実行時依存解析に基づく半自動並列化の効率的実装

早津 政和[†] 田浦 健次朗[†] 近山 隆[††]

本稿では，実行時のメモリアクセス履歴情報からプログラムの依存関係を解析して，抽出した並列性をユーザに提示する「半自動並列化」手法とその効率的な実装方法について述べる．結果が入力に依存するためにすべての並列化候補の安全性を保証することはできないが，実際に起こった依存関係のみを用いることから，既存の静的自動並列化手法より単純で統一的な仕組みで多くの並列化可能性を提示できる．本稿では，その並列性解析のコスト（使用記憶域量，解析時間）を下げるため，実行時に解析を行うことで不要となった履歴情報を破棄する手法を提案・実装し，その評価を行う．

An Effecient Implementation of Semi-automatic Parallelization

Masakazu Hayatsu,[†] Kenjiro Taura[†] and Takashi Chikayama[††]

In this presentation, we describes an effecient implementation of “semi-automatic parallelization”. This approach can extract interprocedural dependency precisely by analyzing access traces gathered at trial run time. This approach cannot guarantee proper parallelization at any execution, but it can avoid extract dependences that never occur in parallel execution. In our previous implementation, it is of a probrem that dependency analysis needs large memory and long execution time. So, in this presentation, we argue an approach to cut down the analysis cost by discarding unnecessary access traces during execution.

1. はじめに

1 台に多数の CPU を搭載した並列計算機は，非常に古くから研究されており，今も大規模化と幅広い普及が続いている．現在では大規模な科学技術計算だけでなく他の幅広い分野からも，並列処理に対する需要は多くなってきており，並列プログラムの開発の効率性は今後さらに重要になっていくと考えられる．

しかし，並列プログラムは記述・デバッグが非常に難しく，人手のみで適切に並列化するためには非常に大きなコスト（手間）がかかる．この問題を解決するため，逐次プログラムを自動的に並列プログラムに変換する自動並列化の研究が長年行われてきたが，それらの方法の多くはループや配列を単純に用いたプログラムにしか適用できないために，一般のプログラムに対しては高いパフォーマンスを得ることが難しい．

そこで，我々は，実行時のアクセス履歴情報を用いて並列性を抽出する手法を提案し，実装を行った[1)]．この手法では，まずプログラムを逐次実行し，そのときのメモリアクセスの履歴情報を収集する．そしてその履歴から依存関係にある（並列実行できない）部分を見つけ，それ以外を「並列化可能候補」としてプログラマに提示する．

この手法は実行時に実際に起こった依存関係を用いるため，複雑な静的解析を必要とせず，静的自動並列化手法よりも多くの並列化可能性を提示できる．ただし，解析結果がプログラムの入力に依存するため，つねに正しい解を保証するものではない．したがって，並列化の可否の判断材料を提示し，プログラマが判断を行う，という「半自動並列化」として用いるのに適している．

Scheme[2)] を対象言語とした過去の我々の実装[1)] では，古典的なコンパイル時自動並列化では適用が難しいとされるプログラムの並列性の抽出に成功している．一方，この履歴解析にかかるコスト（使用記憶領域量，解析時間）の高さが課題としてあがっていた．

そこで本研究では，実行中に依存解析を行い，不要となった情報を破棄することにより，履歴解析時のコストを削減する手法を提案する．そして既存の Scheme 処理系である Chicken コンパイラ[3)] 上に実装を行い，実用的なレベルでの性能の評価を目指す．

† 東京大学情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 東京大学新領域創成科学研究科
School of Frontier Sciences, The University of Tokyo

本稿の構成は以下のとおりである．2 章では関連研究を紹介する．次に，3 章において本研究で用いる実行時依存解析に基づく並列性抽出手法について説明する．そして 4 章において提案する履歴解析コストの削減手法について述べ，5 章において実装・実験を行った結果を示し，解析コストの評価を行う．最後に 6 章においてまとめと今後の課題について述べる．

2. 関 連 研 究

2.1 並列性の抽出

並列性の抽出は，逐次プログラムでの本質的でない記述上の順序関係を取り除き，元の意味を保つために必要最小限の順序の制約を求めることによって達成される．これらの依存関係にないタスクは，互いの結果に影響しない．すなわち，並列に実行しても逐次実行と同じ意味を保つことができる，ということである．

