社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－ARC－178 (14)
2008／5／14

ポインタ解析を用いた制約付きCプログラムの自動並列化

間瀬 正啓† 馬場 大介†,†† 長山 晴美†,††† 村田 雄太† 木村 啓二†
笠原 博徳†

† 早稲田大学 理工学術院 基幹理工学部 情報理工学科
†† 松下電器産業株式会社
††† インテル株式会社
E-mail: {mase,kimura,kasahara}@kasahara.cs.waseda.ac.jp

あらまし 本稿では，自動並列化コンパイラにより並列性抽出が可能なC言語におけるポインタ利用方法の制約について述べる．実際にこの制約を満たすようにプログラムを作成し，flow-sensitive，context-sensitiveなポインタ解析を用いた自動並列化を適用したところ，8コアSMPサーバにおいて，逐次実行と比較してSPEC2000 artで3.80倍，SPEC2006 lbmで6.17倍，MediaBench mpeg2encで5.14倍の速度向上が得られた．
キーワード マルチコア，自動並列化コンパイラ，ポインタ解析，制約付きC言語

Automatic Parallelization of Restricted C Programs using Pointer Analysis

Masayoshi MASE†, Daisuke BABA†,††, Harumi NAGAYAMA†,†††, Yuta MURATA†, Keiji KIMURA†, and Hironori KASAHARA†

† Department of Computer Science and Engineering, Waseda University
†† Matsushita Electric Industrial Co., Ltd.
††† Intel K.K.
E-mail: {mase,kimura,kasahara}@kasahara.cs.waseda.ac.jp

Abstract This paper describes a restriction on pointer usage in C language for parallelism extraction by an automatic parallelizing compiler. By rewriting programs to satisfy the restriction, automatic parallelization using flow-sensitive, context-sensitive pointer analysis on an 8 cores SMP server achieved 3.80 times speedup for SPEC2000 art, 6.17 times speedup for SPEC2006 lbm and 5.14 times speedup for MediaBench mpeg2enc against the sequential execution, respectively.
Key words Multicore, Automatic Parallelizing Compiler, Pointer Analysis, Restricted C Language

1. はじめに

半導体集積度向上に伴うスケーラブルな性能向上，低消費電力，高い価格性能比を達成するためにマルチコアプロセッサが大きな注目を集めている．具体的には，携帯電話，カーナビゲーションシステム，デジタルTV，ゲーム等の情報家電機器を始め，PCからスーパーコンピュータに至る，多くの情報機器でマルチコアプロセッサ採用の動きが進んでいる．しかし，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の並列プログラミングは従来より難易度が高いことで知られており，科学技術計算用のマルチプロセッサシステムではアプリケーションの並列化に数か月以上の期間を要することも珍しくなかった．さらに，製品開発サイクルの短い情報家電分野においては質の高いアプリケーションプログラムを短期間のうちに多数開発していくことが要求され，従来のような長期間にわたる並列化プログラミングは現実的ではない．そのため，プログラマの負荷を大きく軽減できる自動並列化コンパイラが必須となる．

筆者等が開発しているOSCARマルチグレイン自動並列化コンパイラ[1]～[3]では従来のマルチプロセッササーバ用自動並列化コンパイラが対象としていたループレベル並列処理[4]に加え粗粒度タスク並列処理，近細粒度並列処理を組み合わせたマルチグレイン並列処理を実現しており，さらに，メモリ

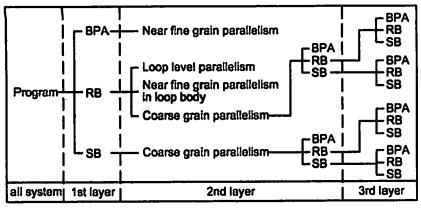

図 1 階層的マクロタスク定義

Fig. 1 Hierarchical macro task difinition

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 [5] さらに，チップ内の各リソースの周波数・電圧・電源制御による消費電力の削減 [6] を実現している．

従来の自動並列化技術は主に科学技術計算分野の FORTRAN プログラムを対象に開発されてきた．しかしながら，C 言語は特にポインタの明示的な利用により記述の自由度が高いため，解析や最適化が困難であり，C プログラムの全自動並列化は事実上不可能になっている．そこで OSCAR コンパイラでは現実的な解決策として，マルチグレイン並列処理を実マルチプロセッサシステム上で実現することを優先し，制約付き C 言語を採用し，入力ソースプログラムに一定の制約条件を設けることにより自動並列化を実現した [7].

