動的なグリッド環境における効率的でセキュアなリソース利用のためのモバイルエージェントシステム Ja-Net on Grid

沼田哲史[†1] 板生知子[†2] 小川剛史[†3]
塚本昌彦[†4] 西尾章治郎[†1]

本論文では，グリッド環境における並列計算を容易にするためのモバイルエージェントシステム Ja-Net on Grid を提案する．Ja-Net on Grid は，適応型ネットワーキングアーキテクチャJa-Net に基づいてエージェント間の関係に基づいた群制御を行うことで，グリッド環境において効率的に動作するプログラムが作成できるようにする．Ja-Net on Grid の実装は，シングルサインオンと機密性の高い情報へのアクセスを実現するために，グリッド計算のためのミドルウェアである Globus Toolkit の上で行っており，X.509 証明書をベースとしたセキュリティ基盤 GSI（Grid Security Infrastructure）を用いてこれらを可能にしている．また，遅延やマシン性能が大きく変動する動的なネットワーク環境において，GridRPC などのマスタ・ワーカ方式よりも優れたリソース利用が可能であることを実験により確認し，本システムの有効性を確認した．

Ja-Net on Grid: A Mobile Agent System for Effective and Secure Resource Use in Dynamic Grid Environment

Satoshi Numata,[†1] Tomoko Itao,[†2] Takefumi Ogawa,[†3]
Masahiko Tsukamoto[†4] and Shojiro Nishio[†1]

We propose a mobile agent system named “Ja-Net on Grid” to make it easy to achieve parallel calculation in Grid environment. Ja-Net on Grid is based on Ja-Net architecture and enables application programmers to write programs that run efficiently in Grid environment by performing collaborative activities based on relationships between agents. Ja-Net on Grid uses GSI (Grid Security Infrastructure), which is based on X.509 certificates and is included in Globus Toolkit, to enable Single Sign-on and secure access to confidential information. In this paper, we describe the design and implementation of Ja-Net on Grid. We also verify the effectiveness of Ja-Net on Grid by showing that it performs better resource distribution than the way of Master/Worker-based task distribution method.

1. はじめに

近年，センサや小型端末など実空間に設置したデバイスとネットワーク上の高度な計算機資源を活用する高度ユビキタス環境[14] の実現が期待されている．高度ユビキタス環境ではユビキタスコンピューティングとグリッドコンピューティングを統合してユーザのサポートを行うことを目的としている．高度な計算資源を用いて実空間に設置したデバイスから得られる膨大な情報の中から対象ユーザに関する情報を抽出し，それを解析してユーザをサポートするためのデータベースを作成することで，実空間における日常生活をさまざまな形でサポートできる．たとえば，街角や店内などのいたるところに設置された公共のカメラから得られる多様かつ大量の情報をグリッド上で動的に発見した計算資源を適切に利用してデータの蓄積や解析を行うことで，店内で人を追跡しながらその人の嗜好に応じた商品の場所を案内したり，ビルでは歩き回る人を監視して随時警備員にその位置を連絡するなどの応用が可能となる．

このようなアプリケーションを実現するためには，

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
†2 NTT 未来ねっと研究所
NTT Network Innovation Laboratories
†3 大阪大学サイバーメディアセンター
Cybermedia Center, Osaka University
†4 神戸大学工学部電気電子工学科
Department of Electrical and Electronics Engineering, Faculty of Engineering, Kobe University

次のような要求事項を満たす必要がある．まず，高度ユビキタス環境におけるストレージや CPU などの資源をセキュアに利用できることが重要である．すなわち，適切な権利を持ったユーザがこれらの資源を利用するために適切に認証を行い，その利用要求が適切に承認されなければならない．次に，ネットワーク上に存在する計算資源は，一般的に CPU 速度やメモリ容量，データベースの種類などが異なり，またそれらを接続しているネットワークの性能も場所によってさまざまであることから，これらの差異を吸収しつつ，適応的かつ効率的に計算できることが求められる．また人の嗜好に応じた処理などを行う場合には，その嗜好の偏りに応じてそれぞれのタスクが持つ重要性が異なる場合や，タスク間に関連性が生じる場合が出てくる．このようなタスクを処理する場合には，それぞれのタスクの重要性や関連性を考慮した処理を行わなければならない．さらに高度ユビキタス環境を実現し普及させるためには，そのシステムにおけるアプリケーション開発のためのプログラミングが簡単に行えることが重要である．その 1 つの指標として，これまで単一のプロセッサからなる計算環境を対象にしてアプリケーションプログラマが行っていた，プログラムコードに示した順に逐次処理を行う構造化プログラミングが，多種多様なリソースの差異を考慮することなく，高度ユビキタス環境を実現するためのシステムでも行えることが必要となる．

しかしこれまでの研究では，これらの要求事項をすべて満たしているフレームワークは存在しない．グリッド環境におけるさまざまなリソースの発見や利用，そしてそれらへのセキュアなアクセスのための標準インタフェースを定義している Globus Toolkit[5] を利用したミドルウェアとして，従来の RPC をグリッド環境に対応させた GridRPC（Ninf/G），Condor-G，Nimrod/G などさまざまなものが提案されているが，Condor-G と Nimrod/G の 2 つについてはいずれもネットワーク性能が変化しない静的なグリッド環境を前提に構築されており，高度ユビキタス環境における動的なネットワーク性能の変化に簡単に対応することができず，またユーザはタスクの投入時にマシン性能などのリソース要求を適切に記述しなければならない．GridRPC ではマシン間の性能差を吸収するためにタスクを細かく分割してサーバにタスクの処理を依頼し，早く処理が完了したサーバから順に新しいタスクを処理させるマスタ・ワーカ方式の適応方法が提案されている．この手法ではネットワーク遅延を考慮しない場合に良いパフォーマンスが得られるが，それを実現するためにはユーザが明示的にタスクの範囲を分割するコードを書く必要があり，それを集約するためのコードも手続き的に記述しなければならないため，ネットワーク遅延が頻繁に変化するような環境においてはマシンの性能を最大限に引き出すことが困難である．

