Vol.2010-SE-167 No.19
2010/3/18

非同期呼び出しモデルにおける遅延評価を用いた同期法の研究

打越　智之[†1]　　杉山　安洋[†2]

分散型ソフトウェアにおけるマルチスレッド処理の一つに非同期メソッド呼び出しモデルがある．非同期呼び出しモデルによるマルチスレッド処理の難しさは，最適な同期場所を見つけ出すことが難しい点である．同期場所の選択を誤ると，並列に実行できる処理が減り，並列化による処理速度の向上が実現できない．本稿では，遅延評価を用いて，最適な同期場所を見つける手法を提案する．

A Thread Synchronization Mechanism for Asynchronous Method Invocation Model Using Lazy Evaluation

Tomoyuki Uchikoshi[†1] and Yasuhiro Sugiyama[†2]

Asynchronous method invocation model is one of thread control models for distributed software. In asynchronous invocation model, it is hard to find proper synchronization points for asynchronously invoked methods. Improper synchronization points result in low parallelism, and poor improvement of execution speed. This paper will propose a thread synchronization mechanism using lazy evaluation to find proper synchronization points.

1. はじめに

近年インターネットの普及により，分散システムの実用化が急速に進んでいる．しかし，分散システムにはネットワーク処理やマルチスレッド処理等が必要となり，スタンドアロンのシステム開発よりも難しくなる．そこで，我々の研究室では，分散型ソフトウェアの開発を容易に行えるようにするために TRMI(Transparent Remote Method Invocation)[4) を開発している．TRMI を用いると，スタンドアロンのソフトウェアを分散環境で実行できるようにネットワーク処理を自動的に追加することができる．

分散システムを開発する目的のひとつに，分散環境上に存在する複数の計算機リソースを活用した処理速度の向上があげられる．しかし，ソフトウェアへマルチスレッド処理をあらかじめ記述しておかなければ，TRMI を用いて分散型ソフトウェアを開発したとしても，複数の計算機リソースを有効活用することは難しい．分散化による実行速度の向上を期待する場合は，ソフトウェアへのマルチスレッド処理の記述が必須となり，開発者にとっては負担となってしまう．

そこで，本研究では TRMI での分散化に適したマルチスレッド処理をプログラムに容易に追加出来るようにすることを目的としている．本稿では，2 節で TRMI について述べる．3 節では非同期呼び出しモデルについて述べ，4 節では遅延評価を用いた非同期呼び出しモデルの同期法について述べる．5 節では非同期呼び出しモデルのマルチスレッド処理のプログラムへの追加方法について述べ，6 節では本手法を元にプログラムにマルチスレッド処理を追加する変換ツールの開発について述べる．7 節では関連研究について述べ，最後に，まとめと今後の課題を述べ本稿を締める．

2. TRMI

2.1 TRMI の概要

TRMI とは分散型ソフトウェアを開発する為の技術である．スタンドアロンソフトウェアを元に分散型ソフトウェアを開発することが出来る．分散型ソフトウェアに必要となる処理としてネットワーク処理が挙げられる．TRMI はスタンドアロンソフトウェアに合う形でネットワーク処理部品を生成する．

図 1 を用いて TRMI を用いて開発した分散型ソフトウェアの処理方式について述べる．サーバオブジェクトが持つメソッドをクライアントオブジェクトが呼び出すスタンドアロンソフトウェアがあったとする．TRMI を用いると，クライアント用ネットワーク処理部品とサーバ用ネットワーク処理部品を TRMI が生成する．クライアントオブジェクトとサーバオブジェクトを別の計算機上に配置しプログラムを実行すると，クライアントオブジェクトはサーバオブジェクトのメソッドを呼び出す代わりにクライアント用ネットワーク処理部

†1 日本大学大学院工学研究科情報工学専攻
Graduate School of Engineering, Nihon University
†2 日本大学工学部情報工学科
Department of Computer Science, College of Engineering, Nihon University

© 2010 Information Processing Society of Japan

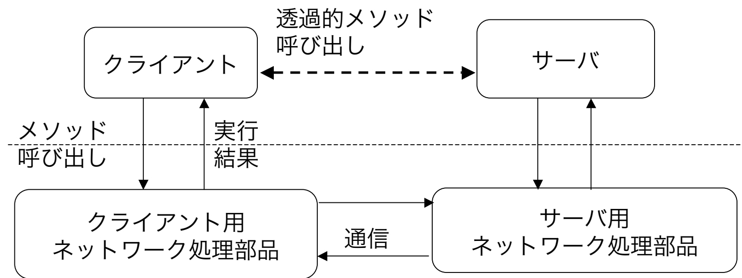

図 1 TRMI の処理方式

品のメソッドを呼び出すようになる．クライアント用ネットワーク処理部品はサーバ用ネットワーク処理部品と通信を行う．通信が行われるとサーバ用ネットワーク処理部品はクライアントオブジェクトが行っていたサーバオブジェクトのメソッド呼び出しを代わりに行う．このように TRMI を用いて開発した分散型ソフトウェアの分散した処理とは，メソッド呼び出しとなる．

