動的なノード群構成機構を備えた階層型グリッド環境: Jojo2

青木仁志[†1] 中田秀基[†2,†1]
田中康司[†3] 松岡聡[†1,†4]

グリッドの階層構造に適合したプログラミングモデルの一つとして階層型マスタ・ワーカ方式がある。しかし既存の階層型マスタ・ワーカ方式を実現するミドルウェアは、動的なノード群構成や動的なノードの参加・脱退といった機構が欠如しているため管理コストが大きい。我々は動的なノード群構成機構を備えた階層型グリッドプログラミング環境 Jojo2 を提案する。Jojo2 はグリッドの階層構造に合致した動的なノード構成を行い、自律的なノードの発見、動的なノードの参加・脱退を行うことで大規模グリッド環境においても管理コストを軽減することができる。さらに動的なノードの参加・脱退をハンドルするプログラミング API を提供することで、動的構成に対応したプログラムを容易に記述できる。また予備的性能評価を行い、その結果、ノードの参加・脱退の API の有効性と、ノードの参加・脱退のコストが小さいことを確認した。

A Programming Environment with Dynamic Node Configuration for Hierarchical Grid: Jojo2

Hitoshi AOKI,[†1] Hidemoto NAKADA,[†2,†1] Kouji TANAKA[†3]
and Satoshi MATSUOKA[†1,†4]

There is a hierarchical master worker style as one of programming models suitable for hierarchical Grid. However its existing middleware lacks dynamic node configuration, node joining and leaving. Therefore they are a heavy burden for users. We propose a programming environment with dynamic node configuration for hierarchical Grid; Jojo2. Jojo2 supports autonomic node discovery and dynamic node configuration. In addition, Jojo2 provides programming API suitable for dynamic node configuration. Therefore Jojo2 reduce the user's burden. We also show preliminary performance evaluation result that prove effectiveness of programming API enabling node joining and leaving, and little cost of node joining and leaving.

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。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り、特に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。

このようなグリッド環境に適したプログラミング環境として、階層型マスタ・ワーカ方式が提案されている[1)2)3)]。階層型マスタ・ワーカ方式では、従来のマスタ・ワーカ方式におけるマスタの機能を複数のサブマスタに分担することで、単一マスタへの負荷集中の問題を解決する。さらにサブマスタとワーカを同一のクラスタ内に配置することで、可能な限り通信をクラスタ内に局所化する。それによりインターネット上で生じる通信を減らし、オーバヘッドの軽減が可能となる。

しかし既存の階層的なマスタ・ワーカのプログラミング環境では、動的なノード群の構成や動的なノードの参加・脱退といった機構が欠如している。グリッド環境上のクラスタやそれらを接続するネットワークは多数の利用者によって共有されていることが一般的であるので、一部の計算ノードがダウンして利用不可能になることは頻繁にある。また共有されているが故に、他のユーザの要望に応じて一部のノードを解放したり、あるいは逆に空いた計算ノードを追加したいという要求が生じたりすることもあり得る。また数千台

†1 東京工業大学
Tokyo Institute of Technology
†2 産業技術総合研究所
National Institute of Advanced Industrial Sceince and Technology (AIST)
†3 早稲田大学
Waseda University
†4 国立情報学研究所
National Institute of Information

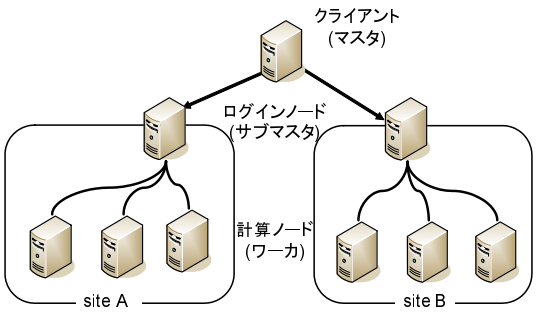

図 1 階層構造

規模の大規模グリッド環境下では、ユーザが使用するノードを手動で管理するには非常にコストが高い。そのため大規模グリッド環境下では、自律的なノードの発見、動的なノードの参加や脱退といったノード群の動的構成が必要不可欠となる。またさらにノード群の動的構成を可能にするプログラミング API もまた必要である。

本稿では、動的なノード群構成機構を備えたプログラミング環境 Jojo2 を提案する。Jojo2 はグリッドの階層構造に適合した動的なノード群の構成を行うことで、自律的なノードの発見、動的なノードの参加・脱退を可能にする。また動的構成を前提としたプログラミング API を提供することで、ノードの参加・脱退に柔軟に対応したプログラミングが容易に可能となる。

