計算機アーキテクチャ 137－28
ハイパフォーマンス 80－28
コンピューティング
（2000．3．3）

粒子輸送モンテカルロシミュレーションコードの自動並列化

川端英之† 上甲聖† 津田孝夫†

大規模数値シミュレーションの一つである粒子輸送コードに代表される、データパラレル指向の並列化に馴染まないアプリケーションに対し、リダクション演算の検出および並列化変換を中心とした自動並列化を施す処理系を開発した。本処理系の特徴は、その強力なリダクション演算検出機能である。従来のリダクション認識手法では解析が困難であった、多重ループ中に複数存在するリダクション演算の認識、及び、配列リダクション変数の検出機能を持つ。間接参照される配列も解析対象である。リダクション認識の過程で、並列化可能性の検査も同時に行なえる。本処理系のリダクション検出アルゴリズムはSSA形式による表現に基づくもので、直接的で簡明であるため、実装も容易である。粒子輸送コードなどに適用し、本手法の有効性を確認した。

Automatic Parallelization of Monte Carlo Particle Transport Code

HIDEYUKI KAWABATA,† KIYOSHI JOKO† and TAKAO TSUDA†

Particle Transport Code is one of huge-scale Monte Carlo simulation codes. From the nature of the physical model it reflects, there exist parallelism in Particle Transport Code. However, complicated control structures and numbers of reduction operations contained in multiply nested loops in such a code prevent it from being parallelized easily. In this paper, we present algorithms to recognize reduction operations in multiply nested loops. The algorithms can also detect arrays used as reduction variables which are referenced by subscripted subscripts. Experimental results show the technique is effective.

1. はじめに

大規模モンテカルロシミュレーションの一つである粒子輸送コードは、有意な数値計算結果を得るために多大な計算時間を費やす必要があることで知られ、高速化の要求は高く、専用計算機や並列計算機を用いた研究が盛んに行なわれている[5]。粒子輸送コードは、対象とする物理モデルの段階で並列性が内在しているため、並列計算機、とりわけ、疎結合大規模並列計算機に向いた応用の一つである。しかしながら、逐次的な記述がなされた粒子輸送コードの多くは、複雑な条件分岐によって記述された長大な試行を、乱数を適宜発生させながら数多く繰り返して、その結果をリダクションによって集計するかたちで記述されている。すなわち、タイトなDOループはほとんど現れず、しかも、リダクションによる大量のループ運搬依存により、モデルの持つ並列性が隠蔽されている。このため、従来の並列化コンパイラでは、自動的な並列性の検出および並列化変換を行なうことができなかった。

これに対し我々は、リダクションによる並列化を積極的に適用することにより、多数の分岐命令を含む規模の大きい多重ループに対して自動並列化を行なう処理系を開発した。本処理系の特徴は、その強力なリダクション演算検出機能である。従来のリダクション認識手法では解析が困難であった、多重ループ中に複数存在するリダクション演算の認識、及び、配列リダクション変数の検出機能を持つ。間接参照される配列も解析対象である。リダクション認識の過程で、並列化可能性の検査も同時に行なえる。本処理系のリダクション検出アルゴリズムはSSA形式による表現に基づくもので、直接的で簡明であるため、実装も容易である。本研究で開発した処理系は、例えば、NAS Parallel Benchmarks[7]の逐次版EPに対して自動並列化を行ない、並列版と同様のコードを出力することができる。

以下、自動並列化の対象とするアプリケーションについて第2章で述べ、第3章ではSSA形式に基づいたリダクション認識及び並列性検出の基本手順を示し、第4章で、多重ループ中のリダクション演算や配列参照を含むリダクション演算の検出方法、並列化可能性判定方法について述べる。第5章では我々の開発した処理系の概要に触れ、第6章で、実規模のコードに適用した結果について述べる。

† 広島市立大学 情報科学部
Faculty of Information Sciences, Hiroshima City University

2. 粒子輸送コード[4)5)]

