Javaによる階層型グリッド環境Jojoの設計と実装

中田秀基[†,††] 松岡聡[††,†††] 関口智嗣[†]

本稿では，グリッド環境でのJavaプログラミングを支援する実行環境Jojoについて述べる．JojoはJavaを用いて実装された，階層構造を持つ環境に適した分散実行環境で，階層構造に適した柔軟な多階層実行機構，Globusやsshを用いた安全な起動と通信，直感的で並列実行に適したメッセージパッシングAPI，プログラムコードの自動アップロードといった特徴を持つ．Jojoを用いれば，グリッド上で動作する並列分散システムが非常に容易に構築できる．本稿ではJojoの設計と実装の詳細，プログラミングAPI，設定ファイル，簡単なプログラム例を示す．さらにマスタ・ワーカプログラムを用いた性能評価を行い，多階層構造の有効性を確認する．

A Java-based Programming Environment for the Grid: Jojo

Hidemoto Nakada,[†,††] Satoshi Matsuoka[††,†††]
and Satoshi Sekiguchi[†]

This paper introduces a java-based programming environment for the Grid; Jojo. Jojo is a distributed programming environment implemented in Java, which is suitable for hierarchal grid environment. Jojo provides several features, including remote invocation using Globus GRAM, intuitive message passing API suitable for parallel execution and automatic user program staging. Using Jojo, users can construct parallel distributed application on the Grid with ease. In this paper, we show design and implementation of Jojo, its programming API, configuration file syntax and a working program example. We also show preliminary performance evaluation results that prove effectiveness of multi-hierarchal execution.

1. はじめに

複数の管理主体に属する計算資源を集合的に活用して，大規模な計算を行うグリッドと呼ばれるシステムが普及しつつある．グリッドシステムにおけるプログラミング環境としては，比較的低レベルなツールキットを提供するGlobus[1]や，GridRPCと呼ばれるミドルウェアであるNinf-G[2]やNetSolve[3]，グリッド上のMPIであるMPICH-G2[4]などが提案されている．

これらのシステムは，すべてのノードがグローバルアドレスを持ち，互いに通信可能なシンプルなグリッド環境では有効であるが，昨今一般的になりつつある複数のクラスタから構成されるグリッド環境では十分に性能を発揮することはできない．また，システムのセットアップ，プログラムコードのアップロード，結果のダウンロードなどをユーザが明示的に行わなければならず，ユーザの負担が大きい．

本稿では，グリッド環境でのJavaプログラミングを支援する実行環境Jojo[5]の設計と実装について述べる．Jojo はJavaを用いて実装された，階層構造を持つ環境に適した分散実行環境で，階層構造に適した柔軟な多階層実行機構，Globusやsshをいたセキュアな起動と通信，直感的で並列実行に適したメッセージパッシングAPI，プログラムコードの自動アップロードといった特徴を持つ．Jojoを用いることで，グリッド上で動作する並列分散システムが容易に構築可能となる．本稿ではさらに，簡単なプログラム例を示すとともに性能評価を行い，多階層実行機構の有効性を確認する．

本稿の構成は次のとおりである．2章では，Jojoの対象とするグリッド環境について述べる．3章でJojoの設計について述べ，4章で実装について述べる．5章でJojoを用いた簡単なプログラムを示す．6章で簡単な評価とその結果を示し，議論を行う．7章で関連する研究について述べ，8章で結論と今後の課題を述べる．

† 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology (AIST)
†† 東京工業大学
Tokyo Institute of Technology
††† 国立情報学研究所
National Institute of Information

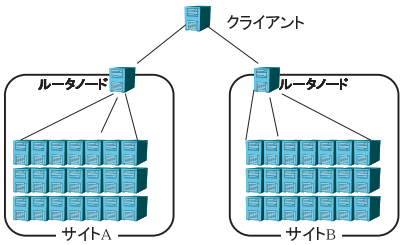

図 1 複数クラスタから構成されるグリッド
Fig. 1 A Grid consists of multiple clusters.

2. 階層的グリッド環境

今後のグリッドにおける計算機資源としてはクラスタが有望である．特に比較的小規模なクラスタを複数個結合する形が，将来のグリッドとして一般的になると考えられる．このようなクラスタの各ノードは，セキュリティやアドレス空間枯渇の問題から，ローカルの IP アドレスを持ち，ルータノードで提供される NAT を用いて外部と通信する場合が多くなると考えられる（**図 1**）．

このような環境では，Globus をベースとするシステムは十分に性能を発揮できない．Globus の GRAM jobmanager を使用して外部からクラスタ上の各ノードにプロセスを起動することは可能だが，そのプロセスから他のクラスタ内部のノードに対して直接通信を行うことはできない．このため MPICH-G2 などを用いても，複数のクラスタを使用した計算を行うことはむずかしい．この問題を解決するために，プロキシサーバを使用する研究も行われている[6]．しかしこのアプローチでは，Globus 本体にパッチを当てる必要があるため，Globus の急速なバージョンアップに追随することが難しく，現時点では公開されおらず，実用化されているとはいいがたい．

Ninf-G を用いてマスタ・ワーカ的な計算を行う場合には，このような構成でもクラスタノード上にワーカを置いて実行することができる．しかし，複数のクラスタを使用する環境ではノードの総数が数百台に達することも考えられ，これらのノードをフラットに管理することは，ファイルディスクリプタ数の制約や，ルートとなるクライアントノードにかかる負荷を考えると，現実的ではない．

これらを考慮すると，今後の大規模なグリッドは必然的に階層構造をとらざるをえないと思われる．すなわち，クライアントを 1 層目とし，クラスタのルータノードを 2 層目，クラスタのノードを 3 層目とする 3 階層の階層構造を持つグリッドである．3 層目のクラスタノードに対しては，外部から直接アクセスすることはできない．また，グリッドの規模が大きくなる場合にはさらに階層が増えることも考えられる．