2. 要　　件

大規模グリッド環境における階層的マスタ・ワーカ方式のプログラミング環境の要件として以下が挙げられる。

- ノードが数千台規模となる環境では全てのノードを手動で管理するには非常にコストが大きく、現実的ではない。そのため最小限の設定で実行可能でなくてはならない。
- グリッド環境は大規模であればあるほど不安定であるので、実行時にノードが落ちることが頻繁に生じる可能性があり、耐故障性が必要不可欠である。また故障ノードへの対処や資源の動的な利用状況の変化に対処するため、実行中にノードの参加・脱退が可能でなくてはならない。
- プログラミング API のレベルで動的なノードの参加・脱退に柔軟に対処できなくてはならない。
- 動的にノード数が変化するのでノード数に依存しないプログラミングモデルを提供しなくてはならない。

3. Jojo2 の設計

3.1 システム構造

グリッドにおける階層構造では、クラスタの外部に構成されるネットワークとクラスタの内部に構成されるネットワークの 2 階層構造をとる。 そこで Jojo2 ではこのグリッド環境の階層構造を反映したシステム構造をとる (図 1)。

このときクラスタ外部のネットワークとクラスタ内部のネットワークで異なるアプローチによってシステムを構成する必要があると考えられる。なぜならクラスタ外部のネットワークでは、ユーザはどのサイトを利用するか意識するため、各サイトのログインノードを明示する必要があるのに対し、クラスタ内部の計算ノードは台数が多く、一般的にユーザは具体的にどのノードを利用するかということは意識する必要がないからである。またサイトによっては、バッチスケジューラ経由でのジョブ実行しか許可していない場合もあるので、ユーザがあらかじめ利用するノードを指定するといった手法は適さない。さらに、ユーザが各サイトの計算ノードを全て手動で管理するのは非常にコストが高い。

そこで Jojo2 ではクラスタ外部のネットワークにおいては従来の Jojo と同様に手動管理によるコンフィグレーションを用い、一方でクラスタ内部のネットワークでは動的にノード群を構成することでツリートポロジを構成する。

具体的には、まずクラスタ外部のネットワーク、すなわち各サイトのログインノードなどに対してはあらかじめユーザが指定したサブマスタプロセスの起動を行う。これはインターネット上での通信となるため、セキュリティを確保する必要がある。そこで Globus GRAM や ssh を用いることでセキュアな起動と通信を行う。

一方で、クラスタ内においては計算ノードは数千台規模となることもあり得、その管理をユーザ手動で行うには非常にコストが高いものとなってしまう。そこで UDP ブロードキャストを利用した動的なノード群構成を行うことで、自律的なノードの発見、動的なノードの参加・脱退を実現する。すなわち典型的には各サイトのログインノードの情報のみで、 Jojo2 は動的にそのトポロジを構築することができる。

グリッド環境上のクラスタやネットワークは多数のユーザに共有されていることが一般的である。そのためクラスタ各ノードやネットワークの負荷は動的に変動する。そのためクラスタ内におけるワーカプロセスの起動では Condor[4] や Jay[5] といったジョブスケジューラを利用することを想定している。

また耐故障性のための故障検知として Jojo2 では Heartbeat & Timeout による故障検知を行う。また故障ノードに対する対処として計算を止めることなく、実行時に任意のタイミングでノードを参加させることが可能である。

3.2 プログラミングモデル

Jojo2 では以下の点を考慮したプログラミング API

を提供する。

- グリッド環境に合致した階層性
- ノードの動的な参加・脱退に対する処理のサポート
- ノード数に依存しないメッセージパッシング機構

3.2.1 階　層　性

Jojo2 はグリッド環境の階層構造に適合したトポロジを構築するが、その各階層のレイヤによらず同じプログラミングモデルでプログラムを実装できる必要がある。そこで Jojo2 では階層構造を持つノードのそれぞれで実行されるプログラムを記述し、レイヤによらず同一のクラスを継承することでプログラミングが可能なクラスフレームワークを提供する。ただし典型的には一つのレイヤで同一のクラスオブジェクトを実装する。

