B-012

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とその評価－

Development of Automatic Translator from C Programs to Parallel Programs Using MPI
- Evaluation of a Task Grain Analysis Method Using Static Metrics of Source Code -

阿加井 星† 小林 裕昌† 甲斐 宗徳†
Sei Akai Hiromasa Kobayashi 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。逐次コードを並列コードに変換する際の難しさのひとつに、並列実行するタスクの粒度の設定がある。実行前にタスクコストの見積もり精度が悪い状態でタスクスケジューリングを行っても、スケジューリング結果通りの並列効果を得にくくなる。そこで本稿ではソースコード静的メトリクスを利用してタスクコストを見積もり、それにより実際の並列処理結果に近い精度でタスクスケジューリングを可能とするようなタスク粒度解析を試行した結果について述べる。

2. Ｃ言語自動並列化トランスレータ

筆者らが開発中の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 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、内部で簡易的な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り、詳しい説明は３章で行う。

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

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。

図 1 は並列化トランスレータの処理手順を表わしている。

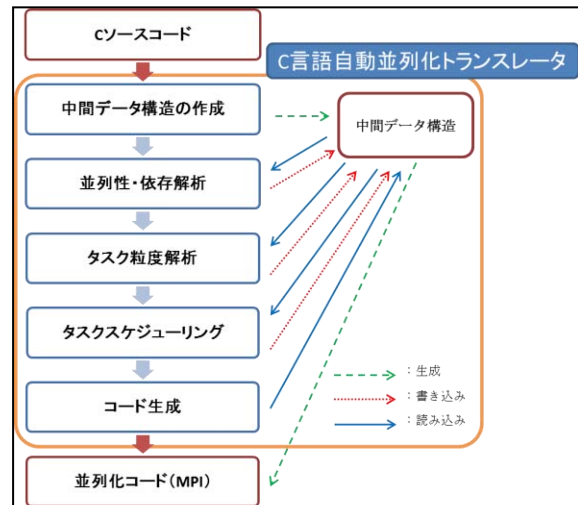

図 1 並列化トランスレータ処理手順

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。

3. タスク粒度解析

並列性・依存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。

本研究では、ステートメント内のメモリアクセス命令、通信にかかるオーバーヘッドなどの各タスクコストを考慮し、より詳細な実行時間を見積もることが可能なメトリクスを提案する。

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

3.1 タスクの種類と解析方法について

本トランスレータでは、タスクの最小単位はステートメントレベルである。また、それらのタスクの組み合わせを if や for といった専用の構造に当てはめることによってタスク群を構成している。この条件節やループ文内の、条件文や制御文といったタスクをまとめたタスク群を Control タスクと定義する。関数や構造体といったタスクも同じく、仮引数、ローカル変数といったすべての変数とブロックスコープで囲まれたタスク群の組み合わせといったもので構成される。タスク群で一つにまとまるタスクは前述のとおり Macro タスクと呼ぶ。以下では初期に識別されるタスクについて述べる。

3.1.1 Expression タスク

このタスクは本トランスレータにおけるタスクの最小単位であり、基本的な式を表現している。代入文や、制御フロー文といったものがこの Expression タスクとして扱われる。

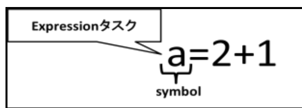

図 2　Expression タスクの一例

3.1.2 Compound タスク

このタスクはブロックスコープ内に宣言されているタスク群をまとめて一つのタスクとしたものである。後述する if タスクや for タスクといったマクロタスクは、この Compound タスクと Control タスクの組み合わせにより構成されている。

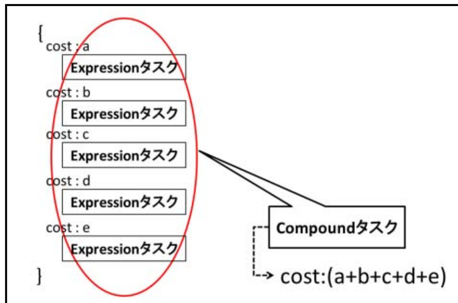

図 3　Compound タスクの一例

3.1.3 if タスク, if-else タスク, switch タスク

これらのタスクは、Control タスクと Compound タスクを組み合わせた条件判定を行う制御フロー文である。これらのコストは、Control タスクのコストと、最も実行コストの大きい Compound タスクの総和によって決定される。