本研究では，あるタスクで代入されたデータを，他のタスクで代入/参照する場合の順序制約（データ依存）に着目して依存性解析を行う．

2.2 競合検出手法

プログラム内のデータ依存性を見落とすと，並列実行時にアクセスが競合してしまい，データの代入/参照のタイミングによって実行結果が逐次での結果と異なったり，実行が非決定的，つまり実行時ごとに（同じ入力であっても）異なる結果を返したりしてしまう．

そこで，並列プログラムのデバッグの手法として，race-detection [4),5)] が研究されている．これはソースコード解析，実行時のデータアクセス解析，すべての状態遷移を調べあげるなどの方法によってアクセス競合を検出する手法である．

この手法はプログラマがあらかじめ指示した並列化が正しいか（競合を起こさないか）を調べるにはとても強力であるが，競合を見つけた場合にそれを指摘するだけにとどまっている．我々の方法は，それをさらに進め，競合を起こさない部分を見つけ出し，プログラマに並列化のヒントとして提示することができる．

2.3 自動並列化手法

自動並列化手法には，大きく分けて静的手法，動的手法の 2 つがある．静的手法とは，コンパイル時にソースプログラムを解析し，並列化する手法であり，配列のインデックスを利用したループ並列化手法[6)] や，ポインタを利用した分割統治の並列化手法[7)]，再帰的手続きの並列化手法[8)] などがある．ただし，それらの多くは対象とする特定の構造で書かれたプログラムにしか適用できず，その書式にも制限がある．また，コンパイル時に依存関係を決められない場合も多く，並列化の機会を逃してしまい，結果として高いパフォーマンスを得ることが難しい．動的手法とは，実行履歴情報を解析し，並列化する手法であり，inspector/executor を用いたループ並列化手法[9)] が多く研究されている．ただし，事前に監視すべき場所を確定できることが前提となるため，深さの未知な再帰，未知な制御の流れ，動的なデータ割当て，などを持った一般のプログラムに対しては適用が難しい．

本研究の手法は動的手法を採用しているため，実際に起きた依存関係を検出することで，静的手法では解析できなかった並列性も抽出することができる．さらに，プログラム構造を仮定しない実行スタック解析・メモリが静的/動的に確保されたかどうかに依存しないアクセス検知によって，既存の自動並列化手法よりも広範囲のプログラムに適用できるという利点がある．

ただし，半自動並列化手法は自動並列化手法の代わりになる，というものではない．それは，半自動並列化手法が並列化の正当性を保証するものではなく，あくまでプログラマに「並列化できる可能性がある」という証拠を提供するものだからである．

2.4 対話的並列化支援技術

本研究の立場と同じく，プログラマと対話的に並列化を進めていく技術も研究されている．それらの手法として，SUIF Explorer System [10)] などがあげられる．

SUIF Explorer System ではまず，逐次プログラムを SUIF Compiler [11)] で並列化した後，依存情報や実行プロファイルを収集する．そしてそれらを解析し，“coverage” と “granularity” の 2 つの指標を用いてループを評価し，プログラマに提示する．プログラマはループの評価と依存情報を基に並列化し，再評価を行う，というプロセスを繰り返していく．

この技術によって，プログラマが扱う情報量は大幅に削減され，並列化プロセスの効率を上げることができる．ただし，並列化の対象がループのみである点が，プログラム構造によらず並列性を抽出できる我々の手法との大きな違いである．

3. 実行時依存解析に基づく並列性抽出手法

3.1 本手法の概要

本研究で用いる「実行時依存解析に基づく並列性抽出手法」では，実行時のアクセス履歴を解析し，並列化不可能な部分を見つけ，それ以外を並列化可能候補としてプログラマに提示する（**図 1**）．プログラムの自動的な依存解析にプログラマの判断・知識を加えた「半自動並列化手法」といえる．

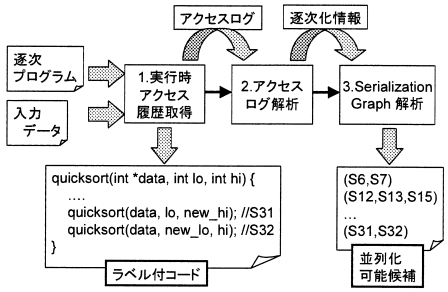

図 1 動作の概要
Fig. 1 An overview of our system.

