日本ソフトウェア科学会第 29 回大会 (2012 年度) 講演論文集

メソッド間の依存関係から適切な計算順序を生成する言語

宗 桜子　武山 文信　千葉 滋

一般にプログラム中に存在するメソッドの計算順序，すなわちメソッドを実行する順番は一意に定まるものではない．従って，プログラマはコードを書くとき，各メソッド間の全ての依存関係についても考えた上で，メソッドの計算順序を決定し，明示的に示さなければならない．この作業はプログラムが複雑になるほどプログラマにとって大きな負担となる．本研究ではこのような問題を解決するため，Java ベースの言語 PersianJ を提案する．PersianJ のコンパイラは，プログラマがメソッド間の最低限の依存関係を書くだけで，その依存関係を解析して各メソッド呼び出しを適切な順番に配置する．また，これに加えて，同期処理を必要とするメソッドが指定されている場合には，対応する同期処理の回数が最小となるように配置することもできる．

1　はじめに

一般に，プログラムを完成させるまでの過程で，プログラマはメソッド呼び出しの順番 (計算順序) を何度か変更する事がある．例えば，計算待ち時間の有効利用やキャッシュのヒット率向上などを考慮し，メソッド呼び出しの順序を変更するチューニング方法はよく知られたテクニックである．

しかし，計算順序には制約があるため，プログラマはメソッド間の依存関係を注意深く考慮しなければならない．一方で，計算順序の制約には必要な制約と不必要な制約があり，容易に判断するのは難しい．そのため，プログラム上でメソッド呼び出しの順序を変更する作業は煩雑であり，プログラマにとって大きな負担となっている．本研究ではこの問題を受け，プログラマが記述した最低限の制約から，それを満たす計算順序を決定する PersianJ を提案する．

A programming language generating appropriate execution order from method dependency

Sakurako Soh, Shigeru Chiba, 東京大学大学院情報理工学系研究科創造情報学科, Dept. of Creative Informatics, The University of Tokyo.

Fuminobu Takeyama, 東京工業大学大学院数理・計算科学科, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology.

2　メソッドの順序関係

プログラムを書く過程で，パフォーマンスを考慮して計算順序を変更する例として，拡散方程式を解くプログラムを挙げる．リスト 1 は，水面にインクを一滴落とした時に，そのインクがどのように水面上を広がっていくのかを単位時間ごとに計算するプログラムである．これは，大規模なシミュレーションを扱う High-Performance Computing (HPC) のプログラムを簡素化したものである．

HPC で扱う問題は一般的に問題サイズが大きいため，データ単位で分割して複数のノードに割り当てた後，並列計算を用いて解くというアプローチが取られる．また，並列計算の際には全てのノードで一斉にデータを交換しながら計算を行う．リスト 1 では，水面の状態を表した計算領域のデータ交換を exchange メソッドが非同期通信によって行い，この通信に対する同期処理を sync メソッドが行う．通信によって得られた，水面の各地点でのインクの濃度を表すデータは，arrange メソッドを実行することで自ノードの計算領域の適切な位置に格納される．また，リスト 1 の後半では，単位時間ごとの計算の繰り返しの終了判定に必要な値を calcMaxResid メソッドで計算し，その後 checkBreak メソッドで終了判定を行う．

リスト 1　拡散方程式のプログラム

```
1  public void sor(DealDataE east, DealDataW west,
2            CalcCells calc, Buffer buffer,
3            DealCondition conditionChecker){
4    while(true){
5      MPJ_AsyncResult eastResult = east.exchange();
6      eastResult.sync();
7      MPJ_AsyncResult westResult = west.exchange();
8      westResult.sync();
9
10     east.arrange();
11     west.arrange();
12
13     double innerResid = calc.calcInner();
14     double outerResid = calc.calcOuter();
15
16     conditionChecker.proceed();
17     double maxResid
18       = calc.calcMaxResid(innerResid, outerResid);
19     buffer.flip();
20
21     if(conditionChecker.checkBreak(maxResid))
22       break;
23   }
24 }
```

並列計算では，プログラムとして妥当であり正しい計算結果が得られても，実行速度が十分でない場合がある．例えば，リスト 1 では，通信の開始 (exchange メソッド) と同期処理 (sync メソッド) の間には何の計算も行われないため，実際に通信を行っている間の時間が無駄になってしまう．通信を行うとき，実際には exchange メソッドは全てのデータを交換するのではなく，各ノードが担当する領域のうち，他ノードの担当領域と隣り合う部分のデータのみを交換している．従って，担当領域のうちデータ交換を行わない部分の値だけを使う計算は，通信を行ったか行っていないかに関係なく行える．リスト 1 においては，この計算は calcInner メソッドが行う．これに対して，境界部分での計算，すなわち通信結果が必要な計算は calcOuter メソッドが行う．