異種リソース間の差異を吸収しつつ適応的に動作するプログラムを簡単に記述するための方法として，Java をベースにしたモバイルエージェントの利用が考えられる．モバイルエージェントはデータベースにアクセスする際の通信コストを低減する手段として利用できる[3),4),17)]．しかし，モバイルエージェントを利用する場合でも，プログラマは適切にエージェントの移動を考慮してプログラムを記述する必要があり，構造化プログラミングを行う際の妨げとなる．

そこで本論文では，Globus Toolkit と Ja-Net アーキテクチャ[15)] を利用して，グリッド上で自律的かつ適応的に動作するモバイルエージェントをベースにしたアプリケーションフレームワーク Ja-Net on Grid を提案する．Ja-Net とは環境に応じたサービスをユーザに提供するシステムである．

Ja-Net on Grid では，GSI（Grid Security Infrastructure）を用いたシングルサインオン機能を Ja-Net 上に実装することにより，ユビキタスな資源のセキュアな利用を可能とする．また，タスクの有効度とネットワーク遅延を反映した自律的なリソースを割当て方式により，計算資源の異種性やネットワーク環境に応じた適応的なリソース配分を行う．さらに，モバイルエージェントの強いマイグレーションと群制御をサポートすることにより，分散環境を意識しない構造化プログラミングを可能とする．Ja-Net on Grid では各エージェントが実行に最適な環境を求めて移動するため，詳細なリソース要求をユーザが記述する必要がなく，また動的に変化するネットワーク環境に柔軟に対応することができる．本研究では，Ja-Net on Grid のシステム構成と基本設計を明らかにするとともに，動的に変化するネットワーク環境に対応する性能を調べるため，実装したシステムを用いて GridRPC 方式との性能差を比較評価することにより，遅延やマシン性能にばらつきのある動的な高度ユビキタス環境において提案方式が有効であることを確認した．

以下，2 章では Ja-Net on Grid における Ja-Net アーキテクチャをベースとしたリソースの適応的な利用方法について述べる．3 章ではシステムの設計および実装の詳細について述べる．4 章では GridRPC 方式との比較実験の結果を示し，Ja-Net on Grid の有効性について考察する．5 章では本論文のまとめと今

後の課題について述べる．

2. Ja-Net on Grid の概要

本章では，Ja-Net アーキテクチャにおける適応的なサービス提供の仕組みと，Globus Toolkit を用いてその仕組みをグリッド上のアプリケーション構築に適用した Ja-Net on Grid の動作について述べる．

2.1 システムの構成

Ja-Net は，ネットワーク接続されたコンピュータノード上で動作する，サイバーエンティティ（CE）と呼ぶ自律的なサービスコンポーネント（モバイルエージェント）の処理系である．CE はすべてのアプリケーションの基本プログラム単位であり，活動を維持するために必要なエネルギーというパラメータを持っており，実行環境（Abstract Cyber Entity Runtime Environment: ACERE）のランタイムサービスを利用するごとにエネルギーを消費する．ユーザが良いと評価するサービス提供を行った CE にはより多くのエネルギーが供給され，ユーザが良いと評価せずにエネルギーの供給量がランタイムサービス利用のためのエネルギー消費量を下回った場合には，エネルギーがなくなることにより CE は死滅する．Ja-Net では，CE 間にリレーションシップ（関係性）という情報を生成することによって，特定の CE どうしで協調動作を行わせ，独自のサービスをユーザに提供する．また CE は環境をセンシングし，たとえばエネルギーのより強い供給源に向かうといったように，ノードからノードへの移動を自律的に行う．Ja-Net on Grid はこの仕組みを取り入れることにより，リソースの適応的かつ効率的な利用を図る．

Ja-Net on Grid においても，Ja-Net と同じくアプリケーションの基本プログラム単位はリレーションシップとエネルギーを持つ CE である．すべての CE は，AbstractCyberEntity クラスから派生したクラスのインスタンスである．CE はランタイムシステムである ACERE の上でタスクを実行し，実行中の ACERE から接続された他の ACERE に移動して実行を続けることができる（**図 1**）．各 CE は UUID による識別子を持っており，その識別子を使用して CE 間で通信を行うことが可能である．また通信を行った内容に応じて，自動または手動で CE 間にリレーションシップを生成することができる（**図 2**）．具体的には，CE が内部に持つリストに，通信相手の CE が持つ UUID と通信の内容をキーとして，キー=バリューの組を生成する．CE はリレーションシップに格納されたデータを元に通信を行う相手を決定し，また移動時には，リレーションシップによって特定の関係を持った CE と同時に移動することが可能であり，協調動作を行う複数の CE に移動先でもその協調動作を続けさせることによって，効率的な実行が可能となる．

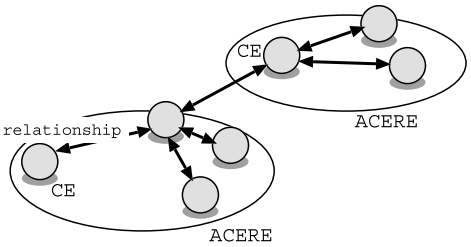

図 1 Ja-Net on Grid の概要図
Fig. 1 Overview of Ja-Net on Grid.

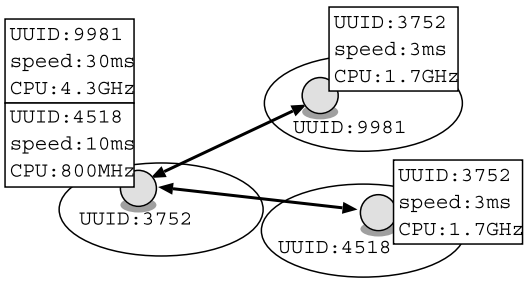

図 2 リレーションシップ
Fig. 2 Relationships.

