B-024

コード変換による強マイグレーション化モバイルエージェントの実現
A Code Transformation Method for Strong Migration Mobile Agent

加藤 史彬*1，田久保 雅俊*2，櫻井 康樹*1，甲斐 宗徳*1
Fumiaki Kato, Masatoshi Takubo, Yasuki Sakurai, Munenori Kai

1．はじめに

並列分散処理は、ネットワーク接続された複数のマシンに処理を分けることで、並列的に処理を行い、さらにマシンにかかる負荷を分散させることで処理の高速化や耐故障性の向上を可能にする技術である。そのようなシステム上でアプリケーションを開発するためには、変化し続けるシステム全体の状況把握や管理が必要となり、それらを考慮してプログラムを記述しなければならない。その負担を大幅に軽減するため、我々はシステムの動的な状況変化を認識し自律的に対応する自律分散処理システムを、モバイルエージェント技術を用いて試作してきた。

我々が最初に試作したJavaベースのシステムでは、エージェントのモビリティに弱マイグレーションを利用したため、移動先で移動前の処理を再開するエージェントを自由に記述することは困難であった。そこで、Java仮想マシン(JavaVM)を変更せずに強マイグレーションモバイルエージェントを実現するために、ソースコード変換を用いてエージェントの実行途中の状態を保存し、移動先で再開する方法を提案した。しかし、従来、我々が提案したソースコード変換手法では、実行時のオーバヘッドが大きく、元のコードの構造によっては変換できない例があったため、新たなソースコード変換手法を提案する。

2．自律分散処理システムとエージェントのモビリティ

2.1 自律分散処理システム

自律分散処理システムとは、分散処理で難しいとされるシステム全体の状況把握とそれに応じた自律的な制御を組み込んだシステムのことである。これにより、分散処理アプリケーションを記述したいユーザは、本来の処理内容を考えることに集中すればよく、負荷の分散や再割当てなどの細かい制御に気を使うことがなくなり、開発にかかる時間を大幅に削減することができる。

そこで我々は自律分散処理システムを実現するにあたって、モバイルエージェントを利用してきた。既存のモバイルエージェントシステムとしては、AgentSpace(佐藤一郎氏)[1]、Aglets(日本IBM社)[2]、JavaGO(東京大学米澤研究室)[3]、MOBA(首藤一幸氏)[4]等が挙げられるが、その中で我々の最初の自律分散処理システムでは、エージェントシステムの拡張がしやすい点からAgentSpaceを利用していた。

2.2 モバイルエージェントシステムのモビリティ

ネットワークに繋がれたコンピュータが分散処理に対し発揮できる性能は常に一定とは限らない。そこで我々が最初に試作した自律分散処理システムでは、負荷分散時に各コンピュータの性能値に合わせて割り当てるタスクエージェント数を調整する機能を構築している[5]。ただしこの割り当ては、分散処理開始時の各コンピュータの性能に基づくものであるため、その後の負荷の変動により自由に移動できるわけではない。例えば分散処理中のコンピュータが何らかの要因で性能が低下し、タスクの処理の完了が大幅に延期される事態が発生してもエージェントはただ待つしかない。そのような場合に自動的に性能の良いコンピュータへタスクエージェントが移動し、大幅な処理の遅延を防ぐ機能が必要となる。

この機能実現には、エージェントが任意の中断した時点での実行時データを保存し、移動先のコンピュータ上で中断時点からの処理を再開ができることが必要となる。しかしAgentSpaceは弱マイグレーションのモビリティを用いているため、移動前のスタック領域やプログラムカウンタを保存することが出来ず、移動先での実行の継続ができない。これらの情報を保存して移動するには強マイグレーションのモビリティを持つエージェントシステムが必要となる。

2.3 強マイグレーション化に必要な情報

Javaにおいてプログラムが使用するメモリ領域には、実行コード領域、スタック領域、ヒープ領域がある。Javaを用いてモバイルエージェントを実装する際に、予めJavaに備わっているシリアライズ機能を用いることで、実行コードに加えヒープ領域内の情報の保存が可能となっている。しかし、スタック領域内の情報やプログラムカウンタを実行状態として保存する機能は現状のJavaVMではサポートされていない。

