計算機アーキテクチャ 137－30
ハイパフォーマンス
コンピューティング 80－30
（2000．3．3）

粗粒度並列化プリプロセッサの実装

仲村柄 真人 † 岩井 啓輔 ‡ 天野 英晴 ‡

† 日立情報システムズ ‡ 慶應義塾大学理工学部

現在，複数のCPUを搭載した，マルチプロセッサ方式の計算機が実用的になり，それに伴ない自動並列化コンパイラの研究が盛んである．本研究では，汎用性の高いPOSIX Threadにより記述されたコードを生成する自動並列化コンパイラを実装した．スタンフォード大学で開発されているSUIFコンパイラをベースにし，粗粒度並列化の追加，中粒度並列化の改良を行ない，性能の向上を目指した．本稿では，コンパイラ実装の詳細と，そのコンパイラでいくつかのプログラムをコンパイルして実行した際の性能評価の結果を示す．

Implementation of coarse grain parallelizing pre-processor

Masato Nakamura† Keisuke Iwai‡ Hideharu Amano‡

†Hitachi Information Systems ‡Keio University

In order to make the best use of recent multiprocessors performance, researches on automatic parallelizing compiler have been widely exerted. In this research, we developed an automatic parallelizing compiler which generates parallelized code with POSIX Thread. Based on SUIF compiler developed by Stanford University. a pre-processor for the corse grain parallel processing is attached, and the loop level parallelizing is improved. Here, the implementation of the compiler and the performance evaluation results are presented.

1 はじめに

現在，複数のCPUを搭載したマルチプロセッサ構成の計算機が普及している．このようなマルチプロセッサの性能を引き出すためには，この上で動作するアプリケーションが並列化されている必要がある．しかし，並列性を考慮したプログラム開発は困難な点が多く，開発者の負担も大きい．そこで記述したプログラム中の並列性の抽出などを自動で行なう，自動並列化コンパイラの研究が盛んに行なわれている．

本報告では，逐次型のプログラムを現在のUNIX環境上で一般的に用いられているマルチスレッドライブラリであるPOSIX Threadライブラリによる並列記述に変換する自動並列化コンパイラの設計，実装について述べる．実装は，スタンフォード大学で研究，開発されている自動並列化コンパイラシステムSUIF[1][2]をベースに粗粒度並列化などの機能を付加する形で行なった．さらに，実装したコンパイラを用い，いくつかのプログラムをコンパイルして実行した際の性能評価結果について報告する．

2 SUIFコンパイラシステム

2.1 SUIFコンパイラの概要

SUIFコンパイラシステム(以下SUIFコンパイラ)はスタンフォード大学で研究，開発が進められている自動並列化コンパイラであり，主にループレベルでの並列性の抽出を目的としている．対象とする言語はC言語またはFortranであり，これらの入力言語を中間言語形式に変換し，ライブラリを使用して並列化されたソースプログラムを出力する．出力されたソースプログラムを既存のコンパイラによってコンパイルすることにより，並列動作するプログラムを生成する．コンパイラ内部では，中間言語であ

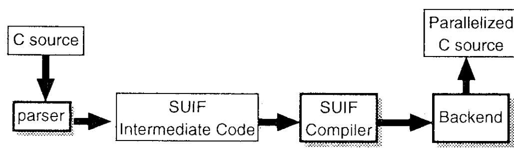

図 1: SUIF コンパイラの処理の流れ

る SUIF(Stanford University Intermediate Format, 以下 SUIF コード) を定義し，これに対してデータ依存解析などの処理を行なう．

図 1に，SUIF コンパイラシステムでのコンパイルの流れを示す．

2.2 SUIF コンパイラの構成

SUIF コンパイラはパーサ，依存解析など処理ごとにユーティリティに分れている．各ユーティリティ，また共通に使用するライブラリ群は C++で記述されている．SUIF コンパイラを構成する，ユーティリティの一部を以下に示す．

- "snoot" パーサ (対象言語 C,Fortran)
- "porky" コード変換など
- "reduction" reduction ループの処理
- "pgen" 並列動作のためのライブラリコールを SUIF コードに付加

