成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.45 No. 1 (2008) pp.31-38

C言語自動並列化トランスレータの開発
—ポインタ／配列依存解析の改良とタスク粒度の決定—

三浦　純[*1]，甲斐　宗徳[*2]

Development of Automatic Translator from C Programs to Parallel Programs Using MPI
- Improvement of Pointer/Array Analysis and a Decision of Task Granularity -

Jun MIURA[*1], Munenori KAI[*2]

ABSTRACT：We have been developing an automatically parallelizing translator for C programs. The translator consists of 5 phases, dependency and parallelism analysis, execution time analysis, task granularity analysis, task scheduling, and parallelized code generator. In this paper, we improve the accuracy of our dependency and parallelism analysis among pointer variables and/or array variables. The access relation of pointers and variables to be pointed by them used in conditional sentences and loop sentences is correctly obtained. The access relation of array variables which have index expression in the form of the first-order expression of loop index variable is also obtained. Moreover, we propose task fusion method using data locality which arranges fine grain tasks into coarse grain tasks.

Keywords：automatic parallelization, pointer analysis, dependency analysis, task fusion

（Received March 25, 2008）

１．はじめに

1．1　背景

コンピュータの性能向上のためマルチプロセッサシステムを構成することが主流となっている。

マルチプロセッサシステムでは並列性と効率を考慮したプログラムを実行しないと優れた性能を十分に引き出すことができない。そのためにはソフトウェア及びハードウェアの詳細な知識が必要とな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となっている。

1．2　研究目的

上記の問題を解決する手段として自動並列化トランスレータの利用が挙げられる。しかし並列性の抽出やコード生成部における不必要な同期待ちのコードの挿入などの問題からマルチプロセッサシステムの性能を十分に引き出すことができず，手動で並列化したプログラムと同様の実行性能を得ることができないでいる[1)]。

プログラム内の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とステートメントレベルの細粒度の並列性の両方を引き出すことが，マルチプロセッサシステムの性能を引き出しプログラムの実行効率を向上させる。従って様々な粒度の並列性を引き出すことが自動並列化トランスレータには望まれている。

本研究では，細粒度タスクから粗粒度タスクまでの並列性を引き出し，実行性能の向上に有効であると判断した並列性を活かすためにポインタを含む依存解析や並列性の抽出をおこなっている。また並列実行時の性能と並列化トランスレータ全体の解析時間を減らすことを考慮し，データの局所性を利用したタスク粒度の決定手法を提案する。

２．C言語自動並列化トランスレータ

C言語自動並列化トランスレータは,C言語で記述されたソースプログラム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プログラムを生成するものである。本研究で開発するC言語自動並列化トランスレ

[*1]：工学研究科情報処理専攻修士学生
[*2]：情報処理専攻教授 (kai@st.seikei.ac.jp)

ータの全体の流れを図 2.1 に示す。

初期作業として解析対象となるソースプログラムの並列性解析を行う。その時点のソースコードでは並列処理不可能と判定されたループが，ループリストラクチャリングによって並列処理可能なループに変換可能と判断するとソースコードの変換作業を行い，変換後の新たなソースコードに対して再び並列性解析を行う。

中盤の作業として実行時間解析，タスク粒度解析，部分タスクスケジューリングを行う。実行時間解析では並列性解析の結果に基づいたタスク毎の実行時間と，タスク間のデータ通信時間を求める。タスク粒度解析は，前段 2 つの解析の結果に基づき，各タスクをどのような粒度でプロセッサに割り当てれば効率の良い並列処理が行えるかを考慮しタスク粒度を決定する。マクロタスクなどの粗粒度タスクに対して，通信時間を考慮した部分タスクスケジューリングを適用することで，どのような順序で実行すれば最短時間でその粗粒度タスクの処理が完了するか解析する。