大規模数値計算の一つである粒子輸送コードは、中性子、電子、光子などの輸送現象のシミュレーションコードで、核分裂－臨界、原子炉内部のシミュレーションなどに多用されている。粒子輸送現象は一般に、多数の粒子の飛翔におけるその一つ一つの挙動が多数回繰り返されるループの一つ一つのイタレーションに対応させられ、所謂「粒子並列」によってモデル化／シミュレートされる。それぞれの粒子の挙動をシミュレートするループのイタレーションは、乱数によって制御されるため各々処理内容が異なる。ここで特徴的なのは、粒子の（原子核との比較における）微小性が故に、各粒子の挙動は独立事象とみなすことができる点である。すなわち、多数回繰り返されるループのイタレーションそれぞれにおいては、本質的には、ループ運搬依存は生じない。

しかしながら、ループのイタレーションそれぞれの実行結果を比較的小数のタリー (tally) に集約する処理（統計処理）は、典型的には総和などのリダクション演算である。タリー（リダクション変数に相当）は、配列を用いてまとめて扱われることが多い。すなわち、粒子輸送コードの自動並列化のためには、多重ループの中でのリダクション検出及び並列性検出能力が必要である。

3. リダクション演算の認識と並列化

3.1 リダクション演算とループ運搬フロー依存

多数個のデータから小数個のデータを生成する処理は一般にリダクション演算と呼ばれる。その典型例としては配列要素の総和や最大値の計算が挙げられる。リダクション演算は通常、ループ中での回帰演算によって記述されるため、依存グラフにおいてループ運搬依存を含む有向閉路の存在に着目することで検出できる[8)11)]。リダクション演算を構成するループとその依存グラフの例を図1に示す。図1右の依存グラフではループ運搬依存を太線で示している。

```
 1:  A = 0.0D0
 2:  DO I = S, E
 3:    B = A + X(I)
 4:    C = B + X(I)
 5:    IF (COND(I)) THEN
 6:      A = C + X(I)
 7:    ELSE
 8:      A = C + Y(I)
 9:    ENDIF
10:  ENDDO
```

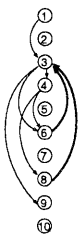

図1 リダクション演算と依存グラフ

3.2 SSA 形式とリダクション演算

プログラムを SSA 形式に変換することにより、データ依存関係はプログラムの字面上に明示される。図2に、図1のプログラムを SSA 形式に変換したプログラムを示す。図中、μ 関数や γ 関数は擬関数と呼ばれる[12)]。なおここでは一部 GSA 形式[3)] による表現を用いている。図2の 2 行目の μ 関数による代入文は、ループ（2 ～ 10 行目）のヘッダ中に含まれ、ヘッダの実行前に実行される、とみなす。

```
 1:  A_0 = 0.0D0
 2:  DO I = S, E    [A_2 = μ (A_0, A_1)]
 3:    B_1 = A_2 + X(I)
 4:    C_1 = B_1 + X(I)
 5:    IF (COND(I)) THEN
 6:      A_3 = C_1 + X(I)
 7:    ELSE
 8:      A_4 = C_1 + Y(I)
 9:    ENDIF        [A_1 = γ (COND(I),A_3,A_4)]
10:  ENDDO
```

図2 図1を SSA 形式に変換したプログラム

SSA 形式においては、ループ運搬依存はループヘッダに付加される代入文の右辺の μ 関数となって顕在化する。この、ループヘッダに配置される代入文に対して、後退代入[3)] を適用すると、以下の式が得られる。

$$A_2 = \mu(A_0, \gamma(\mathrm{COND}(I), \\ A_2 + X(I) + X(I) + X(I), \\ A_2 + X(I) + X(I) + Y(I))) \qquad (1)$$

この式の構造を図3に示す。

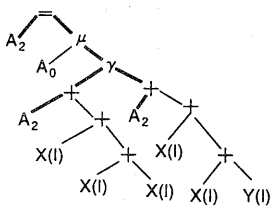

図3 リダクション判定式の構造

後退代入によって式 (1) を得る過程は、図1右の依存グラフにおける有向閉路の探索に相当する。この、ループヘッダに付加された μ 関数を含む式の右辺に後退代入を適用した結果得られる等式を検査することにより、リダクション演算及びループの並列性検出が行なえる。本稿ではこのループヘッダに配置される（μ 関数を右辺に持つ）代入文に後退代入を適用した等式を、**リダクション判定式**と呼ぶ。リダクション判定式は回帰演算の詳細を表す漸化式に他ならない。

