Vol.2012-HPC-133 No.11
2012/3/26

バイナリトランスレーションによるループ反復間のデータ依存解析

佐藤幸紀[†1]　井口寧[†1]　中村維男[†2]

アプリケーションプログラムからループレベル並列性を抽出し効率的にハードウェアリソースに合わせて並列化することは多くの並列処理システムにおける鍵となるプロセスであり、大規模並列な CPU コアやアクセラレータを用いた高度な並列処理が必要になる状況においてはますます重要性は増すと予想される。しかしながら、現状では並列処理を効率的に行うためにはプログラマが手作業でアプリケーション全体の構造を正確に理解した上で並列部分に分割し効率的にハードウェアリソースに合わせてマッピングすることが必要であり、非常にコストのかかる経験的な作業であるといえる。さらに、アプリケーションプログラムは年々その規模と複雑さを増してきているため、大規模・複雑化するアプリケーションプログラムを生産的かつ効率的に並列化する手法を確立することが求められている。本稿ではループ反復間のデータ依存の有無を実行時にプロファイリングすることによりループ並列性の抽出やハードウェアへのマッピングを支援する機構をバイナリトランスレータを用いて実装する。本ループ依存プロファイラは、コンパイル済みの実行バイナリコードを入力としてランタイムにループ反復間のデータ依存関係をモニターし、並列性を妨げる依存を検出する。本機構を有効性を確認するために NAS Parallel benchmark を用いて評価を行った。その結果、プロファイルにより得られたデータ依存関係は実際のソースコードの示すデータ依存関係と矛盾なく一致することを確認した。また、本結果に基づき、本プロファイラによる結果とコンパイラによる静的な並列化の際のデータ依存解析との相違点を示すとともに、HPC のコードチューニングに応用する際の課題について考察を行った。

Runtime data flow analysis among loop iterations using a binary translation system

Yukinori Sato,[†1] Yasushi Inoguchi[†1]
and Tadao Nakamura[†2]

Extracting loop-level parallelism from an application program and mapping it onto actual hardware resources are a key to determine the efficiency and productivity of parallelization. As parallel processing techniques using massively parallelized multicore CPUs and accelerators become popular, the key is expected to become more important. In this paper, we present an advanced technique that can monitor data dependencies across loop iterations at runtime. Using pre-compiled binary code as an input, we can monitor data dependencies via memory accesses together with inter-procedural nested loop structures. We implement our mechanism on a dynamic binary translation system and evaluated it using NAS Parallel benchmark suite. From the results, we confirm that we can extract data dependencies among loop iterations and the obtained data dependencies consist with data dependencies implied by the original source codes. Based on these results, we also show the differences in the obtained dependencies between our run-time loop profiling and the traditional static data analysis and investigate issues to apply them to code tuning for HPC applications.

1. はじめに

プログラムの実行時間の大部分をループが占めているということからも、ループレベル並列性はプログラムの並列化技術として広く使われてきた[1)]。ループ構造はプログラムの反復的な挙動を記述する上で必要不可欠である。もしループを使わないでプログラムを書くならば全ての反復的な挙動をループを展開した形式で記述しなければならなず、ループ構造なしでプログラムを書くことは生産性や可読性の面から現実的ではない。このようなことからもループ構造はアプリケーションプログラムに内在する並列性を抽出したり理解する上で重要な構成要素であるといえる。

ループを特徴づける性質としてループボディのサイズ、ループ反復数、ループの出現回数やループ反復間データ依存の有無やその位置があり、それらは抽出可能なループ並列性の並列度や実際にハードウェアで並列実行する際の性能向上に影響することが知られている[2)]。これらループの特徴はアルゴリズム、プログラムの構造、そして、コンパイラなどの利用するソフトウェアスタックにより決定づけられる。加えて、高効率な並列処理を行うためには

†1 北陸先端科学技術大学院大学 情報社会基盤研究センター
Research Center for Advanced Computing Infrastructure, Japan Advanced Institute of Science and Technology
†2 慶應義塾大学
Keio University

並列なハードウェア上で並列に実行する際のリソースマッピングにおいて存在する多種多様な選択肢からハードウェアとループの双方の特性を最大限活用する解を発見的に見つける必要がある。

近年、CPU 上に多数のコアを集積するマルチコア CPU や SIMD、FPGA といったアクセラレータを組み合わせたヘテロジーニアスな処理要素を組み合わせた大規模並列システムの普及が急速に進んでいる。メニーコアプロセッサやアクセラレータといった強力な演算能力を備える並列処理エンジンから持続的に理論性能に近い性能を引き出すためには、アプリケーションの実装毎に変化する並列化するべき対象を的確に見つけ出した後、その部分を並列実行する適切な手段を並列処理エンジンのマイクロアーキテクチャを意識しつつ決定することが必要である。

