計算機アーキテクチャ 87－7
（１９９１．３．１１）

統合型並列化コンパイラ・システム
― 概要，中間コードおよび解析手法 ―

音成 幹　　久我守弘　　村上和彰　　富田眞治
九州大学大学院総合理工学研究科

筆者らは，利用する並列性のレベルが異なる３種類の並列処理システムを開発しており，その開発と並行して，並列/分散OS，並列プログラミング言語および並列プログラミング環境の開発を進めている．この並列プログラミング環境の１つとして統合的なコンパイル環境を提供するために，“統合型並列化コンパイラ (IPC: Integrated Parallelizing Compiler)” を開発している．IPCは，①複数のソース言語対応，②複数のターゲット，および，③複数レベルの並列性，といった３つの点を考慮している．すなわち，ソース言語依存部分とターゲット依存部分をそれぞれ独自に作成し，コンパイラの共通した部分を共有できるよう設計することで，効率の良いコンパイラ開発が行えることを目標としている．

本論文では，IPCの概要，中間コード構成，ループ再構成のための解析手法について述べている．

Integrated Parallelizing Compiler System
― Overview, Intermediate Code, and Dependency Analysis ―

Miki OTONARI, Morihiro KUGA, Kazuaki MURAKAMI, and Shinji TOMITA

Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University

6-1 Kasuga-koen, Kasuga-shi, Fukuoka 816 Japan

e-mail: {miki, kuga, murakami, tomita}@is.kyushu-u.ac.jp

The authors are developing three parallel processing systems each exploiting distinct level of parallelism. In addition to developing these systems, the authors are developing a parallel/distributed operating system, a parallel programming language, and a parallel programming environment. As one of the parallel programming environment, the authors are developing the “Integrated Parallelizing Compiler (IPC) System”. The IPC is considered following three items; i) Multilingualism, ii) Retargetability(Multitargetability), and iii) Multilevel Parallelism.

In this paper, an overview, a structure of the intermeditate code, and dependency analysis for restructuring of the IPC are described.

1 はじめに

現在，我々の研究室では以下の並列処理システムを開発している．

① 可変構造型並列計算機 (VIP: Variable Interconnection Parallel processor)[1]: 128 台の SPARC マイクロプロセッサを 128×128 のクロスバー網で相互結合する，ダイナミック・アーキテクチャを採用した問題適応型の粗粒度レベル並列処理マシン．

② DSNS(Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscaler) プロセッサ [2][3]: 動的ハザード解消，静的コード・スケジューリング，非均質型機能ユニットといった特長を持つスーパースカラ・プロセッサの試作機．

③ 『順風』[4]: ストリーム FIFO 方式に基づく試作機で，複数本の演算パイプラインによりループ・レベルでの並列処理を行うベクトル・プロセッサ (さらに演算パイプラインを複数命令で共有することにより，ベクトル要素レベルの並列処理も可能)．

また，これらのハードウェア・システムの開発と並行して，並列/分散 OS[5]，並列プログラミング言語 "SERVE"[6]，および，並列プログラミング環境の開発を進めている．この並列プログラミング環境の 1 つとして統合的なコンパイル環境を提供するために，統合型並列化コンパイラ (IPC: Integrated Parallelizing Compiler)[7] を開発している．

本稿では，まず IPC の概要について述べた後，IPC で採用した中間コード，および，ループ再構成を行うために必要な依存解析手法について述べる．

2 統合型並列化コンパイラ

2.1 要件

本システムの開発目的は，並列処理ハードウェアの性能を十分に引き出し，快適な並列プログラミング環境を提供することである．本システムでは以下の要件を統合することで，これに応える．

① Multilingualism

逐次/並列型言語を問わず，複数のソース言語をコンパイル可能とする．並列言語やオブジェクト指向言語といった並列性の抽出に適した言語のみでなく，逐次処理を前提とした汎用言語へのサポートが必要である．

② Retargetability(Multitargetability)

複数の並列処理システムをターゲットとする．これには，現在我々の研究室で開発中である前記の 3 ハードウェア・システム，および，将来開発するであろうハードウェア・システムも含まれる．

従来のコンパイラは目的言語が特定された専用のコンパイラであった．また，ターゲット・マシンについても同様に，専用コンパイラが主流である．この方法では，例えば n 種類の言語と m 種類のターゲット・マシンについてコンパイラを作る場合，$n \times m$ の労力が必要である．しかし，コンパイラの内部ではかなり似通った処理ルーチンが多く，この作業がコンパイラ同士で重複していることは否定できない．そこで，これら共通のルーチンをコンパイラ同士で共有する．つまり，言語依存部分とターゲット依存部分をそれぞれ独自に作り，コンパイラの共通した部分を共有できるよう設計することで，効率の良い開発が行える．