3.3 リダクション判定式と回帰経路

図3（すなわち式 (1)）で太線で示されている経路は、左辺の変数の右辺における出現と左辺とを結んで得られる経路である。本稿ではこれを**回帰経路**と呼ぶ。図3における二股の回帰経路は、図1右の依存グラフにおける二つの有向閉路に対応する。すなわち、回帰経路上の演算を調査することで、この回帰演算がリダクション演算か否かを判断することができる。

ここで、リダクション判定式を“評価”することによりリダクション演算であることを判定する手順を、以下のように示す。なおここでは、多重にネストされたループは考えないものとする。また、リダクション判定式は、左辺の変数について「整理された」多項式になっているものとする。

1. **回帰経路検出**　リダクション判定式中の回帰経路を検出する。これは、リダクション判定式の左辺に現れる変数を右辺式中に探し、“木の根”に向かってパスを張るだけで行なえる。なお、リダクション判定式中に回帰経路が検出出来なければ、ループ運搬フロー依存は存在せず、すなわち回帰演算も存在しない。リダクション判定式の存在にもかかわらず回帰演算が存在しないのは、ループ運搬出力依存のみが存在する場合である。
2. **経路の切れ目確認**　回帰経路上に“γ”あるいは単なる“ϕ”があって、その右下あるいは左下の一方にしか回帰経路が伸びていない場合は、リダクション演算とはみなされない回帰演算が存在する。この状況は、あるループのインスタンスにおいて、リダクション判定式の左辺の変数に代入される値が、その直前のインスタンスで代入された値と全く無関係な値である場合があることを意味する。
3. **多重回帰の除去**　回帰経路上のいずれかの演算子（ここでは擬関数記号も演算子とみなす）が“μ”である場合、リダクション演算とはみなされない回帰演算が存在する。これは、同一ループ中の回帰演算の中間結果を参照する回帰演算が存在することをしめしている。
4. **演算の適合性**　回帰経路上の演算子が、全て、交換法則及び結合法則を満たす“同一の”演算であれば、この回帰経路はリダクション演算を意味する。

3.4　ループの並列性判定

前節では、リダクション判定式を用いてループ中のリダクション演算の検出を行なう手法について述べた。ここでは、リダクション演算を含むループの並列化可能性の判定手法を述べる。

一般にループ中で回帰演算が行なわれる場合は、ループ運搬依存の存在のため、効率良い並列化は容易ではない。一方、リダクション演算はループ運搬依存を生じる回帰演算であるが、計算のアルゴリズムの変更による効率良い並列化、すなわち計算の局所化と少数回の通信への演算内容の分解が容易であるため、一般に並列実行を妨げない演算であるとみなすことができる。しかしながら、この並列性は、局所的な“積算”の途中経過の値を引用されることがない（途中経過の値が回帰演算で引用されない）場合に限り、保たれる。ここでは、単純なリダクション演算以外の回帰演算が存在する場合は、並列化できない、とみなすことにする。ループの並列化可能性判定は、リダクション演算の検出後に、以下の手順で判定を行なう。

(1)　リダクション判定式が唯一つしか存在せず、それがリダクション演算を表すものであるなら、並列化可能である。

(2)　リダクション判定式が複数存在する場合、いずれかの判定式の右辺に、複数の“μ”が存在するならば、並列化できない。

4. 多重ループ中のリダクション演算の検出と並列化判定

多重ループに対して並列化を適用する際には、どのネストレベルで並列化が可能かを把握し、どのネストレベルで並列化を行なうかを決定する必要がある。ここでは、多重ループ中における、リダクション演算の検出と並列化可能性判定を各ネストレベル毎に行なう方法を述べる。

