ハイパフォーマンス
コンピューティング 82−13
（2000. 8. 4）

OpenMP並列プログラムのデータフロー解析手法

佐藤 茂久、 草野 和寛、 佐藤 三久
新情報処理開発機構 つくば研究センタ
{sh-sato, kusano, msato}@trc.rwcp.or.jp

本論文では、OpenMP API を用いて記述された共有メモリ並列プログラムの、スレッド間のデータの流れも考慮したデータフロー解析の方法を述べる。従来の逐次プログラムのデータフロー解析の枠組を、スレッド間の相互作用を反映できるように拡張する。そのために、スレッド内とスレッド間の双方のデータの流れを表現できる中間表現を用いる。さらに、データフローの変換関数に OpenMP の指示文と指示節の意味を反映させる。データフロー解析の具体例として、到達定義の解法を示す。OpenMP の構造的な並列性記述と緩いメモリコンシステンシモデルのために、効率が良く精度の良い解析が可能になる。このような方法を用いることで、これまでの OpenMP の実装では行なわれていなかったスレッド間の相互作用を考慮したプログラム解析と、それを利用した最適化が行なえるようになる。

Dataflow Analysis Techiniques for OpenMP Programs

Shigehisa Satoh, Kazuhiro Kusano, and Mitsuhisa Sato
Tsukuba Research Center, Real World Computing Partnership

In this paper, we present dataflow analysis techniques for shared-memory parallel programs using OpenMP API. We extend traditional dataflow analysis frameworks so as to take account of interaction between threads. We designed internal representation which models both intratherad and interthread flow of data. Transfer functions in the frameworks are defined reflecting semantics of the OpenMP directives and clauses. We also show a reaching definions framework for OpenMP programs as an example. Strucutured parallelism and relaxed memory consistency make such analyses efficient and effective. Our dataflow analysis techniques enable program analyses and aggressive optimizations for multithreaded programs.

1. はじめに

近年、低価格のPCからスーパーコンピュータまで、共有アドレス空間を持つ並列計算機が普及してきている。ここで言う共有アドレス空間には、対称マルチプロセッサ (SMP) の共有メモリだけでなく、CC-NUMA やソフトウェア DSM のような、分散共有メモリも含めている。これらのシステムでは、物理的には共有あるいは分散されているメモリを、一つの共有アドレス空間を通して全てのスレッドで論理的には同じように利用できる。そのため、分散メモリ上のメッセージパッシングモデルよりもプログラミングの容易な共有メモリモデルを用いてプログラムを記述できる。共有メモリプログラミングの API としては、従来 POSIX スレッドのようなマルチスレッドライブラリが多く用いられてきたが、指示文で並列性を記述する OpenMP API[2] が今後普及するものと期待されている。現在、OpenMP は SMP や CC-NUMA で主に用いられ、ソフトウェア DSM を基盤としたクラスタ向けの実装も研究されている[5)7)]。

OpenMP を用いる事により、主としてデータ並列なアプリケーションのプログラミングを容易に行なえる。しかし、論理的には共有アドレス空間が利用できるとは言え、物理的なメモリの構成や、データ共有・同期のオーバーヘッドはプラットホームにより様々であり、高性能を得るためにはプラットホームの特徴に応じたプログラミングが必要になる。それをプログラマがすべて行なうのは負担が大きい上に、現在の OpenMP の仕様では同期やデータ配置などをプログラマが詳細に制御する事は不可能である。そのため、OpenMP の実装、即ちコンパイラと実行時システムによって何らかの最適化を行なう事が望ましい。コンパイラで効果的な最適化を行なうためには、個々のスレッド内のデータフロー解析だけでなく、スレッド間のデータの流れやデータ依存関係も考慮した解析も必要となる。

我々はすでに、SMP、CC-NUMA、コンパイラ支援ソフトウェア DSM などで利用できる OpenMP プログラムの最適化手法を提案している[5)6)]。本論文では、これらの最適化の基礎となる、OpenMP プログラムのスレッド間の相互作用を考慮したデータフロー解析 (並列データフロー解析) の手法について述べる。まず 2 節で、このようなデータフロー解析を行なう上で有益な OpenMP の特徴を、他のマルチスレッドプログラムと比較しながら述べる。次に 3 節で、最適化に有用な並列データフロー情報について考察する。4 節では、OpenMP プログラムの中間表現を定義し、スカラ変数に対するデータフロー解析の枠組を示し、OpenMP の指示文や指示節の意味をどのように反映させるかを述べる。5 節では具体的なデータフロー問題の解法として、到達定義の解法を述べる。最後に、まとめと今後の課題を述べる。

