Vol.2016-HPC-153 No.33
2016/3/3

OSCAR自動並列化コンパイラにおける解析時データ構造変換による並列性抽出手法

影浦 直人[1,a)]　和気 珠実[1,b)]　韓 吉新[1]　木村 啓二[1]　笠原 博徳[1]

概要：C 言語で記述されたプログラムにおいて, 構造体や配列は様々なデータ構造を実現するために広く利用されているデータ型である．しかしながら，コンパイラによる自動並列化を行う際に，複雑にネストされた構造体を用いたプログラムに対して並列性抽出の解析を行うことは困難である．また，C 言語で記述されたプログラムでは，配列添え字をループ制御変数の線形結合によって表すことにより，1 次元配列を多次元配列のように扱うものもよくある．このようなプログラムに対して自動並列化を適用する場合, 多次元配列と同様の依存ベクトルを抽出することは重要である．本稿では, これらの構造体や配列添え字式を持つプログラムから並列性を抽出するため，構造体のメンバー参照や 1 次元配列を，解析用の仮想的な多次元配列に変換し解析する手法を提案する．本手法を OSCAR 自動並列化コンパイラに実装し，画像ノイズ除去処理の一種であるメディアンフィルタ処理や MPEG2 デコード処理を用いて評価した．評価の結果，メディアンフィルタ処理においては 4 コアで 1 コアと比べて 2.85 倍の速度向上が得られ, MPEG2 デコードにおいては 4 コアで 1 コアと比べて 2.11 倍の速度向上が得られた．

1. はじめに

マルチコアプロセッサはスマートフォンやパーソナルコンピュータからスーパーコンピュータまで既に幅広いシステムに利用されており, これらのマルチコアプロセッサを有効利用するためのプログラム並列化に対する需要がますます高まっている．

従来より筆者等は OSCAR 自動並列化コンパイラの開発 [8] により，様々なプログラムの並列化を行ってきた．特に C 言語により記述されたプログラムに対しては，コンパイラによる並列化を行いやすくするプログラミング規約である Parallelizable C を提案 [9] し，これにより記述されたプログラムを並列化してきた．しかしながら，Parallelizable C では構造体の利用規約として，「構造体の配列メンバーに対する配列アクセスを行わない」といった制約を課している．また，配列の扱いに関しても，C 言語で記述されたプログラムでよく用いられる記法である．配列添え字をループ制御変数の線形結合で表した場合は，ループ依存ベクトルが抽出できず，Doacross 並列化が適用できないといった問題点があった．

構造体は，データをまとめる際などに広く利用されるデータ型であり，複数の様々なメンバーの保持や，入れ子の構造など様々な形のデータ構造を用意することができる．構造体を用いたプログラムをコンパイラにより自動で並列化する際には，構造体のメンバーに着目した解析を行う必要がある．構造体の解析手法として，Field-sensitive なポインタ解析 [1][2] が提案されている．Field-sensitive なポインタ解析とは，構造体や配列などを一つの領域であるとせずに，構造体や配列の領域内の各要素を区別する解析である．Field-sensitive なポインタ解析における構造体解析では，構造体のメンバーにそれぞれ別の名前を付けて区別する手法や，メモリ配置を考慮してポインタ演算にも対応する手法がある．しかし，構造体型の配列や構造体のメンバーの配列など，構造体に配列が混在すると，構造体のメンバー間の区別に加えて配列の要素間の区別も必要となり，解析がより複雑なものとなる．

配列もまた広く利用されているデータ型であり，配列添え字解析による並列性抽出手法も数多く提案されている．これらの解析手法の多くは，多重ループ中の配列添え字は各次元で高々 1 つのループ制御変数のみが使われていることを前提としている．しかしながら，C 言語では，動的メモリ確保により実行時に次元サイズの決まる多次元配列のメモリ領域を確保する方法が事実上存在せず，画像処理アプリケーションなどの多次元のデータを扱うプログラムの多くでは，1 次元のメモリ領域を確保しその添え字をルー

