OMT 法による並列化コンパイラ中間言語フレームワークの構築

大森洋一†, 城 和貴†,
福田 晃†, 荒木啓二郎†

†：奈良先端科学技術大学院大学 情報科学研究科

現在、コンパイル時におけるプログラム並列化手法が数多く提案されているが、それらの間の関係は必ずしも明らかでない。このため、各手法の組合せあるいは相互評価ができにくい。そこで、今後の研究を進めるにあたり、共通の条件により比較・検討できる環境が望まれる。この環境は、
1. 実装技術に左右されにくい。
2. 理論的に明解である。
3. 汎用性が高く、安定している。
といった性質を備えているのが望ましい。本稿では、オブジェクト指向設計法の 1 種である OMT を用いて、それぞれの手法を系統的に整理し、中間言語のフレームワーク生成の可能性を検討する。特に、コンパイラの中心となる意味解析木について解析結果を述べる。

A Framework Based on the OMT Method for Intermediate Languages of Parallelizing Compilers

Yoichi Omori†, Kazuki Joe†, Akira Fukuda†and Keijirou Araki†

†: Department of Information Science,
Nara Institute of Science and Technology,
Takayama 8916–5, Ikoma, Nara 630–01 Japan

E-mail: {youiti-o, kazuki-j, fukuda, araki}@is.aist-nara.ac.jp

Various methods to parallelize a program at compile time have been suggested. However, the relation between each method is not methodical. It is required to discuss these techniques on a common standard. Also, it is desired to have attributions, such as
1.Independence from implementation techniques,
2.theoretically clearness, and
3.generality and stability.
The OMT method, an object-oriented analysis one, can systematically classify them from a unified point of view. In this paper, the possibility to construct a framework for intermediate language of parallelizing compilers is examined. The result about a semantics tree is described.

1 はじめに

並列計算分野の研究レベルにおける興味の対象は、まだ性質が良く分かっていない、数千個の処理装置をもつ大規模並列計算機へ移りつつある。しかし、単一プロセッサの性能向上が順調に続いているためもあり、並列計算機が広く普及しているとはいえない状況である。こうした必要性の問題以外にも並列計算機の利用が進まない理由はいくつか考えられる。たとえば、ソフトウェアについて、並列計算機にはさまざまなアーキテクチャが存在し、可搬性のあるプログラムの作成が困難であるといった現状が、並列プログラムの蓄積を妨げている一因と考えられる。このために使いたいプログラムが手に入らず、並列計算機の利用も進まないという循環がある。

これに対して、逐次計算を前提としたプログラムを、並列計算機向けに自動的に最適化できれば、上に述べたような、アルゴリズム設計の前提となる計算モデルにまつわる問題は生じない。また、従来から使い慣れた表現や、多様かつ莫大なソフトウェア資産の蓄積を容易に利用できる。こうした理由により、逐次プログラムに含まれる並列性を自動的に抽出し、並列計算機に効率良くマッピングする並列化コンパイラの実用化が望まれている。特に超並列計算では、プログラムが複雑になりがちであり、開発環境の整備が重要となる。

しかしながら、現在の並列化コンパイラには、いくつもの課題がある。そのうちのひとつに、並列化コンパイラも並列計算機アプリケーションの一種であり、アーキテクチャへの依存と並列化手法の細分化・特殊化がついてまわるという問題がある。すなわち、これまでのコンパイラにおけるプログラム並列化手法は、特定の計算機についての個々に独立した研究であるか、現在の技術では夢のようなモデル上での研究であるものが多く、並列化手法の統一された視点に基づく検討、評価が困難であった。そこで、汎用的な並列化コンパイラ研究を進めるためには、まず並列化手法の相互関係の整理が要請される。

プログラムの分析にはさまざまなソフトウェア工学的手法があるが、オブジェクト指向によるプログラム解析は、幅広いプログラムに適用できることが知られており、こうした目的に適している。くわえて、分析結果の新しい並列化コンパイラへの応用を考慮すると、利用可能な言語やツールが豊富であるという点も優れている。