2. OpenMP の利点

最適化コンパイラから見たOpenMP の重要な特徴として、構造的な並列性記述と、緩いメモリコンシステンシモデルがあげられる。これらはいずれもコンパイラによる解析・最適化を容易にする。この節では、POSIX スレッドや Java と比較しながら、これらのOpenMP の利点を説明する。

OpenMP を用いた並列プログラミングは、ループの並列処理を中心とした科学技術計算に適しており、そのようなプログラムの並列性を構造的に記述できる。即ち、並列実行される部分は `parallel` 構文を用いて明示され、共有可能なデータとそうでないものはデータ属性により区別される。さらに、ワークシェアリング構文によって、各スレッドがどのように処理を分担するかも明示される。このような構造的な記述法により、並列プログラムの構造をコンパイラが容易に認識でき、精度の良い解析が可能となる。

POSIX スレッドや Java を用いた並列プログラムでは、OpenMP と異なり並列性を非構造的に記述しているため、コンパイラがプログラムの構造を認識しにくく、最適化には不利である。また、組み込みのバリア同期を持たないために、同期間の順序関係が精度良く検出できないという問題もある。それでも Java では冗長な同期が頻発するため、不要な同期の削除などの並列性を考慮した最適化が行なわれている [4]。しかし、スレッド間で共有されるデータに対する最適化は限られており、科学技術計算で重要なデータ並列処理に対する効果的な解析・最適化はまだ難しい。そこで、Java 向けの OpenMP を元にした並列プログラミング API も提案されている [1]。これは主としてデータ並列処理の容易な記述を意図した提案であるが、最適化を行ないやすいという利点もある。

最近の共有メモリマルチプロセッサでは、メモリアクセスを効率的に行なうために、緩いメモリコンシステンシモデルを採用することが多くなっている。緩いメモリコンシステンシモデルは、最適化コンパイラにとっても有益である。利点は二つある。第一に、メモリ同期点以外ではスレッド間の相互作用を考慮せずに済むため、メモリ同期点の間のコードでは、その中に共有データの参照があったとしても、逐次プログラムと同様の最適化を行なえる。第二に、本論文で述べるようなスレッド間の相互作用を考慮した解析を行なう際に、メモリ同期点でのみスレッド間にまたがったデータの流れを考慮すれば良いため、全てのプログラム点で相互作用を考慮する必要がある強いメモリコンシステンシモデルに比べて、解析コストが軽減される。そのため、メモリ同期点を越えた、あるいはスレッド間にまたがった最適化も行ない易くなる。

OpenMP では、`flush` 操作によってメモリが同期される緩いメモリコンシステンシモデルを採用しているため、上記の利点を持つ。POSIX スレッドでも同期プリミティブなど一部の関数の呼出によってのみメモリの同期が保証される。しかし、メモリ同期専用の操作は持たないため、共有データに `volatile` 修飾子を付けることも多く、最適化が行ないにくくなる。Java では、メモリモデルの仕様が最適化に適したものではなかったため、メモリモデルを守らない最適化が多く行なわれている。そのため、より現実的なメモリモデルの仕様が検討されている [3]。

3. 最適化に有効な並列データフロー解析

本論文の並列データフロー解析は最適化で利用する事を目的としている。そこで、この節では最適化に有効な並列データフロー情報について検討する。

OpenMP のような並列プログラム向けの最適化はまだ一般的でないが、性能への効果の高い最適化には次のようなものが考えられる。

- 制御同期の最適化
- メモリ同期の最適化
- データ配置の最適化

以下、それぞれの最適化の対象とする問題と、それに必要なデータフロー情報について考察する。

制御同期の最適化

制御同期とは、バリア同期やロックを用いた排他制御のような、複数のスレッドの制御フローを同期させる操作のことである。このような操作を頻繁に行なえば、プログラムの並列性が失われ、実行効率が低下する。制御同期は、スレッド間のデータの流れを制御するために行なわれるため、データの流れに影響のない制御同期は冗長といえる。スレッド間のデータ依存関係を元に最小限必要なデータ同期を求めることで、冗長な制御同期の検出・削除や、クリティカル・セクションの縮小などの最適化が行なえる。