[1] 早稲田大学
Waseda University.
[a)] kage@kasahara.cs.waseda.ac.jp
[b)] waketama@kasahara.cs.waseda.ac.jp

プ制御変数の線形結合として表現することで擬似的に多次元配列を実現している．このような複雑な配列添え字から，並列処理において有用な情報である依存ベクトルを抽出することは非常に困難である．

複雑な配列添え字の別の表現方法としては LMAD (Linear Memory Access Descriptor) [3] 等があるが, LMAD はループで始めにアクセスされる配列の要素の位置, ループ一回転分の誘導変数の変位, ループ開始時から終了時までにアクセスされる配列要素の範囲, の 3 つのみで表されるため, これだけの情報から全ての並列性, 特に Doacross 並列性を引き出すことは困難である．また，文献 [4] のような線形結合式から各次元の情報を抽出する手法もあるが，アルゴリズムの制限上 Doacross 並列化に対応していない．

そこで本稿では，構造体の各メンバーが同一のメモリ領域を指すことはないという前提のもと，これらの構造体や配列添字式を持つプログラムから並列性を抽出するため，構造体のメンバー参照や 1 次元配列を解析用の仮想的な多次元配列に変換し解析する手法を提案する

以下本稿では，第 2 節にて解析時のデータ構造変換による並列性抽出手法について説明する，第 3 節にてこれを実装した当研究室で開発している OSCAR 自動並列化コンパイラについてその概要を説明する, 第 4 節にて拡張後の OSCAR 自動並列化コンパイラを用いたアプリケーションの評価結果について述べる．

2. 解析時のデータ構造変換による並列性抽出手法について

本節では提案手法である自動並列化コンパイラによる並列性解析を行うための構造体及び 1 次元配列の解析時データ構造変換について述べる．両者に共通する基本的なアイデアは，構造体や 1 次元配列を従来からの強力な解析手法が適用可能な多次元配列にマッピングすることである．

2.1 データ構造変換による構造体解析

本手法では，構造体メンバーが同一のメモリ領域を指すことがないという前提のもと，構造体の情報を仮想的な多次元配列にマッピングする．コンパイラの内部表現において，構造体を構造体の情報をマッピングした仮想的な配列に置き換えることで，構造体を配列と同じように解析することを可能にする．データ構造を変換する際には添え字情報として下記の 4 つの情報を持たせる．

・メンバー番号

構造体のメンバーに番号をつけ，メンバー番号を次元情報として持たせることで各メンバーを識別する．

```
struct st{
    int a;
    char b;
    double c;
}

struct st sv;
```

図 1 スカラのみ持つ構造体例

図 1 のような構造体の場合，構造体変数 sv は仮想的な配列変数 sv_ano にマッピングされる．各メンバーに番号が振られ，sv.a は sv_ano[1]，sv.b は sv_ano[2]，sv.c は sv_ano[3] と，sv_ano の各配列要素として解析時に扱われる．

・配列の次元情報

構造体型の配列とメンバーの配列の場合は元々ついている次元情報を仮想的な配列に追加する．

```
struct st{
    int a[N];
}

struct st sv;
struct st sa[N];
```

図 2 配列のメンバーを持つ構造体例

図 2 のような構造体の場合，構造体の変数 sv は配列変数 sv_ano にマッピングされる．sv.a[i] には，a のメンバー番号と a の配列の次元情報の次元情報が付き，sv_ano[1][i] となるようコンパイラ内部で扱われる．同様に構造体の配列変数 sa は配列変数 sa_ano にマッピングされる．sa[i].a[j] には sa の配列の次元情報とメンバー番号と a の配列の次元情報の次元情報が付き，sa_ano[i][1][j] とコンパイラ内部で扱われる．また，構造体型の配列とメンバーの配列が多次元の場合は次元数の分だけ配列の次元情報を追加する．