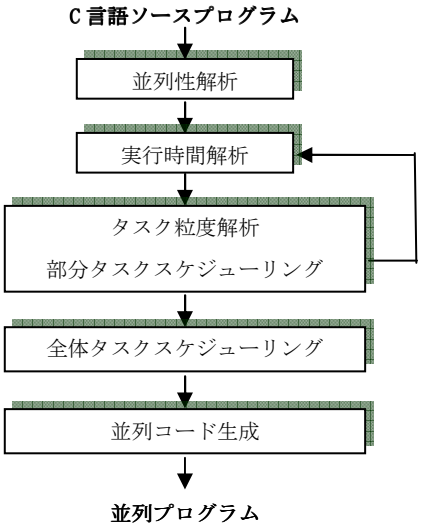

図 2.1 C 言語自動並列化トランスレータの処理の流れ

次に全体タスクスケジューリングを行う。実行時間解析とタスク粒度解析，部分タスクスケジューリングを繰り返した後の最終的な粒度のタスク集合をプロセッサに割り当てる最適スケジューリングを行う。

最後にターゲットマシンにおいて優れた実行性能を実現するような並列プログラムの自動生成を行う。

３．並列性解析

プログラムに記述された順番に実行を行わなくても実行結果に影響を及ぼさない部分を特に並列性がある，という。以下のコードを例にする。

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

S1, S2 の実行順序を入れ替えたとしても同じ実行結果を取得することができる。並列性解析ではこの並列性をより多く抽出することが目的となる。

3. 1 Pointer 解析

プログラムのある時点でポインタ変数がどのロケーション（変数）を指しているか，という情報(Points-to セット)を取得する解析である。その情報には以下の２種類がある。

①ポインタ変数 x がプログラムのある時点でロケーション y を明らかに指している場合 **x-definitely-points-to-y** と呼び，このことを**(x, y, D)**と表す。

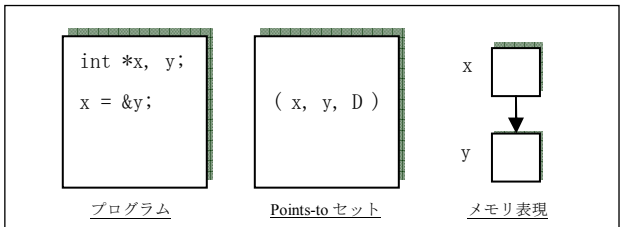

図 3.1 Definitely-points-to 関係の例

②ポインタ変数 x がプログラムのある時点でロケーション y を指している可能性がある場合 **x-possibly-points-to-y** と呼び，このことを**(x, y, P)**と表す。

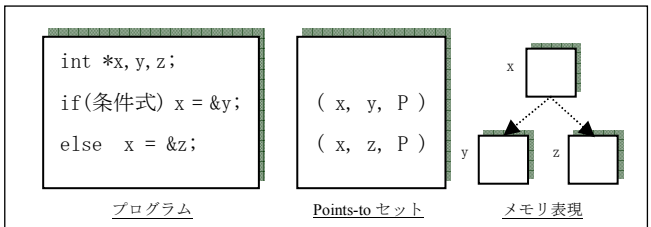

図 3.2 Possibly-points-to 関係の例

ここで図 3.2 のような if-else ブロックを解析した結果をここで表す。本研究ではこれまで提案されていた制御文とループ文での points-to セットを取得するロジックを実装した。ここでは制御文全体の points-to セットを取得するためのロジックを図 3.3 に示す。

図 3.3 は，if ブロック，else ブロック別々に points-to セットを解析しそれを統合する，という概念を表した図である。もし入れ子があったとしても，ブロック内で再帰的に上記のロジックを繰り返せばよい。また points-to セットを統合する時に points-to 関係の種類をどうするか，という問題が発生するが，この問題は表 3.1 に示す統合規則に従って種類を決定するものとする。

