Vol.2009-ARC-184 No.15
2009/8/5

# マルチコアにおけるParallelizable Cプログラムの自動並列化

間瀬 正啓[†1]　木村 啓二[†1]　笠原 博徳[†1]

本稿ではコンパイラによる自動並列化を可能とするためのC言語の記述方法としてParallelizable Cを提案する．Parallelizable Cで記述した科学技術計算およびマルチメディア処理の逐次プログラム6本に対してOSCARコンパイラによる自動並列化を適用し，マルチコアシステム上での処理性能の評価を行った．その結果，逐次実行時と比較して，2コア集積のマルチコアであるIBM Power5+ を4基搭載した8コア構成のサーバであるIBM p5 550Qにおいて平均5.54倍，4コア集積のマルチコアであるIntel Core i7 920プロセッサを搭載したPCにおいて平均2.43倍，SH-4Aコアベースの情報家電用マルチコアRP2の4コアを使用したSMP実行モードにおいて平均2.78倍の性能向上が得られた．

# Automatic Parallelization of Parallelizable C Programs on Multicore Processors

Masayoshi Mase,[†1] Keiji Kimura[†1]
and Hironori Kasahara[†1]

This paper proposes Parallelizable C, a guideline for writing C programs which enables automatic parallelization by a compiler. 6 sequential programs written in Parallelizable C from numerical and multimedia application domains are automatically parallelized by OSCAR compiler. The parallel processing performance for these applications are evaluated on multicore systems. The evaluation results show that the compiler automatic parallelization achieves average 5.54 times speedup on a 8 cores server IBM p5 550Q with 4 dual-core Power5+ processors, average 2.43 times speedup on a 4 cores multicore processor PC with Intel Core i7 920, and average 2.78 times speedup on Renesas/Hitachi/Waseda RP2 with SH-4A cores in SMP execution mode using 4 cores compared with sequential execution, respectively.

## 1. はじめに

マルチコアプロセッサを含め，マルチプロセッサシステム用の並列プログラミングは難易度が高く，所望の性能を得るために長期間の並列チューニングが必要であることで知られており，並列ソフトウェア生産性向上が大きな課題となっている．そのため，CやFORTRANのような既存の逐次型プログラムをコンパイラで自動並列化することによる，短時間での高性能の実現が強く期待されている．

従来より科学技術計算分野のFORTRANプログラムに対しては，Poralis[1)]，SUIF[2)]，OSCAR[3)]といった高性能な自動並列化コンパイラが開発され，多くの並列マシン上で，その有効性が確認されてきた．しかしながら，C言語ではポインタを多用した自由な記述が可能であり，コンパイラによるポインタ解析において，ポインタの指し先を静的に特定するのは非常に困難である[4)]．冗長なポインタエイリアス情報が存在することにより，コンパイラによる依存解析の精度が劣化することが，自動並列化の阻害要因となっている．

そこで，Cプログラムの記述を制約することで自動並列化コンパイラによる階層的な並列性の利用やキャッシュ，ローカルメモリ最適化といった高度な並列処理手法の利用を可能とするために，Cプログラム記述のガイドラインとしてParallelizable Cを提案する．Parallelizable Cでプログラムを記述することにより，プログラマはアルゴリズムのチューニングに専念し，煩雑な並列処理向けのテクニックはコンパイラにより自動で適用可能となる．

C言語の記述の自由度を用途に合わせて制限するコーディングガイドラインは，主に車載ソフトウェア向けの高信頼性，高安全性，高移植性を目的としたMISRA-C[5)]に代表されるように，産業界で広く普及している．マルチプロセッサ向けの自動並列化においても，IMECによるマルチプロセッサ向けのコーディングガイドラインであるClean C[6)]や，Hwuらにより assertion 等によるヒント情報を用いて自動並列化コンパイラを有効利用する暗黙的な並列プログラミングモデル[7)]が提案されている．このようなガイドラインを設定する場合に，そのプログラム記述が現実的かどうか，つまり実際にどの程度の自動並列化の効果が得られるか，そしてその記述法がプログラマに不自由なプログラム記述を強制していないかどうかが，そのガイドラインの普及に向けた課題となる．IMECやHwuらの提案では，

†1 早稲田大学 基幹理工学部 情報理工学科
Dept. of Computer Science and Engineering, Waseda University



© 2009 Information Processing Society of Japan

アプリケーションプログラムを用いたコード書き換えや並列処理による性能向上の定量的な評価がなされておらず，ガイドラインの有効性の明確化は行っていなかった．

本稿で提案する Parallelizable C は C 言語のサブセットとして定義され，C 言語の処理系で正常にコンパイルし実行可能である．Parallelizable C を利用したプログラム開発では，フルセットの C プログラムのうち，並列化を意識したい部分を Parallelizable C 言語で書き換えることで，それに応じた自動並列化を可能とすることを目指している．並列処理による高速化が期待されている，科学技術計算およびマルチメディア処理の逐次プログラムについて書き換えを行ったところ，それらは少量の書き換えにより，コンパイラによる自動並列化が適用可能であることを確認した．

本稿の構成を以下に示す．まず第 2 章では C プログラムの自動並列化のためのポインタ解析手法について述べ，第 3 章では Parallelizable C の概念について述べる．次に第 4 章では Parallelizable C の仕様について述べ，第 5 章では Parallelizable C へのプログラム書き換えについて述べる．そして，第 6 章では性能評価に利用した OSCAR コンパイラの概要について述べ，第 7 章では Parallelizable C プログラムに対して OSCAR コンパイラの自動並列化を適用した際のマルチコアシステム上での処理性能について述べる．最後に第 8 章でまとめを述べる．

## 2. C プログラムの自動並列化のためのポインタ解析手法

ループ並列性の解析，データフロー解析，データアクセス範囲解析等の並列性抽出のためのプログラム解析[8] の精度は C 言語ではポインタ解析の精度に大きく依存する．本章では，提案する Parallelizable C の仕様を決定するにあたり必要となるポインタ解析の現状とその適用範囲を簡単にまとめる．