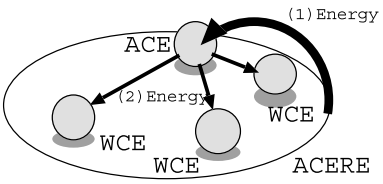

図 3 エネルギーの供給
Fig. 3 Energy feed.

Ja-Net on Grid では，プログラマは AbstractCyberEntity から派生した WorkerCE（WCE）と ApplicationCE（ACE）の 2 種類の CE を実装する．WCE は実際にタスクを処理する CE であり，ACE は WCE にタスクを分配してその処理結果を統合する．ユーザから ACERE にタスクが与えられると，ACERE は ACE を 1 つ生成してそのタスクを割り振り，この ACE は必要に応じて，他の ACERE に派遣するためのサブ ACE とタスクを実行するための WCE を生成してタスクを分配する．ACERE は一定時間ごとに ACE にエネルギーを供給し，ACE は各 WCE にエネルギーを供給し，WCE は ACERE において一定の仕事をするごとにエネルギーを消費する（**図 3**）．CE は他の

ACERE に移動する場合にもエネルギーを消費する．

各 ACERE には，他の ACERE の情報を持っている Diplomatic CE（DCE）がつねに存在する．DCE は他の ACERE へは移動せずに，一定時間ごとに他の ACERE 上の DCE と通信して，ネットワーク遅延やエネルギー供給状態，計算機の性能などの情報を更新し続ける．移動しようとする CE は DCE に移動先 ACERE の問合せを行い，DCE は ACERE の性能とネットワーク環境の情報に基づいて，移動にかかるコストと移動先での利益を考慮したうえで，最も効率的に移動でき最大限のエネルギー供給が得られると思われる ACERE を問合せ元の CE に紹介する．この方法では情報が更新されたときにエネルギー供給量が高い ACERE に移動要求が集中してしまう可能性があるため，DCE は ACERE を紹介するたびに，そのエネルギー供給可能量を一定量減少させる．

なお，本システムでは，全体のエネルギー量については制御を行っていないが，文献 18) をはじめとして，ACERE 間でのエネルギーのやりとりなどを考慮した研究も行われており，本システムに適用可能である．

2.2 適応的リソース配分

1 章で例にあげたユーザ追跡のようなアプリケーションを考えた場合，ある単一の特徴（髪の毛の色が黄色，赤いセータを着ているなど）を持ったユーザを追跡する CE が複数個集まり，それらが協調して動作すると，特定のユーザを追跡するオリジナルのサービスが提供できる．このようなアプリケーションでは，ユーザが服を着替えた場合に，赤いセータを着たユーザを追跡する CE のエネルギーが減少して死滅し，たとえば黒のシャツを着たユーザを追跡する CE が新たにサービスに加わることで適応的にユーザにサービスを提供するといったモデルが考えられる．また科学技術計算グリッドのための環境として考えた場合には，チェスや将棋や囲碁といったゲーム木探索などにおいても，CE 間のリレーションシップに基づいて同じデータを共有する CE どうしがまとまって別のノードに移動して作業を行い，タスク内容に応じたエネルギー配分を行うことによってより効率的に計算を行わせるモデルも考えられる．このように，CE がリレーションシップに基づいてまとまって行動することで，各 CE が持つ情報を共有して，文献 3) や文献 11) で目的としているデータベースへのアクセス時間の短縮も実現できる．すなわち，アプリケーションに含まれる複数のタスクの重要性に偏りがある場合や，複数のタスク間の処理に深い関連性がある場合に，それらのタスクを受け持つ CE にまとまって行動させることで，動的なネットワークに追従する適応性に加えて，タスクの性質に対する適応性をも考慮したアプリケーションの動作を実現できる．CE が他の ACERE に移動を試みるとき，その CE に対する特定のリレーションシップを持つ CE も同じ ACERE に移動を試みて，移動先で引き続き協調作業を行う．

```
class WorkingRange
    implements java.io.Serializable{
  int startPos;
  int length;
  java.io.Serializable info;
  WorkingRange(int startPos, int length,
    java.io.Serializable info) {
    this.startPos = startPos;
    this.length = length;
    this.info = info;
  }
  List divide(int num);
  List divide(int num, int[] speeds);
}
```

図 4 WorkingRange クラス
Fig. 4 WorkingRange class.

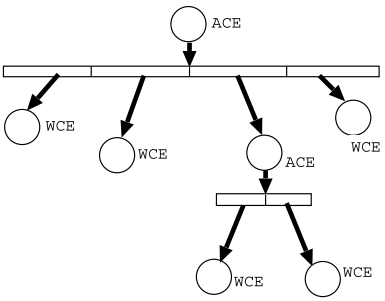

図 5 タスクの分割
Fig. 5 Task division.

2.3 タスク分割

Ja-Net on Grid で扱うタスクは，開始点とサイズを示す 2 つの整数と，特定の条件を示すために必要に応じて与えられる 1 つまたは複数のオブジェクトからなる WorkingRange クラスで表される（**図 4**）．WorkingRange クラスは特定の個数のサブタスクに分割するためのメソッド divide() を持っており，分割後のタスクを割り振るマシンの処理速度を引数に渡すことで偏りのある分割を行うこともできる．

ユーザが ACERE に WorkingRange を与えると，ACERE は最初の ACE にそのタスクをそのまま割り振り，ACE はそこからさらに他の ACERE 上で動作する ACE やタスクを実行する WCE へとタスクを分割する（**図 5**）．タスクの分割は，ACE が DCE にホスト情報を問い合わせて自動的に行う．分割するタス

```
import java.io.*;
class WorkerCE extends AbstractCyberEntity {
  abstract void work(WorkingRange range);
  abstract int evaluate();
}
```

図 6 WorkerCE クラス
Fig. 6 The WorkerCE class.

```
import java.io.*;
class ApplicationCE
    extends AbstractCyberEntity {
  abstract String getWorkerCEName();
  abstract Serializable getDefaultResult();
  abstract Serializable updateResult(
    WorkingRange range,
    Object oldResult, Object newResult);
  abstract Serializable getFinalResult(
    WorkingRange range, Object result);
  abstract int evaluate();
}
```

図 7 ApplicationCE クラス
Fig. 7 The ApplicationCE class.

