依存グラフに基づく並列化処理機構と一貫性保持

内田 智士, 朝倉 宏一, 渡邊 豊英
名古屋大学大学院 工学研究科 情報工学専攻
E-mail:{uchida,asakura,watanabe}@watanabe.nuie.nagoya-u.ac.jp

1 はじめに

近年、計算機をネットワーク機器で接続した計算機クラスタ環境がコスト・パフォーマンスの良い並列処理環境として注目を浴びている。パーソナルコンピュータやワークステーションなどの計算機、および、それらを接続するイーサネットなどのネットワーク機器の高性能化、低価格化により、計算機クラスタ環境の構築が容易となり、一般ユーザに対しても並列処理が普及しつつある。

しかし、並列プログラムの開発には、豊富な専門知識と長年の経験が必要とされるので、一般ユーザにとっては容易な作業ではない。一般ユーザでも並列プログラムを容易に開発可能とするためには、逐次プログラムを並列プログラムへ変換する並列化コンパイラの開発が望まれる。プログラム並列化手法としては、ループ文並列化処理やマルチグレイン並列化処理 [1,2] が数多く提案されている。また、コンパイル時の静的な解析処理だけでなく、プログラム実行中の動的な情報を用いた並列化処理なども研究されている。

このように、新しい並列処理環境や並列化手法が研究されているが、これらの有効性を検証するために並列化コンパイラを構築することは非常に困難で手間のかかる作業である。特に、計算機クラスタ環境の場合、プロセッサの性能や台数、ネットワーク速度やネットワーク・トポロジなどのシステム構成が様々で、環境により最適な並列化手法の構成などは異なるのが普通である。つまり、様々な構成に対応可能な並列化コンパイラの構築は不可能である。したがって、システム構成や応用プログラムに合わせて、並列化アルゴリズムを変更、修正し並列化コンパイラを開発することができる環境を用意することが望ましい。

我々は、応用プログラムに対応した並列化アルゴリズムの開発や計算機環境の構成変更に対応したスケジューリングを支援するために、並列化コンパイラ・ツールキット [3] を開発している。従来のコンパイラ開発支援ツールとして、字句解析、構文解析処理を支援する lex, yacc、通信標準インタフェースを提供する PVM[4], MPI[5], OpenMP[6]、コンパイラの中間言語を提供する SUIF[7] などが挙げられる。しかし、これらのツールは並列化コンパイラ開発の中核となるプログラム並列化処理の開発を直接支援するまでには至っていない。それに対し、我々の並列化コンパイラ・ツールキットは、プログラム並列化アルゴリズムの開発支援を目的として開発している。並列化コンパイラ・ツールキットでは、プログラム並列化処理を 2 つの処理でモデル化している。すなわち、入力されたプログラムを並列処理の最小単位に分割するプログラム分割処理と、分割されたプログラムを計算機環境の構成を考慮して割当てるプログラム割当て処理である。プログラム分割処理部ではプログラム分割方法を明示的に表し、プログラム割当て処理部では計算機環境をパラメータ化して表すことで、様々な並列化手法、計算機環境に対応可能な並列化コンパイラを開発することができる。

本稿では、プログラム分割処理部において、プログラムを並列処理の最小単位に分割する処理機構とプログラム分割手法の知識化表現について述べる。プログラムからの並列性抽出は、プログラム要素の特性とプログラム要素間の依存関係の情報により可能となる。そこで、我々のプログラム分割処理では、プログラムの依存グラフを用いて並列性を抽出する。依存グラフのノードを並列実行単位と捉えると、プログラム分割処理は依存グラフ上のノードを互いに独立、かつ適度な粒度へ再構成す

ることとして考えることが可能となる。すなわち、プログラム分割手法を依存グラフの再構成ルールとして記述可能となり、依存グラフの再構成ルールの集合としてプログラム分割処理戦略を表すことができる。しかし、再構成の結果、プログラムの一貫性を壊すことがある。その一例として、デッドロックを生ずる LOOP 依存関係がある。 LOOP 依存関係において、プログラムの一貫性を保持するようにする修正処理についても述べる。