ポインタ解析はプログラムの内部状態や指し示される領域に関する情報の持たせ方によって分類され，一般的に解析精度と解析コストのトレードオフとなる[4]．

### 2.1 コントロールフローに対する解析精度

ポインタ解析はデータフロー解析の一つであり，コントロールフローに沿って解析が行われる．コントロールフローに対する精度の分類として，flow-sensitivity と context-sensitivity がある．

flow-sensitive な解析では，プログラムのコントロールフローに沿って解析を行い，各ステートメントごとに情報を生成していく．context-sensitive な解析では，関数呼び出しに対して，呼び出し元ごとに解析を行う．flow-sensitive, context-sensitive のアルゴリズムとしては Emami ら[9] が提案したアルゴリズム等が挙げられる．

flow-sensitive，context-sensitive の解析では膨大なメモリ使用量と解析時間が課題となるが，効率を向上させるための様々な手法[10)–12)] が提案され，実用的な時間で解析可能となってきている．

### 2.2 データ構造に対する解析精度

ポインタ解析の情報の持たせ方として，ヒープや構造体に対してどこまで細かい単位で識別を行うかが，ポインタ解析の精度に大きく影響する．このようなデータ構造に対する精度の分類として heap-sensitivity と field-sensitivity がある．

ヒープについては，malloc() 等のメモリ動的確保を行う関数呼び出しのあるステートメントごとに識別するのが一般的である．ただし，ヒープを確保する際にはラッパ関数を用いることが多いため，実際にはラッパ関数を呼び出してから malloc() 等のメモリ動的確保の関数呼び出しまでのコールパスを考慮して，ヒープ情報のクローニングを行うことで精度を高める必要がある．これを heap-cloning[13] と呼び，ヒープを利用したデータ構造を中心としたアプリケーションの解析で必要となる．

field-sensitive なポインタ解析[14] では構造体のメンバ間の要素を区別して解析することが可能となる．プログラム中で使用するデータを一つの構造体にまとめているようなアプリケーションの解析で必要となる．

### 2.3 ループイタレーションに対する解析精度

データフロー解析ではループの収束演算により，そのイタレーションで確保したヒープ領域と前のイタレーションで確保したヒープ領域は，同一領域と解析されてしまう．そこで，確保されたヒープオブジェクトについて，現在のイタレーションで確保されたものなのか，前のイタレーションで確保されたものなのかを区別する aging[15] というテクニックがある．また，ポインタの配列の各要素とヒープオブジェクトを確保したループイタレーションの情報をマッピングする element-wise points-to set[16] がある．

### 2.4 再帰的なデータ構造の解析

リスト構造や木構造のように自身の構造体型へのポインタを持つような，再帰的なデータ構造に特化したポインタ解析が存在する．shape analysis[17] ではポインタの指し先の具体的なオブジェクトを解析するのではなく，ポインタが指し示すデータ構造がリストなのか木構造なのか DAG なのか，といったデータ構造の解析を行う．


© 2009 Information Processing Society of Japan

## 3. Parallelizable C の概念

第 2 章で述べた各種ポインタ解析の実装により，コンパイラによる自動並列化の適用範囲が拡大可能になっていくと考えられる．しかし，C 言語のあらゆる仕様を網羅するポインタ解析の実装は事実上不可能であり，自動並列化コンパイラを現実的に利用していく上では，解析可能な C プログラムの記述を明確化し，その範囲でプログラムを記述する必要がある．

### 3.1 C 言語仕様としての位置づけ

Parallelizable C は C 言語のサブセットとして定義され，C 言語の処理系で正常にコンパイルし実行可能である．本稿で提案する Parallelizable C を利用したソフトウェア開発では，C プログラムのうち，並列化を意識したい部分を Parallelizable C の要件を満たすように書き換えることで，それに応じた並列性抽出を可能とすることを目指している．

### 3.2 Parallelizable C の段階

Parallelizable C の記述は，対象のコンパイラの解析器の精度に大きく左右されるため，レベル分けが行われる．コンパイラの解析精度が高くなるほど，プログラム記述ルールの制約が緩和され，プログラマの負荷を軽減することができる．

#### 3.2.1 Parallelizable C レベル 1 (制約付き C)[18)]

コンパイラでポインタ解析を行わず，構造体についてもメンバごとの区別を行わないことを想定したコーディングガイドラインである．この場合，Parallelizable C 仕様は以下のような C 言語記述のルールから成り立つ．

- 配列を参照渡しする場合の関数のポインタ引数を除き，ポインタを使用しない
- 構造体を使用しない
- 再帰関数呼び出しを使用しない

上記のルールに沿ってプログラムを記述することで FORTRAN77 言語と同等のプログラム記述となり，FORTRAN 向けに開発された自動並列化技術がそのまま適用可能となる．

#### 3.2.2 Parallelizable C レベル 2

本稿で新たに提案するコーディングガイドラインであり，コンパイラにポインタ解析が実装されており，構造体はメンバごとに区別を行って解析することを想定したコーディング規則である．本稿では以下のようなポインタ解析機能を持つコンパイラを想定して Parallelizable C 仕様を決定する．

- ステートメントごとにコントロールフローに沿った解析を行う (flow-sensitive)
- 関数の呼び出し箇所ごとに解析を行う (context-sensitive)
- メモリ動的確保までの関数呼び出し経路ごとにヒープオブジェクトを区別する (heap-cloning)
- 構造体のメンバをそれぞれ別のオブジェクトとして扱う (field-sensitive)
- ポインタがオブジェクトの先頭を指すかどうかを判別する
- ポインタの配列について各要素の指し先に重なりがないかを判別する
- ヒープオブジェクトを確保したイタレーションを判別する