③ Multilevel Optimization

ターゲットに依存/非依存の最適化を，グローバル・レベルからローカル・レベルまであらゆるレベルで行う．

④ Multilevel Parallelization

プログラムに明示/暗黙的に内在する並列性を，粗粒度レベルから細粒度レベルに至るまで，あらゆるレベルでの並列性を抽出し活用する．

ターゲット・マシンによって，それが必要とする並列性は異なる．また，細粒度レベルの並列性のみ利用するターゲット・マシンでも，もっと大きなレベルから並列性を抽出し，コードの移動等によって細粒度レベルの並列度を上げることも可能である．したがって，あらゆるレベルで並列性を抽出することが必要である．

2.2 設計方針

2.1節で挙げた要件に関する，IPC の設計方針を示す [7](図1参照)．

(1) Multilingual Compiler

この機能は，単一プロセッサ用最適化コンパイラでは，フロント・エンドをソース言語対応に設け，共通の中間言語を通してそれ以下の処理を共用することによりすでに実現している．

しかし，既存の並列化コンパイラでは，並列性の抽出がソース言語にかなり依存しており，一般に並列性の抽出ルーチンは各ソース言語対応に作られている．また，source-to-source restructurer で並列化コンパイラを実現している例もあるが，この場合プログラム再構成処理以下は共用可能だが，restructurer 自身はやはりソース言語個別に作られている．

IPC でもプログラム再構成を行うが，その restructurer は各種ソース言語間で共有できるようにする．すなわち，restrcturer の入出力はソース言語のもつ明示/暗黙的な並列性を継承でき，しかもソース言語に非依存な中間コードを使用する．つまりソース言語個別のフロント・エンドを用意し，ソースプログラムは各言語毎のフロント・エンドにより中間コードに翻訳され，共通の restructurer への入力となる．これによって，Multilingualism を可能とする．また，IPC の restructurer は intermidiate-to-intermidiate code restructurer となる．

(2) Retargetable (Multitargetable) Compiler

この要件を満足するのは一般に困難である．ある特殊な条件下 (レジスタのビット数が同じや OS が同じ) のプロセッサ用に異なる条件 (レジスタ構成，数等) をパラメータ化し，コンパイラのコード生成ルーチン以下 (アセンブラ，リンカ等) をターゲット・マシン別に用意することによって実現した例がある．しかし，一般にセルフ・コンパイル環境ではその必要性がないことから，実現例は少ない．

そこで IPC では，その構成モジュールを retargetability に関して次の 3 種類に分類した上で，全体をモジュール構成する．

① ターゲット非依存 (TI: Target Independent) モジュール: ターゲットの属性にまったく依存しない．これらのモジュールは，すべてのコンパイル環境で共有かつ実行される部分が選ばれる (例: 依存解析モジュール，一般的な最適化モジュール等)．

② パラメータ化ターゲット依存 (TDP: Target Dependent/Parameterized) モジュール: ターゲットの属性に依存するが，その属性がパラメータ化可能である．入力パラメータにより，モジュールの実行の戦略が決定される (例: ループ再構成モジュール，仮想コード生成モジュール等)．

③ ターゲット依存 (TD: Target Dependent) モジュール: ターゲットの属性に依存し，かつ，その属性のパラメータ化が困難である．このモジュールはターゲット・マシン個別に用意しなければならない．このモジュールを小さくすることは，IPC 全体の共有性を高める (例: 局所コード・スケジューリング・モジュール等)．