本稿では，OSCAR コンパイラに新たにポインタ解析を実装し，ループ解析，データ依存解析，データアクセス範囲解析等の OSCAR 自動並列化コンパイラの既存の解析をポインタ解析情報に対応させることで，従来は使用を原則的に禁止していたポインタおよび構造体の一部を記述可能とした．この制約を緩和した制約付き C プログラムの自動並列化を実現したので，その報告をする．

本稿の構成を以下に示す．まず 2. では OSCAR コンパイラが実現するマルチグレイン並列処理の概要を述べ，3. で従来のポインタ解析を行わない場合の制約付き C 言語について述べる．次に 4. でポインタ解析について述べ，5. でポインタ解析を行う場合の制約付き C 言語について述べる．6. では，実際にアプリケーションプログラムをそれぞれの制約付き C 言語へ書き換えた際の書き換え項目および変更量について述べ，7. では書き換えた制約付き C プログラムに対し OSCAR コンパイラによる自動並列化を適用した際の SMP サーバ上での並列処理性能について述べる．8. で関連研究について述べ，最後に 9. で本稿のまとめを述べる．

2. マルチグレイン並列処理

本章では，OSCAR コンパイラで実現しているマルチグレイン並列処理とデータローカリティ最適化および，自動並列化のための制約付き C 言語の概要を述べる．マルチグレイン並列処理は粗粒度タスク並列性，ループ並列性，近細粒度並列性を組み合わせ，プログラム全域から並列性を抽出する技術である．本稿では粗粒度タスク並列性とループ並列性を用い，制約付き C プログラムのマルチグレイン並列処理を行う．

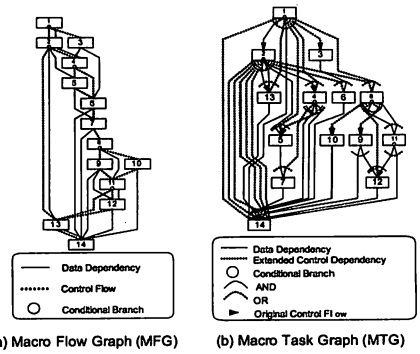

図 2 マクロフローグラフとマクロタスクグラフ

Fig. 2 Macro Flow Graph and Macro Task Graph

2.1 粗粒度タスク生成

粗粒度タスク並列処理では，ソースプログラムは基本ブロックまたはその融合ブロックで構成される疑似代入文ブロック BPA，ループや後方分岐により生じるナチュラルループで構成される繰り返しブロック RB，サブルーチンブロック SB の 3 種類の粗粒度タスク (マクロタスク MT) に分割される [3]．繰り返しブロックやサブルーチンブロックは図 1 に示すようにその内部をさらにマクロタスクに分割し階層的なマクロタスク構造を生成する．

2.2 粗粒度タスク並列性抽出 [1], [3]

マクロタスク生成後，各階層においてマクロタスク間の制御フローとデータ依存を解析し，図 2(a) に示すようなマクロフローグラフ MFG を生成する．

次に，階層的に生成されたマクロフローグラフに対し最早実行可能条件解析を適用し，図 2(b) に示すようなマクロタスクグラフ MTG を生成する．最早実行可能条件とは，制御依存とデータ依存を考慮した，マクロタスクが最も早く実行を開始してよい条件であり，マクロタスクグラフは粗粒度タスク並列性を表す．

2.3 データローカリティ最適化

まずコンパイラは複数ループ間のデータ依存を解析し，データ依存する分割後の小ループ間におけるデータ授受がキャッシュあるいはローカルメモリを介して行われるようにそれらのループを整合して分割するループ整合分割 [8] を行う．

図 3 にループ整合分割を適用したマクロタスクグラフを示す．整合分割後同一データにアクセスするマクロタスクが可能な限り同一プロセッサ上で連続的に実行されるように粗粒度タスクスケジューリングを行うことで，複数のループに渡りキャッシュあるいはローカルメモリ上のデータをそのまま利用することが可能となり，メインメモリアクセスを削減することができる．

