情報処理学会論文誌　プログラミング　Vol. 3　No. 1　36–47 (Mar. 2010)

自動並列化のための Element-Sensitive ポインタ解析

間瀬 正啓[†1]　村田 雄太[†1]
木村 啓二[†1]　笠原 博徳[†1]

マルチコアプロセッサの普及にともない，C 言語のような逐次型言語で記述されたプログラムのコンパイラによる自動並列化が期待されている．しかしながら，科学技術計算やメディア処理アプリケーションのアルゴリズムは潜在的に高い並列性を持っていながら，従来のポインタ解析技術では並列性の自動抽出にはしばしば不十分なことがある．たとえば，アルゴリズム上は多次元配列として扱うことが可能なデータ構造を，ポインタへのポインタとメモリ動的確保を行うループの記述により実装する場合がある．このようなデータ構造に対して従来のポインタ解析の適用を考えた場合，配列の各要素の情報が配列全体で単一の情報に縮退されてしまうため，コンパイラによる依存解析ができず，自動並列化が阻害されてしまう．そこで本論文では，ポインタの配列において各要素の指し先のエイリアス関係を識別可能な Element-Sensitive ポインタ解析を提案する．提案する Element-Sensitive ポインタ解析では，既存のポインタ解析手法に対して簡単な追加情報を加えるだけで，自動並列化に有用なポインタ解析精度を得ることができる．ポインタの配列の各要素に指し示されるオブジェクトに重なりがない場合は，そのようなポインタが指し示すデータ構造を多次元配列と同様に並列性抽出の対象とすることが可能となる．また，Element-Sensitive ポインタ解析のアルゴリズムとともに，その解析結果の自動並列化への適用方法についても述べる．自動並列化による速度向上について，8 コア構成のサーバである IBM p5 550Q 上で性能評価を行ったところ，科学技術計算やマルチメディア処理を行う 4 つのアプリケーションプログラムについて，逐次実行時と比較して平均 5.50 倍の速度向上が得られた．

Element-Sensitive Pointer Analysis for Automatic Parallelization

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

As multicore processors become widely used, there is an increasing need to achieve automatic parallelization of sequential programs written in C language by a compiler. However, traditional pointer analysis techniques are often insufficient in automatically extracting parallelism even when target scientific and media applications have large amount of inherent parallelism in their algorithms. For example, a compiler cannot analyze data dependencies for a data structure implemented by a pointer to pointer variable and memory allocation loops, even though the data structure can be essentially realized by a multiple dimensional array in an algorithm. For these data structures, traditional pointer analysis techniques aggregate information for each element in an array of pointer into information for the whole array, hamper automatic parallelization. This paper proposes element-sensitive pointer analysis, which can distinguish the alias relation among elements in an array of pointer. The proposed element-sensitive pointer analysis can acqire a sensitivity beneficial for automatic parallelization through just adding simple information to conventional pointer analyses. When there is no overlap among objects pointed-to by different array elements of the array of pointers, the pointed data structure can be treated as same as multiple dimensional array in parallelism extraction. The element-sensitive pointer analysis algorithm is described as well as its application method for automatic parallelization is shown. As the result, on IBM p5 550Q 8 cores server, automatic parallelization achieves 5.50x speedup against sequential execution on average for 4 application programs with element-sensitive pointer analysis.

1. はじめに

組み込み機器から PC，スーパコンピュータに至るあらゆる情報機器において，マルチコアプロセッサが主流となってきており，並列ソフトウェア開発の生産性向上のため自動並列化コンパイラの実用化が期待されている．従来より科学技術計算分野の FORTRAN プログラムに対しては，Polaris[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．より広く利用されている C 言語においても自動並列化コンパイラの利用が望まれているが，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，コンパイラがこれらのポインタの指し先を一意に解析できない場合，依存解析の精度が劣化し自動並列化の阻害要因となってしまう．

現状では多くの研究用あるいは商用の自動並列化コンパイラにおいて，C プログラムに対する効果的な自動並列化や自動ベクトル化を期待する場合は，すべてのポインタに対する restrict 修飾[4)] や，ポインタエイリアスの扱いに関する独自のコンパイラオプションを利用

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

することが推奨されており，事実上まったくポインタを使用しないプログラムを記述する必要がある．しかしながら，コンパイラによるポインタ解析の精度を高めることにより，プログラム記述に対する制約条件が緩和され，高度なコンパイラ最適化技術の利用によるマルチコア，メニーコア向けのソフトウェア生産性の向上が可能と考えられる．

ポインタ解析に関しては多くの研究[5]が行われているが，C 言語のあらゆる記述に対してポインタの指し先を静的に特定するのは事実上不可能となっている．そのため，科学技術計算やメディア処理アプリケーションのアルゴリズムは潜在的に高い並列性を持っていながら，従来のポインタ解析技術では並列性の自動抽出にはしばしば不十分なことがある．

そのような例の 1 つに，アルゴリズム上は多次元配列として扱うことが可能なデータ構造を，ポインタへのポインタとメモリ動的確保を行うループの記述により実装する場合がある．多次元配列と同様の機能を実現する際に，C 言語では**図 1** のように malloc によって各次元に相当する領域を確保し，ポインタの参照によってアクセスすることがある．図 1 の例では，添え字とメモリアドレスが 1 対 1 の関係になっており，i および j を変数としたとき $a[i][j]$ は必ず一意なメモリアドレスへの参照となっている．一般に，アルゴリズム上は多次元配列となっていても，各次元の要素数が入力によって決定される場合には，C 言語ではこのようなデータ構造が静的な配列の代替として利用されることが多い．

この例では，LOOP3 の外側ループにおいてイタレーションごとの並列処理が可能である．しかし，このようなポインタへのポインタとヒープで確保されたデータ構造に対して従来のポインタ解析の適用を考えた場合，多くのポインタ解析では配列の個々の要素の指し先を独立には解析しないため，ポインタ配列の各要素の指し先情報は配列全体で単一の情報に縮退される．そのため，図 1 における LOOP3 の外側ループでは，a[i] の示す各要素の間のエイリアスの関係が不明となることから，ループイタレーション間の依存解析ができず，自動並列化が阻害されてしまう．

```
/* 多次元配列のようなデータ構造を確保 */
BB1:
a = (int **)malloc(n * sizeof(int *))
LOOP2:
for (i = 0; i < n; i++) {
    a[i] = (int **)malloc(m * sizeof(int *));
}
/* 確保したデータ構造に計算結果を代入 */
LOOP3:
for (i = 0; i < n; i++) {
LOOP3_1:
    for (j = 0; j < m; j++) {
        a[i][j] = i + j;
    }
}
```

図 1　従来のポインタ解析では識別できないプログラム例
Fig. 1　Program example that conventional pointer analysis does not distinguish.