ク数は ACERE ごとに設定でき，利用可能なリソースの数を超えるタスク分割数が指定されている場合には，利用可能なリソースの数がタスク分割数の上限となる．

2.4 プログラミング

プログラマは，WCE と ACE をそれぞれ継承したクラスを実装する．WCE のサブクラスで実装するメソッドは，タスクを実際に処理する work() と，呼び出された時点までに処理したタスクの処理結果を評価するための evaluate() である（**図 6**）．ACE のサブクラスで実装するメソッドは，WCE の名前を指定する getWorkerCEName()，初期状態での結果を指定する getDefaultResult()，WCE から返された処理結果を集計する updateResult()，最終的な処理結果を算出するための getFinalResult() と，呼び出された時点までに集計した処理結果を評価するための evaluate() である（**図 7**）．計算結果として使用するオブジェクトの種類は，アプリケーションプログラマが任意に設定する．

Ja-Net on Grid は，この 2 種類の CE の evaluate() メソッドを用いてエネルギー配分を決定し，その割当てに応じた動作を各エージェントが行うことで上述のリソース配分を実現して，ゲーム木探索などの，膨大なデータの中から有効なデータを抽出するような，タスクの重要性に偏りのある解析アプリケーションを効率良く実行する．

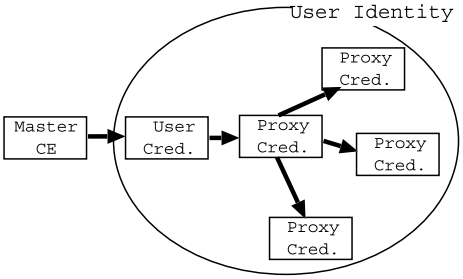

図 8 Ja-Net on Grid の認証動作
Fig. 8 Authentication of Ja-Net on Grid.

2.5 Ja-Net on Grid の API

Ja-Net on Grid において，プログラマが使用できる API は次のとおりである．

- migrate()：明示的に他の ACERE に移動する．
- findCE()：特定の条件を持つ CE を探す．
- findACERE()：特定の条件を持つ ACERE を探す．
- sendMessage()：ID を指定して CE 間通信を行う．
- createRelationship()：リレーションシップを生成する．
- updateRelationship()：リレーションシップを更新する．
- removeRelationship()：リレーションシップを削除する．

これらの API により，プログラマは従来のモバイルエージェントプログラムと同等のエージェントプログラムを書くことや，他の CE と協調動作するプログラムを書くことができる．

3. 設計と実装

本章では，2 章で述べた適応的な CE の動作を実現するフレームワークの設計と実装について述べる．

3.1 Globus の利用

Ja-Net on Grid では，認証のシステムに Globus Toolkit の GSI を用いて CE のセキュアな実行を実現している．ユーザが ACERE にタスクを投入する際にはそのユーザの証明書を用いて ACERE が認証を行い，その証明書から生成されたプロキシ証明書を持った CE を ACERE が生成する．この CE から生成されるすべての子 CE にはさらにそのプロキシ証明書から委譲されたプロキシ証明書が渡され，各エージェントはその証明書を利用して利用可能なホストと認証を行い移動していく（**図 8**）．従来のエージェントシス

テムでは，エージェント移動時の認証は移動元ホストと移動先ホストとの間で行われていたが，この方式によって移動先ホストとエージェントとの間で認証が行われることとなり，各ホストはより厳密に利用を許可するユーザを制限できるようになる．

3.2 エネルギーと評価

各 WCE へのエネルギー供給量は，ACERE の混雑状況や評価の良し悪しによって変化する．CE が移動するタイミングは基本的にエネルギーの供給量が少なくなってきた場合であるが，エネルギーの供給量を観察しているだけではこの判断ができない．そこで Ja-Net on Grid のエネルギー配分システムでは，CE のエネルギー供給時に偏差値という形で他の CE との評価の違いを伝えている．ACERE はこの評価の偏差値とともに CE のエネルギー供給メソッドを呼び出す．CE がこれまでに行った仕事の評価を表す偏差値が高いにもかかわらず，エネルギー供給量が低かった場合には，その ACERE が仕事を行うのに最適な場所ではないと判断して，CE は他の ACERE を探して移動を試みる．移動時に消費するエネルギー量は，CE 自身のデータサイズと，DCE に格納された移動元 ACERE と移動先 ACERE の間のネットワーク速度の情報によって決定する．

3.3 ACE の実装と強いマイグレーションの実現

ACE および ACERE の実装の一部を**図 9** および**図 10** に示す．CE が ACERE に移動したときに ACE の startWork() メソッドが自動的に呼ばれるようになっており，タスク処理の実行中であるかを示すフラグとタスク処理が完了しているかを示す 2 つのフラグを参照してから，ACERE の中で独自のスレッドを生成して実行を開始する．

強いマイグレーションを実現するために，Ja-Net on Grid では，文献 12) のソースコードレベルの変換による強いマイグレーションの実現手法を応用して，独自に開発した Java バイトコード変換プログラムを用いてチェックポイントを挿入することで強いマイグレーションを実現した．この Java バイトコード変換プログラムは，バイトコードの中間コードへの変換と，中間コードからバイトコードへの再変換をサポートし，goto を含めた Java バイトコードレベルでの命令の挿入を可能にする．Javassist [10] よりもバイトコード寄りの操作が可能であり，BCEL [2] よりも高度なステートメントレベルでのバイトコードの操作が可能である．

