Vol.2010-ARC-192 No.10
Vol.2010-HPC-128 No.10
2010/12/16

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

佐藤幸紀[†1] 井口寧[†1] 中村維男[†2]

超並列なマルチコア CPU やアクセラレータにて構成した HPC システムにおいて、大規模なアプリケーションプログラムを適確かつ効率的に並列処理するためには、プログラム中に様々な粒度で出現するループを手掛かりに並列化を行う必要がある。一方で、HPC の分野の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、プログラマがアプリケーション全体のループ構造を正確に理解することは困難な状況となりつつある。そこで、本研究においては、コンパイル後のバイナリコードからプログラム中に現れるループ構造を検出した結果をプログラムチューニングの支援へ応用することを提案する。本稿では、動的バイナリトランスレーション技術を用いて実行バイナリコードから実行時に動的な関数呼び出しをまたぐループネストや、それぞれのループ反復回数を検出する方法の概要を述べ、本手法を用いて MPI プログラムがどのように並列化されるかを解析することの原理を示し、併せて、例題について適応したの結果を示す。

Detecting nested loop structures using a dynamic binary translator and its application to performance tuning

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

To perform effective parallel processing of large-scale programs on massively parallel CPUs and accelerators, loop structures are one of the primary hints for finding parallelism. On the other hand, since current application programs become large and complex year by year, it becomes hard for programmers to fully understand loop structures across all of a program. In this paper, we present a case for applying results of detection of loop structures to assisting performance tuning by programmers. We briefly show how we detect loop nest structures across procedure calls and the numbers of loop iterations, how we analyze parallelized MPI programs, and present an example of how the MPI program is partitioned and parallelized.

1. はじめに

ペタスケール、そしてポストペタスケール時代の HPC システムでの大規模なプログラムを効率的かつ生産的に実行するために SIMD/GPU といった強力なアクセラレータや多数のマルチコア CPU を利用した超並列処理を核とした新しいアーキテクチャ技術やプログラムの並列化・高度化に関する技術の確立が求められている。しかしながら、並列処理を行うためにアプリケーションの事例毎に変化する並列化するべき対象を的確に見つけ出し、その部分を並列実行する適切なハードウェア資源にマッピングしていくことは解空間の広い発見的な作業であり、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、アプリケーションプログラムは年を追うごとにコードの規模や複雑さを増してきており、大規模な並列性を把握する方法論を確立し自動化を進めることが必須であると考えられる。

プログラムの実行時間の大半がループ部分で占められているように、並列処理の対象は何らかのループに属していると考えられる。もし、プログラムにループを使わないならばループを展開した膨大な量の逐次コードを用意しなければならず非生産的で非現実的かつ可読性に欠けるプログラムとなる。図 1 に並列性とループの対応関係を示す。ループにもインナーループやアウターループなどといった階層構造があり、ループの大きさも様々である。同一の関数内で完結する単一のループから、ループ内部で関数呼び出しをすることにより複数の関数やファイルにまたがり入れ子を形成するループも存在する。すなわち、ループは細粒度並列性から粗粒度並列性まで様々な粒度があり、並列性が実現されるレイヤも命令レベル、スレッドレベル、タスクレベル、プロセスレベルなど多種多様である。従って、ループ構造はマルチグレイン・マルチレイヤ並列処理にも対応可能な優れた指標と考えられる[1)]。

しかしながら、HPC の分野の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ことに伴い、これまではソースコード全体を一通り読むことにより把握されてきたループ構造に関してプログラマがアプリケーション全体のループ構造を正確に理解することが困難となりつつある。そこで、大規模なプログラムにおける様々な粒度の並列性を把

†1 北陸先端科学技術大学院大学 情報科学センター
Center for Information Science, Japan Advanced Institute of Science and Technology
†2 慶應義塾大学
Keio University