2.4 制約付き C 言語

OSCAR コンパイラが実現するマルチグレイン並列処理およびデータローカリティ最適化により高い性能を得るためには，まずプログラムが並列性のあるアルゴリズムで記述されており，その上でコンパイラによりプログラム全域にわたる高い精度のデータ依存解析，データアクセス範囲解析を行うことが必須と

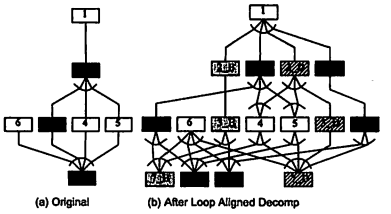

図 3 データローカライゼーションのためのループ整合分割
Fig. 3 Loop Aligned Decomposition (LAD) for data localization

なる．そのため，現在の OSCAR コンパイラでは，入力 C 言語に以下のような制約を加えることで自動並列化を実現している．

- 一括コンパイル
- 関数の再帰呼び出しは行わない
- ポインタ・構造体の利用の制限

データ依存解析，データアクセス範囲解析はポインタの指し先が不明な場合に解析精度が著しく低下するため，ポインタの指し先を解析時に決定できることが望ましい．しかしながら，コンパイラによる完璧なポインタ解析手法は未だ実現されておらず，現実的な解決策が必要となる．以下，本稿では，ポインタ・構造体の利用の制限に焦点を当て，自動並列化のための制約について述べる．

3. ポインタ解析を行わない場合の制約付き C 言語

C プログラムのポインタ解析は非常に困難とされており，C99 標準 [9] の restrict 修飾子のように，プログラマがヒント情報を付加することでコンパイラの解析精度を向上させることも選択肢の一つである．実際，Intel C/C++コンパイラの-fno-alias オプション [10]，IBM XL C コンパイラの-qalias=noallptr オプション [11]，Sun Studio C コンパイラの -xrestrict オプション [12] のように，多くの商用コンパイラはポインタエイリアスに関するヒント情報をユーザが指定可能な枠組みを用意している．

3.1 制約事項

OSCAR コンパイラにおいてポインタ解析を行わずに自動並列化を適用する際の制約事項を以下に述べる．ヒント情報を前提として FORTRAN77 と同等の記述を行っている．

- ポインタおよび構造体は原則的に使用しない

ポインタおよび構造体は原則的に使用しない．メモリ動的確保についても単純な多次元配列を用いて代替する．ただし，関数呼び出しにおける引数の参照渡しを考慮して，以下に示すような関数のポインタ引数は例外とする．

- 関数のポインタ引数に対する制約

C 言語では関数呼び出しにおける引数の参照渡しをポインタの値渡しにより実現するため，実引数と仮引数の静的なエイリアスができず，配列としての情報が失われる．そのため関数のポインタ引数に対して以下の制約を設けることで，FORTRAN77 の参照渡しによる引数と同等に扱う．

（1） ポインタ値の再代入は行わない

（2） ポインタ引数を用いた参照先がエイリアスしない

（3） ポインタ引数のアクセス範囲が関数の呼び元における配列宣言境界を越えない

（4） ポインタのキャストを行わない

(1) および (2) の制約によって全ての引数が restrict 修飾子が指定されているのと等価となる．これにより，コンパイラのインタープロシージャ解析において，実引数と仮引数が静的にエイリアスするものとして扱うことが可能となる．(3) の制約では C 言語のポインタ記述では表現できないポインタによるアクセスの境界を保証でき，データアクセス範囲解析の精度を高めることができる．

4. ポインタ解析

ポインタ解析とは，プログラム中に現れるポインタがメモリ上のどの領域を指すかをコンパイル時に静的に解析するものである．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情報の持たせ方によって分類される [13].