3.2.2 動的なノードの参加・脱退に対する処理

ノードの動的な参加・脱退をサポートをするためには、それらに対する処理が容易に記述できる枠組みが必要である。そこで Jojo2 ではノードの参加・脱退に対するハンドラメソッドを提供する。ユーザはこのメソッドを実装することで、ノードの参加・脱退に対する処理を行うことができる。このハンドラメソッドは計算処理と別スレッドで実行するため、ノードの参加・脱退に対する処理を計算処理と独立して行うことが可能である。

3.2.3 メッセージパッシング機構

各ノードで実行されるクラスオブジェクトは、上位レベルのノード、下位レベルのノード群と通信可能なメッセージパッシング API が提供される。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。受信を行うハンドラメソッドは、ノードの参加・脱退に対するハンドラメソッドと同様に計算処理と別スレッドで実行する。

また送信に関しては Jojo では上位レベル、下位レベル共に一対一通信をサポートしていたが、ノード数が動的に増減するような環境においては、親ノードが特定子ノードと一対一通信を頻繁に行うようなプログラミングモデルは適さないと考えられる。そのため Jojo2 では下位レベルとの通信はブロードキャストのみをサポートする。一方で上位レベルへの通信は Jojo と同様に一対一通信をサポートする。

すなわち Jojo2 におけるマスタ・ワーカ方式のプログラムは、ワーカがマスタに対してジョブを要求する pull 型のプログラミングモデルを推奨している。マスタはどのノードにジョブを配送したかを記憶しておく必要がある。そのため各子ノードはランダムな識別子を持つが、ユーザはこの識別子に依存してプログラムを記述する必要はない。このように記述力を制限することで動的構成に適したプログラミングが容易に可能になると考えられる。

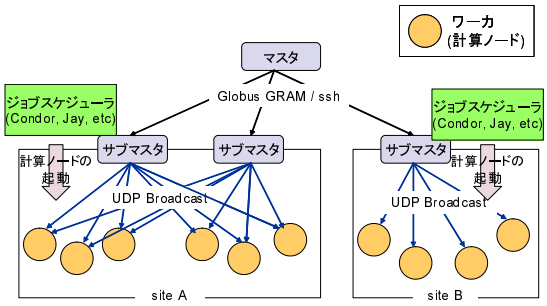

図 2　動的なノード群構成

4. 実　　装

4.1 動的なノード群構成

Jojo2 では UDP ブロードキャストを用いた自律的なノードの発見、動的なノード群の構成を行う。Jojo2 のネットワークトポロジの構築について 2 層構造を例にして述べる。(図 2)

(1) ユーザはマスタプロセスをクライアントとなるノードで起動する。

(2) 次にユーザは Condor や Jay のような各サイトのジョブスケジューラなどを用いて、各サイトの計算ノード上にワーカプロセスを起動する。

(3) マスタプロセスは、あらかじめユーザが設定ファイルに記述したノードに対して、サブマスタプロセスを Globus GRAM あるいは ssh を用いて起動し接続する。このときサブマスタプロセスを起動するノードは典型的には各サイトのログインノードなどになる。

(4) サブマスタプロセスは自身に接続されたワーカの数を定期的に UDP ブロードキャストする。

(5) ワーカプロセスは起動時直後は UDP パケット待ちしており、サブマスタから UDP パケットを受信するとサブマスタに接続する。

ワーカプロセスがサブマスタに接続する際に、ワーカプロセスが UDP パケットを受信してすぐに接続した場合、ある特定のサブマスタに多数のワーカが集中してしまう可能性がある。そこで各ワーカプロセスはランダムな時間だけ UDP パケット待ちし、その中で最も接続数の少ないサブマスタに対して接続することで負荷分散を実現する。

4.2 動的なノードの参加・脱退

サブマスタは定期的に UDP ブロードキャストを行っているので、ユーザが計算ノード上にワーカプロセスを起動するだけで、自律的にそのノードを発見することができる。そのため任意のタイミングでノードを計算に参加させることができる。

サブマスタも実行中に参加させることが可能である。マスタは常にポート待ち受けしており、マスタの IP

を指定してサブマスタを起動することでそのマスタに接続することができる。

そのため現在計算に利用しているクラスタとは別のクラスタログインノード上にサブマスタプロセスを起動し、そのサイト内の各計算ノードに対してワーカプロセスを起動することで、計算を途中で止めることなく実行中に新たなクラスタを計算に参加させることもできる。