そこで本論文では，ポインタの配列において各要素の指し先のエイリアス関係を識別可能な Element-Sensitive ポインタ解析を提案する．ポインタ配列の各要素に指し示されるオブジェクトに重なりがない場合は，このようなデータ構造に対するアクセスを含むループに対し，依存距離と依存ベクトルによるイタレーション間依存解析が適用可能となり，並列性の抽出が可能となる．

提案する Element-Sensitive ポインタ解析では，既存のポインタ解析手法に対して簡単な追加情報を加えるだけで，自動並列化に有用なポインタ解析精度を得ることができる．Element-Sensitive ポインタ解析の利点としては，(1) 既存の逐次プログラムが自動並列化可能となることによる速度向上，(2) 既存の逐次プログラムを書き換えることで自動並列化による速度向上を得ようとした場合のコード書き換え量の削減，(3) ループリストラクチャリング等のプログラム最適化過程におけるポインタ指し先情報の表現として利用，そして (4) 解析アルゴリズムがシンプルであり軽量で効率的な実装が可能，といったことがあげられる．

本論文の構成を以下に示す．従来のポインタ解析に関する研究成果を利用し，Flow-Sensitive[6)–9)]，Context-Sensitive[6)–12)]，Heap-Sensitive[11),12)]，Field-Sensitive[12),13)] ポインタ解析を基準となるポインタ解析精度として採用し，さらにこれらに加えて，従来あまり議論されていなかった Cycle-Sensitivity[14)]，Element-Sensitivity について焦点を当てる．まず 2 章では基準となる既存のポインタ解析手法について述べ，3 章では新たに提案する Element-Sensitive ポインタ解析について述べる．次に 4 章では Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析の併用について述べ，5 章でポインタ解析結果の自動並列化への利用方法について述べる．そして，6 章では Element-Sensitive ポインタ解析を利用して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およびさらなるポインタ解析精度向上の可能性について述べる．最後に，7 章で関連研究について述べ，8 章でまとめを述べる．

2. 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変数がメモリ上のどの領域を指すかを解析するものである．データ依存の解析，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[15] の精度はポインタ解析の精度に大きく依存する．そのため，ポインタ解析ではポインタの指し先を一意に決定できることが望まれる．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，解析精度と解析コストのトレードオフが問題となる[5]．

本章では，本論文において基準となるポインタ解析精度として採用した，Flow-Sensitive, Context-Sensitive，Heap-Sensitive，Field-Sensitive ポインタ解析について述べる．

2.1 Points-to 解析

ポインタ解析では，メモリ上のあるポインタオブジェクトとそのポインタが指し示す可能性があるオブジェクトを解析するものが主流であり，Points-to 解析とも呼ばれる．この Points-to 解析で対象とするオブジェクトは，プログラム中で宣言されるスカラ変数や配列変数あるいはヒープ上の領域等を示し，しばしば複数のメモリ位置を 1 つの名前に抽象化して扱うため，抽象化されたメモリ位置（abstract memory location）とも呼ばれる．Points-to 解析のイメージを**図 2** に示す．図中では，ポインタの指し先関係を → を用いて表し，$(a \rightarrow b)$ はポインタオブジェクト a がオブジェクト b を指す可能性があることを示している．このようなポインタの指し先関係の集合を Points-to 集合と呼び，Points-to 解析は Points-to 集合を解析する．Points-to 集合におけるポインタの指し先情報は，各オブジェクトをノード，指し元オブジェクトから指し先オブジェクトへの関係をエッジで表現した有向グラフ（Points-to グラフ）として扱うことができる．

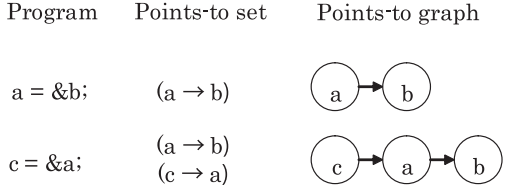

図 2　プログラムと Points-to 集合の例
Fig. 2　Example of program and points-to set.

2.2 Flow-Sensitive ポインタ解析

Flow-Sensitive ポインタ解析はプログラムのコントロールフローを考慮して，プログラムの各地点ごと，一般的にはステートメントごとに個別の解析情報を生成する．Flow-Insensitive 解析ではステートメントの順序を考慮せず，サブルーチンやプログラム全体で有効である単一の解析情報を生成するため，Flow-Sensitive よりも保守的な解析結果となる．

一般的に，Flow-Sensitive ポインタ解析は，データフロー解析[16] の枠組みを用いて実装される．すなわち，対象となる束（lattice）L，束の交わり（meet）演算子 $\wedge$，伝達関数（transfer function）F で定義される．ポインタ解析においては，束 L は Points-to 集合，交わり演算子 $\wedge$ は和集合 $\cup$ となり，伝達関数は各ステートメントにおける入力から出力に対する Points-to 集合の変化を計算する．また，ポインタ解析はコントロールフローグラフに沿って，順方向に行われる．

コントロールフローグラフにおける各ノードを k とすると，各ノードは入力するポインタ解析情報 IN_k と出力するポインタ解析情報 OUT_k の 2 つの Points-to 集合を保持する．それぞれのノードにおいて，IN_k から伝達関数 F_k を用いて OUT_k を計算する．このときのノード k におけるデータフロー方程式は以下のようになる．

$$IN_k = \bigwedge_{x \in pred(k)} OUT_x \tag{1}$$

$$OUT_k = F_k(IN_k) \tag{2}$$

このときの伝達関数 F_k は，以下のとおりである．

$$F_k = gen_k \wedge (IN_k - kill_k) \tag{3}$$

ここで，gen_k にはそのステートメントによって新たに生成される Points-to 集合が登録される．また，$kill_k$ は更新されるポインタに対して強い更新（strong update）を行うか，弱い更新（weak update）を行うかによって内容が異なる．新たに生成される Points-to 集合において，あるポインタの指し元オブジェクト v が確実に更新され，v が単一のポインタに対応することが決定できる場合は，強い更新が行われる．すなわち v の更新前の指し先情報が $kill$ 集合に登録されて，その結果データフロー方程式により除去されることになる．一方，ポインタの指し元オブジェクト v が確実に更新されるとは限らない，あるいは v が単一のポインタに対応することが決定できない場合は，弱い更新が行われる．すなわち $kill$ 集合を空集合とすることでそれまでのポインタ指し先情報はすべて保持される．