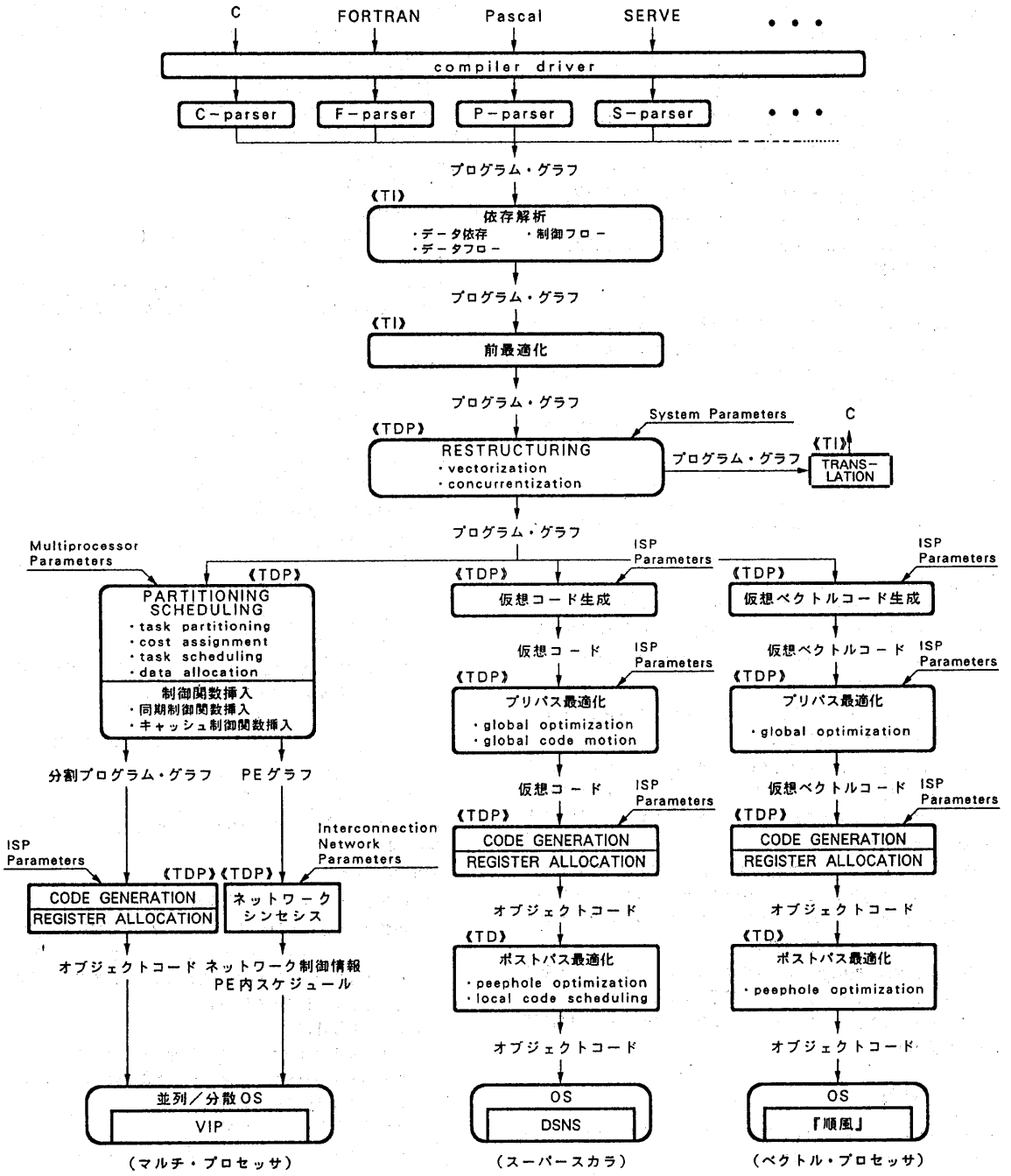

図 1: 統合型並列化コンパイラ・システムの構成

上記①②のモジュールは各種ターゲットマシン用のコンパイラで共用可能である．なお，TDP モジュールへ与えるパラメータとしては，以下の 4 種類を設ける．

i) System Parameters: ターゲット・マシンのシステム的要因 (シングル/マルチ，スカラ/ベクトル等)

ii) Multiprocessor Parameters: マルチプロセッサ要因 (プロセッサ数，メモリ構成，相互結合網構成等)

iii) Interconnection Network Parameters: ターゲット・マシンの相互結合網要因 (トポロジー，ポート構成，バンド巾等)

iv) IPS Parameters: ターゲット・マシンの命令セット・アーキテクチャ要因 (中間コードとオブジェクトコードの対応関係，レジスタ構成等)

(3) Multilevel Optimizing Compiler

この要件は多くの既存のコンパイラで実現されているが，その大部分は 1 ターゲット専用の最適化である．IPC では複数のマシンをターゲットとしているので，最適化のレベルを以下の 2 つのレベルに分けて行う．

① ターゲット・マシン非依存: ターゲット・マシンに依存しない最適化を TI モジュールで行う．

② ターゲット・マシン依存: ターゲット・マシンに依存するような最適化は TDP モジュールでパラメータにより選択，もしくは，TD モジュールでそのターゲット・マシン専用の最適化を行う．

(4) Multilevel Parallelizing Compiler

IPC では，プログラムの並列性のレベルを以下の 4 レベルに分けて，ターゲット・マシンに合ったレベルの並列性を抽出し活用する．

① プロセス・レベル: 並列プログラミング言語等を用いてユーザが明示的に示した並列性．

② ブロック・レベル: 基本ブロック，および，その集合であるマクロブロック間の並列性．

上記①②の並列処理単位 (慣例によりタスクと呼ぶ) は，マルチプロセッサに対して，タスク分割&スケジューリングフェーズにおいてプログラム分割により抽出し，タスク・スケジューリングでその並列性を活用する．

③ ループ・レベル: FORTRAN の DO ループ等に内在する並列性．ループ再構成フェーズにおいて，ベクトル化，および，平行化 (doall，doacross 等) することでベクトルプロセッサやマルチプロセッサ等で活用する．

④ 命令レベル: 機械命令間の並列性．プリパス/ポストパス最適化フェーズにおける広域コード移動，および，局所コード・スケジューリングによりスーパースカラ・プロセッサや VLIW 等で活用する．

2.3 処理過程

2.2節の設計方針に基づき，IPC を大きく分けて以下の 5 つのフェーズで構成する (図 1参照)．

(1) フロントエンド (parser)