2.2 マルチスレッド処理の必要性

分散型ソフトウェアを高速に実行するためにはマルチスレッド処理が必要である．何故なら，マルチスレッド処理が無ければ分散した処理を並列に実行できず，複数の計算機リソースを効率よく使用できないからである．

図 2 を用いて説明する．サンプルプログラムは Client オブジェクト，Server1 オブジェクト，Server2 オブジェクトが別々の計算機上に配置されており Client オブジェクトが Server1, Server2 オブジェクトのメソッドを呼び出すものである．マルチスレッド処理がない場合，図の左のように Server1 オブジェクトのメソッドを呼び出している間は Server2 の処理は実行されない．もしマルチスレッド処理があれば，右の図のように Server1 と Server2 のメソッドを同時に呼び出すことも可能となり，計算機を並列に動かせることとなり計算速度の向上が期待できる．ただし，Server1 の処理と Server2 の処理を並列に実行しても整合性が保たれるものとする．

3. 非同期呼び出しモデル

2 節で述べた TRMI の処理方式に適したスレッド制御として非同期呼び出しモデル[1)3)]を挙げる．

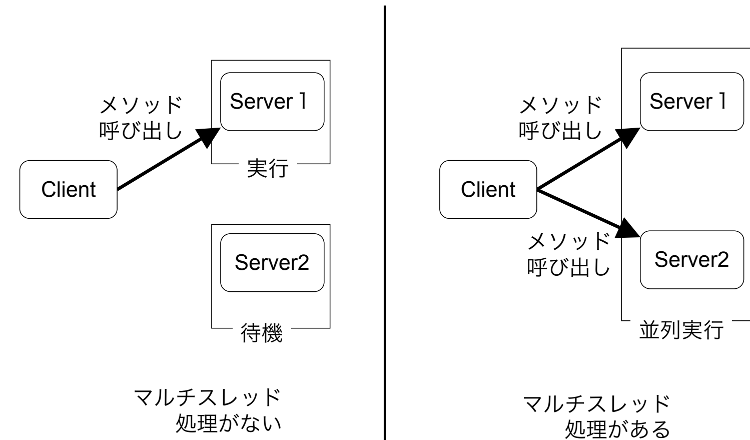

図 2 分散型ソフトウェアへのマルチスレッド処理の必要性

非同期呼び出しモデルとは，スレッドモデルの一つである Fork/Join モデル[2)] を拡張したものである．メソッドの呼び出し側が，呼び出したメソッドの終了を待たずに次の処理に移ることを指す．次の処理の中に別のメソッド呼び出しがあれば，そのメソッド呼び出しも非同期で呼び出すことで，複数のメソッド呼び出しを並列に実行することが可能となる．同期場所は，非同期に呼び出したメソッドの戻り値を使う処理の直前となる．

図 3 のサンプルプログラムは，処理 S1，メソッド呼び出し 1，メソッド呼び出し 2，処理 S2，処理 S3 という順に実行するものである．メソッド呼び出し 1 とメソッド呼び出し 2 の戻り値は処理 S3 で使うものとする．このプログラムに非同期呼び出しモデルを適用するためには，メソッド呼び出し 1 とメソッド呼び出し 2 を非同期に呼び出し，S3 の処理の直前で同期するようにすれば良い．すると図の右のように，メソッド呼び出し 1，メソッド呼び出し 2，処理 S2 が並列に実行される．処理 S3 の直前で同期を行い，メソッド呼び出し 1，メソッド呼び出し 2，処理 S2 が終了した後，処理 S3 が実行される．

非同期呼び出しモデルのスレッドモデルを使う場合には，非同期に呼び出すメソッドとその後の処理を並列に実行する際のプログラムの整合性を開発者が保証する必要がある．この例の場合，メソッド呼び出し 1，メソッド呼び出し 2，処理 S2 を並列に実行しても整合性が保たれる必要がある．