Vol.2010-ARC-192 No.10
Vol.2010-HPC-128 No.10
2010/12/16

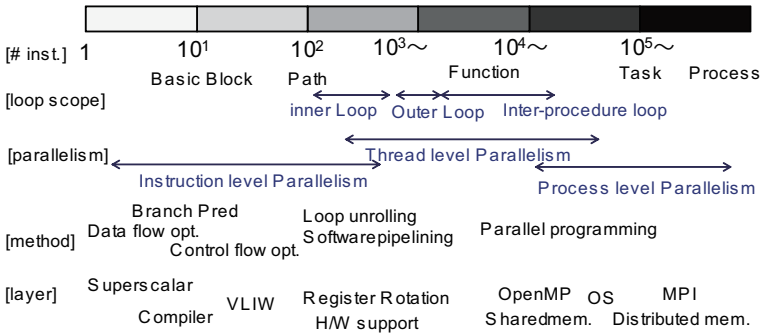

図 1 様々なレイヤと粒度にまたがる並列性とループ

握することを目的として、ループ構造に着目した実行時プロファイリングを行い、ループ構造の把握を自動化する方法論を確立することによりプログラマや開発者が生産的かつ効率的にプログラムの並列化・高度化およびチューニングすることを支援するすることを目指す。

本研究においては、プログラマがソースコード全体を一通り読まなくとも適切なチューニングが可能となることを目指しコンパイル済みのバイナリコードを入力としてプログラム中に現れるループ構造に関する情報を抽出することを行う。ループの実行はプログラム中の条件分岐やコントロールフローにも依存するために、コンパイル時の静的なコントロールフローグラフの構築に加えてプログラム実行中に得られる動的な分岐の挙動の把握が必要となる。

そこで、動的バイナリトランスレーション（DBT）技術を用いて実行バイナリコードから実行時に動的な関数呼び出しをまたぐループネストやそれぞれのループ反復回数を検出する。DBT 技術は、ランタイム最適化・並列化の実現など新しいマイクロアーキテクチャを確立するうえで重要な技術であると同時に、ハードウェアのレイアに近い情報のプロファイルやハードウェアの動的な挙動のモニタとしても応用されている。本報告では、その一例として DBT により実装した実行時ループ検出機構を HPC 分野のプログラムチューニング支援に応用することを取り上げる。特に、アプリケーション・アルゴリズムを実装した MPI プログラムが実際の実行に際しどのようにハードウェアにマッピングされたかということを提示することによりプログラムチューニングを支援可能であるという事例を示す。

本論文の構成は以下の通りである。2 節はで DBT 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 節ではループプロファイリング中にループ反復回数を抽出する手法

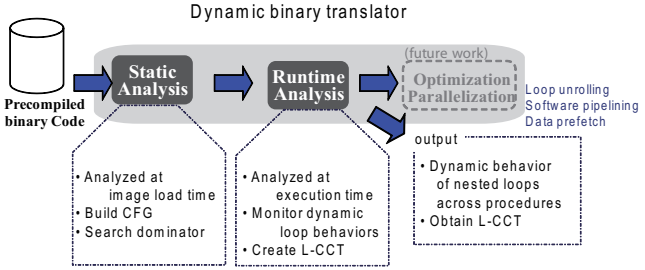

図 2 DBT を用いたループプロファイリング機構

について述べる。4 節では本手法を用いて MPI プログラムがどのように並列化されているかを解析した結果を示す。5 節は結論である。

2. ループ階層構造の抽出

図 2 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）技術を用いてループ構造を検出する手法の概要を示す。本機構により最終的には実際に実行されたループの階層構造やそれぞれのループ反復回数（ループトリップカウント）が出力される。また、コンパイル済みのバイナリコードを入力とするため、シンボル情報等のコンパイラレベルの情報を必要とせず、任意のバイナリコードに対して解析が可能である。

図 2 のライトグレーの部分は DBT システムを表し、濃いグレーの部分は解析のためのモジュールを表している。我々の最終的な目標はランタイムの DBT システムを使ってマルチコア CPU や GPU を用いた超並列処理の性能を生産的かつ効率的に向上させることである。このことを実現するためにはランタイム最適化やランタイム並列化という方向性と、手作業による並列化に有用な情報を提供するという方向性の 2 つがあるが、本報告においては後者について焦点を当てる。

ループ構造を検出する手法の概要を順を追って説明する。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第一のステップとして静的解析を行う。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う。静的解析フェーズにおいてはコントロールフローグラフおよび支配木を構築し Havlak のアルゴリズムにより reducible ループと irreducible ループの双方を検出する[2]。加えて、検出したループの領域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。プログラムの実行を伴わない静的解析にお

