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

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、Globusを用いた起動、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。さらに予備的な性能評価の結果も示す。

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.

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 result.

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とよ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難しい。また、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、負荷が大きい。

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、Globusを用いた起動、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し、予備的性能評価結果も示す。

本稿の構成は次のとおりである。2では、Jojoの対象とするグリッド環境について述べる。3でJojoの設計について述べ、4で実装について述べる。5でJojoを用いた簡単なプログラムを示す。6で予備的評価とその結果を示す。7で結論と今後の課題を述べる。

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。とく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のIPアドレスはセキュリティやアドレス空間枯渇の問題から、ローカルアドレスとなり、NATを用いて外部と通信する場合が多い(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

† 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

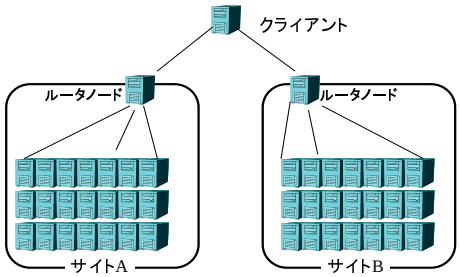

図 1 複数クラスタから構成されるグリッド

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 [5] が、現時点では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。

これらの理由のため、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得ない。すなわち、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のネットワークの 2 階層である。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・ワーカであっても、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。

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、前節での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 を使用する通信ライブラリとしては Sun 自身による RMI[6] や、Java による PVM 実装である JPVM[7]、C による MPI 実装を呼び出すラッパである mpiJava[8]、などが提案されている。pure Java による MPI 実装も行われている [9]。

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。

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。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を直感的に理解しやすい形で定義をすることは難しい。

Jojo は、この両者の中間とも言うべきプログラミングモデルを提供する。Jojo では各ノードにひと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。マルチスレッドを前提として、メッセージの受信を別スレッドで行うセマンティクスとなっているため、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える。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、典型的にはひと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

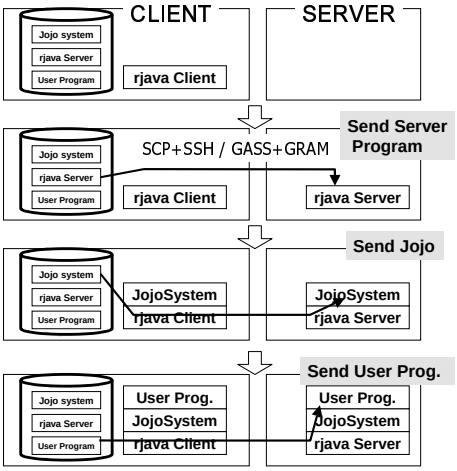

図 2 rjava による起動

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 rjava[10] を用いた。Jojo のシステム起動は次のように行われる (図図 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

(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) Jojo 上のユーザプログラムも同様に、ローカルファイルシステムから読み出されてロードされる。

(5) 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。さらにユーザプログラムもロードして起動する。

```
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

現在起動方法としては、ssh や rsh を用いるバージョンと、Globus の GRAM と GASS を用いて Globus I/O を使用するバージョンが用意されている。

ssh や rsh を用いるバージョンでは、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 を用いるバージョンでは同じ方法は使用できない。GRAM/GASS 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、Jojo 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。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る。

多段接続となる場合には、基本的にこ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常にユーザの手元となるよう、クラスファイル要求がローカルプログラムに委譲される。

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

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

図 4 Context インターフェイス

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。

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。

`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。

`Future callFuture(Message msg)`
: 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` 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、返信オブジェクトが返される。

`void callWithContext(Message msg, Context context)`
: 第 2 引数として `Context` インターフェイ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` インターフェイスの `run` メソッドが呼ばれる。Context インターフェイスは図 4 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あらわす 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。

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

図 5 設定ファイルの DTD

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 5 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 (図 8)。

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 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ロードバランスをとっている。ワーカ側がからマスタ側にジョブを要求しており、マスタ側はジョブの要求に対してジョブを分配している。ジョブの要求と結果を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。

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用し、ssh で実行するには図 8 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