この手法は，2.3 節でも述べたように，既存の自動並列化手法よりも広い範囲のプログラムに対し適用が可能である．解析がプログラムの入力に依存するため，得られたすべての並列性が安全であると保証することはできないが，すべての入力に対し安全な並列性があれば，必ず見つけることができる．そこでこの手法は，

- 代表の入力をいくつかで解析を行い，結果をマージすることにより，より正確な並列化を行う，
- プログラマや性能計測ツールにヒントを与えて，並列化の判断をしてもらう，

というように用いることが適当である．また，本手法で抽出できる並列化可能候補は数多いが，パフォーマンスなどの影響で実際に採用されるものは一般にその中の少数でしかない．よって，候補すべてに対してではなく，採用したもののみに対して安全性を確かめれば，効率的に並列プログラムの開発を進めることができる．

3.2 半自動並列化の方針

本手法では以下のような方針で並列性の抽出を行う．

- 同じ呼び出しレベルにある式間の並列性を調べる．
- データ依存の原因となる呼び出し元の動作を逐次化し，それ以外は並列化する．
- ループを末尾再帰と見なし，ループの並列化を式の並列化に帰着させる．

図 2 左のプログラムを例に並列化のプロセスを見てみる．なお，現在の実装は Scheme を対象に行っているが，説明のためこの例では C 言語で記述した．まず図 2 左のループを図 2 右のように末尾再帰と考える．そして同じ呼び出しレベルにある `foo(i)` と `loop(i+1)` の間の依存関係を調べる．もしも依存関係が見つからなかった場合には，この 2 つの式は並列に実行できる，すなわち元のループの各イテレーションは並列実行可能であると考えることができる．

この依存関係を求めるためには，データへのアクセスを監視する必要がある．変数それ自体への代入/参照とともに，構造を持つデータの場合はその個々のメンバについても監視を行う．Scheme においては配列（vector）の各要素，cons セル（pair）の car/cdr 部，文字列の各文字などがこれにあたる．

```
for(i=0; i<N; i++) {
    foo(i);
}
```

```
loop(int i) {
  if (i<N) {
    foo(i);
    loop(i+1);
  }
} loop(0);
```

図 2 ループの末尾再帰への変形
Fig. 2 Conversion of loop into tail recursion.

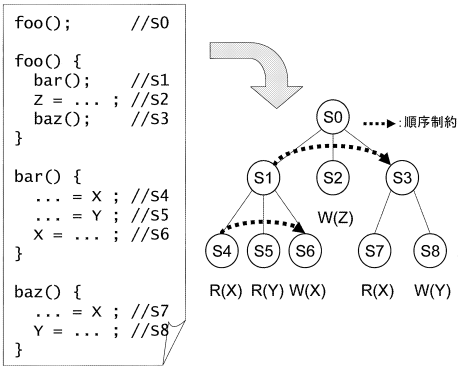

図 3 実行履歴木の例
Fig. 3 An example of access traces.

本手法は，プログラム中の式の間に競合関係があるかを検出するものである．よって，一部の変数に対してロック処理やリネーム・プライベート化を行えば並列実行が可能となるものに関しては本稿の範囲ではない．それらの場合に対応できるようなアルゴリズムへの拡張は今後の課題である．

3.3 アルゴリズム

本手法のアルゴリズムは以下のようになっている．

(1) アクセス履歴取得
  (a) プログラムをコマンド（コードの最小単位）に分解し，各々にラベルをつける（**図 3** 左）．
  (b) プログラムを実行する．実行中，(a) のラベルを用いて実行スタックの状態を管理する．
  (c) オブジェクト（メモリ領域）が確保されたら，各オブジェクトに LocationID を付ける．
  (d) メモリアクセスを検出したら，
    - その時点の実行スタック
    - アクセス動作（READ/WRITE）
    - アクセスしたオブジェクトの LocationID

  を記録する（図 3 右）．
(2) アクセス履歴解析
  (a) 同じ LocationID に対するアクセス履歴から，依存関係にある（逐次化すべき）コマンドを抽出