© 2010 Information Processing Society of Japan

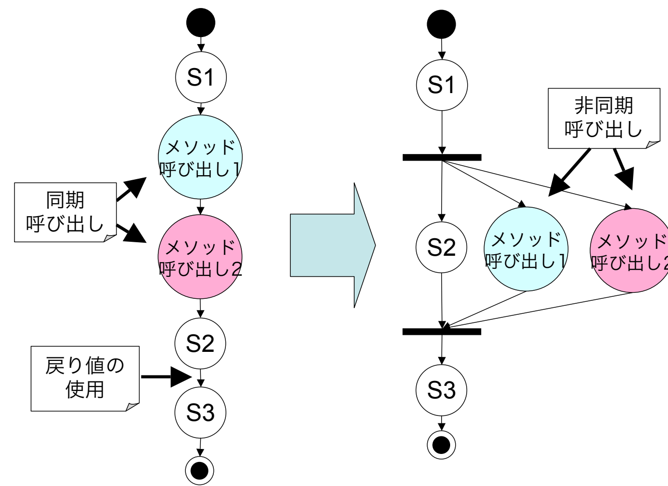

図 3 非同期呼び出しモデル

4. 遅延評価を用いた同期法

3 節で，非同期呼び出しモデルの同期場所として，非同期呼び出しの戻り値を使う処理の直前を挙げた．しかし，メソッドの戻り値を使う最初の処理というのはメソッド呼び出しが終了した直後に戻り値を変数に代入する処理であることが多い．しかし，この時点で同期をとってしまうと並列処理をほとんど行えない．そこで，遅延評価と呼ばれる評価法を用いた同期方式を提案する．

4.1 遅 延 評 価

一般に遅延評価とは式の値が必要になるまで，式の計算を行わないことを言う．図 4 のサンプルプログラムを用いて説明する．図の左のプログラムは次の処理を順々に実行している．変数 a に 5 を代入し，変数 b に 7 を代入し，変数 c に 9 を代入し，変数 d に変数 a の値と 8 を加えた値を代入する．ここで変数 d に代入する処理は，変数 a に 5 を代入する処理に依存しているが，他の処理の実行結果には影響を受けない．従って，変数 a に 5 を代入する処理を変数 d に代入する処理の直前まで遅らせても，プログラムの実行結果が変わらない．このように遅延評価を用いると，処理の実行を，その処理に依存する処理が分かれば遅らせることが可能となる．

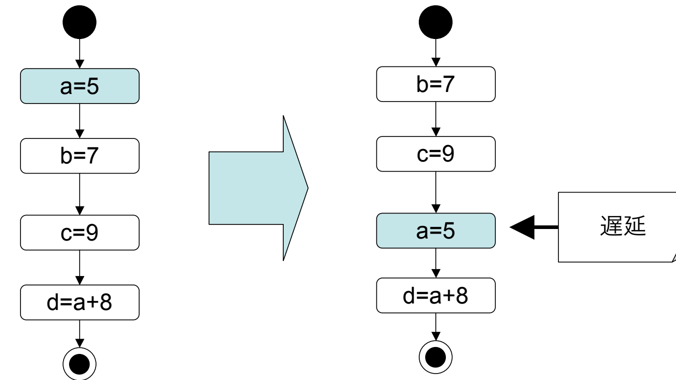

図 4 遅延評価の例

4.2 同 期 方 式

遅延評価を用いた同期法とは，遅延評価を用いて，非同期呼び出しモデルの同期場所を出来るだけ遅らせようとするものである．非同期呼び出しモデルでは，非同期に呼び出したメソッドの戻り値を使う処理の直前で同期を行う．そこで，戻り値を使う処理を遅延評価を用いて遅延させ，その戻り値を使う処理の直前で行う同期も遅延させる．

戻り値を使う処理を大きく分けると，戻り値を計算に使う処理と計算に使わない処理である．遅延評価を用いた同期方式では，戻り値を計算に使わない処理を遅延させる．戻り値を計算に使う処理の直前で同期を行う．

戻り値を計算に使わない処理の例としては，戻り値を変数に代入する処理，戻り値をメソッド呼び出しの引数に渡す処理，戻り値を return 文に渡す処理などが挙げられる．その場合，例えば変数に代入する処理を考えると，変数には本来は戻り値が格納されているはずであるが，別の値が入っていることとなる．代入処理で同期を行わない以上，この変数を使って計算をする処理が発生した場合に，その直前で同期処理を行い，そして遅延した変数に戻り値を代入する処理を実行する必要がある．

具体的に説明すると，

a=f()

といった変数 a に f メソッドの戻り値を格納する文があった場合，f メソッドを非同期に呼び出した場合には，メソッド呼び出し自体は遅延されず，変数 a に戻り値を代入する処理が遅延される．変数 a から値を取り出し計算に使う処理が発生したら，その直前で非同期呼