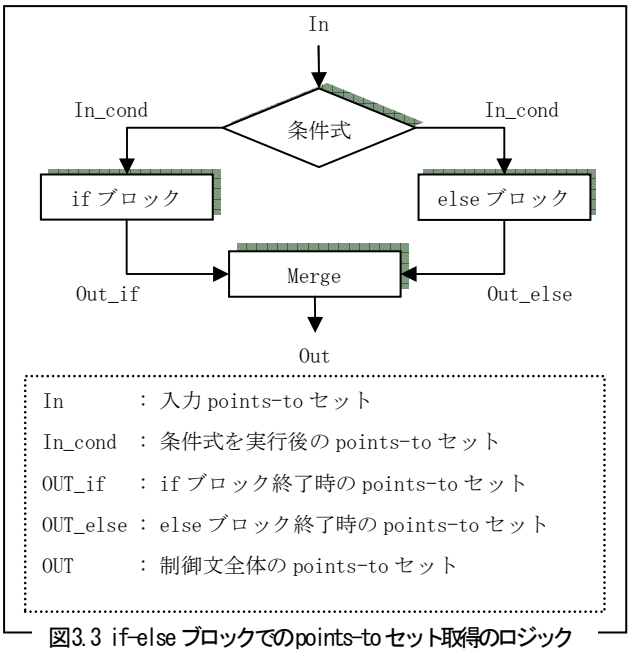

図3.3 if-else ブロックでのpoints-to セット取得のロジック

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

3. 2 Read/Write 解析

Pointer 解析において得られた Points-to セットを使って各ステートメントにおける変数がどのロケーションを read または write アクセスするのかを解析する。変数がポインタ変数の場合，間接参照によりアクセスするロケーションが加わる。それ以外の変数は変数自身のロケーションのみアクセスする。ポインタ変数のアクセスするロケーションの例を表 3.2 に表した。

表3.2 ポインタ変数のアクセスロケーション

Points-to セット
（ a, b, D ）

記述	ロケーション
&a	アクセスな
a	{ a }
*a	{ a, b }

表 3.2 はプログラムのある時点での Points-to セットの状態と，ポインタ変数 a が表のような記述をされた時にアクセスするロケーションの集合を示している。&a と記述されるとポインタ変数 a のアドレスを表すだけで実際のアクセスはない。a と記述されるとポインタ変数 a を参照するだけなので，アクセスしたロケーションは a となる。*a と記述されるとまずポインタ変数 a を参照しそして変数 b をアクセスする。従ってアクセスしたロケーションは a と b になる。

3. 3 依存解析

並列性を抽出するためには依存解析を行い，変数の依存関係を明確にしなければならない。もし依存関係が検出されれば，そこには並列性がないと判断される。

3. 3. 1 データ依存解析

各ステートメントで使用される変数の示すロケーションの read/write の状況からデータの依存関係を解析する。データ依存には 3 種類存在する。

① フロー依存

ステートメント S1 で write されたロケーションが，後続のあるステートメント S2 で read される場合，S2 は S1 にフロー依存していると定義する。

② 逆依存

S1 で read したロケーションが S2 で write される場合，S2 は S1 に逆依存していると定義する。

③ 出力依存

S1 で write したロケーションが S2 で write される場合，S2 は S1 に出力依存していると定義する。

3. 3. 2 制御依存解析

分岐命令に続くブロックの処理は条件文の判定次第で実行するかしないか決定する。そのためそれらブロックの処理は条件文に制御依存していると定義する。以下のコードを考える。

```
if( CS1 ) {
    S1;
}
```

S1 は CS1 にデータ依存がなかったとしても CS1 を評価して，それが true になった後で実行しなければならない。

3. 4 ループイタレーション解析

以上述べた Pointer 解析，Read/Write 解析，依存解析を組み合わせて改良し，さらにポインタや配列を用いたループの解析に適用した。ループのイタレーション間に依存関係があると，ループの処理を並列に行うことができない。この解析では連続するイタレーション間の依存解析と離れたイタレーション間の依存解析を行う。最初に並列実行効率の高い do-all 処理について述べる。

3. 4. 1 do-all 処理

do-all 処理とはループのイタレーション間に依存関係がなく，各イタレーションの処理を並列に行うことができることを言う。プログラムの大部分はループを含むことが多く，その部分に並列処理を施すことにより大幅な実行時間短縮に繋がることが考えられる。図 3.4 に do-all 処理における並列処理の概念図を示す。