4.3 耐故障性

各レイヤで故障ノードが生じたときの処理について述べる。まずワーカノードで故障が発生した場合、その上位であるサブマスタでは下位ノード脱退用のハンドラメソッドが実行される。このハンドラメソッドについては詳しくは 4.4 節で述べる。このハンドラメソッドを並列に実行しながら、サブマスタは通常通り計算を続けることができる。また必要であれば計算ノード上に再びワーカプロセスを起動するだけでワーカを参加することができる。

次に中間ノードであるサブマスタがダウンしたときの処理について述べる。サブマスタがダウンした場合、ワーカのときと同様にその上位ノードであるマスタで計算処理と独立して下位ノード脱退用のハンドラメソッドが実行される。

また上位との接続を失ったワーカノードは起動直後と同様の UDP パケット待ちの状態になり、再度サブマスタからのパケット受信を試みる。このとき同サイト内に別のサブマスタが起動している場合、そのサブマスタからパケットを受信し接続することで、再度計算に参加することができる。

マスタが故障したときの処理に関しては現在サポートしていない。

4.4 クラスフレームワーク

Jojo2 上でのプログラミングは、Jojo2 の提供する抽象クラス `Code` を継承したクラスを実装することで行う。 `Code` では `ParentNode`, `Descendants`, `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。

4.4.1 Code

`Code` クラスの定義を図 3 に示す。 `parent`, `descendants` はそれぞれ上位レベル、下位レベルのノードを指し、これら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。

`init` メソッドは初期化を行う。引数となる Map には Jojo2 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` メソッド終了以前に `start` メソッドや 各種ハンドラ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。

上位レベル、下位レベルからメソッドを受信すると、`handleReceiveParent`、`handleReceiveDescendant` メソッドがそれぞれ起動される。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。また下位レベルのノードが参加、脱退したときは、`handleAddDescendant`、 `handleDeleteDescendant` がそれぞれ起動される。これらのハンドラメソッドを実行するスレッドは、それぞれのイベント発生時に新たに起動される。

```
public abstract class Code {
  ParentNode parent; /* 親ノード */
  Descendants descendants; /* 子ノード */

  /* 初期化 */
  public void init(Map prop);
  /* 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 void handleReceiveParent(Message msg);
  public Object handleReceiveDescendant(Message msg);

  /* 子ノードの参加、脱退に対する処理 */
  public void handleAddDescendantNode(int nodeID);
  public void handleDeleteDescendantNode(int nodeID);
}
```

図 3 Code クラス

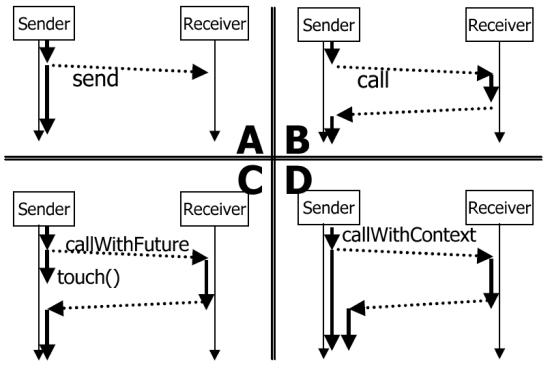

図 4 Jojo の通信モード

4.4.2 ParentNode, Descendants

`Code` クラスでは `ParentNode` クラス、`Descendants` 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、上位レイヤ、下位レイヤと通信を行う。

上位レイヤのノードに相当する `ParentNode` クラスには柔軟な通信ができるよう 4 つのメソッドを提供する。

`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 callWithFuture(Message msg)`
: Future 機構を用いた非同期通信機構を実現する。このメソッドはメッセージオブジェクトを送信し、直ちに返信オブジェクトの `Future` を返す。`Future` オブジェクトの `touch()` メソッドを呼ぶ