本論文のポインタ解析は Emami らの Flow-Sensitive，Context-Sensitive ポインタ解析のアルゴリズム[6] をベースとして実装している．Flow-Sensitive ポインタ解析は解析コス

トが大きいことが問題となっていたが，近年は現実的な時間で解析可能なアルゴリズムが提案されてきている[8),9)]．

2.3 Context-Sensitive ポインタ解析

Context-Sensitive ポインタ解析では関数呼び出しの関数のコールパスごとに個別に解析情報を生成することで，解析精度を高めている．Context-Insensitive ポインタ解析では，コールパスを区別せず，ポインタの引数間のエイリアスの関係が異なる場合は，それらの情報が縮退され，保守的な解析結果となってしまう．

Emami らの Flow-Sensitive，Context-Sensitive アルゴリズム[6)] では，引数におけるポインタ解析情報の関係に応じてサブルーチンをクローニングすることで Context-Sensitive 解析を実現している．関数呼び出しにおいてポインタを引数として渡す場合，呼び出された関数では引数のポインタを介して，スコープ外の変数にアクセスすることがある．そのような場合は，スコープ外の変数を不可視変数（invisible variable）として定義し，解析情報を表現する．インタープロシージャ解析では，Mapping によって呼び元の関数の変数を呼び出される関数の不可視変数に対応付けを行い，呼び元における解析情報から呼び先における解析情報を合成する．そして，呼び先の関数の解析後に Unmapping を行う．すなわち，呼び先の関数における解析情報から呼び元における解析情報を合成する．

2.4 Heap-Sensitive ポインタ解析

ヒープ領域の抽象化についてもこれまでに多くの議論が行われてきたが，多くのポインタ解析では，メモリ動的確保を行うステートメントごとに別のオブジェクトとしている[8),9)]．しかし，メモリ確保用のラッパ関数を用意していた場合，同じデータ型のオブジェクトを確保すると，それらがすべて同一のオブジェクトとして縮退してしまい，解析の精度が低下する．

これを解決するために，ヒープを確保した際にメモリ動的確保が行われたコールパスごとに別名のオブジェクトとする手法があり，これらは Heap-Sensitive，あるいは Heap-Cloning と呼ばれる[11),12)]．

本論文ではこの Heap-Sensitive ポインタ解析を採用している．この Heap-Sensitive 解析はインタープロシージャ解析時に，呼び出された関数の解析情報を呼び出し元の関数に Unmapping する際に，そのつど，別名のオブジェクトとして扱うことで実現できる．

2.5 Field-Sensitive ポインタ解析

Field-Sensitive ポインタ解析[13)] では構造体の各メンバを区別して，それぞれ別のオブジェクトとして解析を行う．Field-Insensitive ポインタ解析では，構造体全体を単一のオブジェクトとして扱い，各メンバの指し先情報が縮退される．構造体をプログラムの主要なデータ構造として利用している場合は，高い解析精度を得るために Field-Sensitive 解析が重要となるため，本論文では Field-Sensitive ポインタ解析を採用している．

3. Element-Sensitive ポインタ解析

本章では，新たに提案する Element-Sensitive ポインタ解析について述べる．多くのポインタ解析では，ポインタの配列に対して各要素の指し先を区別せず，各要素が指す可能性があるすべての領域を，単一の指し先情報に縮退して表現していた．しかし，これでは図 1 で示すようなポインタへのポインタとメモリ動的確保を用いたようなデータ構造において十分な解析精度が得られず，自動並列化の阻害要因となってしまう．このような例では，ポインタの配列において各要素ごとのポインタの指し先を識別することが必要となる．

そこで，本章では既存のポインタ解析の各オブジェクトに対して *element alias* 属性という真偽値型の変数を追加するだけで効率的に Element-Sensitive 解析を実現する方法を述べる．この Element-Sensitive ポインタ解析のデータ構造およびデータフロー解析におけるアルゴリズムを示す．

3.1 *element alias* 属性

本論文で提案する Element-Sensitive ポインタ解析は，ポインタの配列における任意の 2 要素が指し示すオブジェクトのエイリアスの有無を，*element alias* 属性を持たせることで区別する．**図 3** のイメージ図のように，*element alias* 属性は真偽値型の変数であり，各要素の指し先にエイリアスがある場合は真，ない場合は偽の値を保持する．このポインタ型の配列オブジェクトに追加される *element alias* 属性がデータフロー解析の枠組みにより解析される．その際，*element alias* 属性の束（lattice）は，⊤ が偽（FALSE），⊥ が真（TRUE）となり，各オブジェクトの *element alias* 属性は偽の値で初期化され，後述の伝達関数によ

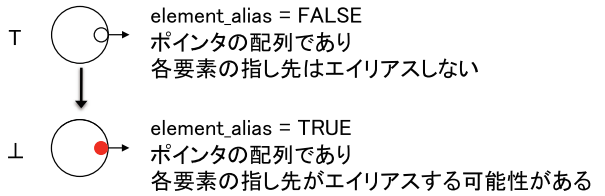

図 3　*element alias* 属性の束（lattice）
Fig. 3　Lattice of *element alias* attribute.

り解析が行われる．

3.2 伝達関数

Element-Sensitive ポインタ解析の伝達関数では，*element alias* 属性を計算するために，従来の Flow-Sensitive ポインタ解析における *gen* の計算に若干の変更を行う．従来の *gen* の計算を gen_{fs} とすると，Element-Sensitive 解析における gen_{es} は以下のようになる．

$$gen_{es} = calc\_element\_alias(gen_{fs}, IN) \quad (4)$$

すなわち，gen_{fs} と IN を入力として，*element alias* 属性の計算 *calc_element_alias* を行う．データフロー解析において，*element alias* 属性を計算するためのアルゴリズム *calc_element_alias* を**図 4** に示す．