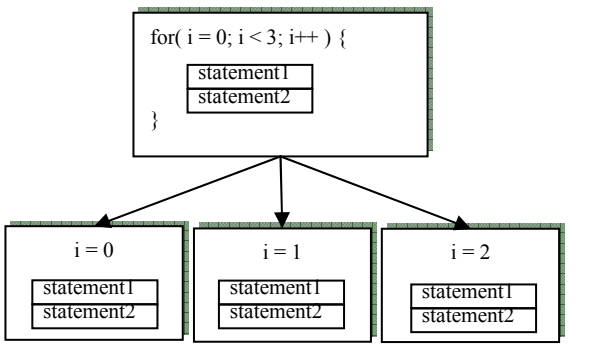

図3.4 do-all 処理可能の概念図

3. 4. 2 配列アクセス

ループ内において配列の要素へアクセスする場合，たいてい添字が変数によって記述されている。そこで従来から添字が変数で記述されている場合を考慮して開発されていたが,ループ変数の係数が 1 の場合のみであった。本研究では添字がループ変数の 1 次式で記述された場合に解析を行えるように拡張した。具体的には外側のループ変数を i, 内側のループ変数を j とする 2 重ループがあり，変数 X，Y の添字がループ変数の 1 次式で記述される場合，図 3.5 に示すような行列とベクトルを用いてアクセスロケーションを表すことができる。

配列の添字は 1 次式で記述される場合ばかりではなく，2 次式や配列記述が入れ子で使われたりもする。従ってこの手法だけでは全ての場合を解析することはできない。しかし，1 次式で記述される場合を考慮することで，最も基本的な配列の要素へのアクセスを解析することができる。

$$X[i-1] \Rightarrow [1\ 0]\begin{bmatrix} i \\ j \end{bmatrix} + [-1]$$

$$Y[1][j+1] \Rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

図3.5 1 次式での配列アクセスの一般化の例

3. 4. 3 連続するイタレーションにおける依存解析

本研究では配列アクセスの変更に伴う，連続イタレーションにおける依存解析の実装を行った。連続するイタレーションでの依存解析は k イタレーション目に write されるロケーションが k+1 イタレーション目に read されるかどうかで依存解析を行うことができる。連続するイタレーションに対して依存解析を行う例を図3.6に示す。

```
int i, j, a[10][10], b[10];
for( i = 0; i < 10; i++ ) {     …(1)
   b[i] = i + 100;
   for( j = 0; j < 9; j++ ) {   …(2)
      a[i][j+1] = a[i][j] * b[i];
   }
}
```

(1)：do-all 処理可能候補

(2)：do-all 処理不可能

図3.6 連続したイタレーションの依存解析の例

図 3.6 のコードはループ(2)のイタレーション間に配列変数 a に関して依存が発生している。実際に例を挙げると，1 イタレーション目の j の値は初期化する値である 0 であり，2 イタレーション目の j の値はインクリメントの値が 1 なので 1 である。i の値はループ(1)のイタレーションに依存するのでここでは任意の値を与えることにする。配列変数 a に対して図 3.5 で表したような一般化を行い，ループ変数の部分に値を代入してロケーション計算を行う。つまり i=0 の時, 1 イタレーション目の write 側の 1 次元目の添字の値は 1×0+0=0 より 0 となり, 2 次元目の添字の値は 1×0+1=1 より 1 となり, a[0][1]に代入が行われることがわかる。read 側の 2 イタレーション目の 1 次元目の添字の値は 1×0+0=0 より 0 となり, 2 次元目の添字の値は 1×1+0=1 より 1 となり，アクセスするロケーションが a[0][1]であることがわかる。その結果，1 イタレーション目に write したロケーションを 2 イタレーション目で read しているのでイタレーション間に依存があることがわかる。しかしループ(1)ではイタレーション間に依存はなく，do-all 処理可能候補となる。

3.4.4 離れたイタレーションにおける依存解析

プログラムの種類によっては連続イタレーションには依存関係はないが，少し離れたイタレーション間に依存関係がある場合がある。本研究ではその解析も行えるようにした。解析できるようになったのはループ変数の値が単調増加（減少）する場合であり，図 3.7 にその種のプログラムの例と解析結果を示す。

```
int i, j, maxi, maxj, a[100], b[10][10];
for( i = 0; i < maxi; i++ ) {     …(1)
   a[7*i+5] = a[3*i+7];  /* S1 */
   for( j = 0; j < maxj; j++ ) {     …(2)
      b[i][j] = b[i][j] * a[7*i+5];  /* S2 */
   }
}
```