flow-sensitivity はコントロールフローに沿って解析を行うかどうかの分類である．flow-sensitive なポインタ解析では，プログラムの流れ順に解析を行い，それぞれのステートメントごとに有効な情報を作成する．これによりポインタの再代入がある場合は以降のステートメントでは新しい指し先情報のみを残し，コントロールフローに沿って情報を伝播させる．ループなど繰り返し情報の流入がある地点では，その情報に変化がなくなるまで解析を繰り返す．

context-sensitivity は同じ関数への呼び出しを行うコールサイトが複数ある場合，それらを別個に扱うかどうかの分類である．context-sensitive なポインタ解析では，関数はコールサイトごとに区別されて解析されるため，異なる呼び出し元の情報が伝播されることがなく，正確な解析ができる．

heap-sensitivity はヒープ領域の内部を区別するかどうかの分類である．heap-sensitive [14] なポインタ解析では，メモリが動的に確保された場合に，確保された箇所ごと，また内部でメモリが動的に確保される関数の呼び出し箇所ごとにヒープ領域を区別する．これにより配列が動的に確保された場合はお互いを区別できるが，リストのような再帰的なデータ構造の場合にはそれぞれの要素の区別は行えない．

field-sensitivity は構造体や配列の各要素を区別するかどうかの分類である．field-sensitive [15] なポインタ解析では名前のついた変数を一つの領域とせずに構造体や配列の各要素を区別する．構造体のメンバに別の名前をつけて区別する手法や，メモリ配置を考慮し，共用体やポインタ演算にも対応する手法がある．

5. ポインタ解析を行う場合の制約付き C 言語

本稿では，OSCAR コンパイラとの親和性と解析精度に重点を置き，Emami ら [16] が提案した flow-sensitive, context-sensitive ポインタ解析アルゴリズムをベースに，さらに heap-sensitive, 構造体のメンバ間および配列の先頭要素とそれ以外の要素に対して field-sensitive なポインタ解析を OSCAR コン

パイラに実装した．この解析では呼び出し元ごとに関数をクローニングして解析を行い，スコープ外の変数を不可視変数として解析を行っている．

このポインタ解析を用いて，3. で述べた従来の制約付き C 言語と同等の並列性抽出が可能となる制約条件を以下に示す．

5.1 制約事項

以下の制約が守られていない場合は OSCAR コンパイラではコンパイルができない，実行結果が不正となる，解析精度が著しく低下するといった不具合が生じる．

- 関数ポインタを使用しない

コールグラフを静的に決定できなくなるため，関数ポインタは使用しない．

- 名前の境界を超えるアクセスを行わない

構造体のメンバへのアクセスを他のメンバからのオフセット計算により行なうといった，名前の境界を超えるアクセスを禁止する．

5.2 非推奨事項

以下に示すような表記が使用される場合は，解析精度が低下し，並列化や最適化が抑制される可能性がある．

- 再帰的なデータ構造

リストのような再帰的なデータ構造は不可視変数の融合 [16] によりすべて 1 つの領域として解析される．

- ポインタの配列

ポインタの配列はポインタ解析器において配列要素ごとの指し先の解析を行わない．そのため配列が全ての指し先を指す可能性があると解析される．ポインタ型のヒープも同様である．

- 配列の先頭以外を指すポインタ

配列の先頭以外を指すポインタが指す配列はデータアクセス範囲解析を行わない．このようなポインタを使用せずに，配列の途中のアドレスを関数に渡すことにより解析可能となる．

- 複数の引数による同一領域の参照渡し

複数の引数による同一領域の参照渡しを行うと不可視変数の融合が起こり，関数内部でのデータアクセス範囲解析が行えない．

- 配列の構造体のメンバに配列を定義

配列の構造体のメンバに配列を定義するとメンバのデータアクセス範囲解析が行えない．このようなデータ構造を定義する場合，構造体の一要素を関数に渡すことにより解析可能となる．

- 条件分岐，ループ内でのポインタ変数の代入

条件分岐，ループ内でのポインタ変数の代入を行うと決定的なポインタ指し先情報が生成できず，生死解析の精度が低下する．

- メモリの動的確保

メモリの動的確保を用いると，配列の上限値が不明なためデータアクセス範囲解析の精度が低下する．

6. 制約付き C 言語への書き換え

本章では SPEC2000 より art, SPEC2006 より lbm, MediaBench より mpeg2encode を例に，ポインタ解析を行う場合，行わない場合それぞれの制約付き C プログラムへの書き換えについて述べる．各アプリケーションについて，リファレンスコードを単一ソースコードにまとめた original，ポインタ解析を行わない場合の制約付き C 言語で記述した no-pointer，ポインタ解析を行う場合の制約付き C 言語で記述した pointer の 3 種類のソースコードを作成した．その際の書き換え項目およびコード変更量について述べる．

