平成23年度 学士論文

メソッドの計算順序を
関心事として分離できる言語と
そのHPCへの応用

東京工業大学 理学部 情報科学科

学籍番号 08-1358-8

宗　桜子

指導教員

千葉 滋 教授

平成24年2月6日

概要

本研究では、計算順序を関心事として分離することのできる Java ベースの言語、PersianJ を提案する。天体の運動や気象変動の予測などの大規模なシミュレーションは、実行に膨大な計算を必要とするような処理、high performance computing(HPC) に分類される。HPC プログラミングでは一般に並列計算が用いられるため、プログラム中には同期処理が散在することになる。また、HPC プログラミングではその計算の特性上、おびただしい量のデータを扱わなければならないこともある。このため、プログラマはメソッド間の依存関係や、各メソッドがどのメソッドからデータを受け取らなければならないかに常に注意を払ってコードを書かなければならない。

HPC プログラミングでは多くの場合、パフォーマンスのチューニングやデバッグなどのために、完成したプログラムに対してコード中のメソッド呼び出しの順番を入れ替えたり、新たな処理を追加する必要がある。従って、プログラマはプログラムを書き換えるたびに、同期処理を行うべきタイミングや各メソッド間の依存関係を考慮しなければならない。そのため、この作業はプログラムが複雑になるほど手間のかかるものとなり、プログラマにとって大きな負担となっている。

これらの問題を解決するためには、メソッド呼び出しや変数宣言、同期処理などを行う順番 (計算順序と呼ぶ) をアスペクト指向プログラミングにおける横断的関心事として捉え、これをプログラム中から分離する必要がある。本研究で提案する PersianJ は、この計算順序の一部を担う「同期処理を行うタイミングの最適化」をコンパイラ中に分離する Java ベースの言語である。PersianJ では、メソッド間の依存関係を表す構文 (precedes 文、follows 文) を導入し、プログラムにおいてメインとなる計算を行う箇所でメソッド同士の関係を記述することとした。また、この構文において、メソッド同士の依存関係を記述すると同時に同期処理の必要なメソッドも指定できるようにした。PersianJ は、プログラム中に記述されたメソッド間の依存関係からそれを表す有向グラフを生成し、各メソッドを適切に実行できる順番をトポロジカルソートによって計算する。そしてその上で、得られたメソッド呼び出しの順序を用いて、同期処理を行うタイミングを最適化し、変数宣言や同期処理の挿入を行う。これにより、各メ

ソッド呼び出しの適切な順番での配置、データフローに対応した変数宣言の挿入と共に、最適なタイミングを考慮した同期処理の挿入を自動的に行う言語を実現した。

PersianJ の実装は、JastAdd を用いて実装された Java コンパイラである JastAddJ を拡張することによって行った。抽象構文木において本研究で導入した構文に対応する部分木を探し、その内容を元に計算順序を決定、本来の Java 言語に翻訳して得られた適切な部分木と置き換えることで、目的の動作を得ている。

謝辞

本研究を進めるにあたり、研究方針や論文の構成方法について様々な点でご指導いただいた千葉 滋教授に心から感謝いたします。また、言語仕様や実装方法、発表と論文の構成方法など多岐に渡って指導していただいた武山 文信氏、HPC プログラミングにおける問題点を考察する際に貴重なご意見をいただいた東京工業大学 学術国際情報センター 青木研究室の下川辺 隆史氏に、深く感謝いたします。最後に、共に研究活動を行い、多くの知識や助言をいただいた千葉研究室の皆様に感謝いたします。

目 次

図 目 次

表 目 次

第1章　はじめに

天体間の運動や気象変動の予測といった大規模なシミュレーションで使われるプログラムは、high performance computing(HPC) に相当する。HPC の分野において使用されるプログラムは一般に並列計算を用いて書かれているが、これに加えて HPC プログラミングではプログラム本文を変更する機会が多い。このためにプログラムの本文は、変更を重ねるごとに煩雑になってしまう傾向にある。

HPC プログラミングにおけるこのような問題点の原因には、

- プログラム中で使用するメソッド間の依存関係を踏まえた上で、変数宣言やメソッド呼び出しを行わなければいけない。
- 並列計算に伴って同期処理が必要となる場合、その最も良いタイミングを考慮しなければいけない。

といった点が挙げられる。プログラマは、プログラムに変更を加えるたびにこれらの点に気をつけてコードを書かなければいけないが、これはプログラムが複雑であればあるほど手間のかかる作業となるため、プログラマにとって大きな負担となっている。

これらの問題を受けて、本稿では HPC プログラミングを支援する Java ベースの言語として、PersianJ を提案する。この PersianJ は、separation of concern の観点から、同期を行うタイミングの最適化処理をコンパイラ内に分離する言語である。PersianJ の提供する機能の概要は、次のとおりである。

- メソッド間の依存関係を指定できる構文を新たに導入する。この構文はプログラム中の限定されたメソッド内でのみ使用することができる。PersianJ では、この構文を用いて、あるメソッドに注目したとき、そのメソッドが他のどのメソッドと関係し、どのメソッドから引数としてデータを受け取る必要があるのかを表現できる。PersianJ は、こうして指定された依存関係からメソッド呼び出しを適切な順番で配置し、同時に目的のデータフローに対応した変数宣言の挿入を行う。
- PersianJ では、上で述べた構文を用いることで、メソッド間の依存関係を表現すると同時に、同期処理の必要なメソッドの指定も行う

ことができる。PersianJ はメソッド間の依存関係の情報を元に、同期処理を行う上での最適なタイミングを計算し、コードへの同期処理の挿入も行う。

本稿の残りは、次のように構成されている。第 2 章では本研究の背景となる HPC プログラミングについて述べ、そのサンプルコードを示す。第 3 章ではメソッドの計算順序を関心事として捉え、これをプログラムから分離する必要性を述べる。第 4 章では、本研究で提案する言語 PersianJ での言語仕様とその実装について、第 5 章では、サンプルコードの PersianJ を用いた書き換えの例を、そして第 6 章でまとめを述べる。

第2章　HPCとそのプログラム例

本章では、high performance computing(HPC) という分野において、本稿を読み進める上で必要となる知識を述べる。また後半では HPC プログラムのサンプルコードを示し、これについても解説する。

2.1　HPC

2.1.1　HPC とは

high performance computing(HPC) とは、膨大な計算量を必要とする処理を指す。これには例えば、地球全体の気象や惑星同士の運動シミュレーションが挙げられる。気象の移り変わりや惑星の運動は人間の手で制御できない自然科学現象であるので、具体的な実験は行うことができないためである。また、自動車の衝突シミュレーションも HPC の一例だが、これは実験自体は不可能ではないものの、一度の実験に必要なコストが高いためである。HPC を行うための手段としては、その計算量の多さからスーパーコンピュータを使用する方法が一般的である。

2.1.2　並列計算と同期処理

HPC プログラミングの際には、並列計算が用いられる。すなわち、与えられた問題を小さく分割して複数のプロセッサに分配し、それぞれの小問題を同時に計算するという手法を取ることで、計算効率の向上を図るのである。これは、上の例で述べたように、HPC プログラミングでは気象や惑星運動などの大規模な問題が扱われるためである。

並列計算には以下のように 2 つの種類があり、それぞれをサポートするライブラリや言語拡張が提供されている。

- メモリ分散型
- メモリ共有型

メモリ分散型並列計算では、各プロセッサは個別のデータ領域を持つ。このため、各プロセッサは自身が計算を行っている間、そのままでは他の

プロセッサが持っているデータにアクセスすることができない。そのため、そのような場合には通信によってデータをやり取りする。メモリ分散型並列計算をサポートするライブラリには、MPI [5] が挙げられる。

一方メモリ共有型並列計算では、各プロセッサは全体で一つのデータ領域を持ち、これを共有する。このため、各プロセッサは計算している間、他のプロセッサが担当している問題のデータに対してもアクセスすることができる。メモリ共有型並列計算をサポートする言語拡張としては、OpenMP や CUDA が挙げられる。

並列計算においては、各プロセッサは同期処理によって制御される必要がある。同期処理とは、プロセッサの行っている処理を一時的に停止させたり、再開させたりする処理のことであり、これに対して同期処理を伴わない処理を非同期処理と呼ぶ。この理由は、メモリ分散型、メモリ共有型のどちらの場合でも MIMD(multiple instruction, multiple data) であるため、ある時刻において各プロセッサが同じ処理を行っているとは限らない。従って、全てのプロセッサがそれぞれのタイミングで計算を行えば、想定した順番で計算が行われずに正しい計算結果が得られない可能性があるためである。

今回使用するサンプルプログラムでは、Java 言語用の MPI ライブラリ、MPJ [1, 2] を用いてメモリ分散型の並列計算を行うこととする。これにより、プログラム中では比較的簡単な関数呼び出しを用いることで、message passing 方式でプロセッサ間のデータの送受信と同期処理を行うことができる。また MPI においては、元問題を分割して複数のプロセッサに分配したとき、それぞれのプロセッサをプロセスと呼ぶ。また各プロセスを識別するため、それぞれのプロセスには rank と呼ばれる番号が割り当てられる。rank の値は 0 から (全プロセス数-1) までの整数である。一般に、あるプロセスを指すときには、そのプロセスの rank の値を用いてプロセス 0、プロセス 1…と表現する。以降、本稿でもこれらの用語を用いることにする。各プロセスと元問題の対応を図 2.1 に示す。

2.2　逐次加速緩和法を用いたプログラム例

ここでは、まず Java 言語による HPC のサンプルコードを示し、具体的にどのような計算を行っているかについて述べる。サンプルコードを図 2.2 に示す。本稿では、以降このサンプルコードを用いて話を進めることにする。

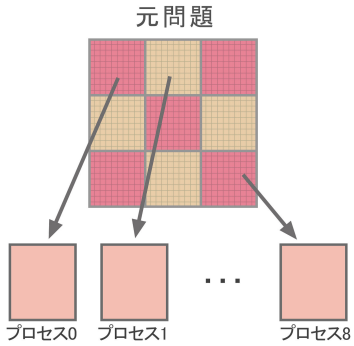

図 2.1: 元問題とプロセス

2.2.1　逐次加速緩和法

サンプルコードでは逐次加速緩和法 (Successive Over-Relaxation) を用いた計算を行っている。

逐次加速緩和法とは、温度分布の様子などをシミュレーションする時に使われる計算法で、n 元連立一次方程式

$$\mathbf{Ax} = \mathbf{b} \tag{2.1}$$

を反復法で解く手法の一つである。反復法の基本的な考え方は、まずある問題に対して適当な初期値 $\mathbf{X_0}$ と漸化式 $\mathbf{X_n} = f_n(\mathbf{X_{n-1}})$ を定めて漸化式から新しい値を $\mathbf{X_1}$、$\mathbf{X_2}$、$\mathbf{X_3}$ と逐次求めていき、新しい値が十分収束するまで計算を続けるというものである。この計算法を以下では SOR 計算と略すことにする。