また，構造体の解析においては以下のような機能を持つものとする．

- スカラとして扱う構造体のメンバは別のオブジェクトとして扱う
- 配列として扱う構造体はそれ自体を配列オブジェクトとして扱い，配列要素内のメンバの区別は行わない

ここで，ポインタの指し先の領域についても，そのポインタからの逆参照時のオフセットが必ずゼロとなるような領域はスカラ，オフセットがゼロ以外になる場合がある領域は配列として扱うものとする．

Parallelizable C レベル 2 の仕様の詳細は第 4 章で述べる．また，shape analysis 等の再帰的なデータ構造に対するポインタ解析を利用することで将来的にはコーディングルールのさらなる制約緩和が可能と考えられるが，ここでは対象外とする．

## 4. Parallelizable C レベル 2 の仕様

本章ではポインタ解析をサポートしたコンパイラを想定した，Parallelizable C レベル 2 の仕様について述べる．以下，Parallelizable C とはこのレベル 2 の仕様を示すこととする．本仕様は C 言語記述におけるルールと，コンパイル時の外部関数に対するヒント情報から構成される．

### 4.1 C 言語記述におけるルール

Parallelizable C では本節で述べるルールに従ってプログラムの記述を行うものとする．ただし，Parallelizable C はフルセットの C 言語に対応した処理系でコンパイルされるため，プログラマの判断でルールを逸脱することも許されるが，コンパイラによる自動並列化は困難になる．

Parallelizable C のルールとともに，MISRA-C[5)]，Clean C[6)] において関連する項目がある場合は，併せて示すことにする．MISRA-C は組み込みシステム向けの高信頼性のためのガイドラインであり，Parallelizable C とは用途が異なるが，共通するコーディングルールを含んでいる．Clean C は Parallelizable C と同様にマルチコア向けの自動並列化のための


© 2009 Information Processing Society of Japan

ガイドラインであり，Parallelizable C のルールよりも抽象的なルールとなっている．

以下に，Parallelizable C の C 言語記述におけるルールを示す．

#### 4.1.1 変数宣言の境界を超えたアクセスを行わない

構造体のメンバや多次元配列の各次元の境界を越えたアクセスを行わない．C コンパイラでは処理系により正常動作することもあるが，C 言語仕様上は未定義である．

MISRA-C, Clean C でも C 言語で未定義の動作は避けることを規定している．

#### 4.1.2 ポインタのキャストを行わない

メモリ動的確保時を除いて，ポインタのキャストを行わない．ポインタ参照先のオブジェクトのオフセットを正しく解析するためである．

MISRA-C ではオブジェクト型ポインタと整数型以外の型間でのキャストは禁止している．また Clean C ではポインタ型とその他の型とのキャストを禁止している．

#### 4.1.3 ポインタ算術演算を行わずに添字アクセスを行う

ポインタ算術演算は行わない．ポインタ指し先の領域にアクセスする場合にはポインタ変数の値は更新せずに，ポインタ変数に対する逆参照演算子 (*) もしくは添字アクセス ([ ]) により行うこととする．

MISRA-C, Clean C においても，ポインタ算術を禁止している．

#### 4.1.4 条件分岐やループ内でポインタ変数への値の代入を行わない

メモリ動的確保時を除いて，条件分岐やループ内ではポインタ変数への値の代入を行わない．

Clean C ではポインタの指し先はプログラム全体で一意に定まることとしている．

#### 4.1.5 同一領域に対する複数のポインタを関数の引数として渡さない

配列の単一次元内の異なるオフセットのアドレスを関数の引数として渡さない．ただし，同一配列でも重なりのない部分配列であればよい．これは，２つの引数の指し先を別々のオブジェクトとして扱えなくなるためである．

MISRA-C では複数のポインタの指し先がそれぞれ単一のデータセットであることとしている．

#### 4.1.6 一時バッファとして利用するヒープ領域を使い回さない

ヒープ領域の使い回しは，通常の変数の使い回しよりも解析が困難であり，並列性抽出の阻害要因となる．ヒープ領域を使い回す例としては，ループのボディにおいてのみ利用するバッファ領域を複数ループイタレーションに渡って確保し続けて再利用する，といったことが挙げられる．

Clean C ではグローバル変数は利用せず，ローカル変数を利用することとしているが，本ルールはヒープ領域に関して有効範囲を明示するという意味で，関連する項目と言える．

#### 4.1.7 構造体の配列のメンバに対する配列アクセスを行わない

配列メンバを持つ構造体の配列を使用しない．構造体のメンバに配列がある場合は，下位の関数に構造体の１要素にあたる領域を引数として渡し，下位の関数内でそのメンバの配列へのアクセスを行う．これは配列要素間と同様に，構造体のメンバ間の識別精度も解析精度に影響するためである．

#### 4.1.8 再帰的なデータ構造を利用しない

リストや木構造等，再帰的に同じ型の構造体オブジェクトへのリンクがあるデータ構造を使用しない．リストや木構造であれば並列処理可能なことがあるが，リストや木構造の解析には専用の解析が必要となる．配列ベースの記述であればコンパイラによる高度な最適化が適用できるため，配列型のデータ構造に変換してから計算処理を行うこととする．

#### 4.1.9 呼び出し間に依存のある外部関数を使用しない

ファイル入出力，エラー処理等の，ライブラリ内で内部状態を持ち，呼び出しごとに依存がある外部関数を使用しない．エラー処理やログ出力は並列化したい部分の外側にくくり出す等の工夫が必要である．