Vol.2010-ARC-192 No.10
Vol.2010-HPC-128 No.10
2010/12/16

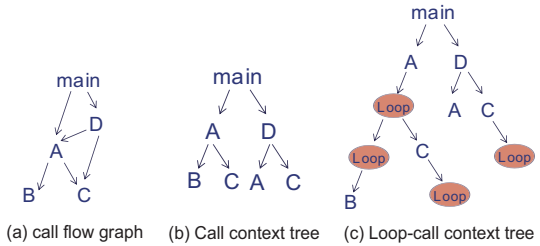

図 3 ループや関数呼び出しフローの表記

いては関数をまたぐ（inter-procedural な）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、これらをプログラムの実行時に抽出する動的解析を行う。

動的なランタイム解析のステップは以下のように動作する。ランタイム解析で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プログラムが実行される。ランタイム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。加えて、ランタイム解析により各ループの反復回数であるトリップカウントをモニタしたりプログラム全体のループ階層構造と関数呼び出しの関係を把握することが可能となる。これらランタイムで得られる情報は最適化に有用である[3)–5)]。

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 **L-CCT (Loop-Call Context Tree)** というデータ構造を構築する。L-CCT はコールコンテキストプロファイリング[6)] にて利用される CCT (Call Context Tree) を拡張し、関数をまたぐループネスト構造を表現できるようにしたものである。

図 3 (a) にコールフローグラフ (CFG) を示す。各ノードは関数に対応している。関数 A は関数 main および関数 D より呼ばれているためコールフローのマージがあることがわかる。このマージにより呼び出しシークエンスに由来するネストの親子関係を正確に把握することができない。図 3 (b) に CCT を示す。CCT は呼び出しシークエンスに由来するフローセンシティブなパスを表現することが可能である。図 3 (c) は L-CCT を示す。L-CCT は CCT にループを示すループノードを追加した表記である。Havlak のアルゴリズムにより検出される reducible ループおよび irreducible ループにおいては、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。ネストしている場合はアウターループが親に、インナーループが子になるようにノードを追加する。互いに素の場合はそれぞれが兄弟となるようにノードを追加する。このような L-CCT によりプログラム実行時に実際に実行されたループネスト構造と関数の位置関係を把握することが可能となる。

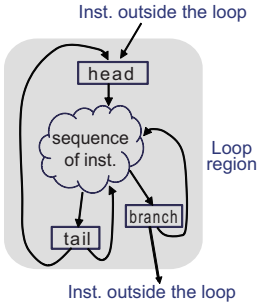

図 4 ループの構造

3. ループ反復回数の抽出

ループ反復における繰り返しの回数であるループトリップカウントはどのようなループ最適化を行うかにおいて非常に重要な指標となると言われている[3)]。また、正確なループトリップカウントを得ることは最適化の判断を適切に行う上で極めて有用であると考えられる。正確なトリップカウントを得るためには、正確ににループ構造を抽出する必要がある。そこで、本節ではランタイム解析のフェーズにおいて各ループの反復回数であるトリップカウントを抽出する方法を説明する。

図 4 に抽象化した典型的な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。

本研究においてはバイナリコードに対して静的解析を行い、コントロールフローと支配木を構成し Havlak のアルゴリズムによりループを抽出している[2)]。Havlak のアルゴリズムでは、すべてのループにおいて head が定義されるため、head の命令が実行される毎にループが反復されているとし、ループトリップカウンタをインクリメントする。また、irreducible ループにおいては 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。

Vol.2010-ARC-192 No.10
Vol.2010-HPC-128 No.10
2010/12/16

4. MPI プログラムの解析

4.1 解析の目的と手法

本研究においては、ソースコードを参照しなくともどのループがどのように並列化されているかを把握できるようになることを目指す。並列プログラムのチューニングにおいてはデータ分割や負荷分散に関するヒントを得ることは非常に重要であり、本報告ではこれらを正確なループ構成を用いて抽出したループトリップカウントを用いて把握することを試みる。

対象とする並列プログラムは MPI (Message Passing Interface) により SPMD (Single Program, Multiple Data) 方式として記述されたものとした。これは、ペタスケール、そしてポストペタスケール時代の超並列プログラムはヘテロジーニアスな分散メモリ型の大規模並列システムが有望視されていることに由来する。本稿の解析においては MPI 環境下で各プロセッサに分散されるプロセス毎に L-CCT の生成やループトリップカウントの解析を行う。したがって、アプリケーション・アルゴリズムを実装した MPI プログラムが実際の実行に際しどのようにハードウェアにマッピングされたかということを観測することが可能となる。