2　プログラム分割処理機構

本章では、プログラム分割処理機構を提案し、プログラム分割処理戦略の表現法について述べる。

2.1　プログラム分割処理のモデル化

本節では、プログラム分割処理の本質を捉えながらプログラム分割処理機構をモデル化し、様々なプログラム分割処理戦略を一様に扱うためのプログラム分割処理機構を提案する。

プログラムの本質は、データの流れと制御の流れである。しかし、制御の流れは、一般に IF 文などの条件文などを除いては逐次的な処理順序が定められており、プログラムの並列性抽出の観点から考えると重要な要因ではない。つまり、プログラムの並列性を決定づける要因は、データの流れと条件文などにおける制御の流れと考えることができる。これらは、データ依存や制御依存として表現することができる。

そこで、我々は、プログラム要素間の依存関係を効果的に表現可能な依存グラフを用いてプログラムの並列性抽出処理、すなわち、プログラム分割処理をモデル化する。依存グラフとして、ループ・ブロックなどを階層で表現する階層型依存グラフを用いることで、制御の流れの情報なしにプログラム全体の構造を表現することができる。例えば、図 1のように、ループボディ内の依存関係（配列 C）はループブロック内にサブグラフとして持ち、ループ間の依存関係（配列 A,B）はループブロック要素間の依存関係として抽出される。

プログラムの並列性は、依存グラフではノードの独立性で表現される。すなわち、互いに独立なノードが並列処理することができる。この依存グラフにおける並列性の表現を利用するために、我々は依存グラフのノードを

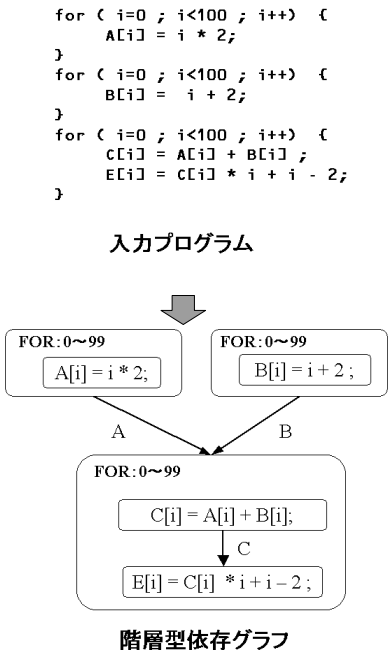

図 1: プログラム・モデル

最小の並列処理単位と定義し、プログラム分割処理における分割プログラム片を表現する。より良い並列性をより抽出するには、依存グラフ上で互いに独立なノードが多くなるように依存グラフを再構成する必要がある。この再構成処理がプログラム分割処理に当り、我々はプログラム分割処理を依存グラフの再構成処理としてモデル化する。図 2下のように再構成することによりノード間の並列性が抽出される。この再構成後のノードを並列タスクと考え、プログラム分割の単位とする。