本研究で扱うモデルは、単一プロセッサモデルとの親和性の高さから、集中共有メモリをもつ並列計算に限定し、また特殊なハードウェアは想定しない。その上で、コンパイラにおける並列化手法を構成するオブジェクトおよびその構造について、中心となる解析木に注目し、ベースとなる構造からいくつかの並列化手法の相互関係を明らかにすることを目標とする。

2 関連研究

2.1 現在の汎用並列化コンパイラ研究

現在、かなりの数の研究グループが並列化コンパイラに関わっているが、汎用的な大規模並列演算を対象とした研究は限られている。

Rice 大学の Keith Cooper らのグループは、Massively Scalar Compiler というプロジェクトで、グラフによるコンパイラの内部表現のカラーリングの問題などを扱っている [2]。これは、同じ Rice 大学の Ken Kennedy らによる、行列の並行演算に関する Fortran の拡張や各種ツールからなる D システム、さらに HPF と強い関連がある [13][15]。

Illinois 大学の Chien らは Concert プロジェクトで、オブジェクト指向に基づいた細粒度並列による大規模並列計算を提案している [3]。

Stanford 大学の flush マルチプロセッサなどを対象とした SUIF compiler グループでは、共有メモリと分散メモリの両アーキテクチャを統一したモデルを用い、行列のユニモジュラ変換による依存解析に基づく並列化を行なっている [16][22]。このグループの Michael Wolfe らによる依存ベクトルに関する研究は、現在の依存解析研究の基礎として、さまざまな方式に影響を与えている [12]。

Carnegie Mellon 大学の Fx プロジェクトは、VLSI 化された LIW チップを用いたシストリックマシンである iWARP などを対象とするコンパイラであり、画像処理や信号処理といった細粒度の並列性を対象とする [17][11]。

David Kuck らは Illinois 大学で Parafrase という Fortran77 から並列化 Fortran への自動変換プリプロセッサを開発した。Parafrase はループのイタレーション間の依存解析やサブルーチン間の依存解析など多くの機能をもち、パスを指定することにより、適用する並列化手法を選択できる [7]。

Kuck & Associates では Parafrase を発展させて、C 言語と Fortran をサポートした KAP を商品化しており、いくつかの商用システムではこれを自社製品に最適化した並列化システムが提供されている [14][4]。

Illinois 大学では現在、複数の入力言語から並列化 C または Fortran を出力する Parafrase-2 を開発している [5]。

またアプリケーションの方面からも、米国 NSF の grand challenge プロジェクトなどにより、Poralis コンパイラが開発されている [21]。

MIT の Prelude system はマルチスレッドによる大規模並列計算環境を指向しており、OS との協調を重視した Autopilot というプロジェクトが進められている [20]。

早稲田大学の笠原らは、マクロデータフローを用い

た Fortran の並列化コンパイラ Oscar を開発している [25][24]。これは、データフローを解析し、並列性のみられるパターンについて、並列化コードに置き換える手法である。

日本で盛んに行なわれたデータフロー計算機に関する研究としては、早稲田大学の村岡らによるグラフエディタとデバッガを中心とする開発環境-晴-などがある [28]。

九州大学の雨宮らは関数型言語に関する並列化コンパイラを作成している [27]。データフロー方式の中でも関数型言語は、参照透明性があるので並列処理で注目されている。

ほかに、シストリック・アレイを対象にしたコンパイル技法の研究 [26] などもある。

また、重点領域研究では超並列計算を対象としており、V 言語は記述されたオブジェクトの関係から並列実行可能な部分を、自動的にハードウェアにマッピングする処理系が検討されている [29]。このように、より高レベルの抽象度から並列性を引き出す研究は、海外でもいくつかなされている [19]。