(1): do-all 処理不可能

(2): do-all 処理可能

図3.7 離れたイタレーション間の依存解析の例

図 3.7 のコードのループ(2)は do-all 処理可能である。連続するイタレーションの依存解析を行った時点ではループ(1), ループ(2)ともに do-all 処理可能候補という判定になる。離れたイタレーションの依存解析を行う条件は，イタレーション毎にループ変数が単調増加（減少）し，ループ変数の係数行列の各要素が 1 より大きい，または単調増加（減少）の値が 1 より大きい（-1 より小さい）場合である。ループ(2)の S2 に関してはループ変数 j の係数行列が 1 で，単調増加の値も 1 なので離れたイタレーションの依存解析を行う必要はない。従ってこの時点でループ(2)は do-all 処理可能となる。

しかしループ(1)の S1 ではループ変数 i の係数が 1 ではないため，離れたイタレーションの依存解析を行う必要があり，この例では依存関係は存在する。ここでも図 3.5 の配列アクセスの一般化を利用して添字の取り得る値を確保した配列の容量だけ計算し，比較することで依存解析を行う。つまり配列変数 a の write 側の添字が取り得る値は 7×0+5=5，7×1+5=12，…，7×13+5=96 であることがわかり，read 側の添字が取り得る値は 3×0+7=7，3×1+7=10，…，3×30+7=97 とわかる。そして write 側の添字の値と read 側の添字の値を比較していくと，図 3.7 のループ(1)は，i=2 の時に write 側の配列変数 a の添字が 19 となり，i=4 の時に read 側の添字が 19 となることが判明し，依存関係があることがわかる。

このように添字の比較を行うことで離れたイタレーション間のデータ依存解析を行えるようになり，依存がなければ do-all 処理可能と見なすループを増やすことができるようになった。

今回ポインタ解析がポインタ変数 p に対する*(++p)や*(p+5)のようなポインタ演算に対応していない。しかしポインタ変数が配列を指している場合，Pointer 解析と連動することでポインタ変数を配列同様に扱うことを可能にした。その場合も添字の記述は 1 次式に限定している。従ってポインタ変数が配列のように記述された場合も解析可能となった。

4．タスク粒度の決定

従来，並列性解析のフェーズにおいてのタスク粒度は 1 ステートメントを 1 タスクとする細粒度であった [4]。逐次処理ではプロセッサ 1 台で処理を行うので通信を行う必要はないが並列処理の場合，プロセッサを複数台使用するため通信が発生する。従ってタスクの処理時間と通信時間のオーバヘッドのトレードオフを考慮しなければならず，細粒度タスクでは逐次処理を行った方が効率の良い場合があり，並列性を活かすことができない。また並列実行をする時に一方が粗粒度タスクで，もう一方が細粒度タスクであると処理時間に差があり，早く処理を終えたタスクがもう一方のタスクの処理が終えるのを待たなければならない場合，並列実行の効果が薄れてしまう。従って，本研究では極端に粒度に差があるタスクが混在することを回避するため細粒度タスクを融合し，粗粒度タスクを生成する手法を考案した。タスク融合を行うことでタスク数が減少するためスケジューリング時間が短縮し，さらに通信を行う必要がなくなるため，通信オーバヘッドを減らすことができることにつながる。

4. 1　タスクの種類

タスクの種類は以下の 4 種類が存在する。

① SELECT(IF, ELSE_IF, ELSE)タスク
制御文のボディを包含するタスクとして定義される。

② LOOP タスク
繰り返し文のボディを包含するタスクとして定義される。

③ FUNC タスク
ユーザ定義関数を呼び出すステートメントとして定義される。

④ BASIC タスク
上記のタスクの種類に当てはまらないタスクの種類として定義される。

4. 2　タスク粒度の決定手法

並列性を解析するためには細粒度タスクで依存関係を解析し始める。それら細粒度タスクを融合することで粗粒度タスクを生成する。本研究はタスク融合を行う時にデータの局所性を利用して粗粒度タスクを生成した。図 4.1 のタスク 1，タスク 2 のようなステートメントは BASIC タスクとして粗粒度にまとめられる。if 文が記述されている場合には SELECT タスクをマクロタスクとし，if 文のボディのステートメントはサブタスクとして扱う