Ja-Net on Grid で ACERE が CE のバイトコードを読み込む際に，以下のようにバイトコードの変換を行う．変換プログラムはまず work() メソッドを探し，

```
class AbstractCyberEntity
    implements java.io.Serializable {
  boolean isWorking = false;
  boolean isFinished = false;
  int energy;
  int pc = 0;
  java.util.List migrationHistory;
  java.util.Map relationshipMap;
  transient ACERE workingACERE;
  transient Thread workingThread;
  void startWork() {
    if (isFinished || isWorking) {
      return;
    }
    isWorking = true;
    workingThread = new WorkingThread();
    workingThread.start();
  }
  void fillEnergy(
    int energy, int stddev);
  abstract void work()
    throws MigrationTrickException;
  class WorkingThread extends Thread {
    public void run() {
      try {
        work();
      } catch (MigrationTrickException ex) {
        // Paused for migration
      }
    }
  }
}
```

図 9 AbstractCyberEntity クラス
Fig. 9 The AbstractCyberEntity class.

```
class ACERE {
  java.util.List acereList;
  java.util.List ceList;
  void addCE(AbstractCyberEntity ce);
  void addACERE(String hostDesc);
}
```

図 10 ACERE クラス
Fig. 10 The ACERE class.

その最初と最後の部分に**図 11** に例示するようなバイトコードを挿入する．ここで MethodStackFrame クラスは**図 12** に示すようなクラスであり，メソッドの識別情報とそのメソッド内で実行が終了した場所の情報を持っている．AbstractCyberEntity が持つ stackFrameList は MethodStackFrame のインスタンスを保持するリストであり，work() メソッドが呼び出されると，実行の中断前に最後に実行していたメソッドから順に実行が再開される．各メソッドの冒頭では，

```
void work() {
  int pc;
  try {
    while (stackFrameList.size() > 1) {
      ((MethodStackFrame) stackFrameList.get(
        stackFrameList.size()-1)).doStart();
    }
    MethodStackFrame topFrame =
      (MethodStackFrame) stackFrameList.get(
        stackFrameList.size()-1);
    pc = topFrame.pc;
    localObj0 = topFrame.getStoredObject(0);
    localObj1 = topFrame.getStoredObject(1);
    // ...
    switch (pc) {
    case 0:
      goto L0;
    case 1:
      goto L1;
    // ...
    }
  L1:
    // Original Code
  } catch (MigrationTrickException ex) {
    MethodStackFrame topFrame =
      (MethodStackFrame) stackFrameList.get(
        stackFrameList.size()-1);
    topFrame.pc = pc;
    topFrame.storeLocalObject(0, localObj0);
    topFrame.storeLocalObject(1, localObj1);
    // ...
  }
}
```

図 11 強いマイグレーションの実現のためのバイトコード挿入の例
Fig. 11 An example of bytecode insertion for realizing strong migration.

```
class MethodStackFrame {
  int pc;
  String methodDesc;
  void storeLocalObject(
            int index, Object obj);
  Object getLocalObject(int index);
}
```

図 12 MethodStackFrame クラス
Fig. 12 The MethodStackFrame class.

MethodStackFrame の pc を取得し，その値に対応した場所まで goto でジャンプするコードが挿入される．

work() メソッドの中で別のメソッドを呼び出している場合には，その前後に新しい MethodStackFrame の生成と削除のためのコードが挿入される．そして各ステートメント間に CE が実行中であるか否かを示す真偽値 isWorking をチェックするコードを挿入し，実行が中断されていることが検出された時点で MigrationTrickException 例外をスローして，例外ハンドラにおいて実行状態の保存を行う．

4. 性能評価

本章では，前章で述べた実装によって得られた Ja-Net on Grid の性能について述べる．

4.1 実験環境

性能評価のための実験には，次のような構成のマシンをギガビットイーサネットワークで 14 台接続したクラスタマシンを用いた．

- CPU：Intel Pentium III 1.4 GHz
- HDD：120 GB
- メモリ：512 MB
- OS：RedHat Linux 7.3
- JDK：JDK 1.4.1_03-b02

Ja-Net on Grid において ACE が移動できる距離は 1 ホップだけとし，マスタ・ワーカ方式と同様に，最初にタスクを投入する ACERE だけが他の ACERE と接続されている環境で実験を行った．今回はマスタ・ワーカ方式との比較を行うために，タスクの重要性に偏りのないアプリケーションを用いることとし，エネルギーの消費はネットワーク間の移動時のみに発生するものとしている．したがって今回の計測では，タスク間の重要度の違いによるエネルギー供給量の増減はない．

4.2 チェックポイントのコスト

強いマイグレーションをサポートするために挿入するチェックポイントによる実行速度の低下について検証するために，モンテカルロ法によって円周率を計算するアプリケーションを用いて，1 台のマシンで計測を行い，チェックポイントを挿入した場合とそうでない場合とで処理にかかる時間の比較を行った．**図 13** に，そのプログラムを示す．その結果，10 万回以上の繰返しを行う場合には，チェックポイントを挿入した場合でもそうでない場合でも同じ程度の時間で計算が完了することを確認した（**図 14**）．10 万回の計算を行うのに要した時間は，双方ともおよそ 200 msec である．Ja-Net on Grid が対象とする人物追跡などのアプリケーションでは，画像解析などにさらに大きな繰返し回数が要求されるため，実用上問題のない性能であるといえる．