2.2　プログラム分割戦略のモデル化

本節では、前節でモデル化された依存グラフに基づくプログラム分割処理に対して、プログラム分割戦略をモデル化し、その表現法を提案する。

一般に、プログラム分割処理は以下のサイクルを繰り返し処理することで、実現される。

1. プログラムを分割することでプログラムの並列性を抽出することができる箇所を決定する。

2. 分割箇所において、分割の手法を決定する。

以上のような処理を依存グラフに適用することで依存グラフが変形され、並列処理の最小単位を生成することができる。このプログラム分割処理を依存グラフの再構成

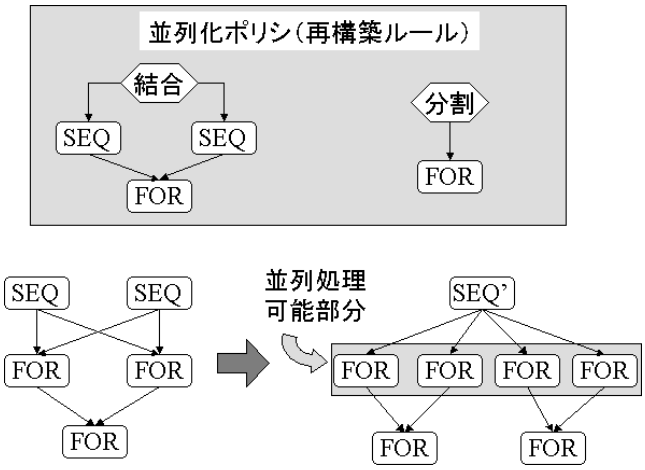

図 2: 依存グラフの変形による並列化

処理として捉えると、一番目の処理は依存グラフ中の再構成箇所を特定するために依存グラフから部分グラフを抽出する処理としてモデル化でき、二番目の処理は抽出した部分グラフの再構成処理としてモデル化することができる。この部分グラフの抽出と再構成処理の適用を決定するのがプログラム分割戦略であり、並列化ポリシと呼ぶ。例えば、図 2上のように、ループ文前の逐次要素を結合したり、ループ文を分割したりすることをグラフパターンとその変形パターンで表現することができる。

依存グラフから部分グラフを抽出する方法として、我々は部分グラフの位相的な特徴を表す部分パターンを用いる。依存グラフを部分パターンのグラフ・マッチング処理により再構成処理を適用する部分を決定する。例えば図 2右上に示す並列化ポリシでは、 FOR ブロックが依存する基本ブロックが 2 つ存在する部分パターンを表している。ここで、部分パターンを構成するノードとエッジには属性を設け、条件を記述することが可能となっている。条件により、再構成の対象となる部分グラフを適切に、かつ、正確に表現することができる。それぞれの属性を以下に示す。

- ノードの属性

  プログラム要素の特性、すなわち、 FOR 文、代入文 (逐次文) であるといったプログラム要素タイプや、実行予測ステップ数、ループ文においては反復回数やどれだけ先のイタレーションへ依存しているかを表すイタレーション依存距離などを表現する。

- エッジの属性

  依存関係の特性を表現する。データ依存においては、依存データ量やイタレーション依存距離、制御依存においては、分岐条件の真偽が表現される。

次に、再構成の方法について考える。依存グラフの再構成では、プログラムの意味が変更しないように注意する必要がある。我々は、基本操作として、複数のノードを合わせて一つのノードにすることで粗い粒度のノードへと変形する**結合**、一つのノードを複数のノードに分けることで細かなノードへと変形する**分割**を定義する。分割はその方法が様々であり、この分割の違いが並列化手法の違いにつながるため、大きな役割を占める。我々は、基本的な方法として文の集合を処理量が均等になるように分割する以外に、ループ文を方向（イタレーション方向、文方向）、分配（Block 分配、 Cyclic 分配）による分割を定義する。これらの操作はプログラム要素のまとまり方を変更するものであり、プログラムの意味変更は生じない。また、一つのプログラム要素を複数の並列処理単位へ含めることを可能とするために**複製**を、複製の結果として必要のなくなったノードを取り除くために**消去**を補助操作として定義する。複製は処理の実行箇所が複数に増加するだけであり、プログラムの意味を変更する操作ではないが、消去はプログラムの一部を取り除くため、プログラムの意味が変更する可能性がある。そのため、並列化ポリシを記述するときに消去を用いる必要があるときは、プログラムの意味が変更されないことを保証する必要がある。現在、その保証を並列化ポリシ作成者で行うこととしており、そのため、消去の適用には注意が必要である。

上記のことから、並列化ポリシを条件付きグラフパターンと再構成処理パターンとして表現できる。また、この基本的な並列化ポリシを組み合わせることで、任意の並列化手法を表現していくことが可能となる。

具体的に表現した並列化ポリシの例を図 3に示す。これは、 Doall 並列化手法を表現している。図 3左の並列化ポリシは、ループ運搬依存のないループ文を適度な粒度まで分割することを意味する。そのため、条件部の FOR ブロックには、 (1) 分割対象となる粒度が粗いことを示すために実行ステップ数が一定以上であることと、(2) ループ運搬依存がないループ文を示すためにイタレーション依存距離が 0 である (イタレーションを依存関係がない) ことを属性の条件として付加する。そして、操