ⓒ 2010 Information Processing Society of Japan

Vol.2010-SE-167 No.19
2010/3/18

び出しの同期を行い，次に遅延した戻り値を変数 a に代入する処理を実行し，最後に変数 a から値を取り出す処理が行われる．

4.3 同 期 例

同期場所の具体例を，図 5 のサンプルプログラムを用いて説明する．このサンプルプログラムは for 文内で Server オブジェクトの call というメソッドを呼び出している．戻り値を配列 array に格納し，配列 cache にも戻り値を格納し，後の処理で array 配列内の値を全て加算し，最後に出力している．call メソッドを非同期で呼び出す場合の同期場所は，遅延評価を用いない場合，最初に戻り値を使う配列 array に代入する処理の直前となる．しかし call メソッドの戻り値を配列 array に代入する処理は，戻り値を使って計算している処理ではない．また，配列 array の値を配列 cache に代入する処理も，戻り値を使って計算している処理ではない．そのため遅延評価を用いた同期法では，これらの代入処理を遅延させ，その直前で行う同期も遅延させる．

実際に戻り値を計算に使っている処理は，２つ目の for 文内の配列 array 内の値を変数 sum に加算する処理であり，その直前で同期と遅延した array への代入処理の実行を行うこととなる．このように遅延評価を用いることで，同期場所を遅らせ並列処理区間を増やすことが可能となる．この例の場合は，100 個のサーバ sv を別々の CPU に割当てることにより，100 個のメソッド呼び出しを並行して実行できる．

5. プログラムへの非同期呼び出しモデルの追加

5.1 非同期呼び出しモデルのマルチスレッド処理

本節では，非同期呼び出しモデルのマルチスレッド処理をプログラムに追加する方法について述べる．本手法は，AspectJ を用いて非同期呼び出しモデルのスレッド制御をプログラムに織り込む方法[3] を基にしている．プログラムに非同期呼び出しモデルのマルチスレッド処理を追加したい場合には，非同期呼び出し処理，同期処理，関連付け処理と呼ぶ処理をプログラムに埋め込む必要がある．これらの処理を行うために埋め込む部品を，これ以降は非同期呼び出し部品と呼ぶ．

非同期呼び出し処理とは，もともとはシングルスレッドで呼ばれていたメソッドを非同期で呼び出す為の処理である．新たにスレッドを生成し，スレッドがメソッドを呼び出すのに必要な情報を渡し，実行を開始させる．これらの一連の処理がメソッド呼び出しを行う処理の代わりに埋め込まれる．

同期処理とは，非同期呼び出しを行ったスレッドと同期を行うための処理である．メソッドが終了するまで待機する処理と，戻り値をスレッドから取得する処理で構成されている．戻り値を計算に使う処理の直前に埋め込まれる．

```
public static void main(String[] args){
   int[] array = new int[100];
   int[] cache = new int[100];
   int sum = 0;
   Server[] sv = new Server[100];
   for(int i=0;i < 100;i++) {
      sv[i] = new Server();
      array[i] = sv[i].call(i);
      cache[i] = array[i];
   }
                  .
                  .
                  .
   for(int i=0;i < 100;i++) {
      sum += array[i];
   }

   System.out.println(sum);
}
```

```
class Server{
   public Server(){
   }

   public int call(int n){
      //重い計算処理
   }
}
```

図 5 同期例

関連付け処理とは，処理を遅延させる場合に，その情報を保持するための処理である．処理を遅延する場合に，その処理の代わり実行されるように埋め込まれる．保持した情報は，遅延した処理が遅延先で正しく実行できるように使われる．

5.2 変換の具体例

次に，5.1 節で述べた非同期呼び出し部品の具体的な埋め込み手法について述べる．非同期呼び出し部品は，Threader クラスと MethodThread クラスで構成している．Threader クラスは，非同期呼び出し処理，同期処理，関連付け処理を行う為のメソッドを持つ．

処理が埋め込まれたプログラムのシーケンス図のサンプルを図 6 に示す．サンプルは，Client オブジェクトが Server オブジェクトの call メソッド呼び出すプログラムを，call メソッドを非同期に呼び出すように変換したものである．Client オブジェクトの Server オブジェクトのメソッドを呼び出す処理には，Threader オブジェクトの非同期呼び出し処理の

ⓒ 2010 Information Processing Society of Japan

メソッド呼び出しが代わりに埋め込まれる．これにより非同期でメソッド呼び出しを行う MethodThread スレッドが新たに生成され実行されるようになる．MethodThread スレッドは Client オブジェクトに代わって Server オブジェクトの call メソッドを呼び出す.