このような階層構造を前提として考えると，階層構造を積極的に利用してシステムを構築することが可能になる．たとえば単純なマスタ・ワーカでは，マスタをクライアントに置くのではなく，中間層ノードで動かすようにすれば，マスタ・ワーカ間の通信レイテンシが低下し，結果として性能の向上が期待できる．より複雑な構造を持つプログラムでも，問題の持つ階層構造をネットワークの階層構造にうまくマップすることができれば，性能上のボトルネックとなる高頻度な通信を高速なローカルネットワーク内に閉じこみ，低速な通信路を流れる通信量を減少させ，性能を向上させることができる．

3. Jojo の設計

Jojo は中小規模のクラスタが分散して存在する環境を対象とし，以下の点を考慮して設計されている．

- グリッドの階層構造を反映した階層的なシステム構造
- スレッドを前提とした柔軟で簡潔なプログラミングモデル
- 動的なシステム構成を可能にするとともに，インストールの手間を最小限にする起動手法

3.1 システム構造

Jojo は，前節で述べた階層的なグリッドをターゲットとし，階層構造を意識したシステム構造をとる．Jojo はクライアントを頂点とする任意段数の階層構造を持つシステムを構成し，それぞれのノードで任意のプログラムを実行することができる．個々のノードは自分と同レベルのノード群だけでなく，上位レベルのノード，下位レベルのノード群とも通信することができる．

3.2 プログラミングモデル

Java におけるプロセス間通信ライブラリとしては，Sun による RMI[7]に代表されるリモートオブジェクトを用いるものと，C 言語でよく用いられる MPI に代表されるメッセージパッシングをベースとしたシステムがある（7 章参照）．

Jojo は，この両者の中間ともいうべきプログラミングモデルを提供する．Jojo では各ノードに 1 つのオブジェクトを配置し，そのオブジェクト間でのメッセージパッシング機構を提供する．メッセージパッシング機構としては，オブジェクト単位の送信と受信を行う．送信は明示的に行うが，受信はハンドラを定義することで行う．すなわち，recv に相当することを

ユーザが書く必要はない．ただし，送信に対する返答を受け取るというパターンは頻繁に使用されるので，これを支援するために，送信に対する戻り値を返す機構を設けてある．戻り値を呼び出し側プログラムに返す機構としては，ブロックする同期呼び出しに加えて，Future オブジェクトを使用する機構とコールバックオブジェクトを登録する機構を用意した．また，マルチスレッドを前提として，メッセージの受信ハンドラは別スレッドで実行される．このため，特にプログラマが意識しなくても，ハンドラが受信したメッセージを処理している間にも，後続するメッセージの受信は別スレッドで行われるため，メッセージの受信と処理を重複させるコーディングがごく自然に行うことができる．API の詳細については次節で述べる．

各ノード上で起動されるオブジェクトクラスは設定ファイルで自由に指定することができるが，典型的には 1 つのレイヤは同一のクラスを実行するものとする．

3.3 起動方法

Jojo は大域での分散実行を指向している．このためすべてのノードが NFS でファイルシステムを共有していることを期待することはできない．しかし，ユーザがコードをすべてのノードにアップロードするのは煩雑である．Jojo ではすべてのユーザプログラムが自動的にクライアントからダウンロードして実行される．

さらに Jojo 自体も自動的にダウンロードされて実行される．これによって実行する Jojo のバージョンがノードによって異なる，というような事態を未然に防ぐことができる．

4. Jojo の実装

ここでは，Jojo の実装について詳細に述べる．

4.1 リモート環境でのプログラム起動

リモート環境でのインストールの手間を最小限にとどめるため，ユーザプログラムだけでなく Jojo 自身の転送も実行時に動的に行われる．動的ロードにはブートストラップサーバ rjava[8] を用いた．

4.1.1 ブートストラップサーバ rjava

Jojo のシステム起動は次のように行われる（**図 2**）．

(1) クラスローダを含む最小限の Java プログラムである rjava のブートストラップサーバを jar ファイルの形でリモートノードに転送する．

(2) ブートストラップサーバを起動して rjava クライアントとの間にコネクションを張る．rjava クライアントはクラスファイルのローダとして機能する．

(3) リモートノード上のブートストラップサーバが Jojo のシステムクラスを起動する．システムクラス群は自動的に rjava クライアントを経由して，ローカルファイルシステムから読み出される．

(4) 同様にローカルノード上でも Jojo システムをロードして起動する．

(5) Jojo 上のユーザプログラムをロードして起動する．この際リモートホストで稼動するユーザプログラムは，ローカルファイルシステムからロードされる．

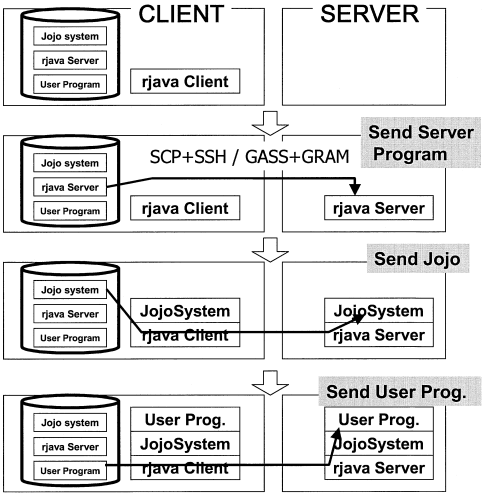

図 2 rjava による起動
Fig. 2 Bootstraping by rjava.

