グリッド計算環境でのマスターワーカシステムの構築

谷 村 勇 輔† 廣 安 知 之†† 三 木 光 範††

本論文では，グリッド計算に適したモデルを検討しながらアプリケーションを開発でき，実行するためのシステム「Grid Master Worker System（GMWS）」を提案する．GMWS はマスターワーカ型の構成をとる．これはアプリケーションの実装モデルと切り離すことが可能であり，既存のグリッドシステムでは構築しにくい並列モデルの記述が容易である．さらに，グリッド環境の動的変化に対応したアルゴリズムの記述を支援する．これらの特徴により，アプリケーションレベルでグリッド環境の特性に合ったモデルを検討することが可能となる．GMWS は，Globus および SSH で構築されるグリッド環境において動作する．アプリケーション開発のために MPI の送受信関数に似た API が提供され，これを利用してデータ通信や計算資源の情報取得が行われる．

Development of Master-Worker System for The Computational Grid

Yusuke Tanimura,† Tomoyuki Hiroyasu†† and Mitsunori Miki††

In this paper, “Grid Master Worker System (GMWS)” is proposed. Using GMWS, users can develop applications that fit to the Grid environment. GMWS has a convetional Master-Worker architecture. This system architecture can be separated from execution model of application. Therefore, users can develop not only a master-slave model but also other models. It is also possible to construct an algorithm that takes care of the dynamic changes of the Grid resources at application side. GMWS works on the Grid computing environment that is built with Globus or SSH. It has two types of simple programming interfaces. One of them is API that is similar to send/recv function of MPI for data communication. The other is API for gathering remote resource information.

1. はじめに

グリッド技術は広域に存在する計算資源や情報資源を結びつけて，分散･並列計算を行うことにより，従来存在しなかったサービスを提供する．なかでもグリッド計算技術は，増大しつつある大規模計算の要求を解決する技術の 1 つとして注目を集めている[5)]．近年のコンピュータ技術，ネットワーク技術の進歩とグリッドの基礎研究の成果により，グリッド研究は実際のアプリケーションを動かして，それらの基盤システムやアプリケーションモデルを検証する段階にきている．

グリッド環境において，計算アプリケーションの開発に利用できるシステムは，いくつかあげられる．Ninf[11)]や NetSolve[3)]に代表される GridRPC[12)]に基づくシステムは，遠隔に存在する計算サービスを簡単なインタフェースで呼び出すことのできるクライアント・サーバ型のシステムである．Condor[15)]は LAN ベースの計算クラスタのために開発されたジョブ実行システムであり，Globus Toolkit[6)]と連係することでグリッド環境にも対応している．Globus を利用したツールでは，MPI 通信ライブラリを拡張した MPICH-G2[9)]があり，従来の MPI プログラムをそのままグリッド上で実行する仕組みを提供している．しかしながら，MPICH-G2 以外のシステムでは，パラメータ探索型やマスタースレーブ型の計算モデルを主に実行することが想定され，それ以外のモデルの開発にはあまり適さない．これは，アプリケーション開発者にとって，利用できるアプリケーションのモデルが限定されるという問題となっている．また，システム側での情報をアプリケーションに提供することはない．そのため，グリッド環境の特性を考慮して，アプリケーション開発者がそれに適した分散・並列計算モデルを構築することは困難である．

本研究では，グリッド環境を考慮したアプリケーションをマスターワーカ形式で記述し，作成できるよう，開発・実行環境からなる基盤システムを構築する．本

† 同志社大学大学院工学研究科
Graduate School, Doshisha University
†† 同志社大学工学部
Department of Engineering, Doshisha University

論文では，主となる計算を実行するワーカとそれら全体を管理するマスターからシステムが構成される場合，それをマスターワーカ型のシステムと呼ぶこととする．構築するシステムでは，マスターおいてアプリケーションが利用可能な資源情報を一元管理する．そして，マスター用のアプリケーション・プログラムの中に計算環境の変化に対する操作をあらかじめ記述しておくことで，各資源に効率良くサブ計算を分配したり，サブ計算プロセス間のデータ交換のための通信を柔軟に行うことが可能となる．マスターワーカ形式のプログラムは，その構成に沿ってパラメータ探索やマスタースレーブモデルを記述できる．一方，適切なマスタープログラムを用意することで，その他の並列モデルを記述することも可能である．

本論文では，グリッド計算に適したアプリケーション・モデルを定め，それを実現できるシステムの枠組みを提示する．それに基づいて開発した提案システムについて詳細を述べる．システムの基本通信性能を測り，さらに提案システムを利用したアプリケーション例として，遺伝的アルゴリズム[7]の並列モデルを実装する．これらを通して，システムの有効性について確認する．