4.1　SSA 形式による多重ループの扱い

多重ループにおいて、どのループネストレベルでリダクション演算が行なわれているかを検出するには、ループ運搬フロー依存がどのバックエッジ[1]に起因するかを把握しておかなければならない。図1右のような依存グラフでは、有向辺にラベルを付加して区別することとなる。これに対応させ、SSA 形式における μ 関数には、これ以降、ループのネストレベルを付加するものとする。ここで、ループ中で定義される変数にも、どのループネストレベルで値が代入されるのかを明示することにする。図4に多重ループの SSA 形式表現の例を示す。

```
      A_0 = 3
      B_0 = 2

1     DO I = ...      [A_3 = μ1(A_0, A_2)]
                      [B_2 = μ1(B_0, B_1)]

2       DO J = ...    [A_2 = μ2(A_3, A_1)]

          A_1 = A_2 + C(J)
        ENDDO
        B_1 = B_2 + A_2
      ENDDO
```

図4　多重ループ内のリダクション(1)

図4の外側ループのヘッダに付加されている式から、以下の二つのリダクション判定式が得られる。これらの構造を図5に示す。

$$A_3^{(\mu_1)} = \mu_1(A_0, \mu_2(A_3^{(\mu_1)}, A_2^{(\mu_2)} + C(J))) \quad (2)$$

$$B_2^{(\mu_1)} = \mu_1(B_0, B_2^{(\mu_1)} + \mu_2(A_3^{(\mu_1)}, A_2^{(\mu_2)} + C(J))) \quad (3)$$

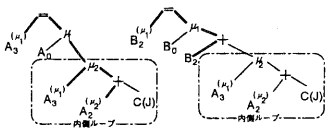

図5　リダクション判定式(式(2)および(3))の構造

4.2 SSA 形式による多重ループ中のリダクション演算の検出と並列化判定

4.2.1 検出例 1

前節の式 (2) において、μ_2 は内側のループに関する回帰演算を表している。その二つの引数は順に、内側ループに至る変数の定義、その変数に関するネストレベル 2 のループ（内側ループ）での演算、をそれぞれ表している。また、式 (2) 中の変数 $A_2^{(\mu_2)}$ は内側ループのヘッダで μ_2 の返り値が代入される変数、すなわち、内側ループでの回帰演算の結果を引用する演算であることが分かる。また、左辺と同じ変数が、内側ループの回帰演算の初期値を意味する μ_2 の第一引数に現れることから、変数 “A” は内側ループでも値が変化していることが分かる。この回帰経路上の演算子は、“μ” 及び “+” だけなので、変数 “A” は外側ループでリダクション変数であるとみなされる。

一方、式 (3) では、回帰経路が μ_2 とは関わらないため、変数 “B” に関する回帰演算には内側ループの内部で行なわれる演算とは関係がない（リダクションの判定に、内側ループの演算子を考慮する必要はない）。しかし、式 (3) の μ_2 の引数に、左辺の変数 $B_2^{(\mu_1)}$ と同じ外側ループで値が代入される変数がある。すなわち、変数 “B” の値は変数 “A” の外側ループでのインスタンスの値に依存するため、図 4は外側のループに関しては並列化できない。

4.2.2 検出例 2

図 7の外側ループに着目すると、以下のリダクション判定式を得る。

$$B_2^{(\mu_1)} = \mu_1(B_0, B_2^{(\mu_1)} + \mu_2(3, A_2^{(\mu_2)} + C(J))) \qquad (4)$$

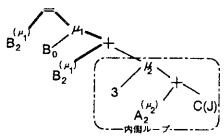

図 6 リダクション判定式 (式 (4)) の構造

式 (4) の構造を図 6に示す。ここでは、回帰経路は μ_2 の引数までは伸びない。さらに、外側ループに関するループ運搬依存は、$B_2^{(\mu_1)}$ のみである。すなわち、変数 “B” は外側ループに関してリダクション変数で、図 7のプログラムは、外側ループ、内側ループのいずれにおいても、並列化可能であると判断できる。

