

グリッド・コンピューティング  
におけるジョブの類型化とパターン化

TL021338 藤岡 直樹

2006 年 2 月

福岡大学工学部電子情報工学科

## 目次

### 1. グリッド・コンピューティングの概要

1-1. グリッド・コンピューティングとは何か？	1
1-2. グリッド・コンピューティングの目的	1
1-3. グリッド・コンピューティングの問題点	2
1-4. 研究目的	2

### 2. グリッド・コンピューティングの基本技術

2-1. グリッド・コンピューティングに必要な機能	3
2-1-1. 砂時計モデル	3
2-1-2. ファブリック層	4
2-1-3. コネクティビティ層	4
2-1-4. リソース層	4
2-1-5. コレクティブ層	5
2-1-6. アプリケーション層	5
2-2. Globus Toolkit 2 の概要	6
2-2-1. Globus Toolkit 2 の全体構成	6
2-2-2. セキュリティ	7
2-2-3. リソースとジョブの管理	12
2-2-4. リソース情報の収集	15
2-2-5. データの管理	16

### 3. ジョブの類型化

3-1. ジョブの性質による分類	19
3-2. ジョブの利用するノード数による分類	20
3-3. ジョブの計算資源へのリソース要求による分類	21
3-4. ジョブの分類による類型化	22

4. ジョブのパターン化	
4-1. パターン化の概要	24
4-2. プログラムの構成	25
4-2. 演算資源への必要性を解析するプログラムの説明	26
5. 考察	
5-1. 演算資源への必要性の解析結果	28
5-1. 通信資源への必要性の解析方法	29
5-1. ジョブのパターン化の方法	30
付録	31
参考資料	38

# 第1章

## グリッド・コンピューティングの概要

この章ではまず、グリッド・コンピューティングとはどのようなものかを理解し、その上で、グリッド・コンピューティングを実現するために必要な機能をみていく。

### 1-1. グリッド・コンピューティングとは何か？

グリッド・コンピューティング(Grid Computing)の「グリッド」とはアメリカの電力送電網（パワー・グリッド）を語源としている。電力は原子力発電や水力発電など、様々な種類の発電機によって発電され、網の目のように張り巡らされた送電線を通って利用者に届けられる。このとき、利用者はどこで発電された電気なのか、どの経路を通ってきた電気なのか、などという事は一切意識せずに、一定のサービスを受け続けることができる。

グリッド・コンピューティングは、このパワー・グリッドの考え方をコンピューティングに当てはめたものである。パワー・グリッドの考え方をコンピューティングに当てはめると、発電機はCPU、メモリ、ストレージといった計算資源、送電網はネットワーク、電気は計算力やデータと見立てられる。つまり、「パワー・グリッドではコンセントにプラグを差し込むだけで、必要なだけの電気を利用できるように、グリッド・コンピューティングでもネットワーク環境にLANを差し込むだけで、必要な種類、量の計算力やデータを利用することができるようになる」ということである。

### 1-2. グリッド・コンピューティングの目的

複数の計算資源をネットワークで接続・共有し利用してコンピューティングを行うという考え方は、決して新しい考え方ではない。一般に、分散コンピューティングと呼ばれる概念がそれにあたる。分散コンピューティングにはメタ・コンピューティングやクラスタ・コンピューティングなどの様々な形態が存在するが、グリッド・コンピューティングも分散コンピューティングの一部に含まれる。それではグリッド・コンピューティングは従来の分散コンピューティングとどのような点が異なるのか。それは、グリッド・コンピューティングでは、「地理的、組織的に広範囲に分散した計算資源を対象にする」という点と「共有される計算資源が動的に変化する」という概念を採用している点である。一方で、従来の分散コンピューティングでは「地理的、組織的に限定された範囲の計算資

源を対象にする」、「共有される計算資源が固定的である」といった傾向があった。これらより、グリッド・コンピューティングが他の分散コンピューティングと違うところ、つまり、グリッド・コンピューティングにおける目的とは「利用者に対して、地理的、組織的な障壁をほとんど意識せることなく、構成している計算資源の構成を動的に変化させた上で、利用者にあたかも仮想的な1つの巨大な計算資源を提供すること」である。

### 1-3. グリッド・コンピューティングの問題点

前節で述べたようにグリッド・コンピューティングは利用者に対して仮想的な1つの巨大な計算資源を提供する仕組みである。このような仕組みを提供するためには、グリッドを構成している計算資源に、適切にジョブを割り当てる機能が必要となる。例えば、「この機能を持っている仮想的な計算資源」と「この機能を持っていない仮想的な計算資源」の2つの仮想的な計算資源に同じ量の処理を行わせた場合について考える。その場合、「この機能を持っている仮想的な計算資源」では適切にジョブを割り当てるに対して、「この機能を持っていない仮想的な計算資源」では当然のことながら、ジョブの割り当てがうまくいかない。その結果として、後者の計算資源の場合、グリッド・コンピューティングによって提供される仕組みは「仮想的な1つの計算資源」ではなく、「複数個存在する計算資源」となってしまう。これでは、グリッド・コンピューティングを実現することはできない。適切にジョブを割り当てる機能を実現するためには、グリッドを構成している計算資源のリソース情報とグリッドに投入されたジョブの情報が必要となる。これらの情報を取得することがグリッド・コンピューティングの仮想化における課題となっている。

### 1-4. 研究目的

前節のような課題に対して、本研究ではグリッドに投入されるジョブに注目する。投入されるジョブは項目によっていろいろな種類に分類することができる。しかし、ここでは、グリッドを構成する各計算資源に適切にジョブが割り当てられるようにしなければならない。そのため、いろいろな観点から項目を設けて、ジョブの類型化を行っていく。

# 第2章

## グリッド・コンピューティングの基本技術

この章ではグリッド・コンピューティングに必要な機能を理解した上で、今回の実験で使用するグリッド・ミドルウェアである Globus Toolkit の機能紹介をしていく。

### 2-1. グリッド・コンピューティングに必要な機能

#### 2-1-1. 砂時計モデル

グリッド・コンピューティングは、CPU、メモリ、ストレージなどの計算資源（リソース）を仮想化してユーザに提供する仕組みである。このリソースからユーザまではそれぞれの役割によって5つの階層、ファブリック層、コネクティビティ層、リソース層、コレクティブ層、アプリケーション層に分割される。これらの階層構造はその特性から砂時計モデルと呼ばれている（図2-1）。

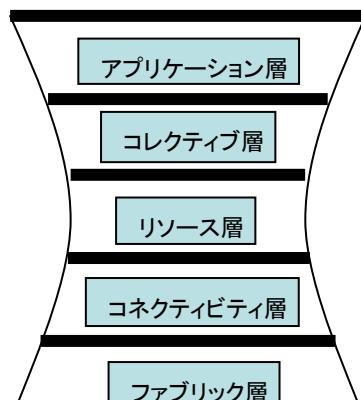


図2-1 砂時計モデル

グリッド・コンピューティングの技術が確立する以前はローカル・リソースとそれを制御するためのアプリケーションとを結ぶ技術が不足していたため、その部分がネックとなってしまい、リソースの仮想化による利用が行えなかった。そして、このネックとなっている真ん中部分がくびれた砂時計を連想させることから、砂時計モデルと呼ばれるようになる。このくびれを解消するために、ローカル・リソース（ファブリック層）とアプリケーション（アプリケーション層）を結ぶコネクティビティ層、リソース層、

コレクティブ層が考案され、この3つの層によって、リソースを仮想化し、ユーザにその仮想化リソースを提供する。それぞれの特徴や役割については次の通りである。

## 2-1-2. ファブリック層

ファブリック層はローカル・リソースとそれをグリッド環境で利用するためのインターフェイスを上位層に提供しているものである。ここでのリソースにはCPU、メモリ、ストレージ、ネットワークといった物理的な計算資源、データベースやファイルシステム上に存在するさまざまな情報資源、分散ファイルシステムやクラスタなどの論理的な計算資源、さらには、これらコンピュータシステムを形成する要素だけでなく、天体望遠鏡やレーダなどの観測機器なども含まれている。

## 2-1-3. コネクティビティ層

コネクティビティ層はファブリック層のリソースにアクセスするために必要な通信プロトコルと認証のプロトコルを提供しているものである。通信プロトコルは、ファブリック層とアプリケーション層、ファブリック層とリソース層とが行う情報通信を仲介している。グリッド・コンピューティングの通信プロトコルにはTCP/IPプロトコルが採用されている。これはグリッド・コンピューティングが仮想組織を実現するために、通信プロトコルを統一する必要があり、TCP/IPプロトコルがインターネット上で情報をやり取りするための世界標準であるからである。また、認証プロトコルはユーザやアプリケーションがリソース層にアクセスする際の、通信の暗号化や通信を行う相手の特定、シングル・サインオン、グリッド環境での様々な資源への権限の委譲といった分散環境であるグリッドにおいて必要なセキュリティの機能を提供している。これらの機能については第3章で説明する。

## 2-1-4. リソース層

リソース層は上位層にローカル・リソースを利用するためのプロトコルを提供しているものです。提供しているプロトコルにはインフォメーション・プロトコルとマネジメント・プロトコルがある。インフォメーション・プロトコルはリソース情報を取得するための標準的な手順を規定しているプロトコルで、例えば、CPU、ストレージ、メモリなどのリソースの種類、ロック数、容量などのリソースの能力、リソースの所有者名、アドレスなどのローカル・リソースの情報から、リソース利用に必要な費用、グリッド環境でのアクセス権限などの仮想化リソースの情報まで取得するものである。マネジメント・プロトコルはデータの転送やプロセスの実行の管理を規定しているプロトコルで、

例えば、リソースへの利用要求、プロセスの生成、データへのアクセス、転送や操作などでのリソースへの要求、操作の状況監視、操作の管理を行うものである。リソース層はコネクティビティ層とでリソースの仮想化を行い、リソースの仮想化が実現できると、ユーザは自分のコンピュータを利用するのに近い感覚で他者のリソースを利用できるようになる。また、グリッド環境に提供されたリソースは仮想組織として、仮想的な1つの組織にまとめられるので、仮想組織のユーザは別々の管理体系に属しているリソースでも、個々の管理体系を意識することなく利用することができるものである。

## 2-1-5. コレクティブ層

リソース層ではローカル・リソースの1つ1つの情報や動作に着目していった。これに対して、コレクティブ層は特定のリソースだけでなくリソース全体を扱い、仮想化されたリソースの利用を手助けするサービスを提供しているものである。グリッド環境において、リソースの仮想化だけであれば、コネクティビティ層とリソース層のみで行うことができる。しかし、これだけでは、仮想化したリソースの1つ1つの情報や動作について、ユーザ自身が扱わなければならず、実際にグリッドの利用が複雑になってしまふ。そこで、それを手助けするサービスが必要となる。コレクティブ層にはそのようなサービスが含まれており、例えば、リソースの一覧を表示するサービス、目的に応じたリソースを検索するサービス、負荷分散サービスといったものが含まれる。

## 2-1-6. アプリケーション層

アプリケーション層はグリッド環境内で実行されるアプリケーションのことである。アプリケーション層は他のあらゆる層のプロトコルやサービスを利用している。リソース層の仮想化リソースの管理プロトコルを使用してプログラムを実行することもあれば、コレクティブ層のリソース検索サービスを利用することもある。また、リソース層を通さずにコネクティビティ層からローカル・リソースにアクセスする場合もある。

