プログラミング 17－4
（1995. 1. 23）

統一的中間表現を用いた自動並列化コンパイラの実装
— ソースコードから統一的中間表現への変換 —

田中栄治† 中西恒夫†† 城和貴††† 山下雅史††††
(† 広島大学大学院工学研究科 †† 奈良先端科学技術大学院大学情報科学研究科
††† 和歌山大学システム工学部 †††† 広島大学工学部)

概要

従来の自動並列化コンパイラで一般的に用いられる中間表現は，タスク間の依存関係のみを表現するのみが多く，データの分割配置ならびに転送の最適化手法において重要なデータフローの情報が明示的に表現されていない．そのためこれらの最適化手法は中間表現から遊離した形で処理されることが多く，これは一貫性のとれた並列化・最適化を難しくしている．

本研究で実装している自動並列化コンパイラ Narafrase では，データフロー情報を明示的に表現する中間表現 DPG を統一的中間表現，すなわち全並列化・最適化手法の共通の中間表現とし，一貫性のとれた並列化・最適化の実現を図る．本稿では，ソースコードから DPG への変換パスの実装について報告する．

Implementation of a Parallelizing Compiler with a Universal Intermediate Representations
— Translating of Source Codes into Universal Intermediate Representations —

Eiji Tanaka,† Tsuneo Nakanishi,†† Kazuki Joe,††† and Masafumi Yamashita††††
(† Graduate School of Faculty of Engineering, Hiroshima Univ. †† Graduate School of Faculty of Information Science, Nara Institute of Science and Technology ††† Faculty of Systems Engineering, Wakayama Univ. †††† Faculty of Engineering, Hiroshima Univ.)

Abstract

Intermediate representations used in present parallelizing compilers do not express data-flow informations explicitly required by optimization techniques for data partitioning, distribution, or transfer but data dependences between tasks. Thus those optimization techniques are performed outside intermediate representations and it makes consistent parallelization or optimization difficult.

Narafrase, which is a parallelizing compiler implemented in a part of our works, aims to perform consistent parallelization or optimization by using a common intermediate representation among all parallelization/optimization techniques implemented in itself. We use the DPG as the common intermediate representation, which represents data-flow information explicitly. In this paper we report implementation of the translation pass which constructs a DPG from a given source code.

1 はじめに

自動並列化コンパイラは与えられた逐次プログラムを並列プログラムに自動的に変換するツールであり, 過去20年にわたって研究用あるいは商用の自動並列化コンパイラの研究開発が行なわれてきた. 特に研究用のものは, 並列性の抽出に関する研究は概ね終了しており, ここ数年のうちに商用化されることが予想される.

集中共有メモリ型マルチプロセッサシステムにおいてはプロセッサ間の通信オーバーヘッドはデータの配置に依存することがないため, これらのマルチプロセッサシステムを対象とする過去の自動並列化コンパイラ, ならびにそれらに実装される並列化あるいは最適化手法は, データの配置についてそれほど重視されることはなかった. しかしながら, 分散メモリ型のアーキテクチャを採る大規模マルチプロセッサシステムの出現, ならびにプロセッサとメモリ, あるいは相互結合網のスピード格差の拡大に伴い, 近年の自動並列化コンパイラでは, データの分割配置ならびに転送を考慮し, 並列化・最適化を行うことが必須となっている. また, いわゆる依存グラフは, データの分割配置ならびに転送の最適化がさほど重要でなかった集中共有メモリ型マルチプロセッサシステム時代の自動並列化コンパイラの中間表現である. 依存グラフにはデータの分割配置ならびに転送の最適化で必要なデータフロー情報が明示的に表現されておらず, これらの最適化は中間表現から遊離された形でなされることが多い. 我々の研究では, プログラム中の変数ならびにそれらへのアクセスを節点あるいは枝として表現するように依存グラフを拡張した有向グラフによる自動並列化コンパイラ中間表現, データ分割グラフ (Data Partitioning Graph: DPG) を提案している. DPG を自動並列化コンパイラの並列化・最適化手法の共通の中間表現, すなわち統一的中間表現として用いることにより, データの分割配置ならびに転送の最適化手法に対応し, さらに依存グラフを中間表現とする既成の並列化・最適化手法との統合を図る. 中間表現の共通化, さらには並列化・最適化手法の統合により, これらの手法の干渉の問題を解決することが期待できる.

上述の着想の有用性を検証するべく, 我々は DPG を統一的中間表現とする自動並列化コンパイラ Narafrase の実装に着手した. 本稿では, Narafrase の実装のうち, ソースコードからその中間表現への変換パスの実装について報告する.