```
public class PiMaster2 extends Code{
 long times, perNode;
 int  divide;
 boolean done = false;
 long doneTrial = 0, doneResult = 0;

 public void init(Map args)
 throws JojoException{
  System.err.println(args);
  Properties prop =
     (Properties)args.get("properties");
  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){
    synchronized (this)
     {done = true; notifyAll();}
    return new Long(0);
   } else
    return new Long(perNode);
  } else
   throw new JojoException(
    "cannot handle the message: " + msg);
 }
}
```

図 6 マスタープログラム

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、GSI を用いた場合と SSH を用いた場合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 9 に示す環境を用いた。以下 TITECH-AIST 間の実験を WAN とし、AIST 内の実験を LAN とする。

図 11 に LAN 環境での結果を示す。SSH を用いた場合に対して GSI を用いたバージョンの性能が大きく低下していることがわかる。また双方ともバンド幅に対して 20 分の 1 程度しか性能がでていない。

```
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 7 ワーカープログラム

```
<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 8 サンプルプログラム用設定ファイル

図 11 に WAN 環境での実験結果を示す。それほどの差はないものの、LAN 環境とは逆に GSI のほうがよい性能を示している。また、ネットワークバンド幅に対する性能低下の割合は、LAN 環境に比べれば小さい。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。

いずれの場合も、本来のスループット値と比較するとかなり性能が低下している。この原因としては、SSH や GSI による暗号化のオーバヘッドとともに、一つの通信ストリームを標準入出力のリダイレクトと Jojo

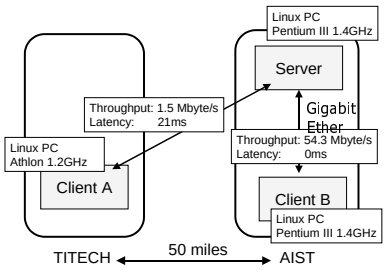

図 9 評価環境

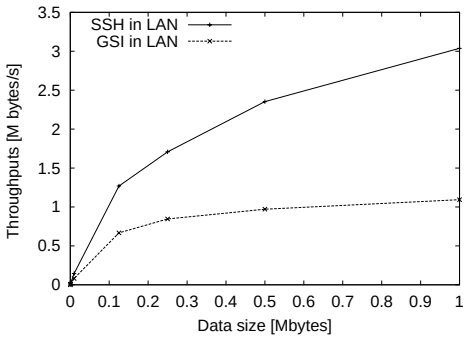

図 10 LAN 環境でのスループット

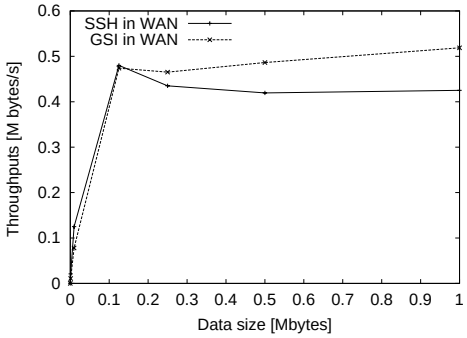

図 11 WAN 環境でのスループット

の通信ストリームにマルチプレクスしているコストが考えられる。

7. 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、予備的な性能評価の結果を示した。

今後の課題としては以下があげられる。

- ストリームマルチプレクス部を再実装し、オーバヘッドの低減を図る。
- Jojo を用いた組み合わせ最適化問題用システムである jPoP[11] 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 へ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だが、今後実装していく。

参 考 文 献

1) Foster, I. and Kesselman, C.: Globus: A metacomputing infrastructure toolkit., *Proc. of 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, *Proceedings of 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. of the IEEE/ACM SC2000 Conference* (2000).

5) 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).

6) 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).

7) Ferrari, A. J.: JPVM: network parallel computing in Java, *ACM 1998 Workshop on Java for High-Performance Network Computing* (1998).

8) Baker, M., Carpenter, B., Fox, G., Ko, S. H. and S.Lim: mpiJava: An Object-Oriented Java interface to MPI, *International Workshop on Java for Parallel and Distributed Computing* (1999).

9) 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).

10) 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).

11) 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-148 (2002).