ハイパフォーマンス
コンピューティング 58－1
（1995．10．18）

HTGの最適化手法への適応に関する考察

丸川 一志*, 城 和貴*, David Craig**, Constantine Polychronopoulos**, 福田 晃*

* 奈良先端科学技術大学院大学 情報科学研究科
** Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

我々の目標は、並列計算に関する研究のための環境としての、また基盤としての自動並列化コンパイラを構築することである。このコンパイラは、最適化パスの独立性と、それらを自由に適用できる実装という特徴を持つ。そのようなコンパイラを構築するためには、中間表現をベースとして構築することが重要で、特に、中間表現とその利用手続きとを統一することが必要となる。本稿では統一された中間表現としてイリノイ大学 CSRD で提案された HTG を用いることについて検討を行う。また、最適化パスの実装を絡めて、利用手続きについての検討も行う。

A Compiler Implementation Framework with the Hierarchical Task Graph

Kazushi Marukawa*, Kazuki Joe*, David Craig**,
Constantine Polychronopoulos**, Akira Fukuda*

* Graduate School of Information Science
Nara Institute of Science and Technology
** Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

It is the goal of our project to develop a new compiler as an infrastructure for research on parallelizing and optimizing compilers. The syster is constructed by passes which depend only intermediate representation. Therefore each pas was applied any time independent of other passes. The intermediate representaion for suc system must be unified to keep consistency among passes. This paper argues that the HTG (Hierarchical Task Graph), which was initiated by CSRD, is suitable as a universal intermediate representation for such system, and presents an implementation framework with it.

1 はじめに

自動並列化コンパイラの目的は、並列性について考慮せずに記述されたアプリケーションのソースコードから、単に高速に実行できるコードを作成するのみならず、複数の計算機や演算ユニットを利用して、高速に並列実行できるコードを生成することにある。

高速に逐次実行できるコードの生成を目的とする、変数間の依存を元にした共通項の削除や定数の展開等のアルゴリズムを、単純な最適化手法と呼ぶ [1]。また並列実行できるコードの生成を目的とする、並列性の検出やコードの並べ替えによる並列性抽出等のアルゴリズムを、並列化のための最適化手法と呼ぶ [6]。過去、こうした最適化手法として様々なアルゴリズムが提案されてきた。

最適化手法は、上述の目的にかなうコードを生成するべく、ソースコードから生成した中間表現上で処理を行う。この中間表現としても、最適化手法のアルゴリズムに応じた様々な表現が提案されてきた。

ここで自動並列化コンパイラ自体の構成について検討しよう。コンパイラの実装においては、コンパイラの仕事を目的に応じて分解し、組み合わせるという方法論がよく利用される。この分解した個々の部分をフェーズと呼び、それらを組み合わせて実装したものをパスと呼ぶ [1]。今日の自動並列化コンパイラでは、それが利用する様々な最適化手法のアルゴリズムを最適化パスとして実装し、それを組み合わせたものに中間表現を流す、バッチ処理のような構成 (図 1) を取ることが多い。パスによっては、新しい中間表現を追加するために、その組み合わせ方が固定されてしまうこともある。

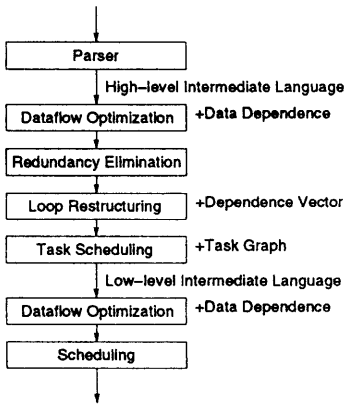

図 1: 従来のコンパイラの構造

このような実装では、パスの適用順序にかなうアプリケーションでしか、良い結果を得られないことがある。これは元々最適化問題が NP 完全の問題であり、それを、多数のヒューリスティックな最適化手法の組み合わせによって解こうとしているために生じる問題とも言える。

我々は、自動並列化コンパイラの研究において、複数の最適化手法を何度も適用できる環境としてのコンパイラが重要な役目を果たすと考えている。そのためには、様々な最適化パスを、ひとつの中間表現上で実装しなくてはならない。これは、従来の最適化手法をベースにした実装とは異なる、中間表現をベースにした実装と言えよう。我々は、そうして実装した様々な最適化パスを、中間表現に対して自由に適用できる、イベント処理的な構成 (図 2) のコンパイラを構築する予定である。