2 Narafrase

2.1 概要

Narafrase は自動並列化コンパイラであり, 次の特徴を持つ.

1. 並列化・最適化の核となる中間表現として, データ分割グラフ (Data Partitioning Graph: DPG)[3] の拡張である, 階層データ分割グラフ (Hierarchical DPG: HDPG)[4] を採用する. これは, 従来の中間表現に見られた制御依存グラフやデータ依存グラフに加え, データのアクセス状況をグラフ化したものである.

2. すべての並列化・最適化手法は, 上で述べた中間表現に基づいて並列化・最適化を試みる. このことにより, 数多くの並列化・最適化手法の中から, 与えられたプログラムの性質と稼働対象のアーキテクチャに最も適した手法を選ぶ, 性能指向型自動並列化コンパイラが実現される.

3. 中間表現と並列化・最適化パスとの間に, インターフェースを持つ. このことにより, 将来中間表現を拡張した場合に, それまでの最適化手法を再実装することなしに稼働させることができる.

Narafrase の内部構成は, 図1で示されるように, 中間表現を中心に4つの部分からなる.

解析部
ソースコードを統一的な中間表現に変換する部分. ソースコードとしては, Fortran 77 を想定している.

中間表現インターフェース部
中間表現と並列化・最適化部分とのインターフェース部分. インターフェースは C++ライブラリの形で構成されており, 最適化手法は必要に応じて同ライブラリをリンクする.

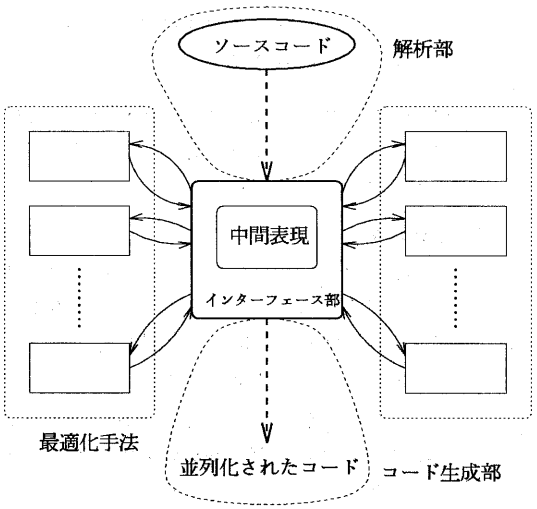

図 1: Narafrase の構成

最適化部

並列化あるいは最適化を試みる部分. 複数のモジュールに分かれており, 一つのモジュールには一つの並列化・最適化手法が実装される.

コード生成部

与えられた中間表現から並列化されたコードを生成する部分. 現段階では並列化されたソースコードを出力する.

3 解析部の実装

ソースコードから本コンパイラで利用する中間表現への変換は, 以下の各段階を経て行なわれる.

1. ソースコードの構文解析
2. 解析結果から制御フローグラフの生成
3. 制御フローから階層タスクグラフ (Hierarchical Task Graph: HTG) の生成 (グラフの階層化, ならびに制御依存枝, データ依存枝の導出)
4. 階層タスクグラフから階層データ分割グラフの生成 (データフロー解析, ならびに HTG との融合)

1 は, Edison Design Group 社の **Fortran Front End** [2] により行なっている. また, 各部分は再利用性を考慮して独立に実装し, データの受け渡しはファイルを用いて行なう.

第 4 節では, 第 3 段階について詳しく述べる.

4 制御フローグラフから階層タスクグラフの生成

第 3 段階では主に以下に示す事を行なう.

- グラフの階層化
- 制御依存グラフの生成

制御フローグラフからはデータ依存関係は分からないため, データ依存グラフはここで述べる生成アルゴリズムとは独立して実装される. そこで, 出力である依存関係はグラフの枝ごとにブロック化されて出力される.

以下では, まず制御フローグラフと階層タスクグラフの定義を行ない, 次に生成アルゴリズムを示す.

4.1 制御フローグラフ

制御フローグラフ (Control Flow Graph: CFG) はプログラムの実行の流れをグラフで表したものである (図 2 参照). 流れの先頭にあるノードを ENTER ノード, 最後にあるノードを EXIT ノード と呼ぶ.

本実装では, ソースコード中の 1 ステートメントがグラフの 1 ノードに対応付けられる.

4.2 階層タスクグラフ

階層タスクグラフ (Hierarchical Task Graph: HTG) は以下に述べる 5 つのグラフからなる複合グラフである. ただし, 階層 CFG 以外の 4 つのグラフは, 階層 CFG 中に入れ子状に存在するそれぞれの CFG ついて定義される.