## 2-2. Globus Toolkit の概要

今回の実験では Globus Toolkit2.4 というグリッド・ミドルウェアを使用している。Globus Toolkit の目的は、分散した計算機、ストレージ、データ、アプリケーションの共有のための手段を各計算機の OS 上に乗せることで、異なる OS でのリソースの認識の相違を吸収するためのミドルウェアとなることである。そのため、Globus Toolkit のアーキテクチャは OS と見間違うばかりに多岐にわたり複雑となっている。ただし、Globus Toolkit はあくまでもグリッドサービスを提供するためのインフラであり、それ自体がユーザに対してサービスを直接提供するものではない。つまり、ユーザに対しては、Globus Toolkit がグリッド環境を提供し、その上に乗るアプリケーションがグリッドサービスを提供するのである。そのような意味でも、Globus Toolkit は OS に近い位置づけのミドルウェアであるということができる。

### 2-2-1. Globus Toolkit2 の全体構成

Globus Toolkit2 のアーキテクチャはよく、3 本の柱とその下にある土台で描かれている（図 2-2）。3 本の柱と、柱を下で支えている土台が、Globus Toolkit を用いて構成したグリッド・コンピューティングの基盤となる。3 本の柱を下で支えている土台の部分が、グリッド・コンピューティングのシステム全体のセキュリティを担当しているコネクティビティ層に相当する。これは「Grid Security Infrastructure (GSI)」と呼ばれているものである。3 本の柱はリソース層、一部のコレクティブ層に相当する。左の柱（Resource Management）はリソースとジョブの管理を担当し、「Globus Resource Allocation Manager (GRAM)」、「Globus Access to Secondary Storage (GASS)」というコンポーネントから構成されている。中央の柱（Information Services）はリソース情報の収集を担当し、「Monitoring and Discovery Service (MDS)」というコンポーネントから構成されている。右の柱（Data Management）はデータの管理を担当し、「GridFTP」いうコンポーネントから構成されている。

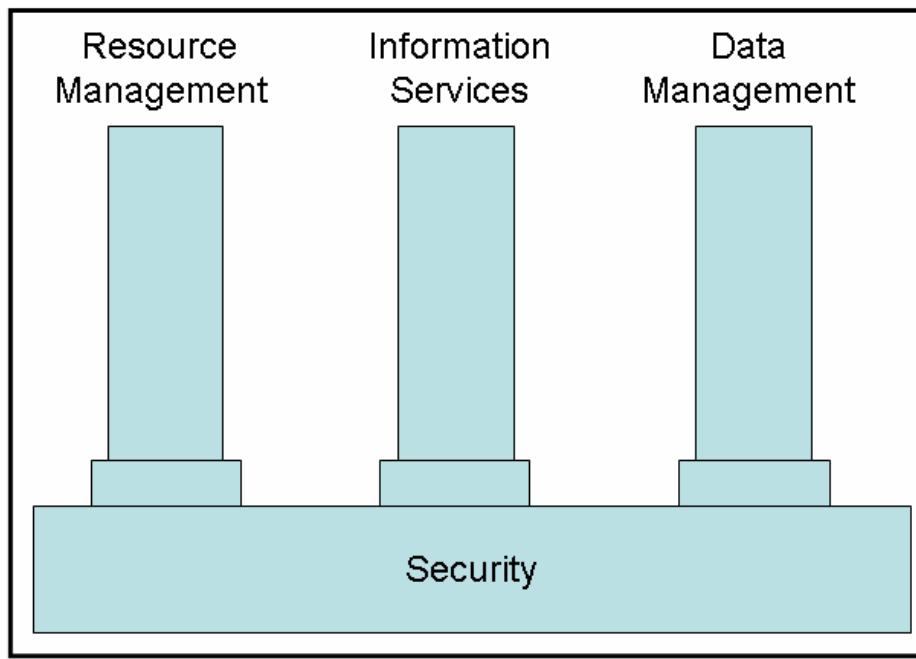


図 2-2 Globus Toolkit のコンポーネント

## 2-2-2. セキュリティ — GSI

グリッド・コンピューティングはオープンなネットワーク環境において、他の計算資源を利用するということを目的としている。しかし、このような状況では、必ずセキュリティにおける問題が発生する。グリッド環境において、利便性を追求していくと安全性が低下し、逆に、安全性を追求していくと利便性が低下していく。Globus Toolkit2 では、認証、認可、通信内容の保護という項目で、グリッド・コンピューティングのオープンなネットワーク環境でのセキュアな利用を実現している。認証と許可、通信内容の保護という項目は Grid Security Infrastructure (GSI) という機能が実現している。GSI は Globus Toolkit2 におけるコンポーネントの基礎となる重要な機能で、認証と許可の項目では「通信相手の特定」、「シングル・サインオン」、「権限の委譲」、通信内容の保護の項目では「通信の暗号化」などを行っている。GSI のレイヤーの上にその他のコンポーネント (GRAM、MDS、GridFTP) が乗ることにより、その他のコンポーネントのセキュリティが保障されているのである。

## 通信相手の特定

Globus Toolkitにおける、ジョブの実行時に必要とされる認証は、「リクエストしてきたクライアントは正規のクライアントであるのか」ということをサーバが知ることと、「リクエストしたサーバは正規のサーバであるのか」ということをユーザが知ることの2つである。つまり、リクエストを投入するクライアントとそのリクエストを受け取るサーバ間での相互認証が必要となる。この認証で必要な条件は「クライアントが自分を証明する証明書を持っていること、サーバが自分を証明する証明書を持っていること、また、その証明書を署名した認証局の証明書をクライアントとサーバがそれぞれ持っていること」である。このようにGISによる認証を行うためには証明書が必要となる。

プロキシ証明書を使用したクライアントとサーバの相互認証は以下の図2-3、図2-4のようになる。まず、クライアントはユーザ秘密鍵のパスワードを入力して、プロキシ証明書を作成する。このプロキシ証明書の中には新しく作った新証明書と新秘密鍵、また、認証局に証明されているユーザ証明書が入っている。クライアントがサーバに対してリクエストを送ると、GISによる認証が始まる。クライアントはプロキシ証明書の中にある新証明書とユーザ証明書をサーバ側に送信する。

- ① サーバ側は送られてきたユーザ証明書に対し、サーバが持っている認証局の証明書を使用し、「本当に認証局に署名されたものなのか」を確認する。真正性を確認したら、このユーザ証明書の中から公開鍵を取り出す。
- ② サーバ側は送られてきた新証明書に対し、先ほど取り出した公開鍵を使用し、「本当にユーザ秘密鍵によって署名されたものなのか」を確認する。真正性を確認したら、この新証明書の中から新公開鍵を取り出す。
- ③ サーバ側の先ほど取り出した新公開鍵とユーザ側のプロキシ証明書の中の新秘密鍵が「本当に正しい組み合わせであるのか」を確認する。真正性が確認できたら、「クライアントが正規のクライアントである」ということが確認できると、次はサーバの証明に移る。
- ④ サーバはサーバ証明書をユーザ側に送信する。
- ⑤ ユーザ側は送られてきたサーバ証明書に対し、ユーザが持っている認証局の証明書を使用し、「本当に認証局に署名されたものなのか」を確認する。真正性を確認したら、このサーバ証明書の中から公開鍵を取り出す。
- ⑥ ユーザ側の先ほど取り出した公開鍵とサーバ側の秘密鍵が「本当に正しい組み合わせであるのか」を確認する。真正性が確認できたら、「サーバが正規のサーバである」ということが確認できる。