一方，通信時間を有効利用するために，計算のオーバーラップを考慮した実装に書き換えるようなチューニングが可能である．計算のオーバーラップとは，通信している間，その通信と関係のない計算を行うことである．リスト 2 では，二つの exchange メソッドで通信を開始した後，sync メソッドで同期処理を行う前に，calcInner メソッドを実行することで通信と関係のない計算を先に終わらせている．

リスト 2　変更後の拡散方程式のプログラム

```
1  public void sor(DealDataE east, DealDataW west,
2            CalcCells calc, Buffer buffer,
3            DealCondition conditionChecker){
4    while(true){
5      MPJ_AsyncResult eastResult = east.exchange();
6      MPJ_AsyncResult westResult = west.exchange();
7
8      double innerResid = calc.calcInner();
9
10     eastResult.sync();
11     westResult.sync();
12
13     east.arrange();
14     west.arrange();
15
16     double outerResid = calc.calcOuter();
17     :
18     :
19   }
20 }
```

しかし，計算順序の変更は大変煩雑な作業であり，プログラマにとって大きな負担となっている．また，計算順序の制約は見ただけでは区別が難しいため誤って制約を壊してしまう事もある．一方で，既存の言語では全てのメソッドについて呼び出し順序，つまり計算順序を確定して記述しなければいけない．

メソッド間の計算順序に関する制約は，プログラムを実行して得られる結果やパフォーマンスに応じて，強い制約，弱い制約，無制約の 3 つに分類できる．制約の種類によっては，プログラマが書かずに済むようにすべきである．

まず，正常な動作を保証するために，プログラマが必ず満たすように考慮しなくてはならない，強い制約が存在する．例えばリスト 1 のプログラムでは，exchange メソッドで通信を開始してから，arrange メソッドを実行しなければならない．arrange メソッドは通信結果を利用するメソッドであるので，exchange メソッドとの前後関係を変えてしまうと，プログラム

が正常に動作しなくなってしまう．このような制約は必ず記述しなければならない．

次に，性能のためにプログラマが与えた，弱い制約が存在する場合もある．リスト 1 からリスト 2 への書き換えの例では，計算のオーバーラップを考慮し，calcInner メソッドを exchange メソッドの後，sync メソッドの前に実行するという制約を手作業で加えてある．このような制約は，性能を改善するためだけのものである．前述の exchange メソッドと arrange メソッドの関係とは異なり，計算の前後関係を入れ替えても結果は変わらない．

最後に，メソッド間に依存関係の制約が全く無いため，計算順序を変更しても結果が変わらない場合もある．例えば，通信を行う exchange メソッドと計算の終了判定を行う checkBreak メソッドを順番を変えて実行しても，プログラムを実行して得られる結果やパフォーマンスに変化はない．このような制約は，プログラマが書かずに済むようにすべきである．

リスト 3　dispatch メソッドの例

```
1  public dispatch void sor(DealDataE east,
2                           DealDataW west,
3                           CalcCells calc, ...){
4
5      async east.exchange() precedes east.arrange();
6      async west.exchange() precedes west.arrange();
7
8      calc.calcOuter() follows east.arrange(),
9                               west.arrange();
10
11     buffer.flip() follows calc.calcInner(),
12                           calc.calcOuter();
13
14     calc.calcMaxResid(inner, outer) follows
15                       calc.calcInner() => inner,
16                       calc.calcOuter() => outer;
17
18     conditionChecker.proceed() follows
19               calc.calcInner(), calc.calcOuter();
20     conditionChecker.checkBreak(maxResid) follows
21                      calc.calcMaxResid() => maxResid;
22
23 }
```

3 PersianJ の提案

前章で指摘した問題を回避するために，我々は PersianJ を提案する．PersianJ は，プログラマがメソッド間の計算順序を変更するとき，プログラムの性能に直接関係する計算順序に集中できる言語である．

この言語は，各メソッド間の計算順序の制約を明示した，Java ライクな実装を可能とする．これに対して，コンパイラは，記述された制約を満たすように自動的に計算順序を決定し，メソッド呼び出しを配置する．リスト 1 のコードは，PersianJ を用いるとリスト 3 のように書き直すことができる．