する（図 3 右の矢印）．

(b) 依存関係が検出されなかったコマンドの組を並列化可能候補としてプログラマに提示する．

3.4 過去の実装

過去の実装では，実際にこの手法で RayTrace や，コンパイル時自動並列化手法では並列化が難しい N-Queen 問題，Quicksort（再帰呼び出しの並列化）などのプログラムの並列性を抽出することに成功した[1)]．これまで提案されてきた手法は，これらのプログラム各々を原理的に並列化できる枠組みであったが，この方法は，これらすべてを統一的かつ単純な仕組みで並列化できる枠組みとなっている．

しかし，この実装ではアクセス履歴をログとしてファイルに書き出し，実行終了後に解析するという設計になっていたため，プログラムの実行時間に比例した量の記憶域が必要になってしまっていた．典型的な場合として，同じ場所へ多数回のアクセスが起きても，それら 1 つ 1 つのアクセスが必ずディスク上である一定の領域を消費する．このため，小規模な実行しか扱えないという問題があった．

4. 履歴解析コストの削減

実行履歴に基づく半自動並列化のアルゴリズムは，3.3 節に示したように履歴取得と履歴解析のフェーズに分かれている．しかし，単純にログを用いて依存性解析を行う設計では，3.4 節で指摘したように実行時間に比例した記憶域が必要となり，大きく複雑なプログラムには適用が難しい．

そこで，依存解析を実行時の履歴取得の際に行い，そこで不要となった情報を破棄するようにアルゴリズムを変更する．以下でその方法について詳しく述べる．

4.1 解析済履歴情報の削減

記憶領域量削減のため，依存性解析が終わったアクセス情報のうち，以後の依存性解析に関係のない不要な情報を破棄していく．

この破棄の仕組みを，具体的に図 3 のプログラムを例に説明する．プログラム中の式（コマンド）をそれぞれラベルで表現すると，S1 の関数の実行が終わった直後の実行履歴（呼び出し構造を表す木構造）は**図 4** 左のようになる．ここで，`W(X)`，`R(X)` はそれぞれ X への WRITE アクセス，READ アクセスを示している．

履歴情報削減の基本的な考えは，部分木の解析が終わってしまえば，

- 部分木内のどのコマンドでアクセスがあったかを区別する必要はない，
- 部分木内で同じオブジェクトへ複数回アクセスがあっても，それらすべてを記録しておく必要はない，

ということである．

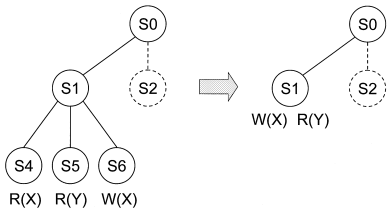

図 4 解析済履歴情報の削減
Fig. 4 Compaction of access traces.

まず前者は，S1 を根とする部分木（S1 部分木）のどこでメモリアクセスがあったか（ここでは S3，S4，S5），という詳細情報は必要でなくなることを示している．たとえば S2 部分木に実行が移って X へのアクセスが起こった場合，導かれる依存関係は実行履歴上の最初の分岐点であるコマンド間の S1 $\Longrightarrow$ S2 であるため，「S0 → S1 → S5 `W(X)`」ではなく，「S0 → S1 `W(X)`」という情報だけ残せばよい**［詳細情報の削除］**．

次に後者は同じ X へのアクセス履歴は 1 つにまとめられることを示している．たとえば，S2 部分木で `W(X)` があった場合には，S1 下の `W(X)`，`R(X)` どちらとの間も依存関係があるため，`R(X)` があるかにかかわらず S1 $\Longrightarrow$ S2 の依存関係を導くことができる．また，S2 部分木の下で `R(X)` があった場合には，S1 下の `W(X)` のみが依存関係にあることから，S1 $\Longrightarrow$ S2 を導く際には `R(X)` は関与しない．このように，複数の X へのアクセス履歴がある場合，

