存する変数が存在しない場合は初期状態で実行可能となる。

表 1 タスクテーブルの例

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	

4.4　通信命令の生成

並列プログラムを実行する為には MPI で記述された通信を行う並列プログラムを出力する必要がある。通信用のコードは関数化されたタスクに対し、それぞれ作成する。中間データ構造に保存されたデータ依存関係は受信タスク、送信タスク、通信情報の 3 つの情報を持つためそれぞれのデータ依存関係に対して個別の ID を割当てる。この ID をインデックスとして、タスク間の通信情報が含まれているデータ送受信表を作成する。

4.5　通信の最適化

逐次プログラムの自動並列化を行う際に自動抽出されたタスクがプロセスに割当てられる。各プロセスはそのタスクを実行するために必要な情報の通信を行わなければならない。この通信は MPI を用いて行う。通信のオーバヘッドを削減するために同一宛先への複数のデータを送る場合は一回の通信で行うようにした。情報の受け渡しは 1 対 1 プロセス間通信だけではなく同一データを複数のプロセスに通信する場合もある。実行時間の短縮を図るためにタスクテーブルとデータ送受信表を用いて 1 対 1 通信とブロードキャストの使い分けを行う。通信命令の使い分け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の実行時間の短縮につながると考えられる。

4.6　実行時再割当て

実行時にタスクの再割当てを行う為には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い。その問題を解決するために図 4 のようにプロセスの 1 つを各プロセスの実行状況を管理するための管理プロセスとする、その他のプロセスはタスク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基に他のプロセスに対してタスクの再割当てとその通知を行う。

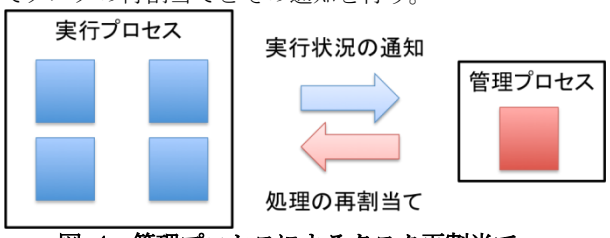

図 4　管理プロセスによるタスク再割当て

5　並列実行可能なプログラムの出力

中間データ構造に加えられた各種並列性解析の結果に基づき、並列実行可能な並列プログラムを出力する必要がある。出力される並列プログラムと中間データ構造を分離することにより一つの中間データ構造から複数のプログラムを出力することができる。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できる。

以下で実際に C 言語自動並列化トランスレータを使った時の結果を示す。

5.1　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;
}
```

図 5　入力プログラム例

図 5 がサンプルとして入力される逐次プログラムである。この逐次プログラムでは 2 つの変数 a,b 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。この例ではタスクが少なく並列効果は見込めないが、並列実行が正

Copyright © 2013 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

確に行われていることを簡潔に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である。しかし、合計を出力する printf(“%d¥n”,a+b)は a=my_work(1)と b=my_work(2)の終了を待機し通信を行ってから実行する必要がある。

```
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);
}
```

図 6　タスクリスト

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

図 7　通信用のデータ構造

図 5 の逐次プログラムから並列プログラムを出力するとそれぞれのタスクが関数化された図 6 のタスクリストが出力される。この関数化されたタスクでは使用される局所変数が図 7 のような別の構造体となっている。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構造体となっている。

図 6 のタスクリストを実行する部分を図 8 で示す。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。

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

図 8　タスクの実行

6　今後の課題とまとめ

本研究では、C 言語の逐次プログラムから構文木ベースの中間データ構造を構築し、それに対して各種の並列性解析結果を付加して最終的に得られた中間データ構造から動的実行制御を可能とする並列プログラムを出力することが出来た。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、今後の課題はより高速に実行できる並列プログラムを出力することである。そのためには各種最適化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為にはタスクにかかる正確な実行時間の算出やタスクを融合し粒度を大きくすることでより効率よく実行できるようにすることが必要になる。

参考文献

[1] P.パチェコ著(秋葉 博 訳) :「MPI 並列プログラミング」, 培風館, 2001.

[2] 小林裕晶・遠山純也・甲斐宗徳 :「C 言語自動並列化トランスレータの開発-タスク粒度解析手法の試作とその評価」, FIT2012, 第 1 分冊, pp.183-186, 2012.

Copyright © 2013 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.