字句解析，構文解析，意味解析を行い，ソースコードを中間コードに翻訳する．このフェーズを構成するモジュールはソース言語に依存するので，対象とするソース言語毎に用意する．これにより，Multilingualism を実現する．中間コードにはオペレーション，オペランド等がソースコードのイメージのまま保持され，次の依存解析部に渡される．また，ソースコードから中間コードに翻訳する際に，定数の丸め込み等の前処理も行う．

(2) 依存解析

中間コードに対して制御フロー解析，データ依存解析，データフロー解析を行う．プログラム・グラフを形成し，これらの情報を基にコード最適化やループ再構成等を行う．

(3) 前最適化 (pre-optimization)

ターゲット・マシンに依存しない最適化を行う．マシンに依存する最適化はマシン専用のバック・エンドで行う．

上記 (2)(3) のフェーズは TI モジュールで構成され，ターゲット・マシンに依らず共有される．

(4) ループ再構成 (restructuring)

逐次ループの並列ループへの再構成を行う．これにより，ループ・レベルの並列性の抽出を行う．

このフェーズは TDP モジュールで構成する．各ターゲット・マシンの System Parameters により，ループ再構成の戦略が決定される．

(5) バックエンド

コード・スケジューリング，および，コード生成を経てオブジェクト・コードを生成する．このフェーズはターゲット・マシンへの依存度が高く，TDP モジュールもしくは TD モジュールで構成する．

現在，我々の研究室で開発している汎用並列処理システムに対するバック・エンド処理の概要は以下の通りである．

1. 可変構造型並列計算機 (VIP)

   i) タスク分割 & スケジューリング

   プログラムをタスクに分割し，プロセッサに割り当てるスケジューリングを行う．これに伴い，同期やキャッシュ制御のための関数の挿入を行う [8].

   ii-1) ネットワークシンセシス

   可変構造化した相互結合網の通信ネットワーク制御のための PE 間通信パターンを生成する [9].

   ii-2) オブジェクト・コード生成

   タスク毎に中間コードからオブジェクト・コード (SPARC コード) を生成する．

   上記処理過程は全てマルチプロセッサ用の TDP モジュールで構成し，VIP の Multiprocessor Parameters によりその戦略を決定する．

2. DSNS プロセッサ

   i) 仮想オブジェクト・コード生成

   中間コードから仮想オブジェクト・コードを生成する．仮想オブジェクト・コードは，オペレーションとしてスーパースカラ・プロセッサの仮想命令セット，オペランドとしては仮想レジスタを用いる．

   ii) プリパス最適化

   与えられたプログラム全体に内在する命令レベルの並列性を抽出するために，広域コード移動を行う．また，広域コード最適化を行う [10][11].

   iii) オブジェクト・コード生成

   グラフ彩色アルゴリズムに基づいたレジスタ割付けを行う [13].

   iv) ポストパス最適化

   DSNS プロセッサのハードウエアを有効に利用するための局所コードスケジューリング，および，peep-hole 最適化を行う．

   上記の処理過程のうち，i), ii), iii) はスーパースカラ用の TDP モジュールで構成する．また，vi) は DSNS 専用の TD モジュールで構成し ISP Parameters によってその戦略を決定する．

3. 『順風』

i) 仮想オブジェクト・コード生成 (ベクトル化)
中間コードから仮想オブジェクト・コードを生成する．仮想オブジェクト・コードは，オペレーションとして仮想ベクトル命令セット，オペランドとしては仮想レジスタを用いる．

ii) プリパス最適化
広域コード最適化 (ループ内不変計算のループ外への移動等) を行う．

iii) オブジェクト・コード生成
ベクトルレジスタの割付けを行う．

iv) ポストパス最適化
ハードウェアを有効に利用するための peep-hole 最適化を行う．

上記の処理過程のうち，i), ii), iii) ベクトルプロセッサ用の TDP モジュールで構成する．また，vi) は『順風』専用の TD モジュールで構成し，ISP Parameters によってその戦略を決定する．

3 中間コード

3.1 要件

IPC における中間コードに対する要件を以下に示す．

(1) ソース言語からの独立

一般に，フロントエンドでソースプログラムを一度何らかの中間コードに翻訳し最適化処理を行った後，バックエンドで中間コードから目的コードを生成する．このとき，ある特定のソース言語に依存した中間コードを使うと，そのソース言語のプログラムをコンパイルする場合はフロント・エンドの負担は軽いかもしれない．しかし，その他の言語をコンパイルする場合は，逆にフロント・エンドに余分な負担をかける．ソース言語別に中間コードを用意すればこのような負担は少なくなるが，これは本質的解決になっていない．また，その後に行う依存解析は中間コードに依存するので，ソース言語別にそれぞれ用意しなければならなくなり，結局コンパイラをソース言語別に作るのと大差ない．

そこで Multilingualism を実現するには，ソース言語に依存する表現がないような中間コードが望ましい．

(2) ソースコードのもつ並列性の継承

ソースコードが持っているループの並列性をそのままループ再構成フェーズに伝える必要がある．一般の中間コードでは，ループはラベルと条件分岐に翻訳されてしまう．したがって，その部分がどのようなループだったかは，制御フローの解析結果から推定し直さなければならない．IPC ではループのイメージをそのままループ再構成フェーズに伝えることで自由度の高い再構成を可能にする．