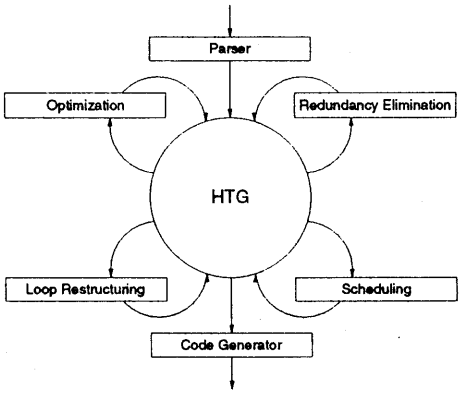

図 2: 目標とするコンパイラの構造

本稿ではその構築のベースとなる中間表現として HTG (Hierarchical Task Graph) [5, 4] を採用した場合の利点や欠点について、またその実装に関して議論を行う。本稿 2 節では関連研究を紹介する。次に 3 節では、HTG を採用する理由について議論する。更に実装に関して、4 節、5 節に最適化パスの実装とも絡めて議論する。6 節では、まとめと今後のコンパイラの実装予定の説明を行う。

2 背景

本節では、中間表現とコンパイラの実装についての関連研究を紹介する。

最も一般的な中間表現には、ソースコードをそのまま中間表現とした抽象構文木 AST (Abstract Syntax Tree) や、AST を基本ブロックごとに分割し弧で結ぶことによってコントロールフローを表わした CFG (Control Flow Graph) 等がある [1]。また近年提案されている種々の最適化手法では、それに加えてコントロール依

存を表わすCDG (Control Dependence Graph) [3]や、データ依存を表わすDDG (Data Dependence Graph) [9, 6]等の中間表現も用いられている。また複数の中間表現をあわせた表現として、CDDG [5]等も利用される。

しかし、このCFGとASTをベースにした中間表現には、いくつかの問題が存在する。まず、循環グラフであるために依存の判別が行えない。そして、依存情報が細かすぎるため粗粒度の最適化に向かない。また、複数のデータ構造を組み合わせて実現されているため、それらを操作する手続きの実装が繁雑になる。

Girkarらは文献[5, 4]において、HTGという中間表現を提案した。HTGでは、CFGの強連結成分を別のグラフとして生成し、それを元のグラフ中から参照することによって、複数のグラフを用いた階層を構築する。これによって、個々のグラフが非循環であることを保証し、同時に階層毎に依存情報をまとめることによって、粒度を選択できることを示した。また、階層によって基本ブロック等をHTGだけで表わすことができる。こうして前述の問題を解決した。

HTGの概要は付録に紹介する。

コンパイラの実装に関しても多くの研究が行われている。インタラクティブなコンパイラの実装としては、自動並列化コンパイラであるParafrase-2 [7]やSUIF (Stanford University Intermediate Form) [8]等がある。これらは、最適化パスを適用する順序を指示できるため、新しく実装した最適化パスの適用結果を分析するのに適した、研究用のコンパイラと言えよう。統一された中間表現を利用したコンパイラの実装としては、自動並列化コンパイラであるSUIFやPolaris [2]等がある。これらは、研究の基盤としてのコンパイラとして位置付けられている。その目標は、中間表現とそれを利用するための手続きとを明確に定義することによって、最適化パスから利用できる統一された中間表現というものを提供し、その上で様々な最適化アルゴリズムを実装することにある。

ここで、これらのコンパイラの中間表現について検討しよう。Parafrase-2ではASTをベースにした中間表現を利用している。しかし、複数の中間表現を必要に応じて追加しているため、それらの間での一貫性に問題がある。SUIFでは、アセンブラレベルのASTと、条件分岐やループ、配列参照という特定の言語に依存しないソースレベルのASTからなる統一された中間表現を利用している。それ以外の中間表現は、必要に応じて生成し利用している。Polarisは、FORTRANのASTをベースにした中間表現を利用している。

これらの関連研究から、基盤としてのコンパイラを構築するには、特定の中間表現を対象に様々な最適化手法を実装し一貫性を保つという方法論が良いと考えられる。そのためには、中間表現だけでなく、それを利用する手続きをも統一することが課題と言えよう。

3 中間表現

中間表現に様々な表現があることは2節に述べた通りである。これらは個々の最適化手法において、より便利な表現として考え出された表現である。また2節において我々は、実装のためには、統一された中間表現とその利用手続きとが必要であるという結論を得た。