- `W(X)` がある場合 → `W(X)`
- `R(X)` しかない場合 → `R(X)`

の履歴を 1 つだけ残せばよい**［重複情報の削除］**．

よって，最終的には，S1 部分木の終了時に `W(X)`，`R(Y)` のアクセス履歴以外は削減できる（図 4 右）．

4.2 解析アルゴリズム

解析済履歴情報の削減を取り入れたアルゴリズムについて述べる．3.3 節に示したアルゴリズムからの大きな変更点は以下のとおりである．

- 実行スタックの代わりに実行履歴木を管理する．
- memory access 時に依存解析を行う．
- pop 時に不要な履歴情報を破棄する．

以下に，解析のアルゴリズムを push，pop，memory access 時の動作に分けて記述する．ここで，呼び出しコマンドは Node，アクセス情報は Leaf というデータ構造で表現し，実行履歴木はそれらのリンクで表現することにする．また，監視されるオブジェクト

```
push(label) {
  new_node = create_node(label);
  current_node->add_child(new_node);
  current_node = new_node;
}
```

図 5 push 時の擬似コード
Fig. 5 Pseudo code for push.

には一意の LocationID と，それらにアクセスした履歴が保持されているとする．

4.2.1 push 時の動作

図 5 に動作の擬似コードを示す．

コマンドの呼び出し情報を Node に格納し，履歴木への追加を行う．

4.2.2 pop 時の動作

図 6 に動作の擬似コードを示す．

部分木内にアクセス情報がない場合は，現在の Node を削除して履歴木を 1 段登る（通常の pop 動作）．

部分木内にアクセス情報がある場合は，その統合・破棄を行う．動作の詳細は，以下のようになっている．

(1) 現在の Node とすべての子 Node のアクセス情報をマージする．同じオブジェクトへのアクセスが複数ある場合，
- WRITE アクセスがある →WRITE 情報
- WRITE アクセスがない →READ 情報

を 1 つだけ残し，それ以外のアクセス情報を破棄する（重複情報の削除）．

(2) 子 Node をすべて削除する（詳細情報の削除）．

(3) 現在の Node を削除せずに，履歴木を 1 段登る．

4.2.3 memory access 時の動作

図 7 に動作の擬似コードを示す．

アクセス情報を実行履歴木とオブジェクトのアクセス履歴に記録し，依存関係を解析する．

動作の詳細は，以下のようになっている．

(1) アクセス情報の格納

アクセス動作，オブジェクトの LocationID を格納した新しい Leaf を作成し，現在の Node につなぐ．

(2) 依存性解析

Leaf のアクセス動作によって以下のように振る舞う．
- READ アクセスの場合

  オブジェクトの直前の WRITE 履歴に対し次の依存コマンド解析を行う．
- WRITE アクセスの場合

  オブジェクトの直前の WRITE 履歴とそれ以降のすべての READ 履歴に対し次の依存コマンド解析を行う．

(3) 依存コマンド解析

同じ親を持つ Node にたどり着くまで，履歴木を登り，2 つの Node に格納されているラベル（逐次化すべきコマンド）を記録する．

(4) オブジェクトのアクセス履歴への登録

```
pop() {
  if (there is any access in current partial tree) {
    current_node->merge_child_accesses();
    current_node->delete_child_nodes();
    current_node = current_node->parent;
  }
  else { /* mere pop */
    parent_node = current_node->parent;
    delete_node(current_node);
    current_node = parent_node;
  }
}
merge_child_accesses() {
  for (each child in my->children) {
    my_leaf = my->accesses;
    ch_leaf = child->access;
    work = NULL;
    while (my_leaf != NULL && ch_leaf != NULL) {
      diff = my_leaf->locationID - ch_leaf->locationID;
      if (diff == 0) { /*duplicate information!!*/
        if (my_leaf->action == WRITE)
          ch_leaf = ch_leaf->remove();
        else  my_leaf = my_leaf->remove();
      }
      else if (diff < 0) {
        work->add(my_leaf); my_leaf = my_leaf->next;
      }
      else {
        work->add(my_leaf); ch_leaf = ch_leaf->next;
      }
    }
    if (my_leaf != NULL) work->append(my_leaf);
    else                 work->append(ch_leaf);
    my->accesses = work;
  }
}
```