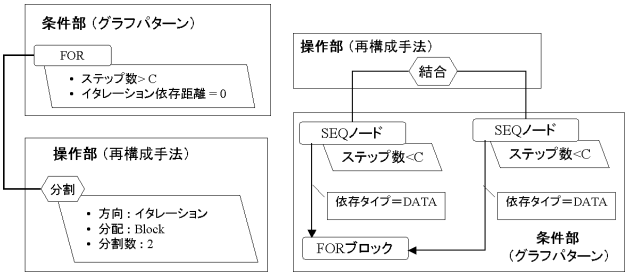

図 3: 並列化ポリシの例～ Doall 並列化～

作部の分割には、 Doall 並列化手法の分割法である (1) イタレーション方向に (2)Block 分割で (3) 半分ずつに分割することをその操作属性として付加する。ここで、分割数を 2 としたのは、操作を繰り返すことで適度な粒度へ変形可能となるための最小の単位だからである。図 3右の並列化ポリシは、あるループ文の前にある細粒度のノードを結合することで細かすぎる粒度のノードをできるだけ無くすこと意味する。これらも同様に、各部に適用の条件などが記述される。結合操作においては、動作が一定であるために操作属性を付加する必要がない。これらの並列化ポリシを適用することで、 Doall 並列化手法が実現される。同様に、各並列化戦略を並列化ポリシとして表現し適用することで、様々な並列化戦略を組み合わせたプログラム分割機構が実現される。

3　依存グラフ変形処理による弊害

前章で、依存グラフに基づく並列化処理のモデル化と並列戦略の表現を述べた。この機構では、並列化ポリシに基づいて依存グラフを任意の形に再構成することが可能となる。しかし、依存グラフの再構成でプログラムの意味を変更しないように、プログラム全体の依存関係を保持しなければならない。したがって、再構成処理の適用時には、再構成により依存グラフの一貫性、正確性が変化しないかを常に監視する必要がある。本章では、依存グラフの一貫性を保持するための LOOP 依存関係の解決について述べる。

LOOP 依存関係は、各ノードが互いに依存するノードの終了を待機し合う状態、すなわち、デッドロック状態を表す依存関係である。この LOOP 依存関係は、ノードを結合したときや、循環依存にあるループ文を分割したときに生じる。 LOOP 依存関係を解消するためには、

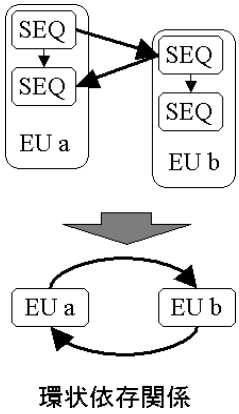

図 4: LOOP 依存関係

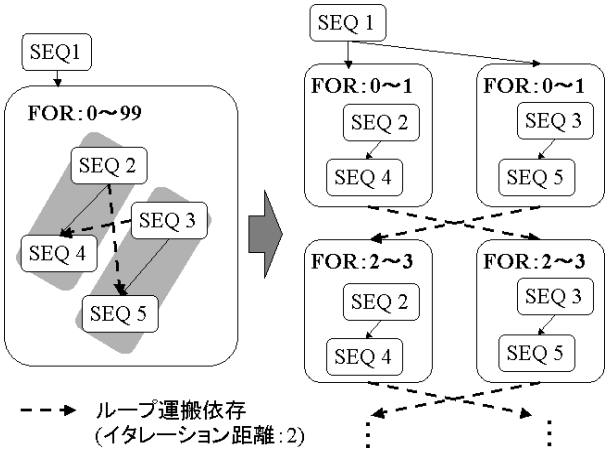

図 5: LOOP 依存関係が存在する場合のループブロックの分割法

ノードを更に分割する必要がある。図 4は、 LOOP 依存関係の一例である。ノード EU a, EU b は、その内部要素の依存関係から、互いに他への依存関係を保持する。これにより、互いの終了を待機し合う状態、デッドロック状態となる。しかし、ノード EU a 内の要素を二つに分割することで、この関係を解消することができる。

循環ループ文を分割した場合は、イタレーション依存距離でループ文を分割すれば LOOP 依存関係は解消される。例えば、図 5のように、イタレーション依存距離で分割することにより、分割された各 FOR 文が最初のイタレーションを含むノードから順に実行することが可能となる。