プログラムカウンタを取得・復元できれば移動直前の位置からのプログラムの再開が可能となる。また、スタック領域内に保持されているローカル変数の値を取得・復元できれば、移動前の計算結果の続きで実行を再開できるようになる。そのため強マイグレーション方式のモバイルエージェントシステムを実現するためには、スタック領域内の情報とプログラムカウンタに相当する情報が必要となる。

しかしこれをJavaで実装するのは容易ではない。Javaを用いた既存の強マイグレーション方式のモバイルエージェントとしてJavaGOやMOBA、Nomads[6]、Brakes[7]などが挙げられるが、これらの研究はJavaVMの変更やランタイムシステムの拡張で強マイグレーションを実現しているため、JavaVMのバージョンアップごとに修正を施す必要があり、システム開発の負担が大きい。

そこで本研究では、JavaVMに手を加えることなく、強マイグレーション方式のモバイルエージェントシステムを実現することを目指している。しかも記述上はプログラム中に単に **migrate(移動先ホスト);** のように記述すれば、マイグレーションが可能で、実行を再開できるようにしたい。

そのための手法として、我々はすでにスタック領域内のローカル変数の取得・復元にはJPDA(Java Platform Debugger Architecture)を用いた手法を、プログラムカウンタ相当の情報の取得・復元にはソースコード変換手法を用いて実現していた。そのソースコード変換手法では、元のコードのステー

*1 成蹊大学大学院情報処理専攻
*2 横河電機株式会社

トメントを一行一行関数に区切る再構成を行い、どの関数を移動後に実行すれば良いかを制御するランタイムコードを別に生成することで、移動直前の処理から再開することが可能となっていた[8]。

しかし、この方式では移動直前のスタックフレームの構造を復元していないためユーザ関数からのマイグレートや、繰り返し回数の判明していない反復文中のマイグレートなど、変換できない制御構造があった。また、ステートメント関数中に実行処理文は1行しか記述されていないが、そこで使われている変数を宣言し、移動直前の値へと復元するために何行ものローカル変数復元代入文を挿入する必要があった。このため、元のコードの一行に相当する処理を行うのにオーバヘッドが大きくなってしまうという問題点があった。

本論文では、これらの問題点を解決するために新たなソースコード変換手法を提案する。

3．エージェントの強マイグレーション化

3.1 スタック領域の取得

スタック領域の中身を取得するためにJPDAを利用した。これはサン・マイクロシステムズ社が提供するJavaVMに標準的に実装されており、Javaアプリケーションのデバッグに使用されるもので、実行中のスレッドや実行情報へのアクセスが可能になっている[9]。本研究では、この機能を用いることで、スタック領域のデータを取得することにする。

図1は、ローカル変数を取得し、持ち出し可能にするまでのプロセスの概要を示したものである。

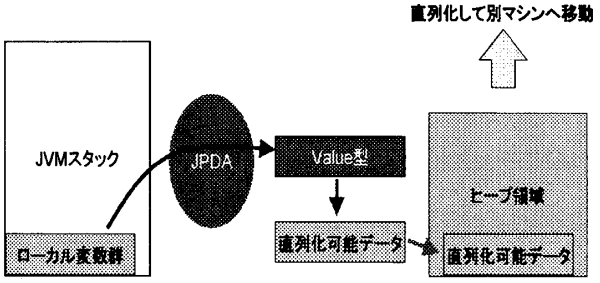

図1　ローカル変数持ち出しまでのプロセス

エージェントはスレッド上で実行されるので、JPDAを用いてそのスレッドにアクセスし、スタック内のローカル変数の値を取得する。しかし、このローカル変数の値はJPDA特有のクラス（Value型）となっているため、外部への持ち出しが許可されていない。そこでこの値をシリアライズ（直列化）可能な変数に変換し、ヒープ領域内に格納する。そうすることで、JavaVMを変更することなく、Javaに備わっているシリアライズ機能のみを用いて実行コード領域・ヒープ領域そしてスタック領域内のローカル変数を保存でき、その結果移動先のマシンに送信することができるようになる。

3.2 スタック領域の復元

スタック領域を取得し、別マシンに移動した後、移動先のマシンで実行を再開するには、マイグレーション命令の直後までの状態を復元し、そこから実行を再開する必要がある。そのためにはスタックの中身だけでなく、プログラムカウンタも必要となってくる。JPDAでは、このプログラムカウンタに類似したデータを扱っており、これを取得することは可能となっている。しかし、この取得したプログラムカウンタを用いてプログラムを途中から実行する機能は、JPDAでもサポートされていない。そこで、本研究ではソースコードの変換を行うことにより、同等の機能をサポートすることにした。