2. グリッド計算に適したアプリケーション

グリッド計算を行う動機はいくつか考えられる．主なものとして，利用者が手元に持っている計算資源では短時間に終えることのできない膨大な計算を行いたいとき，あるいは他者が開発した優れた数値計算ライブラリやデータベースを遠隔から利用しながら計算を行いたいときなどである．しかしながら，グリッド計算環境が有する特徴のため，すべてのアプリケーションがグリッド環境に適しているわけではない．そこで本章では，グリッド計算に適したアプリケーションのモデルを GOCA（Grid Oriented Computing Application model）として定め，GOCA をスムーズに実装するための機能をあげる．

2.1 Grid Oriented Computing Application model（GOCA）

グリッド環境がアプリケーションのモデルに及ぼす特性としては，利用できる資源のアーキテクチャやパフォーマンスがヘテロであること，利用できる資源量やその処理能力は主に他ユーザの影響によって動的に変化すること，各資源を結びつけるネットワークの性能は平均的に低いことが予想されることがあげられる．また多くの組織，個人から提供される資源を利用するために，長時間の実行を行う際，ユーザの意図とは関係なくシステムが停止したり，障害に陥ったりする可能性もある．これらは，従来の並列計算の研究ではあまり検討されてこなかった特性であり，検討内容もシステムのレベルで特性を吸収する方法であった．本論文では，アプリケーションのレベルでそれらの特性に対応するべく，グリッドの特性に対応して，計算性能をある程度維持でき，資源を有効に利用できるアプリケーションのモデルを GOCA と定義する．そして，GOCA を満たすアプリケーションは，単位時間あたりの計算効率だけでなく，単位時間あたりの計算量など多角的な視点から評価できるものとする．以下に GOCA の定義を示す．

- 仕事は複数に分割できる．分割サイズは十分に大きく，資源の状況に応じて柔軟に変更できる．
- 分割された仕事は独立に実行可能であるか，互いに依存関係が少ない．
- 分割された仕事が並列実行可能である場合，互いに同期をとる必要性が少なくデータ交換量も少ない．
- 資源の状況が変わり，ある資源が突然利用できなくなっても，低コストで仕事を継続できる．
- 資源の状況が変わり，新しい資源が突然加わっても，それを処理中の仕事のために取り込んで，有効に利用できる．

2.2 GOCA を実現するシステムの要件

既存のグリッドシステムでは，グリッド環境の特性をシステム内部の機構で吸収する場合が多い．これはグリッド技術のシステムレイヤについて詳細を知らないアプリケーション開発者が簡単にグリッド計算を行うためである．しかしながら，GOCA を実現するには，システムが取得したグリッド環境に関する情報をアプリケーションからアクセスし，遠隔で実行を開始したアプリケーションに動的に指示を行える仕組みが必要といえる．これをふまえて，GOCA を実現するシステム要件を以下のようにまとめた．

- アプリケーションレベルで，グリッド資源の情報を取得できる仕組みを提供する．
- 障害が発生し，利用できなくなった資源で動作していたアプリケーション・プロセスのリカバリをアプリケーション側で制御できる仕組みを提供する．
- 実行を開始したアプリケーション・プロセスに対して，動的なパラメータの変更やモデルの変更を通知できる仕組みを提供する．

上記に加えて，一般的なグリッド・システムに要求される項目として，簡易インタフェース，セキュリティ，信頼性，パフォーマンスがあげられる．

2.3 GOCA の一例としての遺伝的アルゴリズム

遺伝的アルゴリズム（Genetic Alogrithm，GA）は，生物の進化と淘汰を模倣した最適化手法である[7]．GA は従来の厳密に解が保証されている手法では解くことが困難である大規模，あるいは複雑な最適化問題に対して，ある限られた時間の中で満足解を提供することができる．GA の欠点は，反復処理が多く，試行回数が必要であったりと膨大な計算量を必要としたりすることであり，その解決が望まれている．

一方，GA において最も計算負荷の高い部分は通常，個体の評価計算であり，これを並列分散処理する方法は複数考えられる[2]．特に，GA は独立性の高い多点探索を行うので，頻繁に探索点情報を交換しない，あるいは探索点を動的に増減できる仕組みを考えることで，GOCA の多くの性質を満たせる可能性がある．

本論文では，提案システムのアプリケーション例として GA を用いる．5 章では，GA の 2 つの並列モデルを取り上げ，プログラム作成と実行の様子を示す．

3. Grid Master Worker System（GMWS）

3.1 設計目標

Grid Master Worker System（GMWS）はマスターワーカ型で構成され，グリッド環境の特性を生かせるよう GOCA を満たすアプリケーションを開発したり，GOCA をすでに満たしたアプリケーションを実装したりするための枠組みを提供するシステムである．以下にその設計目標を示す．