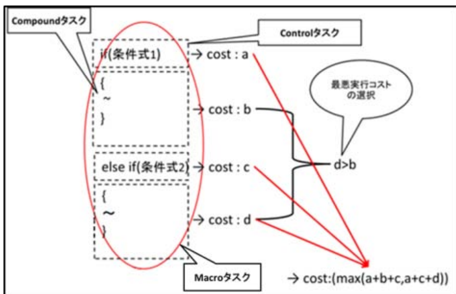

図 4　if-else タスクの一例

3.1.4 for タスク, while タスク

これらのタスクは、Control タスクと Compound タスクを組み合わせた反復制御を行うタスク群である。これらのタスクのコストは、Control タスクのコストと Compound タスクの総和である。このとき、反復回数が判明している場合は、Compound タスクのコストに反復回数を乗じた値を反復全体のコストとする。図 5 では、for タスクの一例を示している。

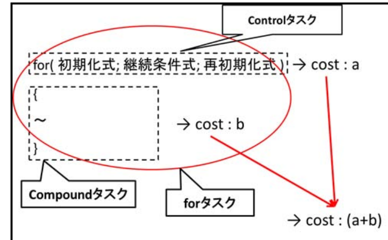

図 5　for タスクの一例

3.1.5 function タスク

これらのタスクは、ユーザ定義関数や main 関数が構成するタスクである。通常のタスクと異なり、Compound タスクと引数、変数宣言といった symbol リストを持つ。function のタスクのコストはその中に含まれる Compound タスクコストと等しいものと定義する。また、function タスクは Compound タスクが持つタスク群の依存関係を解析し、生成されたタスクグラフを持つ。図 6 は、基本的な function タスクの例を示している。

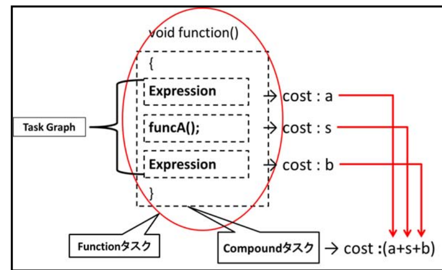

図 6　function タスクの一例

3.2 静的メトリクスを用いたタスクのコスト

一般的に細粒度であるソースコードの実行時間を正確に見積もることは難しい。そこで、本研究では以下にあげるような各タスクの持つ要素に加え、3.1 節で述べたような各タスクの種類についてタスクコストを考えたとき、考慮しなければならない要素を応用メトリクスと定義して解析を行う。以下は、どのタスクでも適用される基本メトリクスを示す。

- LOC(lines of code)
  タスクの持つアセンブリコードにおけるステップ数
- NOV(number of variables)
  タスクの持つ変数の数を計測するメトリクス

これは変数の個数を表す以下の二つのメトリクスの合計となる。

- NTV(number of task variables)
  タスクの持つクラス指定子が付いている変数の数
- NIV(number of instance variables)
  タスクの持つクラス指定子が付いていない変数の数

次に Compound タスクに適用される応用メトリクスを示す。

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

これは、Chidamber 氏と Kemerer 氏が提案したソフトウェアメトリクスである CK メトリクス[3]を元に、タスクの実行時間に関わる複雑度を算出するメトリクスである。

- CBF(coupling between functions)

対象のタスクと依存関係のあるタスクの数

CBF が高いほど、他のタスクに依存していることを示し、複雑でコストがかかることを示唆している。

- WMT(weighted methods per task)

タスクの複雑さの総和

WMT が高いほど複雑なタスクとなり、コストが高いことを示唆している。複雑さの総和とは、McCabe のサイクロマチック数という循環的複雑度を示している。

循環的複雑度とは、プログラム中の分岐/合流点をノード、その他の部分をエッジとしたとき、以下で表される。

ノードの数(v)、エッジの数(e)から、e - v + 2

- RFT(response for a task)

タスク内で呼び出される Compound タスクの回数

RFT が大きいほど、呼び出す Compound タスクの回数が多いことを示し、 複雑でコストがかかることを示唆している。

3.3 各メトリクスを使用したコストの算出方法

前述までに紹介したメトリクスをそのままの数値で使用することは難しく、各メトリクスを使用したコストの算出を行う必要がある。ここでは、その算出方法を示す。以下の式では n はタスク ID、N を全体のタスク ID とする。

○基本メトリクス

◆LOC を利用したコストの算出方法

タスクグラフ内にあるタスクの LOC を合計し、対象となるタスクの割合を算出する。これは、アセンブリコード全体から見た各タスクの持つコードの割合を示している。

$$LOCCost(n) = \frac{100 \times Loc(n)}{\sum_{k=0}^{N} Loc(k)} \cdots (1)$$

◆NOV を利用したコストの算出方法

LOC 内における NOV の割合を算出する。これは、各タスクの持つ命令群の中のメモリアクセス命令の割合を示している。

$$NOVCost(n) = \frac{100 \times Nov(n)}{Loc(n)} \cdots (2)$$

○応用メトリクス

◆CBF,WMT,RFT を利用したコストの算出方法

タスクグラフ内における各タスクの割合を算出する。

$$WMTCost(n) = \frac{100 \times Wmt(n)}{\sum_{k=0}^{N} Wmt(k)} \cdots (3)$$

$$CBFCost(n) = \frac{100 \times Cbf(n)}{\sum_{k=0}^{N} Cbf(k)} \cdots (4)$$

$$RFTCost(n) = \frac{100 \times Rft(n)}{\sum_{k=0}^{N} Rft(k)} \cdots (5)$$

また、タスクコストは以上の式の総和となる。

TaskCost=(1)+(2)+(3)+(4)+(5)