こうした並列化コンパイラに関してのサーベイは文献 [23][6] に詳しい。

2.2 内部フェーズの観点による分類

さて、上にあげたような並列化コンパイラで用いられている並列化手法は、その背景となる理論も、目標とする機能もさまざまである。そこで、まずコンパイラの内部フェーズの観点からこれらの手法を整理する。

一般に、逐次プロセッサ向けのコンパイラの内部フェーズは次のようになっており、中間表現の前後でフロントエンドとバックエンドに分けられる [1]。

1. 字句解析
2. 構文解析
3. 意味解析
4. 中間コード出力
5. 最適化
6. コード生成

複数の中間コードを定義するコンパイラもあり、その場合は出力対象のマシンからの抽象度が異なる。

一方、自動並列化の手順は

1. 依存解析
2. コード変換
3. 並列制御コードの付加

の各フェーズが必要である。この自動並列化のためのフェーズをどこに組み込むかを検討する必要があり、依存解析は意味解析フェーズの、コード変換は中間コード出力の、実際の並列コードの付加はコード生成フェーズの一部として行なわれる場合が多い。

例えば、文献 [6] には、一般化された並列化コンパイラの構成として、次のようなフェーズが挙げられている。

1. 字句解析から高レベル中間言語の生成

   構文解析、意味解析
2. 手続き呼出しの最適化

   インライン展開、テール・リカージョンの除去
3. スカラー化
4. 高レベルのデータフロー最適化

   定数伝搬、共通部分式の削除など
5. 部分評価

   式の簡単化など
6. 冗長な部分の除去

   不要なコード dead code の除去
7. ループの調整 I

   ループの正規化、ピーリンク peeling など
8. ループの調整 II

   行列のアラインメント調整、ループ次数の低減
9. ループの変換

   ループ交換、スキューイング skewing
10. ループ前後の調整

    ループ融合 fusion、ループ分割 distribution
11. 低レベル中間言語の生成
12. 低レベルのデータフロー最適化 (定数伝搬など)
13. 低レベルの冗長な部分の除去
14. 手続き呼出しの最適化 (パラメータの置き換え)

    呼出の深さの低減 frame collapsing
15. コード生成

    ソフトウェアパイプライニングなど
16. ローカルな最適化

    peephole 最適化
17. アセンブル
18. リンク
19. キャッシュ最適化

これは、プログラムの構成という観点から分類すると、手続きに着目したフェーズの設計にあたる。たとえば、Fortran ソースから並列化 Fortran への変換を行なう Parafrase-2 の場合、字句解析から 4 字組による中間言語へ変換し、4 字組に対するループ変換までは行なうが、低レベルの中間言語は用いず、コード生成を行なう。したがって、上の一連のフェーズからなるコンパイラを制限したものと考えられる。

このように上のフェーズ設計はかなり汎用性が高いのだが、以下の問題点がある。

- 新しいフェーズを追加・変更した場合の影響が見極めにくい。

  モジュール内での不用意なデータ変更を避けるには、インターフェイスの定義に細心の注意が必要であり、また多くのデータ型を用意する必要がある。これは現実には大変な作業である。
- データ構造がフェーズにより細分化されるので、入力から出力までの見通しが悪い。

  フェーズ間で細分化されたデータ構造の対応づけも困難な作業であり、人間にとって意味の分かりにくい構造が生じてしまいがちである。
- いったん実装してしまうと、元の設計とコードの対応がとりにくい。

  特に、最適化されたコードは人間にとって分かりにくい。保守・改良のためには十分なドキュメントやコメントが必須である。
- 同じような手続きを何度も繰り返さなくてはならない。

  コンパイラの作業は似たような手続きが多く、ひとつの手続きを多くのモジュールで共用できる。しかしこのような場合、変更の際の影響が思わぬところにでてしまう可能性が増えるので、内部構造が硬直してしまう。

