オーバレイスケジューラ Jojo3 の提案

中田秀基[†] 田中良夫[†] 関口智嗣[†]

グリッド技術およびクラスタ技術の発展によって，個々のユーザがアクセスできる計算機の総量は爆発的に増大した．しかしこれらの計算機を同時に利用するアプリケーションプログラムを記述することは困難であり，膨大な計算資源を有効に活用できているアプリケーションは限定されている．アプリケーションプログラムの記述が困難な理由の一つは，環境の不均質性および非対称性にある．計算機資源は，それぞれ異なるバッチスケジューリングシステムおよびグリッドミドルウェアで管理されており不均質ネットワークの接続性には非対称性があり，ノード間通信も必ず可能なわけではない．本稿では，この不均質性と非対称性をアプリケーションプログラマから隠蔽するオーバレイスケジューラを提案し，その 1 実装である Jojo3 の設計と実装について述べる．オーバレイスケジューラは，グリッドミドルウェアおよびバッチスケジューリングシステムで起動され，均質なインターフェイスをアプリケーションプログラムに対して提供する．アプリケーションプログラマは，オーバレイスケジューラを利用することで，比較的容易にアプリケーションを記述することができる．

The Proposal of an Overlay Scheduler Jojo3

Hidemoto Nakada ,[†] Yoshio Tanaka [†] and Satoshi Sekiguchi [†]

Although recent improvements in the grid and cluster area enabled individual users to access huge amount of computational resources,it is still difficult for them to write their distributed applications that fully utilize the resources . One of the reasons that makes difficult to write distributed application is that the heterogeneity and asymmetry in the Grid environment, where each cluster is managed by different grid middle-ware and local batch scheduling system and configured with private-addressed local network which disables node-to-node direct connection. In this paper we propose a concept called **overlay scheduler** that hides heterogeneity and asymmetry from the application programmers, and an implementation of the overlay scheduler, called Jojo3. Overlay scheduler is invoked via the grid middle-wares and batch queuing systems, and provides the homogeneous view to the application programmers, enabling relatively easy implementation of distributed parallel applications.

1. はじめに

グリッド技術およびクラスタ技術の発展によって，個々のユーザがアクセスできる計算機の総量は爆発的に増大した．典型的なグリッド環境は，各サイトにおかれたローカルスケジューラと，それをつなぎ合わせるグリッドミドルウェアで構成される．これらの機構を利用することによって，大量の計算資源を利用したジョブの実行が可能になっている．

しかし，実際にこのような環境で動作する並列アプリケーションを記述するのは非常に困難で，膨大な計算資源を有効に活用できているアプリケーションは限定されている．特に並列計算の専門家ではない一般のアプリケーションプログラマには並列アプリケーションを記述することは非常に難しい．この記述を補助するために，特定のアプリケーションの通信スタイルに特化したアプリケーションレベルスケジューラを提供することが考えられる．アプリケーションレベルスケジューラはアプリケーションの通信パターンに特化することから，個々のアプリケーションに対して個別に実装する必要がある．しかし，アプリケーションレベルスケジューラを，不均質で非対称なグリッド環境にグリッドミドルウェアとローカルスケジューラを利用して記述することは，けして容易ではない．複数のアプリケーションレベルスケジューラが，個別に不均質性，非対称性に対応することは無駄である．

我々は，グリッドミドルウェアとローカルスケジューラから構成されるベースレベルスケジューラと，アプリケーションレベルスケジューラとの間を埋めるものとして，オーバレイスケジューラのレイヤを導入することを提案する．オーバレイスケジューラは，ベースレベルスケジューラを用いたジョブの起動と管理を行なう．参加ノードの増減などの環境に生じた変化をアプリケーションレベルスケジューラに通知する．さらに，簡単なノード間通信機構の機能を提供する．

† 産業技術総合研究所 / National Institute of Advanced Industrial Science and Technology (AIST)

