成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.54 No.1 (2017)pp.1-8

C言語自動並列化トランスレータの開発
—タスク粒度に着目したコードリストラクチャリング手法の実現—

近藤 竜也[*1], 甲斐 宗徳[*2]

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

Tatsuya KONDO[*1], Munenori KAI[*2]

ABSTRACT：In our automatic parallelizing translator for sequential C programs with MPI, a set of all statements in a block scope is defined as a compound task. In this paper, at first, we implemented a parallelism analyzer for the inner levels of hierarchy of scopes in any compound task. By using this analyzer, we analyzed single loops and nested loops whose processing time may take the most of total processing time of a program in general. Although it seems that a loop has no parallelism at a glance, the loop may be restructured to have parallelism by eliminating data dependencies, called loop distribution. In addition, in order to reduce more processing time of for-loops, a code restructuring method, that has extract the efficiency of cache memory, has been implemented. These implementation result in reducing parallel processing time remarkably.

Keywords：Parallelizing translator, Code generator, Parallel processing, Loop restructuring

(Received May 27, 2017)

1．はじめに

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ためには、タスクスケジューリングを行う必要がある。そして無駄な通信オーバヘッドを減らし最適なタスクスケジューリング結果をできる限り短時間で得るためには、タスクを適切な粒度にまとめ上げておく必要がある。

タスクの適切な粒度を求めるためには、プログラムの全ての階層構造を解析して、コードのリストラクチャリングを行いながらタスク粒度を調整して決定していく必要がある。

本研究では、そのためのソースコードのリストラクチャリング手法を実現することを目的とする。

[*1]：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)

2．C言語自動並列化トランスレータ

ソフトウェア研究室で開発中の自動並列化トランスレータ[1]（以下、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、出力することを目指している。

2．1　並列化トランスレータの構造

図 2.1 はトランスレータの処理手順を表している。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、初期状態ではステートメントレベルとしているタスク間のデータ依存解析と制御依存解析により並列性を抽出する。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。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おいては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強NP困難な組合せ最適化問題である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。

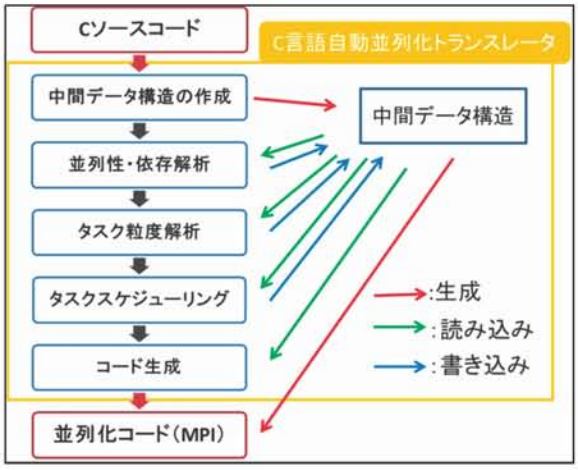

図 2.1　トランスレータ処理手順

2. 2　タスクについて

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。タスクの初期粒度は一部を除きステートメントレベルである。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。

2. 3　データの依存関係

データの依存関係は、プログラムを並列化する妨げとなるものである。データの依存関係には「フロー依存」「逆依存」「出力依存」の 3 種類がある。

フロー依存とは、先行するステートメントでWriteされている変数がその後続のステートメントでReadされている場合に発生する。逆依存とは、先行ステートメントでReadされている変数がその後続ステートメントでWriteされている場合に発生する。出力依存とは、あるステートメントにおいて並べ替えを行うことによって最終的にその変数に期待する値が変わってしまう場合に起こる。これらの依存関係を図 2.2 に示す。

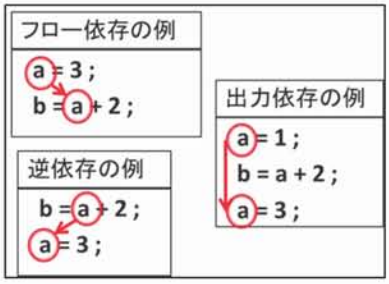

図 2.2　データ依存関係の例

2. 4　タスクグラフについて

タスク間には依存関係に基づく先行制約があり、その関係を図 2.3 のように表現した有向グラフをタスクグラフと呼ぶ。タスクはノードで、先行・後続関係は有向エッジで表す。