#### 4.1.10 関数の再帰呼び出しを使用しない

関数は直接的か間接的かに関わらず，その関数自体を呼び出さない．コールグラフを静的に決定できなくなったり，メモリ使用量の見積もりが難しくなったりするためである．

MISRA-C，Clean C においても再帰呼び出しを禁止している．

#### 4.1.11 関数ポインタを使用しない

関数ポインタを使用せず switch-case を用いて関数呼び出しを記述する．これは関数の呼び出し先の静的な解析が難しくなるためである．

Clean C においても関数ポインタの使用を禁止している．

#### 4.1.12 可変個引数を持つ関数を定義しない

可変個引数には未定義の動作が多いためである．

MISRA-C，Clean C においても使用を禁止している．

### 4.2 外部関数に関するヒント情報

プログラム全体に対して解析を行うことを前提としているが，ライブラリ関数等を利用する場合は，そのライブラリ関数による挙動を指示することでコンパイラの解析精度劣化を防ぐことができる．そのような指示を行う方法としては，以下のようなコンパイルオプション


© 2009 Information Processing Society of Japan

が挙げられる．

- 外部関数においてポインタ引数の指し先は変化しないことを仮定するオプション
- 外部関数からの戻り値がポインタ型の場合，そのポインタの指し先は新たに領域確保されたヒープ領域であることを仮定するオプション

## 5. Parallelizable C へのプログラム書き換え

本章では SPEC2000 より art, equake, SPEC2006 より lbm, hmmer, MediaBench より mpeg2encode を例に，Parallelizable C レベル 2 への書き換えについて述べる．

各アプリケーションの書き換えにおいては，まず最初にファイルを一つにまとめて，入出力や時間計測用のコードを整備した参照コードを作成し，それに対して書き換えを行った際の修正箇所を調べた．

書き換えの過程においては，Parallelizable C に適合するための書き換えに加えて，プログラムの持つ並列性を十分に抽出するためのプログラム構造の書き換えも行った．

### 5.1 コード書き換え内容

SPEC2000 の art については，オリジナルコードの時点で Parallelizable C のルールに適合していたため，書き換えは行わなかった．

SPEC2000 の equake については，Parallelizable C のルールには適合していたが，Parallelizable C コードでは，プログラム実行時間の大部分を占める関数 smvp() において，配列全体に対するリダクション処理による並列化を行うために手動でリストラクチャリングを行った．

SPEC2006 の lbm については，配列の途中を指すポインタと，ループ中におけるポインタ変数の更新があったため，Parallelizable C コードではそれらの除去を行った．図 1(a) のようなループ中でのポインタ更新については，図 1(b) のようにフラグ変数と条件分岐を用いて修正した．

SPEC2006 の hmmer については，図 2(a) のように，複数イタレーションに渡ってバッファ領域を使い回していたため，Parallelizable C コードでは図 2(b) のように各イタレーションごとに領域を確保して解放するように修正した．

mpeg2encode では，MPEG2 エンコードアルゴリズムが持つマクロブロックレベルの並列性とデータローカリティが利用不可能なプログラム構造となっているため，並列性抽出のためのプログラム構造の変更[19] を併せて行った．

```
for( t = 1; t <= param.nTimeSteps; t++ ) {
    …
    LBM_performStreamCollide( *srcGrid, *dstGrid ); /* 主要計算部 */
    LBM_swapGrids( &srcGrid, &dstGrid ); /* srcGrid と dstGrid のポインタ値を交換する */
    …
}
```

(a) オリジナルコード

```
flg = 0;
for( t = 1; t <= param.nTimeSteps; t++ ) {
    …
    if (flg % 2)
        LBM_performStreamCollide( *dstGrid, *srcGrid ); /* 主要計算部 */
    else
        LBM_performStreamCollide( *srcGrid, *dstGrid ); /* 主要計算部 */
    flg++;
    …
}
```

(b) Parallelizable C コード

**図 1** lbm におけるループ中でのポインタの更新の除去
Fig. 1 Removing pointer update inside a Loop in lbm

## 6. OSCAR 自動並列化コンパイラの概要

OSCAR コンパイラ[3] は C や FORTRAN 言語で記述された逐次ソースプログラムを入力し，並列プログラムを自動生成する自動並列化コンパイラである．従来は，FORTRAN77 向けに開発されていたが，新たに C フロントエンド，C ソースコード生成部，ポインタ解析器の実装等の拡張を行うことで，C 言語の自動並列化にも対応した．

OSCAR コンパイラではループイタレーションレベルの並列処理を行うのみでなく，ループ・手続き間の粗粒度タスク並列処理[20]，ステートメント間の近細粒度並列処理[21] を組み合わせたマルチグレイン並列処理[3]，メモリウォール問題に対処するための複数ループにわたるキャッシュあるいはローカルメモリの最適利用[22),23] が実現されている．さらに，プログラム中の各並列処理部に対する適切なリソース割り当てや，各リソースの周波数・電圧・電源制御による消費電力の自動削減[24] が実現されている．

## 7. マルチコアシステム上での性能評価

本章では，Parallelizable C プログラムを OSCAR コンパイラで自動並列化した際の PC，サーバ，組み込み機器向けのそれぞれのマルチコアシステム上における使用するコア数と処理性能について評価を行う．


© 2009 Information Processing Society of Japan

```
static void main_serial_loop() {
    mx = CreatePlan7Matrix(1, hmm->M, 25, 0); /* P7Viterbi() で使用するデータ構造を malloc */
    for (idx = 0; idx < nsample; idx++)
    {
        ...
        score = P7Viterbi(..., mx);  /* 主要計算部 */
        ...
    }
    ...
}
static float P7Viterbi(char *dsq, int L, struct plan7_s *hmm, struct dpmatrix_s *mx) {
    ResizePlan7Matrix(mx, ...); /* 作業領域を realloc により使い回す */
    ... /* score の計算 */
    return score;
}
```

(a) オリジナルコード

```
static void main_serial_loop() {
    for (idx = 0; idx < nsample; idx++)
    {
        ...
        score = P7Viterbi(...);  /* 主要計算部 */
        ...
    }
}
static float P7Viterbi(char *dsq, int L, struct plan7_s *hmm) {
    struct dpmatrix_s *mx;
    mx = AllocPlan7Matrix(...); /* イタレーションごとに作業領域を malloc */
    ... /* score の計算 */
    FreePlan7Matrix(mx); /* イタレーションごとに作業領域を free */
    return score;
}
```