以上のことにより、クライアントとサーバの相互認証が完了するので、コネクションが確立し、実際にクライアントのリクエストをサーバが受け取る。

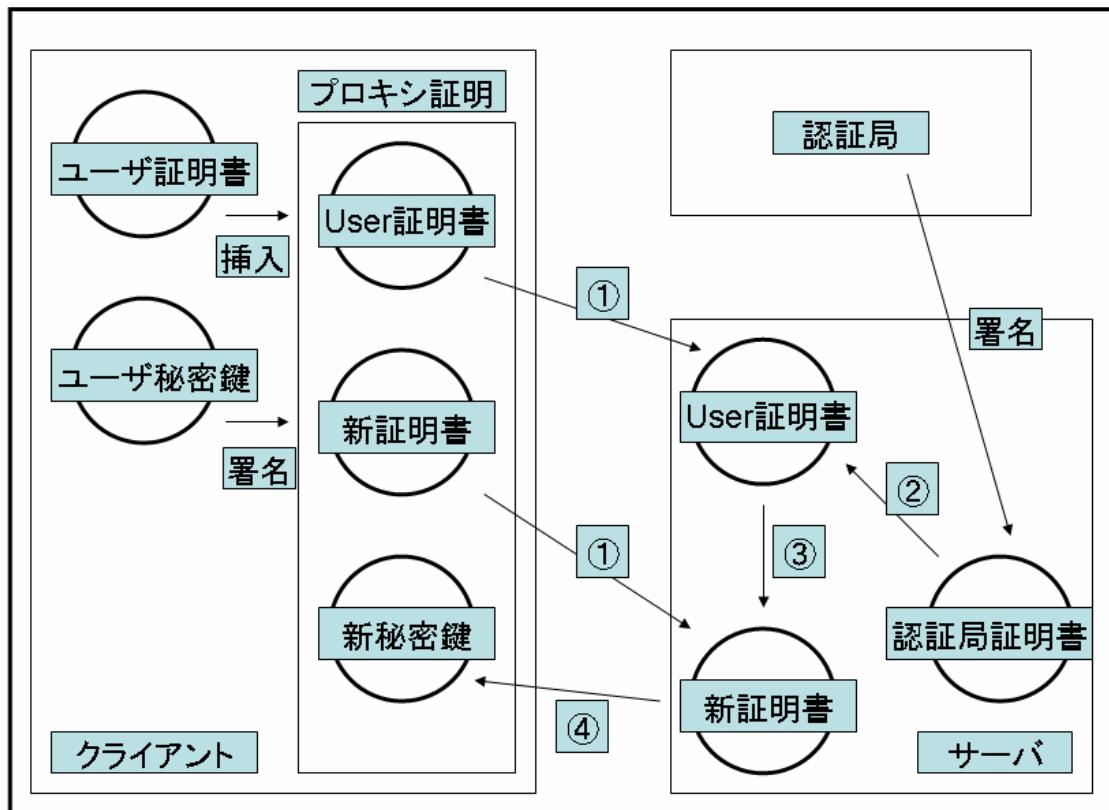


図 2-3 GSI によるクライアントの認証

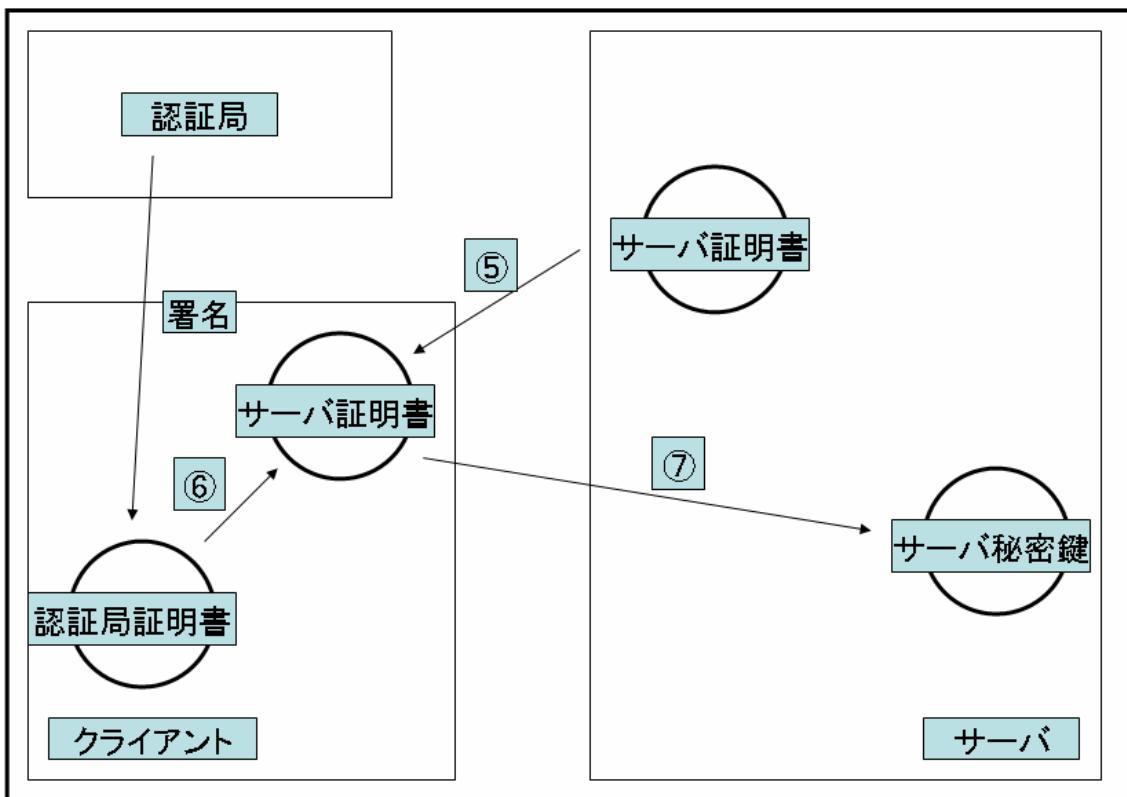


図 2-4 GSI によるサーバの認証

### シングル・サインオン

例えば、ネットワーク上で複数のサーバがサービスを提供しており、各サーバのサービスを利用するためには各サーバへの認証が必要であるとする。このような場合、10台のサーバのサービスを利用するためには、10台のサーバそれぞれに認証を取らなければならない。これでは、せっかくのサービスも認証に時間や手間がかかってしまい、十分に利用することができない。このような問題はグリッド環境においても発生する。先に述べたように、グリッド環境のローカル・リソースはオープンなリソースとして公開されていますので、その利用には認証が必要なのは当然のことである。利用するグリッド環境のすべてのローカル・リソースに対しての認証を1つ1つしていくは、せっかくのグリッド環境の計算資源も、十分に利用することはできない。このような問題を解決するに、GSIではシングル・サインオンという認証ソリューションを提供している。シングル・サインオンとは、それぞれに認証を要求する複数のサービスや機能に対し、1度の認証で利用可能とする認証システムを指す。これは認証を集中的に行う認証サーバを設けてそれに1度認証されれば、認証サーバにあらかじめ登録されたサービスを個々の認証なしに利用することができるようなシステムである。グリッド環境でも、これと同様に、1回のサインオンでファブリック層の複数のリソースにアクセスできるようになる。

## 権限の委譲

GSI では権限の委譲の機能により、グリッド環境における複数台のホスト間でのやり取りの際の利便性を高めている。権限の委譲はプロキシ証明書を使用して、パスワードの入力なしに権限の委譲先で新しくプロキシ証明書を作成できることにより実現される。例えば、GRAM によるリモートジョブの実行プロセスが、さらに GridFTP によるデータ転送を必要としているとする。つまり、リモートジョブの実行プロセス自身が、GridFTP にデータ転送を要求している状態である。rsh（リモートシェルのコマンドで実行時にパスワードを要求される）などで実行されるスクリプトが、さらに別のホストから FTP などでデータを取得する場合、通常の認証ではリモートジョブの実行に対する認証と FTP によるデータ転送に対する認証の 2 つそれぞれにパスワード認証が行われる。手順は以下の図 2-5 のようになる。

- ① クライアントにおいて、GRAM サーバに対するプロキシ証明書を作成する。
- ② そのプロキシ証明書を使用して GRAM サーバの認証を行う。
- ③ クライアントにおいて、GridFTP サーバに対するプロキシ証明書を作成する。
- ④ そのプロキシ証明書を使用して GridFTP サーバの認証を行う。

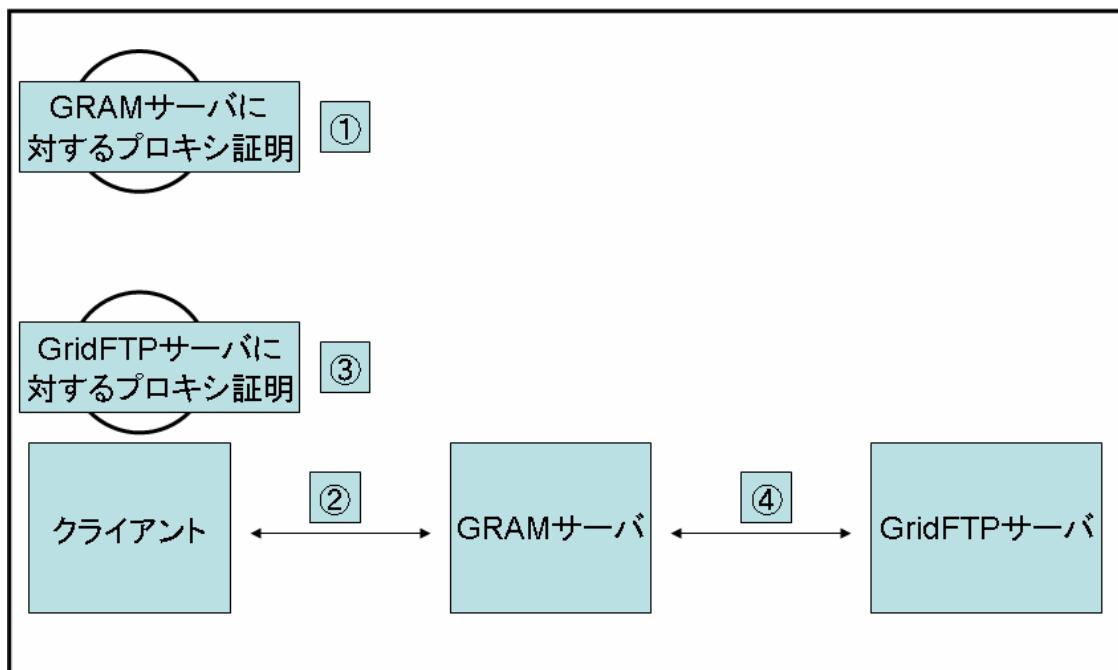


図 2-5 通常の認証

これに対し、GSI の権限の委譲では、いちいち 2 つの認証のプロセスのために、2 つのプロキシ証明書をクライアントが明示的に用意するのではなく、はじめにクライアントで作成したプロキシ証明書のひとつで、GRAM によるリモートジョブの実行と GridFTP

によるデータ転送のクライアント側の認証を行うことが可能である。手順は以下の図 2-6 のようになる。

- ① クライアントにおいて、プロキシ証明書を作成する。
- ② そのプロキシ証明書を使用して GRAM サーバの認証を行う。
- ③ クライアントから GRAM サーバに対して権威委譲が行う。これにより、実行プロセスが GRAM サーバ上にプロキシ証明書を作成する。
- ④ 権威委譲によって GRAM サーバ上に作成されたプロキシ証明書を使用して GridFTP サーバの認証を行う。

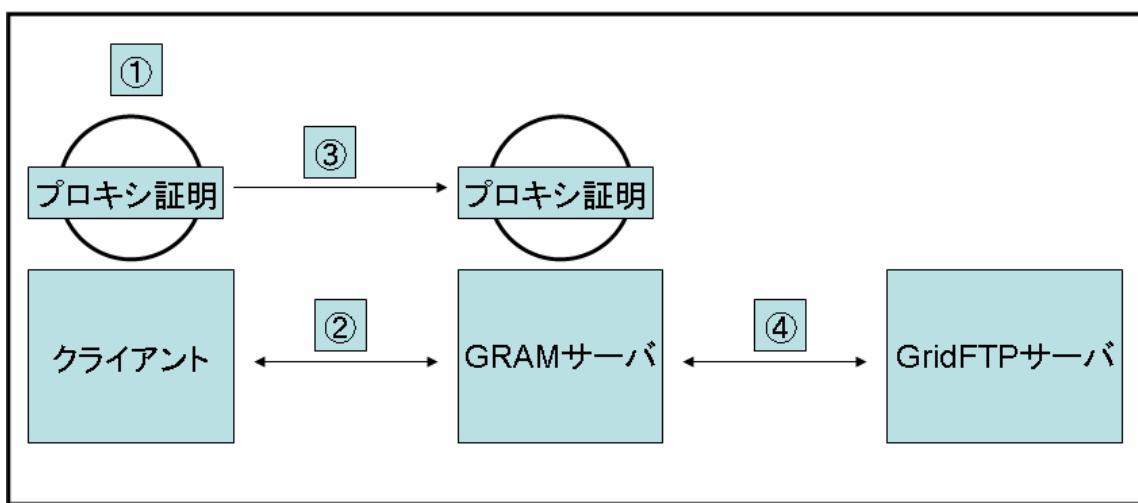


図 2-6 GIS の権限の委譲による認証

### 通信の暗号化

GSI による通信の暗号化には SSL が用いられる。SSL はインターネット上で情報を暗号化して送受信するプロトコルである。現在インターネットで広く使われている WWW や FTP などのデータを暗号化し、通信される情報が安全に送受信されるようにしている。

### 2-2-3. リソースとジョブの管理 — GRAM

GRAM はジョブを実行するためのリクエストとリモートシステムのリソースを使用するための標準的なインターフェイスを提供しているものである。GRAM によるジョブの実行には同期処理と非同期処理の 2 種類を指定することができる。同期処理はリモートの実行ホストの処理をリクエストすると、処理が終わるまでコネクションを持続させるタイプのジョブである。一方、非同期処理はリモートの実行ホストの処理をリクエストしたら、コネクションを切断してしまうタイプのジョブである。まず、同期処理型のジョ

の実行の仕組みは以下の図 2-7 のようになる。GRAM クライアントから GRAM サーバに  
対してリクエストが出される。GRAM サーバは Gatekeeper で GRAM クライアントからのリ  
クエストを受け取る。

- ① Gatekeeper は GSI に基づいた相互認証を実行する。この認証に、クライアントはユ  
ーザ証明書を、GRAM サーバ側はホスト証明書を使用する。
- ② 認証に成功すると、Gatekeeper は実際に処理をする内容をクライアントから受け取  
る。
- ③ Gatekeeper はマッピングファイルを確認し、リクエストしたユーザの識別子を GRAM  
サーバ上のローカルユーザにマッピングする。
- ④ Gatekeeper はマッピングしたローカルユーザで要求された Job Manager を起動し、  
Job Manager にジョブを渡す。
- ⑤ Job Manager は Local Resource Manager にジョブを振り分け、要求されたジョブを  
実行する。Local Resource Manager は GRAM サーバ自身が単体でジョブを実行する  
場合と、PBS や CONDOR などの他のローカルスケジューラと連携して処理を行う場合  
に分けることができる。
- ⑥ GRAM サーバもしくは他のリモートの実行ホストでジョブが終了すると、GRAM サーバ  
のキャッシュディレクトリに処理結果である標準出力と標準エラー出力がファイル  
として出力される。
- ⑦ Job Manager は GASS サーバ経由で処理結果をクライアントに返す。その後、キャッ  
シュディレクトリに出力された処理結果のファイルは削除される。