2.2.2　計算の流れ

今、まず前提として 2 次元ないし 3 次元での格子状の平面、空間を考え、各マス、立方体に値を与える。(以降、このマスや立方体をまとめてセルと呼ぶことにする。) この時与えられる値は、その時関心のある物理量である。つまり、2 次元平面上で温度分布を計算する場合、この値はそのセルでの熱さの度合いを表していることになる。次に、各セルの持つ値に基づいて計算を行い、そのセルでの値を新たなものに更新する。こうして全てのセルの値を新しい値に次々と更新していくことで、時間の経過とともに温度がどのように伝わっていくかを追うことができる。

SOR 計算では、あるセルの近隣のセルが持つ値を用いて、そのセルでの新たな値を計算する。2 次元での格子平面を考えている場合、例えば図

```
1  private void sor(DealDataE dataEast, DealDataW dataWest,
2                   DealDataB dataBottom, DealDataT dataTop,
3                   DealDataN dataNorth, DealDataS dataSouth,
4                   CalcCells calc, double[][] region) {
5
6    /* iteration number */
7    int iter = 0;
8    /* buffer id */
9    int cur = 0;
10   int next = 1;
11
12   while (true) {
13     ArrayList<Request> reqs = new ArrayList<Request>();
14
15     // exchange each direction's surfaces
16     double[] receiveEast = dataEast.exchange(region[cur], reqs);
17     double[] receiveWest = dataWest.exchange(region[cur], reqs);
18     double[] receiveBottom = dataBottom.exchange(region[cur], reqs);
19     double[] receiveTop = dataTop.exchange(region[cur], reqs);
20     double[] receiveNorth = dataNorth.exchange(region[cur], reqs);
21     double[] receiveSouth = dataSouth.exchange(region[cur], reqs);
22
23     // calculate at cells which don't need values from other nodes.
24     double innerMaxResid = calc.calcInner(region, cur);
25
26     // Waitall
27     Request.Waitall(reqs.toArray(new Request[reqs.size()]));
28
29     // Arrange received data into proper place
30     dataEast.arrange(receiveEast, region[cur]);
31     dataWest.arrange(receiveWest, region[cur]);
32     dataBottom.arrange(receiveBottom, region[cur]);
33     dataTop.arrange(receiveTop, region[cur]);
34     dataNorth.arrange(receiveNorth, region[cur]);
35     dataSouth.arrange(receiveSouth, region[cur]);
36
37     // calculate at left over cells
38     double outerMaxResid = calc.calcOuter(region, cur);
39
40     // switch buffer
41     cur = 1-cur;
42     next = 1-next;
43     // increase iteration number
44     iter++;
45
46     // screen max residual
47     double maxResid;
48     maxResid = calc.calcMaxResid(innerMaxResid, outerMaxResid);
49
50     // check residual
51     if (maxResid < 0.000001 || iter >= 10000) {break;}
52   }
53 }
```

図 2.2: SOR 計算のサンプルコード

2.3 のように上下左右のセルの値を用いて計算が行われる。従って、巨大な 2 次元配列をいくつかの領域に分割して複数のプロセスに割り当てた場合、各プロセスが担当する領域の最も外側 (領域の境界部分) に位置するセルでの計算を行うには、データ通信が必要となる。これは、計算を行うために必要なセルの値は他のプロセスが保持しているが、今回はメモリ分散型の並列計算を考えているので、あるプロセスから他のプロセスの持つデータへはアクセスできないためである。ただし、元の 2 次元配列の中で最も外側に位置するセルについては、計算を行わないこととする。

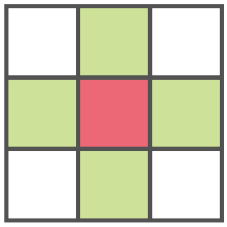

図 2.3: 近隣のセルの値を使った計算

2.2.3　のりしろ領域を使った計算

各プロセスは割り当てられた領域の境界部分 (境界領域) での計算のために、割り当てられた領域と共に図 2.4 に示したような、のりしろ領域と呼ばれる領域を持つ。2 次元の格子平面上において、あるセルでの新しい値を計算する際に必要となるのがそのセルの上下左右の値である場合は、のりしろ領域はプロセスが担当する領域の一周外側にある領域に相当する。それぞれのプロセスは先にデータ通信を行って必要な値をのりしろ領域に確保してから、境界領域のセルで計算を行う。このように通信をまとめて行うと、逐一通信を行ってデータを得るよりも通信にかかるオーバーヘッドが抑えられる。

2.2.4　サンプルコード

ここでは以上の事を踏まえて、図 2.2 に示したサンプルコード (sor メソッド) の解説を行う。

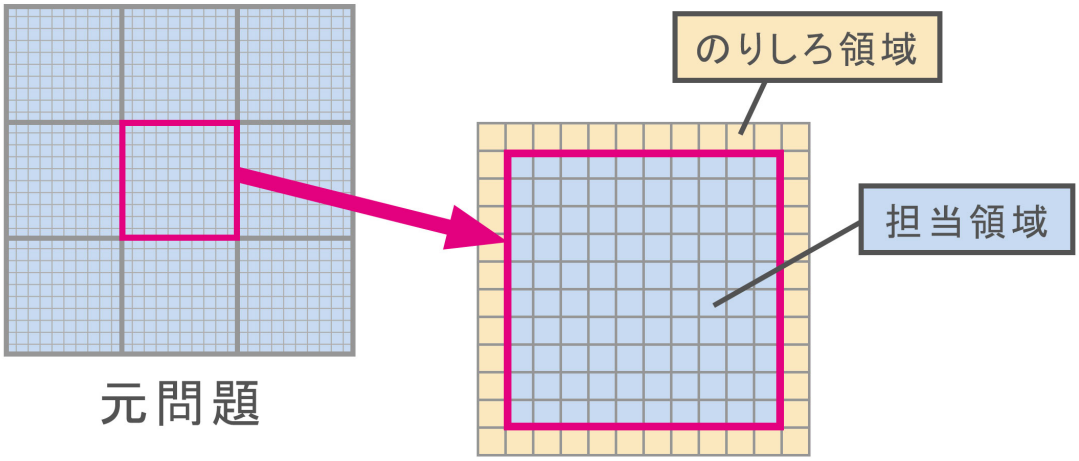

図 2.4: あるプロセスの持つのりしろ領域と担当領域

配列のダブルバッファリング　今回このサンプルコードでは 3 次元の格子空間を考え、この格子空間を double 型の 1 次元配列で表現することとする。その上でこのような配列を二つ (配列 A、配列 B とする。) 用意し、各時刻での状態の計算を、注目する配列を交換しながら次のように行うという方法を取っている。この様子を図 2.5 に示す。ここで配列 A,B はサンプルコードの region[0], region[1] に相当する。

まず、配列 A に現在の値が入っているものとして、その各セルにおいて新たな値を計算し、配列 B に格納していく。この計算が終わった時、配列 B は配列 A から得られた新たな状態を表していることになり、一方で配列 A の持つ状態は以降の計算には不必要なものとなる。そこで今度は配列 B を「現在の状態」と見て、各セルにおいて新たな値を計算する。ここで配列 A の持つ値はもう使われないため、計算された値は改めて配列 A に格納すれば良い。

サンプルコードにおける計算の流れ　sor メソッドの引数として与えられている dataEast、dataWest、...、dataSouth、calc の各オブジェクトは、自プロセスの担当領域の情報を持つ subCube オブジェクトをそれぞれのフィールドに保持し、subCube オブジェクトの持つ情報を元にデータの送受信などの処理を行う。ここで subCube オブジェクトが持つ情報とは、元問題の大きさや担当領域の大きさ、自プロセスの rank の値、各側面に隣接しているセルの有無などである。

以上を踏まえて、sor メソッドで行われる計算は以下のとおりに行われ

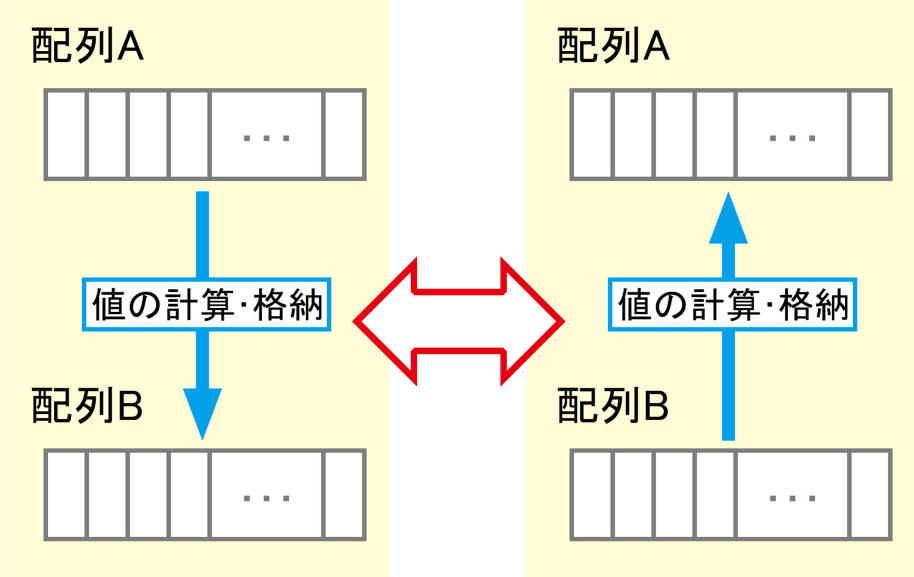

図 2.5: 配列を使ったダブルバッファリング

る。SOR 計算では同じ計算を繰り返し行うため、これを while 文を用いて実装している。

1. のりしろ領域を他のプロセスと交換する。(非同期処理)
2. のりしろ領域を使わずに計算できる領域において新たな値を計算し、この領域について、現在の状態を表す配列の各セルが持つ値と、もう片方の配列の各セルが持つ値との差分の最大値を求める。
3. 1 に対する同期処理を行う。
4. 通信によって得られたのりしろ領域のデータを、自プロセスの持つ配列に格納する。
5. のりしろ領域の必要な領域において新たな値を計算し、この領域について、現在の状態を表す配列の各セルが持つ値と、もう片方の配列の各セルが持つ値との差分の最大値を求める。
6. 配列を交換 (フリップ) する。
7. 全プロセスにおける現在の状態、新しい状態における差分の最大値を求める。