element alias 属性の計算を行う関数 calc_element_alias は，各ステートメントにおける既存の Flow-Sensitive ポインタ解析の Points-to 集合 *gen* と Points-to 集合 IN を入力とし，*element alias* 属性を更新された Points-to 集合 *gen* を返り値として返す．calc_element_alias では，*gen* に含まれるポインタの指し先関係（$lh\_ptr \rightarrow rh\_object$）における，各ポインタの指し元オブジェクト *lh_ptr* について，*lh_ptr* がポインタの配列である場合に *lh_ptr* の *element alias* 属性を計算する．*lh_ptr* がポインタの配列である場合，更新されるポインタ要素が一意とはならないため強い更新が行われることはなく，*kill* 集合は必ず空集合となる．すなわち，伝達関数による処理後の *lh_ptr* の指し先は，ステートメントの IN に含まれる *lh_ptr* の指し先の集合 *rh_objects_IN* と，*gen* 集合で生成された *lh_ptr* の指し先の集合 *rh_objects_gen* の和集合となる．ここで，*lh_ptr* のそれまでの指し先のオブジェクトの集合 *rh_objects_IN* と，新たなポインタの指し先として登録されるオブジェクトの集合 *rh_objects_gen* に重なりがある場合は，ポインタの配列の各要素の指し先がエイリアスする可能性があるため，*gen* における *lh_ptr* の *element alias* 属性を真とする．一方，*rh_objects_gen* と *rh_objects_IN* に重なりがない場合は，この更新によってポインタの配列オブジェクト *lh_ptr* の各要素の指し先にエイリアスが発生する可能性はないため，*gen* における *lh_ptr* の *element alias* 属性は偽のままとなる．

```
function calc_element_alias (Points_to_set gen, Points_to_set IN)
    return : Points_to_set
begin
    for each lh_ptr ∈ {lh_ptr | ∃rh_object (lh_ptr → rh_object) ∈ gen}
        if lh_ptr is array of pointer then
            rh_objects_IN = {rh_object | (lh_ptr → rh_object) ∈ IN};
            rh_objects_gen = {rh_object | (lh_ptr → rh_object) ∈ gen};
            if rh_objects_IN ∩ rh_objects_gen ≠ φ then
                lh_ptr.element_alias = TRUE;
            endif
        endif
    endfor
    return gen;
end
```

図 4　gen における *element alias* 属性計算アルゴリズム
Fig. 4　Algorithm of calculating *element alias* attribute on gen.

3.3 交わり（meet）演算子 ∧

データフロー解析の交わり（meet）演算子 ∧ は，*element alias* 属性に関しては図 3 のように真偽値の 2 通りの値のみをとりうるため，あるオブジェクトについてすべてのパスでその *element alias* 属性が偽の場合は偽，1 つでも真であれば真となる．

各ステートメントの *gen* の計算では，IN における Points-to 集合と *element alias* 属性を入力として *element alias* 属性を計算していたが，コントロールフローの交わり ∧ ではオブジェクトの指し先の関係は考慮せずに各オブジェクトの *element alias* 属性のみを入力として演算を行う．これは，実際にプログラムを実行する際には，交わり後のポインタオブジェクトが指し示すのはその先行パスのいずれかで指し示していた指し先となるためである．このとき，オブジェクトのエイリアス関係には指し先のオブジェクトの重複は影響しないため，各パスの Points-to 集合に重なりがあった場合でも，すべての先行パスの *element alias* 属性が偽であれば，交わり後の *element alias* 属性は偽とすることができる．

3.4 インタープロシージャ解析

インタープロシージャ解析では，複数のオブジェクトを単一の不可視変数に Mapping する場合に，*element alias* 属性の Mapping について考慮する必要がある．この場合，交わり（meet）演算子 ∧ と同様に，呼び出し元におけるそれぞれのオブジェクトの *element alias* 属性が 1 つでも真の場合は Mapping された不可視変数の *element alias* 属性は真，そうでなければ偽とする．

呼び先におけるオブジェクトから呼び元におけるオブジェクトへの Unmapping 時には，解析情報の合成は起こらないため，呼び先の情報から対応する呼び元におけるオブジェクトの *element alias* 属性を設定する．

3.5 Cycle-Sensitive 不使用時の Element-Sensitive ポインタ解析の適用例

図 1 のコード例について，Cycle-Sensitive を使用せずに Element-Sensitive ポインタ解析のみを適用した場合の解析イメージを**図 5** に示す．この例ではまずポインタへのポインタ a の指し先として，ポインタの配列オブジェクトがヒープ領域に確保され，ポインタ解

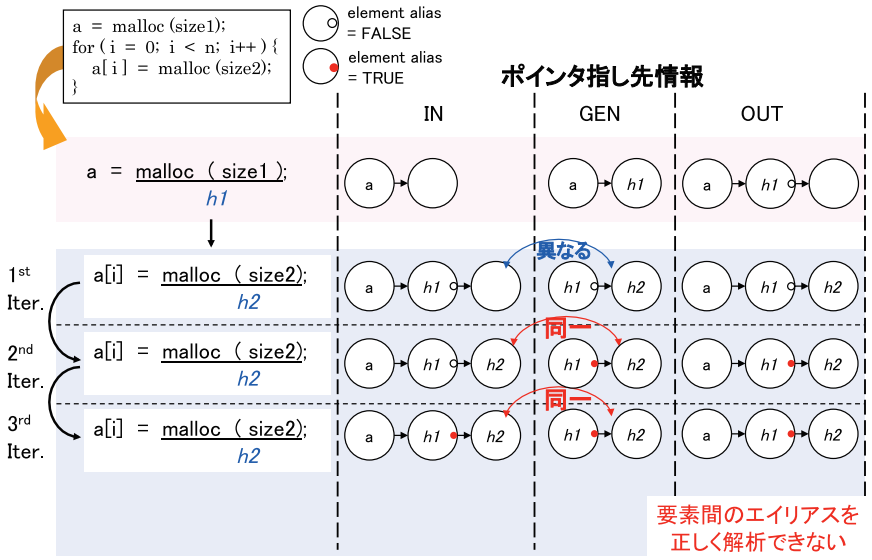

図 5　Cycle-Sensitive を利用しない Element-Sensitive ポインタ解析の適用例
Fig. 5　Example of applying element-sensitive pointer analysis without cycle-sensitive analysis.