このようなハードウェアへのマッピングの困難さに加えて、年々進行する実アプリケーションプログラムの大規模・複雑化に由来するプログラミング生産性低下の問題も同時に考える必要がある。例えば、エクサスケール時代の HPC システムにおいては、マルチスケール・マルチフィジックス現象の統合シミュレーションを処理する機会が増えると予想されている。このようなマルチスケール・マルチフィジックスによるアプローチでは、1 つの分野の理論では対応できない複雑な現象を解析するためにミクロからマクロまでの幅広いスケールにわたり様々な物理現象の基礎方程式を解き、現象を再現しようと取り組む。すなわち、アプリケーション内部には特定の現象の解析のため細分化されたモデルや支配方程式、アルゴリズムが多数存在し、それらが複雑に統合された形で単体のアプリケーションとなる。異なる複数モデル間において時間ステップごとにデータ交換が必要な場合、時系列ループ内部での通信が必要となる。従って、細分化されたレベル毎に並列化するべき対象を的確に見つけ出すことに加えて、アプリケーション全体のデータの流れや並列化対象部分を処理の特性やロードバランスの観点から適切なハードウェア資源にマッピングしていくことが必要となる[3)4)]。

しかしながら、ハードウェア構成はメモリ階層やアクセラレータ、相互接続方式の面で多様化を続ける一方で、それらを的確にマッピングしていくことは並列アプリケーション開発者にとっては非常に大きな負担となっている。さらに、年を追うごとに規模や複雑さを増大するコードから並列化するべき対象を的確に見つけ出すことも多くの労力を必要とする。従って、大規模な並列性を把握する方法論を確立し高度な並列化を生産的に達成する手法を確立することが必須であると考えられる。

そこで、本研究においては大規模・複雑化するアプリケーションプログラムから生産的かつ効率的にハードウェア構成に適する的確なループレベル並列性を抽出することを支援することを目的として、コンパイル済みの実行バイナリコードを入力としてランタイムにループ反復間のデータ依存関係をモニターする手法を提案する。本手法により、プログラマがソースコード全体を一通り読まなくともプログラム実行時におけるループを単位としたデータ依存関係やデータの流れのイメージを得ることが可能となる。また、プログラムの実行時にデータ依存関係の解析を行うというアプローチによりコンパイル時には把握が困難であったポインタによる間接メモリアクセスも詳細に解析することが可能となる。このような詳細なデータ依存解析は効果的に並列化対象部分を抽出することに寄与すると予想される。

本稿では、先行研究[5)] において提案されているループ領域単位を示す Loop-Call Context Tree グラフ上にデータフローグラフを生成する機構を発展させ、動的バイナリトランスレーション（DBT）システム上にコード実行時に出現するループの反復間に存在するデータ依存関係を検出する機構を提案する。また、本システムを実際に構築し、NAS Parallel Benchmark v3.3 を用いて評価実験を行うとともに、結果についての考察を行う。

本論文の構成は以下の通りである。2 節はでバイナリトランスレーション技術によりループ階層構造を抽出する手法の概要を述べる。3 節ではループ領域および反復間毎のメモリを介したデータ依存関係を抽出する手法とその表現方法について述べる。4 節ではベンチマークプログラムの実行を通して本手法の基礎的な評価と HPC のコードチューニングに応用する際の課題についての考察を行う。5 節は結論である。

2. ループ階層構造とループ反復の抽出

図 1 に本研究で提案する動的バイナリトランスレーション（DBT）技術を用いたデータフロー解析システムの概要を示す。本手法は図 1 に示すように静的解析（Static Analysis）と動的解析（Dynamic Analysis）の組み合わせにより実現される。以下、ループ階層構造とループ反復を検出する手法の概要を順を追って説明する。

ループ階層構造を抽出する手法として先行研究[6)] にて報告された pMarker 法を用いる。その第一のステップとして、事前にコンパイルされているアプリケーションプログラムのバイナリコードを入力として静的解析を行う。pMarker 法においては静的解析はバイナリコードのイメージが DBT にロードされるときに行われる。静的解析では関数単位でコントロールフローグラフおよび支配木を構築し、Havlak のアルゴリズム[7)] に基づき reducible ループと irreducible ループの双方を検出する。