メモリ同期の最適化

メモリ同期 (データ同期) とは、スレッド間、あるいはスレッドと仮想的な共有アドレス空間の間で、それらから見える (ことが保証される) 共有データの値を同期することである。プログラムを正しく実行するためには、メモリコンシステンシモデルに応じて定められたメモリ同期点において、共有データが適切に同期されることが必要である。メモリ同期点でコンパイラは、適切な同期命令 (フェンス、メモリバリアなどと呼ばれる) の生成や、同期点をまたいだコード移動やレジスタ割り付けの抑制などによって、スレッド間や、レジスタ・キャッシュ・主記憶などのメモリ階層の間でデータの一貫性を保証する必要がある。そのため、共有データに対するメモリ同期が頻繁に発生すると、メモリ階層を有効に利用できなくなったり、命令実行の逐次化やプロセッサ間の通信が発生するなど、メモリ

```
#include <math.h>
double sub(double *a, double *b, int n) {
  double s = 0.0;
#pragma omp parallel
  {
    int i;
    double tmp;
#pragma omp for
    for (i = 1; i < n; i++) {
      a[i] = (b[i] - b[i-1])/2.0;
      tmp = fabs(a[i]-b[i]);
#pragma omp critical
        if (tmp > s) s = tmp;
    }
  }
  return s;
}
```

図1 OpenMP プログラムの例

同期のオーバーヘッドにより実行性能が低下する。実際の OpenMP プログラムでは、同期すべきデータを指定できない暗黙の flush 操作で必要以上にデータの同期が行なわれる。そこで、プログラムの意味を変えない必要最小限のメモリ同期操作を行なうことで、メモリ同期のオーバーヘッドを削減することができる。

データ配置の最適化

メモリ階層を有効利用するためには、データ参照の局所性が高いことが必要である。そのためには、共有データがそれを参照するスレッドの近くに配置されることや、フォルス・シェアリングが少ないことが望まれる。コンパイラが共有データの参照パターンを解析することで、データ配置を改善することができる。データ配置の最適化は、並列ループで参照される配列や、動的に生成されるプライベートデータで重要である。

図1のプログラムを例にメモリ同期の性能に及ぼす影響を説明する。この関数は排他制御による逐次化のためにスケーラビリティが低いであろうこと以外は、一見問題のなさそうなプログラムである。しかし、このプログラムでは冗長なデータ同期が行なわれる。並列領域内で参照されるスカラの共有データは、n, s, a, b の四つである。このうち n は、スレッドの担当範囲を求めるために並列ループの開始時に一度参照されるだけである。クリティカルセクション内で参照される変数 s は他のスレッドが書き換える可能性があるので、毎回メモリ同期を行なう必要がある。しかし、並列領域内で書き換えられることのないポインタ a と b も、共有データであるため critical 構文の入口と出口で毎回暗黙に同期されてしまう。すると、a,b をレジスタに載せたまま繰り返し利用することはできず、値が変わっていないにも関わらず、毎回メモリからロードする必要が生じる。そこで、コンパイラでこれらのポインタの値が (並列) ループ不変であることを検出し、それをレジスタに割り付けられれば、ループ中で a, b の値を毎回ロードする必要がなくなる。

コンパイラで最適化できなくても、ポインタ a,b を firstprivate 指示節でプライベート化することでも冗長なメモリ同期を回避できる。しかし、プログラマはこのような問題を見落としがちである。そこで、冗長なメモリ同期を行なわないようにコンパイラで最適化を行なうことは有益と考えられる。この例では並列領域内で a と b が書き換えられないことだけわかれば上記のような最適化は可能である。しかし、本論文のデータフロー解析手法では、並列領域内で書き換えられるデータについてもできるだけ正確なデータフローを求めることを目的としている。文献[5)6)]には、OpenMP プログラムに対するその他の最適化の例も示されている。

以上のように、並列領域内での書換えの有無やスレッド間のデータ依存関係のような逐次プログラムで解析していたのと類似の情報や、メモリ同期点で同期する必要のある変数のような並列プログラム特有の情報の解析が最適化で有効である。

4. 並列データフロー解析の枠組

逐次プログラムのデータフロー解析では、多くの問題が単調データフロー解析の枠組を用いてモデル化できる。この節では、並列プログラムに対する同様の枠組を提案する。

実際のプログラムを解析する場合には、プログラム全体を対象に手続き間解析を行なうことが望ましい。しかし、紙面の都合で以下では手続き呼び出しの扱い方については明記しない。また、本論文では共有されるスカラ変数に対するデータフロー解析のみを考える。