この重み付けは、LOC や NOV などの物理的な情報と、依存関係などの理論的な情報から算出されたコストの価値を同等にし、タスクの相対的なコストを算出することを狙っている。

4. 性能評価

4.1 実験概要

実験 1 では、スケジューリング長の減少率を比較し、実験 2 では、実験 1 で得られたスケジュール長の減少率と並列実行した際の減少率を比較し、提案手法の有効性の検証を行う。実験環境と使用するベンチマークは以下である。

・実験環境

OS:CentOS6.5

CPU:intel(R) Xeon(R) CPU E5-4640 8-CORE @2.40GH×4

実装メモリ:256GB

・使用したベンチマーク

①MiBench Version1.0:basicmath_large.c

②MiBench Version1.0:susan.c

③姫野ベンチマーク(dynamic allocate version)

④NAS Parallel Benchmarks:IS (size’ S’)

また、本トランスレータでは早稲田大学笠原氏による DF/IHS (Depth First / Implicit Heuristic Search) [4]を元に、通信を考慮した組合せ探索ができるように、当研究室で拡張した並列分枝限定法による通信遅延を考慮したタスクスケジューリングアルゴリズム[5]を使用している。

4.1.1 実験 1:スケジューリング結果の比較

本提案手法でコストを設定した際のスケジュール長と、実際に計測した実行時間のそれぞれにおいて逐次処理に対して並列処理がどれだけ処理時間を削減できたか、その削減率を比較する。なお、ここでは各ベンチマークの最大並列度を考慮しプロセッサ台数を 4 台とし、探索完了には膨大な時間がかかる可能性があるため探索時間を 10 秒に制限したときの最良解を求めるスケジューリングを行った。

表 1 スケジューリング結果における減少率の比較

		逐次実行	並列実行	削減率(%)
basicmath	スケジュール長	4,582	3,214	29.9%
	実時間 (RDTSC)	866,848,263	669,126,938	22.8%
himeno	スケジュール長	4,347	2,134	50.9%
	実時間 (RDTSC)	1,282,034,208	1,282,006,033	0.00220%
susan	スケジュール長	265	233	12.1%
susan (s)	実時間 (RDTSC)	359,067,889	358,045,623	0.285%
susan (e)	実時間 (RDTSC)	81,608,665	80,457,236	1.41%
susan (c)	実時間 (RDTSC)	40,029,505	38,907,192	2.80%
NasBench (IS)	スケジュール長	55,984,147	55,876,623	0.192%
	実時間 (RDTSC)	1,785,007	1,784,952	0.00308%

4.1.2 実験 2:実行時間の減少率の比較

逐次実行の実行時間と、生成された並列プログラムの実行時間を比較し削減率を算出する。スケジューリングを行った際と同じ設定であるプロセッサ数 4 で並列実行を行った。

表 2 実行時間における削減率の比較

	逐次実行(秒)	並列実行(秒)	削減率(%)
basicmath	74.549	52.776	29.2%
himeno	0.165	0.162	2.17%
susan (s)	0.036	0.041	-13.7%
susan (e)	0.018	0.019	-6.75%
susan (c)	50.817	50.344	0.931%
NasBench (IS)	0.029	0.030	-2.31%

表 2 で減少率がマイナスとなっている個所は実行時間が増加していることを表している。

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

4.2 考察

4.2.1 スケジューリング結果の比較における考察

実験 1 では、提案手法から求められたコストを利用したスケジューリング結果と、実際に実時間を測定し、そこから得られた値をタスクコストとして割り当てた際のスケジューリング結果の比較を行った。

■ basicmath ベンチマークについて

回数不明のループも少なく、どのマシン環境で動作させても計算内容が同一のものになるという特徴から、誤差は最も少ないという結果になった。

■ susan ベンチマークについて

入力される画像、オプションによって演算の内容が変化するという特徴から、ソースコードの内容だけで実行時間の判別を行う本提案手法はあまり有効でないと考えられる。

■ 姫野ベンチマークについて

4.1.1 の実験の中で最も誤差が大きい結果となっている。原因は、マシンのメモリバンド幅などの性能によって演算の内容が変化するベンチマークという特徴である点、計算を行う関数に渡す引数によって繰り返し回数が変化するというプログラムの性質上、本提案手法に対する相性が非常に悪いという点が上げられる。