  逆にそれぞれの手続きを細分化した場合は、一貫性のある変更が難しくなるので、内部構造の管理が厄介になり、これも望ましい状況ではない。

3 OMT による記述

3.1 OMT の特徴

オブジェクト指向設計により、プログラムについて次のような改善が期待できる。

- 理論から実装まで統一したプログラミング・モデルを用いるので、一貫性が向上する。
- モジュール化により、実装・テストが容易になるので、保守性が向上する。
- データフローを中心にインターフェイスが明確になり、拡張性が向上する。
- グラフなどを利用して、モジュール間の関係を視覚的に整理できるので、可読性が向上する。

本研究ではオブジェクト指向分析手法および表記として、ランボーらの提唱する Object Modeling Technique (以下 OMT) を採用した [10]。

OMT はすでにいくつかの分野で応用例があり、用語や表記法が他のオブジェクト指向モデルにも取り入れられてひろく用いられている。

OMT では対象とする問題の分析から設計までを、オブジェクトモデル・動的モデル・機能モデルの 3 つの側面から解析し、図を利用した表記を多用する。

オブジェクトモデルは、各オブジェクト名とその特性を表記したノードと、オブジェクト相互の関係を表すノード間のリンクからなるグラフで記述される。

動的モデルは、システム内の事象および各事象に対するオブジェクトの状態を示したノードと、状態遷移をラベルにもつグラフから構成される状態図で記述される。

機能モデルは、あるオブジェクトの値とそのデータを変換するプロセス、変換に対する制約の流れを示す複数の図からなる。

最終的なオブジェクト図は、オブジェクトモデルに動的モデルと機能モデルを組み込んで得られる。

従来、コンパイラの作成はバッチ処理に分類され、構造化プログラム設計に向いているとされてきたが、自動並列化などの複雑な最適化作業は、データ構造と、データ値に依存した制御フローを示す部分であることが知られており、データの性質に注目したソフトウェアの設計が有効であると考えられる。

これをさらに進めて、多くの並列化手法適用できるような、中間表現による共通の解析木を検討する。

3.2 中間表現の検討

それぞれのコンパイラにおいて、さまざまな中間言語が用いられている。

ここでは、gcc で用いられている中間表現である Register Transfer Language (以下 RTL) を想定する [18]。RTL はアセンブラとほぼ 1 対 1 で対応するシステム独立の中間表現であり、エンディアンや整数の大きさなどシステムに依存する部分は、マシンモードなどのマクロ群によりパラメータ化されている。

```
(plus:m x y)          (not:m x)
(lo_sum:m x y)        (and:m x y)
(minus:m x y)         (ior:m x y)
(compare:m x y)       (xor:m x y)
(neg:m x)             (ashift:m x c)
(mult:m x y)          (lshiftrt:m x c)
(div:m x y)           (ashiftrt:m x c)
(udiv:m x y)          (rotate:m x c)
(mod:m x y)           (rotatert:m x c)
(umod:m x y)          (abs:m x)
(smin:m x y)          (sqrt:m x)
(smax:m x y)          (ffs:m x)
(umin:m x y)
(umax:m x y)
```

図 1: RTL による数式表現

たとえば、数値演算では図 1のような命令が用意され

ている。m はマシンモードであり、整数と浮動小数点の精度、エントリポインタ、整数の組による複素数、ビットフィールドなどを区別する。

gcc では最適化処理の多くを RTL に対して行なっており、十分な記述能力があると判断した。ただし、現在の gcc はマルチプロセッサをサポートしておらず、命令レベルより粒度の大きい並列性が表現できないので、分岐およびループについてフローグラフを利用する [1]。これは fork-join などより直感的に分かりやすく、同時に OMT の表記との親和性が高いからである。