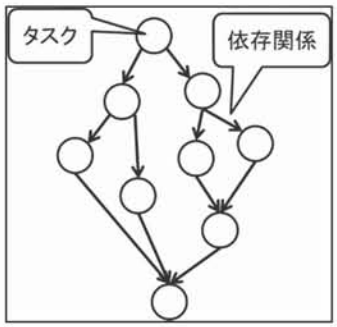

図 2.3　タスクグラフの例

2. 5　タスク粒度解析について

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、トランスレータではそのための手段としてタスク融合手法を組み込んできた。

3. 現在のトランスレータの問題点

現在のトランスレータでは解析対象をmain関数に書かれている第一階層のものに限定している。そのため、ユーザ定義関数やfor文といったブロックはそれ自身を 1 つ

のタスクとみなしている。しかし、実際はそのタスクにはステートメントレベルのタスクが内在し、並列性がある。本年度の研究では、そのような階層構造における全ての階層に対して解析を行うことによって、より多くの並列性を抽出し、並列処理によるプログラムの実行時間の減少を目的とする。

3. 1　コンパウンドタスクに対する並列性解析の拡張

今までのトランスレータは解析対象をmain関数に限定していたため、タスクグラフはmain関数を第一階層でタスクグラフ化した 1 つのみを生成していた。今年度は既に実装されているmain関数のタスクグラフに加えて、ユーザ定義関数やfor文などブロックで表されているコンパウンドタスクのタスクグラフ化を実装した。これらのコンパウンドタスクのタスクグラフをサブタスクグラフと定義する。

複数のタスクグラフをサブタスクグラフとして登録することが可能であり、各サブタスクグラフはタスクグラフ化された自身のタスクIDと紐付けられている。そのため、main関数のタスクグラフを解析している中でコンパウンドタスクにたどり着いた際に、そのタスクを表すサブタスクグラフの内部まで階層的に解析を行うことが可能である。また、各サブタスクグラフは自身のコンパウンドタスク情報（ユーザ定義関数やforタスクなど）を併せて保持する。図 3.1 にサブタスクグラフの例を示す。

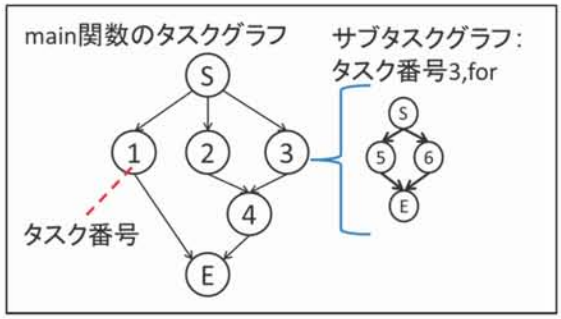

図 3.1　サブタスクグラフの例

サブタスクグラフを作成することにより、今までは解析を行っていなかったコンパウンドタスクの内部のタスク群に対して解析が可能となり、全階層構造に対して解析が可能となったといえる。これにより、前年度までに実装されていた解析がmain関数に対してだけでなく、ユーザ定義関数に対しても適用可能となった。

本研究では主に、プログラムの実行時間の大半を占めると言われるfor文によるループ処理に対して並列性の向上を図る。

4．ループリストラクチャリング

一見並列処理ができそうにないループに対して、ループ内の依存関係を解析し、依存関係を解消することによって、並列処理可能なループに再構築できる場合がある。ループを再構築するための手法としてループリストラクチャリング[2]手法が挙げられる。本研究では、並列性の向上を図るループディストリビューションとループ自身の処理時間の短縮を図るループのブロック化を実装した。

4. 1　ループの解析について

4. 1. 1　イテレーションとDOALL処理

イテレーションとは、1 回ループが回る毎に行われる 1 セットのステートメント群のことである。イテレーション間に依存が無く，ループの制御変数の値におけるイテレーションを全て並列に実行することをDOALL処理という。DOALL処理が可能なループの例を図 4.1 に示す。

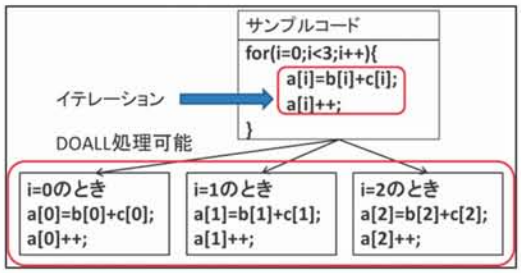

図 4.1　イテレーションとDOALL処理の例

4. 1. 2　イテレーション内・イテレーション間依存

各イテレーションで発生する依存のことをイテレーション内依存という。イテレーション内依存は、DOALL処理を行ったとしても、他のイテレーションに影響を与えることなく実行できる依存のことである。