中間言語は AST(Abstruct syntax tree) になっており，中間言語自体は四つ組である．なお，SUIF には同期などの並列化のためのプリミティブを持たない．そのかわり，Annote と呼ばれる自由に付加できる情報をもち，これに記述された情報に従い各種処理を行なう．主な Annote の種類を以下に示す．

- "Line" SUIF のコードとソースファイルの対応
- "Reduction" Reduction ループ
- "Sum" 合計の計算
- "Doall" DOALL ループ

3 並列化コンパイラの構成

今回実装したコンパイラは，SUIF コードを読み込みマクロデータフロー処理 [3] を行なう．マクロデータフロー処理とは，逐次的に記述されたプログラムからプログラムをブロックに分割するタスク分割，タスク間のデータ依存解析，タスクの実行可能条件解析などを行ない，プログラムブロック間の並列処理（いわゆる粗粒度並列処理）を行なう手法である．実装したコンパイラではマクロデータフロー処理は SUIF コンパイラに対するプリプロセッサとして動作する．

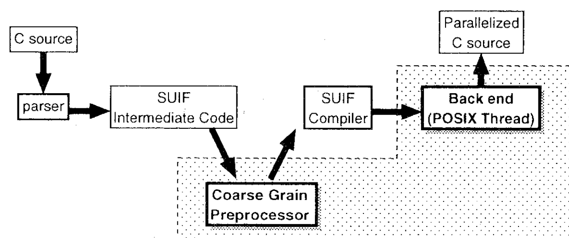

図 2: 実装したコンパイラの処理フロー

また，SUIF コードより POSIX Thread を使用して並列記述した C 言語ソースプログラムを出力するためのバックエンド部の実装も行なった．バックエンドにより出力されたソースプログラムは，SUIF コンパイラと同様に既存のコンパイラでコンパイルし，実行コードを生成する．

パーサおよび各種解析や中粒度並列化は SUIF コンパイラをそのまま使用する．これにより，SUIF の強力な解析機能を活用することができる．

実装したコンパイラによる処理の流れを図 2に示す．

3.1 タスク分割

粗粒度並列化を行なうためには，まずプログラムを粒度の大きいマクロタスクに分割する．マクロタスクには，タスクの外側から内部への飛び込みがない BB(基本ブロック)，for 文などの繰り返しを意味する RB(繰り返しブロック)，およびサブルーチンからなる SB（サブルーチンブロック）がある．

BB は通常の演算のみで構成されるブロックである．基本的に BB 内部ではシーケンシャルに処理を行なう．また，このプリプロセッサではサブルーチンコールも BB の中に含める．

RB は for 文,do-while 文などの繰り返し構造のブロックであり，厳密には最も外側のループである．DOALL などの並列性を含むループも含まれる．DOALL では繰り返しを，使用するスレッド数に分割しそれぞれを並列に実行する．

また，タスク分割時に各タスク毎の情報を抽出する．以下に，抽出する情報について述べる．

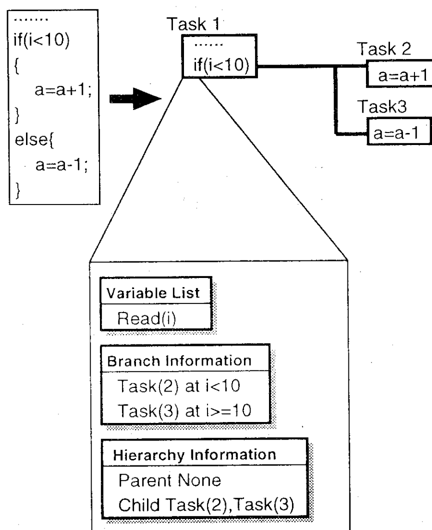

図 3: タスク固有情報の例

- 変数の利用情報
- 分岐情報
- 階層情報

変数の利用情報は，それぞれのタスクにおいて内部で使用されている変数を，読み込みや書き込みといったアクセス手段により分類しリスト形式にしたものである．この情報は，データ依存解析に使用される．

分岐情報は，if 文や switch 文などの構文の情報である．分岐を制御する変数と分岐先のタスク番号を保持する．

階層情報は，ループの内側のタスクなど階層的なタスク構造において親タスクである外側のループ番号を保持している．分岐情報と階層情報は，制御依存解析において参照される．