この結果、入力となるプログラム言語や出力対象のシステムに依存せず、なおかつ強力な内部表現が可能になった。

3.3 意味解析木の記述

以下に、OMT の想定するソフトウェア・ライフサイクルにしたがって、汎用的な解析木の設計手順を述べる。

1. 分析

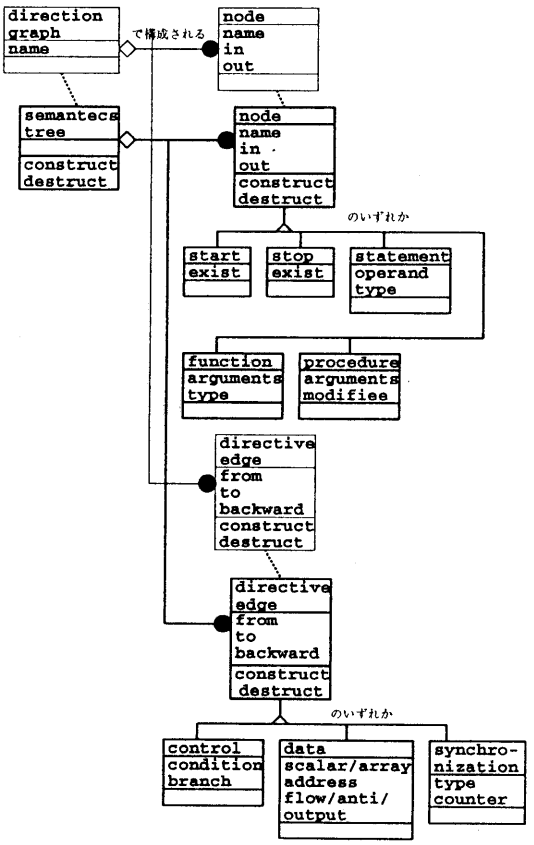

図 2: 解析木のオブジェクトモデル

分析の段階では、最初にオブジェクトやオブジェクトの関連を問題の記述から識別し、静的なデータ構造を保持する図: 2 のようなオブジェクトモデルを構築する。

一般的なプログラムの意味解析木は、各文をノードとする有向グラフとなる。さらに、解析木を構成するデータ構造としては、

- 制御フロー
- データフロー
- 同期

が必要である。また、複数のスコープをもつ言語では、解析木のノードは単なる文ではなく、関数、手続きなども表現できる必要がある。

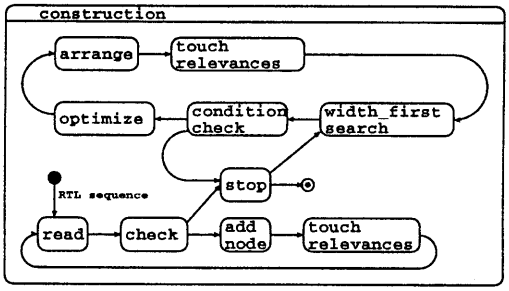

図 3: 解析木の動的モデル

次に、対象とする問題の範囲で生じるイベントについて、それぞれシナリオを用意し、データパスおよびどのオブジェクトに関係したイベントであるかを識別する。これを状態図にまとめ、図: 3 に示すような動的モデルとする。

意味解析木に対する入力としては、構文解析まで終った逐次の RTL 列を考える。

システムに依存しない最適化作業は、意味解析木が構築された後の自動遷移と考えられる。システムに依存する最適化と、ユーザオプションなどによる最適化レベルの選択については、適用する手法を選択できるようにする。

最後にデータフロー解析と同様の手法を用いて、図:4 のような機能モデルを構築する。

コンパイラの場合、入力されるデータによって、さまざまな処理が必要となるので、制約条件や、最適化の基準の記述が重要である。

この場合は RTL の内容により、作られるノードの種類と、グラフ全体の形が変わる。RTL ではコンディションコードを仮定するので、データフローもそのような記述になっている。