8. 計算を終了するかどうかを判定する。条件を満たしていれば while 文を抜けて計算を終了する。

9. 1 に戻る。

1. のりしろ領域の交換　この処理はサンプルコードの 16 行目から 21 行目に当たり、立方体であるセルの六つの側面のうち、例えば右 (東) 方向の側面でのデータの送受信は次のように行われる。

```
// 東方向でのデータの送受信
double[] receiveEast = dataEast.exchange(region[cur], reqs);
```

dataEast, dataWest, ..., dataSouth は自プロセスの担当する領域の各側面におけるデータを扱うためのオブジェクトであり、他プロセスとのデータの交換はこれらの持つ exchange メソッド内の Isend、Irecv メソッドによって行われる。この Isend、Irecv メソッドは MPJ から提供されているメソッドであり、非同期通信を行ってデータの送信、受信を行う。従って、Isend、Irecv メソッドそれぞれについて一つずつ同期処理が必要となる。また、これらのメソッドを実行すると、Request 型のオブジェクトがその返り値として得られる。このオブジェクトはその時行われた通信の情報を持ち、同期処理を行う際使用される。サンプルコードでは、通信時に発行された Request 型のオブジェクトは可変長配列 reqs に保持される。

2. のりしろ領域を除いた計算　サンプルコードの 24 行目に当たる。現在注目している配列においてのりしろ領域を必要としない領域での計算のみを行い、それと共に各セルにおける新しい値と現在の値との差分を計算し、その中で最も大きいものを得る。本来この計算は 3. 同期処理の後にあっても良いが、この例では、データの送受信を行っている間を利用することで、全体の計算時間を短縮を図っている。

3. 1 に対する同期処理　サンプルコードの 27 行目に当たる。1 で行ったデータの送受信に対して同期処理を行う。同期においても Isend、Irecv と同様に MPJ が提供しているメソッド、Wait メソッドや Waitall メソッドを用いて行う。また、1 で述べたとおり、同期処理の際には通信時に発行された request 型のオブジェクトが必要である。Wait メソッドと Waitall メソッドの違いは一度に同期処理を行う対象の数であり、Wait メソッドは一度に一つの通信に対する同期を取るのみであるが、Waitall メソッドは Request 型オブジェクトの配列を引数とすることで複数の通信に対する同期を取ることができる。

```
// Wait
Request.Wait();
// Waitall
Request.Waitall([request オブジェクトの配列]);
```

サンプルコードにおいては Wait メソッドを用いて Isend、Irecv メソッドそれぞれに対して一つずつ同期処理を行うこともできるが、今回は一度に同期を行うので Waitall メソッドを採用している。

4. のりしろ領域のデータの格納　サンプルコードの 30 行目から 35 行目に当たる。2 で同期処理が行われたことによりデータの送受信が完了しているため、変数 receiveEaset, receiveWest, ..., receiveSouth には確実にのりしろ領域のデータが格納されている。ここではこのデータを、次のようなメソッド呼び出しによって自プロセスの持つ配列の適当な位置に格納している。

```
// 東方向ののりしろ領域を現在の状態に反映
dataEast.arrange(receiveEast, region[cur]);
```

5. のりしろ領域での計算　サンプルコードの 38 行目に当たる。この時点でのりしろ領域のデータが得られているので、のりしろ領域を使った計算が可能となる。ここでは 2. のりしろ領域を除いた計算と同様に、担当領域の最も外側にあるセルにおいて計算を行い、それと共に新しい値と現在の値との差分の最大値を得る。

6. 配列の交換　サンプルコードの 41、42 行目に当たる。このコードでは、まず現在注目している配列の番号、新しい値を入れる配列の番号として int 型の変数 cur、next を用意し、片方に 0、もう片方に 1 を代入する。その後全てのセルでの計算が終わった時点で、41、42 行目で行っているように cur と next 双方の値を交換する。

```
// 配列のフリップ
cur = 1-cur;
next = 1-next;
```

このようにすることで、現在注目している配列は region[cur]、新たな値を格納する配列は region[next] で一律に表すことができる。

7. 全プロセス中で最大であるような差分の計算　サンプルコードの 47、48 行目に当たる。2 と 5 では、各領域での計算を行うと共に、それぞれの領域中の各セルにおける現在の値と新しい値との差分の最大値も導出している。ここでは、2 と 5 で得られた差分のうち大きい方を取り、通信によって他のプロセスと比較して、全プロセス中で最も大きい差分を得ている。実際には 48 行目のメソッド呼び出し

```
// 全プロセス中での最大差分の計算
maxResid = calc.calcMaxResid(innerMaxResid, outerMaxResid);
```

の内部において、MPJ が提供する AllReduce メソッドを用いて行われる。このメソッドは前述の Isend、Irecv メソッドとは違い通信と共に同期処理も行うため、これに対して個別に同期を取る必要はない。

8. 終了判定　サンプルコードの 51 行目に当たる。ここでは、7 で求めた全プロセス中での最大差分 maxResid と、今までに繰り返した計算の回数 iter を用いて、while 文を抜けて計算を終えるかどうかの判定を行う。今回の例では、最大差分が 0.000001 未満となるか、計算を 10000 回繰り返した場合に SOR 計算を終える。

第3章　計算順序の関心事としての分離

本章では、本稿での議論の中心となる計算順序について定義した後、これを関心事として捉え、分離することの必要性を例と共に述べる。

3.1　計算順序

本稿において計算順序とは、メソッド呼び出し等の一定単位の処理を行う順番である。一定単位の処理が表すものとしてメソッド呼び出しは最も簡単な例であるが、これと同様に CPU で計算を行う際に使われる命令 (アセンブリ言語) を実行する順番も計算順序であると考える。

また、for 文を書く場合には、for 文の中に書かれた処理はそのループの回し方によって異なった順番で計算される。例えば以下に挙げた例では、どちらのループの回し方でも同じメソッドが同じ回数呼び出されているが、メソッドが取る引数の値、返り値が異なっているため、それぞれが行う計算は同一ではない。つまり「足す」という操作は同じだが、その上で 1 から足す場合と 10 から足す場合があり、計算順序においてはこの 2 つの例は異なるということである。よって本稿では、このようなループの回し方も計算順序として考えることとする。

```
// 1 から 10 までの整数の和
int sum = 0;
for(int i=1; i<=10; i++){
 sum = add(i, sum);
}

// for 文を展開した場合
sum = add(1, 0); // = 1
sum = add(2, 1); // = 3
sum = add(3, 3); // = 6
sum = add(4, 6); // = 10
sum = add(5, 10); // = 15
sum = add(6, 15); // = 21
 :
```

```
// 10 から 1 までの整数の和
int sum = 0;
for(int i=10; i>=1; i--){
 sum = add(i, sum);
}

// for 文を展開した場合
sum = add(10, 0); // = 10
sum = add(9, 10); // = 19
sum = add(8, 19); // = 27
sum = add(7, 27); // = 34
sum = add(6, 34); // = 40
sum = add(5, 40); // = 45
 :
```

3.2　関心事としての計算順序

3.3　アスペクト指向プログラミング

アスペクト指向プログラミング (Aspect Oriented Programming) とは、プログラム中において、オブジェクト指向プログラミングではモジュール化できないような関心事をアスペクトとして分離し、モジュール化することのできるプログラミング技法である。このため、アスペクト指向はオブジェクト指向の欠点を補う技術であると言える。またこのように、オブジェクト指向プログラミングではモジュール化できないがアスペクト指向プログラミングではモジュール化できる関心事は横断的関心事 (crosscutting-concern) と呼ばれる。 [11] 以下では、横断的関心事を単に関心事と略して呼ぶことにする。

3.3.1　計算順序が変更される例

計算順序は関心事として捉えることができる。これは、一般的に計算順序は一意に定まらないためであるが、これに対して以下で計算順序が変更される例について述べる。

コンパイルにおける最適化　プログラムをコンパイルする際、コンパイラでは最適化が行われる。最適化には様々な方法があるが、前述したループの回し方の反転もその中の一つである。for 文の最適化にはそれ以外にも、入れ子になった for 文中に配列の操作があった場合、メモリアクセスを局所化して高速化を図るために以下のようにループの内側と外側を交換するという技法もある。

```
for(i=0; i<height; i++){
  for(j=0; j<length; j++){
     :
    array[i][j] = ... ;
     :
  }
}
```

⇒

```
for(j=0; j<length; j++){
  for(i=0; i<height; i++){
     :
    array[i][j] = ... ;
     :
  }
}
```

またプログラム中で計算に不要な式があった場合 (例えば、変数が定義だけされていて実際には使われていないなど) にその文を削除する他、if 文の then 節、else 節に同じ式があった場合、その式を以下のように if 文

の直前に移動するという技法も使われるが、これらも計算順序の変更にあたる。

```
if(i<10){
  a = 3;
  i = i+3;
}else{
  a = 3;
  i = i-3;
}
```

⇒

```
a = 3;
if(i<10){
  i = i+3;
}else{
  i = i-3;
}
```

アセンブリ言語のアウト・オブ・オーダー実行　アセンブリ言語において、命令を実行する際にはイン・オーダー実行とアウト・オブ・オーダー実行という、二つの実行方法があるが、このアウト・オブ・オーダー実行では計算順序の変更を行う。イン・オーダー実行では命令を読み込んだ時、その命令のオペランド (命令を実行するために必要な値で、メソッドに対する引数にあたる。) が揃っていれば実行するが、オペランドが揃っていない場合は命令の実行を止め、オペランドが揃うまで待つ。これに対してアウト・オブ・オーダー実行では命令の待ち行列 (またはバッファ) を用意しておき、命令を一つ読んだらまずそこに溜める。この時オペランドが揃っていればイン・オーダー実行同様にその命令は実行されるが、揃っていない場合はオペランドが得られるまで待ち行列に格納されたまま実行されない。(この間、プロセッサは以降の命令に対しても同様の処理を行い続ける。) オペランドが得られたら、その命令は自身より前に待ち行列に加えられた命令が残っていたとしても、行列から取り除かれて実行される。すなわち計算順序の変更が行われ、実行されている。 [6]

3.3.2　計算順序の固定条件

さて、あるプログラムにおいてその中の 2 つの処理に限定して注目すると、これらが互いに関連のある処理であれば、どちらが先に実行されるべきかという順序は固定される。これに対して、逆に 2 つの処理の間に何の関連も無い場合は、順序は固定されない。では、2 つの処理が互いに関連しているかどうかを決めるのは何か。それはデータフローである。アウト・オブ・オーダー実行の例でも見られたが、ある処理を実行する時、必要な値が得られていなければその処理は実行できない。そしてその必要な値は、他の処理を行うことによって得られるものである。つまり、プログラム中から 2 つの処理を選んだ時、片方の処理がもう片方の処理の結果を

必要とするのであればそれらは関連し、そうでなければ関連しないと判断できる。

3.3.3　計算順序の多様性

例で述べたように、同じ計算結果が得られる場合でも複数通りの計算順序が考えられるケースは多く存在する。しかし、複数の計算順序が考えられる場合、それらの計算の結果が同じであっても、それ以外の点においても各計算順序が全く同じ振る舞いを見せるとは限らない。「それ以外の点」とは、例えば次のようなものを指す。

- メモリの使用量の大きさ
- 計算全体にかかる時間

メモリを効率よく使うような順序で計算を行えばメモリの使用量は抑えられ、似たような処理をひとつにまとめて行うなどの処置を取れば、計算にかかる時間は短くなるかもしれない。従ってこのような理由から、プログラムの実装の際には、複数ある計算順序の選択肢から、できるだけ良い計算順序を選ぶべきである。