図 6 pop 時の擬似コード
Fig. 6 Pseudo code for pop.

```
trap_access(action, object) {
  new_leaf = create_leaf(action, object->locationID);
  current_node->add_access(new_leaf);
  analyze_dependency(new_leaf,object->access_history);
}
analyze_dependency(leaf, history) {
  last_write = history->last_write;
  if (last_write != NULL)
    analyze_depend_command(last_write, leaf);
  if (leaf->action == READ)
    history->add_read_list(leaf);
  else { /*(leaf->action == WRITE)*/
    for (each read_ent in history->read_list)
      analyze_depend_command(read_ent, leaf);
    history->delete_all_entry();
    history->last_write = leaf;
  }
}
analyze_depend_command(src_leaf, dst_leaf) {
  src_node = src_leaf->parent;
  dst_node = dst_leaf->parent;
  diff = src_node->depth - dst_node->depth;
  if (diff > 0)      /* climb tree to same depth */
    while (diff--) src_node = src_node->parent;
  else if (diff < 0) /* climb tree to same depth */
    while (diff++) dst_node = dst_node->parent;
  while (have_different_parent(src_node, dst_node)) {
    src_node = src_node->parent;
    dst_node = dst_node->parent;
  }
  regist_dependency(src_node->label,dst_node->label);
}
```

図 7 memory access 時の擬似コード
Fig. 7 Pseudo code for memory access.

Leaf のアクセス動作によって以下のように振る舞う．

- READ アクセスの場合
  READ 履歴に新しい Leaf を追加する．
- WRITE アクセスの場合
  過去の WRITE 履歴，READ 履歴をすべて削除し，WRITE 履歴に新しい Leaf を設定する．

5. 実装と評価

5.1 実　　装

本研究では，Lisp の方言である Scheme[2] を対象言語とし，Scheme から C へのコンパイルを行う Chicken[3] コンパイラを元に半自動並列化手法を実装した．

以前の実装時と同様に，N-Queen と Quicksort の並列性を抽出することに成功した．以下では，N-Queen と Quicksort を問題サイズを変えたときの解析のコスト（使用記憶領域量，実行時間）計測の結果について見ていく．なお，今回の実験では，問題サイズを変えても得られる結果は同じ，すなわちより小さい問題サイズで正しく依存関係を導き出すことができていた．

実験に用いた Chicken コンパイラのバージョン 1.22，C コンパイラは gcc 3.3.3（cygwin special）である．クロック 2.00 GHz の AthlonMP プロセッサを搭載したパーソナルコンピュータ上で計測を行った．主記憶容量は 1 GB，OS は WindowsXP である．Chicken でのコンパイル時にはオプションは指定せず，gcc でのコンパイル時には下記のオプションを指定した．

-O3 -fomit-frame-pointer -fstrict-aliasing
-mcpu=i586 -mpreferred-stack-boundary=2

5.2 使用記憶領域量

不要情報削減を取り入れた各段階で，依存解析に使用される記憶領域量を計測した．これにより，記憶領域量の削減の効果が分かる．

N-Queen での結果を**図 8** に，Quicksort での結果を**図 9** に示した．縦軸は，記憶領域量を格納する実行スタック情報（木構造の Node）・アクセス情報（木構造の Leaf）の数で換算してある．ここで実際のメモリ使用量でなく Node・Leaf の総数で比較しているのは，Node・Leaf の格納に必要なメモリ量が実装によって異なるためである．データ系列は，上から順に，

- 履歴情報をログとして出力した場合
- 履歴情報を木構造で保持した場合
- 履歴情報を木構造で保持し，不要情報を削減を行った場合

となっている．

履歴木の構造を取り入れた場合には，ログ形式で保持する場合に比べ，使用記憶領域量を N-Queen では 1/2〜1/25，Quick Sort では 1/5〜1/40 程度に減少させることができた．これは，ログ形式では実行スタックの情報を重複して保持していた部分が木構造ではなくなったためである．

さらに，実行時に解析済み情報を削除する場合は，削除しない場合に比べ，使用記憶領域量を N-Queen では 1/4〜1/40 ，Quick Sort では 1/8〜1/100 程度に減少させることができた．

また，データ量（履歴木の大きさ）が大きくなるほど，これらの削減の効果が大きくなっていることが確

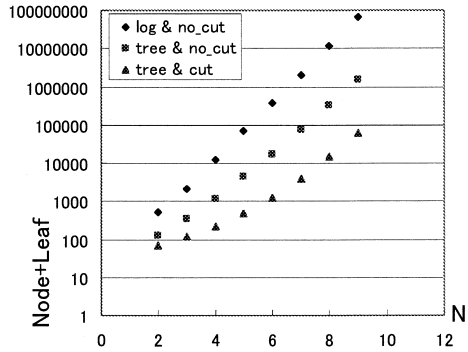

図 8 N に対する解析時使用記憶量（N-Queen）
Fig. 8 Memory size used at analysis vs. N (N-Queen).