制御変数の増減により発生する依存のことをイテレーション間依存という。配列変数を例にとると，同じシンボルでインデックスが違う場合に発生する依存である。イテレーション間依存が存在すると、DOALL処理を行うことができない。

図 4.2 にイテレーション内依存とイテレーション間依存の例を示す。

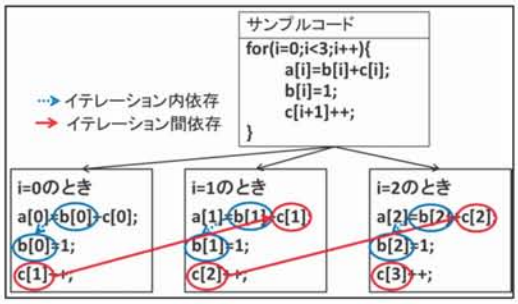

図 4.2　イテレーション内・間依存の例

4. 2　ループディストリビューション

ループディストリビューションとは、ループを複数に分割することにより並列性を上げる手法である。ループディストリビューションは前方へのイテレーション間依存が存在しない場合に適用可能であり、分割後のループをDOALL処理可能なループとして構築できる場合がある。また、1 つのループを複数のループへと分割することによって内部階層の並列性を上位階層で利用することが可能になる場合がある。

図 4.3 はループディストリビューションを適用することによってループの内部階層にある並列性を上位階層で利用を可能とするイメージを示す。ループ内にある１－１～２－２はそのループに内在するステートメントを表している。

実装したループディストリビューションの処理手順は、まず変数のシンボル単位でのループの分割を行い、その後に配列変数のインデックスに注目してループの分割を行う。

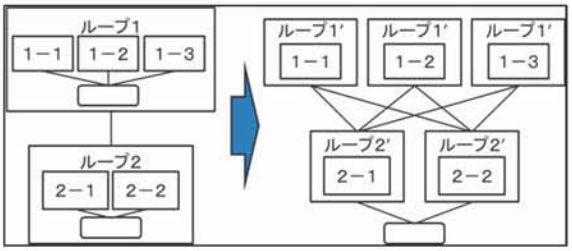

図 4.3　ループディストリビューションによる並列性の変化の例

4. 2. 1　シンボル単位でのループ分割

イテレーション内のステートメントにおける各変数シンボルのRead/Writeを解析し、使用されているシンボルによってイテレーションを複数のステートメント群に分割する。

同一シンボルを使用している複数のステートメントに関しては、イテレーション内にそのシンボルにWriteしているステートメントが存在する場合は、そのシンボルを使用するステートメント群は同一のfor文で実行するように分割を行う。

複数のステートメントにおいてReadのみで使用されているシンボルについては複数のfor文で実行するように分割することが可能である。この分割によって、for文をDOALL処理可能なfor文へと再構築できる場合がある。

図 4.4 は使用している変数シンボルによって、1 つのループを 2 つのループに分割している例である。分割前のループはDOALL処理が不可能であり逐次処理を行わなければならないが、分割後の 2 つ目のループはDOALL処理が可能となった。

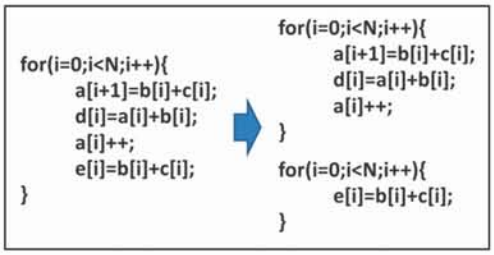

図 4.4　変数のシンボルによってループを分割する例

4. 2. 2　インデックス単位でのループ分割

イテレーション内の各ステートメントで使用される配列変数において、Read/Writeされるインデックスを解析する。同一シンボルを使用している複数のステートメントにおいて、前方へのイテレーション間依存が存在しない場合、それらのステートメント群は複数のfor文で実行するように分割することが可能である。イテレーション間依存の方向についてはインデックスの大小関係から判断する。ただし、分割後のfor文に依存関係は存在する。

この分割によって、for文をDOALL処理可能なfor文へと再構築できる場合がある。

図 4.5 は変数のインデックスによって、図 4.4 の分割後の 1 つ目のループをさらに 2 つに分割した図である。逐次実行をしなければならなかったループが、分割することによって共にDOALL処理が可能な 2 つのループとなった。