ただし、全ての条件が最良であるような計算順序を作り出すことは一般的に難しいため、目的に合った計算順序を選ぶ必要がある。例えばあるプログラムでは、メモリを多く使う代わりに計算にかかる時間は抑えられるような順序で計算を行っているとする。このプログラムにおいて、新たにより少ないメモリで計算を行えるような計算順序を考えた場合、その計算順序では計算時間が前のものより長くなってしまうかもしれない。こういった場合にどちらの計算順序を選ぶか、つまり使用メモリの量と計算時間のどちらを優先させるかは、プログラムの目的に合わせてプログラマが判断しなければならない。

3.4　計算順序の分離

本稿では、前述した計算順序を関心事として分離することを考えるため、まず計算順序を分離していない場合の問題点について述べる。

2.2.1 節で紹介したとおり、SOR 計算、ひいては HPC プログラミングでは、同じ計算を何度も繰り返して行うためプログラム中には while 文や for 文といった計算ループが現れる。実際のシミュレーションの間に最も実行されるのはこの計算ループであり、従って計算ループは SOR 計算の要であると言える。

3.4.1　計算ループにおける計算順序

HPC プログラミングでは、計算ループ中の計算順序を変更する機会が他と比べて多く存在する。これには例えば以下のような場合が挙げられる。

- パフォーマンス向上の為のチューニング
- デバッグ
- シミュレーションの振る舞いのカスタマイズ

HPC プログラミングにおいて、プログラムは一度書き上げたらすぐに実行できるわけではない。実用的なシミュレーションには、一般に数日から 1 年といった長い時間を要するため、計算ループ中で計算順序の細かな調整を行い、より実行速度の速い計算順序を採用することで、シミュレーション効率の改善を図るのである。

2 章のサンプルコードにおいて、担当領域の計算を

- のりしろ領域の必要なセルでの計算
- のりしろ領域の不要なセルでの計算

に分けて行ったのは、このチューニングの一例である。Waitall メソッドを実行する時点でデータ通信が終わっていない場合、通信が終わるまで他の処理を行うことはできない。従って無駄な時間を使ってしまうことになり、非効率的である。これに対して、Waitall メソッドを呼び出す直前に、通信に関係ない計算、すなわちのりしろ領域の不要なセルでの計算を行っておくことで、全体の計算の効率を上げることができる。このように、一般に同期処理を持つ計算ループ内では同期処理の直前に、その同期処理に対応する非同期通信とは無関係な処理をできるだけ行うことで計算効率を上げられる。このようなチューニングは、計算のオーバーラップと呼ばれる。

また、プログラムのデバッグを行う際には、ログ出力などのデバッグ専用の処理を追加しなければならなくなる。更に、既に完成しているプログラムを元にして別のシミュレーションを行うプログラムを作成する時には、新たな処理の追加や変更が必要となることもある。

3.4.2　問題点

こういった局面において、計算ループ内に計算順序がハードコーディングされていると、処理の順序そのものの変更のみならず、追加についても多くの労力を割くことになる。これは次のような問題に起因する。

- 計算ループ内の処理の流れを追い難い
- 関連する処理同士の依存関係に注意した上で、同期処理、変数宣言を行わなければならない

計算順序を考えるためには各処理同士の関連の有無を調べなければならないが、一般的にこれらの処理は計算ループ内に散らばっており、計算の流れが一目で判らないことが殆どである。そのような状況でプログラマは各処理の依存関係に注意して、計算順序の並び替えや処理の追加、またこれに伴って適切な同期処理や変数宣言を行わなければいけないのである。

第4章　PersianJの提案

本研究では、HPC プログラミングにおいての要である計算ループ内の計算順序の分離を、Java 言語を拡張して実現することを考える。本章では本研究で提案する言語 PersianJ の仕様とその実装について述べる。

4.1　PersianJ

PersianJ は、計算順序を決定付ける要素の一つである同期のタイミングの最適化処理を、プログラム中に自動で挿入する言語である。最適な同期のタイミングは、計算ループ内で使われている各メソッド同士の依存関係の定義から計算される。本研究においては、最適化をどのように行うかのカスタマイズを可能にすることを最終的な目標としているが、現時点では最適化処理をコンパイラ内へ分離することに留まっており、最適化の方法を変更することはできない。

4.2　PersianJの言語仕様

サンプルコードに示した SOR 計算の計算ループについて考える。PersianJ では、メソッドの修飾子として新たなキーワード dispatch を導入し、この dispatch キーワードを修飾子として持つメソッド内で、SOR 計算に必要なメソッド同士の前後関係を記述できるようにする。このようなメソッドに対して、PersianJ はユーザーが記述した「メソッドの前後関係」に従ってメソッド呼び出しを最適な順序に並び替え、同期処理や計算の終了判定を追加した上で、サンプルコードのような While 文を生成する。

4.2.1　イベントとルール

各メソッド間の依存関係は、有効グラフで表すことができる。すなわちメソッドをノードとして考えると、メソッド a の返り値をメソッド b が必要とする場合、メソッド a のノードからメソッド b のノードへのエッジが存在する、と見なすことができる。ここではメソッド間のこのような対応関係を表すために、新しい用語として「イベント」と「ルール」を導入す

る。まず、Java 言語において、イベントを次のように定義する。

> イベントとは、あるオブジェクトと、
> そのオブジェクトの持つメソッドのペアである。
> A オブジェクトの a メソッドというイベントは、A.a と表す。

イベントはメソッドと同様に引数と返り値を持ち、有効グラフのノードに相当する。また、イベントを用いて、ルールを次のように定義する。ただし、イベント B.b はイベント A.a と異なるものであるとする。

> ルールとは、依存関係のある二つのイベントのペアである。
> イベント A.a の返り値をイベント B.b が必要とする場合、
> イベント A.a、B.b 間のルールを (A.a, B.b) と表す。

ルールは、上で述べた有効グラフのエッジに相当する。ルールの持つイベントは、エッジの始点であるノードか終点であるノードかのどちらかとして捉えることができる。このため、(A.a, B.b) と (B.b, A.a) は違うルールと見なされる。

4.2.2　ルールに従った計算順序の決定

あるイベントに注目したとき、そのイベントと、そのイベントと関連のあるイベントとの依存関係を表すルールの種類には 2 通りある。

ある SOR 計算の計算ループ中に存在する各イベント間の依存関係が、図 4.1 に示したような有効グラフで表されたとする。この有効グラフにおいて一つのノード、例えばイベント B.b に着目すると、イベント B.b と関連するイベントは A.a、C.c、D.d の 3 つである。これらとイベント B.b の依存関係は、有効グラフから次のように表せる。

- A.a → B.b (B.b は A.a の後に実行される)
- B.b → C.c (B.b は C.c の前に実行される)
- B.b → D.d (B.b は D.d の前に実行される)

これら 3 つの依存関係をルールとして表しなおすと、

- (A.a, B.b)
- (B.b, C.c)

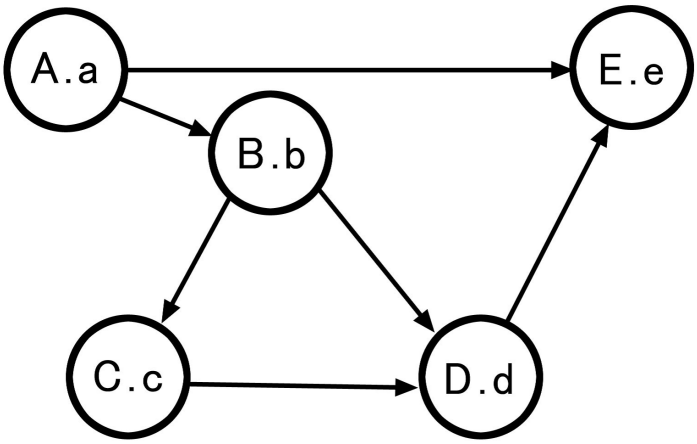

図 4.1: イベント間の依存関係を表した有向グラフ

- (B.b, D.d)

となる。ここで導出されたルールそれぞれについて見たとき、現在注目しているイベント B.b は、各ルールの左の子として表れる場合と右の子として現れる場合がある。すなわち、イベント B.b に対するルールの形には次の 2 種類がある。ただし*は、B.b と関連のあるイベントとする。

- (B.b, *)
- (*, B.b)

このように、あるイベントと他のイベントに対するルールは 2 通りに表現される。

さて、あるイベントに注目したときには 2 種類のルールが生成され得るが、ルールはそれぞれの種類において複数生成される可能性がある。すなわち、前節と同様の例を用いると、イベント B.b に対するルールは

- (B.b, *) の形のもの　…　(B.b, C.c), (B.b, D.d)
- (*, B.b) の形のもの　…　(A.a, B.b)

と分けることができる。(*, B.b) の形であるルールは 1 つであるが、(B.b, *) の形であるルールは 2 つ存在する。そこで、サンプルコード中にある各イベントが、それぞれどちらの形のルールをいくつ持つのかを調べることにする。

まず、この問題に対して、ルールと有向グラフのエッジとの関係から以下のことが言える。
イベント B.b において考えた時、

- (B.b, *) の形を取るようなルールの数は、イベント B.b のノードに入るエッジの数と等しい。
- (*, B.b) の形を取るようなルールの数は、あるイベントのノードから出るエッジの数と等しい。

従って、あるイベントがどのようなルールをいくつ持つのかを調べるためには、以下の値を考えれば良い。

- 自イベントの返り値を必要とするイベント数
- 自イベントが必要とする引数を提供するイベント数

precedes ルール

自イベントの返り値を必要とするイベント数については、次のような結果が得られた。ただしサンプルコードにおいて、あるイベントの返り値が他のイベントの引数以外の場所で使われている場合にも、他のイベントの引数として使われる場合と同様に数えることとする。

- なし　…　dataEast.arrange 等
- 1 つ　…　dataEast.exchange, calc.calcInner 等

この結果によると、サンプルコードにおいてある 1 つのイベントに着目した時、自イベントがその返り値を与えるべきイベントの数は、高々1 つであることがわかる。すなわち、イベント B.b に対して (B.b, *) と表されるようなルールの数は、各イベントにつき 0 または 1 である。
以上の結果から、注目したイベントを左の子として持つようなルールを表す構文を、新たなキーワード precedes、result を用いて次のように定義する。以降、この構文によって書かれた文を precedes 文と呼ぶ。

```
[対象イベント名]() precedes [関連イベント名](result);
```

これは、対象イベント名で指定したイベントが、関連イベント名で指定したイベントより前に実行されるべきであり、対象イベントの返り値は関連イベントに引数として渡されることを表す。例えば、サンプルコード中のイベント dataEast.exchange は、データ通信によって得られたの