(3) ターゲット・マシンからの独立

(1) の場合と同様に，ある特定のマシンに依存した中間コードを用いると，バック・エンドで行うコード生成はかなり容易となる．しかし，他のマシンをターゲットとしてコードを生成しようとする場合，負担が大きくなる．また，最適化も片寄った方向にしか行えない可能性もある．したがって，ターゲット・マシンとは独立した中間コードが望ましい．

3.2 設計方針

3.1節の要件に基づき，IPC の中間コードの設計にあたっては，以下の点を設計方針とした．

① ソース言語からできるだけ独立させる

IPC では中間コードをソース言語とはできるだけ独立にした．これにより，ソース言語別のフロント・エンドを開発するだけで，複数のソース言語に対応できる．しかしながら，言語に依存しない中間コードを作ることは，ソースの情報が欠落する恐れがある．したがって，若干の言語依存部分が存在する．言語依存部分をできるだけ小さくするために，意味が近い似通った部分は同じ表現に収束させ，収束できない部分や収束させないことに利用価値がある部分だけを残している．

② ソースコードのもつ並列性のイメージを継承させる

フロントエンドの出力からバックエンドへの入力までは中間コードによって情報を伝達する．中間コードはステートメント・レベルに設定し，ループや分岐等の表現を残すことで，並列性のイメージをループ再構成フェーズまで伝えることを可能とする．

③ ターゲットマシンから独立させる

IPC では中間コードをターゲットマシンとは独立なコードとする．これにより，複数のターゲットマシンに対してフロントエンドおよび依存解析ルーチン，ループ再構成ルーチンが共用できる．したがって，バックエンドのみを開発すれば複数のマシンに対応できる．

④ 依存情報等の情報を直接保持する

種々の依存情報をオペレーションや変数が各自保持することで，コード移動処理の際に必要なテーブルの更新等の処理が少なくできる．

⑤ コード生成のための構文木を保持する

構文木 (syntax tree) は節として演算子，節の子供として節の演算子のオペランドがくるようにした木である．一般にコード生成ルーチンは，中間コード (ここでは三番地コード) を構文木に読み換え，構文木に対する属性 (変換) 文法，あるいは木のパターンマッチングと置換によって定式化される．よって，直接構文木を保持することはコード生成モジュールを軽減する．また，ソースコードのイメージをそのまま保持できるので，最適化やループ再構成の後でも，元のソース言語に再変換が可能である．

3.3 プログラム・グラフ

IPC の中間コードは，以下の 3 種類のノードから成るグラフ (プログラム・グラフと呼ぶ) で表現する．

① OP ノード
② 依存情報ノード
③ 基本ブロック・ノード

OP ノードはフロントエンドで，また，依存情報ノードと基本ブロック・ノードは依存解析フェーズで生成する．

3.3.1 OP ノード

OP ノードに含むデータは，(1) プログラムの構造を示すもの，(2) 依存情報，の 2 つに大別できる (図2参照)．

(1) プログラムの構造を示すもの

① ラベル (nodelabel) … ノード識別子．
② オペレーションコード (opcode) … ノードの種類を示す．
③ リンクポインタ 1(nodeparent,nodechild,nodeprev,nodenext) … ネストは親子関係，文の字面上の順番は前後関係でリンクする．式の構文木もこれらのリンクポインタにより表現する．

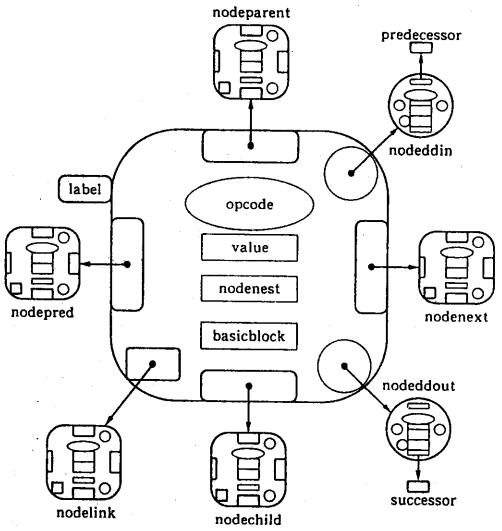

図 2: OP ノード

④ リンクポインタ 2(nodelink_for, nodelink_back) … 依存解析等で使用するポインタ.

(2) 依存情報

① ネスト (nodenest) … ノードの現在のネストの深さ.

② 基本ブロック (basicblock) … ノードが所属する基本ブロック.

③ 依存情報 (nodeddin, nodeddout) … このノードに及ぼす依存関係に対応する依存情報ノード, および, このノードが及ぼす依存関係に対応する依存情報ノードへのリンク.