4.1 並列フローグラフ

OpenMP プログラムの制御構造を表す中間表現として、逐次プログラムの制御フローグラフの代わりに並列フローグラフを定義する。

並列フローグラフ $PFG = (N, E)$ はノードの集合 N とエッジの集合 E からなる有向グラフである。

ノードの集合 N は以下のノードから構成される。

プログラムの始点と終点: PFG の表すプログラムの入口と出口をそれぞれノード s と e で表す。

逐次ノード: OpenMP 指示文を含まない実行文からなる基本ブロックを表す。

指示文ノード: プログラム中の個々の OpenMP 指示文を表す。ただし、指示文とその作用を受けるブロックが組になって構文 (construct) として使用される指示文では、その構文の入口と出口をそれぞれ一つの指示文ノードで表す。

指示文ノードの中で、メモリの同期を伴うノードを特に同期ノードと呼ぶ。同期ノードには、flush 指示文、critical 指示文の入口と出口などが含まれる。また、barrier 指示文を表すノードや、nowait 節の

ないワークシェアリング構文の出口を表すノードはバリア同期を伴う。

エッジの集合 E は、制御エッジと同期エッジから構成される。制御エッジは単一スレッド内での制御の流れを表す。逐次ノード間の制御エッジの張り方は逐次プログラムに準じる。指示文ノードのうち、ワークシェアリング構文以外については逐次実行するのと同様に制御エッジを張る。ワークシェアリング構文に対しては、次のように制御エッジを張る。**single** 構文では、構文内のブロックを逐次実行するのと同じようにエッジを張る。実際にはただ一つのスレッドしかブロック内は実行しないが、ブロックを迂回するエッジは張らない。これは、**single** 構文内の代入でデータの流れをキルできるようにするためである。**sections** 構文では、構文の入口ノードから各 **section** 構文の入口への多方向分岐のようにエッジを張る。また、各 **section** 構文の出口から **sections** 構文への出口へのエッジを張る。**for** 構文に対しては、構文の入口からループ本体の入口と、ループ本体の出口から構文の出口へのエッジを張り、バックエッジは張らない。**sections** 構文や **for** 構文では、明示的な同期がない限りセクション間あるいはループの反復間のデータフローは考慮しなくて良いため、制御エッジでサイクルを作ることはない。

同期エッジは同じあるいは異なるスレッド間での同期ノードの実行順序の制約を表し、同期ノード間に張られる。同期ノード n から m への同期エッジがある時、あるスレッドが n を実行した後、同じあるいは別のスレッドが他の同期ノードを実行せずに m を実行する可能性があることを表している。

同期エッジは次のようにして作成する。まず、各同期ノードから、制御エッジだけをたどり、他の同期ノードを通らずに到達可能な同期ノードへの同期エッジを張る。さらに、バリアでない同期ノードから制御エッジを順方向または逆方向にたどり、バリア同期を通らずに到達可能なバリアでない同期ノードへの同期エッジを張る。ただし、同じラベルの付いた **critical** 構文の入口同士・出口同士には同期エッジを張らない。

同期エッジによって同期点間の半順序関係が定義される。同期ノードの数を n とすると、同期エッジの数は最大で n^2 となる。しかし、実際の OpenMP プログラムの多くはバリア同期の割合が高く、同期の発生順序の制約が多くなる。そのため、同期エッジの数も n^2 よりもずっと少ないと考えられる。緩いメモリコンシステンシモデルとバリア同期の存在により、スレッド間の相互作用を考慮しても、それによる解析コストの増加はシーケンシャルコンシステンシに従うプログラムやバリア同期を持たないプログラムの解析に比べて、効率良く解析できると考えられる。

図 2 は、図 1 のプログラムの並列フローグラフを示す。ただし、図 1 の関数は必ず逐次実行部で実行される (つまり他のスレッドは実行されていない) ものとして、この関数以外は省略する。図では円で書いてあるノードがプログラムの始点 s と終点 e である。四角いノードは逐次ノード、楕円のノードは指示文ノードを表す。各ノードの左にある数字はノード番号である。ノード 9,10,11 はそれぞれの構文の出口ノードである。直線の矢印が制御エッジであり、点線の矢印が同期エッジである。太線で書いてあるノード、即ち **parallel** 構文と **critical** 構文の入口と出口、**for** 構文の出口はメモリ同期を伴う同期ノードである。**for** 構文は逐次ループのようにバックエッジは作られず、ループ制御変数の上下限と増分は入口ノードに仮想的な参照点として付加される。