りしろ領域のデータの一部を返り値として返す。この返り値はイベント dataEast.arrange が必要とするため、dataEast.exchange と dataEast.arrange の関係を表す precedes 文は次のように記述できる。

```
dataEast.exchange() precedes dataEast.arrange(result);
```

precedes 文において、キーワード result は省略することができる。すなわち、あるイベントは他のイベントに返り値を与えないが、前後関係は指定したいという場合にも、キーワード result を省略することでルールを定義できる。

follows ルール

自イベントが必要とする引数を提供するイベント数については、次のような結果が得られた。ただしサンプルコードにおいて、あるイベントが引数を必要としていても、その引数が他のイベントによって提供されるものでない時には、その引数については無視することとする。

- なし　…　dataEast.exchange、 calc.calcInner 等
- 1 つ　…　dataEast.arrange 等
- 2 つ以上　…　calc.calcMaxResid

この結果によると、ある 1 つのイベントに注目した時、自イベントは他の複数のイベントから値を受け取る可能性があることが分かる。すなわち、イベント B.b を用いて (*, B.b) と表されるようなルールの数は、各イベントにつき複数ある可能性がある。

ここでは以上の結果から、注目したイベントを右の子として持つようなルールを表す構文を、新たなキーワード follows、記号=> を用いて次のように定義する。以降、この構文によって書かれた文を follows 文と呼ぶ。

```
[対象イベント名](arg1, arg2, ..., argN)
            follows [関連イベント名 1]() => arg1,
                    [関連イベント名 2]() => arg2,
                      :
                    [関連イベント名 N]() => argN;
```

これは、対象イベント名で指定したイベントが、関連イベント名 1, 関連イベント名 2, ..., 関連イベント名 N で指定したイベント群より後に実行されるべきであり、関連イベント群の返り値は対象イベントに引数として渡されることを表す。ここで関連イベント群の返り値は、対象イベント

の引数の位置で指定された仮引数名 arg1, arg2, ..., argN に、新たに導入した記号=> によって関連付けられる。これにより、対象イベントは必要な値を、適切な順番で受け取ることができる。

例えば、サンプルコード中のイベント calc.calcMaxResid は、

- のりしろ領域を必要としない領域での、新旧の計算値における差分の最大値
- のりしろ領域を必要とする領域での、新旧の計算値における差分の最大値

の二つの値を必要とする。これらの値は、それぞれイベント calc.calcInner、イベント calc.calcOuter から提供される。従って、calc.calcMaxResid と calc.calcInner、calc.calcOuter の前後関係を表す follows 文は次のように記述される。

```
calc.calcMaxResid(inner, outer) follows calc.calcInner() => inner,
                                        calc.calcOuter() => outer;
```

follows 文においても precedes 文と同様に、記号=> による値と仮引数との関連付けは省略可能である。従って、対象メソッドが引数を必要としない場合にも、follows 文によって前後関係を指定できる。

4.2.3　処理の自動挿入

同期処理や計算の終了判定処理の挿入は、各イベントにおけるルールを用いて行われる。ここでは、ユーザーがこれらの処理挿入するために留意すべき点について述べる。

同期処理

dispatch メソッドの内部においてルールを記述する際、対象イベントが同期処理を必要とする場合はキーワード async を付加することとする。async は、同期処理が必要なイベントとそうでないイベントを区別するために新たに導入したキーワードである。例えば、サンプルコードの dataEast.exchange は内部でデータ通信を行うため、同期処理を必要とする。従って、前述した dataEast.exchange と dataEast.arrange との前後関係を指定したルールは

```
async dataEast.exchange() precedes dataEast.arrange(result);
```

```
1  public interface AsyncResult {
2    public void sync();
3  }
```

図 4.2: AsyncResult インターフェース

```
1  public class MPJ_AsyncResult implements AsyncResult{
2    ArrayList<Request> reqs;
3
4    MPJ_AsyncResult(){
5      reqs = new ArrayList<Request>();
6    }
7
8    public void addRequest(Request r){
9      reqs.add(r);
10   }
11
12   public void sync(){
13     Request.Waitall(reqs.toArray(new Request[reqs.size()]));
14   }
15 }
```

図 4.3: AsyncResult を実装したクラス

と書き直せる。

async キーワードの付けられたイベントの返り値は、図 4.2 に挙げるインターフェース、AsyncResult を実装したオブジェクトである必要がある。インターフェース AsyncResult を実装したクラスは、同期処理を行うメソッドとして sync メソッドを持たなければならない。図 4.3 に示した MPJ_AsyncResult クラスは、このようなクラスの今回のサンプルコードに合わせた実装例である。このクラスは、フィールドに Request 型の可変長配列 reqs を持つ。また addRequest メソッドで Request 型のオブジェクトを reqs に追加することができ、sync メソッドを呼び出すことで、reqs に格納されている各要素に対して同期処理が行われる。こうした方法を取るのは、データの送受信に伴って生成される Request 型のオブジェクトが、同期処理の際に必要となるためである。

同期処理のタイミングの最適化においては、現在は同期処理を必要とするイベントがある場合に複数の同期処理を一箇所にまとめ、それを計算順序中のどのタイミングで実行するか検討する、という方法を取っている。しかし、この方法は全ての場合に同期のタイミングを最適化できるというわけではない。記述されたルールによっては、同期を複数回に分けて行うことが最良である場合があるためである。例えば、イベント間のルールが

- (A.a, B.b)

- (B.b, C.c)
- (D.d, E.e)

と表される、すなわち次のように前後関係が定められているとする。

```
A.a → B.b →C.c
D.d → E.e
```

この時、イベント A.a、B.b、D.d が同期処理を必要とする場合、最も効率を上げるためには、同期を 2 回に分けて、例えば次のような順番で行う必要がある。

```
A.a
D.d
A.a、D.d の同期
B
E
B.b の同期
C
```

本研究ではこのような最適化を行うことが目標である。

計算の終了判定

計算ループの終了判定は、dispatch メソッド内で定義されたルールに従ってイベントを配置したとき、最後に配置されるイベントの値によって判定する。すなわち、PersianJ は dispatch 文の変換を行う際、計算ループの終了判定のための if 文を while 文に挿入するが、この if 文の条件式に、最後に配置されるイベントの返り値を与える。従って、計算ループの最後に配置されるべきイベントの返り値は boolean 型である必要がある。

4.2.4　precedes ルールと follows ルールの非同一性

precedes ルールのみ、または follows ルールのみを用いて全ての計算順序を表現することはできない。サンプルコードにおいて、前述したようにイベント calc.calcMaxResid とイベント calc.calcInner、calc.calcOuter との依存関係は follows ルールによって表せる。これを precedes ルールに書き直すと

```
calc.calcInner() precedes calc.calcMaxResid(result);
calc.calcOuter() precedes calc.calcMaxResid(result);
```

となるが、これでは calc.calcInner と calc.calcOuter それぞれの返り値が、calc.calcMaxResid の 2 つの引数のうちどちらに渡されるべきなのかの表現が不可能である。

また、キーワード async は precedes ルールにのみ付加する意味がある。follows ルールは対象イベントと、それより前に実行されるべき関連イベントとの前後関係を定めるものであるので、対象メソッドが同期処理を必要とする場合でも、関連イベントは同期処理のタイミングを考慮せずに配置することができる。また、async は対象メソッドに対してのみ使用できるキーワードである。従って

```
dataEast.arrange(data) follows async dataEast.exchange() => data;
```

と書くことはできない。一方、

```
async dataEast.arrange(data) follows dataEast.exchange() => data;
```

では同期処理を必要とするのが dataEast.arrange イベントとなってしまい、意味が異なってしまう。

4.3　実装

PersianJ は、JastAdd [3, 4, 10] によって実装された JastAddJ を元に、Java 言語を拡張することで実装した。

JastAddJ は、コンパイラを実装するためのフレームワーク JastAdd によって実装された、Java 言語のコンパイラである。JastAddJ は次のようなファイルを持つ。

- ast ファイル … 抽象構文木における各ノード (AST ノード) を表すクラスの継承関係を定義するためのファイル
- parser ファイル … 構文解析のアルゴリズムを記述するためのファイル
- jrag ファイル … AST ノードを表すクラスに対して、新たなメソッドを追加する、あるノードを他の種類のノードと交換するなどの処理を記述するためのファイル

ユーザーはこれらの中身を書き換えることで、新たな構文を Java 言語に導入したり、その動作を変更することができる。

4.3.1　文法の追加

4.2 節で導入した新たなキーワード、precedes、 follows、 async、 => を使った以下の構文を解析するため、イベントやルール、dispatch メソッドを表現する AST ノードを新たに作り、その上で構文解析器に規則を追加した。

AST ノードの追加　まず、今回新たに作った AST ノードと、そのノードの継承するノードについて、AST ファイルに記述した定義と共に述べる。

```
DispatchStmt: Stmt ::= Rule*;
```

AST ノード DispatchStmt は、Stmt ノードを継承し、後述する Rule ノードのリストを子に持つ。Rule ノードは、dispatch メソッド内のルールを表現するノードである。すなわち DispatchStmt は、dispatch メソッドに記述されたルールをひと括りにしたステートメントを表すノードである。

```
Name: ASTNode ::= <Name:String>;
Event: ASTNode ::= ClassName:Name MethodName:Name varName:Name*;
```

Name ノードは文字列を子に持ち、イベントのオブジェクト名などを表現する。

Event ノードは Name ノードを用いて定義している。ひとつのイベントが持つべき情報は

- オブジェクト名
- メソッド名
- 自イベントの仮引数名

であるので、Event ノードはオブジェクト名を表す Name ノード、メソッド名を表す Name ノード、自イベントの取る仮引数名を表す Name ノードのリストを持つ。Event ノードの持つ子ノードの名前は、それぞれ ClassName、MethodName、varName とした。

また、Name ノード、Event ノードはどちらも似た働きをするノードが存在しないため、ASTNode を継承するのみである。

```
Async: ASTNode;
Rule: ASTNode ::= [Async] TargetEvent:Event RelatedEvent:Event*;
PrecedesRule: Rule;
FollowsRule: Rule;
```

Async ノードは、キーワード async を表現するものである。また Rule ノードは、ルールを dispatch メソッド内のルールを表現する。ひとつのルールが表現しうる情報は

- async キーワードの有無
- 対象イベントの情報

- 対象イベントに関連するイベントの情報

の 3 つである。そのため、Rule ノードは対象イベントを表す Event ノード (名前を TargetEvent とする)、関連イベントを表す Event ノードのリスト (名前を RelatedEvent とする)、更にオプショナルな子として Async ノードを持つ。このように、RelatedEvent を Event ノードのリストとすることで、dispatch メソッド内において、ひとつの対象イベントにつき複数の関連イベントを指定できるようにした。更に、precedes ルールと follows ルールでは、保持する情報はどちらも上に挙げた 3 つであるが、対象イベントとその関連イベントの前後関係は異なる。そこでこの二つのルールを区別するため、それぞれのルールを表現するノード PrecedesRule と followsRule を定義し、Rule ノードを継承させた。従って、ノード PrecedesRule と followsRule の実体は同じ Rule ノードである。

また、Event ノード、Name ノードと同様に、Async ノードと Rule ノードにおいても、単に ASTNode を継承しているのみである。

生成規則の追加　新たに導入したノードを用いて、precedes 文、follows 文、dispatch メソッドの生成規則を parser ファイルに追加した。次に示すのは、追加した生成規則の一部である。

```
Rule follows_rule =
  async.a? target_event.t FOLLOWS follows_related_event_list.l
                        {: return new FollowsRule(a, t, l); :}
 ;

Event target_event =
  class_name.c DOT target_name.t LPAREN target_argument_list.l? RPAREN
                                      {: return new Event(c, t, l); :}
 ;
```

4.3.2　イベント情報の収集

構文解析器によって AST 木を生成したら、dispatch メソッド内に記述されたルール、イベントの情報を収集しなければいけない。これに先立って、まずひとつのイベントを表す EventInfo クラス、ひとつのルールを表す RuleInfo クラスを定義した。

EventInfo クラス　EventInfo クラスは、次のフィールドを持つ。

- String eventName
- ArrayList¡String¿ resultProviderList

eventName は、そのイベントの名前を表す文字列であり、

`[オブジェクト名].[メソッド名]`

の形式を満たしている。

resultProviderList は、自イベントの引数に値を与えるイベントの、名前のリストである。例えば自イベントの引数が

`arg1, arg2, ...`

の順番で必要な場合は、resultProviderList には

`arg1` を提供するイベント名，`arg2` を提供するイベント名，...

