成蹊大学工学研究報告
J. Fac. Eng., Seikei Univ.
Vol.40 No.2 (2003) pp.25-33

動的メモリ割当てを含むCプログラムの自動並列性解析

斉藤義功*[1]，美濃本一浩*[2]，甲斐宗徳*[3]

Automatic parallelism analysis of C programs with dynamic memory allocations

Yoshinori SAITO*[1], Kazuhiro MINOMOTO*[2], Munenori KAI*[3]

ABSTRACT: For a last decade, many commercial multiprocessor systems have been produced. However, it seems that there are not any practical and efficient parallelizing C compilers, for general sequential C programs, provided with the systems. Analyzing parallelism of C source codes is one of the very important keys for automatic parallelizing compilers. In this paper, we propose and implement an analysis method of parallelism of C source codes including some dynamic memory allocations as well as pointers. The proposed analyzer is able to analyze each statements in source codes and extract more parallelism from some sequential C programs and some well-known benchmark programs in C.

KEYWORDS: Parallel Processing, Parallelism Analysis, Source Code Analysis, Dynamic Memory Allocation

(Received June 20, 2003)

1．はじめに

1.1 背　景

コンピュータの性能向上の手段としてマルチプロセッサシステムが有効であることは，近年のスーパーコンピュータの多くに利用され，高い性能を実現している[1)]ことからも明らかである。そして，プロセッサ単独での速度向上も限界に近づきつつあることを考えると，マルチプロセッサシステムは今後もよりいっそう重要な基盤技術になっていくであろう。

しかし，マルチプロセッサシステムには，並列性と効率を考慮したプログラムでないとその優れた性能を十分に引き出せないという側面もある。故に，プログラムを並列処理向きに作成する必要があり，それには並列性の抽出作業や，処理すべきタスクの実行順序を決めるためのスケジューリング作業などを行わなければならない。ところが，その作業には相当高度な知識が要求されるため，マルチプロセッサシステムの有効利用という点で大きな障害になっている。

1.2 研究目的

上記のような問題点を解決するための手段として，自動並列化コンパイラの利用が挙げられる。しかし，マルチプロセッサシステムの性能を十分に発揮させることができず，プログラムの実効性能をかえって低下させてしまう傾向を示している[2)]，というのが現状である。そこで，プログラム内にある単純なループの並列性のみを抽出するような単一粒度を対象とした並列化を行うのではなく，関数や関数を含んだループ及びループ間などの粗粒度の並列性や，細粒度であるステートメントレベルの並列性を引き出すことも，今後の自動並列化コンパイラには望まれている[3)]。

本研究では，実用的な自動並列化コンパイラが存在していないC言語に対してそのような複数粒度の並列化を行うことで，プログラムの実効性能の向上を図る自動並列化Cコンパイラの完成を最終目的とする。

*[1] 情報処理専攻大学院生[現：ソニー㈱]
*[2] 情報処理専攻大学院生
*[3] 情報処理専攻助教授(kai@is.seikei.ac.jp)
Associate Professor, Dept. of Information Sciences

2．自動並列化Cコンパイラ

自動並列化Cコンパイラとは，C言語で記述されたソースプログラム内に存在する並列性を抽出し，それを活用することで自動的に並列実行コードを生成するものである。本研究で開発する自動並列化Cコンパイラの全体的な処理の流れを図2.1に示す。

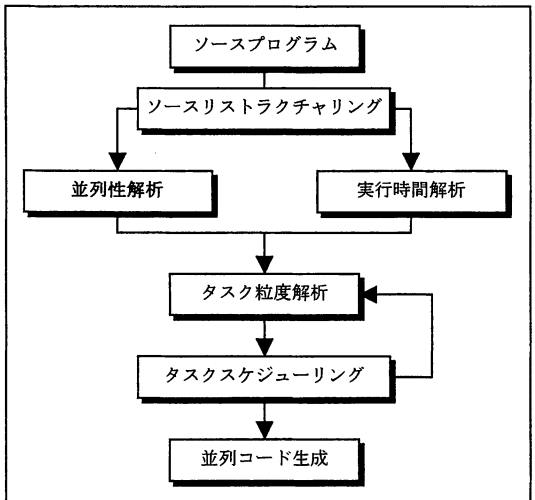

図2.1: 自動並列化Cコンパイラの処理の流れ

初期段階の作業として，解析対象となるソースプログラムの実行時間解析と並列性解析を行う。実際に解析を行うソースプログラムは，ソースリストラクチャリングを施すことで，最適化が行われている。実行時間解析では，ステートメントごとの実行時間をアセンブリレベルで求める。並列性解析では，プログラム内に存在するステートメントレベルでの並列性を抽出する。