(b) Parallelizable C コード

**図 2** hmmer におけるバッファの使い回しの除去
Fig. 2 Removing buffer reuse in hmmer

### 7.1 評価方法

OSCAR コンパイラによる並列化の際には，データローカリティ最適化を含めたマルチグレイン並列化を適用し，OpenMP[25] で並列化された C プログラムを出力する．このときに利用する OpenMP 指示文は OSCAR API[26] で規定されている OpenMP 指示文のサブセット部分にあたる．

自動生成された OpenMP C (OSCAR API C) プログラムを対象プラットフォームの OpenMP あるいは OSCAR API に対応したネイティブコンパイラによりコード生成を行う．また，ネイティブコンパイラにおいて自動並列化をサポートしている環境については，ネイティブコンパイラによる自動並列化結果も併せて示す．

**表 1** 評価環境
Table 1 Evaluation environment

| System | IBM p5 550Q | Intel Core i7 920 | Renesas/Hitachi/Waseda RP2 |
|---|---|---|---|
| CPU | Power5+ (1.5GHz × 2 × 4) | Nehalem (2.66GHz × 4) | SH-4A (600MHz × 4) |
| L1 D-Cache | 32KB for 1 core | 32KB for 1 core | 16KB for 1 core |
| L1 I-Cache | 64KB for 1 core | 32KB for 1 core | 16KB for 1 core |
| L2 cache | 1.9MB for 2 cores | 256KB for 1 core | |
| L3 cache | 36MB for 2 cores | 8MB for 4 cores | |
| Native Compiler | IBM XL C/C++ for AIX Compiler V10.1 | Intel C/C++ Compiler verion 11.0 | SH C Compiler + OSCAR API Parser |
| Compile Option | OSCAR: -O5 -qsmp=noauto Native: -O5 -qsmp=auto | OSCAR: -fast -openmp Native: -fast -parallel | |

### 7.2 評価対象プログラム

第 5 章で述べた，SPEC2000 より art，equake，SPEC2006 より lbm，hmmer，MediaBench より mpeg2encode の 5 種類の C プログラムについて，オリジナル C コードと Parallelizable C コードそれぞれに対し，OSCAR コンパイラによる自動並列化を適用した際の処理性能を示す．また，本稿では書き換えを行っていないが，Parallelizable C を満たすように実装されている音声圧縮プログラムである AAC エンコード (AACencode) についても，併せて評価を行う．AAC エンコードは株式会社ルネサステクノロジ提供のプログラムであり，関数の引数ポインタ以外のポインタ・構造体を原則的に使用せず (Parallelizable C レベル 1) に製品レベルのミドルウェア仕様を参照実装したものである[18]．

### 7.3 評価環境

本評価においては，8 コア搭載の SMP サーバである IBM p5 550Q，Intel の 4 コア CPU である Core i7 を 1 基搭載した PC，ルネサステクノロジ，日立製作所，早稲田大学で共同開発した情報家電用マルチコア RP2[27] における SH-4A を 4 コア構成による SMP モードの 3 つの環境で評価を行った．各環境におけるパラメータを表 1 に示す．IBM p5 550Q では，1 プロセッサあたり 2 スレッド実行の Simultanemous Multi-Threading(SMT) が可能であるが，本評価で SMT は用いないものとした．同様に，Intel Core i7 においても 1 プロセッサあたり 2 スレッド実行の Hyper-Threading が可能であり，またチップ温度を計測してハードウェアで自動的にオーバークロックする Turbo Boost が利用可能であるが，本評価では共に用いないものとした．

### 7.4 自動並列化による処理性能

オリジナルコードと Parallelizable C に書き換えたコードそれぞれに対して，OSCAR コンパイラで自動並列化を適用した際の処理性能を示す．IBM p5 550Q における処理性能を


© 2009 Information Processing Society of Japan

図 3 に，Intel Core i7 における処理性能を図 4 に，ルネサステクノロジ／日立製作所／早稲田大学 RP2 における処理性能を図 5 に示す．

図中，横軸が各アプリケーションとコードの種類を示し，縦軸はオリジナルコードをネイティブコンパイラの 1 コアで逐次実行した場合に対する速度向上率を示す．横軸の各項目内のバーは使用しているプロセッサコア数を示し，それぞれ左から 1PE，2PE，4PE，8PE となる．

#### 7.4.1 IBM p5 550Q における処理性能

OSCAR コンパイラでは各アプリケーションのオリジナルコードに対して，8 コア使用時に逐次実行時と比較して，art で 4.96 倍，equake で 1.69 倍，mpeg2encode で 1.53 倍の性能向上を得ることができた．さらに，Parallelizable C で書き換えを行うことで，オリジナルコードの逐次実行時と比較して equake で 5.61 倍，lbm で 5.35 倍，hmmer で 6.06 倍，mpeg2encode で 5.12 倍の性能向上が得られた．また，元々Parallelizable C で記述されている AAC エンコーダでは 6.16 倍の性能向上が得られた．以上をまとめると，Parallelizable C で記述されたプログラムを OSCAR コンパイラで自動並列化することで，逐次実行と比較して平均 5.54 倍の性能向上が得られた．

なお，IBM p5 550Q の評価においては，equake のみ最適化レベルを-O4 としている．これは，Parallelizable C コードを OSCAR コンパイラで生成した OpenMP プログラムを IBM XL コンパイラで-O5 でコンパイルした際に正常に実行できなかったためである．本評価では並列処理による性能向上を評価するため，equake の全評価において最適化レベルを-O4 に統一した．