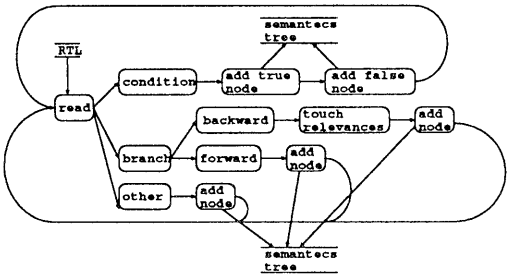

図 4: 解析木の機能モデル

2. システム設計

システム設計の段階では全体の構造を決定し、サブシステムへの分割や、問題領域での並行性の識別を行なう。また、設計の優先順位等を決定する。

コンパイラでは、表の操作にかかる実行時間のコストは、ソース・ファイルの読み込みおよび字句解析にかかるコストに次いで大きくなっている [1]。並列化コンパイラでは、グラフ構造とシンボル操作用の表が多用されるので、グラフとテーブルを中心にデータ構造を整理した。

この際、将来のコンパイラ自体の並行動作を考慮して、並行に実行できるフェーズとして、並列性検出パス、冗長なコードの削除などが挙げられる。

また、フェーズ内で並行に実行できる操作として、宣言シンボルの検索や一部の論理行の中間言語への変換が挙げられた。

3. オブジェクト設計

この段階では、モデルの記述からえられた問題をコンピュータ上の概念へ変換し、各モデルの間の整合性を調整したり、アルゴリズムを検討したりして、設計を詳細化する。

実装するための言語は C++ を想定した [8]。

この結果、図: 5 に示すように、全体をグラフとテーブルを親とする、継承を利用したクラスに階層化でき、設計から実装までの一貫性が向上した。

また、このフェーズではデータ操作アルゴリズムも決定される。たとえば、表のメソッドはイベントにより起動できるように、メッセージの種類を絞り、短いメッセージで効果的な動作ができるようにした。

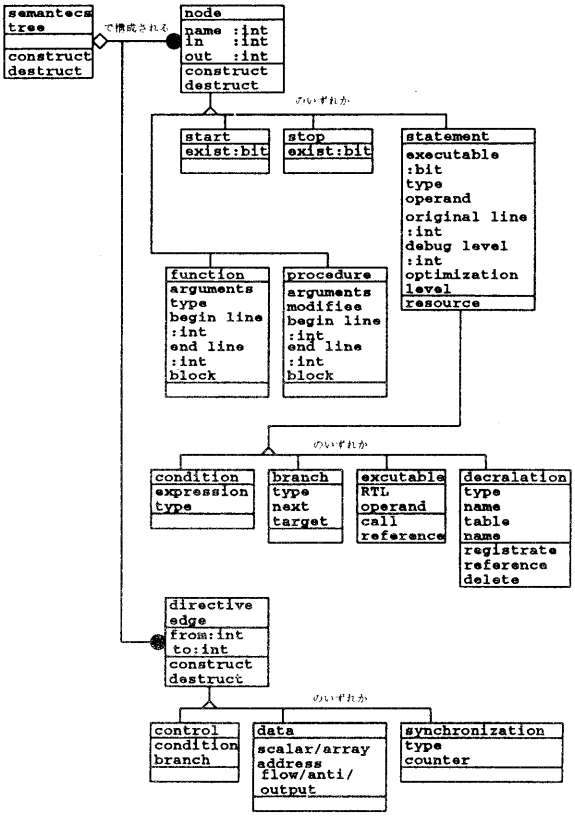

図 5: より詳細なオブジェクトモデル

4 HTG への適用による検証

HTG は Parafrase-2 の内部表現として用いられており、タスクグラフの階層的な拡張の 1 種である [9]。

HTG の特徴として、

- 可約なグラフであるので、最適化が検討しやすい。
- ノードの種類が少ない。

  (compund/simple と block/loop の組合せによる 4 種類のみ。)
- ループ部分を階層化により表現しており、多重ループも統一した表記で扱える。