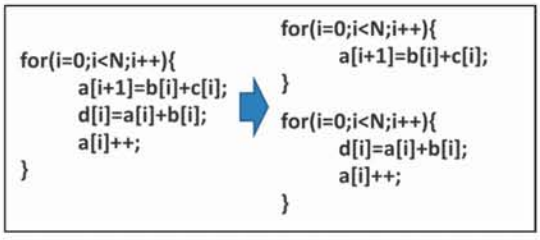

図 4.5　変数のインデックスによってループを分割する例

4. 3　ループのブロック化

上記で説明したループディストリビューションによって、並列性の向上が期待できる。それに併せて，forループ自体の処理時間の短縮を目的としたループのブロック化を実装した。

4. 3. 1　ブロック化の概要

ブロック化とは配列のアクセス範囲を局所化することによって処理の高速化を図る最適化手法である。CPUは、高速で小容量の記憶装置と、低速で大容量の記憶装置を組み合わせたメモリ階層構造[3]を使用している。メモリの階層構造を図 4.6 に示す。

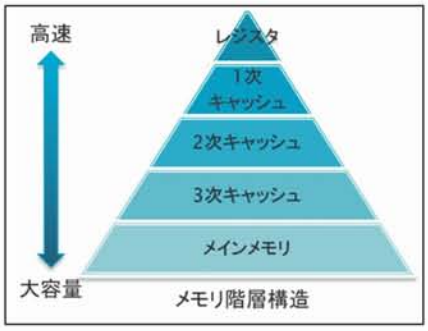

図 4.6　メモリの階層構造

上位層にある記憶装置ほど高速な代わりに容量が小さく，下位層にある記憶装置ほど低速な代わりに容量が大きい。演算装置が必要とするデータが上位層に保存されていなければその下の層の記憶装置に保存されているかを調べ，最終的には低速なメインメモリにあるデータにアクセスすることになる。そのため，一度高速な記憶装置に保存したデータを再利用することができれば、メインメモリにアクセスする回数を減らすことに繋がり、処理の高速化が期待できる。

ブロック化を適用することによりデータのアクセスを局所的にすることによって、メインメモリと比べて高速にアクセスが可能なキャッシュメモリ上にあるデータを再利用することができる。

ブロック化を適用して効果が得られる例として，行列積の計算が挙げられる。行列積の計算におけるデータのアクセス（1 行分）を図 4.7 に示す。図の細い矢印は 1 要素を求めるための繰り返し，太い矢印は求める要素の繰り返しを表す。このとき，1 要素を求めるためのデータがすべてキャッシュに収まりきらない場合、次の要素を求める際には 1 度キャッシュに入れた行列Aの値が追い出されてしまっている。そのためキャッシュ上のデータを再利用できない。

そのような場合に、計算をブロックごとに行う、ループのブロック化を適用する。図 4.7 の行列積の計算にブロック化を適用した例を図 4.8 に示す。図 4.8 は行列の大きさnの半分の大きさでブロック化を行っている。このブロックの大きさをブロック幅という。

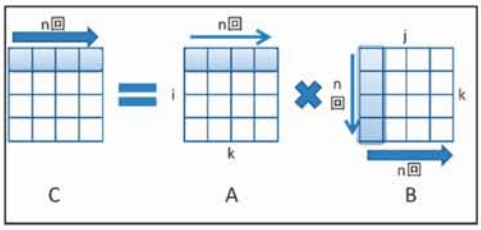

図 4.7　行列積の計算におけるデータアクセス

このブロック化を行うことによって、C_{11}ブロック内の計算は$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$となる。同時に使用する 3 ブロックが全てキャッシュに収まる場合、$C_{11} = A_{11} \times B_{11}$の計算後に$C_{11} += A_{12} \times B_{21}$の計算をするときには、キャッシュに$C_{11}$のデータが残っている。したがって、$C_{11}$はキャッシュ内のデータを再利用することが可能であり、メインメモリへのアクセス数を減らすことができる。