階層 CFG

CFG をループの入れ子構造に基づき階層化したもの. 各階層は CFG であるが, 閉路は取り除かれる. また, 各階層の CFG には, 流れの

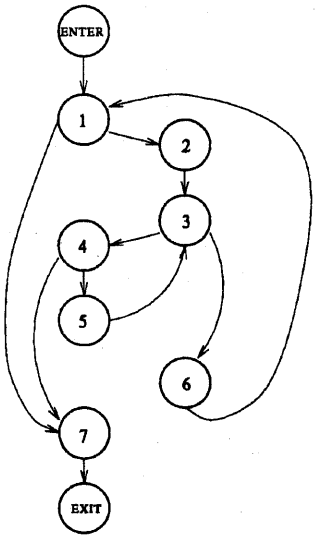

図 2: 制御フローグラフ.

先頭に START ノード, 最後に STOP ノードと呼ばれるノードが加えられる.

図3は図1をループを基に階層化した階層 CFG である. 波線が各階層の境界を表している.

Dominator Tree

CFG において, ノード x がノード yの *dominator* であるとは, START ノードと yを結ぶ任意のパスが x を通過することである.

Dominator Tree (DT) とは, この関係を木で表したものである.

Post-Dominator Tree

CFG において, ノード yがノード x の *post-dominator* であるとは, x と STOP ノードを結ぶ任意のパスが yを通過することである.

Post-Dominator Tree (PDT) とは, この関係を木で表したものである.

制御依存グラフ

CFG において, 次の性質が両方とも満たされるとき, ノード yはノード x に制御依存しているという.

1. y は x の post-dominator ではない.
2. x から y への任意のパスについて, y が, x と y を除くすべてのノードの post-dominator である.

制御依存グラフ (Control Dependence Graph: CDG) はこれらの関係を有効グラフで表したものである.

データ依存グラフ

CFG において, 2 つのノードが同一の共有メモリをアクセスし, かつ, 少なくとも一方がメモリの書き換えを行なっている場合, それらの間にデータ依存があるという.

データ依存グラフ (Data Dependence Graph: DDG) はこれらの関係を有効グラフで表したものである.

4.3 生成アルゴリズム

以下に, 生成アルゴリズムを示す. なお, このアルゴリズムは Girkar[1] に基づいている.

4.3.1 階層 CFG の生成

入力である CFG は, ループを基に階層化する. まず, 階層 CFG を構成するノードを生成する. すなわち, 元の CFG 中のノードに加え, ループを表すノード (ループノード), 各ループに固有の START, STOP ノードを生成する. ただし, ループノードは次のようにして求める.

1. CFG を ENTRY ノードを起点に深さ優先探索し, 枝の終点が探索済であるような枝集合を求める. これを H とする.
2. H の各要素 (枝) について, その終点から始点の間に含まれるノードの集合をそれぞれ求める. このノード集合の集合を L とする. ただし, 同一の終点からなる集合が複数存在する場合には, サイズが最大のものだけを L に含める. この L 中の各要素がループノードの基となる.

次に, 階層 CFG の枝を生成する. これは, 今構築した木を基に生成する. まず, L の各要素に CFG のノードを加えた集合を L' とし, L' の各要素につ

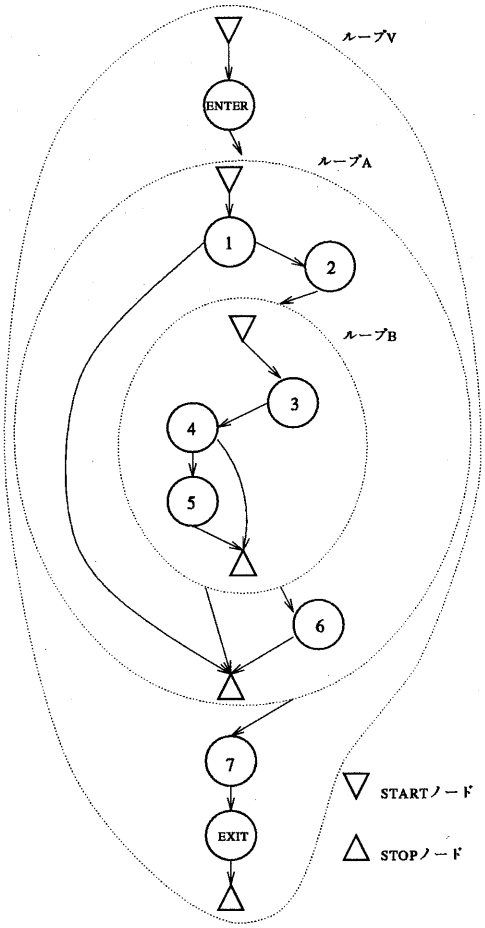

図 3: 階層 CFG.