特筆するべき点としては，JoJo やユーザプログラム自身が必要とするライブラリも自動的にロードされることがあげられる．rjava は自身が起動された際のクラスパスおよびライブラリパスを参照し，その中に含まれている．class ファイルおよび.jar ファイルからクラスファイルをアップロードする機能を持っている．たとえば Jojo 自身は xerces という XML パーザを使用しているが，これはクライアント側のみにインストールされていれば十分で，サーバ側にインストールする必要はない．

4.1.2 rjava での起動プロトコル

起動方法としては，ssh や rsh を用いる方法と Globus の GRAM と GASS を用いて Globus I/O を使用して通信する方法が用意されている．

ssh や rsh を用いる際には，scp や rcp でブートストラップとなる jar ファイルを転送し，ssh や rsh でこの jar ファイルを指定して Java VM を起動する．ローカルプログラムとブートストラッププログラム間の通信は，ssh，rsh が提供する標準入出力用のストリーム

をマルチプレクスして使用している．

GRAM/GASS を用いる際には GRAM/GASS の提供するファイルステージ機能を用いる．ここで問題になるのは，現在最も広く使用されている Globus2.0 の GRAM には実行ファイルそのものと標準入力をステージする機能はあるが，引数として与えるファイルをステージする機能はないことである☆．このためブートストラップコードの jar をそのまま転送することは難しい．そこで，rjava ではシェルスクリプトコードを実行ファイルとして転送し，その標準入力への入力として jar ファイルを与えることで jar ファイルの転送を実現した．このスクリプトは，クライアント側で起動している GASS サーバから jar ファイルを標準入力から受け取って，テンポラリディレクトリに書き出し，次にその jar ファイルを用いて Java VM を起動する．

ローカルプログラムとブートストラッププログラム間の通信にはブートストラップサーバからの Globus-I/O によるコールバック接続を用い，同様にこのストーリムをマルチプレクスして使用している．

4.1.3 多段階の起動

多段接続となる場合には，基本的に上述の手続きを再帰的に繰り返す．ただし，クラスファイルをロードするファイルシステムがつねにクライアントホストとなるよう，クラスファイルの要求はつねにクライアントにまで委譲される．

4.2 API

Jojo 上でのプログラミングは，Jojo の提供する抽象クラス `Code` を継承して具体的なクラスを実装をすることで行う．`Code` では `Node`，`Message` などのサポートクラスを用いてプログラミングを行う．

4.2.1 Code

`Code` クラスの定義を**図 3** に示す．siblings，descendants，parent はそれぞれ同レベル，下位レベル，上位レベルのノードを指す．init メソッドは初期化を行う．引数となる Map には Jojo 起動時に引数として渡す Properties 形式ファイルの内容が渡される．init メソッドの終了以前に，start メソッドや handle メソッドが呼び出されることはない．start メソッドは，実際の処理を行うメソッドである．

メッセージを受信すると handle メソッドが起動される．このメソッドを実行するスレッドは，オブジェクト受信時に新たに起動される．つまり複数のメッセージを連続して受信した場合には，複数のメッセージ処理スレッドが同時に handle メソッドを実行することになる．したがって，ハンドラの中で長大な処理を行っても，他のオブジェクトの受信に影響はない．その反面，共有される資源にアクセスする際には適切な排他処理が必要になる．

プログラマがこの排他処理を煩雑であると感じるならば，handle メソッドに `asynchronous` 修正子を付加すれば，すべての handle メソッドが排他的に起動するようになる．

☆ Globus 2.2 以降では引数ファイルを転送する機能が追加されているが，以前のバージョンとのコンパチビリティを考慮し，使用していない．

```
abstract class Code{
  Node [] siblings;    /** 兄弟ノード */
  Node [] descendants; /** 子ノード */
  Node    parent;      /** 親ノード */
  int     rank;  /** 兄弟の中での順位 */
  /** 初期化 */
  public void init(Map arg);
  /** 本体の処理 */
  public void start();
  /** 送信されてきたオブジェクトの処理 */
  public Object handle(Message mes);
}
```

図 3 Code クラス
Fig. 3 Code class.

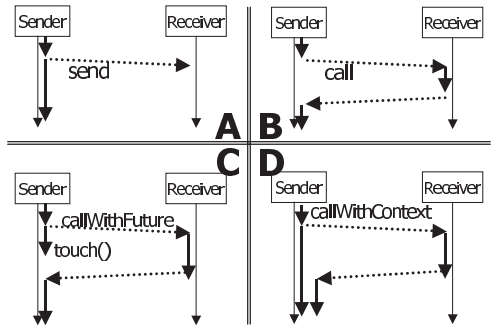

図 4 Jojo の通信モード
Fig. 4 Communication modes of Jojo.

4.2.2 Node

Code クラスでは Node クラスのオブジェクトに対してメソッドを発行することで通信を行う．Node クラスには以下の 4 つのメソッドが提供されており，柔軟な通信を行うことができる．

`void send(Message msg)`
: 単純にメッセージオブジェクトを送信する．送信後はすぐにリターンする（**図 4**：A）．

`Object call(Message msg)`
: メッセージオブジェクトを送信し，返信オブジェクトの到着を待つ（図 4：B）．

`Future callFuture(Message msg)`
: Future 機構を用いた非同期通信機構を実現する．

```
public interface Context{
  public void run(Object o);
}
```

図 5 Context インタフェース
Fig. 5 Context interface.