5. 配列参照を含むリダクション演算の検出と並列化判定

配列変数がループ中で参照される場合、とくに、添字式に配列参照が現れる場合には、厳密な依存解析は困難である。しかしながら、粒子輸送コードの例のように、いくつもの統計情報をリダクション演算で得るプログラムでは、多数のリダクション変数をまとめて少数個の配列として扱うことは多い。

```
      B_0 = 2
1     DO I = ...     [B_2 = μ1(B_0, B_1)]

        A_0 = 3
2       DO J = ...   [A_2 = μ2(A_0, A_1)]

          A_1 = A_2 + C(J)
        ENDDO
        B_1 = B_2 + A_2
      ENDDO
```

図 7 多重ループ内のリダクション (2)

配列変数がリダクション変数として用いられている例を図 8に示す。図 8は、光子輸送問題のシミュレーションコード scalgam からの抜粋である。tally 変数に対するリダクション演算が配列の間接参照により行なわれている。

```
      DO 40 ...

        …  (複雑な条件分岐を含む制御フロー)

      DO 230 KK=1,ISTACK
      TALE(INDEX(KK)+0)=
     +        TALE(INDEX(KK)+0)+WINDEX(KK)
      QALE(INDEX(KK)+0)=
     +        QALE(INDEX(KK)+0)+WINDEX(KK)**2
230   CONTINUE
 40   CONTINUE
```

図 8 リダクションによる並列化が有効な例 (2)

5.1 配列リダクション変数の検出

ループの各々のイタレーションにおける依存情報を得ることはできなくとも、配列全体がリダクション変数として扱われているかどうかを検出するためには、厳密な依存解析情報は必ずしも必要ない。ここで述べる手法では、リダクション変数として用いられる配列が同一ループ中で別の添字式により参照されることがない場合に限定することで、複雑な依存解析を行なわずに済ませている。

ここでは、配列リダクション変数検出の手順を示す。本手法は、ループのあるイタレーションにおいて、同一配列変数に対して添字式の値の異なる参照が生じる可能性がなければ、配列参照であっても、スカラと同様にみなせる、という考えに基づいている。

配列リダクション変数は、以下の手順を反復することによって行なう。

1. **スカラ変数の SSA 化** スカラ変数に対してのみ、SSA 形式に変換する。必要に応じて、ループヘッダの μ 関数も構成する。単純なスカラ変数の定義や引用だけでなく、配列の添字式中で参照されているスカラ変数も、変数名の付け換えを適用する。この時点で、ループボディ中で定義されている変数が全てスカラであったなら、前述のスカラ変数に対する検出手法を適用する。DO ループの制御変数は考

慮しなくてよい。

2. **後退代入**　（可能なら）後退代入によりスカラ変数参照を展開する。これにより、式表現が異なる複数の添字式の同値性を判断する。また、前述のスカラリダクション変数検出手法によるリダクション判定、並列化可否判定も、可能なら行なう。並列化不可能と判断されれば、終了。
3. **配列のスカラ化**　添字式がスカラ変数のみから構成される配列参照（添字式中の配列参照も含む）について、スカラ変数とみなすための別名で置き換える。置き換える際、配列名と添字式との両方で区別して、同一の配列参照を同一の名前にする。ここで、置き換え候補の配列変数参照で、少なくとも一つが定義（左辺の変数）であるときに、配列名が同じで添字式が異なる配列参照が存在すれば、そのループは並列化できないと判断する（この手続きは停止する）。添字式がスカラ変数のみから構成される配列参照について、全て名前の置き換えが出来れば、置き換えた変数をスカラであるとみなして、1. から繰り返す。

ここで、図10を用いて、具体的に説明する。まず、

```
DO I = ...
   A(IX(I)) = A(IX(I)) + W * Q(I)
ENDDO
```

図9　配列変数によるリダクションの例

スカラ変数に別名を付け（手順1）、SSA 化する（図10(a)）。次に、後退代入（手順2）を行なう（この例では変化しない）。そして、別名付け（手順3）により配列をスカラ化する（図10(b)）。その後、手順1から繰り返していき、検出対象ループ中の変数が全てがスカラ変数となるか、（手順3において）並列化不可能と判定されるまで、繰り返す。図10の例では、最終的に $A_2^{(1)}$、すなわち A(IX(I)) がリダクション変数であることが検出される。