3.3.2 依存情報ノード

依存情報ノードに含むデータは, (1) 変数間データ依存関係を示すもの, (2) イタレーション間データ依存関係を示すもの, の 2 つに大別できる (図 3参照).

(1) 変数間データ依存関係を示すもの

① ラベル (ddlabel) … 依存情報の識別子.

② 依存の種類 (ddtype) … データ依存関係の種類.

③ 先行ノード (ddpred) … データ依存関係における先行ノード.

④ 後続ノード (ddsucc) … データ依存関係における後続ノード.

⑤ 次の入力データ依存関係 (ddnextpred) … 後続ノードに対する次の優先順位の依存関係.

⑥ 次の出力データ依存関係 (ddnextsucc) … 先行ノードに対する次の優先順位の依存関係.

(2) イタレーション間データ依存関係を示すもの

① ネスト (ddnest) … ループのネストの深さ.

② 依存関係ベクトル [方向](dddir) … 依存関係を及ぼす方向.

③ 依存関係ベクトル [距離](dddist) … 依存関係の及ぶ距離.

④ リンクポインタ (ddlink) … 関係のある依存情報ノードにリンクする.

3.3.3 基本ブロック・ノード

基本ブロックノードに含むデータは, (1) 基本ブロックについて示すもの, (2) 制御フローを示すもの, (3) データフローを示すもの, の 3 つに大別できる (図 4参照).

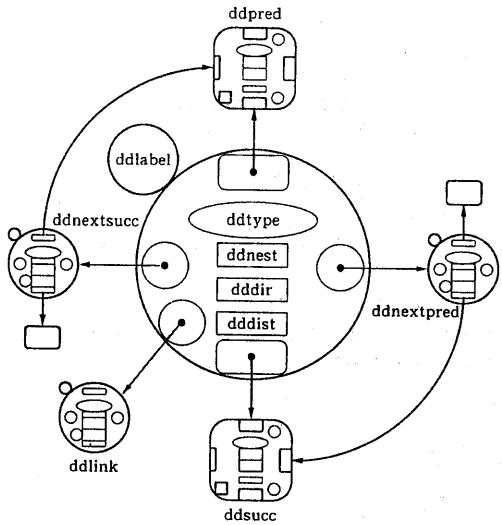

図 3: 依存情報ノード

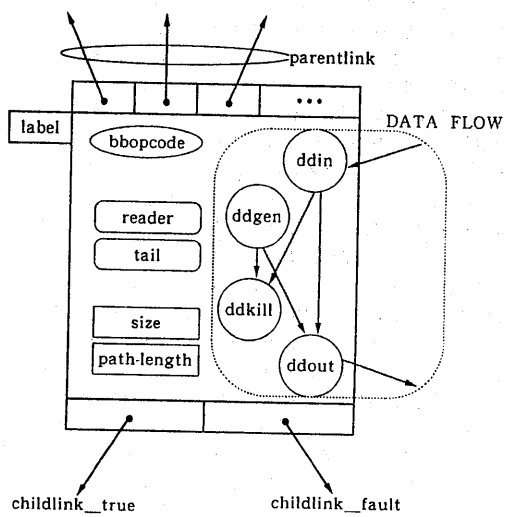

図 4: 基本ブロック・ノード

(1) 基本ブロックについて示すもの

① ラベル (bblabel) … 基本ブロックの識別子.

② オペレーションコード (bbopcode) … 基本ブロックの種類を表す.

③ 先頭ノード (reader) … 基本ブロックの先頭ノード.

④ 最終ノード (tail) … 基本ブロックの最終ノード.

(2) 制御フローを示すもの

① 先行ブロック (bbparent) … 制御フローにおける先行ブロック. 複数存在する可能性がある.

② 後続ブロック (bbchild_true, bbchild_fault) … 制御フローにおける後続ブロック. 後続ブロックは各基本ブロックにつき高々2個存在する.

(3) データフローを示すもの

① サイズ (size) … 基本ブロックが保持する仮想オブジェクト・コードの総数 (推定).

② パス長 (path_length) … 命令実行のクリティカルパスの長さ.

③ データフロー情報 (ddin, ddgen, ddkill, ddout) … 基本ブロックにおけるデータフロー情報. 入力 (in), 生成 (gen), 消滅 (kill), 出力 (out) の4種類.

4 各種解析手法

並列化を行うには, プログラムの実行の流れ (制御フロー) とデータの依存関係を把握しなければならない. 以下, 3章で述べたプログラム・グラフを用いた制御フロー解析とデータ依存解析について述べる.

4.1 制御フロー解析

まずプログラムを基本ブロック (basic block) と呼ぶ小さな命令の集合に分け, 基本ブロック間の制御の流れを解析する. 制御フローは基本ブロックをノード, 制御の流れを有向辺とするフローグラフで表される. 3章で述べたプログラム・グラフはフローグラフに等しい.

4.1.1 基本ブロックへの分割

中間コードを基本ブロックに分割するには以下の戦略で行う. ここで, 基本ブロックの先頭のOPノードをreader, 最後のOPノードをtailとする.