このメソッドはメッセージオブジェクトを送信し，ただちに返信オブジェクトの Future オブジェクトを返す．Future オブジェクトの touch() メソッドを呼ぶとそこで同期が行われる．すなわち，返信オブジェクトがすでに得られていれば即座にそれを返し，まだ得られていなければ返信オブジェクトの受信までブロックして待ち，受信後に返信オブジェクトを返す（図 4：C）．

`void callWithContext(Message msg, Context context)`

返信オブジェクト受信時に実行するべきコンテクストを指定する非同期通信機構を実現する．第 2 引数に受信時に実行する Context インタフェースを持つオブジェクトを指定する．このメソッドはメッセージ送信後すぐにリターンする．返信オブジェクトが到着すると，それを引数として Context インタフェースの run メソッドが呼び出される．run メソッドの実行は callWithContext を行ったスレッドとはまったく別のスレッドで実行される（図 4：D）．Context インタフェースは**図 5** のように定義されている．

4.2.3 Message

Message は送信対象となるオブジェクトである．このクラスは int である tag と，Serializable である contents の 2 つのメンバを持つ．tag は，メッセージの内容を表す ID である．ハンドラは，この ID を見て処理のディスパッチを行う．メッセージの本体は contents に収められる．

4.3 設定ファイル

Jojo では起動時に参加するクラスタ群の構成，起動方法，起動するコードクラスを指定する必要がある．クラスタ群は階層的な構造となるので，これを指定する設定ファイルは階層的な構造を自然に表現できる必要がある．Java で一般的なプロパティ形式ではこの用件を満たすことが難しいので，XML 形式を用いる．**図 6** に設定ファイルの DTD を示す．

設定ファイルには，各ノードのホスト名，実行するコード，起動するための情報が収められる．node 要素には属性値としてホスト名を指定する．ホスト名として default を指定すると，その node の値が，同レベルにあるすべての node のデフォルト値として解釈

```
<!ELEMENT node (code?,invocation?,node*)>
<!ATTLIST node host CDATA #REQUIRED>
<!ELEMENT code (#PCDATA)>
<!ELEMENT invocation EMPTY>
<!ATTLIST invocation
        javaPath CDATA #IMPLIED
        javaOptions CDATA #IMPLIED
        rjavaProtocol CDATA #IMPLIED
        rjavaRsh CDATA #IMPLIED
        rjavaRcp CDATA #IMPLIED
        xtermDisplay CDATA #IMPLIED
        xtermPath CDATA #IMPLIED
>
```

図 6 設定ファイルの DTD
Fig. 6 The DTD of the configuration file.

される．この機構によってクラスタノードなどの起動情報を共有する多数のノードの設定を容易に記述することができる．具体例を 5 章の図 9 に示す．

4.4 ファイル I/O のサポート

プログラムの多くはファイルの入出力を行う．Jojo はグリッド上に分散したプログラムに対して，クライアントのファイルシステムへのアクセスを提供する．この機能を用いることで，グリッド上のどこで起動されるのかを意識せずに，設定ファイルの読込みやログの書出しをすることができる．

この機能を使用するには，通常の FileReader や FileWriter の代わりに rjava の提供する RemoteFileReader，RemoteFileWriter を使用するだけでよい．このため Jojo を用いるとファイル I/O を持つ Java プログラムであっても容易にグリッド対応とすることができる．

また，この機能はファイルのステージングによってではなく，リアルタイムのストリーム通信によって実現されている．したがって，たとえばログファイルを RemoteFileWriter で書き出すようにしておけば，プログラムの動作状態をリアルタイムで監視することもできる．

5. Jojo によるプログラム例

Jojo によるプログラミング例として，マスタ・ワーカ方式でモンテカルロ法によって円周率を求めるプログラムを示す．**図 7** がマスタ側，**図 8** がワーカ側である．

このプログラムはセルフスケジューリングによる動的負荷分散を行う．ワーカがマスタにジョブを要求し，マスタがジョブを分配する．ジョブの要求と結果の返却を 1 つのメッセージで行うことでプログラムを簡潔にしている．

このプログラムを実行するには Jojo の設定ファイルと，実行プロパティファイルの 2 つが必要となる．pad00 から pad03 の 4 台をリモートサーバとして使

```
public class PiMaster extends Code{
 long times, perNode;
 int  divide;
 boolean done = false;
 long doneTrial = 0, doneResult = 0;

 public void init(Map prop) throws JojoException{
  times = Long.parseLong((String)prop.get("times"));
  divide =
       Integer.parseInt((String)prop.get("divide"));
  perNode = times / divide;
 }

 public void start() throws JojoException{
  synchronized (this){
   while (!done){
    try {wait();}
    catch (InterruptedException e) {}
   }
   System.out.println("PI = " +
     (((double)doneResult/doneTrial)*4));
  }
 }

 synchronized public Object
 handle(Message msg) throws JojoException{
  if (msg.tag == PiWorker.MSG_TRIAL_REQUEST){
   long [] pair = (long[])(msg.contents);
   doneTrial  += pair[0];
   doneResult += pair[1];
   if (doneTrial >= times){
     done = true;
     notifyAll();
     return new Long(0);
   } else
     return new Long(perNode);
  } else
   throw new JojoException(
    "cannot handle the message: " + msg);
 }
}
```

図 7 マスタープログラム
Fig. 7 Master program.

```
public class PiWorker extends Code{
 static final int MSG_TRIAL_REQUEST = 1;
 Random random = new Random();

 public void start()
 throws JojoException{
  long trialTimes = 0;
  long doneTimes  = 0;
  while (true){
   Message msg =
     new Message(MSG_TRIAL_REQUEST,
                 new long[]{trialTimes, doneTimes});
   trialTimes =
            ((Long)(parent.call(msg))).longValue();
   if (trialTimes == 0) break;
   doneTimes = trial(trialTimes);
  }
 }

 private long trial(long trialTimes){
  long counter = 0;
  for (long i = 0; i < trialTimes; i++){
   double x = random.nextDouble();
   double y = random.nextDouble();
   if (x * x + y * y < 1.0)
    counter++;
  }
  return counter;
 }
}
```

図 8 ワーカープログラム
Fig. 8 Worker program.

```
<node host="root">
 <code> PiMaster </code>
 <node host="default">
   <code> PiWorker </code>
   <invocation
    javaPath="java"
    rjavaJarPath=
      "/usr/users/nakada/bin/rjava.jar"
    rjavaProtocol="ssh"
    rjavaRsh="ssh"
    rjavaRcp="scp"
   />
  </node>
  <node host="pad00"/>
  <node host="pad01"/>
  <node host="pad02"/>
  <node host="pad03"/>
</node>
```

図 9 サンプルプログラム用設定ファイル
Fig. 9 Configuration file for the pi program.