6.1 art

art は 1 つのソースファイルによって構成されており，このソースコードをそのまま original コードとする．

6.2 lbm

lbm は 2 つのソースファイルによって構成され，OpenMP により並列化されている．ソースコードを 1 つのファイルにまとめ，コンパイラの評価を行うために OpenMP のディレクティブを削除したものを original コードとする．

6.3 mpeg2encode

mpeg2encode は 18 個のソースファイルによって構成される．このソースコードを 1 つのファイルにまとめ，評価のための入力画像の読み込み部を追加したものを original コードとする．

original コードでは，本来 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，制約付き C 言語への書き換え時に並列性抽出のためのプログラム構造の変更 [17] も併せて行った．

6.4 書き換え項目とコード変更量

コードの変更量の測定はコメント，インデント，空白行を削除した上で，文字単位の比較を行い，書き換えにおいて削除した部分と追加した部分を抽出した．表 1 にそれぞれのアプリケーションごとに original コードからポインタ解析を行わない場合の制約付き C 言語 (no-pointer)，ポインタ解析を行う場合の制約付き C 言語 (pointer) への書き換え項目および変更前のコードのサイズを基準とした削除量，追加量の割合を示す．ポインタ解析の利用による制約緩和により，いずれのアプリケーションにおいても書き換え項目が減少し，削除量と追加量を合計した変更量が art では 13%，lbm では 8%，mpeg2encode では 9%削減された．mpeg2encode の変更量が全体的に art，lbm と比べて大きくなっているのは，マクロブロックレベルの処理への書き換えが要因となっている．

7. 並列処理性能

本章では，6. で書き換えを行った art, lbm, mpeg2encode について，OSCAR コンパイラによる自動並列化を適用した際の SMP サーバ IBM p5 550Q 上での並列処理性能について述べる．

7.1 評価条件

IBM p5 550Q は 2 コアを集積した Power5+ プロセッサを 4 基搭載した 8 コア SMP サーバであり，1 チップ (2 コア) あたり 1.9MB の L2 キャッシュ，36MB の L3 キャッシュを搭載している．

OSCAR コンパイラの自動並列化コードは並列化された OpenMP C プログラムとして出力し，ネイティブコンパイラである IBM XLC コンパイラ 8.0 でコンパイルし，実行した．XLC コンパイラのオプションは XLC コンパイラでの自動並列

表 1　制約付き C 言語への書き換え

Table 1　Code rewriting in restricted C language

アプリケーション	originalコードからno-pointerコードへの書き換え			originalコードからpointerコードへの書き換え		
	書き換え項目	削除量	追加量	書き換え項目	削除量	追加量
art	ポインタの配列を多次元配列化 メモリの動的確保の除去 構造体の展開	9.93%	4.15%	ポインタの配列を多次元配列化	1.41%	0.19%
lbm	ループ中のポインタ変数の代入を除去 配列の途中を指すポインタの除去 メモリの動的確保の除去 ポインタ間接参照を直接参照に変更 構造体の展開	5.47%	3.68%	ループ中のポインタ変数の代入の除去 配列の途中を指すポインタの除去	0.97%	0.60%
mpeg2enc	マクロブロックレベルで処理できるように変更 ポインタの配列を多次元配列に変更 ポインタ演算をオフセット演算に変更 ポインタのswapをメモリコピーに変更 構造体の展開	26.40%	23.90%	マクロブロックレベルで処理できるように変更 ポインタの配列を多次元配列に変更	24.10%	17.10%

化時には "-qsmp=auto" を用い，OSCAR コンパイラで自動並列化時には "-qsmp=noauto" を用いた．時間計測結果は I/O 処理をメモリへの読み書きに変更し，実際の I/O 処理を除いた時間を計測した．

7.2　art の評価

図 4 に評価結果を示す．図中，横軸が各コードと使用したコア数を示し，縦軸が XLC コンパイラを使用した各コードの 1 コアの速度に対する速度向上率を示す．左側のバーが XLC コンパイラの自動並列化による速度向上率，右側のバーがポインタ解析を用いた OSCAR コンパイラの自動並列化による速度向上率である．また，no-pointer コードのみ真ん中のバーに，ポインタ解析を行わずに 3. で述べた制約付き C 言語であるというヒント情報を利用した OSCAR コンパイラによる自動並列化の速度向上率を示す．