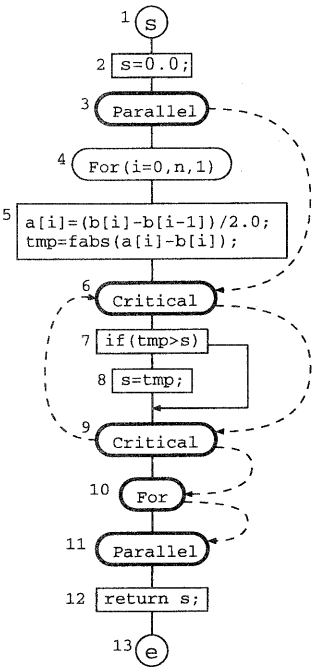

図 2 並列フローグラフ

4.2 データフロー集合

データフロー解析の枠組を用いてモデル化するには、解析したいデータフロー情報に応じて、データフロー集合とそれに対する演算として、データフロー集合の束 $L = (S, \sqcap, \sqcup)$ を定義する。スカラの共有変数に対して、逐次プログラムの場合と同様にデータフロー集合を定義する。

データフロー情報の集合 S は、解析しようとするデータフロー情報の全体を表す。meet 演算子 $\sqcap$ と join 演算子 $\sqcup$ はそれぞれ S 上の二項演算であり、これらの演算は S 上で閉じている。データフロー集合 S には、最大要素 $\top$ と最小要素 $\bot$ が含まれる。このようなデータフロー集合の考え方は、逐次プログラムの場

合と同様である。

4.3 変換関数

プログラムのデータフローへの作用を表すために変換関数の集合を定義する必要がある。データフロー集合 S から S への単調関数全体からなる関数空間を $F : S \to S$ とする。並列フローグラフの各ノードから変換関数への写像を $f : N \to F$ とする。特に F に属する関数が全て分配的であるとき、そのデータフロー問題は、反復法によって正確な解を得られる。また、単調性は反復法が有限回で収束することを保証する。

この f は求めるデータフロー情報の性質に応じて定義する必要がある。その際、OpenMP の指示文・指示節をどのように扱うかを以下に述べる。

まず、変数の属性は以下のように反映させる。

プライベート変数: `private` 節, `firstprivate` 節, あるいは `lastprivate` 節で指定された変数は、それが指定された構文内でプライベート化される。そのため、その構文内でのその変数の参照は、共有変数の参照とはみなさない。ただし、`firstprivate` で指定された変数は、構文の入口で元の共有変数の仮想的な使用点があるものとする。逆に、`lastprivate` で指定された変数は、構文の出口で元の共有変数の仮想的な定義点があるものとする。これらの場合以外は、構文の入口および出口で元の共有変数の情報はキルされる。即ち、構文を越えて情報が伝播されることはない。

スレッドプライベート変数: `threadprivate` 属性を持つ変数は、各並列領域でプライベート化される。しかし、ある条件の元では、プライベート版のデータが複数の並列領域をまたがって利用できる。ここでは、このような場合の精度良い解析は考えず、並列領域の入口と出口にそれぞれ仮想的な使用点と定義点があるものとし、並列領域内の参照は解析対象としない。従って、`copyin` 節の有無は解析に影響しない。

リダクション変数: `reduction` 節で指定された変数は、その構文の入口と出口にそれぞれ仮想使用点と仮想定義点を設け、構文内の参照はプライベート変数と同様に解析対象としない。

共有変数: 並列領域内で共有変数として参照される変数は、逐次部の対応する変数と同一の変数として扱う。

次に各構文毎の扱い方を述べる。

`parallel` 構文: 並列領域の入口と出口を表すノードは、暗黙のメモリ同期とバリア同期を伴うものとして扱う。`if` 節が付いている時は、その中の式で参照される共有データの仮想使用点を入口のノードに置く。