ことにする。for 文，while 文の LOOP タスクも同様である。特に if 文に関しては一般的に SELECT タスク内部に 1 つの IF タスクと任意の数の ELSE タスク，ELSE_IF タスクが存在する。図 4.1 の例では SELECT タスクのサブタスクとして IF タスクと ELSE タスクが存在している。

本研究ではタスク間で依存している変数のペアの数を依存強度と定義する。並列実行を行う場合，依存強度が高いタスク同士を異なるプロセッサに割り当てるとそれだけ通信が多くなることから，経験的にはそのようなタスクは同じプロセッサに割り当てる方が良い。そして同じプロセッサに割り当てて連続実行させるならそれらのタスクを 1 つのタスクに融合して良いことになる。

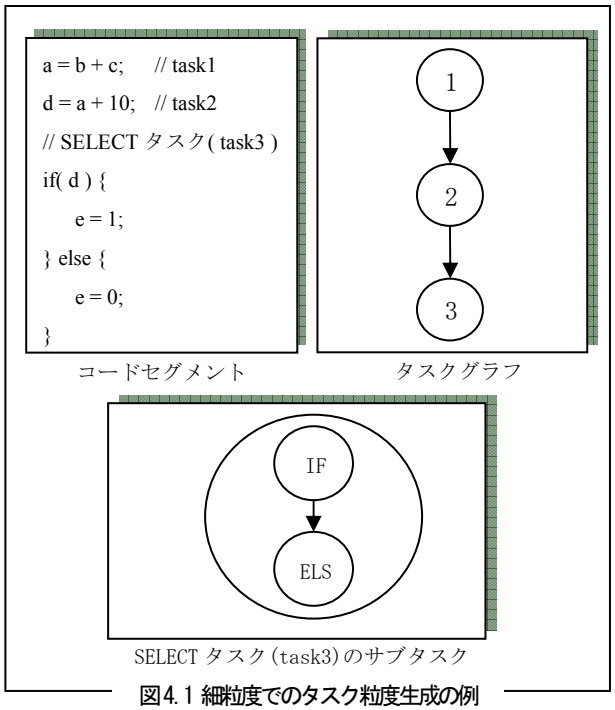

図4.1 細粒度でのタスク粒度生成の例

そこで本研究では依存強度に閾値を与えタスク融合を行う範囲を限定しタスク融合を行った。依存強度に閾値を与えることでそれ以上の依存強度を持ったタスク間のみ融合が行われ，最終的に元の依存強度より平均して低い依存強度にすることを意図している。従って，依存強度の高いタスク間を優先的にタスク融合の候補として選んでいるが，依存強度が等しかった場合

① 融合後のタスクの依存タスク数が最も少ない

② 融合後のタスクの依存強度が最も低い

③ 融合後のタスクのステートメント数が最も多い

の順で検討し，最終的にタスク融合を行うタスク集合を決定する。

全てのタスクが融合されてしまう，すなわち逐次処理に戻ってしまうことを避けるために，タスク融合後のタスク粒度の上限を決めておくことが必要であり，上限を設定するために今回用いた指標はタスクの持つステートメント数で表す。ただし LOOP タスクの処理の負荷はステートメント数というよりループの回数に依存するので，LOOP タスクの負荷の測定としてこの指標を用いることはできないが，LOOP タスクは他のタスクと融合を行わなくてもある程度大きな粒度を持っているであろうと仮定し，LOOP タスクは 1 つのマクロタスクとして扱うことにした。

５．評　価

今回ベンチマークプログラムとして採用したのは NPB(NAS Parallel Benchmarks)のプログラム IS(Integer Sort)である。現状では関数間解析の実装が十分に行われておらず，ユーザ定義の関数コールの部分が解析できないため，関数呼び出し側のコード中にユーザ定義関数を展開した。

ここでは本研究で重点的に進めてきたポインタ／配列を利用したプログラムに対する依存解析とタスク粒度の決定についての解析結果を示すことにする。

図 5.1(1)は IS のコードの一部であり，この部分の依存解析の結果を示す。図 5.1(3)の表のデータ依存の部分に空欄があるが，ここは図 5.1(1)で示さなかった部分と依存関係を持っているため，ここでは空欄とした。前述したように配列の要素の参照に関して今回はループ変数を使った 1 次式のみに対応しており，内側の 2 番目のループの配列は添字で配列を用いているため解析不可能となる。従ってこのループに関しては逐次処理を行うと判定せざるを得ない結果となる。