用し，sshで実行するには**図 9** のように設定ファイルを書けばよい．

プロパティファイルには以下のように書く．それぞれモンテカルロ試行の回数と，それを何等分して実行するかを指定している．

```
times=100000
divide=100
```

こらのファイルをそれぞれ `jojo.conf` および `pi.prop` とする．実行するには以下のように指定する．

```
> Java silf.jojo.Jojo jojo.conf pi.prop
```

6. 予備的性能評価

予備的な性能評価として，ノード間スループットを測定した．さらに簡単なマスタ・ワーカプログラムを使用して階層的な実行環境の有効性を評価した．

6.1 ノード間スループット

基礎的な評価として，GRAM と GASS を用いた場合（以下，GSI と記載）と，ssh を用いた場合（以下，SSH と記載）のローカルノードとリモートノード間のスループットを測定した．

評価環境としては，**図 10** に示すように，AIST に設置された PC をサーバとして用いた．クライアントは TITECH と AIST にそれぞれ 1 つ用意した．TITECH-AIST 間は 80 km 離れているが WAN としては非常に高速なネットワークで接続されており，スループットは 10 Mbytes/s，レイテンシは 7 ms 程度である．AIST 内のクライアントとサーバ間はギガビットイーサで直結されており，スループットは 80 Mbyte/s，レイテンシは 1 ms 以下である．以下 TITECH-AIST 間の実験を WAN とし，AIST 内の実験を LAN と

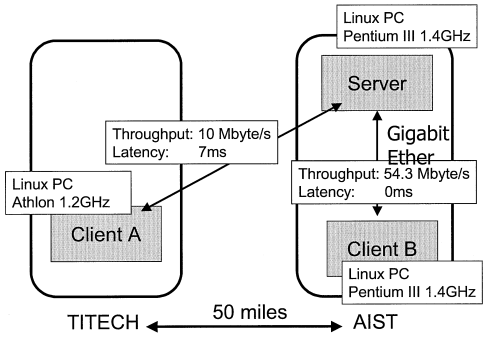

図 10 スループット評価環境
Fig. 10 Environment for throughput measurement.

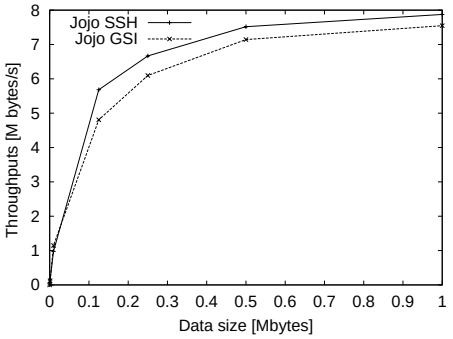

図 12 WAN 環境でのスループット
Fig. 12 Throughput in WAN environment.

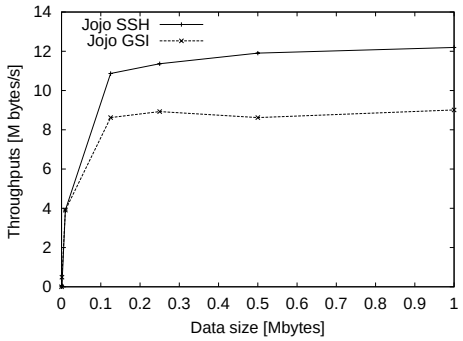

図 11 LAN 環境でのスループット
Fig. 11 Throughput in LAN environment.

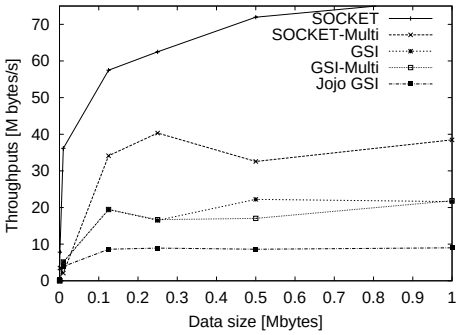

図 13 LAN 環境で GSI を使用した場合のコスト
Fig. 13 GSI cost breakdown.