いてその要素を内包する最小の要素にリンクを張ることによって包含関係を表す木を構築する. この場合, 葉は単一のノードである (図 4参照).

次に, 元の CFG 中のすべての枝に対し次のことを行なう. 枝の両点のノードを, 始点を a, 終点をを b とする. まず, a, b に対し, 木における共通の祖先と, それぞれの祖先までの距離を求める. ただし, 距離とは求めた祖先までの枝の数である. このとき, 今注目している枝が,

- Hに含まれている場合, a と対応する STOP ノードの間を結び, b と対応する START ノードの間も結ぶ.
- Hに含まれていない場合, 2 つの距離の差が 1 の時は, 2 つのノードの間を結び, 2 以上の場合は, それぞれの一つ子孫同士を結ぶ.

結果として階層を跨ぐ枝がある場合には, 始点と対応する STOP ノードの間を結ぶ.

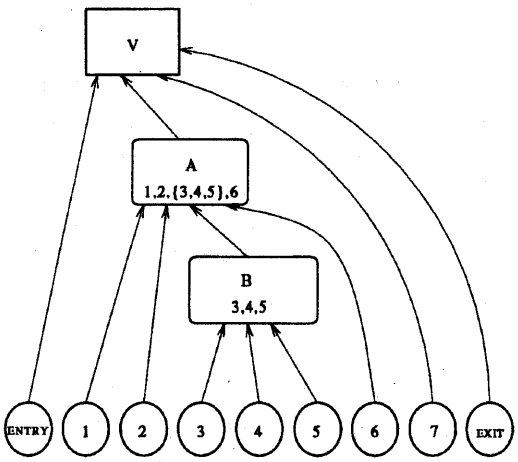

図 4: ノードの包含関係.

4.3.2 Dominator Tree の生成

階層 CFG のすべての部分 CFG に対し, 次のようにして DT を生成する.

まず, CFG のすべての 2 点 x, y について, x が y の dominator かどうか調べる. つまり, START ノードから y までのすべてのパス上に x が含ま

れているかどうか調べる．含まれている場合には x から y へ枝を設ける．次に，得られた枝集合を整理する．つまり x が yの domanator であるときに，zが yの dominator であるような zがないような dominator に対応する枝を選択する．以上より DT が得られる．

4.3.3 Post-Domanator Tree の生成

階層 CFG のすべての部分 CFG に対し，次のようにして PDT を生成する．

まず，CFG の任意の 2 点 x, y について，y が x の post-dominator かどうか調べる．つまり，x から STOP ノードまでのすべてのパス上に y が含まれているかどうか調べる．含まれている場合には y から x へ枝を設ける．次に，DT の場合と同様に得られた枝集合を整理すると，PDT が得られる．

4.3.4 制御依存グラフの生成

階層 CFG のすべての部分 CFG に対し，次のようにして CDG を生成する．

CFG 中の任意の 2 点 x, y について，y が x に制御依存かどうか調べる．つまり，y が x の post-dominator でなく[1]，y が，x から y の間のすべての頂点の post-dominator か調べる．もしそうなら，x から y へ枝を設ける．次に，Dominator Tree の場合と同様に求まった枝集合を整理すると，CDG が得られる．

5 まとめ

本稿では，Narafrase で用いる中間表現を得るために，制御依存グラフから階層タスクグラフ (データ依存グラフを除く) の生成パスの実装を行なったので，これについて報告した．

今後の予定としては，データ依存グラフの生成，また，階層データ分割グラフの生成部の実装が上げられる．

参考文献

[1] Milind Girker and Constantine D. Polychronopoulos, "The Hierarchical Task Graph as a Universal Intermediate Representation," *International Journal of Parallel Programming*, Vol. 22, No.5, pp. 519-551, 1994.

[2] Edison Design Group, Inc., "Fortran Front End," `http://www.edg.com/`

[3] 中西, 城, 福田, 荒木, "DPG: データ分割グラフ", 情処研報 94-ARC-104/94-OS-62, Vol. 94, No. 13, pp. 121–128, Jan. 1994.

[4] 中西, 城, Polychronopoulos, 福田, 荒木, "HDPG: 階層データ分割グラフ", 情処研報 94-HPC-52, Vol. 94, No. 68, pp. 89–94, Jul. 1994.

[1] これは，Post-Dominator Tree を調べればよい