次の段階の作業として，タスク粒度解析とタスクスケジューリングを行う。タスク粒度解析では，実行時間解析と並列性解析から得られた結果に基づき，各タスクをどのような大きさでプロセッサに割り当てれば効率のよい並列処理を行えるのかを考慮してタスク粒度を決める。タスクスケジューリングでは，それら処理すべきタスク集合をどのように各プロセッサに割り当て，どのような順序で実行すればよいのかを決める。

最終段階の作業として，ターゲットマシンにおいて優れた実効性能を実現するような並列実行コードの自動生成を行う。

以上の段階ごとの作業は分担して行われている。本研究では，初期段階の作業で，全体を通して最も重要であると考えられる並列性解析に重点を置く。

3．並列性解析

一般的に，逐次プログラムには実行結果に影響を与えない順序関係を持つ部分が存在する可能性がある。例えば，以下の断片的なコードを考える。

```
S1: a = b + c ;
S2: d = e + f ;
S3: g = a + d ;
```

上記のステートメントS1とS2の実行順序が入れ替わったとしても，S3の実行結果が変わることはない。この時，S1とS2には並列性がある，と判断できる。並列性解析では，このような並列性を抽出することが主要目的となる。

3.1 Non-Pointer解析

基本的に，並列性の抽出を実現するためには，以下で説明する依存解析を行うことで，プログラム内に存在している依存関係を検出する必要がある。そして，依存関係が検出された場合，そこに並列性はないと判断する。

3.1.1 データ依存解析[4)]

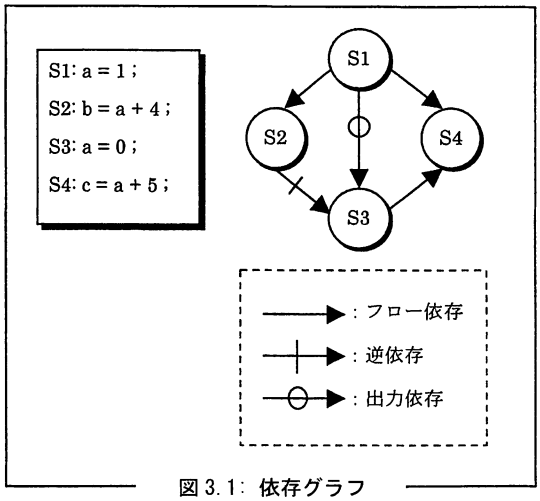

図3.1: 依存グラフ

①：フロー依存

ステートメントS1の出力変数が，ステートメントS2で利用される入力変数である場合，S2はS1にフロー依存していると定義する。

②：逆依存

S1の後にS2が続く場合，S2の出力変数がS1の入力変数と同じならば，S2はS1に逆依存していると定義する。

③：出力依存

S1の後にS2が続く場合，S2の出力変数がS1の出力変

数と同じならば，S2 は S1 に出力依存していると定義する。

データ依存の例を図 3.1 に示す。

3.1.2 制御依存解析[5]

制御依存は分岐命令を伴うプログラム内に存在する可能性がある。例えば，以下の断片的なコードを考える。

```
if ( p1 ) {
        S1 ;
}
```

上記で，ステートメント S1 は仮に条件式 p1 とのデータ依存性がないとしても，p1 を評価した後に実行する必要がある。この時，S1 は p1 に制御依存していると定義する。

3.2 Pointer 解析[6),7)]

プログラム内に存在しているポインタに関する並列性を抽出することが目的である。

3.2.1 Intraprocedural 解析

関数の内部に焦点を当てた Pointer 解析を行う。具体的には，以下で説明する Points-to 解析と Read/Write 解析を行うことでそれを実現する。

[1]. Points-to 解析

Pointer 解析を行うためには，ポインタが何を指しているのかを知る必要がある。そこで，そのような情報(Points-to セット)を集めていく Points-to 解析を最初に行う。