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

図 5 Context インタフェース

とそこで同期が行われる。(図 4:C) 。

`void callWithContext(Message msg, Context context)`

返信オブジェクト受信時に実行す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。

下位レイヤのノード群に相当する `Descendants` クラスには以下の 2 つのメソッドが提供されている。

`void broadcast(Message msg)`

メッセージオブジェクトを全ての子ノードに送信する。送信後はすぐにリターンする。これは分枝限定法などで暫定解の分配などの用途を想定している。

`int size()`

自身に接続されている子ノードの数を返す。

4.4.3 Message

`Message` は送信対象となるオブジェクトである。`Message` クラスは以下のようなメンバを持つ。

int nodeID

メッセージ送信元の子ノードの ID。

int tag

メッセージの内容を表す ID。ハンドラは、この ID を見て処理のディスパッチを行う。

Serializable contents

メッセージの本体。

5. Jojo2 によるプログラム例

Jojo2 によるプログラムの例として、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 6 がマスタ側、図 7 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ひと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。またノード脱退時の処理として、故障ノードに割り当てたジョブをジョブキューである `jobQueue` に戻すことでノード障害に対応することができる。

```
public class PiMaster extends Code {
  boolean done = false;
  long times, doneTrial = 0, doneResult = 0;
  LinkedList<Long> jobQueue = new LinkedList<Long>();
  HashMap<Integer, Long> jobMap = new HashMap<Integer, Long>();

  public void init(Map args) {
    times = Long.parseLong((String)args.get("times"));
    int divide = Integer.parseInt((String)args.get("divide"));
    long perNode = times / divide;
    for(int i=0; i < divide; i++) {
      jobQueue.add(perNode);
    }
  }
  public synchronzied void start() {
    while(! done) {
      try { wait(); }
      catch(InterruptedException e){}
    }
    System.out.println("PI = "
                       + ((double)doneResult)/doneTrial*4);
  }
  public synchronized Object
  handleReceiveDescendant(Message msg) {
    if(msg.tag == PiWorker.MSG_TRIAL_REQUEST) {
      if(jobMap.containsKey(msg.nodeID)) {
        doneTrial  += jobMap.remove(msg.nodeID);
        doneResult += (Long)(msg.contents);
      }
      while(doneTrial < times) {
        if(! jobQueue.isEmpty()) {
          long perNode = jobQueue.remove();
          jobMap.put(msg.nodeID, perNode);
          return perNode;
        }
        while(jobQueue.isEmpty()) {
          try { wait(); }
          catch(InterruptedException e) {}
        }
      }
      done = true;
      notifyAll();
      return 0L;
    } else {
      return null;
    }
  }
  public synchronized void
  handleDeleteDescendantNode(int nodeID) {
    long perNode = jobMap.remove(nodeID);
    jobQueue.add(perNode);
    notifyAll();
  }
}
```

図 6 マスタプログラム

表 1 PrestoIII クラスタのスペック

CPU	Opteron 242
Memory	2GBytes
OS	Linux 2.4.30
Network	1000Base-T

6. 予備的性能評価

予備的性能評価として、ノード間スループットを測定した。さらに遺伝的プログラミングによる遺伝子ネットワーク推定アプリケーション[6)7)] を Jojo2 を用いて並列化し、Jojo2 のスケーラビリティ、耐故障性を評価した。

6.1 ノード間スループット

基礎的な評価として、暗号化なしの場合と、 ssh を用いた場合 (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

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

  public void start() {
    long doneTimes = 0, trialTimes;
    while(true) {
      Message msg = new Message(MSG_TRIAL_REQUEST, doneTimes);
      trialTimes = (Long)parent.call(msg);
      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 7　ワーカプログラム

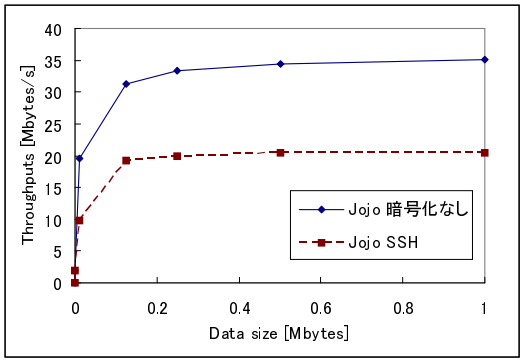

図 8　Jojo2 のスループット

ては、本研究室の PrestoIII クラスタを用いた (表 1)。各クラスタノードはギガビットイーサで接続されており、Iperf で測定したところバンド幅は 72 Mbytes/s であった。

図 8 に ssh を用いた場合の通信と、暗号化なしの場合の通信のスループットを示す。暗号化しなかった場合 35 MBytes/s、SSH が 20 MBytes/s と、絶対値としては十分高速であるが、72 MBytes/s と比較するとかなりの性能低下が見られる。また SSH を用いた場合、暗号化のコストにより 4 割程度性能が低下していることがわかる。

性能低下の原因としては、ストリームをマルチプレクスするコスト、データハンドリングのためのスレッド切り替えのコストが挙げられる。また SSH を用いた場合では暗号化によるコストが上げられる。

