成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.51 No.1 (2014)pp.27-34

C言語プログラムのための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

小林　裕昌[*1]，甲斐　宗徳[*2]

Task Granularity Analysis Method Using Static Metrics of Source Code for C Programs

Hiromasa Kobayashi[*1], Munenori Kai[*2]

ABSTRACT : In the first phase of our automatic parallelizing translator for C program, a source code is decomposed into a set of tasks of the granularity of a statement level at the minimum. In the next phase, task scheduling which determines statically by which processor these tasks are processed is performed. Since this task scheduling is a combinatorial optimization problem, it is important for it to suppress the number of tasks which constitutes the program. Therefore, useless parallelism is removed using the information about the dependencies among tasks and task cost, and the task granularity analysis is required in order to make task granularity reasonable. However, since it is necessary to analyze the processing time of a task only using the information acquired from a source code, exact cost may be unable to be assigned in the time analysis using the conventional computational complexity analysis. So, in this paper, the method of aiming at the improvement in accuracy of execution time analysis is proposed by applying static metrics of source code.

Keywords : automatic parallelizing translator, task granularity, task scheduling, execution time analysis, static metrics of source code

(Received March 31, 2014)

1．はじめに

技術分野における驚異的な成長の背景には，電子計算機，すなわちコンピュータの存在が大きく関わっており，とりわけCPU(Central Processing Unit)の進化が多大な影響を及ぼしていることは間違いない。

CPUの性能を向上させるための方法としては，動作周波数を上げることが代表的ではあるが，ここ数年で周波数を上げることは限界に近付いている。そこで性能向上に対する解決策として，複数のプロセッサを用いて並列動作させるという考えが主流となってきている。そのため，プログラムを効率よく並列処理させるためには，並列プログラミングを用いてプログラムを作成することが必須であると言える。

しかし，並列プログラミングを用いても，効率的なソースコードを作成できなければ，処理速度を向上させることは難しく，そのためには通常のプログラミングの知識の他に，専門的な知識が必要となり，非常に難易度の高い技術であることが伺える。

以上のようなことから，並列プログラムの必要性が高まってきているにも関わらず，効率の良い並列プログラムの作成にはプログラマに大きな負担がかかるという問題が発生していると言える。この問題点を本研究の背景とした。

2．C言語自動並列化トランスレータ

C言語自動並列化トランスレータ[1]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

*1：理工学研究科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)

ログラム内に存在する並列性を抽出し，MPI (Message Passing Interface)による並列実行用コードを埋め込むことで並列化コードを出力する。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。

本研究で開発している並列化トランスレータでは，まだ実用的な自動並列化が存在していないC言語に対して並列化を行い，MPI が挿入された並列プログラムを生成することで，より様々な並列実行可能環境で利用できるようにしている。また並列化ソースコードを出力することで，開発段階では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。

以上のような特徴を持ち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の完成を最終目的とする。

2．1　トランスレータ処理手順

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。

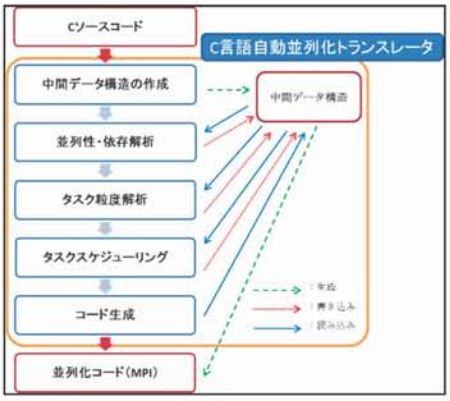

図 1　並列化トランスレータ処理手順

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]。

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。現段階の本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。

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。

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，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。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，ブロックスコープと制御フロー文を持つifタスク，forタスクや，ユーザ定義関数，main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 はタスクグラフの一例を示す。