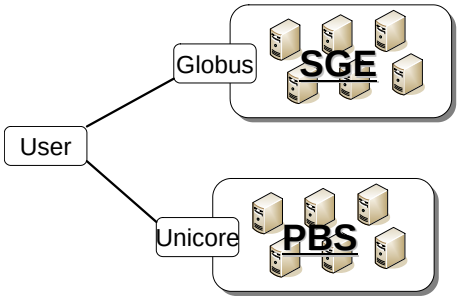

図 1　一般的なグリッドの構成

オーバレイスケジューラを導入することで，ベースレベルスケジューラの持つ不均質性，非対称性が隠蔽され，アプリケーションレベルスケジューラの記述が容易になることが期待される．

本稿ではオーバレイスケジューラに期待される機能を整理し，そのような機能を持つプロトタイプオーバレイスケジューラ Jojo3 を提案する．さらに，Jojo3 を利用したアプリケーションレベルスケジューラとして，JDK1.5 で導入されたの ExecutorService インターフェイスを実装したマスタ・ワーカ計算のフレームワークを示す．

Jojo3 は Java で記述されているが，上位レイヤとのインターフェイスは環境変数とソケット通信であるため，上位のオーバレイスケジューラの実装言語を限定しない．

2. オーバレイスケジューラの必要性

複数のクラスタから構成される典型的なグリッドを図 1 に示す．グリッドは一般に複数のクラスタから構成される．個々のクラスタは多くの場合ローカルなスケジューラとなるキューイングシステム (Grid Engine[1], TORQUE[2] など) で管理される．多くの場合，クラスタはプライベートアドレスしか持たず，限られた数のゲイトウェイを通じて外部ネットワークに接続されている．これらのクラスタは，グリッドミドルウェアと呼ばれる機構を用いて外部からの接続を受け付ける．グリッドミドルウェアは，ユーザからのジョブサブミッションを受け付け，適切なユーザの認証と権限の付与を行い，ローカルスケジューラを用いてジョブの実行を行う．このようなグリッドミドルウェアには，[3]Globus や[4]UNICORE が挙げられる．本稿では以降，ローカルスケジューラとグリッドミドルウェアを総称してベースレベルスケジューラと呼ぶ．

このように構成されたグリッド環境では，ユーザは比較的容易に個々算資源上でジョブを実行することができる．しかし，この環境を活用する分散並列アプリケーションを記述することはそれほど容易ではない．これは，下記の理由による．

- さまざまなグリッドミドルウェアに対応しなければならない．グリッドミドルウェアが，背後のローカルスケジューラを隠蔽するため，個々のローカルスケジューラの差異を考慮する必要はないが，グリッドミドルウェアの相違はアプリケーション側で考慮しなければならない．
- ジョブが実際に起動するタイミングが予測しにくい．キューイングシステムをバックエンドに利用している場合，ジョブはキューに投入される．一般にキューの待ち時間を予想することは難しい．また，キューにジョブが全くない場合でも，グリッドミドルウェアおよびキューイングシステムのオーバヘッドによって，ジョブの投入から実行までに数秒-数十秒のタイムラグが存在する．
- ジョブ間の通信路が提供されない．分散並列プログラムを構築するためには，ジョブ間になんらかの通信路網を確立する必要があるが，多くのキューイングシステムでは通信路網をサポートしていない．標準入力，標準出力，標準エラーの転送がサポートされている場合はあるが，リアルタイムではなく，事前，事後のファイルステージングであるため，通信に利用することはできない．