・ポインタの次元情報

構造体型の変数・配列がポインタの場合とメンバーの変数・配列がポインタの場合は，‘[]’ を用いた添え字アクセスの可能性があるため，このような演算を表現するための次元情報を追加する．添え字アクセスでないポインタには仮の次元情報 [0] をつける．

```
struct st{
    int *p;
}

struct st sv;
struct st *sp;
```

図 3 ポインタ変数のメンバーを持つ構造体例

図 3 のような構造体の場合，構造体の変数 sv は仮想的な配列変数 sv_ano のにマッピングされる．sv.p には，p のメンバー番号と p のポインタの次元情報が付き，sv_ano[1][0] となるようにコンパイラ内部で扱われる．構造体のポインタ変数 sp は仮想的な配列変数 sp_ano にマッピングされる．sp->p は sp のポインタの次元と p のメンバー番号と p のポインタ変数を '[]' によりアクセスする際の仮想的な次元情報が付き，sp_ano[0][1][0] となるようにコンパイラ内部で扱われる．また，ポインタによる添え字アクセスの場合を考えると，sv.p[i] には p のメンバー番号と p のポインタを用いたアクセスによる次元情報が付き，sv_ano[1][i] となり，同様に sp[i].p[j] には sp のポインタ次元と p のメンバー番号と p のポインタの次元の次元情報が付き，sp_ano[i][1][j] となる．このとき，sv.p[0] と*sv.p のように，一つのポインタでの '*' による間接参照と '[]' による添え字アクセスが行われると，どちらも sv_ano[1][0] となり，区別がつかなくなってしまうため，一つのポインタで間接参照と添え字アクセスを混在させていないことを前提条件としている．

・次元数をそろえるための添え字情報

仮想的にマッピングする構造体情報の数がメンバーによって異なる場合も，各メンバーを構成する構造体型の変数や配列から変換される配列の次元数をそろえる必要がある．そこで，メンバーによって仮想的な配列の次元数が異なる場合，最も次元数の多いメンバーに次元数をそろえる必要がある．次元数の少ないメンバーには仮の次元情報 [0] を付け，次元数を調整する．次元数は，各構造体メンバーの中のの最大の次元数とする．

```
struct st{
    int v;
    int a[N][N];
}

struct st sv;
```

図 4 異なる次元数のメンバーを持つ構造体例

図 4 のような構造体の場合構造体変数 sv は仮想的な配列変数 sv_ano にマッピングされる．sv_ano の次元数は元の構造体 st のメンバー a[N][N] より 3 次元となる．そのため，sv.v は v のメンバー番号の次元情報 sv_ano[1] に対して次元数をそろえる仮の次元情報が 2 つ付き，sv_ano[1][0][0] となる．

構造体をこれらの情報をマッピングした仮想的な配列に変換することで，配列と同等に解析でき並列性を抽出することができる．

この構造体解析手法を用いるためには，前提としている条件が 2 つある．1 つ目は，一つの構造体型のポインタまたは構造体メンバーのポインタで '*' による間接参照と '[]' による添え字アクセスを混在させていないという条件である．混在させると，どちらであるか区別することができないため，解析精度を落とす原因となる．2 つ目は，メンバーの指し先どうしがエイリアスしていないという条件である．エイリアスしていると，仮想的な配列の異なる要素が同一のメモリ領域を指すことになるため，解析を誤ってしまう．

また，リスト構造を含む構造体と添え字式に 2 種類以上の変数が使われている配列を含む構造体については，解析が行えないため対象外としている．対象外の構造体についてはデータ構造の変換は行わず，他の対象の構造体についてはデータ構造の変換を行う．これにより，対象外の構造体が並列化対象箇所で使われていなければ，もしアプリケーション中に対象外の構造体が含まれていても，並列化対象箇所は並列化することが可能となる．