この統一された中間表現は、多様性を持った中間表現でなくてはならない。具体的には、最適化パスが利用する様々な中間表現を表わすことができ、またソースコードレベルやアセンブラレベルの中間表現等の独立した中間表現も表わすことができる必要がある。

本節では、統一された中間表現としてHTGが適するかどうか議論する。

3.1 手法に適した中間表現

多くの最適化手法は、その最適化の際に、最低限CFGとASTを、そしてそのAST上でのデータ依存とを必要とする。並列化のための最適化では、更にDDGやCDGも必要とする。また粗粒度の最適化手法では、タスクグラフ等の中間表現を必要とする場合もある[1]。

HTGは付録に示した通り、CFG、CDG、DDGを保持し、それらの間で一貫性を保っている。また、それぞれのグラフを階層毎に分割し、依存情報をまとめることにより、異なる粒度にも同時に対応できる。

また、基盤としてのコンパイラを実装するには、個々の最適化パスにおいて、最適化を行う単位として中間表現そのものが利用できることが望ましい。従来のコンパイラでは、関数やループ等のASTに即した構造を単位として利用していた。我々は、HTGを中間表現として用いた場合には、階層化された個々のHTGを単位として利用できると考える。それによって、より一般的な構造を単位とした最適化の実装が行える。

[1]本論では、特定の最適化手法で特に必要になる、特殊な中間表現については考えない。そうした中間表現は、必要に応じて生成と削除を行うことになる。

我々は、HTGが最適化手法に必要な中間表現を保持しており、目的にかなうと考える。最適化パスは、HTG中から必要な中間表現を選択し利用することによって、HTGを多様性のある中間表現として用いることになる。

ただし、中間表現を統一した場合には、その中間表現において重複する情報の一貫性を保つ必要が発生する。そのため、HTGが統一された中間表現に適するかどうかを考えるためには、HTG内の重複する情報や、その一貫性を保つのに必要となる時間などについても考察を行わなくてはならない。本稿では、5節において実装と絡めてそれらについて議論する。

3.2 独立した中間表現

コンパイラの中間表現は、そのASTの種類に応じて、ソースレベルの中間表現とアセンブラレベルの中間表現とに分けられる[1]。多くのコンパイラでは、前者を後者に変換するパスを用意し、その前後で別々の中間表現を利用した最適化を行う。

我々は、この両者を統一しなくてはならない。SUIFでは、最初からアセンブラレベルのASTを用いて統一を行っていた。ただし、高レベルの依存解析等のために、配列の参照等のソースレベルのASTも一部利用できるよう用意されている。しかしそれだけでは、高レベルの依存解析が十分に行えないという問題がある。

HTGでは、分岐やループ等の構造をグラフ自体によって表わせる。そのため、ASTでは個々のステートメントを表わすだけでよい。上述した変換パスにおいては、ソースレベルのASTを含むSIMPLE節を、アセンブラレベルのASTから構築される基本ブロックに相当するHTGに変換する。我々はASTとして、CやFORTRAN等の式と、アセンブラの単純な式を表わすことのできる、統一されたASTを用意する予定である。これによって、高レベルの依存解析を十分に行える、統一された中間表現を構築できると考える。

HTGは、まずソースレベルの複雑な計算式を含んだ中間表現として生成され利用される(図3のa)。そしてソースレベルでの最適化が終わったと判断したら、ソースレベルのASTを含むSIMPLE節をアセンブラレベルのASTを含む基本ブロックからなるCOMPOUND節に変換し、その後はアセンブラレベルの中間表現として利用するわけである(図 3のb、c)。

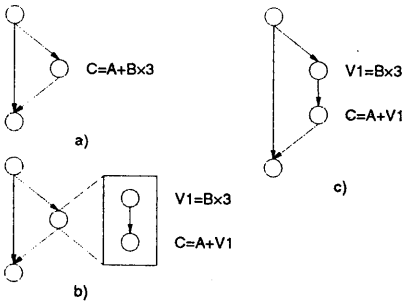

図3: ASTの違い

4 C++による実装

我々の目標は、実装の面から検討すると、HTGという単一のデータ構造をコンパイラ全体で利用することによって、プログラミングの複雑性を上げることなく、高い拡張性を保持した研究の基盤としてのコンパイラを構築することである。

高い拡張性を保持したまま、データ構造が不当に扱われる危険性を排除するために、我々はC++を利用して実装を行うことにした。HTGをC++のクラスとして定義し、同時に中間表現に操作を施す手続きも用意する。拡張時には、データ構造と手続きの両者を、一貫性を保ったまま拡張する。我々は、このような実装によって、一貫性の欠如や不当な操作を排除できると考えている。