Client オブジェクトには，call メソッドを呼び出した後に，戻り値を配列 array に代入する処理がある．この代入処理は，戻り値を計算に使っていないので遅延され，代わりに関連付け処理が埋め込まれる．関連付け処理は，Threader オブジェクトに配列 array の名前と添え字，現在実行を行っているスレッド id，実行中のメソッドを持つオブジェクトを渡す処理である．

Client オブジェクトは，その後に配列 array の値を変数 sum に加算する処理を持つ．配列 array には戻り値が入っているので，戻り値を使って計算を行う処理である．そのため，その直前に同期処理が埋め込まれる．さらに遅延した処理である配列 arary に戻り値を代入する処理が埋め込まれる．この二つが埋め込まれ，同期処理，遅延した処理，変数 sum に配列 array の値を加算する処理という順に実行されるようになる．

これらの処理が埋め込まれることで，図中の Client の並列処理区間と Server の並列処理区間が並列に実行される．

5.3 非同期呼び出しモデルの分散型ソフトウェア

シングルスレッドのプログラムに非同期呼び出しモデルのマルチスレッド処理を追加し，更に TRMI を用いて分散型ソフトウェアにすると図 7 のように Client オブジェクトは Server オブジェクトにネットワーク処理部品の他に，非同期呼び出し部品を経由して間接的にメソッド呼び出しを行うようになる．Threader オブジェクトや MethodThread オブジェクトは Client オブジェクト側にある．複数のメソッド呼び出しを非同期に行うと MethodThread が複数生成され，それぞれの MethodThread オブジェクトがネットワーク処理部品を用いてメソッド呼び出しを行う．これにより，サーバオブジェクトを別々の計算機で並列に処理することが可能となる．

6. 変換ツール

6.1 変換ツールの概要

本手法をもとに，変換ツールの開発を行っている．ツールの概要を図 8 に示す．このツールを用いると，Java 言語を用いて開発したプログラムは並列処理を行うように変換される．ツールを使用する際は並列処理させたいプログラムのバイトコードを用意する必要がある．ツールは，バイトコードを読み込み，非同期呼び出し部品を追加したバイトコードを生成

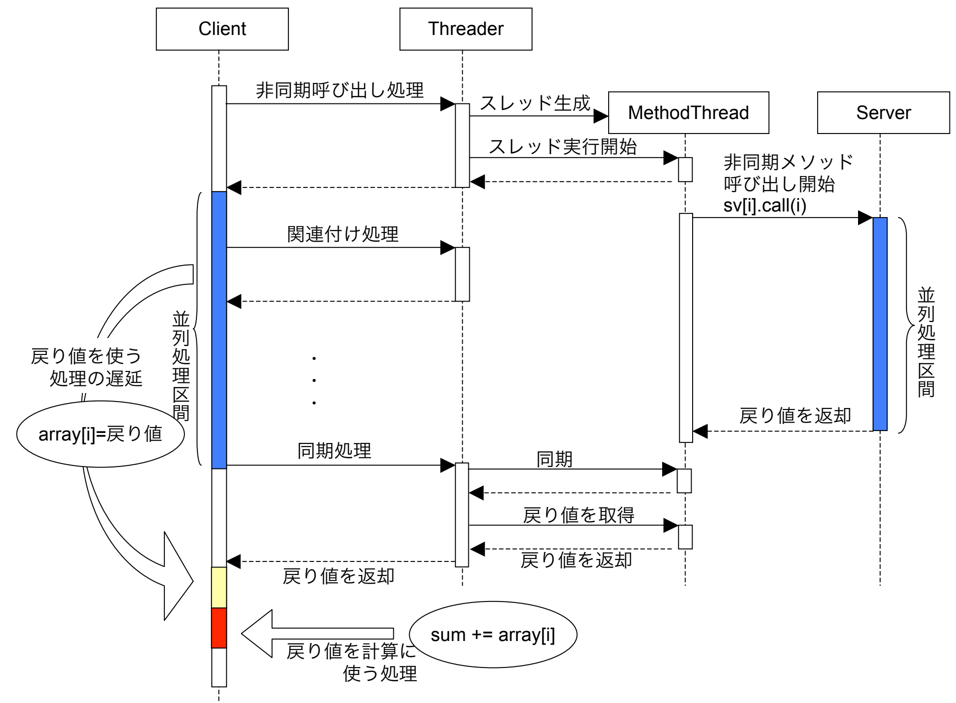

図 6 変換後のプログラムのシーケンス図の例

する．

ツールを実行すると，用意したバイトコードのプログラムを解析し，プログラム中に存在するメソッド呼び出し命令を表示する．ユーザは，非同期で呼び出したいメソッド呼び出しを指定する．指定されたメソッド呼び出しを非同期で呼び出すように，ツールは非同期呼び出しモデルに関する処理をプログラムに追加する．同期に関しても前節の同期法を用い，非同期に呼び出すメソッドの戻り値を使って計算している処理の直前に，同期処理と遅延させた処理が埋め込まれる．

6.2 遅延評価を用いた同期法の問題点

遅延評価を用いた同期法を用いてプログラムに同期処理を追加する場合，いくつか問題が発生する．まず一つ目として，分散型ソフトウェアに同期処理を追加する場合に，同期処理