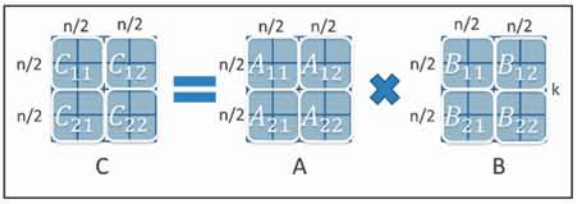

図 4.8　ブロック化を適用した例

4. 3. 2　ブロック化を適用する条件

ブロック化を適用する際には、そのfor文がブロック化による効果を得られるのかという点とブロック幅の大きさを考慮する必要がある。

キャッシュにデータを格納するときは単一のデータのみを格納するのではなく、ある連続データ単位ごとに格納する。そのため、行列の和の計算のような、常に連続データにアクセスするような計算に対してはブロック化を行っても効果が得られない。むしろブロックを作ることによってキャッシュに入っているデータを無視してしまうことになり、処理時間が長くなってしまう。そのようなfor文に対してブロック化は適用しない。

ブロック幅の大きさは同時に使用するブロックが全てキャッシュに収まる大きさで最大の値とする。また、ブロック幅は元の行列の大きさを割り切れなければならない。計算式を図 4.9 に示す。

1イテレーションの実行に必要なメモリ数×ブロック内の繰り返し数
＜キャッシュサイズ

図 4.9　ブロック幅の計算式

5．評価

評価方法は、for文を含むソースコードに対してトランスレータによるループリストラクチャリング手法を適用する。これによってプログラムの実行時間が短縮されるかを評価する。

実験環境は以下のとおりである。

— CPU：Intel(R) Xeon(R) E5-4640 @ 2.40GHz
— OS：OS: CentOS 6.5
— L2 キャッシュ：256000 Byte
— L3 キャッシュ：20480000 Byte

5．1　予備実験 1

予備実験 1 ではfor文を含むソースコードに対してトランスレータによるループディストリビューションを適用する。適用することによって、for文に内在する並列性を抽出でき、プログラムの実行時間が短縮するかを調べる。

実験に使用したソースコードは、同一のfor文内に並列に実行可能な行列積の計算が 2 つあり、そのfor文を 2 回記述している。計算を行う行列は 1000×1000 の大きさと 5000×5000 の大きさの 2 つのパターンで実験を行った。

ディストリビューション適用による実行時間の変化を図 5.1 に示す。並列実行することによって実行時間が 1000×1000 の行列では約 43%、5000×5000 の行列では約 49%短縮された。これは、行列を大きくすることによって計算時間と、その前後に必要な通信にかかる時間との比率が大きくなったためだと考えられる。また、プログラムの処理を均等に 2 並列に処理するため、行列を大きくし続けても減少率は 50%に近似するものと考えられる。

	分割なし（逐次）（秒）	分割あり（2並列）（秒）	分割時間（秒）	減少率(%)
1000*1000行列	31.142725	17.755210	0.0113690	42.951091
5000*5000行列	6737.930035	3407.550380	0.0113690	49.427172

図 5.1　ディストリビューション適用による実行時間の変化

5．2　予備実験 2

予備実験 2 ではfor文を含むソースコードに対してトランスレータによるループのブロック化を適用することによって、プログラムの実行時間が短縮するかを調べる。実験に適用したソースコードにはfor文が 2 つある。1 つ目のfor文では、2 つの行列を適当な値に初期化する。2 つ目のfor文ではそれらの行列の積を計算する。計算を行う行列はfloat型で、1000×1000 の大きさと 5000×5000 の大きさの 2 つのパターンで実験を行った。

図 5.2 にブロック化を適用した 2 つのfor文を示す。初期化をしているfor文はデータアクセスが連続しているデータに対するものなので、ブロック化を適用していないことがわかる。また、for文の上部（プログラム内の変数宣言部）にブロック幅を表す変数（BLK0）とfor文の制御変数用の変数（ii,jj,kk）が追加で宣言されている。図 4.8 の計算式でブロック幅を計算すると、L2 キャッシュに収まるブロック幅は 25、L3 キャッシュに収まるブロック幅は 100 となる。

```
int BLK0=25;
int ii;
int jj;
int kk;
for(i=0;i<NUM;i++){
    for(j=0;j<NUM;j++){
        a[i][j]=j;
        b[i][j]=-j;
    }
}
gettimeofday(&s,NULL);
for(ii=0;ii<NUM;ii+=BLK0){
    for(jj=0;jj<NUM;jj+=BLK0){
        for(kk=0;kk<NUM;kk+=BLK0){
            for(i=ii;i<ii+BLK0;i++){
                for(j=jj;j<jj+BLK0;j++){
                    for(k=kk;k<kk+BLK0;k++){
                        c[i][j]+=a[i][k]*b[k][j];
                    }
                }
            }
        }
    }
}
gettimeofday(&e,NULL);
```