5 HTGの実装

実装について考察するには、データ構造を操作する手続きを明確に定義しなくてはならない。それには、個々の最適化パスが、データ構造をどう扱うかということについて考察する必要がある。本節ではまず、HTGを中間表現として用いた場合に必要とされる、それら手続きについて考察する。また、それぞれの手続きについて、実装上の問題点について議論を行う。

単純な最適化手法は、各ステートメント間の依存を解析し、必要のないコードや変数を削除する。並列化のための最適化手法は、各粒度における依存を解析し、並列性検出や、また必要に応じてコードを並べ替え並列性抽出を行う。両最適化手法が行う作業は、その実装に注目することによって、3つに分けることができる。すなわち個々のステートメント間の依存解析、節点間の依存解析、それら依存解析を元にした節点の並べ替えである。

上記作業のうち、依存解析を行うには、HTGをCFGに沿って探索できる手続きと、そうして探索した個々の節点間の依存を検査できる手

続きが必要になる。また並べ替えには、並べ替えという操作だけではなく、変更後の一貫性を保つための操作も含む手続きが必要になる。

5.1 探索と依存検査

HTG における探索は、CFG に沿って深さ優先に行われる。ある節点が、他の HTG を含んでいる場合には、その HTG を再帰的に探索する。また粒度を調整するために、探索の深さを尺度として用いて、再帰的に探索するかどうかを決定する。

HTG における節点間の依存調査は、依存に注目すれば、コントロールに関する依存の調査とデータに関する依存の調査に分けられる。また、節点に注目すれば、SIMPLE 節間の依存調査と、特定の HTG 内の節点間の依存調査、そして別々の HTG 内の節点間の依存調査に分けられる。

コントロールに関する依存は、個々の HTG 毎にまとめられている。そのため、特定の HTG 内の節点間の依存は即座に調査できる。それ以外の場合には、まず HTG の階層を上がることによって共通の祖先である HTG を見つけなくてはならない。その後、共通の HTG 内でそれぞれの祖先間の関係を調べることによって依存を調査できる。

データに関する依存には、個々のステートメント間の依存を表わす大域的なものと、それを階層間でまとめたものが存在する。SIMPLE 節は定義より個々のステートメントを表わすため、これらの間の依存は直接調査できる。また、個々の HTG にはそのグラフ内の依存がまとめられているため、特定の HTG 内の節点間の依存も即座に調査できる。しかし、別々の HTG 内の節点間の依存調査には、多くの時間が必要になる。その場合には、まず共通の祖先である HTG を見つけなくてはならない。その後、共通の HTG 内でそれぞれの祖先間の関係を調べることによって依存の可能性を調査できる。更に詳しく調査する場合には、次々階層を下りながら階層間の依存の可能性を調べていくことになる。

これらの考察において、処理時間的に最も問題になると思われる手続きは、別々の HTG 内の節点間におけるデータ依存の調査である。しかし、従来特定の粒度の依存調査を行うためには、まず新しいグラフを構築する必要があった。そのためこの問題は、HTG の実装においてどれだけ効率良く実装するかという問題として捉えるべきであろう。また、SIMPLE 節間のコントロール依存の調査も、階層を上がる処理が必要な問題のある手続きである。しかし、階層のない CFG でも同様に CFG を辿る必要があるため、特に HTG に限った問題ではないと考える。

ただし、それらの手続きの実装時には、データ構造等について熟考し、可能な限り高速な処理を実現したいと考える。

5.2 一貫性

並べ替えを行う手続きには、並べ替え後の一貫性を保つ操作も含めなくてはならない。保たなくてはならない一貫性は、CFG と CDG 間の一貫性と、DDG の階層間でまとめてある依存の一貫性に分けられる。

CFG と CDG 間の一貫性は、CFG を変更すると同時に CDG に及ぶ影響を調べ、必要に応じて CDG を変更することによって保つ。その場合、CFG も CDG も階層によって分断されているため、変更された HTG 内だけで調べればよい。DDG 間の一貫性は、DDG を変更すると同時に上位階層に対する影響を調べ、必要に応じて上位の DDG を変更することによって保つ。その場合、階層を上がりながら上位の DDG に影響があるかどうかを調べることになる。影響がなければ、DDG の一貫性を保つ操作はそこで終了する。