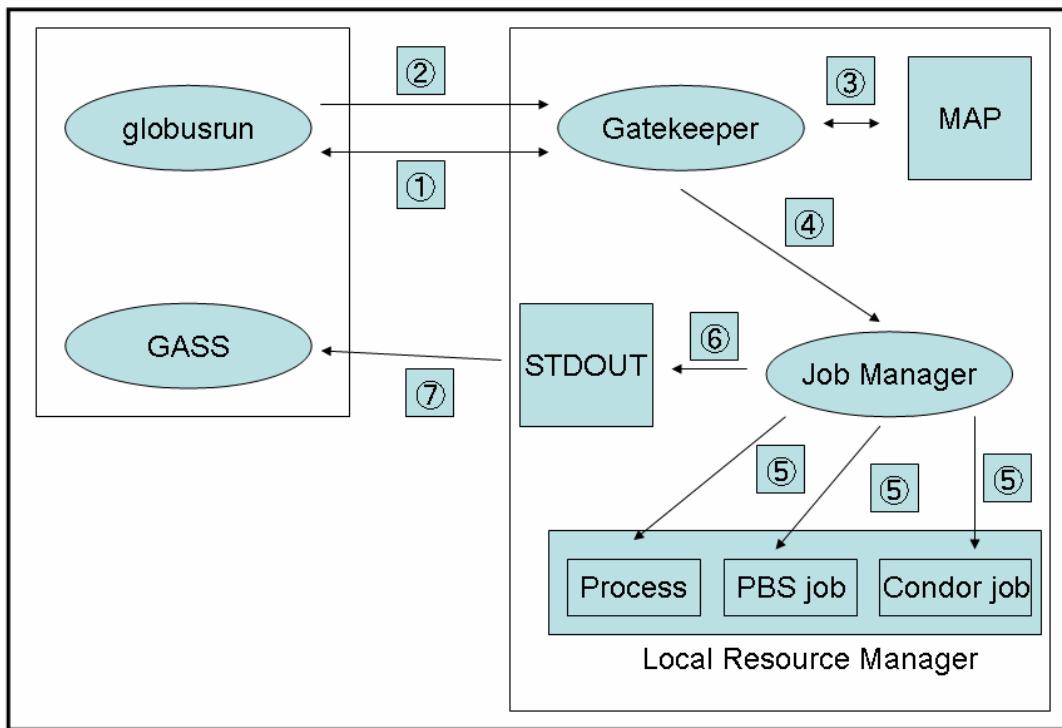


図 2-7 GRAM による同期処理の流れ

次に、非同期処理型のジョブの実行の仕組みは以下の図 2-8 のようになる。

- ① 同期処理型と同じ。
- ② 同期処理型と同じ。
- ③ 同期処理型と同じ。
- ④ 同期処理型と同じ。
- ⑤ 同期処理型と同じ。
- ⑥ 投入したジョブに対するジョブ ID というものをクライアントに返す。同期処理型では投入したジョブが終了するまでコネクションを持続させるが、非同期処理型ではここで投入したジョブの終了を待たずにコネクションを切断する。
- ⑦ クライアントはジョブ ID を利用して Job Manager にジョブの処理結果を確認する。ジョブの処理結果は Job Manager がファイルで保持している。GRAM クライアントがジョブの処理結果をリクエストすると、まず、クライアント上で GASS サーバが起動する。
- ⑧ サーバ側はリクエストを受けると、ジョブ ID に関連した処理結果のファイルをクライアントの GASS サーバに転送する。これにより、クライアントは処理結果を取得することができる。

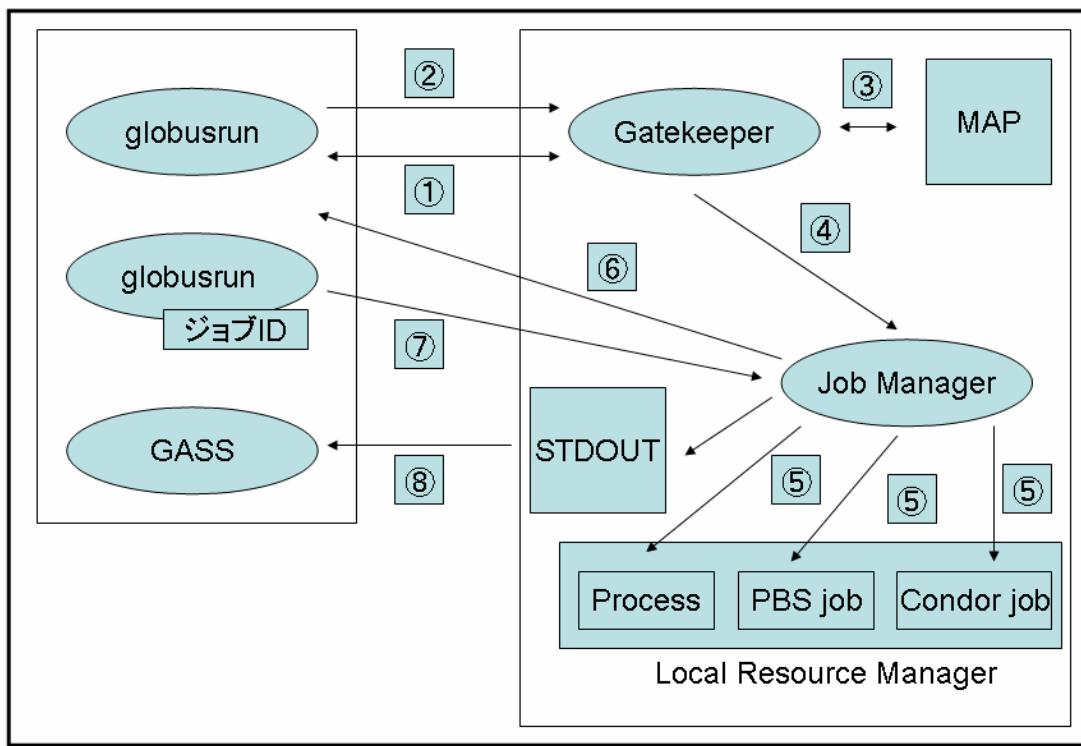


図 2-8 GRAM による非同期処理の流れ

## 2-2-4. リソース情報の収集 — MDS

MDS はグリッド環境で使用されるリソース情報の定義を提供する。MDS によって検索することのできるリソース情報には、OS の情報、ネットワークのインターフェイス情報、といった静的なリソース情報や、CPU 情報、メモリ情報、といった動的なリソース情報がある。それでは、MDS はどのようにしてリソース情報を取得することができるのでしょうか。それにはまず、MDS コンポーネントを「Grid Resource Information Service(GRIS)」と「Grid Index Information Service(GIIS)」という 2 つのコンポーネントに分けて考える。まず、GRIS はローカルマシンのリソース情報を調べて、キャッシュしていくコンポーネントである。キャッシュは一定間隔ごとに GRIS 本体に報告されていくので、GRIS はローカルのリソース情報全体を保持し、公開するようになる。次に GIIS では下位の GRIS または GIIS から報告されたリソース情報を管理するコンポーネントである。この GIIS の働きにより、グリッド環境に参加しているマシンのリソース情報は分散化して管理することができる。しかし、GIIS 単体、もしくは GIIS のみではリソース情報を収集することはできず、GIIS は GRIS によって収集されたリソース情報によって情報を保持していく。これら 2 つコンポーネントにより MDS には以下の図 2-9 のような情報収集の方法がある。

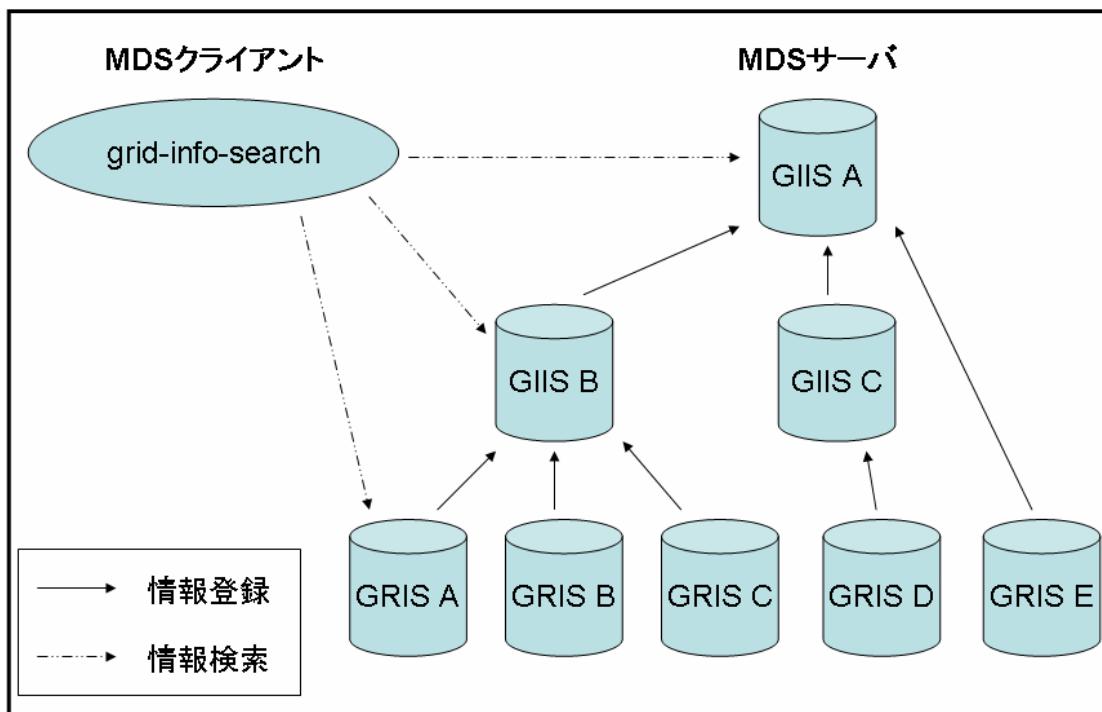


図 2-9 MDS によるリソース情報の取得の流れ

1つ目のパターンは最上位の GIIS サーバ A に対しての検索である。この検索によって得られる情報は GRIS サーバ A、B、C、D、E すべてのリソース情報である。GRIS サーバ A、B、C のリソース情報は GIIS サーバ B に、GRIS サーバ D のリソース情報は GIIS サーバ C に、GRIS サーバ E のリソース情報は GIIS サーバ A にキャッシュされている。2つ目のパターンは中間に位置する GIIS サーバ B に対しての検索である。この検索によって得られる情報は GIIS サーバ B に対して、リソース情報の報告を行っているので、GRIS サーバ A、B、C のリソース情報である。3つ目のパターンは直接 GRIS サーバに対して行う検索である。GRIS サーバ A に対して検索を行うと、得られる情報は GRAS サーバ A のリソース情報のみになる。このような3つの検索方法を比較すると、上位の GISS サーバに対しては一定間隔で情報を報告するので、下位の方のサーバに対して検索を行ったほうがより新しい情報を取得することができる。

## 2-2-5. データの管理 — GridFTP

GridFTP はグリッド環境にセキュアなデータ転送メカニズムを提供しているものである。データ転送は GASS コンポーネントでも行うことができるが、GridFTP は GASS コンポーネントよりも高度なデータ転送を行うことができるので、通常、Globus Toolkit におけるデータ転送は GridFTP を使用する。それでは GridFTP のデータ転送はどのように行われるのかを、GridFTP にはいくつかある便利なデータ転送機能のうちで代表的な

サードパーティファイル転送を例にとって、その仕組みを見ていく。まず、従来のFTPによるサーバAからサーバBへのファイル転送での手順は以下の図2-10のようになる。クライアントはサーバAにGSIによる相互認証を行う。

- ① 転送したいファイルをサーバAからクライアントに転送する。
- ② クライアントはサーバBにGSIによる相互認証を行う
- ③ 転送したいファイルをクライアントからサーバBに転送する。