© 2010 Information Processing Society of Japan

Vol.2010-SE-167 No.19
2010/3/18

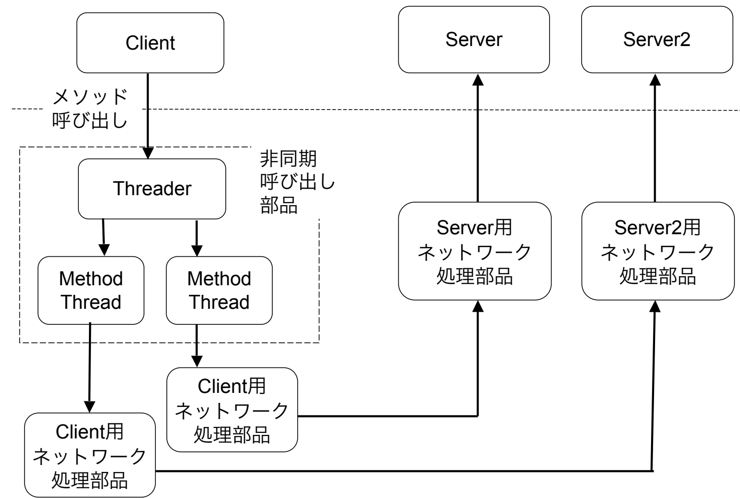

図 7 非同期呼び出しモデルの分散型ソフトウェア

が分散する問題が起こる．図 7 を見ると非同期呼び出し部品はクライアントオブジェクト側にしかないことがわかる．しかし，遅延評価を用いた同期法では，同期をクライアントオブジェクト中だけでなく，サーバオブジェクト内まで遅らせる場合が出てくる．図 9 にそのサンプルプログラムを示す．このサンプルプログラムでは，call メソッドの戻り値を Sum クラス内で使っている．call メソッド呼び出しを非同期に行う場合，Sum クラス内の add メソッド内の変数 all への加算処理の直前が同期場所となる．しかし，Sum クラスもサーバオブジェクトとしてクライアントオブジェクトと違う計算機上に配置すると，同期処理を MethodThread スレッドが動いている計算機と違う計算機上で行わなければならなくなる．

二つ目として，同期場所を探す解析処理をプログラム実行前に行うことによる問題が発生する．例えば，インスタンス変数に対し非同期に呼び出したメソッドの戻り値を入れる場合に，どのプログラムがそのインスタンス変数を参照するのか実行前に特定しきれない．その為，同期処理をどこまで遅延すればいいのか問題となる．他にも，インタフェースのメソッド呼び出しを行うプログラムの場合に，実装したクラスが実行時まで分からず，同期処理の遅延の解析が行えない．

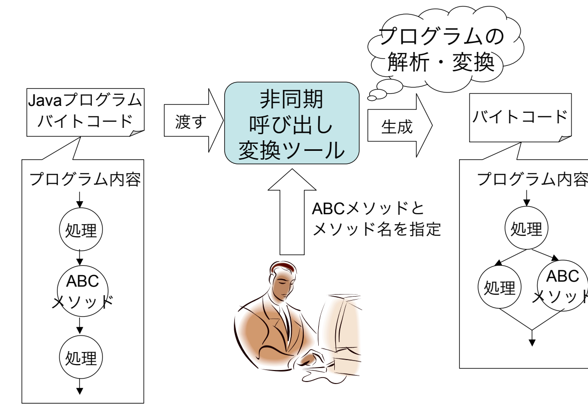

図 8 変換ツール

6.3 ツールの制約事項

遅延評価を用いた場合，いくつかの問題点が発生する可能性があると 6.2 節で述べた．そこで，ツールの解析機能に制約を設けることで，これらの問題を発生させないようにする．

一つ目の，同期処理を複数の計算機にまたがって行う必要がでる問題点は，同期処理の追加を非同期呼び出しを行うクラス内のみに限定することにより解決する．他のクラスまで同期処理が可能であっても，他のクラスの処理に移る直前を同期場所とし同期処理をその直前に追加する．これにより，同期処理が MethodThread スレッドが存在しない計算機上で行われることがなくなる．

二つ目の，実行時にならないと解析を行う必要のあるクラスが分からないという問題点は，実行時に同期処理を追加する場所を解析する方式をとれば問題とならない．しかし今回のツールは静的にバイトコードを解析している．その為，具体的にクラスが分からない場合には，処理をそのクラスまで遅延させないという方法をとる．例えば，インスタンス変数へ非同期に呼び出したメソッドの戻り値を代入する場合は，そのインスタンス変数を参照するクラスが特定しきれないのならば，インスタンス変数への戻り値の代入を遅延させず，代入直前で同期を行う．同様にインタフェースのメソッド呼び出しの引数に非同期に呼び出したメソッドの戻り値を渡す場合には，その時点でクラスが分からなければ，インタフェースのメソッド呼び出しの直前で同期を行う．