4.3 マスタ・ワーカ方式との比較

Ja-Net on Grid のタスク分配方式の有効性を確かめるために，マスタ・ワーカ方式との性能比較を行った．Ja-Net on Grid 方式のプログラムは図 13 に示すものである．マスタ・ワーカ方式のプログラムには

```
class PiWCE extends WorkerCE {
  int countPi(int repeatTime) {
    int count = 0;
    for (int i = 0; i < repeatTime; i++) {
      double x = Math.random();
      double y = Math.random();
      if (x * x + y * y < 1.0) {
        count++;
      }
    }
    return count;
  }
  void work(WorkingRange range) {
    int count = countPi(range.length);
    finish(new Integer(count));
  }
}
class PiACE extends ApplicationCE {
  String getWorkerCEName() {
    return "PiWCE";
  }
  java.io.Serializable getDefaultResult() {
    return new Integer(0);
  }
  java.io.Serializable updateResult(
    WorkingRange range, Object oldResult,
    Object newResult) {
    return new Integer(
      (Integer) oldResult.intValue()+
      (Integer) newResult.intValue());
  }
  java.io.Serializable getFinalResult(
    WorkingRange range, Object result) {
    int count = (Integer) result.intValue();
    return new Double(
      4.0 * (double) count / range.length);
  }
}
```

図 13 円周率の計算を行う WCE と ACE
Fig. 13 WCE and ACE for Pi Calculation.

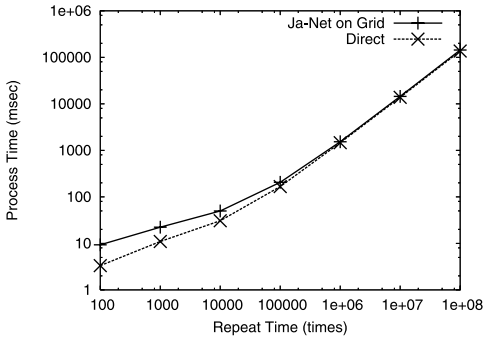

図 14 チェックポイントのコスト
Fig. 14 Cost of checkpointing.

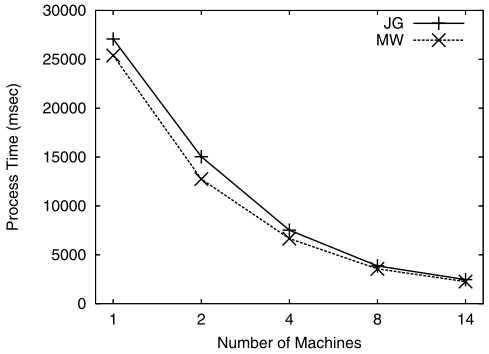

図 15 遅延がない場合の処理時間
Fig. 15 Process time without delay.

GridRPC[8] と同様にモンテカルロ法による試行回数と結果の値を表す int 値をソケット通信によってやりとりするプログラムを用意した．マスタ・ワーカ方式の実装には Java を用いており，繰返し計算を行うためのメソッドには図 13 の PiWorkerCE において使われている countPi() メソッドを使用している．いずれも同じ Java 1.4.1 の VM を用いて実行しており，通信方式の差異を除けば両者の実装に相違点はない．

Ja-Net on Grid 方式の場合もマスタ・ワーカ方式の場合も，最初にタスクを投入する 1 台のサーバマシンに複数台のクライアントマシンを 1 対 1 で接続している．Ja-Net on Grid におけるタスク分割数の上限については，今回はリソースの数と同じ 14 個とした．

図 15 に，遅延がない環境でモンテカルロ法による円周率の計算を 10^8 回の繰返しで行い，マスタ・ワーカ方式においては 100 個のタスクに分割した場合の計測結果を示す．**図 16**，**図 17**，**図 18** は，マスタ・ワーカ方式と Ja-Net on Grid の双方について高速化率，並列化効率，オーバヘッドを算出したグラフである．図中において MW はマスタ・ワーカ方式を表し，JG は Ja-Net on Grid 方式を表し，Optimal は 1 台のマシンでタスクを処理した場合の処理時間をマシンの台数で割った理想的な値を表している．1 台のマシンでの処理にかかった時間を T(1)，p 台のマシンでの処理にかかった時間を T(p) として，高速化率 S(p) = T(p)/T(1)，並列化効率 E(p) = S(p)/p，オーバヘッド O(p) = T(p) – T(1)/p として計算している．この算出方法は，文献 16) による．

1，2，4，8，14 台で計測したいずれの場合においてもマスタ・ワーカ方式の方が速い処理時間となっており，2 台のマシンを使用したときで 2,290 msec，14 台のときで 205.5 msec の差となっている．

図 19 および**図 20** に，マシン間の性能差とネット

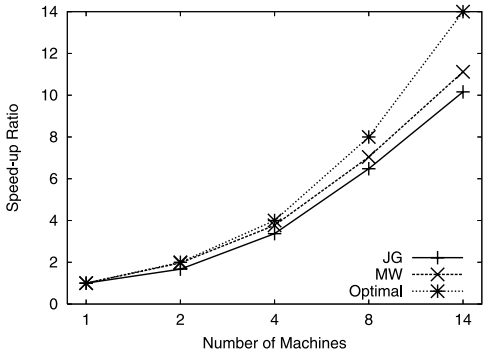

図 16 遅延がない場合の高速化率
Fig. 16 Speed up without delay.

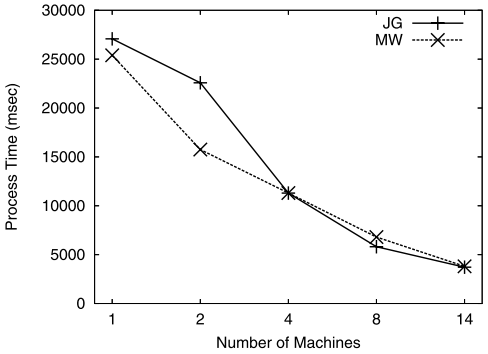

図 19 遅延がある場合の処理時間
Fig. 19 Process time with delay.