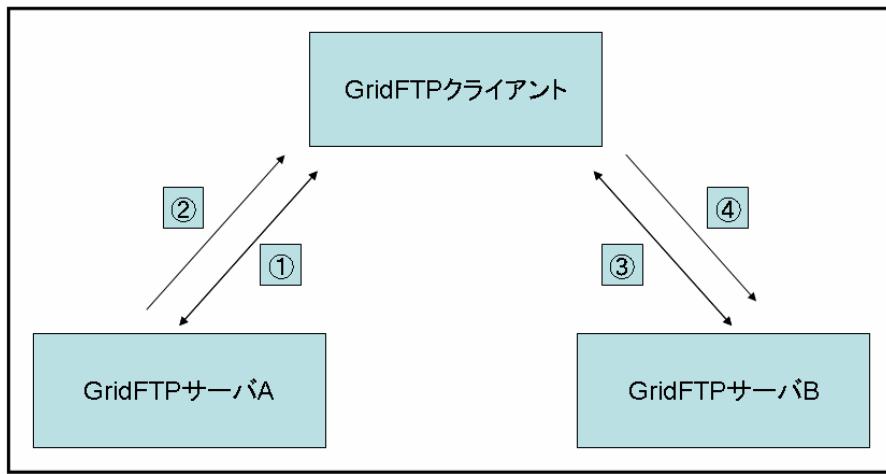


図 2-10 FTP のファイル転送

次に、GridFTP のサードパーティファイル転送による、サーバAからサーバBへのファイル転送での手順は以下の図2-11のようになる。

- ① クライアントはGridFTPサーバAに相互認証を行う。
- ② クライアントはGridFTPサーバAにGridFTPサーバBに転送したいファイルを転送するようにリクエストを出す。
- ③ クライアントはGridFTPサーバBに相互認証を行う。
- ④ GridFTPサーバAからGridFTPサーバBにファイル転送が行われる。

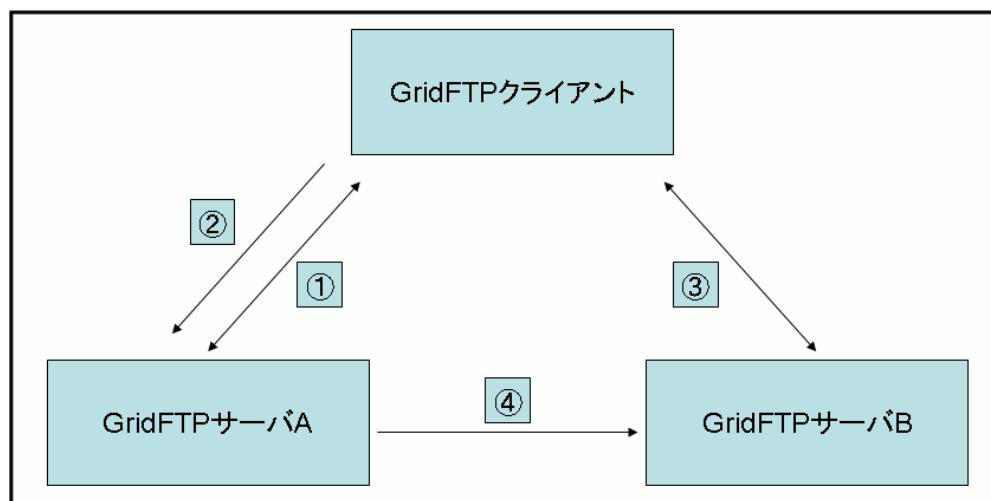


図 2-11 GridFTP のファイル転送

これらのこととを比較すると、従来の FTP でのファイル転送では転送を 2 回行わなければならぬのに対して、GridFTP では GridFTP サーバ A から GridFTP サーバ B に直接転送できるので 1 回の転送で済むのである。

# 第3章

## ジョブの類型化

第2章で述べたように、Globus Toolkit はあくまでもグリッドサービスを提供するためのインフラであるので、Globus Toolkit はユーザに対して直接、サービスを提供してはくれず、グリッドに投入されたジョブに適切な処理が行われるとは限らない。そのため、そのようなサービスはアプリケーションが提供してやらなければならない。この章では、グリッドに投入されたジョブを、「ジョブの性質」、「利用するノード数」、「計算資源に対するリソース要求」の3項目によって分類を行う。この類型化はジョブを分類していくことにより、ジョブがグリッド環境の適切な実行ホストに割り振られるようにするものである。

### 3-1. ジョブの性質による分類

ここでジョブは、ジョブの性質によってインタラクティブ・ジョブとバッチ・ジョブの2つに分類する。インタラクティブ・ジョブは GRAM サーバ上において、Job Manager でジョブを実行するときに、リモートの実行ホストが処理を完了するまでクライアントとのコネクションは継続されて、実行されるジョブである。このジョブはターミナルへの出力やユーザからの入力を必要とするため、ジョブの実行を行っている間、他の処理を行うことができない。そのため、素早いレスポンスが要求され、計算資源へのリソース要求が小さいジョブがこれにあてはまる。一方、バッチ・ジョブは GRAM サーバ上において、Job Manager でジョブを実行するときに、リモートの実行ホストの処理をリクエストすると、クライアントとのコネクションは切断されて、実行されるジョブである。このジョブは実行中の入出力を必要としないため、ジョブの実行をおこなっている間でも、他の処理を行うことができない。そのため、素早いレスポンスを要求されず、計算資源へのリソース要求が大きなジョブがこれにあてはまる。しかし、グリッド上で実行した場合、インタラクティブ・ジョブは、ジョブの実行時間におけるオーバーヘッドが無視できなくなる。つまり、実際のジョブの実行時間は短く、全体の実行時間における通信と認証にかかる時間が、無視できなくなるのである。これらのオーバーヘッドにより、グリッド上のインタラクティブ・ジョブの実行では、グリッドによる恩恵を得られなくなる。一方で、バッチ・ジョブは実際のジョブの実行時間は長く、全体の実行時間における通信と認証にかかる時間は、無視でき、また、グリッドにより全体の実行時間が短縮される可能性があるので、グリッドによる恩恵を得られるのである。よって、グリッド環境での実行にはバッチ・ジョブが適しているので、投入されるジョブはバッチ・ジョブということに限定して、次のジョブ

の分類を行っていく。

### 3-2. ジョブの利用するノード数による分類

ここでのジョブは、利用するノードによる分類により、シングル・ジョブとパラレル・ジョブの2つに分類する。シングル・ジョブはGRAMサーバ上において、Job Managerにジョブを振り分けられたLocal Resource Managerが、GRAMサーバで処理を行うジョブである。当然のことながら、GRAMサーバが単体で処理を実行するので、処理の形式は逐次処理となる。パラレル・ジョブはGRAMサーバ上において、Job Managerにジョブを振り分けられたLocal Resource Managerが、その他のローカルスケジューラと連携して処理を行うジョブである。複数の実行ホスト上で処理が実行されるので、処理の形式は並列処理となる。ここで重要なことは、シングル・ジョブはグリッド環境では、これ以上分割することのできない最小単位のジョブであるということである。つまり、シングル・ジョブは直接実行ホストに投げられるので、投げられるジョブの形態が変わることがなく、最も正確に分類することができる。一方で、パラレル・ジョブはいくつかのジョブに分割されて、分割された各ジョブが実行ホストに投げられるので、投げられるジョブの形態が変わってしまい、正確に分類することができない。しかし、分割された各ジョブはグリッド環境では直接実行ホストに投げられるジョブであるのでそれ以上分割されることはない。そのため、分割されたパラレル・ジョブの各ジョブはシングル・ジョブとみなすことができる。つまり、パラレル・ジョブとはシングル・ジョブの集合であるとみなすことができる所以である（図3-1）。よって、グリッド環境で実行されるジョブは全てシングル・ジョブから成り立っているので、投入されるジョブはいったんシングル・ジョブに変換された後に、次のジョブの分類を行っていく。

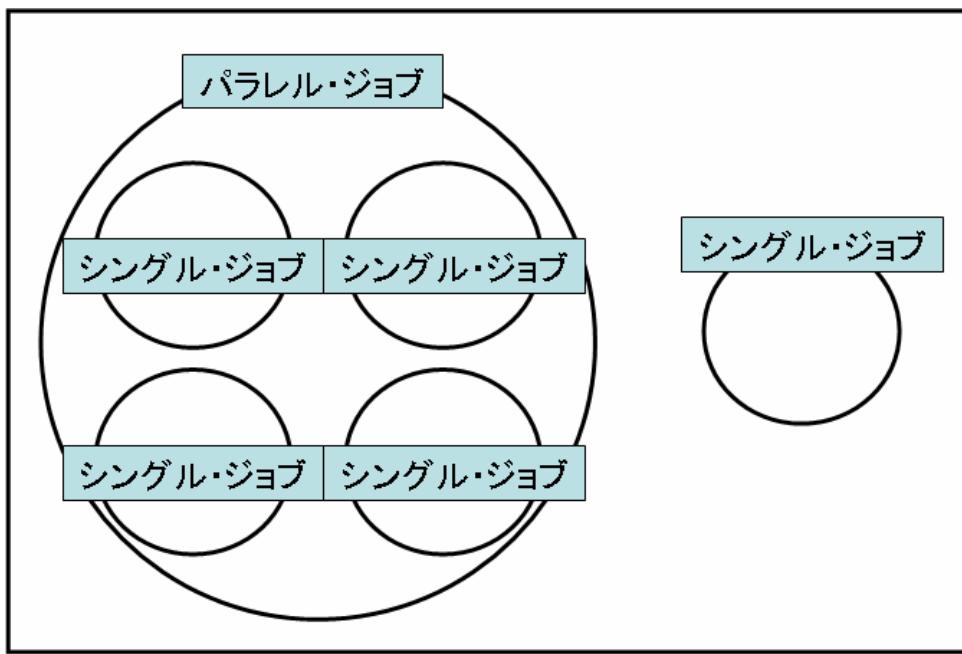


図 3-1 パラレル・ジョブとシングル・ジョブ

### 3-3. ジョブの計算資源へのリソース要求による分類

同一のジョブの実行を行う場合、一般に、処理を行うリソースの性能が高ければ高いほど、より早く処理を完了することができる。しかし、提供されているリソースの性能にも上限があり、場合によっては、性能の低いリソースしか使用できないかもしれない。そのような場合でも、最良の処理を行わなければならない。最良の処理を行うためには、与えられたシングル・ジョブもしくはシングル・ジョブ群（パラレル・ジョブを分割したもの）を適切な実行ホストに投入しなければならない。そのため、ここでのジョブは、計算資源へのリソース要求による分類により、4つに分類する。ジョブを4つに分類するためには、「演算資源を必要とする要求」と「通信資源を必要とする要求」の2つの項目それについて、「演算資源を必要とする要求」をX軸に、「通信資源を必要とする要求」をY軸にとり、ジョブを解析して、それぞれに値を振っていき、数値を求めていく。数値を求めていくと、以下の図3-2のようなグラフの4つの分類のどれかにあてはまっていき、4つのそれぞれ異なったリソース要求を持つ4つのジョブに分類することができる。「演算資源を必要とする要求」はCPUやメモリなど、演算処理を行う際の実行速度に関連するものから成り立つ。この項目の値が高いほど、演算資源におけるリソース要求は大きくなる。「通信資源を必要とする要求」はネットワークなど、データ通信を行う際の通信速度に関連するものから成り立つ。この項目の値が高いほど、通信資源におけるリソース要求は大きくなる。