これらの例を 図 3に示す．

3.2 依存解析

各タスク間のデータ依存解析は，ブロック分割時の変数利用情報により行なう．タスク間ではシンボリックな依存解析のみを行ない，RB 内部の子タスク間では，SUIF では判定しきれない DOALL のために，各配列の GCD テスト [4] によるデータ依存解析を行なっている．

ここで，GCD テストとは，ループ内の各イタレーション間の配列の依存関係を調べるものである．図 4

$$
\begin{array}{l}
for(i=0;i<100;i++)\{ \\
\quad a[3*i+1] = ...; \\
\quad .... = a[4*i+3]; \\
\} \\
3*i_1+1 = 4*i_2+3 \\
3*i_1-4*i_2 = 2 \qquad (1) \\
(\text{ただし } 0<i_1<100\ ,\ 0<i_2<100)
\end{array}
$$

図 4: GCD テストによる依存解析

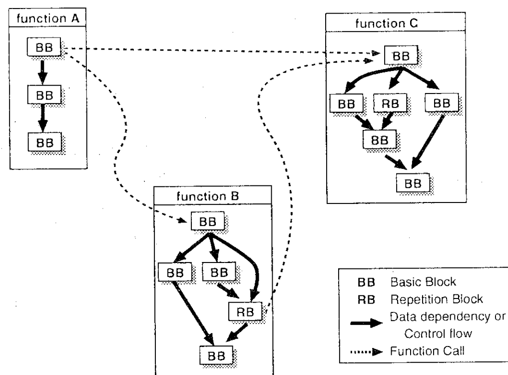

図 5: 関数ごとのマクロフローグラフの例

においては，a[3*i+1] と a[4*i+3] の間に依存関係があるか調べる．この方法では，それぞれの配列の添字である $3*i+1$ と $4*i+3$ について，図 4の式 (1) を満たす i_1, i_2 が存在するかチェックすればよい．存在するならば，これらの間にはデータ依存が存在することになる．

制御依存の解析は，ブロック分割時に抽出した分岐情報や階層情報を基に行なわれる．制御依存は，if 文や switch 文などデータ依存がなくても，タスクの終了を待たなければならないときに存在する．

次に，これらの依存情報よりタスク間の実行順序を表すマクロフローグラフ [3] を生成する．実装したコンパイラでは関数のインライン展開を行なわないため，マクロフローグラフは各関数ごとに生成される．マクロフローグラフの例を図 5に示す．

3.3 スケジューリング

バックエンドが生成するスケジューリングコードは，分散スケジューリング方式と集中スケジューリング方式 [3] に分られる．分散スケジューリング方式

とは，各プロセスがタスクの開始時や終了時に他の実行可能なタスクを見つけるものである．これに対し，集中スケジューリング方式ではスケジューリング専用のスレッドを設け，それ以外のスレッドはスケジューリングを行なわない．集中スケジューリング方式はスレッドを一元的に管理できるため効率の良いスケジューリングができる反面，スケジューリング用として専用のスレッドを用意しなければならない．一方，分散スケジューラでは，スケジュール用の共有データを用意し，すべてのスレッドがこの共有データに排他アクセスしスケジューリング作業を行なう．スケジューリング作業はスレッドの担当タスクの処理の終了後に行なうため，スケジューリング用スレッドといった本来の処理からすれば不必要なスレッドを用意する必要がなく，より多くのスレッドを利用できる．しかし，各スレッドの処理が増加してしまうため，パフォーマンスが低下する．今回実装したコンパイラでは，分散環境を想定し，分散スケジューリング方式を採用した．スケジューリングデータへの排他アクセスは POSIX Thread のロック機構である mutex 変数を用いる．実装したコンパイラでは，スケジューリング用データとして以下のデータを用意している．

関数間でグローバルなデータ

- スレッド使用状況 (thread_flag)

  各スレッドの使用状況 (使用中，待機中) を格納する．

関数内でローカルなデータ (同一コールによる関数内タスクでグローバルなデータ)