- マスターワーカ形式でアプリケーションを記述する仕組みを提供する．
- グリッド資源の情報に簡単にアクセスできるインタフェースを提供する．
- 実行を開始したアプリケーションに対して，実行の停止，再開，パラメータやモデルの変更などの命令を簡単に出せるインタフェースを提供する．
- 障害が発生しても，システム全体が停止しない仕組みを持つ．アプリケーションから停止した計算機を特定でき，計算全体を止めないモデルの構築が可能なインタフェースを提供する．
- MPI の送受信関数に似たデータ通信のための API を提供する．ただし，実際の資源間の通信はシステム内部で行われ，送受信関数ではローカル資源内でデータを読み書きするだけである．
- 資源間通信に Globus または SSH を利用でき，そのいずれかで構築されたグリッド環境で動作する．

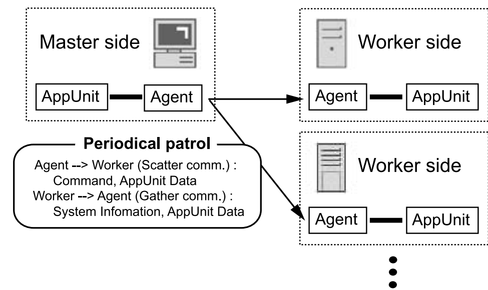

図 1 GMWS の基本アーキテクチャ
Fig. 1 Basic architecture of GMWS.

```
# Grid map to use for GMWS

@WORKER_COUNT  8

@HOSTNAME        host1.university1.ac.xxx
@DISABLE         no
@GLOBUS_ROOT     /usr/local/globus
@GOCAES_ROOT     /home/foo/gmws
@EXECUTABLE      /home/foo/gmws/bin/workerAppUnit
@ARG_COUNT       2
@ARGS            -f 10
        ⋮
```

図 2 利用資源を記述するマップファイル
Fig. 2 Mapfile to describe resources that user is going to use.

3.2 システム構成と動作

GMWS は，**図 1** に示すように，システム層からマスターワーカの構成をとる．それぞれの計算資源においてデーモンとして常駐し，GMWS アプリケーションの実行をサポートするのが Agent であり，唯一のマスター Agent と，それによって管理されるワーカ Agent に分けられる．マスター Agent の役割は，その下で働くワーカ Agent の起動，ワーカ Agent へのジョブの割当て，ワーカ間の通信の仲介である．ワーカ Agent はマスター Agent の指示に従って動作し，ワーカ Agent どうしはお互いを知らず，通信も行わない．

GMWS はアプリケーション開発のために，GMWS ライブラリを提供する．これを利用して作成されたプログラムが GMWS 上で動作し，これを AppUnit と呼ぶ．AppUnit は Agent と通信し，Agent どうしのネットワークを借りる形で，遠隔の AppUnit と連係する．以下に GMWS の動作を示す．

Agent の起動　ユーザはアプリケーションを実行する前に，Agent を起動しておく．GMWS のマスター Agent を起動する際は，**図 2** に記すような，利用資源の情報が記述されたファイルを与えることで，該当する資源においてワーカ Agent も起動される．ワーカ Agent はマスター Agent に接

続し，アプリケーション・プログラムの実行環境が整う．

AppUnit の起動　ユーザは，マスター AppUnit を起動することで計算を開始する．プログラム内で指定された数だけ，自動的にワーカ AppUnit が起動され，計算が始まる．ワーカ AppUnit のバイナリは，あらかじめワーカとなる計算機上に展開されていなければならない．

Agent と AppUnit 間の通信　AppUnit は Agent と通信し，送りたいデータを Agent に渡し，実際の通信を依頼する．一方，受け取るはずのデータを Agent に問い合わせる．通信データはプログラマによって指定されたタグによって特定される．

Agent による定期巡回機構　マスター Agent は，マスター AppUnit のプログラム内で指定された時間間隔で，ワーカ Agent と通信する．この通信は，ワーカ Agent からデータを回収する Gather 通信とワーカ Agent にデータを配布する Scatter 通信の 2 種類が存在し，交互に行われる．AppUnit のためのデータ通信のほか，Gather 通信ではワーカ側のシステム情報がマスター Agent に集められ，Scatter 通信ではマスター側からの指令がワーカ Agent に伝達される．

システム情報の取得　AppUnit は，GMWS によって予約されたタグで受信関数を記述することにより，システム情報を取得できる．マスター AppUnit は全資源の情報を取得でき，ワーカ AppUnit はそれ自身が動く資源の情報を取得できる．

障害の検出　障害によりワーカ側の資源が利用できなくなった場合，マスター Agent による定期巡回では，該当するワーカ Agent との通信がタイムアウトになる．システム全体は停止せず，アプリケーション・プログラム内で対応を記述することになる．