2.2 多次元配列的にアクセスされる一次元配列の解析

次に解析時のデータ構造変換による並列性抽出手法について述べる．第 1 節で述べたように, 配列添え字にループ制御変数が複数含まれているなど複雑な形式の場合, そのままの状態で並列性を抽出することは難しい．そこで本節では, 多次元配列的にアクセスする一次元配列の場合にも並列性を抽出するため, 仮想的に多次元配列へと変換する方法について述べる．これにより依存ベクトルが計算できるようになり, 並列性を抽出することが可能となる．

一次元配列の多次元配列への変換アルゴリズムについては, Grosser 等による手法 [4] をベースとする．Grosser 等による手法では一次元配列から多次元配列を構成する各次元の添字情報を抽出可能だが, その際, 正答性条件を満たさないため変換できない場合がある．その一つの例が DOACROSS 処理が必要となるループキャリード依存がある場合である．そこで本稿ではこのアルゴリズムを拡張し, DOACROSS ループのような依存がある場合でも正しく多次元配列へと変換する手法を述べる．ここでは変換後の配列は 3 次元配列の場合の例を示す．変換対象の配列は**図 5** のループ内の配列とし, o0, o1, o2 は各ループ変数の定数項, s0, s1, s2 は各ループの上限値を表す．

```
for (i = 1; i < s0; i++) {
  for (j = 1; j < s1; j++) {
    for (k = 1; k < s2; k++) {
      a[n2 * n1 * (i + o0) + n2 * (j + o1) + (k + o2)] = 1;}}}
```

図 5 多次元的にアクセスしている一次元配列例

本アルゴリズムは以下のステップにより構成される．

- ステップ 1. 配列添え字情報からループ制御変数を含む

項を取り出し, 係数のソートを行う.
配列添え字のうちループ変数を含む項は n2 * n1 * i + n2 * j + 1 * k となり, 係数についてソートを行うと, n2 * n1, n2, 1 となる.

- ステップ 2. ソート結果について, 大きい項が小さい項で割り切れる場合, この一次元配列は多次元配列に変換可能であることがわかり, a[ ][n1][n2] 形の多次元配列と推定する.
- ステップ 3. 各ループ変数に関する配列添え字情報を三次元目の推定サイズで割る.
各ループ変数に関する配列添え字情報の変数項と定数項をそれぞれ三次元目の推定サイズで割る. i に関する配列添え字情報を三次元目の推定サイズで割ると以下のようになる.

$$(n2 * n1 * i + n2 * n1 * o0 + n2 * o1 + o2)/n2 = n1 * i + n1 * o0 + o1...o2$$

同様に j に関する配列添え字情報を三次元目の推定サイズで割ると以下のようになる.

$$(n2 * j)/n2 = 1 * j...0$$

k に関する配列添え字情報を三次元目の推定サイズで割ると,

$$(1 * k)/n2 = 0...1 * k$$

- ステップ 4. 各 i, j, k に関する割り算結果のうち, 余りを三次元目の式と推定する. 今回の例では, a[ ][ ][k + o2] と推定される.
- ステップ 5. 正しい推定であるかどうか確認する.
推定された式について, 誘導変数の動作範囲に対して以下の条件を満たしているかどうか確認する.

$$0 <= k + o2 < n2$$

満たしている場合はこの推定は正しいとし、ステップ 7 へ進む. 満たしていない場合, この推定は間違っているので, ステップ 6 へ進み, 別の推定を行う.