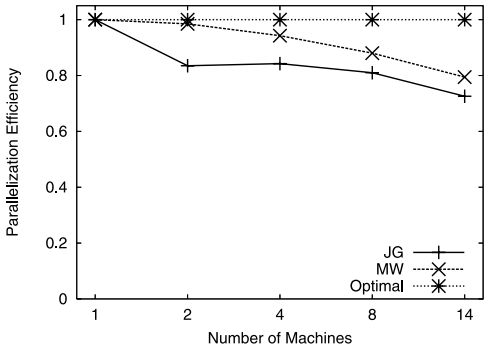

図 17 遅延がない場合の並列化効率
Fig. 17 Parallelization efficiency without delay.

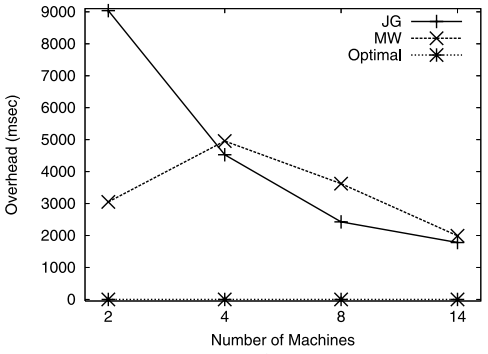

図 20 遅延がある場合のオーバヘッド
Fig. 20 Overheads with delay.

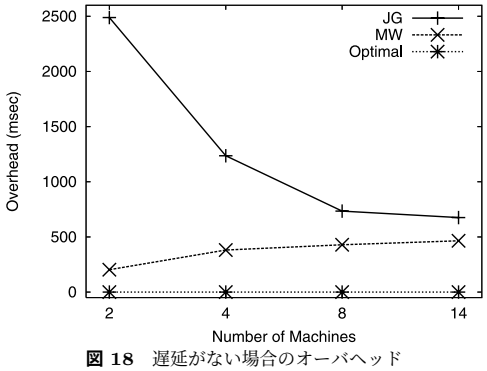

図 18 遅延がない場合のオーバヘッド
Fig. 18 Overheads without delay.

ワーク遅延を設定してマスタ・ワーカ方式と比較した場合の処理時間とオーバヘッドを示す．今回は，2，4，8，14 台で計測を行った場合について，半数のマシンで計算にかかったのと同じ時間スリープさせることでマシン性能を 2 分の 1 にし，残りのマシンではソケット通信のメッセージ受信時と送信時に 10 msec スリープさせてレイテンシを上げることでネットワーク遅延を設定した．

4 台以上のマシンを使用したときには Ja-Net on Grid 方式の方が速い処理時間となり，4 台のマシンの使用時で 6.5 msec，8 台のマシンでは 985.5 msec，14 台のマシンでは 90.25 msec の差となった．

5. 考　　察

Ja-Net on Grid 方式では，タスクの範囲だけでなく CE のデータも強いマイグレーションによって移動していることから，マスタ・ワーカ方式よりもオーバヘッドが大きくなることが予想され，実際に遅延がない環境における計測では Ja-Net on Grid 方式の速度がつねに下回った．しかし図 18 においては，マシン台数が増えるごとにオーバヘッドは少なくなっており，14 台のマシンを使用した場合のオーバヘッドはマスタ・ワーカ方式と比較して 200 msec 程度上回っている．

それに対して，遅延がある環境で計測した場合には，4 台以上のマシンを使用した場合に Ja-Net on Grid 方式の方が処理時間が短くなることが確認できた．8 台のマシンを使用した場合には 985.5 msec 速くなり，14 台のマシンを使用した場合には 90.25 msec 速くなっ

ている．14 台のマシンを使用した場合の性能差が 8 台の場合よりも小さくなっているのは，マシンの数が多くなり，1 台あたりに割り当てられるタスク数が少なくなったので，遅延の影響が少なくなったためと考えられる．多数のマシンを利用する場合には，マスタ・ワーカ方式ではネットワーク遅延が大きいノードのアイドル時間が増加する．今回の実験ではマスタ・ワーカ方式と比較して Ja-Net on Grid が格段に速くなっているわけではないが，より細かなタスク配分を行うために分割数を大きくし，使用するマシンの台数が増加して，その性能のばらつきが大きくなった場合には，さらに Ja-Net on Grid 方式が有効に機能するものと考えられる．

Ja-Net on Grid と同様に Globus Toolkit を利用している Java のスレッド・マイグレーションをサポートするシステムに，MOBA[6] がある．MOBA では強いマイグレーションをサポートしており，この上でモバイルエージェントを作成することが可能であるが，実際のモバイルエージェントの枠組みやモバイルエージェントの振舞いを制御するための仕組みは用意されていない．また MOBA では強いマイグレーションのサポートに JDK 1.1 において可能な JIT によるスレッド状態の取得を行っているが，これは最新の Globus Toolkit 3.0.2 が必要とする JDK 1.3.1 以上の環境と共存できない．そのため，Ja-Net on Grid ではバイトコードの変換によって強いマイグレーションを実現した．

6. おわりに

本論文では，動的に変化するグリッド環境において適応的にリソースを配分するためのモバイルエージェントシステム，Ja-Net on Grid を提案し，その設計と実装の詳細について述べた．マスタ・ワーカ方式のプログラムとの性能比較を行い，ネットワーク環境やマシンの性能差が大きい場合により良い性能を示すことを確認した．

今後は，カメラによる人物追跡サービスや，タスク間の重要性が大きく異なる将棋などの実アプリケーションに本手法を適用して本リソース配分方式の有効性を検証していく予定である．

謝辞 本論文を執筆するにあたってご協力をいただいた西尾研究室の諸氏に感謝する．なお，本研究の一部は，文部科学省 21 世紀 COE プログラム「ネットワーク共生環境を築く情報技術の創出」ならびに，文部科学省特定研究領域（C）「Grid 技術を適応した新しい研究手法とデータ管理技術の研究」（プロジェクト番号：13224059）によっている．ここに記して謝意を表す．