析においては h1 という名前で識別される．次に，a[i] の各要素の指し先として，ループ中でそれぞれヒープ領域を確保しており，これらをまとめて 1 つのオブジェクトとして扱い，h2 という名前が付けられる．すなわち，プログラム実行時には a[i] の指し先は各イタレーションで確保されたそれぞれ独立した領域を指し示すことになるが，ポインタ解析におけるヒープの名前付けはプログラム中の静的なステートメントの場所によって行われるため，各イタレーションで確保されるヒープ領域はすべて h2 という名前に縮退されて表現される．

データフロー解析では，ループについては反復的に解析を行い，解析結果が収束するまで計算を続ける．この例ではまず 1 イタレーション目では「h1 が h2 を指す」という解析情報が *gen* として生成される．この時点では，*IN* において h1 は未初期化状態のため，このイタレーションでは h1 の指し先が *IN* と *gen* で重複することはなく，*OUT* は h1 が h2 を指し，h1 の *element alias* 属性は偽という情報となる．

次にデータフロー解析の 2 イタレーション目でも，h1 が h2 を指すという解析情報が *gen* として生成される．しかし，このときはすでに *IN* において「h1 が h2 を指す」という情報が存在するため，*IN* と *gen* において h1 の指し先が重複する．そのため，*gen* における h1 の *element alias* 属性は真となり，*OUT* は h1 が h2 を指し，h1 の *element alias* 属性は真となる．3 イタレーション目は 2 イタレーション目と同様に処理が進み，*OUT* が 2 イタレーション目の *OUT* と同一になるため，h1 が h2 を指し，h1 の *element alias* 属性が真という結果でデータフロー解析が収束する．

この例においては，各イタレーションで確保されたヒープオブジェクトが単純に同一の名前のオブジェクト h2 として扱われるため，各イタレーションで確保する領域が別の領域であることを認識できず，*element alias* 属性を正しく偽と設定できなかったことになる．そこで，4 章では Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析を併用することで，*element alias* 属性を解析する方法を示す．

4. Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析の併用

ループのイタレーションを識別する Cycle-Sensitive 解析を実現する方法の 1 つとして，Aging [14] という手法がある．この手法では，ヒープオブジェクトの名前付けを行う際に，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ことで，簡易な実装で Cycle-Sensitive 解析を実現する．Aging により各イタレーションで確保するヒープ領域を識別することで，Element-Sensitive ポインタ解析において，*element alias* 属性を適切に設定することが可能となる．

4.1 Element-Sensitive ポインタ解析と Aging の併用例

図 1 におけるデータ構造構築部分のコードを例に，Element-Sensitive ポインタ解析と Aging を併用した場合の解析イメージを**図 6** に示す．

この例においても a が h1 を指すという情報を生成する地点までは，図 5 と同様であるが，a[i] の指し先を動的確保するループの部分について，Aging を利用することで解析の流れが変わってくる．

まず 1 イタレーション目では「h1 が h2 を指す」という解析情報が *gen* として生成される．このとき，Aging を用いる場合は，新たに確保されたヒープオブジェクトについては，NEW という属性が設定され，現在のループイタレーションで確保されたオブジェクトであることが明示される．この時点では，*IN* において h1 は未初期化状態のため，このイタレーションでは h1 の指し先が *IN* と *gen* で重複することはなく，*OUT* は h1 が h2 を指し，h1 の *element alias* 属性は偽という情報となる．

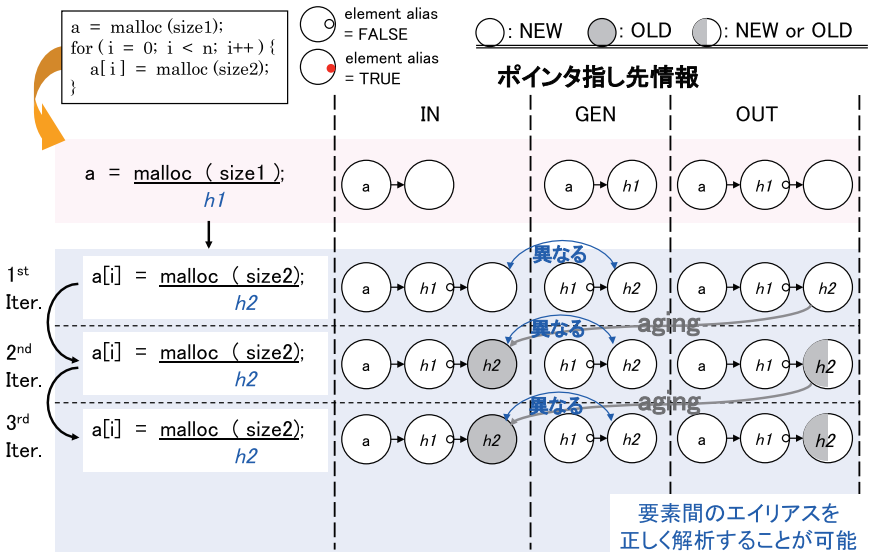

図 6　Element-Sensitive ポインタ解析と Aging の併用例
Fig. 6　Example of applying element-sensitive pointer analysis with aging.

次にデータフロー解析の 2 イタレーション目の処理の開始時に Aging が適用され，「h1 が h2 の NEW のオブジェクトを指す」という情報が「h1 が h2 の OLD のオブジェクトを指す」という情報に変化する．すなわち，*IN* の情報は h1 が h2 の OLD を指す，という情報となる．ここで *gen* の生成は 1 イタレーション目と同様に h1 が h2 の NEW を指すという解析情報となるため，h1 の指し先は *IN* では h2 の OLD，*gen* では h2 の NEW となり，異なるオブジェクトなので h1 の *element alias* 属性は偽となる．そして，*OUT* は h1 が h2 の NEW または OLD を指し，h1 の *element alias* 属性は偽となる．3 イタレーション目の処理の開始時に再度 Aging が行われ，「h1 が h2 の NEW または OLD を指す」という情報が「h1 が h2 の OLD を指す」という情報に変化する．その後は 2 イタレーション目と同様に処理が進み，*OUT* が 2 イタレーション目の *OUT* と一致するため，h1 が h2 の NEW または OLD を指し，h1 の *element alias* 属性は偽という結果でデータフロー解析が収束する．このように，Aging の併用により正しく *element alias* 属性を解析することができる．

5.　Element-Sensitive ポインタ解析の自動並列化への利用

本章では Element-Sensitive ポインタ解析による解析情報の自動並列化への適用について述べる．Element-Sensitive ポインタ解析における *element alias* 属性の情報は，ポインタへのポインタを介したアクセスに対して，配列ベースのプログラム向けの自動並列化技術[15),17),18)] を適用するための条件となっている．これにより多次元配列をデータ構造としてプログラムを記述した場合と同様に自動並列化が適用可能となる．

5.1　イタレーション空間における依存解析

自動並列化のためのループレベル並列性やデータローカリティ最適化はイタレーション空間におけるループ依存解析結果をもとに行う．多次元配列においては，配列の添字とメモリアドレスが一意に対応可能なことから，ループ制御変数と依存ベクトルを用いてイタレーションスペースにおける依存解析が可能である．