#### 7.4.2 Intel Core i7 における処理性能

OSCAR コンパイラではオリジナルコードに対して，4 コア使用時に逐次実行時と比較して art で 1.51 倍，equake で 1.15 倍，mpeg2encode で 1.81 倍の性能向上を得ることができた．さらに，Parallelizable C で書き換えを行うことで，オリジナルコードの逐次実行時と比較して equake で 1.45 倍，lbm で 1.28 倍，hmmer で 3.34 倍，mpeg2encode で 3.60 倍の性能向上が得られた．また，元々Parallelizable C で記述されている AAC エンコーダでは 3.48 倍の性能向上が得られた．以上をまとめると，Parallelizable C で記述されたプログラムを OSCAR コンパイラで自動並列化することで，逐次実行と比較して 4 コア使用時に平均 2.43 倍の性能向上が得られた．

#### 7.4.3 ルネサステクノロジ／日立製作所／早稲田大学 RP2 における処理性能

OSCAR コンパイラではオリジナルコードに対して，4 コア使用時に逐次実行時と比較し

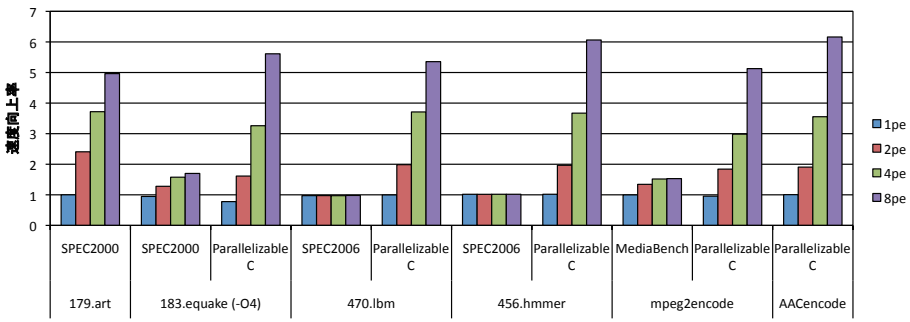


**図 3** IBM p5 550Q における OSCAR コンパイラによる自動並列化結果
Fig. 3 Automatic parallelization results for OSCAR Compiler on IBM p5 550Q

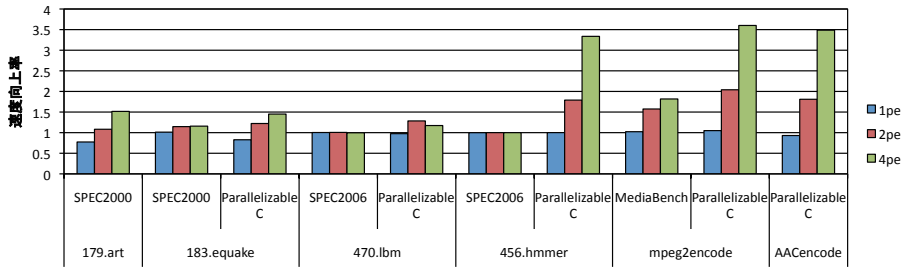


**図 4** Intel Core i7 920 における OSCAR コンパイラによる自動並列化結果
Fig. 4 Automatic parallelization results for OSCAR Compiler on Intel Core i7 920

て art で 2.53 倍，equake で 1.33 倍，mpeg2encode で 1.61 倍の性能向上を得ることができた．さらに，Parallelizable C で書き換えを行うことで，オリジナルコードの逐次実行時と比較して equake で 1.95 倍，mpeg2encode で 3.27 倍の性能向上が得られた．また，元々 Parallelizable C で記述されている AAC エンコーダでは 3.34 倍の性能向上が得られた．以上をまとめると，Parallelizable C で記述されたプログラムを OSCAR コンパイラで自動並列化することで，逐次実行と比較して 4 コア使用時に平均 2.78 倍の性能向上が得られた．

なお，SPEC2006 のアプリケーションについては，本稿では RP2 上での評価を行っていない．


© 2009 Information Processing Society of Japan

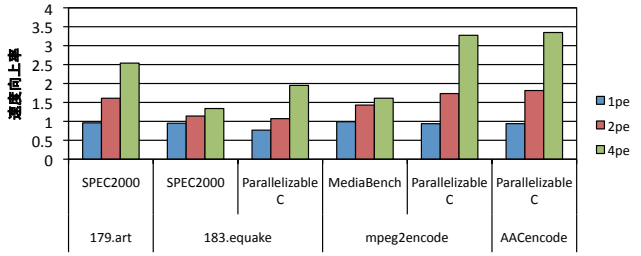


**図 5** ルネサステクノロジ／日立製作所／早稲田大学 RP2 における OSCAR コンパイラによる自動並列化結果
Fig. 5 Automatic parallelization results for OSCAR Compiler on Renesas/Hitachi/Waseda RP2

### 7.5 OSCAR コンパイラによる自動並列化結果

本稿で書き換えを行った art, equake, lbm, hmmer, mpeg2encode について，適用された自動並列化の概要を以下に示す．その際に，IBM XL コンパイラ，Intel コンパイラによる自動並列化結果についても併せて示す．

#### 7.5.1 SPEC2000 art

art では，オリジナルコードにおいて，OSCAR コンパイラの自動並列化により IBM p5 550Q において 8 コア使用時に 4.96 倍の性能向上が得られている．art では実行時間の 95% 程度は train_match と match という 2 つの関数によって占められており，ともに処理の大部分は 7 つの並列化可能なループの収束演算によって占められている．OSCAR コンパイラではこれらのループを並列化することにより性能向上が実現された．

IBM XL コンパイラ，Intel コンパイラでは自動並列化による性能向上は得られなかった．

#### 7.5.2 SPEC2000 equake

オリジナルコードでは IBM p5 550Q において 8 コア使用時に 1.69 倍と性能向上が限定的であったが，書き換えを行うことで 5.61 倍と大きな性能向上が得られるようになった．書き換えを行った smvp() のループでは，配列全体に対するリダクション計算が行なわれている．各プロセッサごとに対象の配列の一時配列を用意し，各プロセッサで配列の全要素について，部分和の計算結果を一時配列に格納する．その後，各プロセッサで部分和を格納した一時配列について，その総和を元の対象配列に格納することで，並列処理を実現している．このため，配列全体のコピーを行うオーバーヘッドにより逐次処理時間が 20%ほど増加しているが，プログラムの大部分が並列処理可能となり，プロセッサコア数に応じた性能向上が得られた．