参考文献

1) Andrew, S. and Luc, M.: Engineering an Agent-Based Peer-to-Peer Resource Discovery System, *Agents and P2P Computing 2002* (2002).
2) Apache Software Foundation. http://jakarta.apache.org/bcel/
3) David, K., George, C., Robert, S.G., Guofei, J. and Ronald, A.P.: Performance Analysis of Mobile Agents for Filtering Data Streams on Wireless Networks, *ACM Mobile Networks and Applications* (2002).
4) Danny, B.L. and Mitsuru, O.: Seven Good Reasons for Mobile Agents, *Comm. ACM*, Vol.42, No.3, pp.88–91 (1999).
5) The Globus Project. http://www.globus.org/
6) Gregor, V.L., Kazuyuki, S. and Yoichi, M.: Grid-based Asynchronous Migration of Execution Context in Java Virtual Machines, *Proc. European Conference on Parallel Computing* (*Euro-Par 2000*), pp.22–33 (2000).
7) Junwei, C., Stephen, A.J., Subhash, S., Darren, J.K. and Graham, R.N.: ARMS — An agent-based resource management system for grid computing, *Scientific Programming*, *Special issue on Grid computing* (2002).
8) Keith, S., Hidemoto, N., Satoshi, M., Jack, D., Craig, L. and Henri, C.: GridRPC: A Remote Procedure Call API for Grid Computing, *Grid 2002*, pp.274–278 (2002).
9) Robert, S.G., David, K., Ronald, A.P., Jr., Joyce, B., Daria, C., Peter, G., Martin, H., Jeffrey, B., Maggie, B., Renia, J. and Niranjan, S.: Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task (2001).
10) Shigeru, C. and Muga, N.: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators, *Proc. 2nd Int'l Conf. on Generative Programming and Component Engineering* (*GPCE '03*), LNCS 2830, pp.364–376, Springer-Verlag (2003).
11) Yu, J. and Ali, R.H.: Mobile Agents in Mobile Data Access Systems, *CoopIS-37* (2003).
12) 阿部洋丈，一杉裕志，加藤和彦：ソースコード変換技術を用いた Java 言語におけるスレッドのモビリティの実現法，情報処理学会論文誌：プログラミング，Vol.41, No.SIG 2 (PRO 6) (2000).
13) 阿部洋丈，加藤和彦：強いマイグレーションと資源消費制御をサポートしたモバイルエージェン

トシステムの実現，*SPA2000* (2001).
14) 板生知子，塚本昌彦，山本 淳，田中 聡：高度ユビキタス環境のための Ja-Net アーキテクチャ，電子情報通信学会総合大会論文集 (2004).
15) 須田達也，松尾真人，板生知子，中村哲也，今田美幸，大塚卓哉，田中 聡：アプリケーション創発のための適応型ネットワーキングアーキテクチャJa-Net，情報処理，Vol.43, No.6 (2002).
16) 須田礼仁：並列プログラミングのキーポイント，東大情報理工 ARA プログラム：平成 15 年度第 9 回フォーラム資料 (2004).
17) 佐藤一郎：モーバイルエージェントシステムの比較と研究動向，日本ソフトウェア科学会主催チュートリアル「モーバイルエージェント」第 2 章，pp.79–109, 日本ソフトウェア科学会 (1999).
18) 中村哲也，松尾真人，須田達也：サービス創発のための適応型ネットワーキングアーキテクチャ Ja-Net におけるシステム制御に関する一考察，情報処理学会第 64 回全国大会，1J-2 (2002).

(平成 16 年 3 月 20 日受付)
(平成 16 年 7 月 31 日採録)

(担当編集委員 有次 正義)

沼田 哲史（学生会員）

1978 年生．2002 年大阪電気通信大学大学院工学研究科情報工学専攻博士前期課程修了．同年大阪大学大学院情報科学研究科マルチメディア工学専攻博士後期課程入学．モバイルエージェントを主としたプログラミング環境の研究開発に興味を持つ．

板生 知子（正会員）

1994 年東京工業大学電気電子工学科卒業．1996 年 Stanford 大学大学院 Computer Science 学科修士課程修了．同年日本電信電話（株）入社．同社未来ねっと研究所研究員．ネットワークサービスシステムの研究開発に従事．

小川 剛史（正会員）

1997 年大阪大学工学部情報システム工学科卒業．1999 年同大学院工学研究科博士前期課程修了．2000 年同研究科博士後期課程中退後，大阪大学サイバーメディアセンター情報メディア教育研究部門助手となり，現在に至る．博士（情報科学）．グループウェア，バーチャルリアリティ，オーグメンティッドリアリティ，モーバイルコンピューティングに興味を持つ．電子情報通信学会，日本バーチャルリアリティ学会会員．

塚本 昌彦（正会員）

1987 年京都大学工学部数理工学科卒業．1989 年同大学院工学研究科修士課程修了．同年シャープ（株）入社．1995 年大阪大学大学院工学研究科情報システム工学専攻講師，1996 年同専攻助教授，2002 年同大学院情報科学研究科マルチメディア工学専攻助教授となる．2004 年神戸大学工学部電気電子工学科教授となり，現在に至る．工学博士．ウェラブルコンピューティングおよびユビキタスコンピューティングに興味を持つ．ACM，IEEE 等 8 学会の会員．

西尾章治郎（フェロー）

1975 年京都大学工学部数理工学科卒業．1980 年同大学院工学研究科博士後期課程修了．工学博士．京都大学工学部助手，大阪大学基礎工学部および情報処理教育センター助教授，大阪大学大学院工学研究科情報システム工学専攻教授を経て，2002 年より同大学院情報科学研究科マルチメディア工学専攻教授となり，現在に至る．2000 年より大阪大学サイバーメディアセンター長，2003 年より大阪大学大学院情報科学研究科長を併任．この間，カナダ・ウォータールー大学，ビクトリア大学客員．データベース，マルチメディアシステムの研究に従事．現在，Data & Knowledge Engineering，Data Mining and Knowledge Discovery 等の論文誌編集委員．ACM，IEEE 等 9 学会の会員．