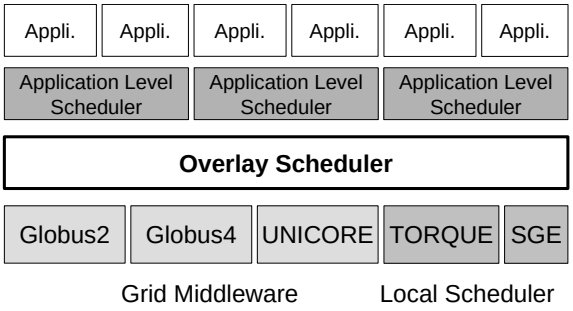

図 2　スケジューラのレイヤ構造

筆者らは，このような環境でも特定のパターンをもつ分散並列プログラムを比較的容易に記述することのできるアプリケーションレベルスケジューラとして，GridRPC システム Ninf-G[5] を提案，実装してきた．Ninf-G は，さまざまなグリッドミドルウェアに対応する機構を独自に実装している[6]．

しかし，この機能は Ninf-G に内蔵されてしまっているため，他のアプリケーションレベルスケジューラを実装する際には，新たにこの機能を実装し直さなければならない．これは非常に無駄であると同時に煩雑である．

この問題を解決するためには，アプリケーションレベルスケジューラが共通して必要とする機能を抽出して，あらたにミドルウェアのレイヤとして実装し，その機能を用いて，アプリケーションレベルスケジューラを実装すればよい．このレイヤを本稿ではオーバレイスケジューラと呼ぶ (図 2)．オーバレイスケジューラは，ベースレベルスケジューラの不均質性非対称性

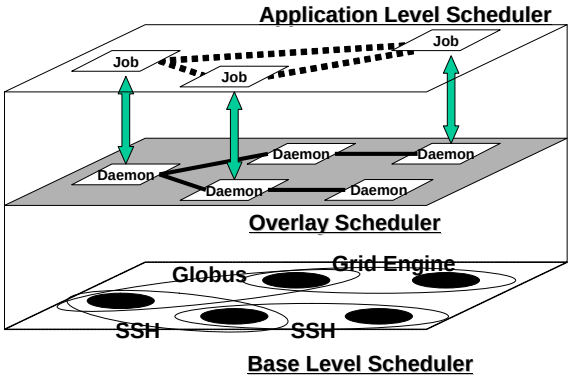

図 3　Jojo3 の概要

を隠蔽し，均質な資源を上位レイヤに提供する．このレイヤを利用することで，アプリケーションレベルスケジューラの記述は非常に容易になる．

3. Jojo3 の設計

3.1 要　　請

オーバレイスケジューラに要求される機能は以下のとおりである．

- 使用可能なノード群の管理
- ジョブの起動と管理
- ジョブ間通信の提供

3.2 アーキテクチャ

Jojo3 は，各ノード上で起動するデーモンのネットワークとして構成される．ユーザは，ルートとなるデーモンを手元のノードで起動する．起動したデーモンは，設定ファイルを参照して使用可能なクラスタを取得し，そこに再帰的にデーモンを起動していく．デーモン間の親子関係に沿ってネットワークが構築される．各デーモンはノードの状況を監視し，ブロードキャストする．

ユーザの分散並列アプリケーションは，いずれかのデーモンを指定して，単一のジョブを実行することで起動する．ジョブは，Jojo3 の機能を利用して，利用可能なノードを発見し，そこで子ジョブを起動する．ジョブとそこから起動された子ジョブ群は一つのジョブグループを構成する．Jojo3 は任意のジョブ間に対して通信機能を提供する．ジョブと子ジョブ群はこの機能を利用して通信路を構成し，相互に通信してアプリケーションの実行を進める．この様子を図 3 に示す．

3.3 キューイングシステムの取り扱い

キューイングシステムで管理されたノードは，潜在的な資源として管理される．サブミットノード上のデーモンがキューイングシステムに問い合わせた結果を他のノードに報告する．このノードを資源として利用したい場合には，まず，サブミットノード上の管理デーモンに対して，ノードの活性化要求を発行する．管理デーモンはキューイングシステムを用いて，各ノード上で Jojo3 のデーモンを起動する．これで，各ノードは Jojo3 の資源の一部として実際に利用可能な状態となり，他のデーモンからのジョブ起動要求を受け付けることができる．