新しいソースコード変換手法では、プログラムコード中に記述された移動用関数であるmigrate関数の位置で実行時データを取得して移動し、到着後に移動直前の状態に復元するように変換する。これにより、ステートメント一行一行に対し、ローカル変数の取得、復元処理を行わないため、オーバヘッドを大幅に減らすことができる。

この変換手法を実現するために、まず、エージェントが持って移動する情報の中に、マイグレーション直後かどうかを示すフラグMigFlagを用意した。このフラグは、初期値はfalseであり、エージェントの移動が行われると移動直後を示すtrueになる。このMigFlagと、if～else構文を使い、移動直後には実行再開位置までの計算処理を全てスキップし、スタック中の変数領域の構成を再現するようにソースコードを変換する。これにより移動前のプログラムカウンタを保存して続きが実行できるのと同じ状態を実現した。

図2はその変換の様子を示している。変換前のコードでは、移動関数であるmigrate()が存在するスコープに注目する。まず変換の前準備として、そのスコープ内で定義している宣言文をスコープの前半にまとめるように変換する。これにより構成された**「宣言部・処理群・migrate関数・処理群」**という流れが変換対象の基本形となり、どのような構文でもこの基本形で表すことができる。この基本形にMigFlagを用いたif～else構文を挿入して変換後のコードを得ることになる。

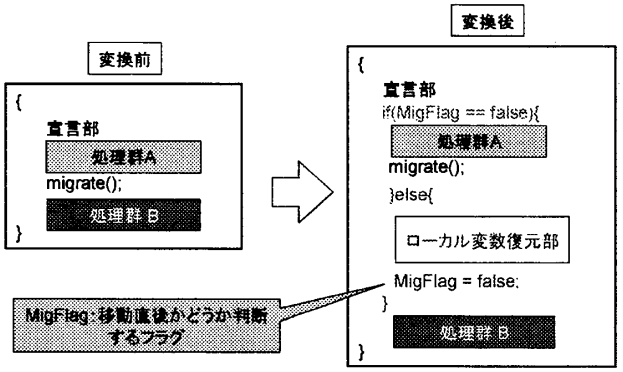

図2　変換の基本形

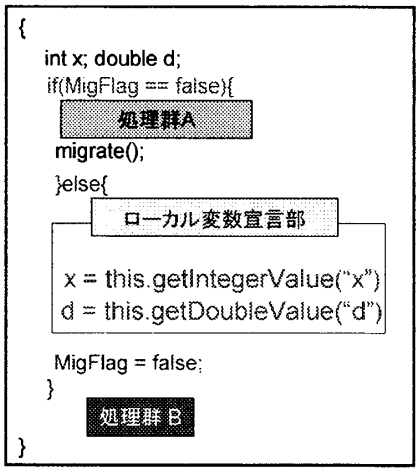

図3　ローカル変数復元代入文

ローカル変数の復元は、移動前のローカル変数の値を対応するローカル変数に代入することで実現する。この代入文はソースコード変換時に挿入する。図 3 に、実行再開位置にソースコード変換時に挿入するローカル変数復元代入文の一例を示す。移動時にヒープ領域にコピーしていたデータセットから、そのスコープ内で宣言されている変数に該当するデータを取り出して格納している。このようにしたのは以下の理由による。JPDA にはスタックフレームにアクセスし、そのスタックフレームに直接値をセットする機能がサポートされている。しかし、この機能を用いると、スタックフレームにアクセスするために対象となるスレッドを停止させておかなければならず、再現したスタック領域にすべての変数の値を書き戻すのに時間がかかる。そこでスタック領域の変数の書き戻しには、JPDA によるスタックへのアクセス機能を使わずに、getIntegerValue()などのメソッドを作成し、それを用いて移動前のローカル変数の値を再代入することでローカル変数の復元を実現した。

このようにして変換が行われたソースコードが実際にどのように動き、いかにして途中実行が実現されるかを説明する。まず、エージェントは生成時か到着時かを問わず常に同じ決まった関数から実行が開始される。次に処理に必要なローカル変数の宣言が行われる。次の MigFlag を調べる条件文では、MigFlag の初期値は false であるため処理群 A が実行される。そして migrate 関数で別マシンへと移動する。この時、migrate 関数の内部処理で MigFlag が true に切り替わる。エージェントが目的のマシンへ到着すると、プログラムの先頭から実行が開始される。次にローカル変数の宣言が行われる。そしてフラグの判定では、今度は MigFlag が true のため else 文へ入る。この else 文の中で、ローカル変数に移動前に取得しておいたローカル変数の値を代入し、MigFlag を false に戻す。これにより、ローカル変数の復元が行え、migrate 関数直後の位置から実際の計算処理を再開することができるようになる。