- ステップ 6. ステップ 5 で推定が間違っていると判断された場合は別の推定結果を行うため, ステップ 3 における定数項の割り算結果について別の候補を試行する. 定数項 (o2) が 0 より大きい場合は先ほどの商に 1 を加えたものを新しい商とし, 0 より小さい場合は先ほどの商から 1 引いたものを新しい商とし, それに応じた余りを計算する. この新しい推定結果について, 正答確認を行い, 正しい場合は新しい商と余りを元に三次元目の式推定を行う. ここでは説明のため, ステップ 5 における推定が正しかったとし説明を進める.
- ステップ 7. ステップ 3 における商を二次元目の推定サイズで割る
各ループ変数に関するステップ 3 の商を, 変数項と定数項についてそれぞれ二次元目の推定サイズで割る. i に関する商を二次元目の推定サイズで割ると,

$$(n1 * i + n1 * o0 + o1)/n1 = 1 * i + o0...o1$$

j に関する商を二次元目の推定サイズで割ると,

$$(1 * j)/n1 = 0...1 * j$$

となる.

- ステップ 8. 各 i, j に関する割り算結果のうち, 余りを全て集めて二次元目の式と推定する. 本例では, a[ ][j + o1][k + o2] と推定される.
- ステップ 9. 正しい推定であるかどうか確認する
ステップ 5 と同様に, 推定された式について, 誘導変数の動作範囲に対して以下の条件を満たしているかどうか確認する.

$$0 <= j + o1 < n1$$

満たしている場合はこの推定は正しいとし, 満たしていない場合, 先ほどと同様の手順で新しい推定結果を得る.

- ステップ 10. ステップ 7 における各 i, j に関する割り算結果のうち, 商を全て集めて一次元目の式と推定する. 今回の例では, a[i + o0][j + o1][k + o2] と推定される.

以上の手法により多次元配列的にアクセスされる一次元配列を多次元配列に変換することができる. この変換後の多次元配列情報を仮想的に保持し, 依存ベクトルを計算し並列性を抽出する. 解析終了後にはこの仮想多次元配列情報は消去することで, 並列性解析時のみ多次元配列としての情報を用いることができる.

3. OSCAR 自動並列化コンパイラ

第 2 節で述べた解析時のデータ構造変換による並列性抽出手法を OSCAR 自動並列化コンパイラに実装した.

3.1 OSCAR 自動並列化コンパイラ概要

OSCAR 自動並列化コンパイラ [5][6] は Fortran 言語や C 言語で記述された逐次プログラムを全域に渡り並列性を自動的に抽出し，階層的な並列処理を実現することができるコンパイラである. OSCAR コンパイラはソースプログラムよりコントロールフロー解析・データ依存解析を行い, 並列化, メモリ最適化, 電力最適化等の最適化を行う. 特に, 並列化手法としてマルチグレイン並列処理 [7][8][11] を特徴としている. マルチグレイン並列処理とは, ループやサブルーチン等の粗粒度タスク間の並列処理を利用する粗粒度タスク並列処理，ループイタレーションレベルの並列処理である中粒度並列処理，および基本ブロック内部のステー

トメントレベルの並列性を利用する近細粒度並列処理を階層的に組み合わせてプログラム全域にわたる並列処理を行う手法である．ここでは本稿で使用する中粒度並列処理について説明する．中粒度並列処理は，ループイタレーション間の並列性を利用するループ並列処理である．コンパイラが依存解析を行い，ループの並列処理が可能であった場合，ループは DOALL，リダクションループ，DOACROSS の 3 種類のいずれかと判定される．DOACROSS ループはループイタレーション間に依存があるが，イタレーション間で同期を取りながら並列化を行うことが可能である．依存ベクトルはイタレーション間の依存情報として利用される．また，コンパイラにより並列処理が不可能と判断されたループは逐次ループと判定される．並列化後, コンパイラは並列化ソースコードを生成する．並列処理用に拡張された OSCAR API を含む Fortran あるいは C ソースコードを出力する．

C 言語においては Parallelizable C[9] というプログラミング規約を提案しており，この規約に従ってコーディングされたアプリケーションは自動で並列化することができる．Parallelizable C の規約によるプログラム記述上の主な制限事項は以下のとおりである．