ノードの結合により生じた LOOP 依存関係は、 LOOP 依存関係を構成しているノードに対して適した分割をする必要がある。このときの分割ノード選択の方針

を以下に示す。

1. 多くの LOOP 依存関係を解消可能なノードを分割。

2. 処理量の少ない単位を生成しないノードを分割。

また、ノード内の分割方針を以下似示す。

1. 他のノードに対して、実行可能となる条件、および、他の処理単位の実行可能性への影響が異なる部分をまとめた集合へ分割する。しかし、まとめて実行しても LOOP 依存関係が生じない部分同士はまとめたままとする。

2. 他のノードと依存しない要素は、自ノード内の要素に依存されるノードのなかで一番最初に行われる部分と統合する。そのようなノードがない場合は、最初に実行可能となるノードと結合する。

これをアルゴリズム化すると次のようになる。

1. ノード n 内の LOOP 依存関係を構成するノード集合 N を求める。

2. $n \in N$ において、以下の操作を繰り返す。

   (a) 各 LOOP 依存関係 i において他のノードから依存されているプログラム要素集合 $OUT_{n,i}$ と他のノードへ依存しているプログラム要素集合 $IN_{n,i}$ を計算する。このとき、$OUT_{n,i}, IN_{n,i}$ の両方に含まれる要素は互いの集合から除く。

   (b) 分割基準を構成するプログラム要素集合を計算する。
   $ELEMENT \leftarrow \bigcup_i \{OUT_{n,i} \cup IN_{n,i}\}$

   (c) $OUT_{n,i} = \phi$ 、または、 $IN_{n,i} = \phi$ なる i は今後の計算の対象外とする。

   (d) 新たなノード候補を計算する。
   $UNIT$=$ELEMENT - \{\cup_i IN_{n,i}\}$

   (e) ノード候補集合の要素を各情報から取り除く。
   $ELEMENT \rightarrow ELEMENT - UNIT$、
   有効な計算対象iに対して、 $OUT_{n,i} = OUT_{n,i} - UNIT$

   (f) $\exists i$ $OUT_{n,i} = \phi$ なる i は今後の計算の対象外とする。このとき、LOOP 依存関係解消数カウントを１つ増やす。

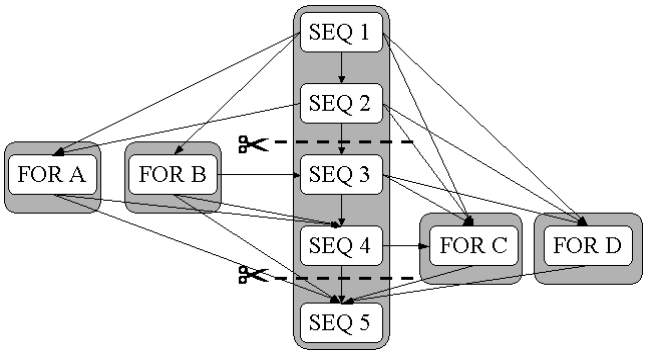

図 6: LOOP 依存関係が存在する場合の文集合ノードの分割法

   (g) 計算対象 i が存在するならステップ 2-(d) へ。

   (h) 残りのノード内の要素を対応した分割候補集合へ分散させる。

3. ステップ 1,2を繰り返し、全てのノードに対して、分割候補集合を計算する。

4. ステップ 3までによって計算算された分割候補のうち、最も LOOP 依存関係解消数が大きく、小さな処理単位を最も生成しないノードを選択し、分割することで、 LOOP 依存を解消する。

5. 全ての LOOP 依存関係が解消されるまで以上のステップを繰り返す。

図 6に、文が結合されたノードにおける分割箇所決定の例を示す。この依存グラフでは、各 FOR 文に対して LOOP 依存関係が構成されている。中央の逐次文が結合されたノードに注目して、分割点の決定手順を説明する。各ノードに対して、逐次ブロックの IN、OUT の関係と ELEMENT は次の通りとなる。