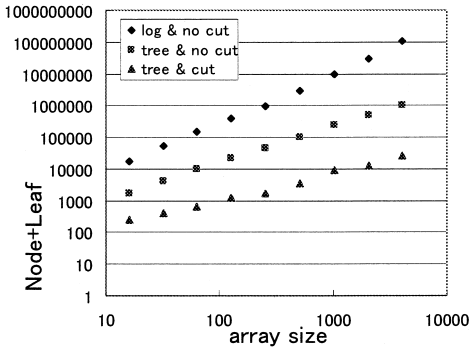

図 9 配列要素数に対する解析時使用記憶量（Quicksort）
Fig. 9 Memory size used at analysis vs. array size (Quicksort).

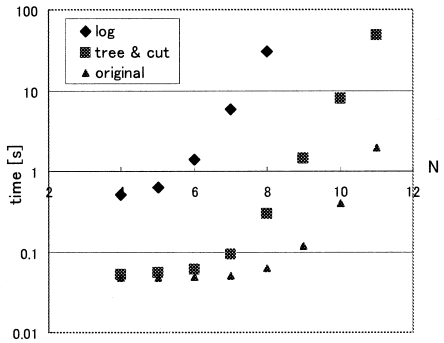

図 10 N に対する実行時間（N-Queen）
Fig. 10 Execution time vs. N (N-Queen).

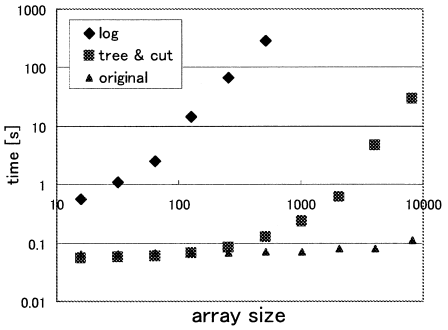

図 11 配列要素数に対する実行時間（Quicksort）
Fig. 11 Execution time vs. array size (Quicksort).

かめられた．

5.3 実行時間

依存性解析を導入する前とした後の実行時間を計測した．これにより実行時にアクセス履歴解析を行うことによる時間的コスト削減の効果と，単純実行に対するアクセス履歴解析の時間的オーバヘッドを見ることができる．

N-Queen に対する結果を**図 10** に，Quick Sort に対する結果を**図 11** に示した．縦軸は，総実行時間（履歴の解析をする場合はその時間も含む）を示している．データ系列は，上から

- 履歴をログとして出力後に解析を行う場合
- 実行時に履歴取得と依存性解析を行う場合
- 単純に実行のみを行う場合

となっている．

実行後に履歴解析を行う場合に比べ，実行時に履歴解析を行う場合には，総実行時間を N-Queen では 1/10〜1/100，Quick Sort では 1/10〜1/2000 程度に減少させることができた．これは，前者の履歴をログとして書き出し，それを再び読み込むという動作よりも小さなオーバヘッドで後者の不要情報削減の動作を行うことができていることを示している．

一方で，単純に実行する場合に比べ，実行時に履歴解析を行う場合には，N-Queen，Quicksort ともに問題サイズが大きくなるにつれ実行時間の増加の割合が高くなってしまった．これは，問題サイズの増加とともに，作られるオブジェクトの種類が増え，木の深さも深くなるために，履歴木がどんどん大きくなり，情報の削減や依存解析時の探索などにより多くの時間がかかってしまうようになったのが理由だと考えられる．

6. おわりに

本稿では，まず実行時履歴解析に基づく半自動並列化手法について述べた．さらに，記憶領域量を削減する手法を提案・実装し，実験結果を示した．過去の実装に比べ，使用記憶領域量を 1/100〜1/4000，総実行時間を 1/10〜1/2000 に削減することができた．