キューイングシステムからみれば，Jojo3 のデーモンがジョブであるが，実際には，そのデーモン上でさらに Jojo3 のジョブが動くことになる．この際 Jojo3 としてのジョブの数は 1 つであるとは限らないことに注意されたい．キューイングシステム上で起動した Jojo3 デーモンは，資源の不必要な占有を防ぐために，一定時間以上利用するジョブがなければ自動的に終了する．もちろん，ジョブが実行中であっても，事前に定められた Walltime を経過すればキューイングシステムによって終了させられる．

3.4 Jojo3 プログラムの実行

Jojo3 を用いた分散プログラムは，プログラム全体の起動のルートとなるプログラムと，そこから起動されるワーカプログラムに大別される．ユーザは，プログラムの実行に先立って，Jojo3 を起動しておく．ルートプログラムは，ユーザによって起動され，ローカルホスト上の Jojo3 デーモンにアクセスし，利用可能なノード情報を取得する．利用可能なノードの中から適切なものを選んで，それらのノード上にワーカプログラムを起動する．ワーカプログラムは，起動するとやはりローカルホスト上の Jojo3 デーモンにアクセスし，ノード情報を取得する．この時点で，ワーカプログラムの起動がルートプログラムにも告知されるので，ルートプログラムとワーカプログラムは相互に認識する．あとは，どちらかからネットワーク接続を依頼し，通信を行う．

4. 実　　装

4.1 構　　成

Jojo3 はデーモンとクライアントライブラリから構成される．クライアントライブラリは，現在は Java 言語による実装のみが存在するが，他の言語でも比較的容易に実装が可能であるように考慮されている．

4.2 デーモンの実装

Jojo3 デーモンは，Java 言語で記述した．これは，ほぼすべての計算機アーキテクチャで利用可能であり，バイトコードによるバイナリレベルでのポータビリティがあるからである．また，ポータビリティを考慮し JDK1.4 相当の機能のみを用いて記述した．これは周囲のクラスタを調査したところ JDK 1.4 相当の Java がデフォルトでインストールされているものが散見されたためである．

デーモンは，情報を集中的に管理するコアモジュールの他に，デーモン間通信を司るルータモジュール，クライアントライブラリとの通信を司るサーバモジュー

ルから構成される．ルータは，ルーティングテーブルを管理し，メッセージの配送を行う．

4.3 デーモンの起動

デーモンの起動には，Jojo[7] で用いた rjava と呼ばれる機構を利用した．rjava はブートストラップとなるクラスローダを含んだ小規模な jar を送信，起動し，本体プログラムはネットワーク越しにオンデマンドで送信する機構である．この機構を用いることで，あらかじめプログラムを対象ノードにインストールすることなく，デーモンを起動することができる．

4.4 デーモン間通信

rjava はクライアントとサーバの間のソケット接続をパケットベースで分割し，クラスファイル転送や，標準入出力のリダイレクトなど，さまざまな目的に利用しているが，上位のプログラムにもこのパケット通信路を提供している．デーモン間の通信では，rjava の提供するパケット通信路を利用して，メッセージオブジェクトをシリアライズしたものをパケットとして送受信する．後述するクライアントプロトコルと異なり，言語中立性を考慮する必要がないためである．

デーモン間通信は，すべてのノードに対するブロードキャスト通信と，特定のノードに対するユニキャスト通信に大別される．ジョブの状態変化 (起動，終了など) や，ノードの状態変化 (参加，離脱，負荷の大きな変化) などの情報はブロードキャスト通信ですべての参加ノードに配信される．ジョブの起動要求や通信要求などはユニキャスト通信となる．ユニキャスト通信のパケットは途中経由するノードではデコードされることなく目的地まで配送される．

パケットの配送にはルーティングテーブルが必要である．ルーティングテーブルは，送受信されるブロードキャストパケットから自動的に学習される．

4.5 ジョブ間通信

ジョブ間の通信はデーモンを介して行われる．ジョブとデーモンの間は通常のソケット通信となる．デーモン間は上述したデーモン間通信のパケット配送で行われる．したがってジョブ間通信の本数が増えても，デーモン間のソケットの数が増えることはない．