■ NAS Parallel Benchmarks について

本提案手法の性質上最も複雑で大規模な構造をしているループ文に対してコストを重く設定してしまうという点が原因で誤差が発生していると考えられる。実際に動作させた結果、条件分岐が多く最も大規模なループ文よりも、すべての値を一つずつ比較する簡潔なループ文がボトルネックとなっていることが判明した。

✧ まとめ

以上の結果から、提案手法が大規模かつ外部の要因によって演算の内容が変化するタスクになるほど精度が低くなる。これは、本提案手法では演算の内容が演算を行う関数に渡す引数によって変化する場合、そのタスクコストは引数に関わらず一定の値として計算してしまうことが原因である。

4.2.2 実行時間の減少率の比較における考察

実験 2 では、実際に対象のベンチマークを逐次で実行した場合と、本トランスレータによって生成された並列化コードの実行時間を比較し減少率を算出した。

■ basicmath ベンチマークについて

実験 1 の結果と同じくスケジューリング結果の減少率と実際のプログラムの減少率との誤差が最も少ない。以上のことから、提案手法でスケジューリングを行い、その結果を元に生成した並列化コードがほぼ狙った通りの並列実行であることを示し、これは提案手法により高い精度で実行時間を概算できたことを示していると考えられる。

■ susan ベンチマークについて

実験 1 にも記したとおり、本提案手法による実行時間解析では、実際の処理時間を概算することは難しく、それに伴ってスケジューリング結果から得られた減少率と実際に実行を行った際の減少率には誤差が生じている。これは、本提案手法によるタスクの重み付けがうまくいかず、タスクスケジューリング部分において適切な並列性を生かすことができなかったからと考えられる。

■ 姫野ベンチマークについて

実際に並列実行を行った際に僅かながら実行時間の減少に成功した。内部では、計算を行う関数がボトルネックとなっており、そのタスク以外はすべて細粒度のタスクである。したがって、計算を行うタスクを処理するプロセッサが実行時間の大部分を占めており、その他のプロセッサはほとんどアイドル状態である。また、ボトルネックとなる関数は上記でも示した通り、引数によって繰り返し回数が変わり、演算を行う処理時間が変動する。このようなプログラム場合、susan と同じく提案手法では実行時間を推測することが困難であり、スケジューリング結果から得られた減少率と、実際に並列実行した際の減少率に誤差が生じている。

■ NAS Parallel Benchmarks について

このベンチマークに関しても、susan ベンチマークと同じく並列実行を行うことにより実行時間が増加してしまっている。これも susan ベンチマークと同じく適切なタスクコストの見積もりが出来なかったことが原因の一つと考えられる。また、ボトルネックとなっている粗粒度タスクと細粒度タスクとの差が非常に大きく無駄な並列性が多くあることが実行時間の増加に繋がっていると考えられる。これに関しては、タスク粒度最適化部分であるタスク融合が有効であると考えられる。

✧ まとめ

以上の結果から、本トランスレータで想定していた特徴と一致する basicmath ベンチマークに関しては提案手法が有効であると言える。しかし、入力される値によって演算の回数が変わるといった性質のプログラムに対しては提案手法では不十分であることが考えられる。そのような、動的に演算の処理量が変化するタスクに対しては依然として対策が必要であると考えられる。

5. まとめ

本研究では、タスクコスト算出のためのメトリクスを提案し、タスク粒度解析における実行時間解析の拡張を行った。

本提案手法における今後の課題は以下の二つが上げられる。一つは、ここで示したコストメトリクスにプライオリティを設定し重みづけをすることにより、より詳細な見積もりを行える可能性がある。二つ目は、このメトリクスを導入したことによってタスク粒度最適化の結果、タスクスケジューリングの求解時間、出力されたプログラムの並列実行した際の実行時間に対してどのような影響を与えているかをより詳しく実験し、メトリクスの改良、選択を行う必要がある。

＜参考文献＞

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

[2]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.

[3] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object-oriented design”. IEEE Trans. on Software Engineering, 20(6):476–493, (1994).

[4] H. Kasahara, S. Narita, “Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing”, IEEE Trans. on Computers,Vol. C-33, No. 11, pp. 1023–1029, Nov. (1984).

[5] 栗田 浩一, 宇都宮 雅彦, 塩田 隆二, 甲斐 宗徳:「通信を考慮したタスクスケジューリング問題の効率的な並列探索解法の提案」, FIT2011（第 10 回情報科学技術フォーラム）,第 1 分冊 RA-006, pp.37-42, (2011).

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.