Points-to セットは以下のように定義することができる。

定義 3.1

ポインタ変数 x が，確実に(definitely)任意の場所 y を指しているならば，**x definitely-points-to y** という。このことを，**(x, y, D)** と表す。

定義 3.2

ポインタ変数 x が，もしかすると(possibly)任意の場所 y を指すかもしれないならば，**x possibly-points-to y** という。このことを，**(x, y, P)** と表す。

定義 3.1 と定義 3.2 の例を，それぞれ図 3.2 と図 3.3 に示す。

また，definitely-points-to と possibly-points-to の 2 つ

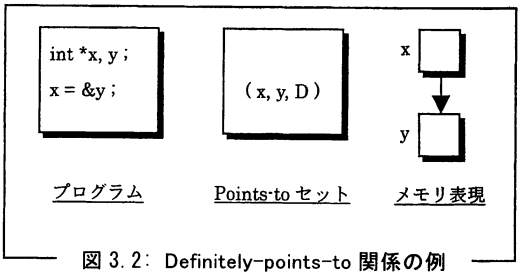

図 3.2: Definitely-points-to 関係の例

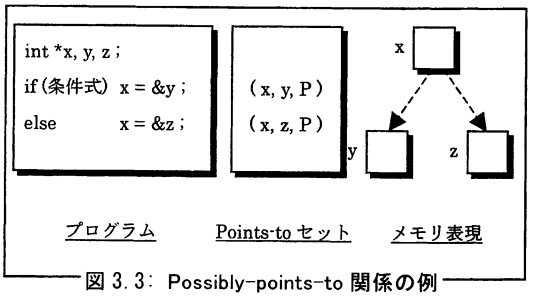

図 3.3: Possibly-points-to 関係の例

の関係を統合(merge)する場合は，以下の表 3.1 を使用して統合を行う。この統合規則は，Points-to セット(x, y, rel)において source である x と destination である y が同じ場合の Points-to セットに対して適用される。

表 3.1 を利用して統合する必要がある例を図 3.4 に示す。この例では，各ブロックを別々に Points-to 解析した後に統合を行うことで，この if 文全体としての Points-to セットを獲得している。

表 3.1: 統合規則

∽ : 2 つの関係を統合することを示す

∽	D	P
D	D	P
P	P	P

図3.4の中で使用された記号の意味は以下の通りである。

- In : 各ブロックで Points-to 解析を行う前の Points-to セット
- Out1 : if ブロックを抜けた時点での Points-to セット
- Out2 : else ブロックを抜けた時点での Points-to セット
- Out : if-else 文が最終的に保有する Points-to セット

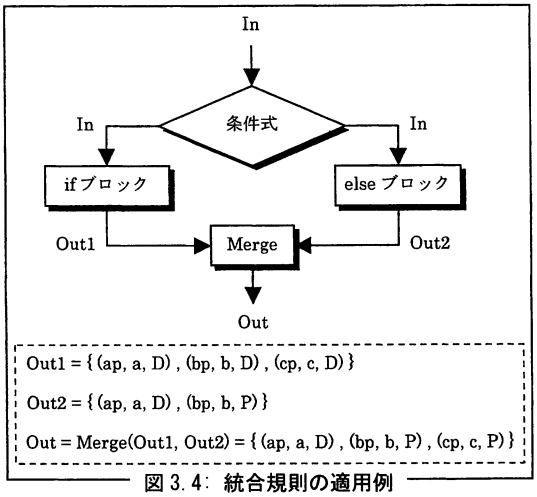

図 3.4: 統合規則の適用例

”ap” と ”a” は，Out1 と Out2 の両方で definitely-points-to 関係であるため，表 3.1 を利用(D∞D=D) した結果，Out において (ap, a, D) という Points-to セットを持つことになる。”bp” と ”b” は，Out1 で definitely-points-to 関係であるのに対して，Out2 で possibly-points-to 関係であるため，表 3.1 を利用 (D∞P=P) した結果，Out において (bp, b, P) という Points-to セットを持つことになる。”cp” と ”c” は，Out1 には存在するが Out2 には存在しないため，Out において (cp, c, P) という Points-to セットを持つことになる。

以上のような統合作業は，Pointer 解析を正確に行うためには欠くことができない作業であり，プログラム内のある地点へ到達可能な実行パスが複数存在する場合に行う必要がある。