このジョブ間通信は，デーモンを介するためオーバヘッドが大きいうえ，経路が最適でないことが予想される．たとえば，クラスタ内の隣接したノードであっても，親となるデーモンを介した通信になる図 4．アプリケーションが，より高速で最適化された通信を必要とする場合には，ジョブ間通信を用いてポートやアドレスを交換し，アプリケーションレベルで直接通信を行うことが考えられる．

4.6 デーモンとクライアント

デーモンは，クライアントからの通信を受け付けるために，ローカルホストに対する TCP 接続をオープンする．ここで Unix ドメインソケットを用いないのは，実装言語である Java でサポートされていないからである．ルートとなるデーモンのポートは固定ポートとしている．ワーカとなるデーモンのポート番号は動的に決定され，環境変数で起動されたジョブに通知される．

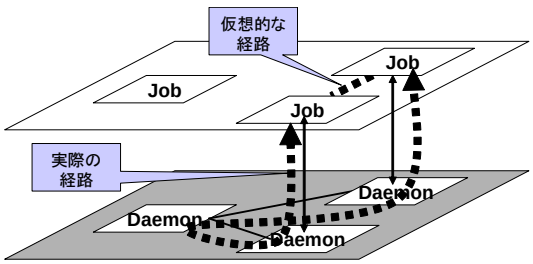

図 4　ジョブ間の通信路

クライアントライブラリは，自動的に環境変数を認識し，ローカルノード上のデーモンにアクセスする．

4.7 クライアントプロトコル

クライアントライブラリとデーモン間の通信には，以下の要件がある．

- 言語中立であること
- 非同期であること

Condor の GAHP プロトコル[8] に類似したテキストベースのプロトコルを採用した．このプロトコルでは，テキスト 1 行単位で通信を行う．構造体の授受には，JSON(JavaScript Object Notation)[9] 形式を用いた．テキストベースの構造体表現形式としては XML 形式が考えられるが，JSON は構造体から変換する際に恣意性がないためスキーマの定義が不要なため利用が容易であることを重視し，JSON を採用した．Java Beans からの変換ライブラリには JSON-lib[10] を用いている．

4.8 Java クライアントライブラリ

Java による Jojo3 クライアントライブラリのインターフェイスを図 5 に示す．インターフェイスのメインクラスは，Client クラスである．Client クラスは非同期に設計されている．すなわち，すべてのメソッドの呼び出しはブロックせずにリターンする．結果は初期化時にユーザが提供する UpdateListener interface を介して返却される．たとえば，他のジョブに対するネットワーク接続を依頼する connect メソッドは，connection の識別子を返す．接続要求が成功した場合には，UpdateListener の connected メソッドがその connection 識別子を引数に呼び出される．失敗した場合には，connectionFailed が呼び出される．このように非同期にすることによって，複数のノードに対する接続要求を同時に発行することが可能になる．

4.9 アプリケーションレベルスケジューラ記述例

Jojo3 を利用したアプリケーションレベルスケジューラの例として Java の ExecutorService を分散化した．ExecutorService は JDK1.5 で標準ライブラリの一部となったもので，実行可能なオブジェクト (Runnable

```
class Client {

  // コンストラクタ
  Client(UpdateListener listener);

  // 自分のジョブ識別子取得
  String getJobId();

  // 自分の所属するジョブグループの識別子取得
  String getJobGroupId();

  // ジョブグループの起点ジョブの識別子取得
  String getRootJobId();

  // ノード群の情報取得
  NodeInfo [] getNodeInfo();

  // 自分の属するジョブグループの他のジョブの状態取得
  JobState [] getJobState();

  // ジョブの接続要求
  String invoke(String nodeId, JobDescription job);

  // 他のノードへの接続要求
  String connect(String jobId);
}

interface UpdateListener {

  // 接続成功
  void connected(String conId, Connection con);

  // 接続失敗
  void connectionFailed(String conId);

  // ジョブグループに属するジョブの状態変化
  void jobStateChanged(String     jobId,
                       JobState   jobState);

  // ノードの状態変化
  void nodeStateChanged(NodeInfo  nodeInfo,
                        NodeState nodeState);
}
```