する．

6.1.1 結　　果

図 11 に LAN 環境での結果を示す．SSH が 12 MByte/s，GSI が 8 MByte/s と，絶対値としては十分高速だが，ギガビットイーサの 80 MByte/s に達するバンド幅と比較すると，かなりの性能低下がみられる．また，GSI 版の性能は SSH 版よりも低いことが分かる．

図 12 に WAN 環境での実験結果を示す．LAN 環境と同様に SSH 版のほうが性能が良いものの，GSI との差が縮まっている．また，ネットワークバンド幅に対する性能は 7 割程度となっており，LAN 環境よりも性能低下の割合が小さい．これは通信速度が遅いために，Jojo のオーバヘッドが隠れているためであると考えられる．

6.1.2 コストの解析

上述のスループット低下の原因としては，大別して以下のコストが考えられる．

(1) 通信路のコスト
SSH は通信に対して暗号を行う．GSI では MD5 によるメッセージタイジェストを行う．これらの操作には計算コストを要する．

(2) マルチプレクスのコスト
Jojo では 1 つの通信ストリームを，マルチプレクスして複数の通信ストリームとして使用し，ユーザプログラムのアップロード，標準出力のリダイレクトなどに用いている．このマルチプレクスには余分なバッファのコピーや，余分なスレッド切替えといったコストがかかる．

(3) (2) 以外の Jojo 内部のコスト
Jojo 内部では上記のほかに，データ構造のシリアライズ，デシリアライズ，ハンドラスレッドの作成とそのスレッドへの切替え，などのコストが存在する．

これらのコストを切り分けるために，上記 (1) および (2) のコストを独立に付加できるテストプログラムを作成し，LAN 環境において測定を行った．その結果を**図 13** に示す．この実験には GSI 版を用いた．

図中の **SOCKET** は通常の TCP を直接用いた場合，**SOCKET-Multi** は TCP 上で直接マルチプレクサを実行した場合（上記 (2) のコスト），**GSI** は Globus-IO で直接通信した場合（上記 (1) のコスト），**GSI-Multi** は Globus-IO 上でマルチプレクサを実行した場合（上記 (1) と (2) のコスト），**Jojo-GSI** は

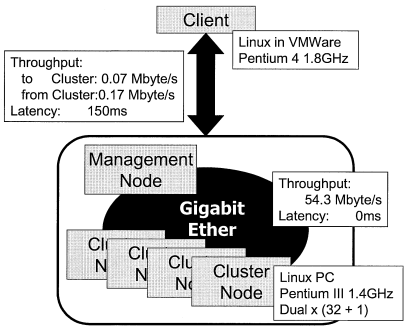

図 14 マスタ・ワーカ評価環境
Fig. 14 Master-Worker evaluation environment.

GSI を用いた Jojo（上記 (1)，(2)，(3) のコスト）をそれぞれ表す．

SOCKET-Multi で 40 Mbyte/s 程度，**GSI** で 20 Mbyte/s 程度となっていることから，マルチプレクサのコストやメッセージダイジェストのコストが大きいことが分かる．ただしこの 2 つを組み合わせた **GSI-Multi** は，**GSI** とほぼ同程度の速度が出ていることから，マルチプレクスとメッセージダイジェストと組み合わせた場合には，メッセージダイジェストのコストが支配的となり，マルチプレクスのコストが隠蔽されることが分かる．

また，**Jojo GSI** と **GSI-Multi** の差から，マルチプレクス，暗号化以外の Jojo 内部コストがかなり大きいことが分かる．

6.2 マスタ・ワーカプログラムでの評価

Jojo の提供する多階層実行環境の有効性を検証するために，マスタ・ワーカプログラムでの評価を行った．

6.2.1 実 験 環 境

マスタ・ワーカプログラムとしては 5 章に示したものを用いた．評価環境には AIST 内のクラスタを用い，これを外部に設置したクライアントから使用した．クラスタの諸元，クライアントとのネットワーク性能などを**図 14** に示す．

クライアントと AIST は CATV ネットワークを介して接続されており，上りが 0.17 Mbyte/s，下りが 0.17 Mbyte/s，レイテンシはラウンドトリップで 300 ms 程度である．クラスタ内部のネットワークはギガビットイーサで，スループットは 54.3 Mbyte/s，レイテンシは 0 ms である．

実験は **2 層モデル**と **3 層モデル**の 2 つで行った．2 層モデルでは 1 層目（クライアント）にマスタを設置し，2 層目にワーカを配置した．3 層モデルでは，1 層目ではなにも行わず，2 層目にマスタを，3 層目にワーカを設置した（**図 15**）．

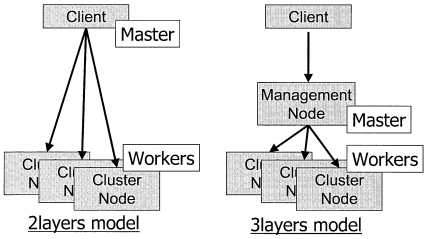

図 15 2 階層モデルと 3 階層モデル
Fig. 15 2-layer setup and 3-layer setup.

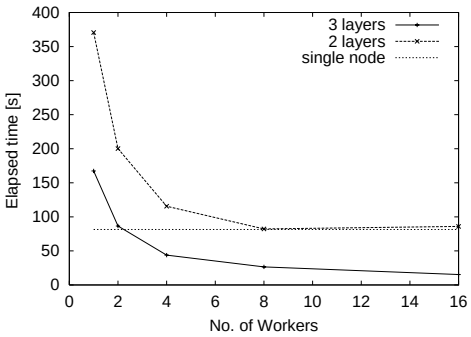

図 16 2 層モデルと 3 層モデルでのマスタ・ワーカ計算の結果
Fig. 16 Master-Worrker result in 2-layer and 3-layer.