① プログラムの先頭は必ずreaderである (関数の先頭もreaderとする).

② ループの入り口はreaderである. その次のOPノードもreaderである. ループは3種類存在 (DOループ, whileループ, do_whileループ) する.

③ ラベルはreaderである.

④ 関数呼出しを含む命令はそれ自身で基本ブロックである.

⑤ readerの前に命令が存在すれば, それは前の基本ブロックのtailである.

⑥ 条件, 無条件分岐はtailである.

⑦ tailの次の命令はreaderである.

⑧ 次の命令がなければtailである.

このとき, 基本ブロックのbbopcodeを決定しておく, bbopcodeはその基本ブロックのネストの原因 (条件文, ループ) もしくは, 関数呼出しである.

4.1.2 制御フロー解析アルゴリズム

IPCにおける制御フロー解析のアルゴリズムの概要を以下に示す. 制御フローの解析は, 制御の流れを追いながら, 基本ブロックをノードとするフローグラフを作ることに等しい.

制御フロー解析は, 基本ブロックを順に走査し, すべての基本ブロックを走査するまで以下の処理を行う.

① 基本ブロックのtailが条件分岐のとき, 分岐先をreaderに持つ基本ブロック (childlink_true) とtailの次の命令をreaderに持つ基本ブロック (childlink_fault) にリンクを張る.

② 基本ブロックのtailが無条件分岐のとき, その分岐先をreaderに持つ基本ブロック (childlink_true = childlink_fault) にリンクを張る.

③ 基本ブロックのbbopcodeが関数呼出しのとき, 次の基本ブロックへリンクを張る.

④ それ以外でtailのOP次のノードが存在するとき, そのノードをreaderにもつ基本ブロックへリンクを張る.

⑤ それ以外でtailの次のノードが存在しないとき,

(a) bbopcodeがループの場合, ループの次の命令をreaderにもつ基本ブロックへリンクを張る.

(b) bbopcodeが条件文の場合, 条件文の次の命令をreaderにもつ基本ブロックへリンクを張る.

(c) tailが関数の終わりの場合, リンクはNULLとする.

4.2 データ依存解析

4.2.1 イタレーション間データ依存関係

ループ内のデータ依存に加えて, イタレーション間でのデータの依存関係も問題となる. 特にDOループにおいて, イタレーション間で変化するループカウンタを配列の添字に使った場合の依存関係は, ループの並列化に大きな影響を与える. そこで, DOループのイタレーションと添字との関係について解析を行う. つまり, ある配列が同じデータ (場所) を定義または参照するまでのイタレーション距離 (添字間の差) とその方向についての解析を行う. ここで, 各DOループにおいて, イタレーションに関するデータの依存の向きをデータ依存方向ベクトルと呼ぶ.

whileループについては, ループカウンタを使用しないので, イタレーションと配列の添字についての解析は一般には困難である. しかし, イタレーションによる影響 (依存の有無) については判断できる.

4.2.2 データ依存解析アルゴリズム

(1) アルゴリズム

データ依存関係の解析のアルゴリズムの概略を以下に示す.

① 変数名別に双方向リンクをはる (linkio_node). 変数テーブルに登録された変数名順に行い, すべての変数名について②以降の処理を行う.

② 1つの変数名について定義参照関係を調べ, 関係がある物だけをリストアップする (dd_intersect_list).

③ ②でリストアップされた関係についてループとの関連について調べる (dd_intersect).

④ ループ内の変数であれば依存方向ベクトルを調べる (dd_subscript). ただし, 依存方向ベクトルが判るのはDOループだけであり, その他のループは依存関係の有無だけしかわからない.

⑤ 依存ノードを作る (make_ddnode).

(2) 変数のリンク (linkio_node)

同じ変数名ごとに，変数の宣言ノードを起点として字面上の順番にそれぞれリンクを張る (nodelink_for, nodelink_back)．これにより，変数ノードは同じ変数名ごとのグループに分けられる．すなわち，中間コードを深さ優先で走査し，ある変数のOPノードを見つけると同じ変数名のテーブルよりその変数を宣言しているOPノードを調べ，そこから張られている双方向リストの最後にその変数のノードを挿入していく．

(3) 依存のリストアップ (dd_intersect_list)

同じ変数名の変数ノード同士の定義参照関係を調べる．前処理として構文解析時に変数の性質を解析し，以下の4種類に分類する．

i) L-Value … 左辺値 (left value)．すなわち変数の定義がされている変数ノード．

ii) R-Value … 右辺値 (right value)．すなわち変数の参照がされている変数ノード．

iii) I-Valie … 配列の添字および関数呼出しのパラメータ．ここでの取扱いはR-Valueと同じである．

iv) RL-Value … 左辺値と右辺値の性質を持つ．(例:C言語のインクリメンタル演算子等)．依存関係は両方の性質とも考えなければならない．

これらの出現の順番をプログラムの字面上の順番に見ていくことで変数間の依存関係をフロー依存，逆依存，出力依存，入力依存の4種類に分ける．ここで，入力依存は先行関係を一切規定しないので，他の3種類の依存関係だけをリストアップする．