6.2　遺伝的プログラミングでの評価

遺伝的プログラミングによる遺伝子ネットワーク推定アプリケーション[6)7)] を、Jojo2 を用いてマスタ・サブマスタ・ワーカの 3 層モデルによる実装を行った。本アプリケーションは遺伝的アルゴリズムと同様な世代交代を繰り返すことで、遺伝子間の相互関係を

表 2　ワーカ上での平均処理時間

刻み幅	1/4	1/8	1/16
処理時間 [ms]	143	228	490

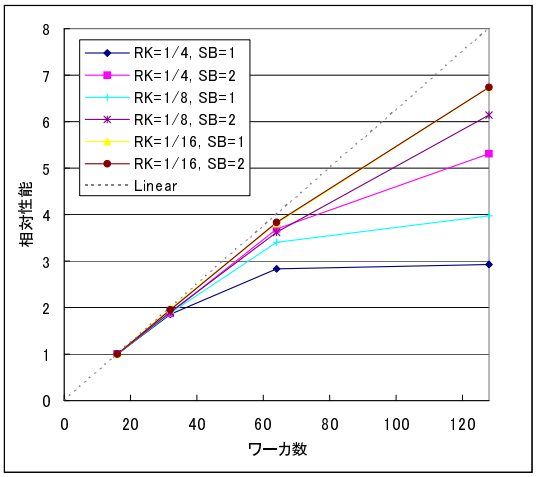

図 9　ワーカ数と並列化効率

表す微分方程式を求める。本アプリケーションではマスタは個体の生成、母集団の管理、世代交代の評価を行い、ワーカ側は個体の係数最適化と適合度の計算を行う。サブマスタは定期的にジョブを取得することで、ワーカの数の 2 倍の個数のジョブを保持するようにし、そのジョブをワーカに配分する。そしてワーカからジョブの結果を取得すると、結果をマスタに返すのと同時にマスタから再度ジョブを取得する。

本アプリケーションを実行することで Jojo2 のスケーラビリティと耐故障性の評価を行った。実験には本研究室の PrestoIII クラスタを用いた。

6.2.1　スケーラビリティの測定

以下の 3 次の問題を設定した。

$$\begin{cases} dX_0/dt = -1.2X_0 - 0.6X_1 + 0.2 \\ dX_1/dt = 0.5X_0X_1 - 1.6X_1 + 1.2 \\ dX_2/dt = 2.0X_0 + 1.7X_1 - X_2 \end{cases} \quad (1)$$

$$X_0(0) = 3 \quad X_1(0) = 2 \quad X_2(0) = 1$$

またワーカで行う RungeKutta 法の刻み幅を 1/4, 1/8, 1/16 として並列化効率を測定した。刻み幅が小さければ小さいほどワーカ上での処理時間が長くなる。

実験結果を図 9 に示す。図中の RK は RungeKutta 法の刻み幅の値、 SB は配置したサブマスタの数を示す。また表 2 に各刻み幅のワーカ上での平均処理時間を示す。

図 9 から刻み幅が小さいときほど、並列化効率が良いことがわかる。これは表 2 で示すように刻み幅が小さくなるにつれ、ジョブの平均実行時間の増加し、相対的にマスタ、サブマスタへの負荷が軽減されたからだと考えられる。さらにサブマスタが 2 台のときの方が並列化効率が良い。これはサブマスタの台数が増え

表 3　正常実行時と故障ノード発生時の処理時間の比較

	(a)	(b)	(c)
平均実行時間 [sec]	682	708	839

ることで接続されるワーカの数が減り、一つのサブマスタあたりの負荷が軽減されたからだと考えられる。

6.2.2　耐故障性の検証

Jojo2 の耐故障性を検証するために、それぞれ次のような外乱を与えながら遺伝子ネットワーク推定アプリケーションを実行した。

(a)　いずれのプロセスも一度も停止せずに実行する。
(b)　16 台のワーカプロセスを停止し、10 秒後に再びその計算ノード上でワーカプロセスを起動する。これを 3 分おきに計 3 回行う。
(c)　プログラムを実行してから 3 分後に 16 台のワーカプロセスを停止し、その後は 48 台のワーカで計算を行う。

RungeKutta の刻み幅は 1/16 とし式 (1) を解く。サブマスタは 1 台とし、初期のワーカ数は 64 ノードとして起動した。それぞれ 3 回ずつ実行し、それぞれの平均実行時間を表 3 に示す。