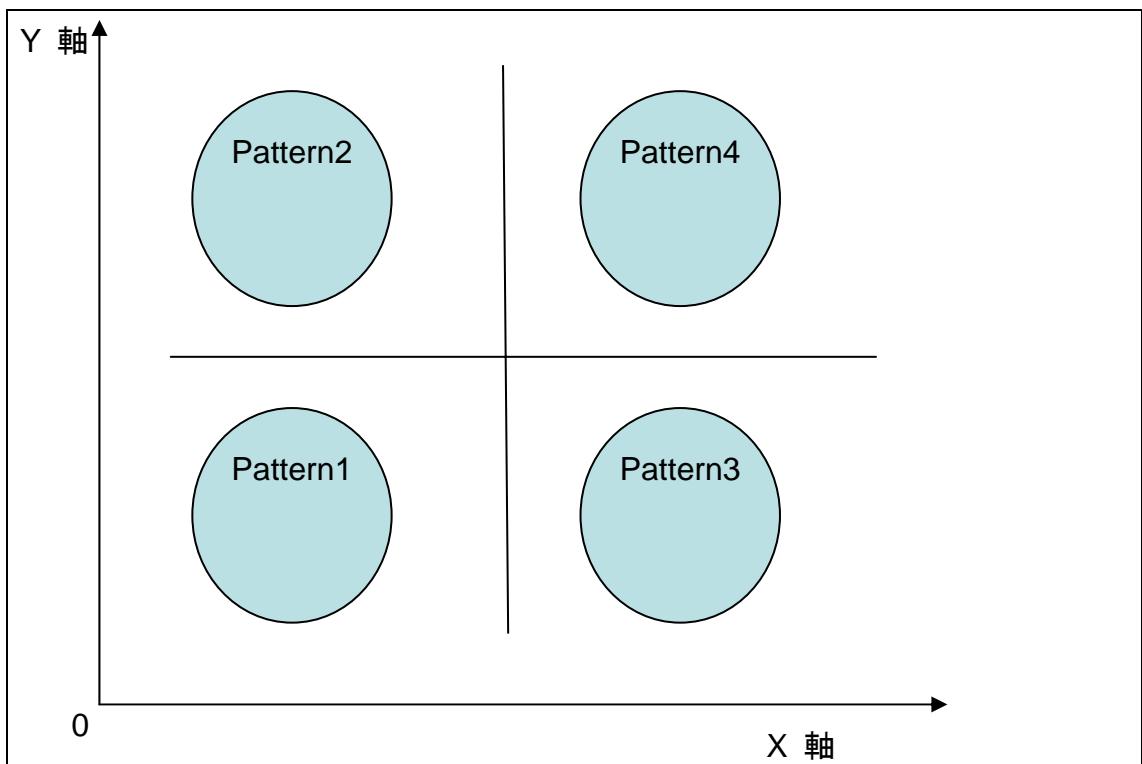


図 3-2 ジョブの計算資源へのリソース要求による分類

Pattern1 は、低性能演算資源、低性能通信資源というリソース上で、最低限の最良な処理が行えるジョブの類型である。ここに分類されるジョブは演算処理と通信処理の両方とも軽い処理で、リソース要求も比較的小さくなる。Pattern2 は、低性能演算資源、高性能通信資源というリソース上で、最低限の最良な処理が行えるジョブの類型である。ここに分類されるジョブは演算処理よりも通信処理に重点が置かれたものである。Pattern3 は、高性能演算資源、低性能通信資源というリソース上で、最低限の最良な処理が行えるジョブの類型である。ここに分類されるジョブは通信処理よりも演算処理に重点が置かれたものである。Pattern4 は、高性能演算資源、高性能通信資源というリソース上で、最適な処理が行えるジョブの類型である。ここに分類されるジョブは演算処理と通信処理の両方とも重い処理で、リソース要求は比較的大きくなる。

#### 3-4. ジョブの分類による類型化

グリッドに投入されたジョブを、「ジョブの性質」、「ジョブの利用するノード数」、「ジョブの計算資源に対するリソース要求」の 3 項目により分類することによって、いくつかのジョブの類型を作成することができる。まず、「ジョブの性質」による分類では、インタラクティブ・ジョブとバッチ・ジョブの 2 つに分類されるが、インタラクティブ・ジョブはグリッド環境での処理には向いていないので、この段階でのジョブは全て、バ

バッチ・ジョブとなる。次に、「ジョブの利用するノード数」による分類では、シングル・ジョブとパラレル・ジョブの2つに分類されるが、パラレル・ジョブはジョブの正確な分類のために、分割されてシングル・ジョブ群となり、シングル・ジョブとして扱われる所以、この段階でのジョブは全て、バッチ・ジョブかつシングル・ジョブとなる。最後に、「ジョブの計算資源に対するリソース要求」による分類では、「低性能演算資源、低性能通信資源を必要とするジョブ」、「低性能演算資源、高性能通信資源を必要とするジョブ」、「高性能演算資源、低性能通信資源を必要とするジョブ」、「高性能演算資源、高性能通信資源を必要とするジョブ」の4つに分類される。よってこれらの分類により、バッチ・ジョブかつシングル・ジョブであるジョブを分類する「低性能演算資源、低性能通信資源を必要とするジョブ」「低性能演算資源、高性能通信資源を必要とするジョブ」、「高性能演算資源、低性能通信資源を必要とするジョブ」、「高性能演算資源、高性能通信資源を必要とするジョブ」という4つの項目の類型が作成される。

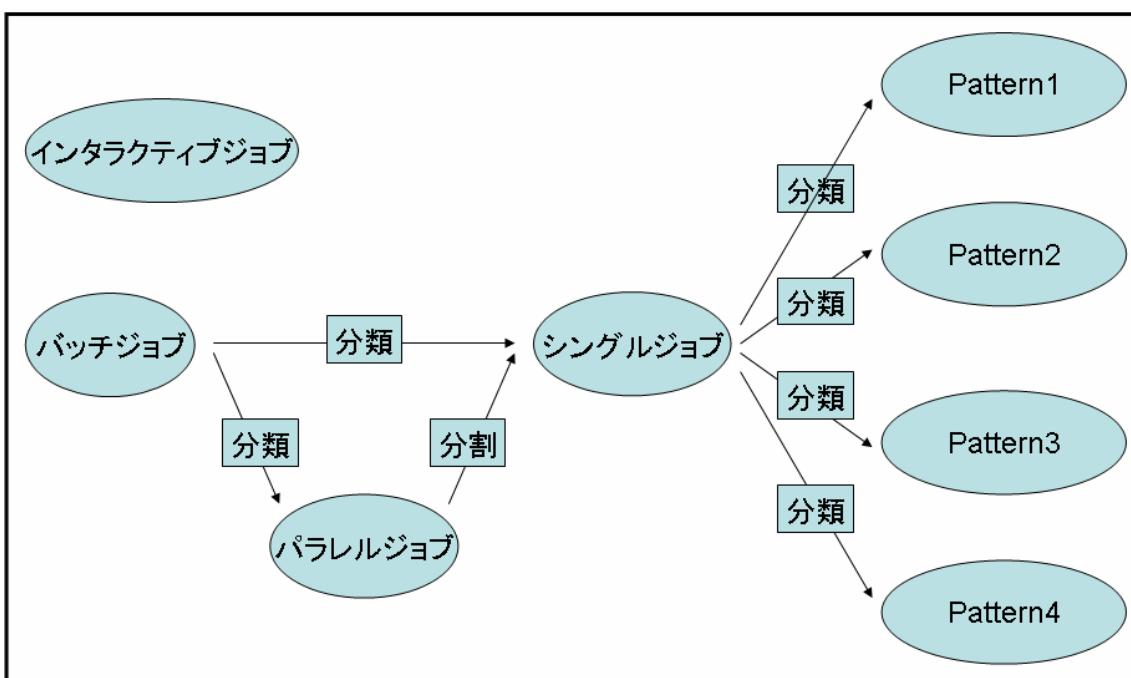


図 3-3 ジョブの分類

# 第4章

## ジョブの分類によるパターン化

この章では第3章で述べた分類化の理論から、実際にPerlのスクリプトに対して、パターン化を行う解析プログラムを作成していく。

### 4-1. パターン化の概要

まず、第3章で用いた分類を行うための2つの項目「演算資源を必要とする要求」と「通信資源を必要とする要求」に注目する。これら2つの項目によって、最終的な分類が行われるので、これらの項目において、分類のための解析を行っていく。本実験においては、スクリプト言語であるPerlで記述されたプログラムを分類対象とする。本実験で、スクリプト言語を分類対象のプログラムとしたのはグリッド環境に投げられたジョブ（プログラム）が、それ単体で実行することができ、プログラムの解析が容易であるからである。コンパイル型の言語の場合、それ単体で実行することはできるが、プログラムを解析するためにはソースファイルを必要とする。そのため、ソースファイルが存在しない場合は、プログラムの解析を行うことができないので、本実験ではスクリプト言語で記述されたプログラムの解析を行う。スクリプト言語の中でも、Perlを用いたのはPerlが文字列処理などに優れているので、グリッド環境で処理を実行するのに適しているからである。しかし、グリッド環境での実行にPerlが最も適しているというわけではない。その他にもスクリプト言語は存在し、その用途に応じて優れたものがある。よって、本実験ではPerlに対して解析を行うが、状況によっては他のスクリプト言語を使用することもあることがある。

#### 演算資源に重点がおかれる場合

演算資源に対するリソース要求が大きくなるのは繰り返しなどにより、大規模の演算処理が行われるときである。但し、繰り返しが記述されている全てのプログラムで、演算資源に対するリソース要求が大きくなるわけではない。プログラムの中に記述されている繰り返しはあくまでも可能性である。繰り返しが記述されているが、短時間で処理が終了するものもあれば、長時間かかるものもあるのである。つまり、繰り返しの内容も解析していく必要がある。Perlの繰り返しにはfor文、while文、until文がある。ここではこれらの使用については言及しないが、解析処理ではこれらの繰り返しの命令文を検索し、その繰り返しの内容について解析することにより、より正確な、ジョブ（プログラム）の演

算資源に対する必要性を明らかになる。

### 通信資源に重点がおかれる場合

通信資源に対するリソース要求が大きくなるのは大容量のデータの通信が行われるときとで頻繁な通信が行われるときである。データの通信では、ジョブを投げる管理ホストとジョブを受け取る実行ホストが存在する場合の管理ホストと実行ホスト間での通信とジョブを実行する実行ホストとデータの参照を行われる実行ホストが存在する場合の実行ホストと実行ホスト間での通信の 2 つの場合を考えることができる。管理ホストと実行ホスト間で行われる通信は投げられたジョブ（プログラム）とそれに対する結果であり、実行ホスト実行ホスト間で行われる通信はデータの通信である。これらの通信データを解析することにより、通信資源に対する必要性を明らかにしていく。