4.2 実験環境

提案するループネスト解析プロファイリングを Pin tool set[7] を用いて実装した。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。

以下に実装の詳細について説明する。本報告にて提案するループネスト解析プロファイリングシステムを SGI AltixXE320 上に構築した。AltixXE320 は 2 基の Intel Xeon E5462 CPU と 8GB の主記憶メモリ、Red Hat Enterprise Linux Server 5.2 にて構成される汎用的な x86_64 のクラスタであり、1 ノードあたり 8 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 MPI 版の IS 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る。コンパイラとして Intel C++/Fortran Compiler version 11.0 に '-O2 -debug full' オプションを用いてバイナリコードを生成した。また、MPI ライブラリとして Intel MPI Library 3.2 を利用した。また、コンパイル時に並列度を 1 および 4 としてそれぞれ実行バイナリコードを生成した。

MPI にて並列化する環境においては各プロセッサに分散されるプロセス毎に L-CCT の生成やループトリップカウントの解析ために Pin を動作させ、プロセスごとに独立したループに関するデータを生成する。MPI による各プロセッサに分散は mpiexec コマンド以下に pin と解析コードとアプリケーションのバイナリコードを指定することにより行った。また、本実験ではすべてのプロセスは 8 コア CPU をもつ 1 ノード内にマッピングした。

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 L-CCT 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た Hot L-CCT を結果の解析に用いた。ここで Hot ノードを全命令実行数の 1%以上を占めるノードとして、Hot 領域は Hot ノードおよび Hot ノードを子孫に持つノードと定義した。Hot L-CCT の可視化は graphviz により行った。

4.3 評価結果

図 5 に is.A の逐次実行した際の Hot L-CCT を示す。丸いノードはループを示し、四角のノードは関数を表す。ノード内部にはループノードの場合にはループ ID、関数ノードの場合には関数名が上段に示され、下段にはプログラム全体におけるそのノードの実行命令数の割合、および括弧内にそのノードの子孫まで含めた実行命令の割合が示されている。また、ノードの右下にはループトリップカウントの平均値、およびそのループが出現し実行された回数（ループ外からそのループに新しく流入した回数）を示す。ループトリップカウントは各ループの反復回数の総和をループ出現回数で割った値である。

図 5 の結果より、is.A のループ 17 を除くすべてのループは 2^{23} 回 (8 メガ回: 1M=1048576) だけ実行されていることがわかる。また、ループ 17 は 10 回反復されており、その中で関数 rank を呼び出しているため、関数 rank 内の 3 つのループが 10 回出現していることが観測される。

図 6 に is.A を mpiexec コマンドにより MPI にて 4 並列で実行した際の Hot L-CCT を示す。mpiexec コマンドにて pin および解析コードとアプリケーションバイナリコードを実行すると各プロセスごとに解析結果が出力されるが、本報告ではそのうちの 1 つについての Hot L-CCT を示す。結果より、ループ 17 を除くすべてのループの反復回数が 4 分割されていることがわかる。ループ 16、22、24 に関しては 2^{21} 回 (2 メガ回: 1M=1048576) だけ実行されていることが観測される。また、ループ 18 と 33 に関してはおおよそ逐次実