といった点が挙げられる。この HTG については、前節のオブジェクトモデルを、上の特徴について若干、条件を特殊化するだけで、図: 6 のように表現できた。

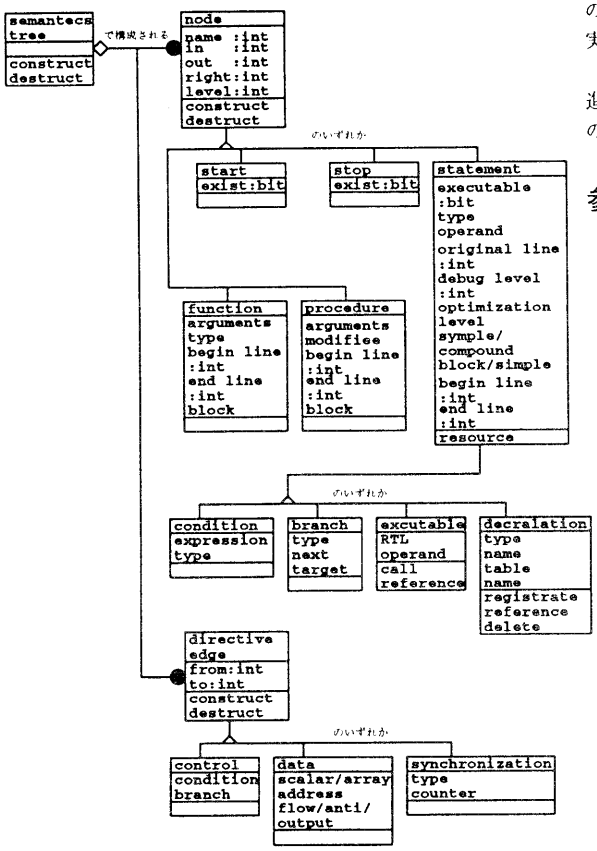

図 6: HTG のオブジェクトモデル

具体的には、

1. エッジの属性として、任意の子供を区別していなかったのに対し、階層性をもたせた。
2. 後方へのジャンプが除去される。
3. ループなどのノード属性が付加される。

などである。

このように、OMT 表記により並列化コンパイラプログラムの解析結果の再利用性が高まっており、統一性のある比較が可能になっている。

5 おわりに

本稿では、並列化コンパイラ研究を概観し、意味解析木の OMT による記述を試みた。

この結果、研究のテストベッドとなりうるような、汎用性、安定性の高い並列化コンパイラを構築できる見通しを得た。OMT を用いた記述により、比較的高レベルの記述を可能にするので、理論的な明解さを保ったまま、実装をふくんだ議論が可能になる。

今後、並列化コンパイラの他の部分についての設計を進めると共に、OMT から実行可能な記述言語への変換の自動化についても研究したい。

参考文献

[1] R. Sethi, A. V. Aho, and J. D. Ullman (原田賢一 訳): コンパイラ 原理・技法・ツール (I,II), サイエンス社, 1990.

[2] P. Briggs and K. D. Cooper: Effective partial redundancy elimination, In *Proc. of the SIGPLAN 94 Conf. on Programming Language Design and Implementation*, pp. 159–170, 1994.

[3] A. A. Chien and J. Dolby: The illinois concert system: Programming support for irregular parallel applications, In *'94 Parallel Compuation and Computing Environments*, pp. 5–10, 1994.

[4] SGI Computer co.: IRIS Power C User's Guide, 1992.

[5] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten: Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors, In *Proc. 1989 Int. Conf. Parallel Processing*, pp. 39–48, 1989.

[6] S. L. Graham, D. F. Bacon, and O. J. Sharp: Compiler transformations for high-performance computing, Technical Report CSD-93-781, Computer Science Division, UCB, 1993.

[7] D. J. Kuck, Y. Muraoka, and S. C. Chen: On the number of operation speedup simultaneously executable in fortran-luke programs and their resulting speed-up, *IEEE Trans. Computer*, Vol. C-21, No. 9, pp. 1293–1310, 1972.

[8] M. A. Ellis and B. Stroustrup (斉藤 信男 監訳): 注解 C++ リファレンスマニュアル, トッパン, 1990.