## 4-2. プログラムの構成

本実験で作成するプログラムは以下の図 4-1 のようになる。

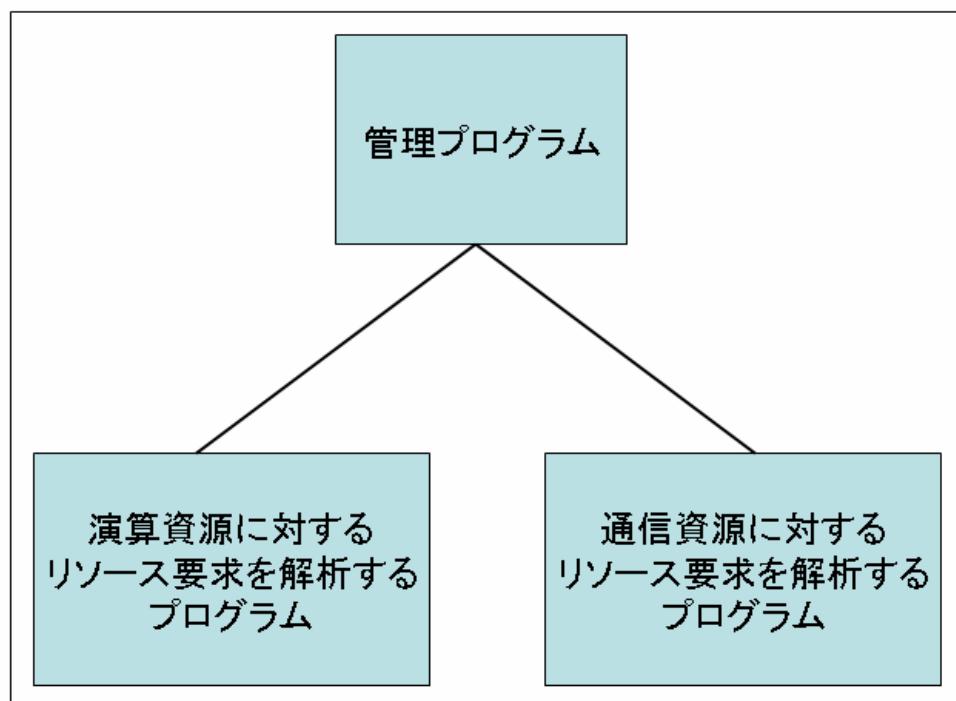


図 4-1 解析プログラムの構成

図4-1は「管理プログラム」の下に「演算資源に対するリソース要求を解析するプログラム」と「通信資源に対するリソース要求を解析するプログラム」が位置している構成になっている。「演算資源に対するリソース要求を解析するプログラム」は投げられたジョブの演算資源に対する必要性の情報を取得するためのプログラムであり、「通信資源に対するリソース要求を解析するプログラム」は投げられたジョブの通信資源に対する要求の必要性の情報を取得するためのプログラムである。これら2つのプログラムは投げられたジョブの解析のためのプログラムとなる。「管理プログラム」は下位層(2つのプログラム)から与えられた情報を管理するためのプログラムで、これらの管理された情報によりジョブを類型化の枠に収めていくためのプログラムである。このプログラムは投げられたジョブのパターン化のプログラムとなる。つまり、本実験で作成するプログラムの基本動作は、下位層の解析プログラムによって取得された情報を上位層に送ることにより、上位層のパターン化のためのプログラムでジョブの類型化を行うというものである。

### 4-3. 演算資源への必要性を解析するプログラムの説明

演算資源への必要性を解析するには3つの項目「繰り返しの有無」、「繰り返しの回数」、「繰り返し1回あたりの演算資源への必要性」について解析していく必要がある。

#### 繰り返しの有無

検索対象となる繰り返しfor文をスクリプトの先頭から検索する。for文のスコープ内に別のfor文が存在する場合は外側の繰り返しと内側の繰り返しを区別して分類を行う。

#### 繰り返しの回数

繰り返しの回数は、実際にスクリプトの実行時によって変化する可能性があるので、本実験のプログラムでは、最大となる繰り返し回数を求める。

#### 繰り返し1回あたりの演算資源への必要性

1つの命令を実行したときの演算処理能力が均一であるとすると、演算処理の回数は記述されている命令の個数に比例するので、命令の個数が多いほど演算資源への必要性が大きくなる。つまり、繰り返しのスコープ内に記述されている命令の個数をカウントしていくことにより、「繰り返し1回あたりの演算資源への必要性」を数値として求める。但し、繰り返しのスコープ内に別の繰り返ししか存在しない場合の繰り返しの必要性は基数1とする。これは繰り返しの大きさが、繰り返しの回数に比例しているからである。

実際に、演算資源への必要性の値として算出する数値は、「繰り返しの回数」と「繰り返

しの1回あたりの演算資源への必要性」の積となる。これは演算資源への必要性とスクリプトの演算量が比例しているからである。実際のプログラムではスクリプトの解析を行い、それから外側の繰り返しの「回数(out\_roop\_count)」と「1回あたりの演算資源への必要性(out\_count)」、内側の繰り返しの「回数(in\_roop\_count)」と「1回あたりの演算資源への必要性(in\_count)」を求める。

以上の3つの項目の解析の手順は図4-2のようになる。

但し、繰り返しのスコープ内に存在する繰り返しの検索は1回のみ行う。

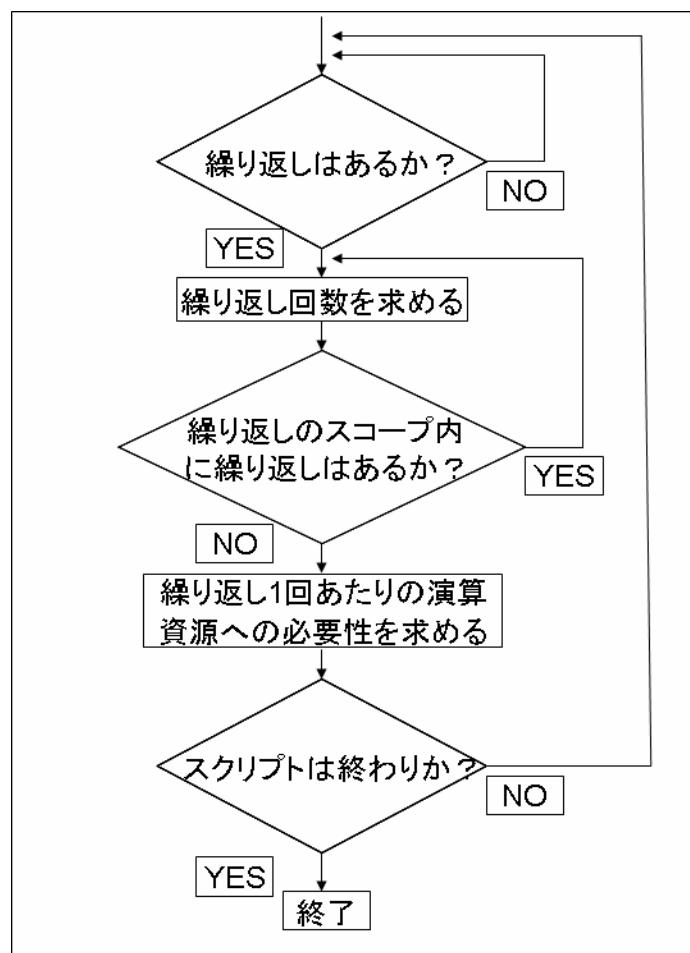


図4-2 演算資源への必要性を解析するプログラムの手順

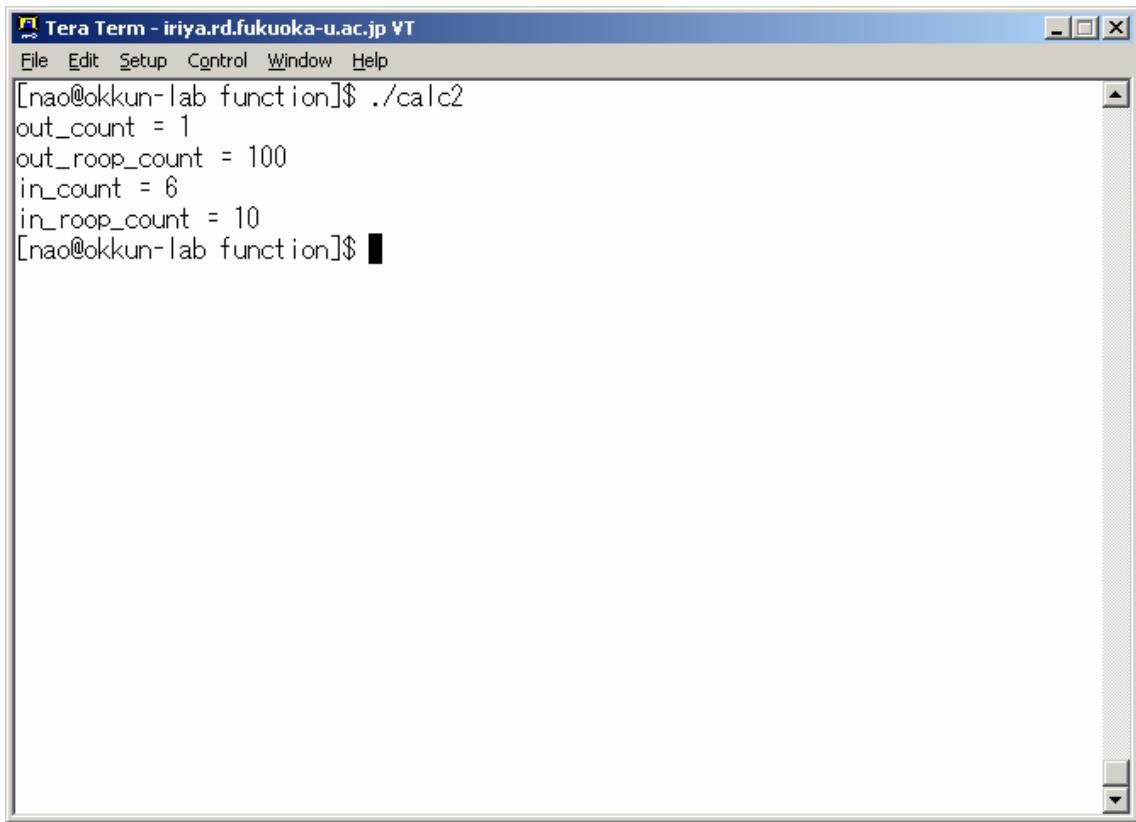
# 第5章

## 考察

この章では本実験において用いた理論、作成したプログラムから今後の方針を考えていく。

### 5-1. 演算資源への必要性の解析結果

実験による結果は以下のようになる。付録に記載されているスクリプト（test.pl）を解析プログラム（calc2.c）が読み込み演算資源への必要性の解析に必要な要素（各繰り返しにおける「繰り返し回数」と「繰り返し1回あたりの演算資源への必要性」）を出力する。実験画面は図5-1に示す。



The screenshot shows a terminal window titled "Tera Term - iriya.rd.fukuoka-u.ac.jp VT". The window has a menu bar with "File", "Edit", "Setup", "Control", "Window", and "Help". The main area displays the following text:

```
[nao@okkun-lab function]$ ./calc2
out_count = 1
out_roop_count = 100
in_count = 6
in_roop_count = 10
[nao@okkun-lab function]$ █
```

図5-1 実験画面

図5-1の実験画面に出力されているout\_countの値は1となっている。これは繰り返しの中に演算子が含まれていない状態である。out\_roop\_countの値は100となっている。

これは外側のループの繰り返しは最大 100 回あることを示している。次に、in\_count の値は 6 となっている。これは繰り返しの中に演算子が 5 つ含まれている状態である。in\_roop\_count の値は 10 となっている。これは内側のループの繰り返しは最大 10 回あることを示している。これを実際のソースプログラムと見比べると正確な解析が行われていることがわかる。

ここでは、本実験において作成した演算資源への必要性を解析するプログラムの評価を行っていく。この解析プログラムは文字列解析を行うことにより、解析対象となるプログラムの演算資源への必要性を見つけていくものである。そのため、命令の種類や命令の記述の幅によって、その解析手法も異なってくる。例えば、命令の種類が少なく、命令の記述の幅が狭い言語を解析する場合は比較的容易になる。しかし、一方で、命令の種類が多く、命令の記述の幅が広い言語であれば解析を行うことは難しくなる。本実験では作成時間や作成手法から、解析プログラムにある程度制限を設けて文字列解析を行っている。制限を設けた点は、「繰り返しの命令を for 文に限定」、「for 文の記述を一般的な記述形式に」、「for 文のスコープ範囲を中括弧の中に限定」などである。