© 2010 Information Processing Society of Japan

```
public static void main(String[] args){
    int[] array = new int[100];
    Server[] sv = new Server[100];
    Sum sm = new Sum();

    for(int i=0;i < 100;i++)  {
        sv[i] = new Server();
        array[i] = sv[i].call(i);
    }
                    .
                    .
                    .
    for(int i=0;i < 100;i++)  {
        sm.add(array[i]);
    }

    System.out.println(sm.getSum());
}
```

```
class Server{
    public Server(){
    }
    public int call(int n){
        //重い計算処理
    }
}
```

```
class Sum{
    int all;
    public Sum(){
        all = 0;
    }
    public void add(int n){
        all += n;
    }
    public int getSum(){
        return all;
    }
}
```

図 9 同期処理が分散するサンプルコード

6.4 変換ツールの実装概要

ツールは渡されたプログラムのバイトコードの解析と変換を行っている．その際，次に示すバイトコード命令に着目し解析を行っている．メソッド呼び出しを行う invoke 命令，ローカル変数に対し入出力する load 命令と store 命令，インスタンス変数に対し入出力する putfield 命令と getfield 命令，メソッドを終了しメソッドの戻り値を返す return 命令等である．まず，非同期呼び出しを行わせる invoke 命令を探す．invoke 命令を行った際の戻り値をどのように使うかは直後の命令を見ることで分かる．戻り値を変数へ代入する場合は，store 命令や putfield 命令がある．戻り値を引数にメソッド呼び出しを行う場合は，invoke 命令がある．戻り値を return 文に渡す場合は，return 命令がある．これらの命令は戻り値を計算に使わない命令とみなしている．その他のバイトコード命令がある場合は，戻り値を計算に使っているとみなしている．

変数へ戻り値が代入された場合は，その変数に対し load 命令や getfield 命令で値を取得し計算に使うか，代入以降の続く処理を調べていく．もし load 命令や getfield 命令が行われていた場合に，取り出した値を計算に使う命令が続くならば，同期を行う必要があると判断する．メソッドの引数に戻り値を渡す場合は，invoke 命令の情報からメソッドを特定できるので，そのメソッドの仮引数に対応する変数へ戻り値が代入されているものとし，メソッドの処理を調べていく．仮引数を計算に使う処理があればその処理の直前で同期を行う必要があると判断する．return 文に戻り値を渡す場合は，プログラム全体を解析し return 先を特定し，その先の処理で戻り値を使って計算を行うか調べていく．これらの，戻り値を計算に使わない処理を調べていく途中で，調べていた戻り値が，その他の戻り値を計算に使わない処理に使われるならば，それらの処理も同様に調べていく．

変換を行う際は，非同期呼び出し部品である Threader クラスのメソッド呼び出しを追加必要な箇所に追加していく．非同期に呼び出したい invoke 命令は，代わりに Threader クラスの非同期呼び出し処理を担当するメソッドの invoke 命令に書き換わる．戻り値を計算に使わない命令が発生した場合は，それらの命令の代わりに関連付け処理をを担当するメソッドの invoke 命令が追加される．同期を行う必要があると判断された場所には，同期処理を担当するメソッドの invoke 命令が追加される．その直後に，遅延した処理である変数に戻り値を代入する put 命令や store 命令等が追加される．

7. 関 連 研 究

OpenMP[5] とは，並列処理環境に合うソフトウェアを開発する為の技術である．現在は，標準化が C 言語，C++言語，FORTRAN 言語などで進んでいる．この技術を用いて並列処理を行うソフトウェアを開発する場合，ソースコードに並列処理コードを OpenMP の指定する構文を用いて記述する．OpenMP に用意されている構文を用いることで，いくつかのスレッドモデルを実現できる．例えば，cobegin/coend モデルのスレッドモデルは for 構文を用いることで実現できる．for 構文内に記述した for 文は，各ループが並列に実行されるようスレッドが生成される．また fork/join モデルのスレッドモデルは，sections 構文を用いることで実現できる．sections 構文内に複数の section 構文を追加すると，各 section 構文内に記述した処理が並列に実行されるようスレッドが生成される．

OpenMP は開発者に容易に並列処理を記述させることを狙っている．一方，本研究の手法は，開発者が並列処理を記述しなくても並列処理が効率よく行えることを狙っていることに大きな違いがある．

遅延と分散型ソフトウェアに関する研究として，RMI における引数渡しの遅延[7] を行うといったものがある．これは，サーバオブジェクトのメソッドを呼び出しをした際に，呼び