今後の課題としては，履歴解析速度の向上に加え，末尾再帰のサポートがあげられる．Scheme の実装は正しく末尾再帰を行う（末尾再帰時にはスタックを消費せず，呼び出しの段数には制限がない）ことが要求されている[2]．しかし，現在の実装で管理している履

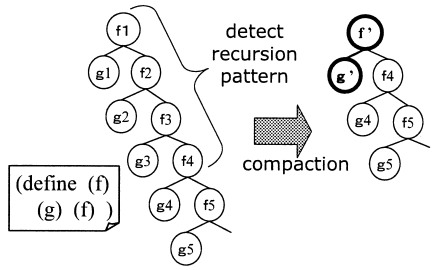

図 12 末尾再帰時のスタック圧縮
Fig. 12 Compaction at tail recursion.

歴木はこれをサポートしておらず，非常に深い末尾再帰を実行できない．そこで，コンパイル時の解析により末尾再帰を検出し記録しておき，実行時に**図 12** 左のように再帰のパターンを見つけたら図 12 右のようにスタックを潰すという動作が必要となる．解析の正しさを保証しつつ小さなコストでこの動作を行う方法について今後考案，実装していく予定である．

参 考 文 献

1) Nguyen, H.V., Taura, K. and Yonezawa, A.: Parallelizing Programs Using Access Traces, *6th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (2002).
2) scheme.org, http://www.scheme.org/
3) The CHICKEN Scheme Compiler. http://www.call-with-current-continuation.org/
4) Emrath, P.A. and Padua, D.A.: Automatic Detection of Nondeterminacy in Parallel Programs, *Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, United States, pp.89–99 (1988).
5) O'Callahan, R. and Choi, J.-D.: Hybrid Dynamic Data Race Detection, *9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, Calif., pp.167–178, ACM Press (2003).
6) Burke, M., Cytron, R., J.F.W.H., et al.: Automatic Discovery of Parallelism: A Tool and an Experiment, *ACM SIGPLAN Conference on Parallel Programming: Experiences with Applications, Languages and Systems*, Vol.23, New Haven, Connecticut, pp.77–84 (1988).
7) Rugina, R. and Rinard, M.C.: Automatic Parallelization of Divide and Conquer Algorithms, *Principles and Practice of Parallel Programming*, pp.72–83 (1999).
8) Gupta, M., Mukhopadhyay, S. and Sinha, N.: Automatic Parallelization of Recursive Procedures, *International Journal of Parallel Programming*, Vol.28, No.6, pp.537–562 (2000).
9) Chen, D.-K., Torrellas, J. and Yew, P.-C.: An efficient algorithm for the run-time parallelization of DOACROSS loops, *Supercomputing*, pp.518–527 (1994).
10) Liao, S.-W., Diwan, A., Jr., R.P.B., Ghuloum, A.M. and Lam, M.S.: SUIF Explorer: An Interactive and Interprocedural Parallelizer, *Principles and Practice of Parallel Programming*, pp.37–48 (1999).
11) Wilson, R., French, R., Wilson, C., et al.: An Overview of the SUIF Compiler System, Technical report, Computer System Lab, Stanford University (1993).

(平成 16 年 7 月 2 日受付)
(平成 16 年 10 月 15 日採録)

早津 政和（学生会員）
1980 年生．2003 年東京大学工学部電子情報学科卒業．同年 4 月より東京大学大学院情報理工学系研究科電子情報学専攻修士課程在学中．

田浦健次朗（正会員）
1969 年生．1997 年東京大学大学院理学博士（情報科学専攻）．1996 年より東京大学大学院理学系研究科情報科学専攻助手．2001 年より東京大学大学院情報理工学系研究科電子情報学専攻講師．2002 年より同助教授．

近山 隆（正会員）
1953 年生．1977 年東京大学工学部計数工学科卒業．1982 年東京大学大学院情報工学専門課程修了，工学博士．同年より ICOT において第五世代コンピュータプロジェクトに参加．1995 年より東京大学に移り，現在同新領域創成科学研究科基盤情報学専攻教授．