図 2 に抽象化したループの構造を示す。ループは head となる命令、tail となる命令、その

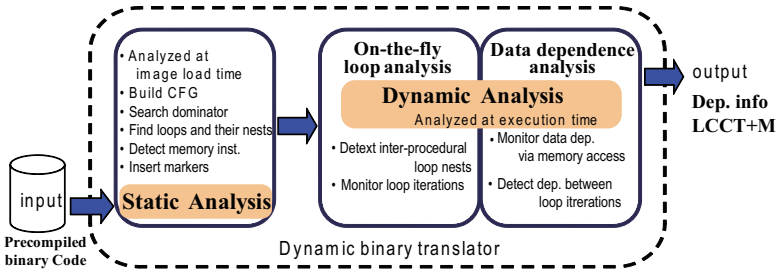

図 1 DBT を用いたにデータフロー解析機構

他の命令にて構成される。ループの head 命令がループ領域内のすべての命令を支配する場合は reducible ループであり、ループ領域外からの流入は head 以外にない。また、ループ内には少なくとも 1 つの head に戻るエッジがあり、制御フローをループさせる。irreducible ループについては少なくとも 1 つ以上の head 命令以外への流入がある点が異なる。

ループ内には少なくとも 1 つの head に戻るエッジがあり、ループが反復される毎にループボディのすべての命令を支配する head の命令が実行される。この特徴を利用してループ反復をモニターする。ここで、ループが何回反復実行したかを示すループトリップカウンタは head 命令を実行するごとにインクリメントするとした[4]。また、irreducible ループについては head 命令以外への流入があり得るためループに流入した時点でカウントを開始するとした。

実行時にループ階層構造を追跡するという目的を実現するためには、すべてのコントロールフローに着目する必要はなく、ループ領域外からの流入とループ領域外への流出に着目すればよい。そこで、pMarker 法ではループへの流入および流出となるエッジを監視することにより、実行時にループ階層構造を追跡する。この際、複雑に入り組むループネストを追跡するためにループへの流入と流出の状態を保持するスタックを用いて効率的にループ階層構造を追跡できるようにする。

プログラムの実行を伴わない静的解析においては関数内のループは解析可能であるが関数をまたぐ（inter-procedural な）ループ階層の解析や動的なコントロールフローの推定は非常に難しい。そこで、これらを動的解析によりプログラムの実行時に抽出する。Havlak のアルゴリズムにより reducible ループと irreducible ループの双方を検出した後、検出したループの領域およびループの head 命令を示すマーカーと関数呼び出しやリターンの位置を示すマーカーを生成し、動的解析における instrumentation のポイントとする。また、動

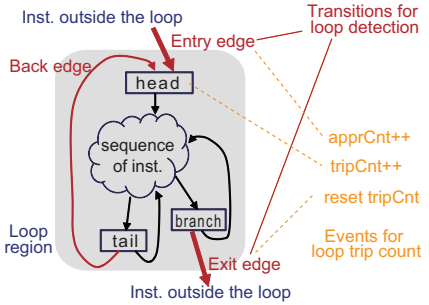

図 2 ループの構造

的解析時にマーカーのポイントが実行される毎に実行される解析コードを各マーカーのポイントに挿入する。解析コードとしてループ情報を記録するためのプログラムを用意する。

次のステップである動的解析においては、生成したマーカーのポイントが実行される毎にループ情報を記録するための解析コードが実行される。動的解析においてはプログラム実行中の条件分岐の挙動やコントロールフローを反映した解析が可能となるため、動的な実行で実際に現れるループ構造に関する情報を抽出することができる。

3. メモリを介したデータ依存の検出

3.1 ループ領域間のデータ依存検出

メモリを介したデータ依存の抽出するためには pMarker 法によるループ構造解析に加えて、先行研究[5] で論じられているようにメモリアクセス命令を検出しデータの読み込みおよび書き込みを監視することによりデータ依存を抽出する必要がある。そこで、本節にてループ領域や関数を解析の単位としてデータ依存を抽出する手法の概要を説明する。

ループ領域間のメモリを介するデータ依存を検出する第一のステップである静的解析では、ループ構造解析のためのマーカーに加えメモリアクセス命令に対してマーカーを生成し、メモリアクセスに関する情報を記録するための解析コードをマーパーのポイントに挿入する。第二のステップである動的解析においては、生成したマーカーのポイントが実行される毎にメモリアクセスに関する情報を記録するために挿入された解析コードが実行され、動的なメモリ依存の挙動を得る。さらに、メモリアクセスの際に実行される解析コードにおいては、データ依存関係をループ階層構造に基づき表現する機構を用意する。