XL コンパイラ，Intel コンパイラではオリジナルコード，Parallelizable C コードともに自動並列化による性能向上は得られなかった．

#### 7.5.3 SPEC2006 lbm

オリジナルコードでは自動並列化による性能向上を得ることができなかったが，Parallelizable C に書き換えることで，OSCAR コンパイラによる自動並列化により IBM p5 550Q で 8 コア使用時に 5.36 倍の性能向上が得られた．

lbm では LBM_performStreamCollide という関数が実行時間の 90% 程度を占めており，この関数は単一の並列化可能なループで構成されている．オリジナルコードでは，この関数の呼び出し部を含むループにおいてポインタの更新が行われており (図 1)，このポインタ更新により，ポインタの指し先情報が曖昧になっていたことが並列性抽出の阻害要因となっていた．これを，Parallelizable C への書き換えによって，ポインタの更新を除去することにより並列性抽出が可能となり，大きな性能向上が得られた．

IBM XL コンパイラでは自動並列化による性能向上は得られなかったが，Intel コンパイラではオリジナルコード，Parallelizable C コード共に自動並列化による性能向上が得られており，OSCAR コンパイラによる Parallelizable C コードの自動並列化時と同等の性能となっていた．

#### 7.5.4 SPEC2006 hmmer

オリジナルコードでは自動並列化による性能向上を得ることができなかったが，Parallelizable C に書き換えることで，OSCAR コンパイラによる自動並列化により IBM p5 550Q で 8 コア使用時に 6.06 倍の性能向上が得られた．

456.hmmer の全プログラム実行時間の 90%以上を占めるループは図 6 で示される構造となっている．逐次ループ内部にイタレーション間の依存がある部分とない部分が存在するため，並列処理を行うためにはループのリストラクチャリングが必要となる．OSCAR コンパイラによってループディストリビューションを行い，データローカライゼーション手法[22)]を適用した．このイメージを図 7 に示す．

この例では，まず図 6 のループに対してスカラエキスパンションが適用される．このとき，図 6 のランダムシーケンス生成部とシーケンス数値化部の関数戻り値はポインタであり，これらに対してスカラエキスパンションを適用するとポインタの配列となる．これらのポインタ配列の各要素間にエイリアスがないことが解析され，ループディストリビューションが適用される．その後，粗粒度タスク並列化におけるデータローカライゼーション手法が適用され性能向上を得ている．


© 2009 Information Processing Society of Japan

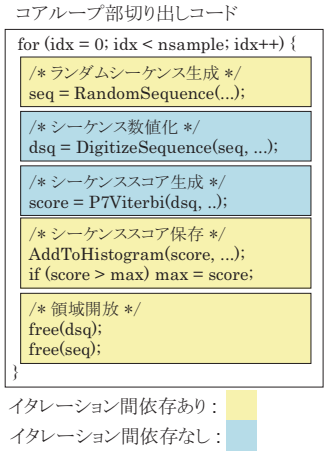


**図 6** hmmer における主要処理ループ構造
Fig. 6 The structure of the main loop in hmmer

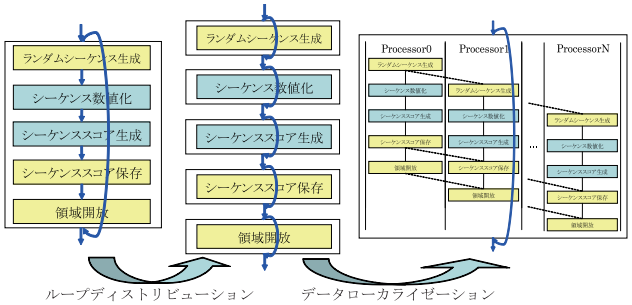


**図 7** hmmer における主要処理ループのデータローカライゼーション
Fig. 7 Data localization on the main loop in hmmer

オリジナルコードに対しては図 6 のループ内部で，ヒープ領域に確保したバッファをイタレーションを超えて再利用しているため，ポインタ解析を利用しても並列性の抽出ができず，性能向上は得られなかった．

IBM XL コンパイラ，Intel コンパイラによる自動並列化では，オリジナルコード，Parallelizable C コードともに性能向上は得られなかった．

#### 7.5.5 MediaBench mpeg2encode

オリジナルコードでは，IBM p5 550Q で OSCAR コンパイラによる自動並列化により 4 コア使用時に 1.53 倍の性能向上が得られた．この性能向上は動き推定処理中の関数である frame_estimation() が粗粒度タスク並列処理された結果である．コンパイラにより解析された並列度は 2.5 となっており，3 プロセッサが割り当てられ並列処理が行われた．そのため，4 コア使用時と 8 コア使用時では性能の差が見られなかった．

Parallelizable C コードでは OSCAR コンパイラの自動並列化により 8 コア使用時に 5.13 倍の性能向上が得られた．これは，プログラムの書き換えにより，OSCAR コンパイラによるマクロブロックレベルの並列性とデータローカリティの抽出[19] が可能となったためである．

IBM XL コンパイラ，Intel コンパイラによる自動並列化では，オリジナルコード，Parallelizable C コードともに性能向上は得られなかった．

## 8. ま と め

本稿ではコンパイラによる自動並列化を可能とするための C 言語の記述方法として Parallelizable C レベル 2 を提案した．Prallelizable C で記述した科学技術計算およびマルチメディア処理の逐次プログラムに対して自動並列化を適用し，マルチコアシステム上での処理性能の評価を行った．その結果，2 コア集積のマルチコアである IBM Power5+ を 4 基搭載した 8 コア構成のサーバである IBM p5 550Q において逐次実行時と比較して平均 5.54 倍，4 コア集積のマルチコアである Intel Core i7 920 プロセッサを搭載した PC において平均 2.43 倍，SH-4A コアを集積した情報家電用マルチコア RP2 の 4 コアを使用した SMP 実行モードにおいて平均 2.78 倍の性能向上が得られた．