- 定義された境界を超えたアクセス
- ポインタの算術演算
- ポインタのキャスト
- ループや条件分岐内でポインタへの値の代入
- 同一領域に対する複数ポインタの関数への引数渡し
- 一時バッファとして利用するヒープ領域の使い回し
- 構造体メンバの配列に対する配列アクセス
- 再帰的なデータ構造の使用
- 呼び出し感に依存のある外部関数の使用
- 関数ポインタの使用
- 引数が可変個な関数の定義

3.2 OSCAR コンパイラにおける構造体解析部の概要

OSCAR 自動並列化コンパイラにおけるデータ構造変換による構造体解析の大まかな処理フローを以下に述べる．

1. 仮想的な配列の準備
まず，構造体型の各変数・配列の仮想的な配列の枠組みを用意する．ここで，各構造体の深さの解析を行い，持たせる添え字情報の次元数を決定する．その際に，構造体にリスト構造が含まれていないか解析を行い，含まれている構造体は解析の対象外として，データ構造の変換は行わない．

2. 仮想的な配列の添え字情報作成
各構造体型変数・配列の情報を解析し，その情報を，1 で用意した仮想的な配列の枠組みへ入れて，仮想的な配列に構造体の情報を持たせる．ここで，添え字式に 2 種類以上の変数が使われている構造体が含まれているかを確認し，含まれている構造体は解析の対象外とし，データ構造の変換は行わない．

3. データ構造の変換
変数の定義参照解析内で，対象となる構造体が各ステートメントの変数の定義参照情報に追加される際に，2 で用意した仮想的な配列に置き換える．

4. データ依存解析
3 で解析対象の構造体を仮想的な配列に置き換えた変数の定義参照情報を基に，データ依存解析を行う．

3.3 OSCAR コンパイラにおける配列添え字解析からループ判定までの処理フロー

OSCAR 自動並列化コンパイラにおける配列添え字解析では, 対象添え字を走査し, 変数ごとに係数とオフセットをペアとして保存していく. 配列添え字解析部からループ判定部までの大まかな処理フローは, 以下のようになっている.

- 配列添え字情報の生成
  まず配列添え字情報の生成を行う. 配列添え字情報はループ制御変数の係数とオフセットの 2 つで表され, 1 つの添え字内に複数の変数が含まれていた場合は, 変数ごとに管理される. また, 定数項が複数ある場合は計算され 1 つにまとめられる. 例えば a[10*(i-1) + (j+1)] のような配列があった場合, i, j の配列添え字情報は以下のように生成される.

  $$i : 10 * i - 9$$
  $$j : 1 * j + 0$$

- 多次元配列へ変換可能であるかどうか確認するため, 第 2.2 節のステップ 1, ステップ 2 を行う.
- 変換可能であった場合, 第 2.2 節のステップ 3 からステップ 10 に沿って多次元配列への変換を行う.
- 変換後の多次元配列情報を仮想的な配列添え字情報として保存しておく.
- 生成された仮想的な多次元配列添え字情報を元に依存ベクトルの計算を行う.
- 計算された依存ベクトルによるループ判定を行う.

4. 性能評価

OSCAR コンパイラの配列添え字解析部の拡張により, これまで解析することができなかった構造体や複雑な配列添え字を持つ場合でも解析可能となった. まず 4.1 項で評価環境について述べ, 4.2 項および 4.3 項にて評価結果を述べる.

4.1 評価環境

本稿で評価に用いたサーバーマシンは, 6 コアの Intel(R) Xeon(R) CPU X5670 プロセッサを 2 プロセッサ搭載し

た HA8000/TS20 である．HA8000/TS20 の諸元を**表 1** に示す．

表 1 HA8000/TS20 性能

CPU	Intel Xeon CPU X5670
CPU core	12 cores
CPU Frequency	2.93 GHz
L1 D-Cache	32KB
L1 I-Cache	32KB
L2 cache	256KB
L3 cache	12MB for 6 cores