図 5 Java クライアントライブラリ API

および Callable) をキューイングしておき，順次実行するものである．ライブラリに含まれているものは単一プロセス内で，複数のスレッドを用いて実行するものであるが，これを分散化することで，複数の計算資源を用いた並列実行を容易に行うことができる．

我々はまず，通常のソケット通信を直接行う形で ExecutorService を分散化した．分散化した ExecutorService は，マスターモジュールとワーカモジュールから構成され，シリアライズされたオブジェクト通信を行う．さらにこれを Jojo3 を用いるよう拡張した．Jojo3 対応には，マスターモジュールで約 100 行，ワーカモジュールで約 50 行の記述が必要であった．比較的小規模の記述でアプリケーションレベルスケジューラを Jojo3 対応にすることができることを確認できた．

5. 関連研究

5.1 Condor Glide-In

Condor[11)12)] はウィスコンシン大学で開発されたジョブスケジューリングシステムである．Condor は，Grid Engine などと同様にベースレベルスケジューラであると考えることもできるが，Glide-In と呼ばれる機能を用いることでオーバレイスケジューラとして利用することもできる．

Glide-In は，他のジョブサブミッション機構が管理する計算機群を一時的に「のっとり」Condor の管理下で運用するための機構である．Condor は，セントラルマネージャ，サブミットノード，実行ノードの3 種類のノードから構成される．Glide-In は，実行ノードの機能を司るデーモン群を，他のジョブサブミッション機構を利用して対象ノードで起動する．起動したデーモンは，あらかじめ設定されたセントラルマネージャに接続し，Condor プールの一部を構成する．Glide-In を利用することで，たとえば Grid Engine や TORQUE などの他のジョブサブミッションシステムで管理され，Globus GRAM[13)] などのグリッドミドルウェアによる外部インターフェイスを持つクラスタを，Condor プールの一部として利用することができる．

また，Condor はジョブ実行ノードとサブミットノードの間で，Chirp[14)] と呼ばれる通信路を提供する．この通信路を利用することで，アプリケーションレベルスケジューラを実装することが可能で，マスターワーカ型のジョブ実行に最適化された Condor MW[15)16)] が提供されている．MW は，C++で記述されたクラステンプレートで，テンプレートクラスを継承したクラスを記述することで，容易にマスターワーカ型のプログラムを記述することができる．

5.2 GXP

GXP[17)] は東京大学で開発されたグリッド環境でのシェルで，一種のオーバレイスケジューラである．GXP は Python で記述されており，起動ノードから主に ssh を用いて他のノードに GXP 本体を順次に送り込み起動していくことで，オーバレイスケジューラ網を構成する．構成したスケジューラ網上で，ジョブの起動および, ジョブの入出力を起動ノードに対してリダイレクトすることができる．さらにジョブ間のブロードキャストもサポートしている．また，Sun Grid Engine などのキューイングシステムを利用したジョブの投入にも対応している．

ただし，ブロードキャストの機能は，Unix ファイルディスクリプタの番号を固定して提供されているため，任意のファイルディスクリプタ番号に対して入出力を行うことのできない言語 (Java など) では，この機能を利用することができない．

5.3 Jojo と Jojo2

Jojo[7)] および Jojo2[18)] は，筆者らが過去に提案した一種のオーバレイスケジューラである．SSH や Globus GRAM を利用して階層的なスケジューラ網を構築する．Jojo3 との本質的な相違は，アプリケーションレベルスケジューラと分離されておらず，アプリケーションレベルスケジューラを直接 Jojo および Jojo2 で記述する構成を取っていることである．このため，アプリケーションレベルスケジューラを記述できる言語が，

