計算機アーキテクチャ 142−19
ハイパフォーマンス 85−19
コンピューティング
（2001. 3. 9）

マクロデータフロープリプロセッサの実装

小野 徹行† 岩井 啓輔‡ 天野 英晴‡

†日立製作所 ‡慶應義塾大学理工学部

今日，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．慶應義塾大学天野研究室では現在，スタンフォード大学から提供されているSUIFコンパイラをベースにして粗粒度並列化の追加，中粒度並列化の改良を行なった自動並列化コンパイラGarafraseの開発が進められている．このGarafraseは，いくつかのテストプログラムによって動作確認が行なわれているものの，まだ実用的に使用できる段階ではなかった．本研究ではGarafraseのマクロデータフロー処理を完成させ，より一般的なプログラムにも対応可能とした．またNAS Parallelベンチマークによる評価を行ない，並列化による処理速度の向上を確認した．

Implementation of Macro data flow pre-processor

Tetsuyuki Ono† Keisuke Iwai‡ Hideharu Amano‡

†Hitachi, Ltd. ‡Keio University

In order to make the best use of recent multiprocessors performance, researches on automatic parallelizing compiler have been widely exerted. We developed an automatic parallelizing compiler "Garafrase" 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 "Garafrase" and the performance evaluation results of the NAS Parallel Benchmark 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．

慶應義塾大学天野研究室では，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Garafrase[1]の開発が進められている．このコンパイラは，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[2][3]をベースに粗粒度並列化などの機能を付加する形で実装が行なわれている．

本報告では，自動並列化コンパイラGarafraseの実装と，NAS Parallelベンチマークを用いて行なった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

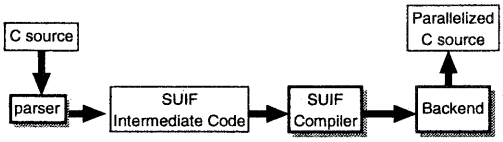

図 1: SUIF コンパイラの処理の流れ

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) と呼ばれる木構造で表現されている．なお，SUIF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

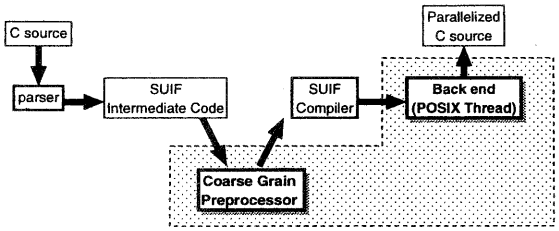

図 2: Garafrase の構成

3 Garafraseの構成

SUIFコンパイラは，ループレベルでの並列化を行なう中粒度並列化をサポートしているが，ループの並列化だけではプログラムの並列性を完全に引き出すことはできない．そこで，SUIFコンパイラのような既存のコンパイラよりも効率的な並列化を実現させるため，自動並列化コンパイラGarafraseの開発が行なわれている．

GarafraseはSUIFコンパイラをベースとして，マクロデータフロー処理[4]を行なうプリプロセッサ部と，POSIX Threadライブラリを用いた並列コードを出力するバックエンド部を追加するという形で実装が行なわれている．Garafraseの構成を図 2に示す．

プリプロセッサ部では，SUIFコードを読み込みマクロデータフロー処理を行なう．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．

また，Garafraseのバックエンド部では，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生成し，出力する．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．

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などの並列性を持つループも含まれる．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，抽出するタスク情報について述べる．

- 変数の利用情報
- 分岐情報
- 階層情報

変数の利用情報とは，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．

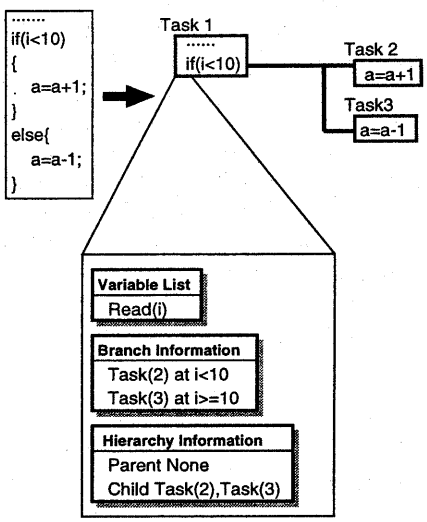

図 3: タスク固有情報の例

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 [5] によるデータ依存解析を行なっている．

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 [4] を生成する．Garafrase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 4に示す．