というように、各イベント名は自イベントの引数と対応した順番で保持されるようにする。

RuleInfo クラス　RuleInfo クラスは、次のフィールドを持つ。

- int hasAsync
- EventInfo from
- EventInfo to
- ArrayList<String> fromArgs
- ArrayList<String> toArgs

フィールド from、to それぞれには、自オブジェクトが表すルールのエッジの始点、終点のイベントを表す EventInfo オブジェクトが代入される。

フィールド hasAsync は、自ルールの対象イベントが同期処理を必要とするかを表す。ただし、「同期処理が必要である」という場合にも 2 通りある。precedes ルールであれば同期を取る必要があるのは from に保持されているイベントとなり、follows ルールであれば to に保持されているイベントとなるのである。そのため、hasAsync の取り得る値とその意味を次のように定義した。

- 0 … from、to のどちらのイベントも同期処理を必要としない
- 1 … from の保持するイベントが同期処理を必要とする
- 2 … to の保持するイベントが同期処理を必要とする

フィールド fromArgs、toArgs は、自オブジェクトの表現するルールの対象イベント、関連イベントの引数情報を保持する。ここで引数情報とは、precedes 文、follows 文中に実際に現れる、各イベントの引数を示す文字列である。例えば次のルール

```
dataEast.exchange() precedes dataEast.arrange(result)
```

から得られる関連イベントの引数情報は `result` であり、

```
calc.calcMaxResid(inner, outer) follows calc.calcInner() => inner,
                                        calc.calcOuter() => outer;
```

から得られる対象イベントの引数情報は `inner` と `outer` の 2 つであり、その関連イベント calc.calcInner の引数情報は `inner`、calc.calcOuter の引数情報は `outer` である。

この例からわかるように、自オブジェクトが precedes ルールから生成されたか、follows ルールから生成されたかでその内容に違いがある。

自オブジェクトが precedes ルールから生成された場合、対象メソッドには引数情報が記述されていないため、fromArgs のリストは空となる。対して関連メソッドの引数に result が指定されているならば、toArgs のリストには文字列 result が格納される。

一方、自オブジェクトが follows ルールから生成された場合、対象メソッドには複数の仮引数が与えられている場合もあり、そのような場合は toArgs のリストにはそれら仮引数の文字列が保持される。対して関連メソッドにおいては、返り値が記号=> で仮引数に代入されている場合がある。このようなときに、fromArgs にはその仮引数の文字列が保持される。これにより、対象メソッドの EventInfo オブジェクトを生成する際に、各仮引数を与えるイベントの名前を得ることができる。

ここで、改めてルールやイベントの情報を収集することを考え、ノード PrecedesRule、followsRule、DispatchStmt に新たな属性 ruleInfo() を導入する。ruleInfo() はそれぞれのノードで呼ばれると、自ノードの子孫として持っているルールの情報を収集し、返すようなメソッドである。具体的には、ruleInfo() は JastaddJ に用意されているコレクションのクラスである SimpleSet クラスを用いて、各ルールを表現する RuleInfo オブジェクトの集合を返している。SimpleSet クラスは、要素の重複を許さない集合を現すクラスである。ruleInfo() はアスペクトとして jrag ファイルに記述し、PrecedesRule ノードと FollowsRule ノード、DispatchStmt ノードにおいて次のように定義する。

PrecedesRule ノード、FollowsRule ノードにおける ruleInfo() は、以下のような操作を行う。

1. 子ノード TargetEvent から、対象イベントの名前を得る。
2. 子として持つリスト RelatedEvent の全ての要素に対して、以下の操作を行う。
3. 関連イベントの名前を得る。
4. 対象イベント、関連イベントそれぞれの引数情報のリストを得る。
5. 対象イベント、関連イベントの EventInfo オブジェクトを生成する。
6. 生成した EventInfo オブジェクトを用いて RuleInfo オブジェクトを生成し、返り値として返す SimpleSet オブジェクトに追加する。

前述したように、ここで得られる引数情報は、result や follows ルールで記述された仮引数名の文字列のみであり、この時点ではイベントごとの引数と返り値の関係の全てを解析することはできない。そのため、EventInfo オブジェクトのコンストラクタはそのイベントの名前とそのイベントが持ち得る引数の個数だけを受け取るものとする。このとき、resultProviderList は空のままにしておく。各 EventInfo オブジェクトの resultProviderList の内容は、後述する DispatchStmt の WhileStmt への書き換えの際に EventInfo の集合を生成した後、RuleInfo オブジェクトの集合に含まれる各要素を参照しながら埋めていくことになる。

また、EventInfo オブジェクトを生成した後、生成した EventInfo オブジェクトを用いて RuleInfo オブジェクトが生成される。属性 ruleInfo() が返り値として返すのは、この RuleInfo オブジェクトの集合である。

DispatchStmt ノードにおける ruleInfo() は、次のような操作を行う。

1. 子として持つ Rule ノードのリストの全ての要素に対して、以下の操作を行う。
2. ruleInfo() を呼び、RuleInfo オブジェクトの集合を得る。
3. 2 で得られた集合の要素を、返り値として返す SimpleSet オブジェクトに追加する。

これにより、DispatchStmt において ruleInfo() を呼ぶことで、RuleInfo オブジェクトの重複のない集合が得られる。これは、dispatch 文に定義されたルールを表す RuleInfo オブジェクトを全て持つ集合である。

4.3.3　イベントのルールに従ったソート

DispatchStmt では、子ノードから回収したイベントやルールの情報から、最適なタイミングで同期処理が行えるようにイベントの計算順序を計算する。ここではイベントの計算順序を決定するために、トポロジカルソートを用いる。

トポロジカルソートとは、閉路のない有効グラフにおいて、全てのノードが、自身から出るエッジの指しているノードよりも先になるような順番付けを行うものである。一般に、トポロジカルソートには V をノード数、E をエッジ数として、O(|V|+|E|) の実行時間を要する。代表的なトポロジカルソートのアルゴリズムを次に挙げる。

Kahn のアルゴリズム　Kahn が 1962 年に発明したアルゴリズム [7] は、先頭に置くべきノードから考え始め、次に配置するべきノードを探していくというものである。具体的には、次のようにソートを行う。

まず、次のような集合 S とリスト L を用意する。

- 集合 S 開始ノードの集合
- リスト L ソート結果を保持するためのリスト

ここで開始ノードとは、自身に入るエッジがないようなノードを指す。集合 S、リスト L を用いて、以下のようにノードのソートを行う。

1. S からノードをひとつ選び、S から削除する。このノードを n とする。
2. L にノード n を追加する。
3. ノード n から出る各エッジ e に対して、以下の a)、b) を行う。
   a) エッジ e を元のグラフから削除する。
   b) エッジ e が指す先のノード m が、e 以外のエッジに指されていないならば、ノード m を S に追加する。
4. S が空であれば L を解として計算を終了し、そうでなければ 1 に戻る。

このソート方法では、各エッジは一度しか訪問されないが、ノードは複数回訪問される可能性がある。

深さ優先探索をベースとしたアルゴリズム　このアルゴリズムは、グラフが含む各ノードについて、ソートを開始してから既に訪れたノードに到達するまで深さ優先探索を行うものである。 [9]

具体的には、まず Kahn のアルゴリズムと同様に、次のようなリスト S とリスト L を用意する。

- リスト S 全てのノードのリスト
- リスト L ソート結果を保持するためのリスト

この上で、次のような操作を行うメソッド visit を定義する。ただし、visit メソッドは、引数としてノードを表すオブジェクトを取るものとする。

1. 引数として受け取ったノード (n とする) を既に訪問している場合、計算を終了する。
2. ノード n を訪問したとして、印を付ける。
3. ノード n に入ってくるような各エッジについて、その出力元となっているノード (m) に対して、visit(m) を呼び出す。
4. ノード n を L に追加する。

トポロジカルソートを行うためには、リスト S に含まれる全てのノードに対して、visit メソッドを呼び出せばよい。

このアルゴリズムでは、まだ訪問していないノードに対して visit メソッドが呼ばれたとき、そのノードが依存している、すなわちそのノードより先にリスト L に追加されるべきノードを前もって訪問する。これにより、あるノードがリスト L に追加されるときには、そのノードが依存するノードは全て L に追加された後であることが保障される。また、このソート方法では、全てのエッジとノードは高々1 回しか訪問されないため、Kahn のアルゴリズムよりも効率が良い。

JgraphT　PersianJ では、トポロジカルソートを行う際に JgraphT [8] を利用した。JgraphT とは Java のグラフライブラリであり、無向グラフ、有向グラフをはじめとした様々なグラフのオブジェクトや、それらを用いたアルゴリズムを提供する。JgraphT におけるトポロジカルソートは、上で述べた深さ優先探索をベースとしたアルゴリズムを用いて行われる。

例えばこのライブラリにおいて、有向グラフは図 4.4 のように簡単に生成することができる。このコードでは、2、3 行目で空の有向グラフを用意し、グラフを表すクラスに用意されている addVertex メソッド、addEdge メソッドでノードやエッジをグラフに追加している。また、ここで生成し

```
1  // Create new directed graph.
2  DirectedGraph<String, DefaultEdge> graph =
3    new DefaultDirectedGraph<String, DefaultEdge>(DefaultEdge.class);
4
5  // Define Nodes.
6  String [] eventList = new EventInfo[9];
7  eventList[0] = "dataEast.exchange";
8  eventList[1] = "dataWest.exchange";
9  eventList[2] = "dataEest.arrange";
10  :
11
12 // Add nodes to the graph.
13 graph.addVertex(eventList[0]);
14 graph.addVertex(eventList[1]);
15
16 // Add edges to the graph.
17 graph.addEdge(eventList[0], eventList[1]);
18  :
```