表 5.1 はタスク融合を行った結果を示している。今回同じマクロタスク内に存在するタスク間の依存強度に閾値を設定し，融合後のタスク粒度の上限を決める指標にステートメント数を利用してタスク融合を行った。

表 5.1 における各項目について説明する。ステートメント数はプログラム全体のステートメント数，最大粒度はタスク融合後のタスクの持つステートメント数の上限である。依存強度の閾値は，その値以上の依存強度を持つタスクに対してタスク融合を行うことを意味している。タスク数の変化はタスク融合前とタスク融合後のタスク数の変化の割合であり，依存強度の変化はタスク融合前とタスク融合後のタスクグラフ上の全ての依存強度を合計し，依存の数で割った依存強度の平均値の変化の割合である。

```
……
int main( int ac, char **av ) {
   ……
   for( …; …; … ) {                /* …;S0;… */
     for( i=0; i<MAX_KEY; i++ )     /* S1;S2;S4 */
        key_buff1[i] = 0;           /* S3 */
     ……
     key_buff_ptr = key_buff1;      /* S5 */
     for( i=0; i<NUM_KEYS; i++ )    /* S6;S7;S9*/
       key_buff_ptr[key_buff2[i]]++; /* S8 */
     for( i=0; i<MAX_KEY-1; i++ )   /* S10;S11;S13 */
       key_buff_ptr[i+1] += key_buff_ptr[i]; /* S12 */
     ……
   }
   ……
}
```

(1) IS(一部抜粋)

ループ開始 statement	解析結果
S1	do-all
S6	do-across
S10	do-across

(2) ループ解析の結果

Statement	制御依存	データ依存
S1	S0	
S2	S0	S1, S4
S3	S2	S1, S4, S9, S12
S4	S2	S1, S2, S4
S5	S0	S3, S8, S12
S6	S0	
S7	S0	S6, S9
S8	S7	解析不可
S9	S7	S6, S7, S9
S10	S0	S6, S9
S11	S0	S10, S13
S12	S11	S3, S8, S10, S12, S13
S13	S11	S10, S11, S13

(3) 依存関係(一部抜粋)

図5.1 ISの依存解析結果

表5.1 タスク粒度生成結果

ステートメント数	最大粒度	依存強度の閾値	タスク数の変化 (%)	平均依存強度の変化 (%)
277	5	2	98.5	88.4
		0	83.6	131.0
	10	2	98.5	88.4
		0	82.1	137.5
	20	2	95.6	93.2
		0	80.4	143.1
	タスク融合を行わなかった場合		100.0	100.0

表5.1の結果から依存強度の閾値を0にする，つまりタスク融合を行わない部分を設定せずに，どの部分でも良いから1タスク内ステートメント数の上限を満たすまで融合を進めていく，という手法では，タスク数はタスク融合前と比較して16%～20%減少するが，同時に依存強度が31%～43%増加してしまう。しかし依存強度の閾値を2にする，つまりタスク融合を行わない部分を設定することによってタスク数はタスク融合前と比較して1.5%～4.4%減少と，依存強度の閾値を0に設定した時より減少率は大幅に減少するが，依存強度の変化が 11.6%～6.8%減少することができ，依存強度を減少させつつタスク融合を進めることができたことがわかる。

今回の評価のサンプルコードである IS コードを手動で並列化する意図で見てみたところ，並列化可能であると判断されるステートメントやループなどの制御構造について，正しく並列化可能であると判断できるようになったことが確認できた。

タスク融合はスケジューリング時間短縮のためにタスク数を減らすことと融合により通信オーバヘッドを低減することが目的であったがこれら両方を実現した。これらタスク融合の結果が明らかになるのは共有リソース[5]のソースコードインタフェースに各タスクの情報が記述される時であり，その情報を利用してコード生成部では並列プログラムを生成することができる。