図 3 にループ領域間の依存を検出するためのメモリライトおよびメモリリードアクセス

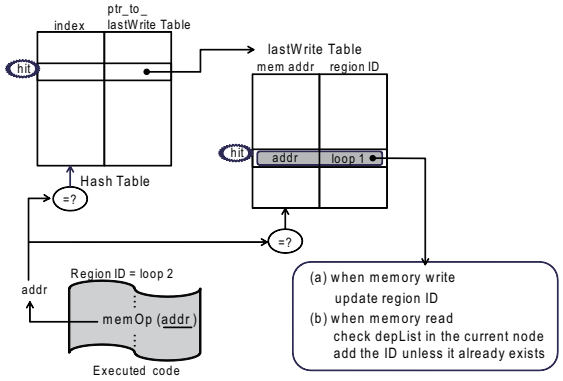

図 3 ループ領域間のメモリを介する依存の検出

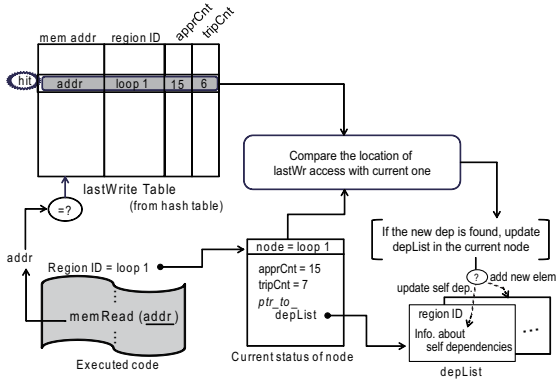

図 4 ループ反復間のデータ依存関係の検出

の際の動作の概要を示す。メモリを介したデータ依存関係を把握するためにメモリアドレス毎に最後に書き込みを行ったループの ID を保持するテーブルである lastWrite table を用いる。メモリライトアクセスが実行されるポイントで書き込みを行うメモリアドレスに対応するインデックスの書き込みを行ったループの ID（lastWrite LoopID）を現在実行されているループ ID に更新する。

メモリリードアクセスが実行されるポイントでは読み込みを行うメモリアドレスに対応するインデックスのループ ID を得る。このときのループ ID はそのメモリにある値が生成された領域を示す。すなわち、現在実行されているループはそのメモリにある値が生成された領域と依存しているということになる。このようにして得られた依存関係は 3.3 節で示す兄弟次男表現による LCCT+M 上のノード毎にどの領域と依存しているかを示す depList というリストにて保持される。そこで、現在のループノードに対応する depList に検出された依存のあるループ ID を追加する。

現在実行しているノードが LCCT+M 上の関数ノードである場合はループ ID の代わりに関数に対応する ID を用いることにより LCCT+M のどのノードにて生成された値と依存関係があるかを把握できるようにする。以上のような動作によりメモリを介したデータ依存をループ領域単位で抽出することが可能となる。

3.2 ループ反復間のデータ依存検出

ループレベル並列性のソースはループ反復間の並列性である。それぞれのループ反復が独立に実行可能である場合（すなわち、いかなる 2 つの反復間に依存関係がない場合となる）、そのループからループレベル並列性が抽出され、複数のループ反復は同時に実行される。しかしながら、3.1 節で論じたループ領域単位のデータ依存解析ではループ反復間の並列性は抽出できない。そこで、本論文では、先行研究[5] でのループ領域を単位とするデータ依存解析に加え、ループ反復を単位とするデータ依存を検出する手法を提案する。

図 4 にループ反復間のデータ依存関係の検出するためのメモリライトおよびメモリリードアクセスの際の動作の概要を示す。本手法は 2 節にて論じられたループトリップカウンタとループ出現カウンタを用いて効率的にループ反復間のデータ依存を検出する。ループトリップカウンタとループ出現カウンタは lastWrite テーブルのエントリとして保持され、ループ内に存在するメモリ書き込み命令が実行される毎に更新される。データ依存はループ内に存在するメモリ読み込み命令が実行される際に現在のカウンタ値を依存のあるアドレスに対応する lastWrite テーブルのエントリのカウント値と比較することにより検出される。ループの反復間、あるいは出現間に依存があった場合、ループ領域間の場合と同様に depList に依存の情報が記録される。