4.2 構造体の性能評価結果

OSCAR コンパイラの構造体解析の拡張により，構造体に配列が混在する場合も解析が可能となった．そこで，MPEG2 デコードを用いて性能評価を行った．MPEG2 デコードは，6 つのステージから成り立っており，可変長複合化，逆量子化，逆量子化後の各係数値の制限処理，逆離散コサイン変換，動き補償予測，足し合わせ処理を行っている．MPEG2 デコード処理には，スライスレベルでの並列性と，スライス処理内でのマクロブロックレベルでの並列性がある．

評価では，既存の研究により OSCAR 自動並列化コンパイラ向けに構造体の削除などチューニングされた MPEG2 デコードプログラム [10] を構造体を使用した形に一部書き戻したものを使用する．入力ファイルは，300 フレームの 352x240 サイズのストリームデータを用いた．OSCAR コンパイラにおいて本稿で実装した構造体解析を用いた速度向上率を図 6 に示す．

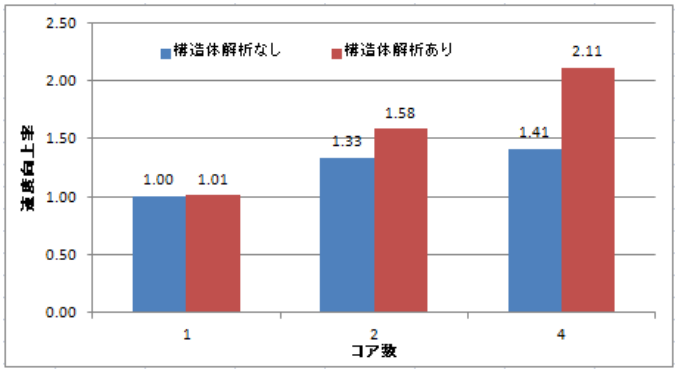

図 6 MPEG2 デコードの速度向上率

今回実装した構造体解析機能を適用することで，構造体の解析精度向上により，並列性を抽出でき，4 コアで 1 コアに対して 2.11 倍の速度向上を得ることができた．また，手法を適用しない場合は 4 コア使用時に 1 コアに対して 1.41 倍の速度向上であったのに対し，手法を適用した場合は 4 コア時に 2.11 倍と，手法を適用しない場合に対して 1.49 倍の性能を得ることができた．MPEG2 デコードには，コストが大きく，並列性のあるループが 2 か所ある．そのうちの片方のループに構造体が含まれており，今回実装した構造体解析機能を適用しない場合は，構造体の解析ができずに，そのループを並列化することができない．そのため，今回実装した構造体解析機能の適用・非適用で実行時間に差が出る結果となった．

4.3 配列添え字解析部拡張の性能評価結果

OSCAR コンパイラの配列添え字解析部の拡張により，複雑な配列添え字を持つ場合でも必要に応じて多次元配列に変換し解析が可能となった．解析可能となったプログラム例を**図 7** に示す．この例の場合，一次元配列 a は 3 次元配列に変換され，DOACROSS ループとして判定することができる．

```
for (i = 1; i < N; i++) {
  for (j = 1; j < N; j++) {
    for (k = 1; k < N; k++) {
      a[n2*n1*i+n2*j+k] = a[n2*n1*(i-1)+n2*(j+1)+k];
}}}
```

図 7 DOACROSS ループ判定例

続いて画像フィルタ処理の一つであるメディアンフィルタ処理の評価結果について述べる．メディアンフィルタ処理とは，注目画素の周囲 9 方向の画素データの中央値を求め，注目画素の値として置き換えるノイズ除去処理の一種である．通常のメディアンフィルタ処理では入力画素と出力画素の指し先が別になっているため依存がないが，本サンプルでは入力画素と出力画素の指し先が同じである一面共有タイプの画像フィルタを対象とした場合でも，DOACROSS ループと見なし並列化が可能となった．メディアンフィルタ処理の OSCAR 自動並列化コンパイラによる速度向上率を**図 8** に示す．