6．終わりに

6．1　まとめ

本研究が重点を置いたポインタ／配列の依存解析の改良により多くのプログラムが解析可能になった。特に配列の1次式の記述に対応したことにより配列の基本的アクセスに関して対応できるようになり，ループのイタレーション間解析も幅広く解析できるようになった。特に離れたイタレーション間の解析は正確な do-all 判定に必要不可欠なことであり，重要なことと考える。ポインタ解析においてもポインタ変数が指している可能性のあるロケーションを網羅し，それを依存解析に応用できるようになったことでより正確な依存解析になった。また並列処理時の通信オーバヘッドを減らすようなタスク粒度の決定が並列性解析のフェーズでできるようになったことは並列処理時間の減少への効果とともに自動並列化トランスレータの全体の解析時間の短縮の効果があり，重要なことと考える。

6. 2　今後の課題

• 対応できる配列アクセスのパターンの増加

配列アクセスのパターンを全て網羅することは難しいが，対応できるパターンを増やすことは並列性の向上のために必要なことである。例えば添字に配列が使われた場合，動的な解析が必要になるが必要以上に動的解析を行うと解析時間が膨大になってしまう。解析量と得られる情報のトレードオフを検討する必要がある。

• ポインタ解析の強化

C 言語におけるポインタの使われ方は多岐に渡るのでこちらも全て網羅することは難しい。特に動的メモリ割り当てが入れ子で使われた場合の問題や関数ポインタの問題など検討事項は多い。また配列アクセスパターンの増加と同様に，動的な解析が必要になるケースも考慮する必要がある。

• ユーザの知識の利用

上記のように，並列性解析には動的な解析は必要不可欠である。しかし動的な解析を全てシステム側で行う必要はなく，ユーザから指示文などにより提供してもらうことも検討すべきである。その時，並列プログラムの知識がないユーザでも指示文を記述できるようなフォーマットや情報の内容を考慮する必要がある。

• タスク粒度の決定の方式の検討

本研究ではタスクの最大粒度としてステートメント数という指標を利用したが，他の基準も考慮することは重要である。そして実際のコード変換時にどの指標を使うかはユーザに指定させる，という方法も考えられる。

• 関数間解析の実施

現状では関数間解析が十分に行われず，情報のやり取りができていない。関数間におけるグローバル変数の取り扱いや再帰関数への対応などを検討する必要がある。

参考文献

1) “アドバンスト並列化コンパイラ技術（概要編）”, 財団法人日本情報処理開発協会, 2002
2) 手代木　進 :“自動並列化 C コンパイラのための並列性解析”, 修士論文,成蹊大学工学部工学研究科情報処理専攻.2002.
3) 斉藤　義功 :“動的メモリ確保を含む C プログラムの自動並列性解析”, 修士論文,成蹊大学工学部工学研究科情報処理専攻.2003.
4) 美濃本　一浩 :“C 言語自動並列化トランスレータの開発”, 修士論文,成蹊大学工学部工学研究科情報処理専攻. 2005.
5) 田中　康之 :“C 言語自動並列化トランスレータの開発～マクロタスクの取り扱いに伴う拡張”, 修士論文, 成蹊大学工学部工学研究科情報処理専攻.2007.
6) 尾高　輝: “通信遅延を考慮したタスクスケジューリングのためのタスク粒度解析”, 修士論文,成蹊大学工学部工学研究科情報処理専攻.2007.
7) Alfred V.Aho, and Monica S.Lam, and Ravi Sethi, and Jeffrey D.Ullman : “Compilers principles, Techniques, & Tools Second Edition”, Addison Wesley, 2007
8) Maryam Emami, Rakesh Ghiya, Laurie J.Hendren : “Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers”, conference on Programming language design and implementation, ACM, pp.242-256, 1994
9) 関口龍郎 :“C 言語のための現実的なポインタ解析”, 日本ソフトウェア科学会, コンピュータソフトウェア Vol 21, pp.456-471, 2004
10) 五月女健治 :“yacc/lex プログラムジェネレータ on UNIX”, テクノプレス, 1996
11) NAS Parallel Benchmarks in OpenMP
http://phase.hpcc.jp/Omni/benchmarks/NPB/
2008/01/30 現在，参照可能
12) NAS Parallel Benchmarks Changes
http://www.nas.nasa.gov/Resources/Software/npb.html
2008/01/30 現在，参照可能