- タスク状態

wait	まだ先行のタスクが終了していなく，すぐには実行できない状態
ready	先行タスクがすべて終了し，スレッドさえ割り当てられればすぐに実行を開始できる状態
running	現在，実行中のタスク
ended	実行が終了し，後続タスクの実行に影響を与えない状態
unnecessary	分岐条件により，実行が不要になったタスク

- 先行タスク情報

  各タスクが終了を待たなければならないタスク番号を保持する．キュー形式になっており，該当するタスクが終了，または実行不要になるとキューからデータが除去される．キューにデータがなくなると，状態が ready に変化する．

これらのデータは実行時の最初のタスクで初期化される．また，ループの繰り返しごとにも子タスクの状態を初期状態に戻す．

スケジュールルーチンは関数間でグローバルなものと各関数でローカルなものに分られる．グローバルなものは，スレッド割り当ての許可やスレッド解放などの資源の管理を行なう．ローカルなルーチンはタスク状態の遷移やタスク本体の実行を担当する．

ready 状態のタスクは，スケジューラによりスレッド使用状況を調べアイドルとなっているスレッドに割り当てる．タスクは POSIX Thread の pthread_create 関数により，別スレッドとして生成され，すぐに pthread_detach 関数により呼び出し側関数の制御から離れる．よって，スレッド終了待機関数である pthread_join は使用していない．

3.4 バックエンド

バックエンド部では，各種解析処理の行なわれた SUIF コードを入力し，POSIX Thread により並列記述された C 言語ソースプログラムを生成する．生成されたコードは，タスク本体とスケジュールルーチンにより構成される．生成するコードのうちスケジュール部を 図 6に，タスク本体を 図 7 に示す．

スケジューラは，関数 create_thread と関数 check_status により構成される．関数 check_status は，各タスクの実行条件のチェックを行ない，先行タスクがなくなったものを順次 ready 状態にする．この関数の引数である sche_data はスケジュールに必要なデータの入った構造体である．また，タスク本体の起動は関数 create_thread により行なわれる．これらの関数はタスクごとにタスク本体の処理が終了後，呼び出される．

タスク処理関数では，最初にタスク自体の処理が実行され，その後自分の状態を ended にしてスケジューリングのための関数をコールする．

```
/* タスク状態ロック変数*/
pthread_mutex_t status_mutex;
/* タスクの実行 */
void create_thread(sche_data){
  int pid;/* プロセス番号 */
  for(すべてのタスク){
    pthread_mutex_lock(status_mutex);
    if(タスク状態が ready?){
      pid=get_thread();/* スレッドを確保 */
      if(スレッド確保失敗?)
        タスク実行処理から脱出;
      switch(タスク番号){
      case 1:
        /*タスク本体の実行*/
        pthread_create(...,block_1);
        break;
        ........
      case n:
        pthread_create(...,block_n);
        break;
      }
      pthread_detach();/*タスクを切り離す*/
    }
    pthread_mutex_unlock(status_mutex);
  }
}

/* タスク状態の遷移 */
void check_status(sche_data){
  for(すべてのタスク){
    pthread_mutex_lock(status_mutex);
    if(タスクが wait 状態?){
      do{
        if(先行が ended or unnecessary){
          先行タスクをキューから取り除く;
          if(キューが空?)
            タスク状態を ready に遷移する;
        }
        else
          break;
      }while(先行タスクが存在?);
    }
    pthread_mutex_unlock(status_mutex);
  }
}
```

図 6: 生成コード - スケジュール部

```
/*タスク状態ロック変数*/
pthread_mutex_t status_mutex;
/*タスク処理関数*/
void* block_1(shared_data){
  タスク本体の処理;
  pthread_mutex_lock(status_mutex);
  自分の状態を ended にする;
  pthread_mutex_unlock(status_mutex);
  スレッドの解放;
  /*他のタスク状態の遷移*/
  check_status(sche_data);
  /*他のタスクの実行/*
  create_thread(sche_data);
}
.....
void* block_n(shared_data){}
```

図 7: 生成コード - タスク処理関数

プログラム名	動作
test1	配列にランダムに数値を生成し，合計を求める．要素数は 30M.
test2	配列から配列へとデータをコピーする．要素数は 60M.