[2]. Read/Write 解析

依存関係を検出するためには，各ステートメントにおいてどの場所が読み書きされるのかを知る必要がある。そのような情報 (Read/Write セット) を Points-to 解析で得られた Points-to セットに基づいて導き出す作業をこの解析で行う。図 3.5 に解析例を示す。

図 3.5 の中で使用された記号の意味は以下の通りである。

Read(S) : ステートメント S でリードされる場所の集合 (Read セット)
Write(S) : ステートメント S でライトされる場所の集合 (Write セット)

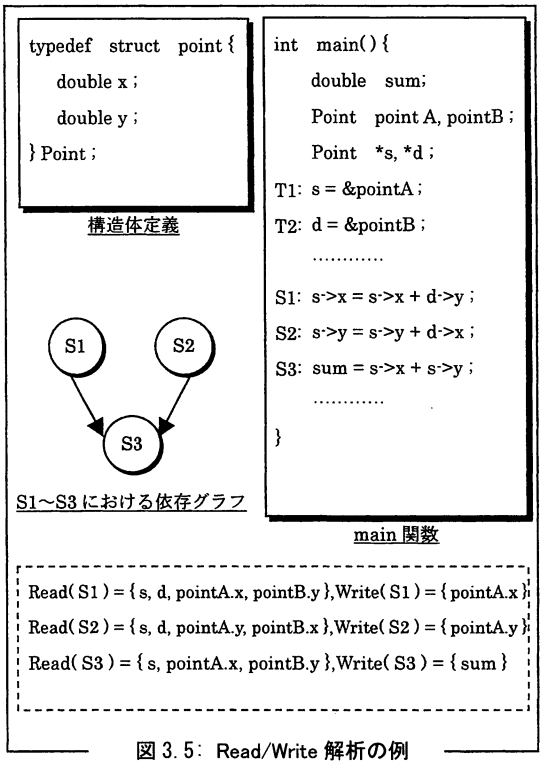

図 3.5: Read/Write 解析の例

ステートメント T1 と T2 より { (s, pointA, D), (d, pointB, D) } という Points-to セットが得られており，この Points-to セットはステートメント S1 を処理する時点でも有効であると仮定する。この情報を利用することで，ステートメント S1 の左辺にある s->x が，実際には pointA.x への書き込みを行っていることや，右辺にある d->y が pointB.y からの読み込みを行っていることが分かる。最終的な Read/Write セットは図 3.5 の中で示した通りである。

ここまでの段階で，図 3.5 では Points-to セットと Read/Write セットの両方が得られたことになる。つまり，図 3.5 で発生する並列性の抽出が可能になったということだ。ここでは，簡単のために S1 から S3 における依存関係のみを図 3.5 に示す。この結果は，Write(S1) と Write(S2) に含まれる pointA.x と pointA.y が，共に Read(S3) に含まれることから導き出される。

基本的には，以上のような手順で解析を行うことによって，プログラム内に存在している並列性を抽出していく。

3.2.2 Interprocedural 解析

プログラムの中に存在しているユーザ定義関数内部の Points-to セットを収集するための解析である。

Interprocedural 解析は以下のようなプロセスを行うことで実現される。

[1]. Map プロセス

すべての実引数と仮引数を比較し，利用可能なすべての Points-to セットを仮引数へセットする。また，同様の作業をグローバル変数に対しても行う。

[2]. 関数プロセス

呼び出される側 (Callee) の関数内部に対して Intraprocedural 解析を施す。

[3]. Unmap プロセス

関数プロセスで得られた Points-to セットを利用して，呼び出し側(Caller)の関数が持っている Points-to セットの更新を行う。

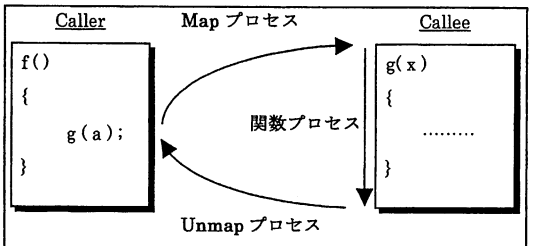

図 3.6: Interprocedural 解析の全体像

4. 解析ツール用プリプロセス

解析対象となるプログラムで使用された構造体の並列性を解析するためには，構造体のメンバリスト情報が必要となる。この情報はコード内に記述された構造体定義部分から取得するよう設計されている。故に，標準ヘッダ内で定義される構造体(例: timeval 構造体)のように，構造体定義部分が実際にコード内に記述されていない場合は解析できないでいた[8]。