図 4.4: 有向グラフの生成

```
1  // Create iterator of the graph.
2  Iterator<EventInfo> iterator =
3    new TopologicalOrderIterator<String, DefaultEdge>(graph);
4
5  // Print nodes in sorted order.
6  while(iterator.hasNext()){
7    System.out.println(iterator.next());
8  }
```

図 4.5: 有向グラフに対するトポロジカルソート

た有向グラフに対して、トポロジカルソートを行った順番でノードを図 4.5 のように取り出すことができる。

2,3 行目では先のコード中で生成した有向グラフ graph に対するトポロジカルソートを行うイテレータを生成し、6-8 行目でこれを用いてノードを取り出している。

同期を考慮したエッジの追加　PersianJ では、単に与えられたルールから各イベントの計算順序を決定するのではなく、最も良いタイミングで同期処理が行われるように、その命令を挿入しなければいけない。そのため、JgraphT でトポロジカルソートを行う際、同期処理に関係するイベントと同期処理とのエッジをグラフに追加している。具体的には、次のような precedes 文に対して、result の有無に関係なく以下に示したエッジを追加する。

`async [対象イベント]() precedes [関連イベント]()`

対象イベント → [同期処理]

同期処理 → [関連イベント]

ただし、同期処理は同じ dispatch 文の中では一度だけ行われるものとする。従って、同期処理を表すイベントはひとつだけであると考える。follows 文に対しては、対象イベントと関連イベントとの間に同期処理は挟まれないので、上のような操作を行う必要はない。

また、3 章で述べたような計算のオーバーラップもこのソートで実現する。そのためには単に計算順序中に同期処理を挿入するだけでなく、同期処理と直接的にも間接的にも関係のないイベントをできるだけ同期処理の直前に挿入する必要がある。同期処理と間接的に関係があるイベントとは、同期処理を必要とするイベントに依存するイベントがあった場合、その「同期処理が必要なイベントに依存するイベント」に依存するようなイベントである。すなわち、あるイベントよりも前に実行されなければいけないイベントの中に同期処理があれば、そのイベントは同期処理と間接的に関連するイベントである。サンプルコードにおいては、calc.calcInner がこのようなイベントにあたる。PersianJ では、次のリストを用いてこのようなイベントを検索する。

- リスト L　解となるリスト。初期値では全イベントを保持する。
- リスト M　検索用のリスト。間接的であっても同期処理と関連のあるイベントを一時的に保持する。

1. 上記の precedes 文における関連イベント (e とする) 全てに対して、次の操作を行う。
2. イベント e が始点となっているようなルール r に対して、
   a) L から r の終点であるイベントを削除する。
   b) M が r の終点であるイベントを持たない場合、これを M に追加する。
3. イベント e を、M から削除する。

この操作により、リスト L には同期処理と直接的にも間接的にも依存関係にないようなイベントのみが残る。従って、このリスト L の各要素 e に対して、新たなエッジ e → [同期処理] を生成し、グラフに追加すればよい。

4.3.4　コード生成

イベントのソートの次には、そのソート結果から適切なコード生成を行う。正確には、適切なコードを表す AST 部分木を作成し、DispatchStmt

ノード以下の部分木と差し替える。前述したとおり、このとき DispatchStmt は WhileStmt に変換される。ノード WhileStmt は While 文を表すノードであり、条件式と While 文中の式 (または式の集合) を用意する必要がある。今回条件式には値 true を設定する。また、イベントのソート結果に基づいて、次に挙げる式を While 文中の式として設定する。

- イベントに対応したメソッド呼び出し
- 変数宣言
- 同期処理

以下では、それぞれの式をどのように While 文中に挿入するかを述べる。

メソッド呼び出しの挿入　While 文中には、各イベントに対応したメソッド呼び出しを、イベントのソート結果に応じた順番で挿入する必要がある。ただし、イベントの返り値が void 型以外である場合は、変数宣言と共にメソッド呼び出しを行い、変数にメソッドの返り値を代入しなければいけない。これについては次で述べる。すなわち、イベント A.a の返り値が void 型であれば

```
A.a(arg1, ...);
```

それ以外であれば

```
[A.a の返り値の型] result0 = A.a(arg1, ...);
```

というように、異なる式を挿入する必要がある。

メソッド呼び出しを生成する際に必要となる情報は、次の通りである。

- (メソッドの属する) オブジェクト名
- メソッド名
- メソッドが引数として受け取る値を保持する変数名のリスト

オブジェクト名とメソッド名は、EventInfo オブジェクトが保持するイベント名からそのまま得ることができる。

また、PersianJ ではコード生成の間に、イベント名とその返り値を保持する変数名との対応表を作成する。この対応表はイベント名をキー、それに対する変数名を値とする HashMap を用いて作成する。メソッドに引数として渡すべき値がどの変数に格納されているかは、この対応表により得られる。すなわち、あるイベントの引数について調べるとき、そのイベントを表す EventInfo オブジェクトが持つ resultProviderList 中の各要素 (イベント名) をこの対応表に渡せば、それらイベントの返り値を持つ変数名を得ることができる。

変数宣言の挿入　上で述べたように、返り値が void 型以外であるようなイベントは、その値を保存しておかなければいけない。そのためこのような場合には、変数宣言と同時にその変数にメソッド呼び出しの返り値を代入する式を生成する。サンプルコードにおいては、例えば次のような式が生成される。

```
double result6 = calc.calcInner();
```

このとき、変数名には result0、result1 というように、文字列 `result` に数字を付加したものを使用する。この数字は 0 から始まり、新たに変数宣言が必要となるたびに 1 ずつ増加する。また、このような式を生成する際には、上で述べた対応表に対して、自イベントのイベント名と、その返り値を持つ変数名の組を追加する。

同期処理の挿入　全てのイベントの中で、同期処理を表すイベントは単一であることは先に述べた通りである。従ってソート結果において、同期処理を表現するイベントが表れる箇所には、同期が必要なイベントに対する全ての同期処理を挿入する必要がある。

具体的には、まずルールの中から async を持つものを探し、同期処理の必要なイベントを得る。次に、そのイベントの返り値を持つ変数に対して sync メソッドを呼び出す式を生成する。サンプルコードにおいては、例えば次のような式が生成される。

```
result0 = dataEast.exchange();
 :
result0.sync();
```

同期処理の必要なイベントの返り値は、インターフェース AsyncResult を実装したものに限られているため、sync メソッドが定義されていることは保障されている。また、このときも目的の変数は、対応表に同期を必要とするイベントのイベント名を渡すことで得ることができる。

終了判定の挿入　イベントをソートした結果、最後に配置されるイベントの返り値は、計算ループの終了判定に使用される。すなわち、While 文が持つ最後の式は、最後に配置されるイベントの返り値を持つ変数を条件式として持つような if 文である。この if 文の本体は break 文である。例えばサンプルコードにおいて、計算ループを抜けるかどうかを判定するメソッドとして conditionChecker.checkBreak というメソッドが用意されている場合、次のような if 文が生成される。

```
result10 = conditionChecker.checkBreak(...);
```

```
if(result10){
  break;
}
```

ここでも対応表を用いて、最後に配置されるイベントの返り値を持つ変数の名前を得る必要がある。またこのようにすることで、最後に配置されるイベントの返り値が true であったときに、無限ループをなす While 文から抜けることができる。

第5章　PersianJを用いたSOR計算の実装例

ここまでで PersianJ の言語仕様と、その動作の様子について述べてきた。本章では、2 章で紹介したサンプルコードを実際に PersianJ を用いて書き直し、PersianJ を使わない場合と比べてどのような点で効果があったのか、考察を行う。

5.1　SOR メソッドの dispatch メソッドへの書き換え

サンプルコードの SOR メソッドは、図**??**のような dispatch メソッドに書き換えることができる。

以下では、2 章で紹介したサンプルコードを、図 5.1 に示したコードに書き直した際の変更点を述べる。

exchange メソッドの返り値の変更　元のサンプルコードでは、オブジェクト dataEast、dataWest、…、dataSouth が持つ exchange メソッドの返り値の型は double[] であった。これは、送受信によって得られた他プロセスのセルの値を、同オブジェクトの arrange メソッドに渡すためである。しかし、同期を必要とするメソッドの返り値は、図 4.2 に示した AsyncResult を実装するクラスのオブジェクトでなければいけない。そこで今回の実装では、送受信によって得られたデータは自オブジェクト内のフィールドに一度保存することにした。同期の後であれば、このフィールドには目的のセルの値が正しく保持されていることが保障される。従って同期後に arrange メソッドが呼び出された時点でこのフィールドの値を使って処理を行えば、サンプルコードと同じ動作を得ることができる。

Buffer クラスの導入　サンプルコードの sor メソッドでは、計算領域を表す 2 次元配列をそのまま引数として取っていたが、自プロセスの担当領域を持ち、ダブルバッファリングが行える Buffer クラスを定義した。この Buffer クラスのオブジェクトはプログラム中で 1 度だけ生成される。この単一の Buffer オブジェクトは直接 sor メソッドに渡される他、dataEast や calc などのオブジェクトそれぞれのフィールドに保存される。このよ

```
1  public dispatch void sorMethod(DealDataE dataEast,
2                                 DealDataW dataWest,
3                                 DealDataB dataBottom,
4                                 DealDataT dataTop,
5                                 DealDataN dataNorth,
6                                 DealDataS dataSouth,
7                                 CalcCells calc,
8                                 Buffer buffer,
9                                 DealCondition conditionChecker){
10
11   async dataEast.exchange() precedes dataEast.arrange(result);
12   async dataWest.exchange() precedes dataWest.arrange(result);
13   async dataBottom.exchange() precedes dataBottom.arrange(result);
14   async dataTop.exchange() precedes dataTop.arrange(result);
15   async dataNorth.exchange() precedes dataNorth.arrange(result);
16   async dataSouth.exchange() precedes dataSouth.arrange(result);
17
18   calc.calcInner() follows dataEast.exchange(),
19                            dataWest.exchange(),
20                            dataBottom.exchange(),
21                            dataTop.exchange(),
22                            dataNorth.exchange(),
23                            dataSouth.exchange();
24
25   calc.calcOuter() follows dataEast.arrange(),
26                            dataWest.arrange(),
27                            dataBottom.exchange(),
28                            dataTop.exchange(),
29                            dataNorth.exchange(),
30                            dataSouth.exchange();
31
32   buffer.flip() follows calc.calcInner(), calc.calcOuter();
33   calc.calcMaxResid(inner, outer) follows calc.calcInner() => inner,
34                                           calc.calcOuter() => outer;
35
36   conditionChecker.proceed() follows calc.calcInner(),
37                                      calc.calcOuter();
38   conditionChecker.checkBreak(maxResid)
39                           follows calc.calcMaxResid() => maxResid;
40 }
```