ワーカとして使用する計算機は，両モデルともに同一のものを使用している．3 層モデルでは，クラスタの管理ノードを 2 層目とし，ここにマスタを設置する．クラスタの管理ノードはクラスタのノードと同じスイッチに接続されており，クライアントと管理ノード間の通信環境はクライアントと各クラスタノード間の通信環境と同一であると考えられる．

実験の乱数試行回数は総計 1 億回とし，これを 1 万回ずつに分割して 10,000 個のジョブを作成して，マスタ・ワーカで処理することとした．このプログラムをワーカに使用した計算機単体で実行した場合の所要時間は 81.4 秒程度である．したがって，個々のジョブの実行時間は 8 ms 程度である．また，すべてのノード間接続には ssh を使用している☆．

6.2.2 実 験 結 果

実験の結果を**図 16** に示す．この実験はクラスタを占有して行うことができなかったため，数回実行した結果のうち最良の値を実験結果として採用した．図中 single-node と示したのが，クラスタのノードで単体

☆ ssh はシングルサインオンではないため，通常は 2 層目から 3 層目への接続に ssh を使用することはできない．ここでは実験のために あらかじめ 2 段目にログインして ssh-agent を実行しておき，このときの環境変数 SSH_AUTH_SOCK と SSH_AGENT_PID を Jojo に与えることでこの構成を実現した．

プログラムを実行した際の結果である．

3層モデルのほうが全般的に2層モデルよりも高速であることが分かる．また，2層モデルでは8ノード以上では実行時間の短縮がみられない．これに対して3層モデルでは16ノードでも実行時間は短縮されている．

6.2.3 議　　論

評価対象のプログラムは，ワーカでの実行時間が8 msと非常に短く，グリッド上での並列実行には不利なプログラムである．これは従来のGridRPCでの実行形態である2層モデルでは性能が出ないことからも裏づけられる．これに対して，Jojoの提供する多階層実行環境で実現された3層モデルでは，十分高速に実行できている．これは，マスタ・ワーカ計算が高速なLAN内で閉じており，事実上LANのみを使用して実行を行っているためである．この結果は，多階層実行環境による3層モデルを用いれば，従来グリッド上での実行には適さないとされていたアプリケーションを実行の対象にできる可能性があることを示唆している．

ここで評価に用いたプログラムでは，マスタ・ワーカ間でやりとりされるジョブのデータサイズは入出力ともにlong値1つのみであり，この種のプログラムとしては非常に小さい．したがって，スループットが小さい環境でもそれほど大きい影響を受けるとは考えにくい．にもかかわらず，実行時間が大きく異なるのは，マスタ・ワーカ間のレイテンシの影響であると考えられる．ワーカはジョブの実行を終了すると，新しいジョブのリクエストをマスタに投げるが，その間ワーカはアイドルすることになる．このため実行の効率が低下したものと考えられる．

また，この実験において，サーバ側のクラスタにはJojoのシステムや実行対象となるプログラムのインストールを行う必要はまったくなかった．これは，Jojoの動的なコード転送によるユーザ支援の有効性を示すものである．

7. 関 連 研 究

本章では，Javaによる通信ライブラリのプログラミングモデルに関する関連研究を示す．

7.1 RMI（Remote Message Invocation）

RMIの計算モデルは，同期的なリモートメソッド呼び出しを基本とした分散オブジェクトモデルである．このモデルの一般性は高く，基本的にどのような計算機構でも記述することはできるが，並列実行を行う場合にはユーザがスレッドと組み合わせなければならない，起動時にオブジェクトの生成と公開をサーバ側で行うという手間が必要，といった問題点がある．

7.2 メッセージパッシングライブラリ

Javaにおけるメッセージパッシングライブラリとしては，Javaによる PVM 実装である JPVM[9]，Cによる MPI 実装を呼び出すラッパである mpiJava[10] がある．また，pure Javaによる MPI 実装も行われている[11]．

MPI や PVM はデータを send や recv で明示的に送受信するメカニズムを提供する．これらは貧弱なスレッド環境しか持たない C や Fortran 向けに開発されたメッセージ転送モデルであり，スレッドが言語レベルで高度に統合されている Java 言語に適しているとはいいがたい．MPI 標準はマルチスレッドに言及しているが，各関数のスレッドセーフ性に関する言及が主で，たとえば，複数のスレッドが，同一のタグに対して同時に recv を行った場合の挙動に関しては，errorneous と定義しているだけである．

8. お わ り に

本稿では，グリッド環境での Java プログラミングを支援する実行環境 Jojo の設計と実装について述べた．さらにマスタ・ワーカ型プログラムを例として性能評価を行い，多階層実行環境の有効性を示した．

現在，NMR 画像からの蛋白質の構造決定を遺伝的プログラミングで行うシステム[12]を JoJo 上に実装し，システムの性能評価とスケーラビリティの検証を行っている[13]．また，分枝限定法に本システムを適用した，汎用の分枝限定法による最適化システムも開発中である[14]．

今後の課題としては以下があげられる．

- より複雑な構造を持つプログラムを Jojo の多階層実行環境にマップし，有効性を検証する．現在我々は，遺伝的アルゴリズムや分枝限定法などの組合せ最適化問題をグリッド環境で実行するためのフレームワーク jPoP[15]を，Jojo を用いて開発している．この開発を通じて Jojo の有用性とスケーラビリティを確認する．
- 現在の実装では sibling 間の通信は直接行われず，parent を介して行われている．これは，現在対象としている最適化問題では sibling 間の通信が必要とされないためである．また，実行環境によっては sibling 間の安全な直接通信が不可能な場合もある．しかし逆に，たとえばクラスタ内のように通信の安全性に特に配慮する必要がない場合も多い．今後，このような場合を対象に直接通信の

実装を検討していく．

- グリッド環境では対故障性は本質的に重要であるが，現在の実装では故障発生時の処理に関しては通常のエラーハンドリング以上の配慮はしていない．今後はAPIの見直しも含めて，対故障性の実現を検討する．また動的なノード群の再構成に関しても検討する必要がある．