以上のような理由から，本解析ツールでは GNU Compiler Collection(GCC)のプリプロセッサを利用することで，プリプロセスのみを施したプログラムの解析を実現する機能を付加することにした。具体的には図 4.1 のような手順で解析ツール用のプリプロセスを行う。図 4.1 のようにプリプロセスを行うことで，ユーザ定義のマクロのみ置き換えられ，かつヘッダ部分が展開されたプログラムが生成される。あとは，そのプログラムに対して並列性解析を行うことで，はじめに述べた構造体に関する解析上の問題点を排除することができる。ヘッダ以外のコードを維持するのは，標準ライブラリヘッダ内に存在するマクロ定義によってコードの置き換えが行われる(例: stderr, assert)のを防ぐためである。これによって，予想外の部分で置き換えが行われることによる不具合をなくすことが可能である。

以上のようなプリプロセスを行うことで，より多くのプログラムに対して解析を行うことが可能になると考えている。

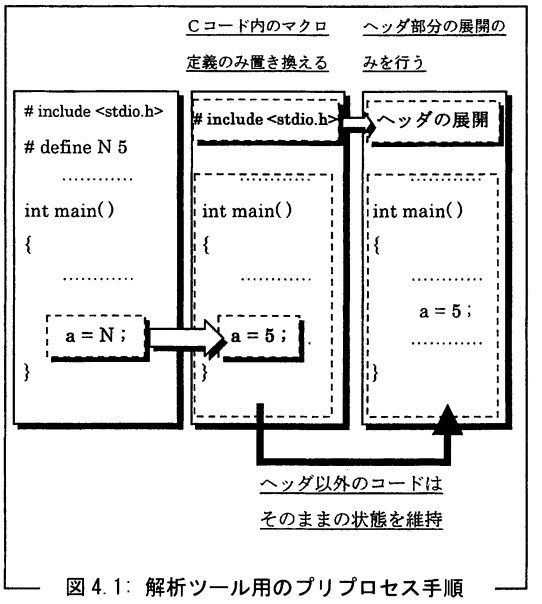

図 4.1: 解析ツール用のプリプロセス手順

5. 解析事例[8]

ここでは，簡単な例を用いることによって，今回より解析可能となった主な事例を示す。ただし，ここで示した事例はあくまで主要事例に過ぎないということに注意してもらいたい。本解析ツールは，従来は解析可能となっていた部分に対する処理の修正・追加を行うことによって，解析精度の向上が図られているだけでなく，解析可能パターンも追加されている。ここでは，5.3 ～ 5.5 がそれに相当する。

5.1 動的メモリ割り当てを含む場合の解析

現時点では標準 C ライブラリ関数である malloc によって動的メモリ割り当てが行われる場合のみ解析可能であるが，その他の関数で動的メモリ割り当てが行われる場合の解析も見据えた設計になっている。図 5.1 に解析結果を示す。

この解析例では，malloc による動的メモリ割り当てが S1 と S3 で行われている。S2 と S3 では，S1 で確保され

た TEST 構造体の領域へ書き込みが行われ，S4 と S5 では，S3 で確保された TEST 構造体の領域へ書き込みが行われる。このように，構造体の領域を動的に確保するケースが解析可能になったことの意味は大きい。なぜなら，多くの逐次プログラムで利用される可能性があるからだ。

```
struct TEST{
  int val;
  struct TEST *next;
};

int main()
{
  struct TEST *root;

  root =
    (struct TEST*)malloc(sizeof (struct TEST)); /* S1 */
  root->val = -1;                               /* S2 */
  root->next =
    (struct TEST*)malloc(sizeof (struct TEST)); /* S3 */
  root->next->val = 0;                          /* S4 */
  root->next->next = NULL;                      /* S5 */
    .....
}
```

プログラム

S1: { **(root, heap00, D)** }
S2: { (root, heap00, D) }
S3: { (root, heap00, D), **(heap00.next, heap01, D)** }
S4: { (root, heap00, D), (heap00.next, heap01, D) }
S5: { (root, heap00, D), (heap00.next, heap01, D),
(heap01.next, NULL, D) }