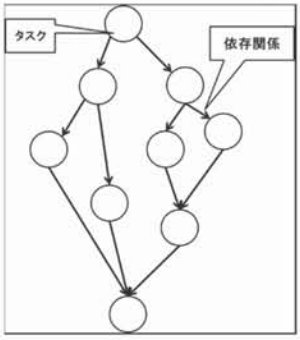

図 2　タスクグラフ の一例

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であるため，タスクの数はソースコード内のステートメント数に相当する。すなわち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，多くのタスクは細粒度のタスクであるため，タスク数も大きくなる。一般的に，並列処理を行うことにより発生する通信のオーバーヘッドは，処理時間に大きな影響を及ぼし，細粒度タスクに関しては逐次処理を行った方が良い場合が多い。それどころか，タスクスケジューリングが組み合わせ最適化問題であるため，探索解法の過程において大量の組合せが発生し，求解時間が指数関数的に増大する。このため，ステートメント数が多く存在する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言える。

前年度では，タスク粒度の調整に用いる個々のタスクコストを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を依存強度（通信によって渡さなければならない変数の個数）という代替案を提案し，プロセッサ数に基づく最大並列度を考慮したタスク融合というアプローチによって粒度を求めた[3]。

本研究では，タスクの持つ様々な要素を考慮し，より

詳細なタスクの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。

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，タスクの最小単位をステートメントレベルとしている。また，ステートメントレベルのタスクを条件文や制御文とみなし，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制御フロー構文を構成している。この，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ており，それらを前述のとおりMacroタスクと呼ぶ。

ここではステートメントレベルのタスク，ブロックスコープで囲まれたタスク群，基本的な種類のMacroタスクの解析方法を示す。

3．1．1　Expressionタスク

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。これらタスクの情報は木構造によって保管されている。図 3 では，代入文であるExpressionタスクの一例を示している。

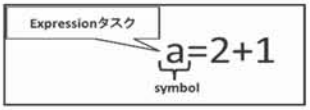

図 3　Expressionタスクの一例

3．1．2　Compoundタスク

本トランスレータでは，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て扱う。ifタスクやforタスクといったマクロタスクは，Compoundタスクと，制御フロー文であるExpressionタスクの組み合わせにより構成されている。また，ユーザ定義関数やmain関数といったfunctionタスクもCompoundタスクと，仮引数，変数宣言されたsymbolで構成されている。図 4 は，Compoundタスクの一例を示している。

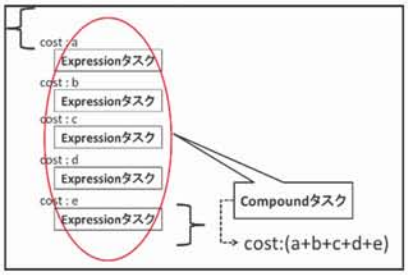

図 4　Compoundタスクの一例

3．1．3　ifタスク,if-elseタスク,switchタスク

これらのタスクは，コントロールタスクと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。これらのタスクのコストは，コントロールタスクのコストと，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 5 では，基本的なif-elseタスクの例を示している。

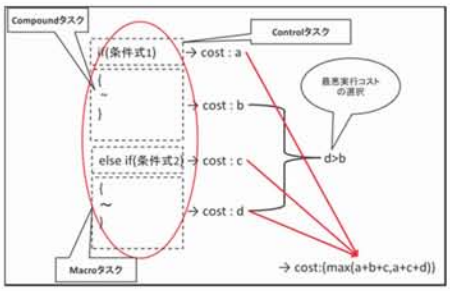

図 5　if-elseタスクの一例

3．1．4　forタスク,whileタスク

これらのタスクは，コントロールタスクとCompoundタスクを組み合わせた反復制御を行う制御フロー文である。これらのタスクのコストは，コントロールタスクのコストとCompoundタスクの総和である。また，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算し，判明していないものは乗算を行わない。図 6 では，forタスクの一例を示している。

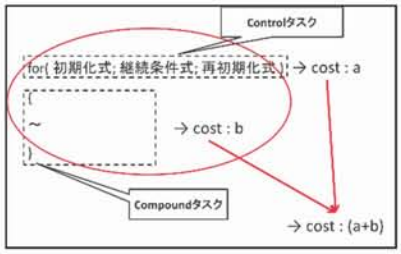

図 6　forタスクの一例

3．1．5　functionタスク

これらのタスクは，ユーザ定義関数，main関数といっ

た関数タスクである。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。これらのタスクのコストはfunctionタスクの持つCompoundタスクの一回分のイタレーションの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7は，基本的なfunctionタスクの例を示している。