表 1: テストプログラム

4 評価

実装したコンパイラの粗粒度並列化による効果を確認するため，いくつかのテストプログラムを作成し，これを実装したコンパイラによりコンパイルし実行することで性能評価を行なった．使用したテストプログラムの内容は表 1の通りである．

実行環境には，UltraSparcII 300Mhz を 4 基搭載し，メモリ容量が 3GBytes の SUN ENTERPRISE を使用した．OS は SUN OS 5.6 である．

本テストでは，コンパイル前の逐次型プログラムとコンパイル後での性能比較を行なった．また，同時に実行するスレッド数を変化させ，全体性能に対する影響も調べた．

図 8にテストの結果を示す．縦軸は各テストプログラムの逐次型での実行時間を 1 として正規化してあり，逐次型と比較しての性能向上を表す．横軸は，実行時に生成できるスレッド数の最大値である．

図 8より，プログラム test1，test2 ともに性能の改

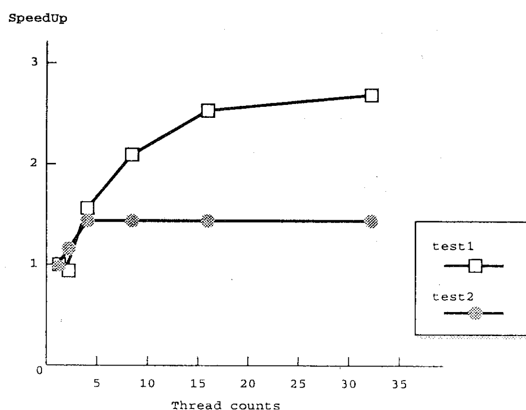

図 8: テストプログラムによる性能評価

善が見られた．また，test1 においては，スレッド数の増加による性能の向上が確認された．一方，test2 ではプロセッサ数である 4 スレッドまでは性能が向上するが，これ以上スレッドを増しても性能に変化はない．これは，test2 は比較的規模の小さい DOALL ループが複数あり、スケジューリングが頻繁に発生することにより、共有データへの排他アクセスのためにブロックされる時間が長くなることが原因と考えられる．test1 でも，スレッド数増加によるスケジューリングのオーバーヘッドを考えると，これ以上スレッド数を増しても性能向上には結びつかないと考えられる．同時に発生したスケジューリングをまとめるなどの，スケジューリングルーチンの最適化が必要である．

今回は，インタープロシージャ解析 [5] など関数間での依存解析は実装中であるため，より一般的なプログラムについての評価を取ることができなかった．実装が完了しだい評価を取る予定である．

5 おわりに

本研究では，自動並列化コンパイラである SUIF コンパイラに粗粒度並列化のためのプリプロセッサ，POSIX Thread ライブラリにより記述されたコードを出力するバックエンドを実装した．また，実装したコンパイラにより，テストプログラムをコンパイルし実行して評価を行なった．

今後の課題としては，並列性抽出を向上させるためのインタープロシージャ解析の実装，また各タスクの処理時間を考慮したスケジューリングアルゴリズムへの改善などが挙げられる．また，SUIF2 が最近公開されたので，その対応も現在検討中である．

参考文献

[1] Sutanford university: SUIF Homepage, http://www-suif.stanford.edu

[2] Mary W.Hall, Jennifer M.Anderson, Saman P.Amarasinghe, Brian R.Murphy, Shih-Wei Liao, Edouard Bugnion, Monica S.Lam: "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, December 1996

[3] 合田 憲人，岩崎 清，岡本 雅巳，笠原 博徳，成田 誠之助: "共有メモリ型マルチプロセッサシステム上での Fortran 粗粒度タスク並列処理の性能評価"，情報処理学会論文誌，Vol.37(No.3), p.418-429 (1996)

[4] 笠原 博徳："並列処理技術"，コロナ社 (1991)

[5] 松井 嚴徹，岡本 雅巳，松崎 秀則，小幡 元樹，吉井 謙一郎，笠原 博徳: "マルチグレイン並列処理におけるインタープロシージャ解析"，情報処理学会第 56 回 (平成 10 年前期) 全国大会 (2E-4), p.299-300 (1998)