ポインタアクセスされる領域についても，対象のループにおいてそのポインタを用いて添字アクセスを行う場合に，その添字とアクセスするメモリアドレスが一意に対応可能なことが保証できれば，多次元配列と同様にイタレーション空間における依存解析が可能となる．

5.2　イタレーション空間における依存解析の例

図 1 の例と同様のコードサンプル（**図 7** (a)）を用いた場合の Element-Insensitive および Sensitive ポインタ解析結果と，それぞれの場合のイタレーション空間における依存解析結果を示す．

図 7 (b) の Element-Insensitive ポインタ解析を適用した場合は，保守的にポインタの配列 a[i] の各要素が指しうるオブジェクトに重複がある可能性があるものとして解析する．指しうるオブジェクトに重複がある可能性がある場合は，Points-to グラフの葉ノードにあたるオブジェクト，すなわち添字の最右側次元のアクセスに対応するメモリ領域については，添字とメモリ領域との対応が一意となるため，安全に依存解析を行うことができる．よって，図中 a[i][j] の j によるアクセスではオブジェクトの先頭からのオフセットは j の値により一意に決定される．しかし，左側次元の添字については，メモリ上の領域との対応関係が不明なため，多次元配列のように依存解析を行うことはできない．

それに対し，Element-Sensitive ポインタ解析を適用した場合（図 7 (c)）は，ポインタの配列 a[i] の各要素が指す領域が独立していることを保証できるため，a[i][j] の各次元の添字のパターンとそのアクセス先のメモリアドレスの対応付けが可能となり，多次元配列と同様に扱うことができる．

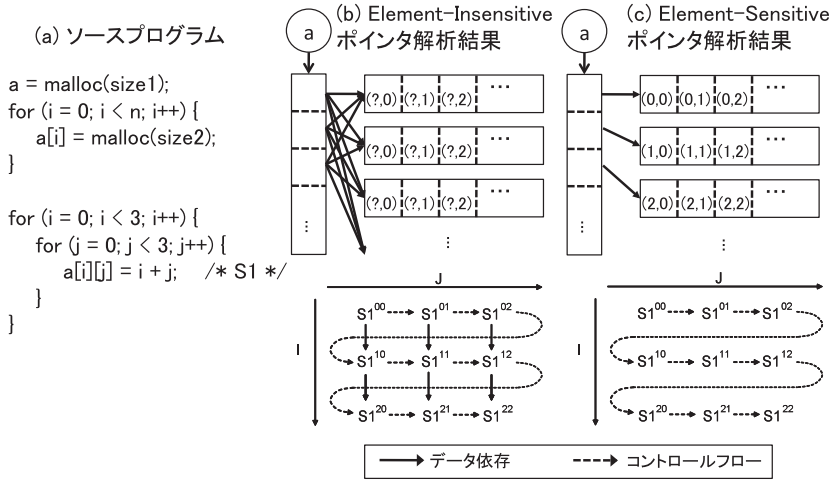

図 7　ポインタ解析を利用したイタレーション空間における依存解析結果
Fig. 7　Dependence analysis results in iteration space using pointer analysis.

6. マルチコアシステム上での性能評価

本章では，Element-Sensitive ポインタ解析を適用した際の，コンパイラによる自動並列化結果について述べる．Element-Sensitive ポインタ解析の有無による処理性能の違いについて評価する．

2 章で述べた基準となる Flow-，Context-，Heap-，Field-Sensitive ポインタ解析，および 3 章で述べた新たに提案する Element-Sensitive ポインタ解析と 4 章で述べた Aging（Cycle-Sensitive ポインタ解析）を，OSCAR 自動並列化コンパイラ[3] に実装し評価を行った．OSCAR コンパイラを用いて自動並列化した際の 8 コア構成の SMP サーバである IBM p5 550Q 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．

6.1　自動並列化を適用する際のコンパイル手順

OSCAR コンパイラは C 言語等のソースコード変換をサポートしており，逐次の C 言語で記述されたプログラムに対して，OSCAR コンパイラは自動並列化を適用し，OpenMP[19] で並列化された C プログラムを自動生成する．このときに利用する OpenMP 指示文は parallel sections，flush，critical の 3 種類の指示文のみであり，OpenMP 仕様のうちこれらの構文をサポートしている環境であれば，並列コード生成が可能である．この 3 つのディレクティブは情報家電向けのマルチコア用並列処理 API である OSCAR API[20] においても，並列処理に必要な OpenMP 指示文のサブセットとして定義されている．

この OSCAR コンパイラにより自動生成された 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．

表 1　評価環境
Table 1　Evaluation environment.

System	IBM p5 550Q
CPU	Power5+ (1.5 GHz × 2 × 4)
L1 D-Cache	32 KB for 1 core
L1 I-Cache	64 KB for 1 core
L2 cache	1.9 MB for 2 cores
L3 cache	36 MB for 2 cores
Native Compiler	IBM XL C/C++ for AIX Compiler V10.1
Compile Option	OSCAR: -O5 -qsmp=noauto Native: -O5 -qsmp=auto
備考	SMT: disabled

6.2　評価対象プログラム

SPEC2000 より art のオリジナルコード，equake を並列処理向けに修正を行ったコード，および SPEC2006 より hmmer，lbm をポインタ解析精度を考慮して修正を行ったコード（Parallelizable C Level 2[21]）に対して自動並列化を適用した．この Parallelizable C コードは，Element-Sensitive ポインタ解析を用いてもオリジナル C コードではポインタの指し先が解析できない場合に，ポインタ解析精度を考慮したソースコードの修正を行うことで自動並列化を適用可能としたものである．

6.3　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 において評価を行った．評価環境のパラメータを**表 1** に示す．

ただし，本論文の評価では equake のみ IBM XL C/C++コンパイラの最適化レベルを-O4 としている．これは，OSCAR コンパイラで自動生成した OpenMP プログラムを IBM XL C/C++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

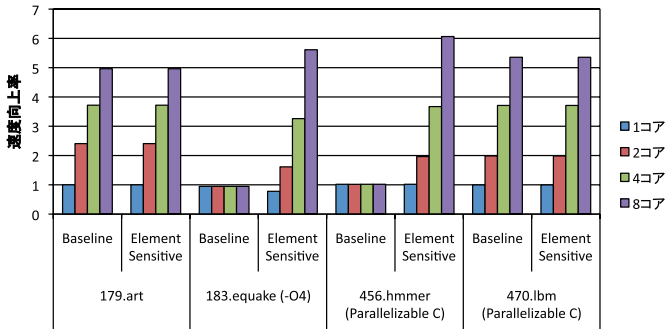

図 8　IBM p5 550Q における OSCAR コンパイラによる自動並列化結果
Fig. 8　Automatic parallelization results of OSCAR Compiler on IBM p5 550Q.