(4) ループとの関係の解析 (dd_intersect)

(3) でリストアップされた依存関係とループとの関係を解析する．まず，依存関係にある2つのOPノードについて，どちらもループ内部のオペレーションのノードでないものは，この段階で排除して (6) の処理を行う．ループの内部であった場合，それぞれの変数ノードについて，ループのネストと，それらの変数ノードを内部に含むループの関係 (同じループか，異なるループか) を調べる．異なるループであればループに関する依存関係はないので (6) の処理を行う．

(5) イタレーション間データ依存関係解析 (dd_subscript)

(4) で集められた情報を基にして，2変数ノードのイタレーション間で生じる依存解析を行う．イタレーション間のデータ依存関係の解析アルゴリズムは，GCDテスト，Baerjeeの不等式 [12] などがある．現在，IPCでは簡単に配列要素同士のGCDテストを行っている．

まず．配列変数とそうでない単純変数に分け，単純変数に関してはイタレーションによって起こる自分自身との依存関係を解析して終わる．例えば，変数ノードの性質がL-Valueであった場合は，イタレーションにより自分自身への出力依存関係が生じることとなる．配列変数については，以下の条件をチェックする

① 添字が単調増加 (減少) でないなら依存関係があるとする．

② 添字が定数，またはイタレーションによって変化しないならば，イタレーションに関するに関する依存関係はない．

③ GCDテストを行い，依存方向ベクトルを調べる．

④ 依存距離 (依存のある配列間の添字の差) とループインデックスの最小値と最大値との差 (trip count) を比較したとき，依存距離が大きいか等しければイタレーションが重なっても2つの配列同士が同じ場所を示すことがないため，イタレーションに関する依存はないと判断する．trip countが大きければ，依存があるとする．

(6) 依存ノードのリンク (make_ddnode)

(5) までに判定された依存関係の情報を持つ依存ノード作り，依存のある変数ノード間に置く．ただし，それ以前に依存情報がOPノードに付属していれば，字面上近い方が影響が大きいため優先順位を高くする．

5 おわりに

以上，我々が開発中の統合型並列化コンパイラについて，その概要，中間コード，および，ループ再構成のための各種解析手法について述べた．IPCは言語依存部分とターゲット依存部分をそれぞれ独自に作成し，コンパイラの共通した部分を共有できるよう設計することで，効率の良いコンパイラ開発を行うことが可能である．

IPCに含まれる種々のモジュールの詳細については，関連文献，および，今後発表する論文を参照して頂きたい．

参考文献

[1] 森ほか: "可変構造型並列計算機のPE間メッセージ通信機構," 情報処理学会論文誌, vol.30, no.12 (1989年12月).

[2] 村上ほか: "SIMP(単一命令流/多重命令パイプライン)方式に基づくスーパースカラ・プロセッサの改良方針," 信学技法, CPSY90-54 (1990年7月).

[3] 原ほか: "SIMP(単一命令流/多重命令パイプライン)方式に基づく改良版スーパースカラ・プロセッサの構成と処理," 信学技法, CPSY90-55 (1990年7月).

[4] 弘中ほか: "ストリームFIFO方式に基づくベクトル・プロセッサ『順風』," 信学技法, CPSY89-39 (1989年8月),

[5] 福田ほか: "可変構造型並列計算機の並列/分散OS," 情処研報, 89-OS-43-8 (1989年6月).

[6] 草野ほか: "並列プログラミング言語ＳＥＲＶＥの処理系," 信学技報, CPSY90-82 (1990年11月).

[7] 村上ほか: "統合型並列化コンパイラ・システム —概要—," 第40回情処全大予稿集, 1G-1 (1990年3月).

[8] 岩田ほか: "統合型並列化コンパイラ・システム —コンパイラ支援キャッシュ・コヒーレンス制御—," 情処研報, ARC-83-21 (1990年7月).

[9] 蒲池ほか: "統合型並列化コンパイラ・システム —ネットワーク・シンセシス—," 第40回情処全大予稿集, 1G-2(1990年3月).

[10] 入江ほか: "SIMP(単一命令流/多重命令パイプライン)方式に基づくスーパースカラ・プロセッサ『新風』のための静的コード・スケジューリング技法," 情処研報, 1G-3 (1990年3月).

[11] 入江ほか: "統合型並列化コンパイラ・システム —局所コード・スケジューリング—," 第40回情処全大予稿集, 1G-3 (1990年3月).

[12] Banerjee, U. et al.: "Time and Parallelprocessor Bounds for Foetran-Like Loops", *IEEE Trans. Comp.*, Vol.C-28, No.9, pp.660-670, Sep. 1979.

[13] F. Chow and J. Hennesy: "Register allocation by Priority-Based Coloring," *Proc. SIGPLAN '84 Symp. Compiler Construction*, June 1984.