Vol.2010-ARC-192 No.10
Vol.2010-HPC-128 No.10
2010/12/16

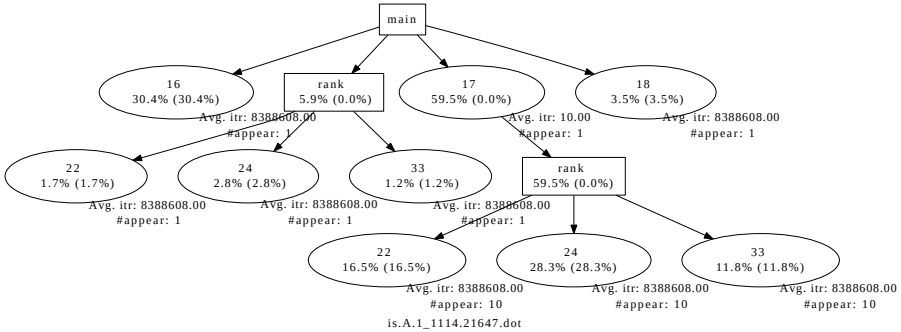

図 5 is.A の逐次実行した際の Hot L-CCT

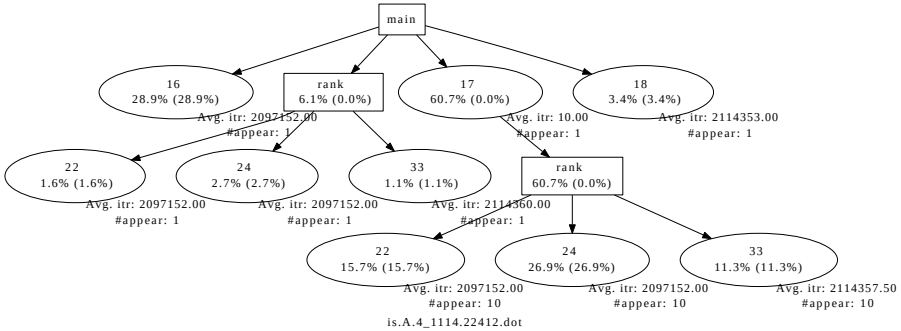

図 6 is.A の並列実行（並列度 4）した際の Hot L-CCT

行のループ反復回数の 4 分の 1 に相当し、他の 3 つの解析結果にあるループ反復回数の平均の総和をとると逐次実行における反復回数と一致することが確認された。また、ループ 17 は分割されず、すべてのプロセスにおいて重複して実行されていることが観測された。

以上のように、is.A の 4 並列の MPI プログラムにて並列化されてる部分はループ 16、18、22、24、33 であり、それぞれのループは逐次実行時と比べて 4 分割されて並列実行されることを解析した。次に、これらのバイナリコードに現れるループがソースコードのどの部分に対応するかが興味深い問題となる。しかしながら、Tallent らの研究[8] により議論されているようにコンパイラの最適化やループ変換あるいは関数のインライニングによりによりソースコードに現れるループが必ずしもバイナリコード中に現れるループと一対一に対応するわけではないことに注意しなければならない。

バイナリコードとソースコードにそれぞれ現れるループ構造の対応をとるためにコンパイラの出力するデバッグ情報によりバイナリコードの命令アドレスに対応するソースコードを addr2line コマンドにより出力させた。その結果、並列化されたループ 16、18 はそれぞれ is.c の 972 行目、1077 行目に対応すると出力された。972 行目は関数 create_seq() を、1077 行目は関数 full_verify() を呼び出していることから、これらのループは関数にインラインによるものと考えらえる。同様に並列化されたループ 22、24 は is.c の 584 行目、596 行目に対応し、NUM_KEY 回だけループが反復される for ループであることが出力された。

並列化されたループ 33 は is.c の 739 行目に対応しその直前のループにて生成される j の値だけ反復されるため、ループ反復数は静的には固定されていない。同様にループ 18 に対応する関数 full_verify() の内部においても、ループ反復数は静的には固定されていない変数となる。これらのループは動的に反復数を変えるように記述されているが、ループ 18 と 33 に関してはおおよそ逐次実行のループ反復回数の 4 分の 1 がそれぞれ実行されているということが各プロセスのトリップカウントを解析することにより確認される。

また、Hot ループを子孫に持つループ 17 は is.c の 1009 行目に対応し、'MAX_ITERATIONS' 回だけ反復する for ループであった。この反復回数はソースコード中に#define MAX_ITERATIONS 10 にて定義されており、10 回の固定の反復を行う for ループがループ内部で rank() を呼び出していることをソースコードにおいて確認できた。

以上より、実行バイナリコードから実行時に動的な関数呼び出しをまたぐループネストやそれぞれのループ反復回数を検出することにより、MPI プログラムがどのように並列化されているかを把握することを支援可能であるということが示されたといえる。

5. 関 連 研 究

ループ構造はプログラムの並列化・最適化において重要な役割を果たすため様々な論文において議論されてきている。実行時にループ領域を認識することを目的として後方分岐命令に着目する研究は広く行われてきた[9),10)]。しかしながら、後方分岐命令は必ずしもループを形成するわけではないため、正確にプログラム中からループを検出しようとするとに Havlak のアルゴリズムのようにコントロールフローグラフから支配木を形成する必要がある。Moseley らも実行時プロファイリングにより動的な基本ブロックの繰り返しを監視することによるループ検出を試みている[11)]。しかしながら、彼らの論文では並列化されたプログ