`for` 構文: `for` 構文で表される並列ループは、ループ内で明示的な同期を行なわなければ DOALL 型のループである。そのため、バックエッジは張られない。`for` 構文のループの上下限および増分の式に現れる共有変数は、構文の入口を表すノードに仮想使用点を設ける。ここで述べる枠組では `ordered` および `schedule` 節の意味は考慮しない (安全な近似解を求める)。しかし、`schedule` 節のチャンクサイズの指定に共有変数が現れる時は、その変数の仮想使用点を構文の入口におく。

`sections` 構文: 構文の入口では、各 `section` への多方向の条件分岐のように扱う。しかし、構文の出口では、逐次制御フローの合流とは異なり、`section` の出口のいずれかに必ず行なわれる定義は、すべてのパスで定義があるものとみなす。

`flush` 指示文: 同期する変数が明示されている場合、その変数の仮想使用点と仮想定義点を設ける。

その他の指示文を表すノードの変換関数はデータフローに影響しない。また、各構文内の逐次ノードでは、そのノード内の共有データの参照に対して逐次プログラムと同様に変換関数を定義する。

ワークシェアリング構文 (`for`, `sections`, `single`) と `master` 構文では、ある共有変数をいずれか一つのスレッドが必ず定義するならば、その構文の出口で全てのスレッドでデータの流れをキルする、あるいはその定義が必ず出口に到達するものとする。これは、OpenMP プログラムではデータレースがないという仮定に基づいている。その定義が全スレッドから見えるようになるまでに、その変数の値を参照するスレッドがあるなら、データレースが生じていることになるからである。また、これらの構文の出口に同一変数の異なる定義が必ず到達する場合も、データレースが生じていると考えられる。

5. 到達定義の計算

前節で述べた枠組を用いたモデル化の例として、到達定義の解析法を示す。ここでは並列フローグラフの各ノードに到達する可能性のある定義の集合を求める事とする。

データフロー集合 S は、定義点の集合のべき集合となる。meet 演算および join 演算はそれぞれ集合の積 $\cap$ と和 $\cup$ である。S の最大要素 $\top$ は空集合であり、最小要素 $\bot$ は定義点全体の集合である。

並列フローグラフの各ノード n の変換関数を f_n とすると、逐次プログラムと同様に f_n は以下のように定義できる。

$$f_n(V) = Gen(n) \cup (V - Kill(n))$$

ここで、V はノード n の入力となるデータフロー集合であり、$Gen(n)$ と $Kill(n)$ はノード n の意味に応じて定められる。$Gen(n)$ はノード n で生成される定義の集合であり、$Kill(n)$ は n を通り越すことの出来ない (n でキルされる) 定義の集合である。

プログラム中の到達定義を求めるためには、次の

n	Gen(n)	Kill(n)	In(n)	Out(n)
1	a,b,n	$\emptyset$	$\emptyset$	a,b,n
2	s_2	s_2,s_8	a,b,n	a,b,n,s_2
3	$\emptyset$	$\emptyset$	a,b,n,s_2	a,b,n,s_2
4	$\emptyset$	$\emptyset$	a,b,n,s_2	a,b,n,s_2
5	$\emptyset$	$\emptyset$	a,b,n,s_2	a,b,n,s_2
6	$\emptyset$	$\emptyset$	a,b,n,s_2,s_8	a,b,n,s_2,s_8
7	$\emptyset$	$\emptyset$	a,b,n,s_2,s_8	a,b,n,s_2,s_8
8	s_8	s_2,s_8	a,b,n,s_2,s_8	a,b,n,s_2,s_8
9	$\emptyset$	$\emptyset$	a,b,n,s_2,s_8	a,b,n,s_2,s_8
10	$\emptyset$	$\emptyset$	a,b,n,s_2,s_8	a,b,n,s_2,s_8
11	$\emptyset$	$\emptyset$	a,b,n,s_2,s_8	a,b,n,s_2,s_8
12	$\emptyset$	$\emptyset$	a,b,n,s_2,s_8	a,b,n,s_2,s_8
13	$\emptyset$	$\emptyset$	a,b,n,s_2,s_8	a,b,n,s_2,s_8

図 3 到達定義の計算

データフロー方程式を解けば良い。

$$In(n) = \cup_{p \in pred(n)} Out(n)$$

$Out(n) = f_n(In(n)) = Gen(n) \cup (In(n) - Kill(n))$

ただし、各ノード n の $In(n)$ と $Out(n)$ は最大要素(空集合)で初期化する。このデータフロー方程式は逐次プログラムの解析と同じであるため、並列性がどこに反映されているかはわかりにくい。しかし、$In(n)$ の計算で同期エッジを通して他のスレッドから情報が伝搬されることと、$Gen(n)$ と $Kill(n)$ を求める際に指示文・指示節の意味を考慮する事で、並列性が反映されている。また、このように逐次プログラムと同様の反復法で解が求められる事から、並列性を考慮しても効率良く計算できる事がわかる。