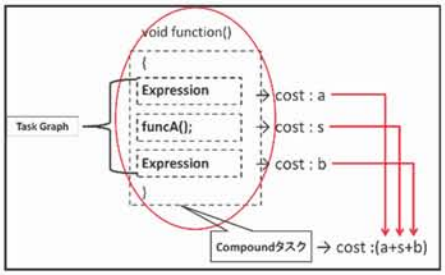

図5　functionタスクの一例

3．2　静的メトリクスを用いたタスクの実行時間解析

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，本研究ではタスクの持つ要素に加え，タスクの属性を考慮し，実行時間解析を行う。そのために必要な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。

○基本メトリクス

- 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
  これは前年度のトランスレータに採用されていた実行時間解析と同義であり，変数の個数を表す二つのメトリクスの合計となる。二つのメトリクスの詳しい内容は次に説明を行う。
- 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

次に示すのは，ifタスクやfunctionタスクといったCompoundタスクに適用されるメトリクスを示す。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[4]を元に，タスクの実行時間に関わると考えられる複雑度を算出するメトリクスである。

○応用メトリクス

- 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、
    ノードの数(v)、エッジの数(e)から、e - v + 2
    という式で求めた値となる。
- 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
  これらの応用メトリクスは，LOCと同じく，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)

これらの割合といった重み付けは，LOCやNOVといったソースコードの物理的な情報と，依存関係といった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

本章では，前述の解析手法を並列化トランスレータに実装し，求められたコストと，実際のタスクの実行時間との相対的な比較を行うため，以下の実験を行った。なお，紙面の都合上，本実験ではタスク粒度解析における実行時間解析のみを適用した場合を示しており，タスク粒度最適化は行っていない

4．1　実験概要

本提案手法が実装された並列化トランスレータが生成したスケジュール長，並列プログラムを用いて以下の二つの実験を行う。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の通りである。

・実験環境

マシン仕様

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

1　MiBench Version1.0:basicmath_large.c

2　MiBench Version1.0:susan.c

3　姫野ベンチマーク(dynamic allocate version)

4　NAS 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）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スケジューリングアルゴリズムを使用している。

時間計測関数は，intel(R) Xeon(R) CPU E5-4640 に組み込まれている高精度タイマ「Invariant TSC」をシリアライズすることなく呼び出せるRDTSCP命令を使用した。

4．1．1　実験 1:スケジューリング結果の比較

この実験では，スケジューリングを行った結果を用いて相対的な比較を行う。比較対象は，本提案手法でコストを設定した際のスケジューリング結果における減少率と，実際に計測した実行時間をコストとして設定した際のスケジューリング結果の減少率である。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，探索時間は 10 秒で設定しスケジューリングを行った。これは，最適解が 10 日以上の探索を行っても得られなかったことに加え，おおよそ 10 秒ほどで大部分の分枝限定法による枝切りが行われていたことによるものである。

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	減少率(%)
basicmath	提案コストによるスケジュール長	4,582	3,214	29.800
	実時間（RDTSC）	866,848,263	669,126,938	22.800
himeno	提案コストによるスケジュール長	4,347	2,134	50.909
	実時間（RDTSC）	1,282,034,208	1,282,006,033	0.002
susan	提案コストによるスケジュール長	265	233	12.075472
susan(s)	実時間（RDTSC）	359,067,889	358,045,623	0.2846999
susan(e)	実時間（RDTSC）	81,608,665	80,457,236	1.4109151
susan(c)	実時間（RDTSC）	40,029,505	38,907,192	2.8037144
NasBench(IS)	提案コストによるスケジュール長	55,984,147	55,876,623	0.1920615
	実時間(RDTSC)	1,785,007	1,784,952	0.0030812

4．1．2　実験 2:実行時間の減少率の比較

この実験では，実際に逐次実行を行った実行時間と，生成された並列プログラムの実行時間を比較し減少率を算出する。なお元のソースプログラムと生成されたソースコードは共にmpiccでコンパイルしており，オプションの設定も同じである。また，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　実行時間における減少率の比較