4．本システムの動作確認

今回このシステムの動作確認をするにあたって、バブルソートを行うエージェントを用意した。このエージェントは 10 個のデータを持ち、処理中に 2 台のマシン間を移動しながら昇順にソートを行うものである。移動のタイミングは配列の先頭から末尾まで隣接比較交換作業を一巡するごとである。図 4 には、動作確認に用いた migrate 関数記述後のソースコードの一例を、そして図 5 にはソースコード変換後のコードを示す。

表 1 は、今回評価に使用した 2 台のマシンである。

表 1　動作確認に使用したマシン一覧

マシン名	CPU名	クロック数	メモリ
マシンA	Core 2 Duo	2.2GHz	2.0GB
マシンB	PentiumD	2.8GHz	1.0GB

これら 2 台のマシン間でエージェントを移動させながら処理を行った時のマシン A でのエージェントの出力結果を図 6-1 に、マシン B での同じエージェントの出力結

```
import java.io.Serializable;
import agentsphere.*;
public class BubbleSort extends StrongMigAgent implements Serializable {
    public void create() {
    /* sample data to be sorted */
    int data[] = { 117, 75, 24, 32, 98,72, 88, 43, 60, 35};
    print(data);
    System.out.println("Sort Start!!");
    sort(data);
    //Display sorted result
    System.out.println("SortFinished!!");
    print(data);
}
void sort(int[] data){
    int i, j, temp;
    for ( i = 0; i < data.length - 1; i++ ) {
        for ( j = data.length - 1; j > i; j-- ) {
            if ( data[j - 1] > data[j] ) {
                temp = data[j - 1];
                data[j - 1] = data[j];
                data[j] = temp;
            }
        }
        print( data );    /*配列表示用の関数*/
        if (i%2==0) migrate(IPAddress2);
        else migrate(IPAdress1)
    }
  }
}
```

図 4　動作確認に用いたソースコードの一例

```
public class BubbleSort extends StrongMigAgent
                        implements Serializable{
  private String IPaddress;
  private AgentInfo Ainfo;
  public void create() {
  /* ソート対象のデータ */
  int[] data = { 117, 75, 24, 32, 98,72, 88, 43, 60, 35};

  if(Ainfo.MigFlag == 0){
     print(array);
     System.out.println("Sort Start!!");
  }else{
    // ローカル変数復元部
    array = this.getArrayIntegerValue("array",0);
  }
  sort( array );

  // ソート結果の確認
  System.out.println("SortFinished!!");
  print(array);
  }
 public void sort( int[] array ) {
   int tmp=0;
   for( int i=0;Ainfo.MigFlag!=0 || i<array.length-1; i++ ) {
      if(Ainfo.MigFlag == 0){
         for( int j=0;j<array.length-i-1; j++ ) {
            if(array[j] > array[j+1] ) {
               tmp = array[j];
               array[j] = array[j+1];
               array[j+1] = tmp;
            }
         }
          if(i%2 == 0) IPaddress = "Host address 1";
         else IPaddress = "Host address 2";
         System.out.print(i + " : ");
         print(array);
         migrate(IPaddress,1);
      }else{
        // ローカル変数復元部
         array = this.getArrayIntegerValue("array",0);
         tmp = this.getIntegerValue("tmp",0);
         i = this.getIntegerValue("i",0);
         Ainfo.MigFlag = 0;
      }
   }
}
```