評価では並列処理による速度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．

6.4　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**図 8** に示す．

図中，横軸が各アプリケーションと使用するポインタ解析における Element-Sensitive および Cycle-Sensitive の有無を示す．Baseline は基準となる Flow-，Context-，Heap-，Field-Sensitive ポインタ解析を利用して自動並列化を適用した場合，Element Sensitive は基準となるポインタ解析に加え，Element-Sensitive ポインタ解析と Aging を併用した場合を示す．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 1 コア，2 コア，4 コア，8 コアとなる．

6.4.1　IBM p5 550Q における処理性能

art，equake については Element-Sensitive ポインタ解析により自動並列化に必要なポインタの指し先情報を解析可能であり，図 8 に示すように，8 コア使用時に逐次実行時と比較して，art で 4.96 倍，equake で 5.61 倍の速度向上が得られた．hmmer，lbm についてはオリジナルコードは Element-Sensitive ポインタ解析を用いても十分なポインタ解析結果を得ることができなかったが，ポインタの利用法について若干の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 8 コア使用時に hmmer で 6.06 倍，lbm で 5.35 倍の速度向上が得られた．これらの，ポインタ解析の解析精度を考慮して書き換えを行ったプログラムを含めると，Element-Sensitive ポインタ解析を用いた自動並列化により，逐次実行と比較して 8 コア使用時に平均 5.50 倍の性能向上が得られた．

このうち，Element-Sensitive ポインタ解析の利用によって大きな並列化効果が得られたのは equake と hmmer である．equake と hmmer については，基準となるポインタ解析では自動並列化による速度向上がまったく得られていなかったが，Element-Sensitive ポインタ解析を利用することで自動並列化による大きな速度向上が得られた．

6.5　OSCAR コンパイラによる自動並列化結果

評価を行った art，equake，hmmer，lbm の各アプリケーションプログラムについて，適用された自動並列化の概要およびポインタ解析結果を以下に示す．

6.5.1　SPEC2000 art

art では，オリジナルの C ソースコードを修正することなく，自動並列化による速度向上を得ることができた．art では，Element-Insensitive の基準となるポインタ解析において，OSCAR コンパイラの自動並列化により 8 コア使用時に 4.96 倍の性能向上が得られている．

art の主要なデータ構造は，複数のスカラ変数をメンバに持つ構造体の配列がヒープ領域に確保され，ポインタを介してアクセスされる形状となっている．すなわち，主要なデータ構造は構造体の 1 次元配列となっており，主要なループは構造体の配列の各要素に対して処理を行う．そのため，基準となるポインタ解析でも十分にポインタの指し先が解析されていた．

6.5.2　SPEC2000 equake

equake では Element-Insensitive ポインタ解析では自動並列化による速度向上を得ることができなかったが，Element-Sensitive ポインタ解析により 5.61 倍と大きな速度向上が得られた．

これは，equake の主要なデータ構造が図 1 で示したようなポインタへのポインタとヒープ領域を用いたデータ構造となっており，解析には Element-Sensitive ポインタ解析が必要不可欠となるためである．

6.5.3　SPEC2006 hmmer

オリジナルコードでは Element-Sensitive ポインタ解析を用いても，自動並列化による速度向上を得ることができなかったため，自動並列化可能となるように Parallelizable C に書き換えたコードに対する自動並列化結果を示している．Parallelizable C で記述することで，

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 倍の速度向上が得られた．

hmmer のソースコード中にはポインタの配列は存在しないが，自動並列化の過程で，ループディストリビューションおよびスカラエキスパンションが適用され，その後に Element-Sensitive ポインタ解析が行われることで，大きな速度向上が得られた．すなわち，自動並列化の過程において，ソースコード上でスカラ変数であったポインタ変数を，コンパイラの中間表現ではポインタの配列として扱っている．このように，コンパイラにおけるプログラムリストラクチャリング適用後の解析情報の表現に Element-Sensitive ポインタ解析が有用となる．

6.5.4　SPEC2006 lbm

lbm においても，オリジナルコードでは Element-Sensitive ポインタ解析を用いても，自動並列化による速度向上を得ることができなかったため，図 8 では自動並列化可能となるように Parallelizable C に書き換えたコードに対する自動並列化結果を示している．Parallelizable C で記述することにより，OSCAR コンパイラによる自動並列化により 8 コア使用時に 5.36 倍の速度向上が得られた．lbm の主要なデータ構造はヒープ上に確保された 1 次元化された配列であり，Parallelizable C に書き換えを行うことにより，基準となるポインタ解析で自動並列化可能となった．

7. 関 連 研 究

本論文で Cycle-Sensitive 解析を実現するために利用した Aging[14] は，ループのイタレーションの先頭でバッファを確保し，次のイタレーションでも使い回すようなパターンに対する解析として提案された．このような例では，通常のデータフロー解析では，各イタレーションで確保されるバッファがデータフロー解析の収束演算により同一領域と見なされてしまう．そこで，Aging を適用することで，そのイタレーションで確保されたヒープ領域とそれ以前のイタレーションで確保されたオブジェクトを別のオブジェクトとして解析可能とすることを可能としている．Aging の提案においても Element-Sensitive 解析との補完関係について言及されていたが，実際に本論文で提案する Element-Sensitive ポインタ解析と Aging を併用することで高い解析精度を得ることができた．

既存の Element-Sensitive ポインタ解析として，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[22] がある．Element-Wise Points-to Set は Java の自動並列化を指向して提案されたもので，シンボリック式を用いてオブジェクトの対応関係を計算している．本論文で提案する方式では *element alias* 属性という真偽値のみで表現しているが，これだけで自動並列化のための解析を行う必要条件を満たしており，軽量で効率的な実装が可能となる．

Points-to 解析ではなく，任意のポインタ変数の間のエイリアス関係を解析するポインタ解析も提案されている．Connection 解析[23] では複数のポインタオブジェクトが指し示す領域に重なりがないかどうかを解析する．本論文で提案した *element alias* は複数のポインタオブジェクトの関係ではなく，単一オブジェクトの各要素の指し先の関係を解析するものであるが，Connection 解析と同様の概念を採用している．