[9] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten: The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran., In Proc. the Third Workshop on Languages and Compilers for Parallel Computing, pp. 472-492, 1990.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (羽生田 栄一 監訳): オブジェクト指向方法論 OMT モデル化と設計, トッパン, 1992.

[11] D. O'Hallaron, J. Subhlok, J. Stichnoth, and T. Gross: Exploiting task and data parallelism on a multicomputer, In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 13–22, 1993.

[12] D. A. Padua and M. J. Wolfe: Advanced compiler optimizations for supercomputers, *Communications of the ACM*, Vol. 29, No. 12, pp. 1184–1201, 1986.

[13] R. Allen and K. Kennedy: Automatic translation of fortran programs to vector form, *ACM Transaction on Programming Language and Systems*, Vol. 9, No. 4, pp. 491–542, 1987.

[14] B. Leasure, R. H. Kubn, and S. M. Sbab: The kap parallelizer for dec fortran and dec c programs, Technical Report 3, Digital Equipment Corporation, 1994.

[15] K. Kennedy, S. Hiranandani, and C. W. Tseng: Compiling fortran d for mimd distributed-memory machines, *Communications of the ACM*, Vol. 35, No. 8, pp. 66–80, 1992.

[16] M. S. Lam, S. P. Amarasinghe, J. M. Anderson, and A. W. Lim: An overview of a compiler for scalable parallel machines, In *Proc. of the 6th Workshop on Languages and Compilers for Parallel*, pp. 5–10, 1993.

[17] D. O'Hallaron, T. Gross, and J. Subhlok: Task parallelism in a high performance fortran framework, *IEEE Parallel and Distributed Technology*, Vol. 2, No. 3, pp. 16–26, 1993.

[18] The Free Software Foundation: Using and Porting GNU CC for version 2.7.0 edition, 1995.

[19] T. Ruppelt and G. Wirtz: Automatic transformation of high-level object-oriented specification into parallel programs, *Parallel Computing*, Vol. 10, pp. 15–28, 1989.

[20] C. A. Waldspurger and W. E. Weihl: Register relocation: Flexible contexts for multithreading, In *Proc. 20th Annual Symposium on Computer Architecture*, pp. 120–130, 1993.

[21] J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, W. Blume, R. Eigenmann, and P. Tu: Automatic detection of parallelism, *IEEE Parallel and Distributed Technology*, Vol. 2, No. 3, pp. 37–47, 1993.

[22] M. E. Wolfe and M. S. Lam: A loop transformation theory and an algorithm to maximize parallelism, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 452–470, 1991.

[23] H. Zima and B. Chapman (村岡 洋一訳): スーパーコンパイラ, オーム社, 1995.

[24] 岡本 雅巳, 笠原 博徳: OSCAR マルチグレインコンパイラにおける階層型マクロ データフロー処理, 情報処理学会論文誌, Vol. 35, No. 4, pp. 513–521, 1994.

[25] 笠原 博徳, 成田 誠之助: マルチプロセッサ・スケジューリング問題に対する実用的な最適および近似アルゴリズム, 電子情報通信学会論文誌, Vol. J67-D, No. 7, pp. 792–799, 1984.

[26] 吉田 紀彦: プログラム変換に基づくシストリック・アレイの導出, 情報処理学会論文誌, Vol. 30, No. 12, pp. 1530–1537, 1989.

[27] 高橋 英一, 谷口 倫一郎, 雨宮 真人: データフロー解析に基づく関数型言語 valid の 並列化コンパイラ, 情報処理学会論文誌, Vol. 4, pp. 561–570, 1994.

[28] 山名 早人, 神館 淳, 安江 俊明, 村岡 洋一: 並列処理システム-晴-におけるデータフロープログラム 開発環境, 電子情報通信学会論文誌, Vol. J73-DI, No. 6, pp. 569–579, 1990.

[29] 雨宮真人: 超並列言語とその処理系, 情報処理, Vol. 36, No. 6, pp. 514–520, 1995.