図11にもう一例示す。これは図11(c)において配列参照が同一ループでのリダクション変数を参照していることが判明するので、並列化不可能と判断されて停止する

DO I = ...
$A(IX(I)) = A(IX(I)) + W_1 * Q(I)$
ENDDO
(a) スカラの SSA 化

DO I = ...
$A(IX^{(1)}) = A(IX^{(1)}) + W_1 * Q^{(1)}$
ENDDO
$[IX^{(1)} : \text{“}IX(I)\text{”}, Q^{(1)} : \text{“}Q(I)\text{”}]$
(b) 配列のスカラ化

DO I = ...
$A(IX_1^{(1)}) = A(IX_1^{(1)}) + W_1 * Q_1^{(1)}$
ENDDO
$[IX^{(1)} : \text{“}IX(I)\text{”}, Q^{(1)} : \text{“}Q(I)\text{”}]$
(c) スカラの SSA 化

DO I = ...
$A^{(1)} = A^{(1)} + W_1 * Q_1^{(1)}$
ENDDO
$[IX^{(1)} : \text{“}IX(I)\text{”}, Q^{(1)} : \text{“}Q(I)\text{”} A^{(1)} : \text{“}A(IX_1^{(1)})\text{”}]$
(d) 配列のスカラ化

DO I = ...
$[A_2^{(1)} = \mu(..., A_1^{(1)})]$
$A_1^{(1)} = A_2^{(1)} + W_1 * Q_1^{(1)}$
ENDDO
$[IX^{(1)} : \text{“}IX(I)\text{”}, Q^{(1)} : \text{“}Q(I)\text{”} A^{(1)} : \text{“}A(IX_1^{(1)})\text{”}]$
(e) スカラの SSA 化　（全てスカラに帰着）

図10　配列変数によるリダクションの検出

```
J = 0
DO I = ...
  B(I) = A(J) + X
  A(J) = B(I) + Y
  J = J + Z
ENDDO
```
(a) オリジナルループ

```
J_0 = 0
DO I = ...
        [ J_2 = μ (J_0, J_1) ]
  B(I) = A(J_2) + X_0
  A(J_2) = B(I) + Y_0
  J_1 = J_2 + Z_0
ENDDO
```
(b) スカラの SSA 化

```
J_0 = 0
DO I = ...     [ J_2 = μ (J_0, J_2+Z_0) ]
  B(I) = A(J_2) + X_0
  A(J_2) = B(I) + Y_0
  J_1 = J_2 + Z_0
ENDDO
```
(c) 後退代入

図11　配列変数によるリダクションの検出(2)

6. 自動並列化コンパイラへの実装

本研究で開発した処理系は、FORTRAN77 のソースレベルでの自動並列化コンパイラで、入力された逐次プログラムに対し、リダクションによる並列化を適用し、MPI[6] を用いて並列化したコードを出力する。

現状では、リダクション検出に基づくループ並列化機能のみを実現している。データ分割や、リダクション以外の同期命令挿入の必要な並列化は行なっていない。また、ループの並列化においては、並列化可能と判断された最外側ループをサイクリック分割により並列化する。通信は、ループのイタレーションを各プロセッサで分担して実行した後に MPI_ALLREDUCE() により行なう。

7. 評　　価

実プログラム中に存在するリダクション演算の検出機能の評価のため、NPB[7] の EP、および、光子輸送問題シミュレーションコード scalgam を、開発した処理系に適用して実測を行なった。実測環境は、Alpha21164(600MHz, 主記憶 4MB) 4 台が Ethernet(100BaseTX) で接続されたワークステーションク