3.3 ループ階層構造間のデータ依存の表現

関数をまたぐプログラム全体のループ階層構造を効率的に保持するために **LCCT+M (Loop-Call Context Tree with Memory)** というデータ構造を構築する。LCCT+M はコールコンテキストプロファイリング[8] にて利用される CCT (Call Context Tree) を関数をまたぐループネスト構造を表現できるように拡張した L-CCT (Loop-Call Context Tree)[6] にメモリ依存情報を追加したものである。

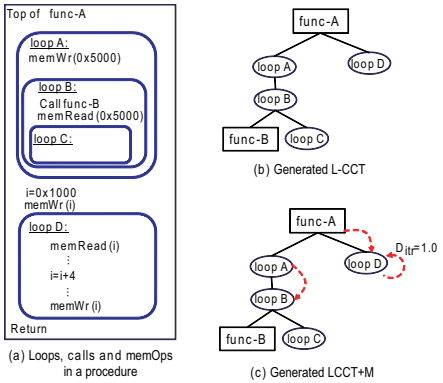

図 5 LCCT+M (Loop-Call Context Tree with Memory) グラフとデータ構造

図 5 (a) は関数 func-A のループ階層構造、call 命令、メモリ命令とそれぞれの相対位置を示す。関数 func-A は内部に 4 つのループを持ち、ループ B の内部で関数 func-B を呼び出しを行っている。L-CCT はこれらの呼び出しシークエンスを leftmost child right sibling binary tree（長男次男表現）を用いてにて正確に把握する。図 5 (b) に L-CCT にて表現された関数呼び出しとループの呼び出しのコンテクストフローを示す。ここで円形のノードはループを、四角のノードは関数を示す。Havlak のアルゴリズムにより検出される 2 つの任意のループはそれぞれがネストしているか、互いに素であるかのどちらかとなる。そこで、ループ内部で呼ばれるノードはそのノードの子ノードとし、また、素であるループは兄弟ノードとして追加する。図 5 (c) は最終的に生成される LCCT+M を示す。L-CCT に加えてデータ依存のあるノード間に点線で示されるエッジが追加されている。

考えられるデータ依存のエッジは関数/ループ領域間、ループ出現間、ループ反復間の 3 パターンのうちのいずれかであり、LCCT+M においてはそのいずれも表現可能である。ループ反復間、ループ出現間のエッジは自身のノードへの依存となるため、それらの平均依存距離である D_{itr}、D_{apr} を付加し、それぞれを識別する。その他の自身のノードへの依存は同一のループ反復内あるいは関数内での前方にある命令から後方の命令への真依存であるため、LCCT+M においてはエッジは追加しないとした。

4. 評価実験

4.1 実験環境

提案するループ反復間のデータ依存解析を評価するために動的バイナリトランスレーション技術を用いて実環境を構築し、ベンチマークプログラムにより基礎的な評価を行う。本機構の構築には DBT システムとして Pin tool set[9] を用いた。Pin はよく知られた DBT システムの 1 つであり、同一の ISA への変換を行う。DBT システムにおいては一度変換された命令群はコードキャッシュに保持されるため高速に参照することが可能である。

システムの評価には汎用的な x86 クラスタの 1 ノードを用いた。評価に用いたクラスタの詳細は以下である。ハードウェアとして 4 基の 6 コアを持つ 2.6GHz の CPU (AMD Opteron 8435) と 128GB の主記憶メモリを備える Appro 1143H というサーバーを、OS として Red Hat Enterprise Linux 5.4 を用いた。本システムは汎用的な x86_64 のクラスタであり、1 ノードあたり 24 コアの CPU が利用できる。この上に Pin のバージョン 2.8 (33586) Intel64 用の構成にて DBT システムの環境を構築した。

評価実験を行うためのベンチマークプログラムとして、NAS Parallel Benchmark v3.3 の逐次版の IS および CG を利用した。データサイズは class A を用いた。IS は C 言語により実装される大規模整数ソートであり、CG は共役勾配法のカーネルである。

ループネスト解析に関しては実行バイナリに含まれる領域のみを解析の対象として、動的にリンクされるライブラリはループネスト解析の対象から外した。加えて、解析は main 関数が実行される時点から開始し main 関数が終了した時点で停止することとした。また、プログラム全体の LCCT+M は非常に大きくなるため、実行頻度が高い Hot 領域を興味対象として結果の解析に用いた。本報告では Hot ノードを決める閾値として子孫を含む累計の実行命令数が全命令実行数に占める割合を用いる。閾値の値はノード自身が実行した命令数が 8 番目に大きいノードの子孫を含む累計命令実行の割合を用いた。また、Hot 領域をグラフとして可視化するためのツールとして graphviz を用いた。