Vol.2010-ARC-192 No.10
Vol.2010-HPC-128 No.10
2010/12/16

ラムにおけるループプロファイリングによる並列部分の推定については議論されていない。

HPCToolkit[8] では我々の研究と同様に Havlak のアルゴリズムを用いてバイナリコードからループを静的に解析している。しかしながら、ループの動的なトリップカウントの解析や L-CCT の構築は行っていないため、並列化がどのように行われているのかを知るためにはソースコードを読むことが必要となる。

Beyls らは loop call tree により関数呼び出しやループ実行、ループ反復を階層的に表現し、局所性の最適化に活用することを提案している[4]。また、Rul らもプログラムの関数呼び出しやループを表現することを自動でプログラムをパーティショニングに活用することを提案している[12]。しかしながら、これらの論文ではプロファイリングにより関数やループネストのツリーがどのように構築されるかということについて具体的な記述はない。

既存の逐次コードをマルチコアプロセッサ上で並列に動作させるというランタイム並列化に関して様々な技術が提案されている[13)–15)]。ランタイム並列化は大規模・複雑化するアプリケーションに関してもユーザーの明示的な並列性記述なしに高度に並列化することが可能であり、生産性な並列化手法であるといえる。ランタイム並列化を行う上で重要な動作として、並列化の対象として適切な部分を推定することがある。本ループネスト解析手法はデータ依存解析[16] と共に、ランタイム並列化の基盤技術となる。加えて、同一の ISA への変換を行う DBT システムは変換のためのオーバーヘッドを減らすことができればランタイムに得た情報に基づき動的バイナリ最適化や動的バイナリ並列化へも応用可能と考えらている。

6. 結　　論

本稿では、動的バイナリトランスレーション技術を用いた実行時のプロファイリングにより MPI プログラムがどのように並列化されているかを解析する手法を提案した。実行時プロファイリングにおいては実行バイナリコードから実行時に動的な関数呼び出しをまたぐループネストやそれぞれのループ反復回数を検出することにより、アプリケーション・アルゴリズムを実装した MPI プログラムが実際にどのようにハードウェアにマッピングされたかということを解析し、これらがプログラムチューニングに対してプログラムの理解を助けるという面で支援できるということを示した。

参 考 文 献

1) 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.
2) Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, Vol.19, No.4, pp. 557–567, 1997.
3) Youfeng Wu, Mauricio Breternitz, and Tevi Devor. Continuous trip count profiling for loop optimizations in two-phase dynamic binary translators. In *INTERACT'04*, pp. 3–12.
4) Kristof Beyls and ErikH. D'Hollander. Refactoring for data locality. *Computer*, Vol.42, No.2, pp. 62–71, 2009.
5) Mary Hall, etal. Loop transformation recipes for code generation and auto-tuning. *LCPC 2009, LNCS*.
6) 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.
7) Chi-Keung Luk, etal. 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.
8) NathanR. Tallent, JohnM. Mellor-Crummey, and MichaelW. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI2009*, pp. 441–452.
9) J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *HPCA1998*, p.14.
10) Yukinori Sato, Ken-ichi Suzuki, and Tadao Nakamura. Run-time detection mechanism of nested call-loop structure to monitor the actual execution of codes. In *Proceedings of First International Workshop on Software Technologies for Future Dependable Distributed Systems*, pp. 184–188, 2009.
11) Tipp Moseley, etal. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th international conference on Computing frontiers*, pp. 143–152, 2007.
12) Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Towards automatic program partitioning. In *Proceedings of the 6th ACM conference on Computing frontiers*, pp. 89–98, 2009.
13) Michael Chen and Kunle Olukotun. TEST: A tracer for extracting speculative threads. In *Proceedings of the international symposium on Code generation and*

Vol.2010-ARC-192 No.10
Vol.2010-HPC-128 No.10
2010/12/16

optimization, pp. 301–312, 2003.

14) Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 158–167, 2006.

15) Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 185–196, 2008.

16) 佐藤幸紀, 中村維男. 実行時データ依存解析によるループ階層構造に着目した並列性抽出. 情報処理学会研究会報告 Vol.2009-ARC-184 No.8, pp. 1–8, 2009. 2009 年 並列/分散/協調処理に関する『仙台』サマー・ワークショップ SWoPP2009.