ラスタで、mpich 1.1.2 により並列実行を行なった。

EP については、逐次版の EP に自動並列化を適用し、実測を行なった。出力されたコードと、並列版 EP との違いは、ループの並列化がブロック分割かサイクリックかの違いであった。実行時間はほとんど変わらず、Class W において、4PE 時に 6.3 秒（ただし 1 台の CPU 時間）であった。

scalgam についても、CPU 時間は台数に応じて線形に減少し、1,280,000 粒子の試行で 4PE 時に 13.5 秒（ただし 1 台の CPU 時間）であった。scalgam は、そのリダクション演算部分（図 8）を見ると明らかなように、配列の間接参照がリダクション変数になっている場合を扱えないとリダクション認識による並列化は行なえない。この点からも、本稿で示したリダクション変数検出手法の意義は大きい。

8. 関連研究

依存グラフを用いたリダクション演算の認識手法はいくつかの提案がある[8)11)]が、いずれも多重ループ中でのリダクション演算検出と並列性検出手法について、明確に示されていない。

代表的な粒子輸送コードパッケージ MCNP[4)] には、並列化されたパッケージも含まれている。また、手動での並列化や特殊なハードウェアを用いて高速化する研究例もある[5)]。

アルゴリズムを巧妙に変更することにより並列化・ベクトル化を行なう手法[9)10)]は示されているが、大規模で複雑な制御構造を持つループ内での回帰演算を自動的に検出して効果的に並列化を行なうことについては言及されていない。

リダクションは前述の通りループ中で回帰演算の一種として記述できる。ループ中の回帰演算を一般化インダクション変数 (GIV) と呼んでその closed form を抽出する手法も提案されている[2)13)]。これらの研究はループ中の並列性の抽出機能強化に関するもので、本稿で述べたリダクションによる並列化のために直接利用することができる。

9. おわりに

本稿では、リダクション演算の認識手法と並列性検出手法を、多重ループや配列参照まで含めて包括的に扱えるように整理して示した。配列がリダクション変数である場合のリダクション検出を可能にしたことは、粒子輸送コードをはじめとする大規模シミュレーションの自動並列化の可能性を高めたと言えるであろう。リダクション検出による並列化以外の並列化手段と組み合わせた最適化機能の実現が今後の課題である。

謝辞 本研究を行なうにあたり、いろいろとご意見を下さった広島市立大学情報科学部情報工学科 仲達章雄氏に感謝致します。

参考文献

1) A. V. Aho, R. Sethi, J. D. Ullman: Compilers: Principles, Techniques, and Tools, Addison-Wesley (1986).
2) M. P. Gerlek, E. Stoltz, M. Wolf: Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form, ACM Transactions on Programming Languages and Systems, Vol.17, No.1, pp.85–122 (1995).
3) P. Tu, D. Padua: Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers, Proceedings of ACM International Conference on Supercomputing, pp.414–423 (1995).
4) J. F. Briesmeister (ed.): MCNP – A General Monte Carlo N-Particle Transport Code Version 4B Manual (1997).
5) 樋口健二, 川崎琢治: 粒子輸送モンテカルロ・コード MCNP の並列処理, JAERI-Data/Code 96-019 (1996).
6) MPI フォーラム, MPI 日本語訳プロジェクト: MPI: メッセージ通信インターフェース標準（日本語訳ドラフト） (1996).
7) D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga: The NAS Parallel Benchmarks, RNR Technical Report RNR-94-007, March 1994.
8) 中村 素典, 津田孝夫: “自動ベクトル化コンパイラにおけるイディオム認識法”, 情報処理学会論文誌, 第 32 巻, 第 4 号, pp.491–503 (1991).
9) 津田孝夫: “数値処理プログラミング”, 岩波書店 (1988).
10) 中村 素典, 津田孝夫: “ベクトル計算機のための一次回帰演算の高速アルゴリズム”, 情報処理学会論文誌, 第 36 巻, 第 3 号, pp.669–680 (1995).
11) T. Suganuma, H. Komatsu, T. Nakatani: Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines, In Proceedings of International Conference on Supercomputing, pp.18–25 (1996).
12) M. N. Wegman, F. K. Zadeck: Constant Propagation with Conditional Branches, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 2, pp.181–210 (1991).
13) M. R. Haghighat, C. D. Polychronopoulos: Symbolic analysis for parallelizing compilers, ACM Transactions on Programming Languages and Systems, Vol.18, No.4, pp.477–518 (1996).