4.2 評価結果

図 6 および図 7 に IS と CG について gcc4.1.2 に '-O -g -gdwarf-2' オプションを用いて生成したバイナリコードを利用してプロファイルを行った際の LCCT+M の Hot 領域を示す。ここで、丸いノードはループを示し、四角のノードは関数を表す。実線のエッジはコントロールフローを表し、点線のエッジはノード間のデータ依存を示す。各ノード内には自身のノードで実行された命令の全実行命令に占める割合およびカッコ内にそのノードと子孫の

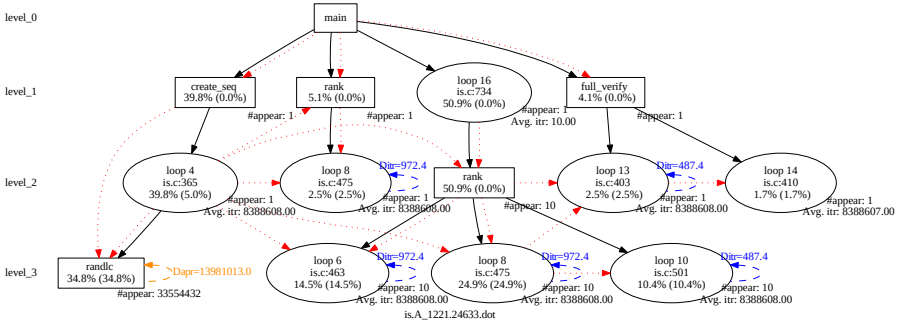

図 6 IS の Hot LCCT+M (gcc -O, Thr=1.66%).

総計の実行命令の割合が示されている。また、各ループノードの内部には、ループを識別するためのループ ID と、バイナリコードの位置に対応するデバッグ情報から得られたソースコード上の位置 (*filename:line*) がそれぞれ示されている。

図 6 に示されるように IS の実行においては 7 つのホットループ (loop 4, 6, 8, 10, 13, 14, 16) が検出された。それらのうち 4 つ（loop 6, 8, 10, 13）はループ反復間にメモリ依存があることが検出された。先行研究[5] では自分自身のノードと依存がある場合にもデータ依存エッジを追加しないとしていたため自身のノード内のデータ依存を検出することができなかったが、本実装においてはループ毎にトリップカウントや出現カウントを保持することによりループ反復間の依存のような自身のノードに依存がある場合を検出できたことが確認できる。実際のソースコードと比較し確認したところ、これらのループはポインタによるメモリの間接アクセスがあり、反復間の依存があり得る。従って、検出した結果とソースコード上の記述について矛盾はなかった。自身に依存のない 3 つのループ (loop 4, 14, 16) は反復間にメモリ依存がないためループレベル並列性を持つ可能性がある。そこで、ソースコードを確認したところこれらのループは ループ並列性をもつ DOALL な $'for'$ で記述されていることが分かった。

図 7 に CG の LCCT+M を示す。CG の実行においては 13 個のホットループが検出され、3 つのループネスト構造 (loop 11, 12, 13, 14, 15), (loop 25, 26, 27), (loop 32, 33) が (loop 4, 6, 8, 10, 13, 14, 16) があることが分かる。特にループ 32, 33 はそれぞれに自身への依存がなくアウターループであるループ 32 からインナーの 33 への依存も見られなかった。すなわち、アウターループにおいてループ並列性が利用できる可能性があるということ

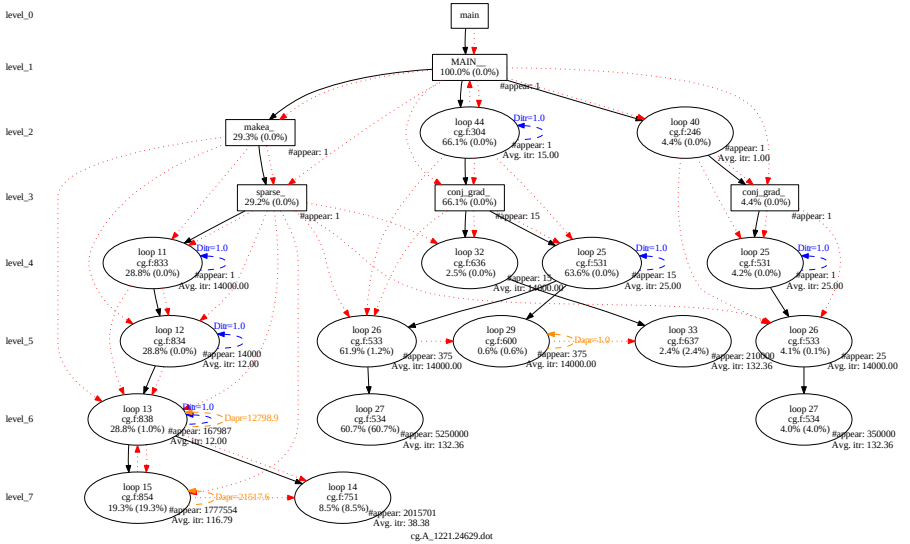

図 7 CG の LCCT+M (gcc -O, Thr=0.63%).