## 5-2. 通信資源への必要性の解析方法

本実験においては通信資源への必要性について、解析プログラムを作成することができなかつたので、ここでは問題解決のための方法について言及していく。通信資源への必要性は、スクリプトの総通信量に依存していると考えることができる。総通信量は通信されるデータのサイズと、通信が行われる頻度によって求めることができる。このことを、管理ホストと実行ホスト間での通信と実行ホストと実行ホスト間の通信に当てはめてみる。まず、管理ホストと実行ホスト間での通信について考える。通信されるデータはスクリプトであるので、通信されるデータのサイズはスクリプトのサイズであり、通信が行われる頻度は 1 回である。よって、管理ホストと実行ホスト間の通信における通信資源への必要性は、スクリプトのサイズを調べることにより明らかになる。次に、実行ホストと実行ホスト間の通信について考える。通信されるデータは実行ホストに保存してあるデータであるので、通信されるデータのサイズは実行ホストに保存してあるデータのサイズであり、通信が行われる頻度はスクリプトに記述されている Perl の通信のための命令の個数である。よって、実行ホストと実行ホスト間の通信における通信資源への必要性は、通信のための命令とその命令によって通信されるデータのサイズを調べることにより明らかになる。しかし、ここで問題が発生する。それは、実行ホストに保存してあるデータのサイズを解析プログラムがどのようにして知るかである。解析プログラムはジョブとしてグリッド環境に投げられるスクリプトに対してのみ解析を行うので、実行ホストに保存してあるデータのサイズについては分からぬのである。このことを解決するには、実行ホストに

保存してあるデータのサイズを事前に解析プログラムが知っておかなければならぬので、通信資源への必要性を解析するプログラムには、各実行ホストに保存してあるデータのサイズを参照できるような仕組みが必要となる。

### 5-3. ジョブのパターン化の方法

本実験において、ジョブのパターン化を行う管理プログラムを作成することができなかったので、ここでは管理プログラムを作成するための方法について言及していく。管理プログラムの役割は「演算資源への必要性を解析するプログラム」と「通信資源への必要性を解析するプログラム」から与えられた数値をもとに、ジョブの類型化の枠に各ジョブを当てはめていくことである。ここで問題となるのは、類型化の枠の境界線の位置である。この境界線の位置によって、各ジョブは4つのパターンに分類されるので、この位置が大きすぎても小さすぎても、分類が偏ってしまう。そのため、この境界線の位置はある程度の統計結果から求められなくてはならない。つまり、初めは、境界線の値をある値に設定しておき、数回のジョブの分類を行った後に、適切な位置に境界線が近づくように境界線の値を修正する。この修正作業を繰り返していくことにより、境界線の値は統計的に適切なものへとなっていく。

## 第6章

### 結論

本研究における目的はグリッド環境に投入されたジョブが適切に割り当てられるよう  
に、グリッドに投入されたジョブの情報を取得し、分類を行うことである。ジョブの情報  
とはジョブの性質を表すもので、本研究では「演算資源へのリソース要求」と「通信資源  
へのリソース要求」という2つの項目から求め、これらの上位アプリケーションの「管  
理プログラム」で実際の分類を行っていく。しかし、本実験において、「通信資源へのリ  
ソース要求を求めるためのプログラム」、「管理プログラム」は作成することができなかつ  
た。また、作成した「演算資源へのリソース要求」と求めるためのプログラムも解析項目  
の欠落や各種制限のため、不完全なものとなった。このような結果になった理由としては、  
作成時間の不足や作成方法の誤りなどが挙げられる。作業方法の誤りは、文字列解析にお  
いて、解析を文字単位で行わずに、行単位で行ってしまったことである。

## 謝辞

本研究を進めるにあたって、日頃より多大なご指導、助言を頂きました福岡大学工学部電気工学科奥村勝助教授に深く感謝いたします。

## 参考資料

- [1] 溝口文雄、「グリッドコンピューティング 情報処理の新しい基盤技術」、岩波書店、2005
- [2] 日本アイビーエムシステムズエンジニアリング、「グリッドコンピューティングとは何か Globus Toolkit ではじめるグリッドの基礎」、SOFT BANK、2004
- [3] Ian Foster・Carl Kesselman、「The Grid : Blueprint for a new computing infrastructure second edition」、Morgan Kaufmann
- [4] 「The anatomy of the Grid」、  
<http://www.globus.org/alliance/publications/papers/anatomy.pdf>
- [5] 「PC クラスタにおけるジョブの管理」、  
<http://www.is.doshisha.ac.jp/SMPP/report/2000/000510yoshida.pdf>
- [6] 「Grid でつながる Linux HPC クラスタ」、  
<http://www.ylug.jp/download/ylug-kernel46.pdf>
- [7] 「Globus Toolkit による Grid 環境～応用例としての Access Grid」、  
[http://www2.nict.go.jp/dk/c216/Sgepss\\_data/murata/GRID.pdf](http://www2.nict.go.jp/dk/c216/Sgepss_data/murata/GRID.pdf)

## 付録

■ 演算資源への必要性を解析する (calc2.c) ■

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<math.h>
#include<stdlib.h>

int main()
{
    FILE *fp;          /*スクリプトファイルを格納*/
    char tool[256];   /*スクリプトファイルの1行を格納*/
    char c[2];
    int counter = 0;   /*配列 tool のカウンタ*/
    int num1, num2;   /*繰り返しの初期値と条件値を格納*/
    int out_count = 1; /*外側のループの演算子をカウント*/
    int in_count = 1;  /*内側のループの演算子をカウント*/
    int out_roop_count = 0; /*外側のループの回数をカウント*/
    int in_roop_count = 0; /*内側のループの回数をカウント*/
    int check_flag1, check_flag2; /*ループ条件の判断のためのフラグ*/

    if((fp = fopen("tet.pl","r")) == NULL) /*解析対象となるスクリプト*/
        exit(0);

    for(;fgets(tool, 256, fp)); { /*スクリプト全体の検索*/
        counter = 0;
        if((strstr(tool,"for")) != NULL) { /*外側の for 文を発見したときの処理*/
            while(!isdigit(tool[counter])) counter++; /*初期値の取得*/
            /*初期値の取得後、for文の内部処理*/
        }
    }
}
```

```

*c = tool[counter];
num1 = atoi(c);
counter++;
while(isdigit(tool[counter])) {
    num1 = num1 * 10;
    *c = tool[counter];
    num1 += atoi(c);
    counter++;
}
while(tool[counter] != '$')  counter++; /*条件値を取得*/
counter++;
counter++;
if(tool[counter] == '<')  check_flag1 = 1;
else  if(tool[counter] == '>')  check_flag1 = 2;
counter++;

*c = tool[counter];  /**
num2 = atoi(c);
counter++;
while(isdigit(tool[counter])) {
    num2 = num2 * 10;
    *c = tool[counter];
    num2 += atoi(c);
    counter++;
}
while(tool[counter] != '$')  counter++; /*繰り返しの変数の変化の条件を取得*/
counter++;
counter++;
if(tool[counter] == '+') {
    counter++;
    if(tool[counter] == '+')  check_flag2 = 1;
}
else  if(tool[counter] == '-') {
    counter++;
}

```

```

if(tool[counter] == '-')  check_flag2 = 2;
}

if(check_flag1 == 1 && check_flag2 == 1)      /*変数がある値未満のとき繰り返し
                                                条件*/
while(num2 - num1) {
    num1++;
    out_roop_count++;
}
else if(check_flag1 == 2 && check_flag2 == 2)  /*変数がある値より大きいときの
                                                繰り返し条件*/
while(num1 + num2) {
    num1--;
    out_roop_count++;
}
counter = 0;

if(strchr(tool,'{')) {          /*外側の for 文のスコープ範囲が続く場合の処理*/
for(;fgets(tool, 256, fp);) {    /*外側のループ内の検索*/
    counter = 0;
    if((strstr(tool,"for")) != NULL) { /*内側の for 文を発見したときの処理*/
        while(!isdigit(tool[counter]))  counter++; /*初期値の取得*/
        *c = tool[counter];
        num1 = atoi(c);
        counter++;
        while(isdigit(tool[counter])) {
            num1 = num1 * 10;
            *c = tool[counter];
            num1 += atoi(c);
            counter++;
        }
    }
    while(tool[counter] != '$')  counter++; /*繰り返しの条件を取得*/
    counter++;
    counter++;
}

```

```

if(tool[counter] == '<')  check_flag1 = 1;

else  if(tool[counter] == '>')  check_flag1 = 2; /*条件値を取得*/
counter++;
*c = tool[counter];
num2 = atoi(c);
counter++;
while(isdigit(tool[counter])) {
    num2 = num2 * 10;
    *c = tool[counter];
    num2 += atoi(c);
    counter++;
    while(isdigit(tool[counter])) {
        num2 = num2 * 10;
        *c = tool[counter];
        num2 += atoi(c);
        counter++;
    }
}

while(tool[counter] != '$')  counter++; /*繰り返しの変数の条件を取得*/
counter++;
counter++;
if(tool[counter] == '+') {
    counter++;
    if(tool[counter] == '+')  check_flag2 = 1;
}
else  if(tool[counter] == '-') {
    counter++;
    if(tool[counter] == '-')  check_flag2 = 2;
}

if(check_flag1 == 1 && check_flag2 == 1) /*変数がある値未満の時の繰り
                                         返し条件*/
while(num2 - num1) {
    num1++;
    in_roop_count++;
}

```

```

    }

else if(check_flag1 == 2 && check_flag2 == 2) /*変数がある値より大きいときの繰り返し条件*/
{
    while(num1 + num2) {
        num1--;
        in_roop_count++;
    }
    counter = 0;

    if(strchr(tool,'{')) { /*内側のループ内の検索*/
        for(;fgets(tool, 256, fp);) {
            counter = 0;
            while(tool[counter] != '\0') { /*内側のループのカウント*/
                if(tool[counter] == '+') in_count++;
                if(tool[counter] == '-') in_count++;
                if(tool[counter] == '*') in_count++;
                if(tool[counter] == '/') in_count++;
                if(tool[counter] == '%') in_count++;
                if(tool[counter] == '}') break;
                counter++;
            }
            if(tool[counter] == '}') break;
        }
    }
}

else {
    while(tool[counter] != '\0') { /*外側のループ内のカウント*/
        if(tool[counter] == '+') out_count++;
        if(tool[counter] == '-') out_count++;
        if(tool[counter] == '*') out_count++;
        if(tool[counter] == '/') out_count++;
        if(tool[counter] == '%') out_count++;
        if(tool[counter] == '}') break;
        counter++;
    }
    if(tool[counter] == '}') break;
}

```

```
        }
    }
}

printf("out_count = %d\n",out_count); /*外側のループの演算子の個数を出力*/
printf("out_roop_count = %d\n",out_roop_count); /*外側のループの個数を出力*/
out_count = 1;
out_roop_count = 0;

if(in_count != 1) {
    printf("in_count = %d\n",in_count); /*内側のループの演算子の個数を出力*/
    printf("in_roop_count = %d\n",in_roop_count); /*内側のループの個数を出力*/
    in_count = 1;
    in_roop_count = 0;
}
}
```

■ 与えられた値の和、減、積、商、余りを出力する (test.pl) ■

```
# test.pl
```

```
for($j=0; $j<100; $j++) {  
    for($i=10; $i>0; $i--) {  
        $a = $j + $i;  
        $b = $j - $i;  
        $c = $j * $i;  
        $d = $j / $i;  
        $e = $j % $i;  
  
        print "a = $a\n";  
        print "b = $b\n";  
        print "c = $c\n";  
        print "d = $d\n";  
        print "e = $e\n";  
    }  
}
```