評価の結果，OSCAR コンパイラでは書き換えを行っていない original コードにおいても pointer コードと同じ 3.80 倍の速度向上が得られた．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高速化を実現している．original コードではポインタの配列を使用しているため並列実行されないループが存在するが，処理コストが非常に小さい上に時間測定の範囲外にあり，速度向上率が一致する結果となった．

original コード，pointer コードに比べ，no-pointer コードでは OSCAR コンパイラによる自動並列化による速度向上率が最大 4.80 倍と高くなっているが，並列化された箇所は同一となっていた．そこで，no-pointer コードは pointer コードと実行時間を比較すると 1 コアによる逐次処理が no-pointer コードの方が 2.2 倍高速となっており，逐次性能の変化が原因と考えられる．この逐次性能の高速化は，original, pointer コードでは処理の大部分が動的確保された構造体の配列をアクセスしているが，no-pointer コードではこれらを単純な配列に置き換えたことで，アドレス演算が減少する上に，メモリに対して連続アクセスするようになるためと考えられる．

また，no-pointer コードにおいて，ポインタ解析を利用した自動並列化も，従来の制約付き C 言語であるというヒント情報を利用した自動並列化と同様の速度向上が得られていた．

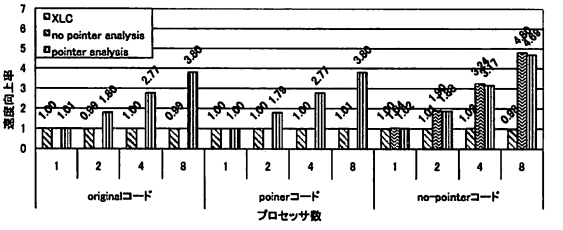

図 4　art の速度向上率

Fig. 4　speedup ratio for art

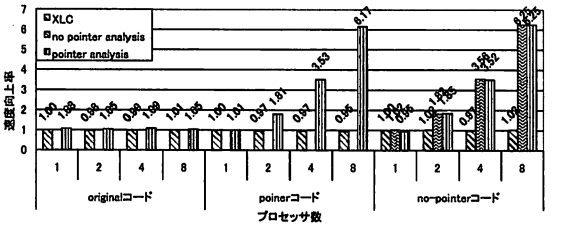

図 5　lbm の速度向上率

Fig. 5　speedup ratio for lbm

7.3　lbm の評価

図 5 に評価結果を示す．original コードでは XLC，OSCAR コンパイラともに並列化による速度向上を得ることができなかったが，pointer コードに書き換え，OSCAR コンパイラによるポインタ解析を用いた自動並列化を適用することにより 8 コア使用時に 6.17 倍の速度向上が得られた．no-pointer コードでも OSCAR コンパイラによる自動並列化により同様の速度向上が得られている．lbm では実行時間の 90% 程度は LBM_performStreamCollide という関数によって占められており，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．OSCAR コンパイラではこのループを並列化することにより，速度向上が得られた．

7.4　mpeg2encode の評価

図 6 に評価結果を示す．original コードでは，OSCAR コンパイラによる自動並列化により 8 コア使用時に 1.44 倍の速度向上が得られたのみである．この速度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．一方，pointer コードでは OSCAR コンパイラの

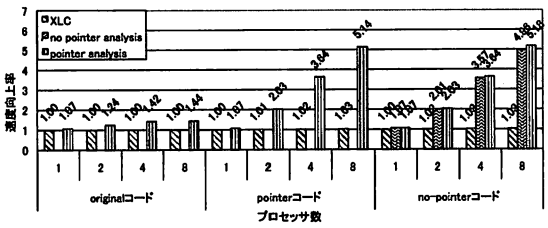

図 6 mpeg2encode の速度向上率
Fig. 6 speedup ratio for mpeg2encode

自動並列化により 8 コア使用時に 5.14 倍の速度向上が得られた．また，no-pointer コードでも OSCAR コンパイラの自動並列化により同様の速度向上が得られている．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が可能となったためである．

8. 関連研究

Prabhu ら [18] はスレッドレベル投機（TLS）を用いて，将来的なコンパイラによる自動並列化を念頭に手動並列化を行い，その知見から，TLS ハードウェアサポートを前提とした自動並列化が適用しやすい逐次プログラムの記述方法についても触れている．本稿で述べた制約は TLS のような強力なハードウェアサポートを前提としないものであり，実際に開発した自動並列化コンパイラを用いて，商用 SMP サーバ上で自動並列化による高い並列処理性能を実現している．