が分かる。これらを実際のソースコードと比較し確認したところ矛盾なく一致することが分かった。

4.2.1 ループ並列性を抽出するために

実験結果より、本手法はループ反復間にメモリを介するデータ依存が存在する場合には検出できることを確認した。本節ではこれらの結果を用いてどのようにループレベル並列性の有無を識別できるかということを論じる。

本システムはループ反復間のメモリ依存に着目してループ並列性の有無を実行時にプロファイリングする機構であるが、コンパイラによる静的な並列化と異なり以下の点を留意する必要がある。第一に、コンパイラにおけるデータ依存解析は全ての入力データや動的なコントロールフローに依存しない並列コードを生成することを目的とした出力が行われるが、本システムによる実行時のプロファイリングで得られる情報は実行を行った入力データに限ったデータ依存を出力することである。従って、実行時の解析では他のデータを入力として用いた場合や動的にコントロールフローが変化した場合において同様のデータ依存の情報が得られる保証がない。

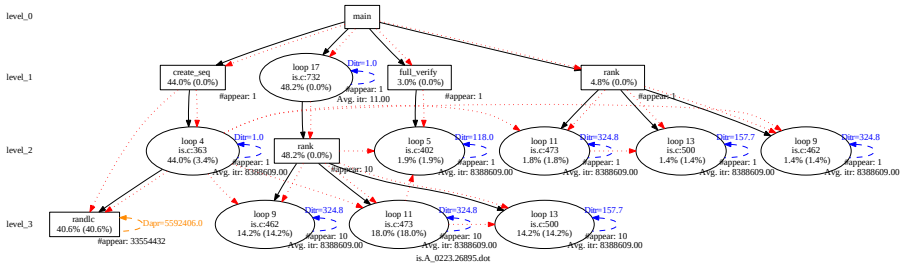

図 8 IS の Hot LCCT+M (gcc -O0, Thr=1.42%).

第二に、本システムの現在の実装においてはメモリを介するデータ依存に限定してデータ依存を検出しているが、実行コード中にはレジスタを介したデータ依存も存在することである。本研究にて想定している実際の並列化には、ソースコードを修正し並列化を行う、あるいは、バイナリコードをライタイムに書き換える等によりランタイムに並列化を行うという 2 つの方法がある。ソースコードを修正し再コンパイルする場合においては引き続きコンパイラがレジスタを介する依存を隠ぺい可能であるが、ランタイムの並列化においてはレジスタ依存の取り扱いや他のデータを入力したときの扱いを検討する必要がある。

具体的な例を示すために、objdump コマンドを用いてバイナリコードをダンプし、いくつかのホットループについてのレジスタ依存を確認した。その結果、図 7 の 2 重ループ 32, 33 はリダクション処理を行うが、そのインナーループ 33 は総和を求めるためにレジスタを介した反復間の依存がある。従って、DOALL のようなループ並列性があるわけではないといえる。なお、同様のリダクション処理はループ 27 でも見られる。

レジスタを介する依存のもう一つの例として、図 7 のループ 25 がある。ループ 25 はループ 26, 27 を内部に持つ 3 重ループのアウターループである。ループ 25 は自身に反復間の依存があると検出されたが、ソースコードにおいては DOALL な並列化が可能なループであった。これは、ループの反復数をカウントするためのループ変数がレジスタからスピルアウトしてしまったためにメモリに格納され、メモリを介した依存と判定されてしまったことに由来する。従って、元々のソースコードではループレベル並列性を持つと考えられても、実際に実行されるバイナリコード上で並列化可能であるかはコンパイラがどのようなコードを出力しているかに依存する。

図 8 に'-O0 -g -gdwarf-2' オプションを用いて生成した IS のバイナリコードを実行した際の LCCT+M の Hot 領域を示す。各ループの ID はランタイムに動的につけられるため図 6 でのループ ID と異なる。結果より、出現したすべてのループノードの反復間に依存があることが分かる。これは gcc は-O0 レベルではいかなる最適化も行わないため、おおよそすべてのデータはレジスタではなくメモリに格納されることに由来する。