Jojo および Jojo2 を記述した言語 (Java) と同一でなければならないという制約が存在する．このため，C や C++，Fortran などの言語で記述された高速なライブラリをアプリケーションの一部とすることが難しい．

6. おわりに

本稿では，グリッド環境で必要となるスケジューラをベースレベルスケジューラ，オーバレイスケジューラ，アプリケーションレベルスケジューラの 3 階層に整理し，オーバレイスケジューラの機能としてジョブの実行と通信機構の提供と規定した．さらにそのようなオーバレイスケジューラの Java による 1 実装として Jojo3 を提案し，その設計と実装について述べた．

今後の課題としては以下が挙げられる．

- 実装の完了
  Jojo3 は現在実装中であり，様々な機能が未実装である．とくに様々なベースレベルスケジューラ (Condor, Grid Engine など) に対応していくことが重要である．
- Java 以外の言語向けクライアントライブラリの実装
  さまざまなアプリケーションレベルスケジューラを実装するためには，それぞれの言語に対してクライアントライブラリを用意する必要がある．まずは，C 言語と python に対してライブラリを用意する予定である．また，Java のクライアントインターフェイスに関しても，プログラミングの容易さという観点で再検討する必要がある．
- さまざまなアプリケーションレベルスケジューラの実装
  オーバレイスケジューラの実用性を確認するためには様々なアプリケーションレベルスケジューラを，Jojo3 上に実装する必要がある．具体的には，統計処理パッケージ R[19] の並列化や遺伝的アルゴリズムに特化したスケジューラ [20]，Ninf-G5[6] の実装を検討している．

参考文献

1) Grid Engine. http://gridengine.sunsource.net.
2) TORQUE Resource Manager. http://www.-clusterresources.com/pages/products/torque-resource-manager.php.
3) Globus Project. http://www.globus.org.
4) UNICORE. http://www.unicore.org/.
5) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *Journal of Grid Computing*, Vol. 1, No. 1, pp. 41–51 (2003).
6) 中田秀基, 田中良夫, 関口智嗣: グリッド RPC システム Ninf-G の可搬性および適応性の改善, 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, Vol. 2007-HPC-112, pp. 37–42 (2007).
7) Nakada, H., Matsuoka, S. and Sekiguchi, S.: A Java-based Programming Environment for hierarchical grid: Jojo, *CCGrid 2004* (2004).
8) Globus ASCII Helper Protocol. http://www.cs.wisc.edu/condor/gahp/.
9) JSON.org: The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627 (2006).
10) JSON-lib. http://json-lib.sourceforge.net/.
11) Condor. http://www.cs.wisc.edu/condor/.
12) Raman, R., Livny, M. and Solomon, M.: Matchmaking: Distributed Resource Management for High Throughput Computing, *Proc. of HPDC-7* (1998).
13) Czajkowski, K., Foster, I., Kesselman, C., Karonis, N., Martin, S., Smith, W. and Tuecke, S.: A Resource Management Architecture for Metacomputing Systems, *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing* (1998).
14) Chirp. http://www.cs.wisc.edu/condor/chirp/.
15) Goux, J.-P., Kulkarni, S., Linderoth, J. and Yorde, M.: An Enabling Framework for Master-Worker Applications on the Computational Grid, *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsuburgh, Pennsylvania, pp. 43 – 50 (2000).
16) MW Homepage. http://www.cs.wisc.edu/condor/mw/.
17) Taura, K.: GXP : An Interactive Shell for the Grid Environment, *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems* (2004).
18) Aoki, H., Nakada, H., Tanaka, K. and Matsuoka, S.: Autonomically-Adapting Master-Worker Programming Framework for Multi-Layered Grid-of-Clusters, *Proc. of HPC Asia 2007*, pp. 3–10 (2007).
19) The R Project for Statistical Computing. http://www.r-project.org/.
20) 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).