© 2010 Information Processing Society of Japan

出し時に引数を渡さず，３種類の同期タイミングで引数渡しを行うことを提案している．１つ目は，サーバオブジェクトにプログラマが明示的に引数の取得を記述するものである．２つ目は，サーバオブジェクト内で引数の参照があった場合に，クライアントオブジェクトに取得しにいくものである．３つ目は，クライアントオブジェクトが引数を渡せるようになったら渡すというものである．

本研究でも，遅延評価を用いて，引数の値を渡す処理を遅延させている．同期タイミングは，２つ目の引数の参照があった場合に引数を取得すると同じである．その他の２つの同期タイミングを用いた場合を検討する必要がある．

本研究でマルチスレッド処理を埋め込む手法は，アスペクト指向の考え方に基づいていることは既に述べたが，類似の研究に，メソッドの非同期呼び出しをアスペクトとする研究[6]がある．この研究では，アスペクト指向の手法を用いて，非同期メソッド呼び出し機能をプログラムに織り込む Loom というツールの作成をしている．このツールの実装にはアスペクト指向プログラミング言語を用いず，JavaCC を使い独自に織り込みツールとして作成している．ツールを使用する場合，Loom Policy Language という独自の言語を用いて，ポイントカットを記述する．織り込み後のソースコードには，メソッド呼び出し部分に，非同期呼び出しを行う為のスタブを，呼び出すように変更が行われ，更にスタブに関するコードも追記される．

本研究と比較すると，Loom を用いる際には非同期呼び出しの同期を開発者自身が指定しなければばならない点があげられる．本研究では，メソッド呼び出しだけを指定することにより，同期場所は解析処理で見つけ出せる点が優れていると考える．

8. まとめと今後の課題

分散型ソフトウェアにおいて，複数の計算機上に配置されている処理を並列に実行する為には，マルチスレッド処理が必要となる．分散型ソフトウェアに適したスレッドモデルとして非同期呼び出しモデルを挙げた．TRMI を用いて開発した分散型ソフトウェアの分散処理とはメソッド呼び出しとなるからである．非同期呼び出しモデルでは，メソッドを非同期に呼び出し，その後の処理とメソッドを並列に実行させるものである．通常のプログラムを非同期呼び出しモデルに変換すると，戻り値をメソッド呼び出しの直後に使っている為，並列処理をほとんど行えない．そこで，遅延評価を用いた非同期呼び出しモデルの同期方式を提案した．この同期方式は，同期処理をできる限り遅延させ，並列処理区間を延ばすことが出来る．

また，非同期呼び出しモデルと遅延評価を用いた同期をプログラムに追加する際の具体的な方法についても述べた．その際に，いくつか問題が発生することがわかった．一つ目は，同期処理が複数のホストにまたがってしまい，同期をとるのが難しくなる点である．二つ目は，実行時にプログラムの解析を行わない場合には，同期処理がプログラムに追加できない場合がある点である．これらの問題に対し，遅延評価を用いた同期法の遅延に制限をかけることで対処した．

今後の課題がいくつかある．一つ目は，遅延評価を用いた同期法を評価し改善することである．今回，遅延評価を用いた同期法を分散型ソフトウェアに適用した際に，同期処理がサーバオブジェクト側にまで散らばる可能性があることがわかった．そのような場合でも，解析可能な場合は，制限なく処理できるようにしたいと考えている．二つ目は，プログラムにマルチスレッド処理を追加しやすくする為に，追加手法を元にした非同期呼び出し部品の追加を行う変換ツールの評価を行い，改善を行いたいと考えている．

参 考 文 献

1) G.R.Andrews and F.B.Schneider:”Concepts and notations for concurrent programming”,Computing Survey,15,1,pp.3-43(1983).
2) M. Conway: ”A multiprocessor system design”, AFIPS Conference Proceedings 24,pp.139-146(1963).
3) 打越智之, 杉山安洋:”AspectJ を用いたスレッド記述法の検討”, 電子情報通信学会 技術研究報告 SS2008-30(2008-10),pp.19-24,2008.
4) 杉山安洋，TRMI によるオブジェクトの分散化，コンピュータソフトウェア，Vol.19, No.5, pp.40-59, 2002.
5) ”OpenMP Home Page”, http://openmp.org/wp/.
6) Michael Nelson, Aspect-oriented Asynchrony in Java, University of Maryland at College Park Computer Science Honors Thesis, 2003.
7) Christopher Line, L. R. Jayaram, Patrick Eugster, Lazy Argument Passing in Java RMI, Purdue University, Aest Lafayette, USA, ACM International Conference Proceeding Series; Vol. 347, pp127-136, 2008.

ⓒ 2010 Information Processing Society of Japan