図 5.1: サンプルコードの dispatch メソッド

うな実装にすることで、例えば dataEast 内で buffer の持つ配列を変更したとき、その変更を calc など、他のオブジェクトも知ることができる。

Buffer クラスのフィールドは計算領域を表す 2 次元配列と、その配列中で現在注目している配列のインデックス値である。Buffer クラスの持つメソッドは以下のとおり。

- double[] getCurrent() … 現在注目している配列を返す。exchange メソッドや arrange メソッドなどで使用される。
- double[] getNext() … 新しい値を入れる配列を返す。calcInner メソッドや calcOuter メソッドで使用される。
- void frip() … 配列を交換し、新しい値の入った配列を次に注目するべき配列とする。sor メソッド内で使用される。

DealCondition クラスの導入　Buffer クラスと共に、そのときの計算ループの状態を扱うクラス DealCondition を導入した。このクラスは計算を繰り返した回数として int 型のフィールドのみを持ち、計算ループの終了判定も行う。DealCondition クラスの持つメソッドは以下のとおり。

- void proceed() … 計算を繰り返した回数を 1 増やすことで、状態を更新する。
- boolean checkBreak(double maxResid) … 計算を繰り返した回数と、引数として得た最大差分を用いて計算ループの終了判定を行う。

5.2　dispatch メソッドを変換して得られるコード

図 5.1 に示した dispatch メソッドとしての SOR メソッドをコンパイラ内で解析し、While 文を持つ通常のメソッドに変換された時に生成されるコードを図 5.2 に示す。このコード中のコメントは、判りやすいように後から書き加えたものである。

5.3　考察

ここでは、PersianJ の dispatch メソッドを用いて SOR 計算を実装した場合とそうでない場合とを比べて、PersianJ を用いた場合の効果について考える。

まず、ルールを使ってメソッド間の依存関係を指定するようにしたことによって、dispatch メソッドを用いた sor メソッドでは、計算ループ中の

```
1  public void sor(DealDataE dataEast,
2                  DealDataW dataWest,
3                  DealDataB dataBottom,
4                  DealDataT dataTop,
5                  DealDataN dataNorth,
6                  DealDataS dataSouth,
7                  CalcCells calc,
8                  Buffer buffer,
9                  DealCondition conditionChecker) {
10
11   while (true) {
12     // exchange each direction's surfaces
13     AsyncResult result0 = dataEast.exchange();
14     AsyncResult result1 = dataWest.exchange();
15     AsyncResult result2 = dataBottom.exchange();
16     AsyncResult result3 = dataTop.exchange();
17     AsyncResult result4 = dataNorth.exchange();
18     AsyncResult result5 = dataSouth.exchange();
19
20     // calculate at cells which don't need values from other nodes.
21     double result6 = calc.calcInner();
22
23     // synchronous processing
24     result0.sync();
25     result1.sync();
26     result2.sync();
27     result3.sync();
28     result4.sync();
29     result5.sync();
30
31     // Arrange received data into proper place
32     dataEast.arrange();
33     dataWest.arrange();
34     dataBottom.arrange();
35     dataTop.arrange();
36     dataNorth.arrange();
37     dataSouth.arrange();
38
39     // calculate at left over cells
40     double result7 = calc.calcOuter();
41
42     // switch buffer
43     buffer.frip();
44     // increase iteration number
45     conditionChecker.proceed();
46
47     // screen max residual
48     double result8 = calc.calcMaxResid(result6, result7);
49
50     // check residual
51     if(conditionChecker.checkBreak(result8)) {
52       break;
53     }
54
55   }
56 }
```

図 5.2: 図 5.1 を変換して作られるメソッド

```
1 debugger.outputCells(iter) follows
2                               conditionChecker.getIter() => iter;
3 conditionChecker.getIter() precedes conditionChecker.proceed();
```

図 5.3: dispatch メソッドへの追加コード

あるメソッドが必要とするデータがどのメソッドから得られるかが、1 つの式から明白に判る。これに伴って、dispatch メソッド内では変数宣言を行う必要がないので、どの変数がどのメソッドの返り値を持つのかという点について、プログラマは考慮しなくて良い。

また、dispatch メソッド内では、書き換え前のサンプルコードのように同期処理自体の記述を行わなくて良い。同期を行いたい場合は、同期処理を必要とするメソッドを指定するだけで良いのである。PersianJ では今まで述べてきたとおり、最適なタイミングで同期処理を行うように計算順序を構築する。従って、同期処理をどの時点で行うか、またどの Request オブジェクトに対して行うかを考えずにプログラマはコードを書くことができる。

PersianJ を用いた場合、メソッドの追加も容易になる。例えばデバッグ用のオブジェクト debbuger の持つ outputCells メソッドを追加して計算の様子を出力したいとき、次のコードを図 5.1 の dispatch メソッド中の好きな箇所に追加するだけでよい。新たに挿入するメソッドが同期を必要とする場合でも、precedes 文、follows 文の先頭に async を付加するという点を除いて、同様にメソッドの追加を行うことができる。ただし、反対にメソッドの削除を行いたいときには、そのメソッドを対象メソッドとしたルールを削除するだけでなく、そのメソッドを含むようなルールについても見直しが必要となり、PersianJ を使わない場合と同じく注意が必要となる。

更に sor メソッド本体の行数について考える。書き換える前と後の sor メソッド本体のコードのうち、計算に関係のある行のみを数えると、次のような結果が得られた。ただし、書き換え前の while 文において、変数宣言とその変数への代入は、合わせてひとつの行として考えた。

- 書き換え前 … 21 行
- 書き換え後 … 12 行

この結果によると、今回のプログラム例では、SOR 計算に関係のある部分においては 6 割までそのコード量を減らすことができたと言える。

以上から総括すると、PersianJ において計算ループ中にあるメソッドを挿入する際には、そのメソッドと依存関係にあるメソッドと、そのメソッ

ド自身についてのみ考えればよい。これにより、PersianJ を用いることで、計算ループを構築する間だけでなく、計算ループが完成した後に計算順序を変更する場合にも、より少ない負担でコードを書くことができるようになったとしてよいと考えた。

第6章　まとめと今後の課題

6.1　まとめ

本稿では、プログラム中に表れる計算ループ内のイベントの計算順序を関心事として分離し、同期処理が最適なタイミングで実行されるよう、自動で挿入する言語 PersianJ を提案し、実装した。これにより、プログラマ自身は同期のタイミングを考えることなく、適切なタイミングで同期処理を行うプログラムを実装することが可能になる。また、新たなイベントを計算ループ内に挿入する場合には、プログラマは追加したいイベントと、その関連イベントにのみ注目すればよい。

一般に、計算ループ中に計算順序をハードコーディングしてしまうと、チューニングやデバッグなどのために計算順序を変更することは煩雑な作業となり、プログラマにとって負担である。そこで PersianJ では、計算ループを示すメソッドとして dispatch メソッドや、dispatch メソッド中でのみ使用できる新たな構文として precedes 文、follows 文、あるイベントが同期を必要とすることを表すキーワード async を導入し、dispatch メソッド中でイベント間の依存関係、同期処理の有無を記述するようにした。そして dispatch メソッドに対して、コンパイル時に記述された内容から各イベントの計算順序を計算し、最適なタイミングで同期が行われるように同期処理を挿入することで、同期を考慮せずとも、最適なタイミングで同期処理が実行されるコードを書くことが可能になった。また、dispatch メソッド内で記述される新たな構文により、計算順序の変更のみならずイベントの追加も従来より容易に行えるようになった。

PersianJ は、JastAddJ を拡張することで実装した。JastAddJ は、コンパイラ実装フレームワークである JastAdd によって実装された Java コンパイラである。具体的には、JastAddJ に文法を追加し、構文解析の後に dispatch メソッドの本体を適切な While 文に変換することによって、本言語の目的である、イベントの実行順序の決定と同期処理の自動挿入を実現した。

6.2　今後の課題

プログラマの HPC プログラミングにおけるサポートを行う上で、PersianJ の現在の機能はまだ十分でないと考える。まず 4 章で述べたとおり、同期処理のタイミングの最適化において、現在の実装では、計算ループ中で複数の同期処理を一度にまとめて行うことを前提とした最適化しかできない。この方法では同期処理のタイミングを最適化しきれない例、また計算順序を決定できない例も存在する。従って、PersianJ ではこれに対して、同期処理の最適化をどのように行うかのカスタマイズを可能にすることを目標とする。

また、同期処理のタイミングの最適化以外にも、HPC プログラミングには煩雑な作業が伴う。例えば、MPI においてデータの送受信を行うとき、送信用データにはひと続きの配列しか使用することができない。しかし今回のサンプルコードのように 2 次元または 3 次元の計算領域を 1 次元の配列で表している場合には、通信を行う前に、送信用データを一時的な配列に配置し直さなければいけないことがある。また、更なる計算の並列化を目的として GPU を用いる場合、プログラム中には CPU 用のコード、GPU 用のコードが散在することになるため、プログラムはより複雑になってしまう。

これを受けて、PersianJ が今後実装するべき機能としては次のようなものが考えられる。

- データ通信の際の送信用配列の自動生成
- CPU、GPU 用コードの切り出しと切り替え

加えて、プログラムの実行中に計算ループ中にあるイベントの Dynamic scheduling を行うことができれば、プログラムのパフォーマンスの測定がより行いやすくなるのではないかとも考えている。

参考文献

[1] Bryan Carpenter, Aamir Shafi, M. B.: MPJ Express, `http://mpj-express.org/`.

[2] Carpenter, B., Getov, V., Judd, G., Skjellum, A. and Fox, G.: MPJ: MPI-like message passing for Java, *Concurrency: Practice and Experience*, Vol. 12, No. 11, pp. 1019–1038 (2000).

[3] Ekman, T. and Hedin, G.: The JastAdd extensible Java compiler, *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, New York, NY, USA, ACM, pp. 884–885 (2007).

[4] Ekman, T. and Hedin, G.: The JastAdd system - modular extensible compiler construction, *Science of Computer Programming*, Vol. 69, pp. 14 – 26 (2007).

[5] Gropp, W. and Lusk, E.: The Message Passing Interface (MPI) standard, `http://www.mcs.anl.gov/research/projects/mpi/`.

[6] John L. Hennessy, D. A.: コンピュータの構成と設計—ハードウエアとソフトウエアのインタフェース (第 2 版) 〈上〉, 日経 BP 社 (1999).

[7] Kahn, A. B.: Topological sorting of large networks, *Commun. ACM*, Vol. 5, pp. 558–562 (1962).

[8] Naveh, B.: JgraphT, `http://www.jgrapht.org/`.

[9] Tarjan, R. E.: Edge-disjoint spanning trees and depth-first search, *Acta Informatica*, Vol. 6, pp. 171–185 (1976). 10.1007/BF00268499.

[10] Team, T. J.: JastAdd, `http://jastadd.org`.

[11] 千葉滋: アスペクト指向入門　-Java・オブジェクト指向から AspectJ プログラミングへ, 技術評論社 (2005).