また，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 解析[24] 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木構造なのか DAG なのかサイクルなのか，といったデータ構造の解析を行う．しかしながら，既存の高度な並列化あるいはデータローカリティ最適化技術[15),17),18)] は主に多次元配列とループによる処理を対象としており，再帰的なデータ構造には対応していないのが現状である．そのため，再帰的なデータ構造に対する並列化技術とあわせた解析手法の改良が今後の課題となる．

8. ま と め

本論文ではコンパイラによる自動並列化のためのポインタ解析として Element-Sensitive ポインタ解析を提案した．提案した Element-Sensitive ポインタ解析は，*element alias* という真偽値型の変数のみによる表現により，軽量で効率的な実装を可能とし，既存の Flow-Sensitive ポインタ解析を容易に拡張可能である．Element-Sensitive ポインタ解析と Cycle-Sensitive ポインタ解析の併用により，C 言語で頻出のデータ構造に対する自動並列化を可能とし，本手法が有効なアプリケーションプログラムが存在することを示した．科学技術計算およびマルチメディア処理を行う 4 つのプログラムに対して Element-Sensitive ポインタ解析を利用した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ところ，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 5.50 倍の速度向上が得られた．

本論文で提案した Element-Sensitive ポインタ解析により，既存の C プログラムの自動並列化による速度向上に加え，ユーザプログラマがプログラムを修正することによる自動並列化の適用可能性も広がる．ポインタ解析を利用したユーザプログラマへのプログラム書き

換え支援ツールの開発等により，マルチコア向けソフトウェア開発における自動並列化の利用を促進し，ソフトウェア生産性の向上が実現可能と考えられる．

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”，“メニーコア・プロセッサ技術（グリーン IT プロジェクト）の先導研究”，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. parallel and distributed systems*, Vol.9, 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) ISO/IEC 9899:1999 — Programming Language C (1999).
5) Hind, M.: Pointer Analysis: Haven’t We Solved This Problem Yet?, *PASTE’01* (2001).
6) Emami, M., Ghiya, R. and Hendren, L.J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers, *PLDI’94* (1994).
7) Wilson, R.P. and Lam, M.S.: Efficient Context-Sensitive Pointer Analysis for C Programs, *PLDI’95* (1995).
8) Kahlon, V.: Bootstrapping: A Technique for Scalable Flow and Context-Sensitive Pointer Alias Analysis, *PLDI’08* (2008).
9) Hardekopf, B. and Lin, C.: Semi-Sparse Flow-Sensitive Pointer Analysis, *POPL’09* (2009).
10) Whaley, J. and Lam, M.: Cloning-Based Context-Sensitive Pointer Alias Analyses Using Binary Decision Diagrams, *PLDI’04* (2004).
11) Nystrom, E.M., Kim, H.-S. and Hwu, W.-M.W.: Importance of heap specialization in pointer analysis, *PASTE’04* (2004).
12) Lattner, C., Lenharth, A. and Adve, V.: Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World, *PLDI’07* (2007).
13) Pearce, D.J., Kelly, P.H.J. and Hankin, C.: Efficient Field Sensitive Pointer Analysis for C, *PASTE ’04* (2004).
14) Ryoo, S., Rodrigues, C.I. and Hwu, W.-M.W.: Iteration Disambiguation for Parallelism Identification in Time-Sliced Applications, *LCPC’07* (2007).
15) Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company (1996).
16) Aho, A.V., Lam, M.S., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques and Tools*, 2nd Edition, Pearson Education, Inc. (2007).
17) Lim, A.W., Liao, S.-W. and Lam, M.S.: Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning, *PPoPP’01* (2001).
18) 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).
19) OpenMP Application Program Interface Version 2.5 (2005).
20) Optimaly Scheduled Advanced Multiprocessor Application Program Interface (OSCAR API) version 1.0. http://www.kasahara.cs.waseda.ac.jp/
21) 間瀬正啓，木村啓二，笠原博徳：マルチコアにおける Parallelizable C プログラムの自動並列化，情報処理学会研究会報告 2009-ARC-174-15 (SWoPP2009) (2009).
22) Wu, P., Feautrier, P., Padua, D. and Sura, Z.: Instance-wise Points-to Analysis for Loop-based Dependence Testing, *ICS’02* (2002).
23) Ghiya, R. and Hendren, L.J.: Connection Analysis: A Practical Interprocedural Heap Analysis for C, *LCPC’95* (1995).
24) 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).

(平成 21 年 9 月 29 日受付)
(平成 22 年 1 月　5　日採録)

間瀬　正啓（学生会員）

昭和 58 年生．平成 17 年早稲田大学理工学部電気電子情報工学科卒業．平成 19 年同大学大学院理工学研究科情報・ネットワーク専攻修士課程修了．平成 19 年同大学院基幹理工学研究科情報理工学専攻博士後期課程進学．平成 20 年早稲田大学基幹理工学部情報理工学科助手，現在に至る．

村田　雄太

昭和 59 年生．平成 19 年早稲田大学理工学部コンピュータ・ネットワーク工学科卒業．平成 21 年同大学大学院修士課程修了．平成 21 年ソニー株式会社入社，現在に至る．

木村　啓二（正会員）

昭和 47 年生．平成 8 年早稲田大学理工学部電機工学科卒業．平成 13 年同大学大学院理工学研究科電気工学専攻博士課程修了．平成 11 年早稲田大学理工学部助手．平成 16 年同大学理工学部コンピュータ・ネットワーク工学科専任講師．平成 17 年同助教授．平成 19 年同大学情報理工学科准教授，現在に至る．マルチコアプロセッサのアーキテクチャとソフトウェアに関する研究に従事．

笠原　博徳（正会員）

昭和 55 年早稲田大学理工学部電気工学科卒業，昭和 60 年同大学大学院博士課程修了，工学博士，昭和 61 年早稲田大学理工学部専任講師，平成 9 年同教授，現在情報理工学科教授，アドバンストマルチコアプロセッサ研究所長．昭和 60 年カリフォルニア大学バークレー校，平成元年〜2 年イリノイ大学 Center for Supercomputing R & D 客員研究員．昭和 62 年 IFAC World Congress Young Author Prize，平成 9 年情報処理学会坂井記念特別賞，平成 20 年 LSI オブザイヤー準グランプリ受賞．IEEE Computer Society 理事，情報処理学会 ARC 主査，論文誌 HG 主査，文部科学省地球シミュレータ中間評価委員，経済産業省/NEDO “アドバンスト並列化コンパイラ” “リアルタイム情報家電用マルチコア” 等プロジェクトリーダ歴任．