図 5　ソースコード変換後のコード

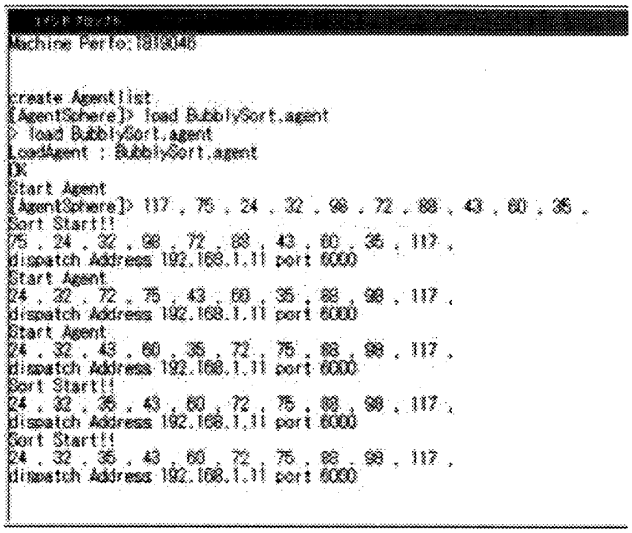

図 6-1　マシン A でのソート進行中の出力

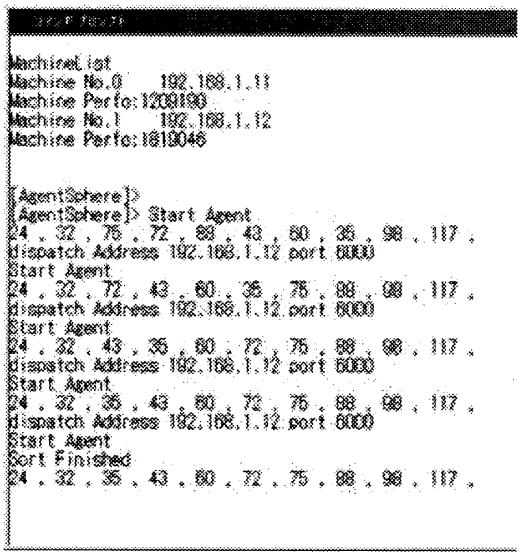

図 6-2　マシン B でのソート進行中の出力

果を図 6-2 に示す。

図を見て分かるように、移動前の配列の状態を保持しながら移動し、移動後ではその続きから処理を行っていることが分かる。この結果から、新しいソースコード変換手法により、強マイグレーションの動作が正しく行われていることが確認された。

5．終わりに

本研究により、ユーザはエージェントの記述において移動したい位置で migrate 関数を自由に使い、強マイグレーションのモビリティを活用することができるようになった。またその migrate 関数を含むコードは提案した新しいソースコード変換を行うことにより、既存の JavaVM に変更を加えることなく、スタック領域の取得・復元、移動前の実行位置からの再開処理を行うことが出来るようになった。

本研究では、エージェントコードのどんな構文構造においても migrate 関数を使用しても、正しく動作できるようにソースコード変換する仕様を定めた。今後は、このソースコード変換仕様に基づいて自動変換器を作成し、移動オーバヘッドに関する評価を行い、ソースコード変換仕様・ローカル変数取得復元機能を改善し、モバイルエージェントシステムとしての性能の向上を目指していく。

参考文献

[1] 佐藤一郎：「AgentSpace：モバイルエージェントシステム」,日本ソフトウェア科学会, Dec.1998

[2] 日本 IBM 東京基礎研究所：http://www.trl.ibm.com/aglets/ , Jul.2007 参照可

[3] 米澤明憲,関口龍郎,橋本政朋：「移動コード技術に基づくモバイルソフトウェア」http://homepage.mac.com/t.sekiguchi/javago/index-j.html, Jul.2007 参照可

[4] 首藤一幸：http://www.shudo.net/moba/, Jul.2007 参照可

[5] Ryusuke Sasaki,et.al: "IMPLEMENTATION AND EVALUATION AUTONOMIC DISTRIBUTED PROCESSING SYSTEM USING MOBILE AGENT", IEEE Pacific Rim conference 2005, No.237

[6] Suri, N. et al.: "Strong Mobility and Fine-Grained Resource Control in NOMADS", In Proceedings of ASAMA'2000, Springer, Zuerich, Germany (2000) 2-15.

[7] Truyen, E. et al.: "Portable Support for Transparent Thread Migration in Java", In Proceedings of ASAMA'2000, Springer, Zuerich, Germany (2000) 29-43

[8] 櫻井康樹,田久保雅俊,佐々木竜介,甲斐宗徳：「自律分散処理システムのための強マイグレーション化モバイルエージェント」, FIT2006(第 5 回情報科学技術フォーラム), B(ソフトウェア)分冊 pp.115-118, 2006.9

[9] Sun Microsystems:"Java Platform Debugger Architecture" http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jpda/