本稿で提案した Parallelizable C と OSCAR コンパイラのような自動並列化コンパイラを組み合わせることで，マルチコア向けソフトウェア開発におけるソフトウェア生産性の向上を実現し，今後のマルチコア・メニーコア技術の研究開発の促進につながると考えられる．

**謝辞** 本研究の一部は NEDO“情報家電用ヘテロジニアス・マルチコア技術の研究開発”，および早稲田大学グローバル COE“アンビエント SoC” の支援により行われた．

## 参 考 文 献

1) Eigenmann, R., Hoeflinger, J. and Padua, D.: On the Automatic Parallelization of the Perfect Benchmarks, *IEEE Trans. on parallel and distributed systems*, Vol.9,


© 2009 Information Processing Society of Japan

No.1 (1998).
2) Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S., Bugnion, E. and Lam, M.S.: Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer* (1996).
3) 笠原博徳：最先端の自動並列化コンパイラ技術，情報処理, Vol.44 No. 4(通巻 458 号), pp.384-392 (2003).
4) Hind, M.: Pointer Analysis: Haven't We Solved This Problem Yet?, *In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp.54–61 (2001).
5) *MISRA-C:2004 - Guidelines for the use of the C language in critical systems* (2004).
6) *Clean C*. http://www.imec.be/CleanC/.
7) Hwu, W. et al.: Implicitly Parallel Programming Models for Thousand-Core Microprocessors, *DAC 2007* (2007).
8) M.Wolfe: High Performance Compilers for Parallel Computing, *Addison-Wesley Publishing Company* (1996).
9) Emami, M., Ghiya, R. and Hendren, L. J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers, *PLDI 1994*, pp.242–256 (1994).
10) Lattner, C., Lenharth, A. and Adve, V.: Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World, *PLDI'07* (2007).
11) Kahlon, V.: Bootstrapping: A Technique for Scalable Flow and Context-Sensitive Pointer Alias Analysis, *PLDI'08* (2008).
12) Hardekopf, B. and Lin, C.: Semi-Sparse Flow-Sensitive Pointer Analysis, *POPL'09* (2009).
13) Nystrom, E.M., Kim, H.-S. and mei W.Hwu, W.: Importance of heap specialization in pointer analysis, *PASTE'04* (2004).
14) Pearce, D.J., Kelly, P. H.J. and Hankin, C.: Efficient Field Sensitive Pointer Analysis for C, *PASTE '04* (2004).
15) Ryoo, S., Rodrigues, C.I. and Hwu, W.-M.W.: Iteration Disambiguation for Parallelism Identification in Time-Sliced Applications, *International Workshop on Languages and Compilers for Parallel Computing (LCPC)* (2007).
16) Wu, P., Feautrier, P., Padua, D.A. and Sura, Z.: Instance-wise Points-to Analysis for Loop-based Dependence Testing, *Proceedings of the 16th Annual International Conference on Supercomputing (ICS 2002)*, pp.262–273 (2002).
17) Ghiya, R. and Hendren, L.J.: Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C, *POPL'96* (1996).
18) 間瀬正啓，馬場大介，長山晴美，田野裕秋，益浦　健，宮本孝道，白子　準，中野啓史，木村啓二，笠原博徳：情報家電用マルチコア SMP 実行モードにおける制約付き C プログラムのマルチグレイン並列化，組込みシステムシンポジウム 2007 (2007).
19) 小高　剛，中野啓史，木村啓二，笠原博徳：チップマルチプロセッサ上での MPEG2 エンコードの並列処理，情報処理学会論文誌，Vol.46, No.9 (2005).
20) 本多弘樹，岩田雅彦，笠原博徳：Fortran プログラム粗粒度タスク間の並列性検出手法，電子情報通信学会論文誌，Vol.J73-D-1, No.12, pp.951–960 (1990).
21) 木村啓二，加藤孝幸，笠原博徳：近細粒度並列処理用シングルチップマルチプロセッサにおけるプロセッサコアの評価，情報処理学会論文誌，Vol.42, No.4 (2001).
22) 吉田明正，前田誠司，尾形　航，笠原博徳：Fortran マクロデータフロー処理におけるデータローカライゼーション手法，情報処理学会論文誌，Vol.35, No.9, pp.1848–1994 (1994).
23) 石坂一久，中野啓史，八木哲志，小幡元樹，笠原博徳：共有メモリマルチプロセッサ上でのキャッシュ最適化を考慮した粗粒度タスク並列処理，情報処理学会論文誌，Vol.43, No.4 (2002).
24) 白子　準，吉田宗弘，押山直人，和田康孝，中野啓史，鹿野裕明，木村啓二，笠原博徳：マルチコアプロセッサにおけるコンパイラ制御低消費電力化手法，情報処理学会論文誌，Vol.47, No.ACS15 (2006).
25) *OpenMP Application Program Interface Version 2.5* (2005).
26) *Optimaly Scheduled Advanced Multiprocessor Application Program Interface (OSCAR API) version 1.0*. http://www.kasahara.cs.waseda.ac.jp/.
27) Ito, M., Hattori, T., Yoshida, Y., Hayase, K., Hayashi, T., Nishii, O., Yasu, Y., Hasegawa, A., Takada, M., Ito, M., Mizuno, H., Uchiyama, K., Odaka, T., Shirako, J., Mase, M., Kimura, K. and Kasahar, H.: An 8640 MIPS SoC with Independent Power-off Control of 8 CPU and 8 RAMS by an Automatic Parallelizing Compiler, *2008 IEEE International Solid-State Circuits Conference* (2008).


© 2009 Information Processing Society of Japan