CFG と CDG 間の一貫性を保つ操作は、特定の HTG 内で閉じているため簡単に処理することができる。従来は、必要になる度に CDG を CFG から生成していた。CFG の変更は集中的に行われることも多いため、場合によっては CDG を新たに生成するほうが良いかもしれない。これは今後の実装における問題点と言えよう。

また、DDG 間の一貫性を保つ操作は、従来の中間表現では必要がない操作であった。つまり、最も問題になる操作であると言える。そのかわりに、我々は DDG を階層間でまとめることができ、それによって自由な粒度での依存解析を行える。

ここで、DDG の操作と調査について検討しよう。DDG の操作は変数の変更や配列の変換によって生じる。DDG の調査は並列化のためのすべての手法で必要になる。また、まとめられた DDG がない場合に同様の処理を行うには、関係するすべてのステートメント間で依存調査を行う必要がある。我々は、調査を行う回数の方が多いと考え、それに適した中間表現を選択した。実装時には、最適化パスにおいて操作と調査がそれぞれ何回行われるか等の解析等を行いたいと考えている。

6 まとめ

本稿では、研究の基盤として利用できる自動並列化コンパイラについて考察を行った。その実装には、統一された単一の中間表現が必要であると考えられる。そうした中間表現として、HTG を選択することについて議論した。

我々は、HTG のような階層のある中間表現が必要であり、それを利用することについても処理時間の増大以外には実装上問題がないという結論を得た。

今後、本稿で議論したことを踏まえ、HTG を利用した研究の基盤としてのコンパイラを構築する予定である。その作業は、Parafrase-2 [7] という、フロントエンドだけ実装されている自動並列化コンパイラ上で行う。この上に、HTG をもとにしたバックエンドを構築し、その後既に AST 上で実装されているフロントエンド中の様々な最適化パスを、徐々に HTG 上に移行していく予定である。

付録 HTG

HTG は、節点と弧からなる [5, 4]。節点は 5 種類に分類され、それぞれ START、STOP と呼ばれるそれぞれ HTG の入口と出口になる特殊な節点、SIMPLE と呼ばれる単純な節点、COMPOUND と呼ばれる別の HTG を内包する節点、そして LOOP と呼ばれるループのボディを内包する節点となる。

このうち SIMPLE 節は、個々のステートメント等を表わす AST を含む。また、COMPOUND 節と LOOP 節によって、他のグラフを含む階層的な構造を構築する。

そしてそれらの節点を control flow edge、control dependence edge、data dependence edge 等の種々の弧で結ぶことによって、それぞれ CFG、CDG、DDG といったグラフ構造を構築する。図 4に CFG を HTG から抜き出した例を挙げる。

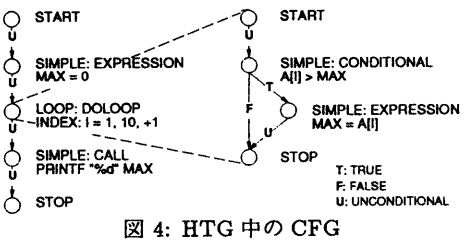

図 4: HTG 中の CFG

これらのグラフは、HTG の階層によって異なる取扱いを受ける。HTG は CFG の強連結成分を元に分解されて階層化されるために、CFG は階層毎に分断されて保持される。また CDG もそうして生成した CFG から生成されるために同様に分断される。

一方 DDG は階層を跨がって AST 内の変数や配列等の symbolic analysis に基づいて保持される。まず最も基本的な data dependence edge が上記 SIMPLE 節間に構築され、次にこの基本的な edge を HTG 毎にまとめた依存情報が保持される。

参考文献

[1] A. V. Aho, R. Sethi, and U. J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] W. Blume, R. Eigenmann, K. Faigin, et al. Polaris: Improving the effectiveness of parallelizing compilers. In *Proc., 7th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 141–154, August 1994.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[5] M. B. Girkar. *Functional Parallelism: Theoretical Foundations and Implementation.* PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1991. Tech. Rep. TR-1182.

[6] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[7] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, et al. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In *Proc., Internat. Conf. on Parallel Processing*, pages 472–492, 1989.

[8] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. In *Proc., 4th Internat. Workshop on Languages and Compilers for Parallel Computing*, 1991.

[9] R. A. Towle. Control and data dependence for program transformations. Technical Report UIUCDCS-R-76-788 (Ph.D. thesis), University of Illinois, Urbana-Champaign, 1976.