表 3 の (c) に比べ (b) の実行時間が短縮されていることから、途中で脱退、参加したノードに対して再びジョブが割り当てられていることがわかる。また (b) でワーカプロセスを起動してから実際にジョブが割り当てられるまでの平均時間を測定すると約 22.7 秒だった。プロセスを停止してから 10 秒後に再びプロセスを起動しており、これを 3 回繰り返しているため 16 台のワーカプロセスがそれぞれ $(22.7+10)\times 3 = 98.1$ 秒間計算していなかったことになる。これは全体のワーカプロセスのうちの 4 分の 1 にあたるので、ジョブ割り当てまでのオーバヘッドがない場合の実行時間を推測すると $709 - 98.1 \div 4 = 684$ 秒と求まる。これは正常実行時の (a) とほぼ同じ値となり、このことからも Jojo2 の動的なノードの参加・脱退が有効に機能していることがわかる。

7. 関 連 研 究

Ninf-G[8]/Ninf-1[9] による階層的マスタ・ワーカ方式のプログラミングが提案されている[2)3)]。これらはクライアント・サーバモデルを多段的に適用することにより、グリッドに適合した階層構造に対応している。具体的には、インターネット上でのを通じて行われるマスタ・サブマスタ間の通信はセキュリティの確保のために Ninf-G[8] を利用し、クラスタ内で行われるサブマスタ・ワーカ間の通信には暗号化などを備えない代わりに非常に高速な通信が実現可能な Ninf-1 を利用している。ProActive[10] は Java で実装され、リモートへのオブジェクトの配置や、リモート間でのオブジェクトの移動が容易なプログラミング環境である。また特定のクラスやインタフェースを継承する必要のない柔軟なプログラミングモデルを提供する。ProActive はグループオブジェクトにより、グリッドの階層構造に合致したプログラミングが可能である[11]。

これらは耐故障性は備えているものの、そのトポロジは実行前に静的に決定され、動的なノードの増減を考慮したシステム構成やプログラミング API は提供していない。また Ninf の場合、複数のプログラミングツールを組み合わせる必要がある上に、Ninf IDL と呼ばれる言語を用いて IDL を記述しなくてはならないため、プログラマへの負担が大きい。

動的なノード群構成や動的なノード参加・脱退が可能な技術として P2P が広く利用されつつある。P2P 分散処理ミドルウェアとして P3[12] が提案されている。 P3 は JXTA を用いて多数の PC を集約し、マスタ・ワーカ方式によるプログラミングライブラリも提供している。JXTA を利用することで事前設定不要なピアの組織化、発見が可能である。しかしグリッドの階層性は考慮しておらず、必要以上に通信遅延の大きいインターネット上の通信が頻出してしまう可能性がある。Jojo2 ではノード数が多く手動での管理コストが非常に大きい末端ノードに対しては、P2P 指向のプロトコルを用いて自律的なノード発見を行っており、また各レイヤで動的なノードの参加・脱退が可能である。さらにグリッド環境の階層構造に適したトポロジ、プログラミングライブラリを提供することで、マスタへの負荷集中、インターネット上の通信によるボトルネックの回避を図っている。

DNAS[13] は階層構造の通信トポロジを持ち、環境の変化に応じて動的にトポロジの構成を変化させることができる。DNAS はデーモンプログラムとして動き、ロードアベレージなどシステムの情報を取得可能で、システムの状況を考慮したアプリケーション開発を可能にする。これに対し Jojo2 では各サイトのジョブスケジューラを利用することで各サイトの利用状況、サイトポリシーに従ったジョブ実行ができ、より汎用的だと考えられる。また DNAS では動的構成に適したプログラミング API についての考察がないが、Jojo2 では動的なノードの参加・脱退に対するハンドラ、動的構成に適したメッセージパッシングライブラリを提供する。

Phoenix[14] は動的な資源の増減に対応したプログラミング環境である。Phoenix ではプロセスを仮想ノード番号と呼ばれる比較的大きな識別子集合に重複なくマップさせ、仮想ノード番号を用いてノードの間の送受信を行う。仮想ノード番号はノード数に非依存なのでマッピングを変更するだけで、容易にノードの増減に対応することが可能である。Phoenix は MPI ライクなメッセージパッシングライブラリを提供しており、マスタ・ワーカ方式に特化した Jojo2 とはプログラミングモデルが異なる。

8. おわりに

本稿では、動的なノード群構成機構を備えたプログラミング環境 Jojo2 の設計と実装について述べた。Jojo2 はグリッドの階層構造に適合した動的なノード群構成を行うことで、自律的なノード、動的なノードの参加・脱退を可能にし、また動的構成に適したプログラミング API を提供する。