S5 終了時点での Points-to セット

Statement	Write(S)	Read(S)	依存先
S1	{root}		
S2	{heap00.val}	{root}	S1
S3	{heap00.next}	{root}	S1
S4	{heap01.val}	{root, heap00.next}	S1, S3
S5	{heap01.next}	{root, heap00.next}	S1, S3

S5 終了時点での Read/Write セットと依存先

図 5.1: 動的メモリ割り当てを含むプログラムの解析例

5.2 三項演算子使用時の解析

今回より解析可能になったものである。現段階では，単独での使用以外に，代入文での使用と変数宣言時での使用が可能である。それ以外のケースで使用されているプログラムは解析することができない。解析可能なケースを図 5.2 に示す。

```
int toPositive( int n )
{
  int ans = ((n < 0) ? (n * -1) : n );
  return(ans);                                  /* S1 */
}
int main(int argc, char *argv[ ])
{
  int num;

  num = atoi(argv[1]);                          /* S2 */
  num = ((num < 0) ? toPositive(num) : num );   /* S3 */
    .....
}
```

プログラム

図 5.2: 解析可能な三項演算子使用例

図 5.2 の例では，main 関数において，S3 が S2 に対して出力依存，かつフロー依存していることが解析結果から判断できる。

5.3 構造体使用時の解析

従来は，構造体が

”○.■” 形式か ”○->■” 形式

で利用された場合，”○” 部分には普通の変数のみ使用されることを前提としていた。よって，構造体を使用したプログラムを解析する際に，そのことが原因で解析不可能となるケースが頻繁に起こっていた。そこで，今回からはそのような制約を取り除くことで，構造体使用時の解析パターンの増加を行った。その例を図 5.3 に示す。

図 5.3 の例では，S1・S2・S5・S6 が従来の解析可能なパターンである。そして，残りの S3・S4・S7・S8 が今回より解析可能なパターンである。図 5.3 の構造体使用例が意味することは，先程示した ”○” 部分が再帰的に解析されることで，”○” 部分が普通の変数ではなく，”○.■” 形式か ”○->■” 形式で使用された構造体変数の場合にも解析が可能になったということである。

また，st_info 構造体は，前方参照型の構造体メンバ (data) を持っているが，これも今回より使用可能となったパターンの 1 つである。

本解析ツールでは，共用体にも対応した設計になっている。

```
struct st_info {
   int type;
   struct data_info* data;
} st1, st2;

struct data_info {
   int      idata;
   double ddata;
} d1, d2;

int main()
{
   st1.type = 1;                              /* S1 */
   st1.data = &d1;                            /* S2 */
   st1.data->idata = 5;                       /* S3 */
   st1.data->ddata = 12.5;                    /* S4 */

   st2.type = 2;                              /* S5 */
   st2.data = &d2;                            /* S6 */
   st2.data->idata = st1.data->idata * 2;     /* S7 */
   st2.data->ddata = st1.data->ddata * 2;     /* S8 */
    ..........
}
```

プログラム

図 5.3: 解析可能となった構造体使用例

5.4 do-all 処理判定の精度の強化

従来のものに加えて，多重ループの場合を考慮した do-all 処理判定が可能になった。その例を示す前に，以下で簡単に do-all 処理の説明を行う。

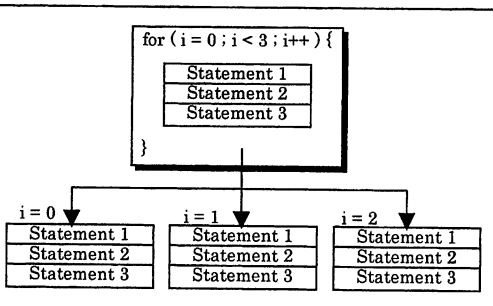

図 5.4: do-all 処理可能な時のイメージ図

[1]. do-all 処理[9)]

do-all 処理とは，ループ文のイタレーション間での並列性を利用した並列化技法のことである。図 5.4 に示したように，各イタレーション間に依存が発生していない場合は，各イタレーションを並列に実行することができる。それによって，逐次処理の時に比べて実行ステップ数を減らすことができるため，実行時間の短縮に繋がる。図 5.4 の例では，do-all 処理を行うことで，逐次処理では 9 ステップかかるところを，1/3 の 3 ステップで実行することが可能になる。