$ELEMENT = \{SEQ1, SEQ2, SEQ3, SEQ4, SEQ5\}$
$OUT_A = \{SEQ1, SEQ2\}, IN_A = \{SEQ4, SEQ5\}$
$OUT_B = \{SEQ1\}, IN_B = \{SEQ3, SEQ4, SEQ5\}$
$OUT_C = \{SEQ1, SEQ2, SEQ3, SEQ4\}, IN_C = \{SEQ5\}$
$OUT_D = \{SEQ1, SEQ2, SEQ3\}, IN_D = \{SEQ5\}$

これにより、最初の新たなノード候補 $UNIT = \{SEQ1, SEQ2\}$ が決定され、各値が以下のように更新される。

$ELEMENT = \{SEQ3, SEQ4, SEQ5\}$
$OUT_A = \{\} = \phi, IN_A = \{SEQ4, SEQ5\}$
$OUT_B = \{\} = \phi, IN_B = \{SEQ3, SEQ4, SEQ5\}$
$OUT_C = \{SEQ3, SEQ4\}, IN_C = \{SEQ5\}$
$OUT_D = \{SEQ3\}, IN_D = \{SEQ5\}$

FOR A,FOR B 文に対しては、IN 集合がないため、今後のアルゴリズムから対象外となる。同様に求めていくと、$\{SEQ1, SEQ2\}, \{SEQ3, SEQ4\}, \{SEQ5\}$ という分割集合群が求められる。各 FOR 文ノードは分割不可能なため、逐次文集合ノードを先の集合に分割する。

このように、本アルゴリズムで分割方針に従い、プログラム分割戦略の意図への影響が少ない LOOP 依存関係の解消が可能となる。つまり、実行不可能に構成された並列処理単位を、構成戦略の意図をなるべく残した実行可能な処理単位へ変換することができる。これにより、依存グラフの一貫性を保持することが可能となる。

4 おわりに

本稿では、様々な並列化戦略を反映可能な並列処理機構を提供するために、依存グラフの再構成に基づくプログラム並列化手法と、並列化戦略の知識化表現について述べた。本手法では、プログラムの意味の保持を依存グラフの再構成として基本的に保証し、再構成ルールを自由に構成することを可能にすることで、任意の並列化処理を表現可能にしている。我々はこの再構成ルールを並列化ポリシと呼び、コンパイラ開発者が新たな並列化戦略を並列化ポリシとして宣言したり、プログラムに合わせた並列化ポリシの組合わせを構成することで、容易に環境や応用プログラムに適した並列化処理機構を構築することが可能となる。

再構成法によっては、LOOP 依存関係と呼ばれるデッドロックの原因を起こす依存関係も生じるという問題がある。しかし、これらは一定の基準により並列処理単位を更に細分化することにより、LOOP 依存関係を解消しデッドロックを回避することが可能となる。このための基準として、なるべく並列戦略の意図を壊さず、より適した並列処理が可能なように基準を設定した。これにより、依存グラフの再構成における並列化手法の不十分性に対して対処し、並列化プログラムの生成が可能となる。

現在の並列化処理機構は、階層間に跨がる再構成ルールの表現を定義しておらず、ループ文が入れ子になっている場合にのみ適用するルールなどの表現が不十分である。今後、並列化ポリシの表現の拡張が必要となる。

参考文献

[1] 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, Vol. J73-D-I, No. 12, pp. 951–960 (1990).

[2] 笠原博徳, 小幡元樹, 石坂一久: “共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理”, 情報処理学会論文誌, Vol. 42, No. 4, pp. 910–920 (2001), [in Japanese].

[3] 朝倉宏一, 渡辺豊英: “並列化コンパイラ構築のためのツールキットの構成”, 信学技法 COMP, Vol. 100, No. 144, pp. 97–104 (2000).

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam: “*PVM 3 Users Guide and Reference manual*”, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831 (1994).

[5] Jack J. Dongarra, Steve W. Otto, Marc Snir and David Walker: “A message passing standard for MPP and workstations”, *Communications of the ACM*, Vol. 39, No. 7, pp. 84–90 (1996).

[6] http://www.openmp.org/.

[7] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam and John L. Hennessy: “SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers”, *SIGPLAN Notices*, Vol. 29, No. 12, pp. 31–37 (1994).