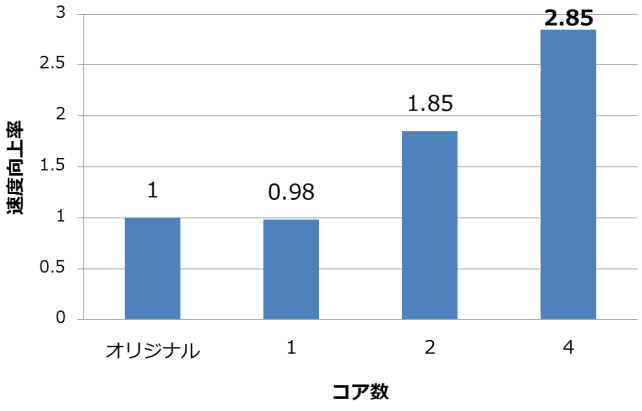

図 8 メディアンフィルタ処理の速度向上率

拡張後の OSCAR 自動並列化コンパイラを用いることで

正しく並列性を抽出し, 4 コアで 1 コアと比べて 2.85 倍の速度向上を得ることができた. このようなプログラムは以前は依存関係が解析できず逐次ループとして判定されてしまっていたが, 拡張後の OSCAR 自動並列化コンパイラを用いることで正しく並列性を抽出することができた.

5. まとめ

本稿では, コンパイラによる解析時のデータ構造変換による並列性抽出手法についてを提案した. OSCAR 自動並列化コンパイラにおいて配列添え字解析部の拡張を行い, 以前は構造体や複雑な配列添え字から依存が解析できず逐次ループ判定となってしまっていたループから並列性を抽出することが可能となった. その結果, MPEG2 デコードについては 4 コアで 1 コアと比べて 2.11 倍，メディアンフィルタ処理については 4 コアで 1 コアと比べて 2.85 倍の速度向上を得ることができた. また，この解析手法により，Parallelizable C による制約を緩和することができ，より多くのアプリケーションを自動で並列化が可能となった.

謝辞

本研究の一部は科研費基盤研究 (C)15K00085 の助成により行われた.

参考文献

[1] Pearce, D. J., Kelly, P. H. J. and Hankin, C.: Field-sensitive pointer analysis for C, ACM Transactions on Programming Languages and Systems (2007).
[2] Mine, A.: Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics, LCTES (2006).
[3] Paek, Y., Hoeflinger, J. and Padua, D.: Efficient and Precise Array Access Analysis, ACM Transactions on Programming Languages and Systems (2002).
[4] Grosser, T.: On Recovering Multi-Dimensional Arrays in Polly, Impact 2015 (2015).
[5] Hironori, K., Motoki, O. and Kazuhisa, I.: Automatic Coarse Grain Task Parallel Processing on SMP Using OpenMP, Workshop on Languages and Compilers for Parallel Computing (2001).
[6] Motoki, O., Jun, S., Hiroki, K., Kazuhisa, I. and Hironori, K.: Hierarchical Parallelism Control for Multigrain Parallel Processing, Lecture Notes in Computer Science (2005).
[7] 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 (1990).
[8] 小幡元樹，笠原博徳，木村啓二：OSCAR チップマルチプロセッサ上でのマルチグレイン並列処理，情報処理学会 (2002).
[9] 木村啓二，間瀬正啓，笠原博徳：「組み込みソフトウェア向けコーディング規約の作成方法」を用いた Parallelizable C の定義，ETNET2012 (2012).
[10] 宮本考道，浅香沙織，見神広紀，間瀬正啓，木村啓二，笠原博徳: 情報家電用マルチコア並列化 API を生成する自動並列化コンパイラによる並列化の評価，情報処理学会論文誌 (2008).
[11] 笠原博徳：並列処理技術，コロナ社 (1991).