さらに遺伝的プログラミングによる遺伝子ネットワーク推定アプリケーションを実装し、予備的な性能評価としてスケーラビリティの測定と、耐故障性の検証を行った。

今後の課題としては以下が挙げられる。

- より現実的なシナリオでの評価実験を行う必要がある。我々は 800CPU 規模のグリッド環境で Jojo2 の稼動実験を行ったが[15]、より長時間に渡っての実験や、そのような環境でのスケーラビリティなどの測定を行う必要がある。
- 我々は遺伝的アルゴリズムや分枝限定法などの組み合わ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として jPoP[16] を Jojo を用いて実装している。jPoP を Jojo2 を用いて再設計・再実装を行うことで、様々なアプリケーションに対して Jojo2 の有用性とスケーラビリティを確認する。

参考文献

1) Hidemoto Nakada, Satoshi Matsuoka, and Satoshi Sekiguchi. A java-based programming environment for hierarchical grid: Jojo. Proceedings of CCGrid 2004 (2004).
2) Kento Aida, Tomotaka Osumi. A Case Study in Running a Parallel Branch and Bound Application on the Grid. Proceedings of IEEE/IPSJ The 2005 Symposium on Applications & the Internet (SAINT2005), pp.164-173 (2005).
3) 小野 功, 水口 尚亮, 中島 直敏, 小野 典彦, 中田 秀基, 松岡 聡, 関口 智嗣, 楯 真. Ninf-1/Ninf-G を用いた NMR 蛋白質立体構造決定のための遺伝アルゴリズムのグリッド化. 先進的計算基盤システムシンポジウム SACSIS 2005, Vol, No. 5, pp.143-151 (2005).
4) Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A Hunter of Idle Workstations, Proceedings of the 8th International Conference of Distributed Computing Systems, pp.104-111 (1988).
5) 町田 悠哉, 中田 秀基, 松岡 聡. ポータビリティの高いジョブスケジューリングシステムの設計と実装. 情報処理学会研究報告 2004-HPC-99 (SWoPP 2004), pp.217-222 (2004).
6) 徳田 拓, 田中 康司, 中田 秀基, 松岡 聡. Jojo による遺伝的プログラミングの並列化. 情報処理学会研究報告, 2004-ARC-157 2004-HPC-97 (HOKKE-2004), pp.187-192 (2004).
7) 田中 康司, 中田 秀基, 岡本 正宏, 松岡 聡. 遺伝子ネットワーク推定のための並列 GP アルゴリズムの評価. 情報処理学会シンポジウムシリーズ 数理モデル化と問題解決シンポジウム論文集, 2004(12), pp.171-178 (2004).
8) Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada and Satoshi Sekiguchi. Design, implementation and performance evaluation of GridRPC programming middleware for a largescale computational Grid. Proceeding of 5th IEEE/ACM International Workshop on Grid Computing (2004).
9) 中田秀基, 高木浩光, 松岡 聡, 長嶋 雲兵, 佐藤 三久, 関口 智嗣. Ninf による広域分散並列計算. 情報処理学会論文誌 vol.39, no.6, pp. 1818-1826 (1998).
10) Francoise Baude, Denis Caromel, Fabrice Huet, Lionel Mestre and Julien Vayssiere. Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), pp.93-102 (2002).
11) Laurent Baduel, Francoise Baude, Denis Caromel. Efficient, Flexible and Typed Group Communications for Java. Joint ACM Java Grande - ISCOPE 2002 Conference (2002).
12) Kazuyuki Shudo, Yoshio Tanaka and Satoshi Sekiguchi. P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources. CCGrid 2005 Fifth Int'l Workshop on Global and Peer-to-Peer Computing (2005).
13) 折戸 俊彦, 廣安 知之, 三木 光範. 階層型グリッドミドルウェア DNAS の設計と実装. ハイパフォーマンスコンピューティングと計算科学シンポジウム HPCS 2006, pp.1-8 (2006).
14) Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix : a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003).
15) 青木 仁志, 中田 秀基, 田中 康司, 松岡 聡. 大規模グリッド環境における Jojo2 の稼動実験. 先進的計算基盤システムシンポジウム SACSIS 2006 ポスタ (2006).
16) 中川 伸吾, 中田 秀基, 松岡 聡. 並列組合せ最適化システム jPoP の分枝限定法の実装. コンピュータシステム・シンポジウム論文集, pp.85-92, 2004.