[2]. 多重ループを考慮した do-all 処理

多重ループを考慮した do-all 処理判定の例を図 5.5 に示す。

```
int main()
{
   int i, j, x[10], ans[10] ;

   for(i = 0 ; i < 10 ; i++) scanf("%d", &x[i]);     … ①

   for(i = 0 ; i < 10 ; i++){                        … ②
     ans[i] = 1;

     for(j = 0 ; j < 5 ; j++) ans[i] *= x[i];        … ③
   }
}
```

① → do-all 処理可能（注）
② → do-all 処理可能
③ → do-all 処理不可能

注：入出力の順序に意味がある場合は、設定を変えることで do-all 処理不可能と判断するよう設計してある。

図 5.5: 多重ループを考慮した do-all 処理判定の例

図 5.5 の例では，ループ①と②で do-all 処理可能と判断される。よって，使用可能なプロセッサ数に制限がないと仮定した時，ループ①と②におけるステップ数はそれぞれ図 5.6 のようになる。

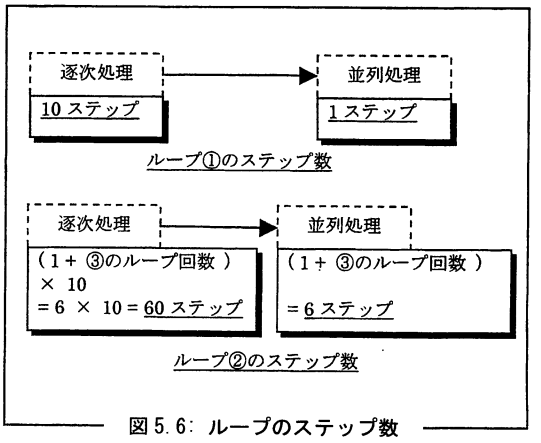

図 5.6: ループのステップ数

以上より，図 5.5 の例では，プログラム全体の総ステップ数を最高で約 1/10 に減らすことが可能であると分かる。したがって，実行時間についても，かなり短縮されると考えられる。

5.5 ポインタのアクセス形式の追加

今回より，ポインタを配列形式で利用することが可能になった。つまり，以下の図5.7にあるS2の左辺のような形式のことである。

```
int *ap, a[10] ;

ap = a;                 /* S1 */
  .....
ap[0] = 0;              /* S2 */
  .....
```

図5.7：ポインタの配列形式での使用例

また，図5.7にあるS1の右辺のように，配列名を配列の先頭アドレスとして利用しているが，これも今回より使用可能となった形式の1つである。

6．評　価

従来までは，開発中の本解析ツールを評価するために自作プログラムを使用していたため，客観的な評価を行っているとはいえなかった。そこで，並列化コンパイラの性能評価を行うための評価プログラムとして，世界的に有名なベンチマークプログラムに含まれているプログラムを使用する傾向がある[3]ことを受けて，本解析ツールにおいても，そのようなベンチマークプログラムを評価のために利用することにした。

今回採用したベンチマークプログラムは，HPC(High Performance Computer)のベンチマークとして有名[10]なlmbench である。そして，その中に含まれるプログラムから，動的メモリ割り当てが含まれているという点に着目して 1 つのプログラム(memsize)を解析対象として選択した。現時点では分割コンパイルする必要があるプログラムを解析することができない。そのため，それに該当するmemsizeを解析するにあたって，必要な関数や変数をmemsizeに追加する作業を行っている。ここでは，本研究で重点を置いた動的メモリ割り当て部分の解析結果を図6.1に示す。

図6.1の解析結果より，malloc関数が使用されているS4とS7の依存先がきちんと検出されていることが分かる。もちろん，ここで省略した部分に対しての依存も同様に検出されている。

以上のように，認知度の高いベンチマークプログラムに含まれるmemsizeというプログラムが解析可能になったことは，従来の評価プログラムの解析と比べて客観的な評価が行えるため，非常に重要なことであると考えられる。

```
.....
int main(int ac, char **av)
{
  .....
  while ( size < max ) {              /* S1 */
      free(where);                    /* S2 */
      size += 1024*1024;              /* S3 */
      where = malloc(size);           /* S4 */
      if (!where) {                   /* S5 */
            size  -= 1024*1024;       /* S6 */
            where = malloc(size);     /* S7 */
            .....
      }
  }
  .....
}
.....
```

memsize（一部抜粋）

Statement	制御依存先	データ依存先
S1		
S2	S1	
S3	S1	S1
S4	S1	S2, S3
S5	S1	S4
S6	S5	S1, S3, S4
S7	S5	S2, S3, S4, S5, S6