Ryoo ら [19] はポインタ解析を含むコンパイラによる解析により，MPEG4 エンコーダのリファレンスプログラムから粗粒度並列性が抽出可能かどうかを検証しており，インタープロシージャ配列解析，heap-sensitive, field-sensitive ポインタ解析，値の制約条件解析を統合することで並列性の抽出が可能と結論付けている．ただし，この論文で述べている必要な解析器の統合による，コンパイラによる全自動並列性抽出はまだ実現していないようである．

9. まとめ

本稿では，ポインタ解析を用いたコンパイラによる自働並列化のための制約付き C 言語について述べた．従来の OSCAR コンパイラではポインタ解析を行わずに，FORTRAN77 と同等のプログラム記述を行うことで自動並列化を実現していたのに対し，新たに OSCAR コンパイラにポインタ解析器を実装し，各解析器を拡張することにより，解析精度を保ったまま一部のポインタ・構造体の利用が可能となるように制約を緩和した．緩和された制約付き C 言語で作成したプログラムに対して OSCAR コンパイラによる自動並列化を適用することで，8 コア SMP サーバである IBM p5 550Q においてネイティブコンパイラの逐次実行に対し art で 3.80 倍，lbm で 6.17 倍，mpeg2encode で 5.14 倍の速度向上が得られた．ポインタ解析を利用することで，ポインタ解析を行わない場合と比較して，同様の並列性を抽出した上でコード変更量が art では 13%，9%，lbm では 8%，mpeg2encode では 9% 削減された．また，art ではオリジナルコードのままでポインタ解析を行う場合の制約付き C 言語への書き換え時と同等の性能が得られており，ユーザによるプログラムの理解により，さらに書き換え量を減らすことが可能と考えられる．

謝辞 本研究の一部は NEDO“リアルタイム情報家電用マルチコア技術の研究開発”，及び 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” の支援により行われた．

文献

[1] 本多, 岩田, 笠原：“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, **J73-D-1**, 12, pp. 951–960 (1990).
[2] H. Kasahara, et al.: “A multi-grain parallizing compilation scheme on oscar”, Proc. 4th Workshop on Language and Compilers for Parallel Computing (1991).
[3] 笠原：“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] M.Wolfe: “High performance compilers for parallel computing”, Addison-Wesley Publishing Company (1996).
[5] 石坂, 中野, 八木, 小幡, 笠原：“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, **43**, 4 (2002).
[6] 白子, 吉田, 押山, 和田, 中野, 鹿野, 木村, 笠原：“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, **47**, ACS15 (2006).
[7] 間瀬, 馬場, 長山, 田野, 益浦, 宮本, 白子, 中野, 木村, 笠原：“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).
[8] 吉田, 前田, 尾形, 笠原：“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, **35**, 9, pp. 1848–1994 (1994).
[9] “ISO/IEC 9899:1999 - Programming Language C” (1999).
[10] “Intel(R) C/C++ Compiler 10.1 for Linux - Documentation” (2007).
[11] “IBM XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference” (2005).
[12] “Sun Studio11 C User's Guide” (2005).
[13] M. Hind: “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).
[14] E. M. Nystrom, H.-S. Kim and W. mei W. Hwu: “Importance of heap specialization in pointer analysis”, PASTE (2004).
[15] D. J. Pearce, P. H. J. Kelly and C. Hankin: “Efficient field sensitive pointer analysis for c”, PASTE '04 (2004).
[16] M. Emami, R. Ghiya and L. J. Hendren: “Context-sensitive interprocedural points-to analysis in the presence of function pointers”, Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI 1994), pp. 242–256 (1994).
[17] 小高, 中野, 木村, 笠原：“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, **46**, 9 (2005).
[18] M. K. Prabhu and K. Olukotun: “Exposing speculative thread parallelism in spec2000”, PPoPP '05 (2005).
[19] S. Ryoo, S.-Z. Ueng, C. I. Rodrigues, R. E. Kidd, M. I. Frank and W.-M. W. Hwu: “Automatic discovery of coarse-grained parallelism in media applications”, Transactions on High-Performance Embedded Architectures and Compilers (2007).