	逐次実行(秒)	並列実行(秒)	減少率(%)
basicmath	74.549	52.776	29.20568
susan(s)	0.16530238	0.161720437	2.16690358
susan(e)	0.036015911	0.040963003	−13.735851
susan(c)	0.017626467	0.018816229	−6.7498582
himeno	50.81711283	50.34413226	0.93075059
NasBench	0.029267821	0.029942926	−2.3066465

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。

なお元のソースプログラムと生成されたソースコードは共にmpiccでコンパイルしており，オプションの設定も同じである。また，プロセッサ数4で並列実行を行った。

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より，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。これは，basicmathのような計算を行うプログラムであれば本提案手法は有効であると考えられる。

➢ 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。このようなベンチマークに対しては，今後の実行時間解析の改良の他に，本トランスレータに昨年度搭載された動的実行制御を行う並列プログラムが有効であると言える。

➢ 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大きい結果となっている。原因は，マシンのスペック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に加え，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。これは，同一の関数であることからそれらのタスクのコストは引数に関わらず同値であるとみなされるからである。

このことから，同一の関数であってもその引数によって実行時間が変化するという性質を考慮すべきであると考えられる。

➢ NAS Parallel Benchmarks について

このベンチマークに関しても無視できない誤差が発生してしまっている。これは，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になっ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。

このことから，より詳しく構造を解析し，実行時間を推測する必要があると言える。

➢ 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り，それによる誤差が生じるものと考えられる。susanベンチマーク，姫野ベンチマーク，NAS Parallel Benchmarksに関しては，更なる実行時間解析の改良が求められる。basicmathに関しては，最も誤差の少ない結果となったが，約7%の誤差があった。これは，暫定解であることが誤差を生む原因の一つであると考えられる。

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の実行時間と，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。また，実験1の表5.1と実験2の表5.2から，スケジューリング上の減少率と実際の実行時間の減少率の比較を行った。

➢ 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。

また，susan(e)とsusan(c)に関しては，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。この原因に関しては，まとめにおいて考察を行う。

➢ 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。それ以外の原因に関しては以下のまとめで考察を行う。

➢ 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。

また，実際に通信遅延を考慮した静的タスクスケジューリングを行い，その結果を元に並列化コードを出力したが実行時間が増加してしまう場合が存在した。このことに関しては，提案手法が有効に作用しなかった以外にトランスレータ全体の問題として以下のことが原因としてあげられる。

1．main関数のみの並列化

今年度のトランスレータでは，並列化を行う部分がプログラム中のmain関数だけである。これは各関数やループ文などのタスクに対してはそれ以上の並列性解析を行わず一つのタスクとして扱い，その呼び出しを行うmain関数のみを並列処理するというものである。しかし，多くのプログラムは関数やループ文がボトルネックとなっておりそれらのタスクに対して並列処理を行わない限り速度の向上は見込めない。

実際にボトルネックと思われるタスクに対して並列性を抽出することは現時点で可能となっているが，そのタスクに対してどのようにして適切な数のプロセッサを割り当てるのか，特定のタスクを複数のプロセッサで処理を行う場合のスケジューリング方法といった技術が未成熟であるため実装の段階に至っていない。

2．MPI命令の選択と挿入

今年度のトランスレータで出力される並列化コードは，最低限の通信を行う命令のみである。本来であればMPI_Gather，MPI_Reduceといった集団通信命令，MPI_Barrierといった命令を適切に使用しなければ効率の良いコードの生成は不可能である。過去の研究により，MPI_Gather，MPI_Reduceの使用するための技術は研究されてきたが，どのような条件で命令を挿入するのかといったことが問題となり実装には至っていない。

以上の結果から，本提案手法が有効なプログラムは一部存在するもの，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．まとめと今後の課題

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。

結果として，前年度のメモリアクセス命令だけを考慮した計算量解析と比べ，ソースコード静的メトリクスを利用することによりタスクのもつ性質をより詳しく解析することが出来，精度の向上に成功した。

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，選択を行っていく必要がある。

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] 竹本 拓未: ” 最大並列度を考慮したタスク粒度解析手法の提案と評価”, 学士論文, 成蹊大学理工学部情報科学科, 2013.

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