表 3 は、図 1 のプログラムの到達定義の計算結果を示す。表の各欄は左からノードの番号、各ノード n に対する $Gen(n)$ および $Kill(n)$ の内容、および反復計算終了時の $In(n)$ と $Out(n)$ の内容を示す。各変数の定義点は、変数 a,b,n について変数名で、変数 s は変数名と定義点のあるノード番号で識別する。

プログラムの入口ノード s に対する $Gen(s)$ は、入口ですでに有効な値の設定されている変数 a,b,n の仮想定義点からなる。変数 s の定義点のあるノード 2 と 8 では、$Gen(n)$ にそれぞれの定義点が入り、$Kill(n)$ には s の全ての定義点が含まれる。

到達定義の集合 $In(n)$ と $Out(n)$ を見ると、ノード 6 からノード 13 までの各ノードに二つの s の定義が到達する事がわかる。しかし、ノード 4 と 5 には定義点 s_8 が到達していない。同じスレッドでこの関数中の並列ループの複数の反復を実行すれば、前の反復で書き換えた s の値が次の反復の入口に到達することがあるはずである。しかし、このことは保証されない。これは、OpenMP の `for` 構文を用いた並列ループが DOALL 型のループである事に起因する。並列ループの各反復を全て異なるスレッドで実行したとすると、ループ本体で最初のメモリ同期であるノード 6 に達するまでは他のスレッドの実行の影響は受けず、s_8 の定義がノード 4,5 に到達する事はない。特定のループスケジューリングに実行結果が依存してはならないため、ノード 4,5 に s_8 が到達する事を仮定してはならない。ただし、到達しない事を保証するのでもないことには注意する必要がある。そのため、ノード 4,5 で変数 s を参照する時には、適切な同期が必要である。

一方、変数 a,b,n については、全てのノードに関数入口の値が到達することがわかる。このことから、並列領域内のこれらの変数の使用点に対するメモリ同期は不要であり、プライベートデータと同様にメモリ同期点を越えて値を再利用できることがわかる。

6. ま と め

本論文では、OpenMP プログラムに対する並列性を考慮したデータフロー解析の枠組と具体的な問題のモデル化の方法を示した。OpenMP は仕様自体がまだ発展途上であることもあり、コンパイラによる最適化に関する研究は他には殆んど見られない。しかし、共有メモリ並列プログラミング (マルチスレッドプログラミング) は今後ますます普及していくと思われ、汎用的なプログラミングモデルを様々なプラットホーム上で効率良く実装するためには、コンパイラによる最適化が重要な役割を果たすものと考える。そのためには、スレッド間の相互作用を考慮し、同期を越えたり複数のスレッドにまたがったりするような最適化を行なうことが望ましい。そのためには、本論文で述べたようなデータフロー解析手法が必要になってくる。

本方式は現在実装中であり、今後定量的な評価も行なっていく予定である。

謝辞 本研究について有益な御討論・アドバイスをして頂いている、Omni コンパイラグループの皆様、筑波大・電総研・東工大などの研究者からなる TEA グループの皆様、東大の大山恵弘氏に感謝いたします。

参 考 文 献

1) J.M.Bull and M.E.Kambites, JOMP – an OpenMP-like Interface for Java, *Java Grande 2000*, June 2000.
2) OpenMP: Simple, Portable, Scalable SMP Programming (http://www.openmp.org/).
3) Bill Pugh, The Java Memory Model (http://www.cs.umd.edu/~pugh/java/memoryModel/).
4) Erik Ruf, Effective Synchronization Removal for Java, *PLDI '00*, June 2000.
5) 佐藤茂久他, OpenMP コンパイラにおけるメモリ一貫性制御の最適化, *JSPP 2000*, June 2000.
6) Shigehisa Satoh, et al. Compiler Optimization Techniques for OpenMP Programs, *EWOMP 2000*, Sep. 2000(to appear).
7) 佐藤三久他, 分散共有メモリシステム SCASH 上の OpenMP コンパイラ, *SWoPP 2000*, Aug. 2000.