PersianJ では，メソッドの計算順序を次のように記述する．

- 計算の依存関係に伴う計算順序の制約は，直接記述する．
- プログラマがよりパフォーマンスの良い計算順序を考え出すこともあるため，プログラムの性能に関する順序関係を直接記述するかはプログラマに選ばせる．プログラマが記述しない場合には，コンパイラが自動的にある程度パフォーマンスの良い計算順序を生成する．
- 本質的な依存関係がないメソッド間においては，計算順序を記述しなくて良い．

3.1 dispatch メソッドの導入

PersianJ では，まず，メソッド内の計算順序が自動的に決定されるメソッドとして，dispatch メソッドを導入した．ここでいう計算順序とは，メソッドの呼び出し順序のことである．dispatch メソッドは，リスト 3 に示したように dispatch 修飾子の付いたメソッドであり，計算の繰り返し部分を抽象化したものである．

メソッド間の計算順序の制約は，以下に示す precedes / follows 構文を使って必要な分だけ dispatch メソッド内に記述する．dispatch メソッドでのメソッド呼び出しの順序は記述された precedes / follows 文に基づいて決められ，メソッドはそれに従って実行される．例えばリスト 3 のように dispatch メソッドを記述すると，リスト 2 に示したコードと同じ動作が得られる．

precedes 文

precedes 文は，あるメソッドを他のメソッドより先に実行する場合に使う．例えば east.exchange メソッドを east.arrange メソッドより先に実行し，更に east.exchange メソッドの返り値を east.arrange メソッドに渡す必要がある場合，precedes，result キーワードを用いて次のように記述する．

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

follows 文

follows 文は，あるメソッドを他のメソッドより後に実行する場合に使う．例えば calc.calcMaxResid メソッドを calc.calcInner メソッド，calc.calcOuter メソッドより後に実行し，更に calc.calcInner メソッド，calcOuter メソッドの返り値を calc.calc メソッドに引数として与える必要がある場合，follows，=> キーワードを用いて次のように記述する．

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

precedes 文と follows 文は使い分ける必要がある．あるメソッドに複数の値を引数として与えるときには，precedes 文でなく follows 文を用いる必要がある．follows 文では，対象メソッドに，どのメソッドの返り値をどの順番で与えるかを指定することができる．例えば，precedes 文を用いて

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

としても，これでは calcInner メソッドと calcOuter メソッドそれぞれの返り値が，calcMaxResid の 2 つの引数のうちどちらに渡されるべきなのかが表現できない．

3.2 async 修飾子の導入

dispatch メソッド，precedes / follows 構文の他にも，async 修飾子を導入した．プログラマが async 修飾子によって同期処理を必要とするメソッドを指定すると，コンパイラが計算のオーバーラップについて考慮したうえで，同期処理の回数が最小になるように計算順序を決定する．

async 修飾子を使用する場合には，precedes 文を用いる必要がある．async 修飾子は，同期処理を必要とするメソッドと，同期処理を終えた後に実行するべきメソッドの計算順序を定める時にのみ必要となる修飾子である．follows 文はあるメソッドとそのメソッドの前に実行されるメソッドを指定する構文であるので，async 修飾子を用いても意味がない．例えば，次の precedes 文は follows 文で書き換えることができない．

```
async east.exchange()
     precedes east.arrange();
```

3.3 実装

PersianJ は，コンパイラのフレームワークである JastAdd [1] を用いて Java を拡張することで実装した．この言語では，コンパイル時に precedes / follows 文によって表されたメソッド間の依存関係を有向グラフで解析し，distaptch メソッド内のメソッド呼び出しの順番を決定する．その後，dispatch メソッドに対応する部分木を，決定した計算順序に基づいて標準の Java で作った部分木で置き換えることで，目的の動作を実現している．また，TSUBAME2.0 (Oracle Java SE 7 u5) 上で実際に正しく動作することを確認した．

4 まとめ

本研究では，プログラマによって記述されたメソッド間の最低限の計算順序の制限から，計算順序を自動で決定して実行する言語として，PersianJ を提案，実装した．実装は JastAdd を用いて Java を拡張し，コンパイル時に dispatch メソッドに対応する部分木を適切な部分木に置き換えるようにすることで行った．これにより，プログラマは dispatch メソッド内に precedes，follows 構文を用いて必要最低限の制約を記述できるようになり，性能に直接関係のある計算順序に集中してコードの計算順序を変更することが可能になった．

参考文献

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