4.3 HPC コードのチューニングへの応用に関して

ループ構造およびその反復間に存在する並列性は年々大規模・複雑化するアプリケーションにおいて適確にプログラムを分割し効率的に並列処理するための重要な手掛かりである。一般的に HPC のコードチューニングはソースコードにアクセスすることにより行われるため、ソースコードを熟読することにより依存関係を理解することは可能である。しかしながら、他人が書いた大規模なソースコードを細部まで読むことは非常に労力がかかるため、プログラム実行の概略を簡易的な方法で理解したいという要求もある。このような要求に対しては、本実行時プロファイリングを用いてループ並列性を妨げる依存関係を検出することによりプログラム実行の概略を提示することでチューニングの戦略を立てやすくなることが期待される。

実行時のプロファイリングによるデータ依存検出は、その出力が実行するために用いた入力に依存するという性質がある。この入力データセンシティブな特性を利用することにより、コンパイラが行っている静的なコード解析では得られないレベルの最適化を行うことができる可能性を秘めている。例えば、入力データセンシティブなコードについて入力データの傾向を分析することにより、それぞれの傾向に合うチューニングを行うためのヒントが提供できると期待される。

また、このようなランタイムのチューニングのためのヒントは自動チューニングの概念に直結する。例えば、自動チューニングの施行ランのフェーズとしてループ並列性のプロファイリングを行うことにより、より優れたアルゴリズムやシナリオを選択するために有用となる情報を提供できる可能性がある。

本プロファイリング機構はプログラマの操作を介さずにループ並列性を妨げる依存関係を検出するが、完全にループ並列性が存在するとは断定できない。従って、ランタイムのループ並列化を行う際にはループ並列性がないという予想が外れた際にも正しい結果を得るために Rauchwerger らが提案しているような投機的ループ並列実行[10] を検討する必要もある。

なお、HPC コードは MPI により並列化される場合も多いが、本プロファイリング機構は各プロセスに対してデータ依存解析を行うことにより対応可能である[4]。

5. 結　　論

本稿ではループ並列性を阻害するメモリを介するデータ依存関係を実行時にプロファイリングする機構を提案した。本機構はコンパイル済みの実行バイナリコードを入力としてランタイムにループ反復間のデータ依存関係を監視する。本機構をバイナリトランスレータを用いて実装し、評価を行った。その結果、プロファイルにより得られたデータ依存関係は実際のソースコードのものと矛盾なく一致することが分かった。しかしながら、コンパイラによる静的な並列化の際のデータ依存解析と違い、入力データ依存性があること、レジスタを介するデータ依存も存在することを留意する必要があることを示した。また、本機構を HPC のコードチューニングに用いる場合のユースケースとその際の課題を整理した。

謝　　辞

本研究は科研費 21700056 の助成を受けたものである。

参 考 文 献

1) Arun Kejariwal, et al. On the exploitation of loop-level parallelism in embedded applications. *ACM Trans. Embed. Comput. Syst.*, Vol.8, , February 2009.
2) J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, Vol.4, pp. 812–826, July 1993.
3) 佐藤幸紀. ループ構造に着目したマルチグレイン・マルチレイヤ並列処理システムの提案. 情報処理学会研究会報告 2008-ARC-172, pp. 25–28, 2008.
4) 佐藤幸紀, 井口寧, 中村維男. 動的バイナリトランスレーションによるループネスト検出とプログラムチューニング支援への応用. 情報処理学会研究会報告 Vol.2010-ARC-192 Vol.2010-HPC-128 No.10, pp. 1–7, 2010. 第 18 回ハイパフォーマンスコンピューティングとアーキテクチャの評価に関する北海道ワークショップ（HOKKE-18）.
5) 佐藤幸紀, 井口寧, 中村維男. Loop-call context tree を用いたランタイムデータフロー解析. 情報処理学会研究会報告 Vol.2011-ARC-196 No.13, pp. 1–6, July 2011. 2011 年並列/分散/協調処理に関する『鹿児島』サマー・ワークショップ (SWoPP 鹿児島 2011).
6) Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, May 2011.
7) Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, Vol.19, No.4, pp. 557–567, 1997.
8) Glenn Ammons, Thomas Ball, and JamesR. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pp. 85–96, 1997.
9) Chi-Keung Luk, et al. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 190–200, 2005.
10) Lawrence Rauchwerger and DavidA. Padua. The LRPD Test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, Vol.10, pp. 160–180, Feb. 1999.