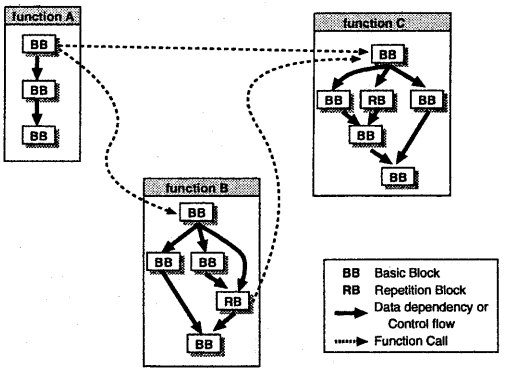

図 4: 関数ごとのマクロフローグラフの例

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[4]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，分散スケジュール方式では，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ことなく，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個々のスレッドのパフォーマンスは若干低下する．

Garafrase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ている．

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．

- 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．

- 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，スレッドが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．

スケジュールルーチンでは，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，また，タスク状態の遷移やタスク本体の実行を行なう．

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．

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5に，タスク本体を図6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waitから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2つのスケジュールルーチンは，各タスクの処理の最後に呼び出される．

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 5: 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 6: 生成コード - タスク処理関数

4 評価

Garafraseの粗粒度並列化による効果を確認するため，実際にプログラムのコンパイルを行ない，実行時間の測定を行なった．使用するテストプログラムにはNAS Parallelベンチマークの ISを用いている．ISは整数値のソートを行なうプログラムで，処理の対象となる配列の要素数はclassS～classCの5種類が用意されている (表 1).

class	array size
class S	2^{16}
class W	2^{20}
class A	2^{23}
class B	2^{25}
class C	2^{27}

表 1: 配列サイズ (IS)

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，オリジナルの逐次型プログラムとGarafraseによって生成された並列プログラム間での

性能比較を行なった．

図 7に，各クラスごとに行なった評価の結果を示す．縦軸は実行時間を表しており，逐次型プログラムの値を1として正規化したものである．

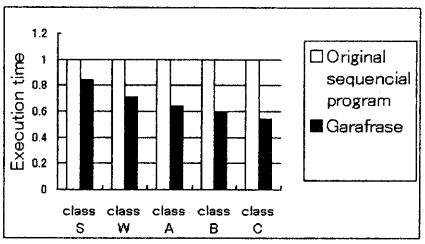

図 7: 実行結果 (NAS Parallel: IS)

グラフより，Garafraseによる実行時間の短縮が確認できる．また，データサイズが大きくなるにつれ，並列化の効果も大きくなっていくという結果も見てとれる．これは，個々のマクロタスクの規模が大きくなることによりスレッドの生成等によるオーバーヘッドが隠蔽されるためであると考えられる．

さらに，サブルーチンごとの実行時間の比較も行なった (図 8)．ISに含まれるサブルーチンのうち full_verify() と rank() というサブルーチンについて，class Cのプログラムを用いて比較した．グラフから rank() では約4倍の速度の向上が確認できるが，使用したプロセッサ数は4つであることから，これはほぼ理想的な値と言える．

一方 full_verify() では，むしろ若干の性能の悪化が見られる．これは，full_verify() には同時に実行可能なマクロタスクがほとんど無く，またループブロック (RB) についてはDOALLループが一つもないために，粗粒度並列化，中粒度並列化のどちらの効果も得られなかったためと考える．

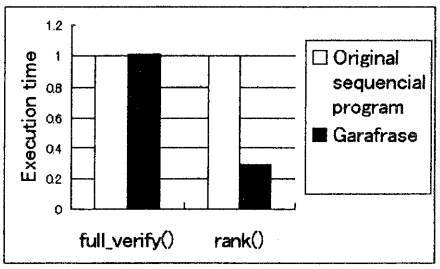

図 8: サブルーチンごとの実行時間の比較 (class C)

このように並列性の高いルーチンと低いルーチンが混在し，全体としては class C で約1.9倍のスピードアップとなっている．今回の評価では以上のような結果となったが，テストプログラムの性格によってはこれ以上の性能向上が期待できるはずである．今後は他のベンチマークプログラムについても評価を行なっていく予定である．

5 おわりに

本研究では，自動並列化コンパイラ Garafrase においてマクロデータフロー処理を完成させ，粗粒度並列化を実現させた．また，NAS Parallel ベンチマークによる評価を行ない処理速度の向上を確認した．

今後の課題としては，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 への対応も現在検討中である．

参考文献

[1] 中村柄 真人，岩井 啓輔，天野 英晴: "祖粒度並列化プリプロセッサの実装"，第80回ハイパフォーマンスコンピューティング研究会，第129回計算機アーキテクチャ研究会 (2000)

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

[3] 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

[4] 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)

[5] 笠原 博徳 : "並列処理技術"，コロナ社 (1991)

[6] 松井 巌徹，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)