図 5.2　ブロック化を適用したコード

ブロック化適用による実行時間の変化を図 5.3 に示す。この実験のソースコードは並列実行可能な部分が無いため、並列処理は行っていない。また実行時間の計算は行列積の計算を行うfor文の部分のみである。ブロック化を適用することによって、1000×1000 の行列では約 9%、5000×5000 の行列では約 44%実行時間が短縮された。

	ブロック化なし（秒）	ブロック化あり(秒)	ブロック化時間(秒)	減少率(%)
1000*1000行列（ブロック幅25）	4.752734	4.308241	0.004643	9.254673
1000*1000行列（ブロック幅100）	4.752734	4.297442	0.004643	9.481889
5000*5000行列（ブロック幅25）	961.541800	544.968487	0.004981	43.322956
5000*5000行列（ブロック幅100）	961.541800	537.317177	0.004981	44.118689

図 5.3　ブロック化適用による実行時間の変化

5. 3 評価実験

評価実験では、ベンチマークプログラムに対してトランスレータによるループリストラクチャリング手法を適用した。ベンチマークにはUnixBenchテストの 1 つであるDhrystoneベンチマーク[4]を使用した。Dhrystoneベンチマークは整数演算を行うプログラムで、1 秒間にメインループを回った回数で性能を評価するベンチマークである。

Dhrystoneベンチマークに対してトランスレータによるループリストラクチャリングを適用する場合としない場合でのループ回数の比較を図 5.4 に示す。ループディストリビューションによってメインループを小さくすることにより、ループの繰り返し回数を約 10%増やすことができた。その際にメインループから分割されたループは並列に実行されている。

	リストラクチャリングなし（回）	リストラクチャリングあり（回）	ループ回数増加率（%）
実行時間1秒	30,247,558	33,738,730	11.541996
実行時間10秒	305,533,741	344,341,531	12.701638
実行時間60秒	1,833,388,952	2,081,591,217	13.537894

図 5.4　リストラクチャリング適用有無でのループ回数の比較

ループを繰り返す回数でCPUの性能を評価するDhrystoneベンチマークとしては、ループを分割することによって意図が変わってしまうといえるが、そのような並列実行を想定していないプログラムに対して並列性の抽出・ループディストリビューションの適用ができた。

また、今回行ったループディストリビューションはイテレーション内を分割する手法である。そのため、DOALL処理が可能であっても並列実行可能な部分がイテレーション内に存在しない場合は適用することができない。そのようなループの実行は、ループ全体を 1 つのプロセッサに割り当て、逐次処理を行う必要がある。

現在トランスレータには、ループをイテレーションごとにブロック分割してプロセッサに処理をさせるようなコード生成の実装はされていない。そのため、今回実装した手法と併せて、DOALL処理が可能なループに対するコード生成を実装する必要がある。そのようなコード生成が実装された場合、1 つのループに対して複数のプロセッサで処理が行われるため、その分だけ処理速度が上がり、処理時間は短縮されると考えられる。

6. 結論

本研究では、プログラムの実行時間の短縮を目的として、コンパウンドタスクに対する解析の拡張、ループリストラクチャリング手法をトランスレータに実装した。これにより、プログラムの実行時間の大半を占めるループ文や関数に内在する並列性の解析が可能となった。

ループディストリビューションを適用することによってfor文に内在する並列性を上位層で利用することが可能となった。キャッシュメモリサイズを考慮したループのブロック化を適用することによって処理時間のかかるfor文を再構築し処理時間を短くすることが可能となった。

予備実験より、ループディストリビューション、ブロック化は共に適用対象となるfor文のサイズに依存して最大で 40%以上の時間を削減した。

評価実験より、並列実行を想定していないプログラムに対しての並列性の解析、ループリストラクチャリングの適用により処理速度を 10%以上向上させることができた。

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] 本多 弘樹:“マルチプロセッサシステムのためのコンパイラ技術”, 情報処理, Vol.31, No.6, pp.744-752, June 1990

[3] ジョン・L・ヘネシー, デイビッド・A・パターソン:“コンピュータアーキテクチャ　定量的アプローチ第 5 版”, 第 2 章, 翔泳社, Mar. 2014

[4] Dhrystone ベンチマーク：
https://github.com/kdlucas/byte-unixbench
2017/2/1 現在、参照可能