参考文献

1) Foster, I. and Kesselman, C.: Globus: A metacomputing infrastructure toolkit, *Proc. Workshop on Environments and Tools, SIAM* (1996).
2) 田中良夫，中田秀基，平野基孝，佐藤三久，関口智嗣：Globusによる Grid RPCシステムの実装と評価，情報処理学会ハイパフォーマンスコンピューティングシステム研究会，No.77 (2001).
3) Casanova, H. and Dongarra, J.: NetSolve: A Network Server for Solving Computational Science Problems, *Proc. Super Computing '96* (1996).
4) Roy, A., Foster, I., Gropp, W., Karonis, N., Sander, V. and Toonen, B.: MPICH-GQ: Quality-of-Service for Message Passing Programs, *Proc. IEEE/ACM SC2000 Conference* (2000).
5) 中田秀基，松岡 聡，関口智嗣：グリッド環境に適したJava用階層型実行環境Jojoの設計と実装，情報処理学会HPC研究会2002-HPC-92 (2002).
6) Tanaka, Y., Hirano, M., Sato, M., Nakada, H. and Sekiguchi, S.: Performance Evaluation of a Firewall-compliant Globus-based Wide-area Cluster System, *9th IEEE International Symposium on High Performance Distributed Computing* (*HPDC 2000*), pp.121–128 (2000).
7) Breg, F., Diwan, S., Villacis, J., Balasubramanian, J., Akman, E. and Gannon, D.: Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++, *ACM 1998 Workshop on Java for High-Performance Network Computing* (1998).
8) 中田秀基，早田恭彦，小川宏高，松岡 聡：Javaによるソフトウェア分散共有メモリシステムの構築—広域環境への対応，情報処理学会PRO研究会 (2000).
9) Ferrari, A.J.: JPVM: Network parallel computing in Java, *ACM 1998 Workshop on Java for High-Performance Network Computing* (1998).
10) Baker, M., Carpenter, B., Fox, G., Ko, S.H. and Lim, S.: mpiJava: An Object-Oriented Java interface to MPI, *International Workshop on Java for Parallel and Distributed Computing* (1999).
11) 日下部明，廣安知之，三木光範：JAVAによるMPIの実装と評価，並列処理シンポジウムJSPP2000論文集，pp.269–275 (2000).
12) Ono, I., Fujiki, H., Ootsuka, M., Nakashima, N., Ono, N. and Tate, S.: Global Optimization of Protein 3-Dimensional Structures in NMR by a Genetic Algorithm, *Proc. 2002 Congress on Evolutionary Computation*, pp.303–308 (2002).
13) 中田秀基，中島直敏，小野 功，松岡 聡，関口智嗣，小野典彦，楯 真一：グリッド向け実行環境Jojoを用いた遺伝的アルゴリズムによる蛋白質構造決定，情報処理学会HPC研究会2002-HPC-93，pp.155–160 (2003).
14) 秋山智宏，中田秀基，松岡 聡，関口智嗣：グリッド環境に適した並列組み合わせ最適化システムjPoPにおける分枝限定法の実装，*SPA 2003* (2003).
15) 秋山智宏，中田秀基，松岡 聡，関口智嗣：Grid環境に適した並列組み合わせ最適化システムの提案，情報処理学会HPC研究会2002-HPC-91 (2002).

(平成15年2月 4 日受付)
(平成15年5月30日採録)

中田 秀基（正会員）

1967年生．1990年東京大学工学部精密機械工学科卒業．1995年同大学大学院工学系研究科情報工学専攻博士課程修了．博士（工学）．同年電子技術総合研究所研究官．2001年独立行政法人産業技術総合研究所に改組．現在同所グリッド研究センター主任研究官．2001年より東京工業大学客員助教授を兼務．グローバルコンピューティング，並列実行環境に関する研究に従事．

松岡　聡（正会員）

1963 年生．1986 年東京大学理学部情報科学科卒業，1989 年同大学大学院博士課程から，情報科学科助手に採用，同大学情報工学専攻講師を経て，1996 年に東京工業大学情報理工学研究科数理・計算科学専攻助教授．2001 年 4 月に東京工業大学学術国際情報センター教授，2002 年より国立情報学研究所の客員教授を併任．博士（理学）（東京大学）．高性能システム，並列処理，グリッド計算，クラスタ計算機，高性能・並列オブジェクト指向言語処理系等の研究に従事．ソフトウェアの技術開発によりコモディティ技術の大幅な活用で従来の 100 倍の計算パワーを計算科学に広域に提供することを目指す．現在進行中のプロジェクトは，(1) 産業技術総合研究所等と共同して種々のグリッド計算のプロジェクト（Ninf，GFarm プロジェクト等），(2) 大規模コモディティPC クラスタ構築プロジェクト（Presto クラスタ群），ならびに (3) 計算環境に適合・最適化を目指す Java 言語の開放型 Just-In-Time コンパイラ OpenJIT，JDSM 等．1996 年度情報処理学会論文賞，1999 年情報処理学会坂井記念賞受賞．国際学会 ISOTAS'96，ECOOP'97，ISCOPE'99 のプログラム委員長や Reflection2001 の大会委員長等を務め，2002 年に ACM OOPSLA'2002，2003 年には IEEE CC-Grid のプログラム委員長．また，Global Grid Forum の Steering Group 委員ならびに Area Director を務める．日本ソフトウェア科学会，ACM，IEEE-CS 各会員．

関口　智嗣（正会員）

1959 年生．1982 年東京大学理学部情報科学科卒業．1984 年筑波大学大学院理工学研究科修了．同年電子技術総合研究所入所．情報アーキテクチャ部主任研究官．以来，データ駆動型スーパーコンピュータ SIGMA-1 の開発等の研究に従事．2001 年独立行政法人産業技術総合研究所に改組．2002 年 1 月より同所グリッド研究センターセンター長．並列数値アルゴリズム，計算機性能評価技術，グリッドコンピューティングに興味を持つ．市村賞受賞．日本応用数理学会，日本ソフトウェア科学会，SIAM，IEEE 各会員．