依存関係（一部抜粋）

図6.1：memsizeの解析結果（一部抜粋）

7．最後に

7.1 まとめ

今回追加された解析ツール用のプリプロセス機能や，動的メモリ割り当てを含むプログラムの解析によって，より多くの逐次プログラムが解析可能になったと考えている。特に，動的メモリ割り当てを利用したプログラムが解析可能になったことは，完成に向けての大きな前進である。

また，構造体使用時の解析パターンは，従来と比較してかなり増加したと考えている。リスト構造やツリー構造などを実現する場合など，構造体を利用する頻度は高い。また，構造体の領域を動的に確保するケースも多い。よって，構造体の解析可能なパターンが増加したことは，かなり重要なことであると考えられる。

さらに，多重ループを考慮したdo-all処理判定も可能

になった。従来まではその点を考慮せずに do-all 処理判定を行っていたため，正確な結果が得られないでいた。ループ文は並列処理の効果が得られやすいため，その判定が正確に行えることは重要なことである。

以上のように，動的メモリ確保を含むプログラムが解析可能になっただけではなく，既存の部分の修正や追加によって，より多くの逐次プログラムに対して並列性解析を行うことが可能になった。

7.2 今後の課題

自動並列化 C コンパイラを完成させるためには，以下に示すような課題に重点を置いて開発を進めていく必要がある。

●動的メモリ割り当てを行う場合の解析パターンの増加

現時点では malloc 関数によって動的メモリ割り当てが行われた場合のみ対応している。今後はより多くの関数に対応していくべきであろう。

●関数へのポインタが使用された場合の解析

今回より，プロトタイプ宣言で使用された場合は考慮したが，戻り値として関数へのポインタを返す場合や，実際に使用された場合は考えていない。そこで，ポインタ変数だけではなく，関数の Points-to セットも集めることで，この解析の実現を進めていくと良いだろう。

●再帰関数が使用された場合の解析

基本的には通常の関数と同じ解析手法を採るが，再帰の終了条件の判定といったことは，新たにその方法を検討する必要があるだろう。

●ユーザの知識を利用する方法の検討[3)]

プログラムの並列性解析には動的情報が必要となる場合があるため，静的情報のみに頼る方法に固執する必要はないと考えている。そこで，ユーザの知識を反映させるような指示文の導入も検討する必要があるだろう。しかし，より多くのユーザに利用されることを想定しているため，自動並列化の実現も念頭に入れることを忘れてはならない。

●評価用プログラムの検討

評価用プログラムとして，認知度の高いベンチマークプログラムに含まれているプログラムを使用することは，客観的な評価のためには必要なことである。現時点で解析可能な評価用プログラムは 1 つだけだが，今後は評価用プログラムとして適したプログラムを検討して，多くのものに対して正確に解析が行えるよう研究を進めるべきである。

参考文献

1). TOP500 http://www.top500.org/

2). 「アドバンスト並列化コンパイラ技術」産業科学技術研究開発基本計画 http://www.nedo.go.jp/informations/koubo/120324_4/ad-kihon.html

3). 平成 13 年度 アドバンスト並列化コンパイラ技術報告書（概要編）財団法人日本情報処理開発協会

4). Kai Hwang. "Advanced Computer Architecture: Parallelism, Scalability, Programmability". McGraw-Hill, Inc. 1993.

5). John L. Hennessy and David A. Patterson. "Computer Architecture A Quantitative Approach, Second Edition". Morgan Kaufmann Publishers, Inc. 1996.

6). M.Emami. "A practical interprocedural alias analysis for an optimizing/parallelizing compiler". Master's thesis, School of Computer Science, McGill University, September 1993.

7). R. Ghiya. "Putting pointer analysis to work". Doctor's thesis, School of Computer Science, McGill University, May 1998.

8). 手代木進. " 自動並列化Cコンパイラのための並列性解析" . Master' s thesis, 成蹊大学工学部工学研究科情報処理専攻. 2002.

9). 湯淺太一，安村通晃，中田登志之. " はじめての並列プログラミング" . 共立出版. 1999.

10). Kai Hwang and Zhiwei Xu. "Scalable Parallel Computing: Technology, Architecture, Programming". WCB/McGraw-Hill. 1998.