AppUnit の終了　マスター AppUnit 側で終了関数が呼ばれると，すべてのマスター AppUnit が終了した後，マスター AppUnit も終了する．

Agent の終了　専用の Halt コマンドを利用して，すべての Agent を終了する．

3.3 GMWS の実装とセキュリティ

GMWS は C 言語で記述されたシステムであり，Globus，あるいは SSH [4] を利用している．これらは，マスター Agent からのワーカ Agent の起動，各計算資源間の通信に利用される．ユーザは，利用するグリッド環境に合わせて，適切なオプションでコンパイルを行うことによりこれらを選択する．

SSH を利用したシステムでは，ssh コマンドを利用して遠隔にてジョブを実行する．資源間の通信は，SSH のポート・フォワーディングを行う代理ユニットを遠隔で起動し，それを介して通信を行う．この機能は，SSH アクセスだけに制限されたシステムからなるグリッド環境において，認証，権限および通信の暗号化を付加する．

Globus を利用したシステムでは，SSH を利用したものより強固なセキュリティ機構を持つ．ジョブの起動に関しては GRAM を利用し，資源間の通信は Globus I/O を利用する．X509 の公開鍵の仕組みを利用して，ジョブの実行，通信接続の確立の際にユーザ認証を行う．GMWS のユーザ権限はそれを実行するユーザに制限され，通信データは SSL で暗号化される．

3.4 GMWS API

アプリケーション開発は，GMWS ライブラリを利用する．これは AppUnit の起動や終了に関する制御関数，利用資源の情報を取得する関数，遠隔の AppUnit とデータ通信を行う関数からなる．アプリケーション・プログラムは，マスター AppUnit とワーカ AppUnit の両方を用意することになる．いくつかの関数はマスター用とワーカ用に分けられる．

3.4.1 制御関数

主な制御関数は以下のとおりである．

- int GMWS_init()
  AppUnit の初期化処理を行う．
- int GMWS_fin()
  AppUnit の終了処理を行う．
- int GMWS_registToAgent()
  自身を Agent に登録し，計算の開始を通知する．
- int GMWS_m_spawnAppUnit()
  マスター AppUnit 用の関数である．マスター Agent を介して，遠隔でワーカ AppUnit を起動する．
- int GMWS_m_haltAppUnit()
  マスター AppUnit 用の関数である．遠隔で実行しているワーカ AppUnit を終了する．

3.4.2 通信関数

GMWS の通信関数は MPI の Send()，Recv() 関数ライクなインタフェースを持ち，1 対 1 の同期通信や非同期通信を記述できる．ただし，実際の通信は Agent によって行われるため，GMWS の送受信関数は Agent に届いているデータをポーリングして読み出したり，送信されるデータを Agent 側のメモリ空間に書き出したりするためのものである．アプリケーション・プログラムからは通信関数に見えるので，本論文では

これらを送受信関数と読んでいる．以下に，主なワーカ AppUnit 用の通信関数を示す．マスター AppUnit では，GMWS_m_writeInit（int hostId）のようになり，各関数の引数に hostId を追加指定する．これは，ワーカ AppUnit では通信相手がマスター AppUnit のみなのに対し，マスター AppUnit では複数のワーカ AppUnit と通信することになるので，どのワーカ AppUnit のデータを扱うかを明確にする必要があるためである．これらの仕組みを利用して，GMWS では，プログラマが定期巡回機構に合わせて，非同期であり，かつ動的変化に対応できる通信パターンを記述することが可能となる．

- int GMWS_w_writeInit()
  通信データを Agent に送信するための初期化処理を行う．この関数と GMWS_w_writeFin() の間に呼び出された GMWS_w_write() で送られたデータが，ひとまとまりとして扱われ，他の Agent に送られる．
- int GMWS_w_write(char tag,...)
  通信データを実際に Agent に送る．通信データの属性として，タグ，データ型，データの個数を与える．タグによって，どのデータであるか識別するため，受信関数は同じタグを与えて呼び出す必要がある．
- int GMWS_w_writeFin()
  通信データの送信を終了する．これが呼ばれるまで，データは他の Agent に送られる対象とならない．
- int GMWS_w_readInit()
  通信データを Agent から受信するための初期化関数であり，Agent に新しいデータ届いているかを確認する関数でもある．返り値に，Agent にまったくデータがない，受信済みのデータしかない，新しいデータが届いているなどを表す値が設定される．アプリケーション・プログラマは，これを利用して，必要に応じて受信イベントをポーリングする記述をすることになる．
- int GMWS_w_read(char tag,...)
  通信データを実際に Agent から受信する．通信データの属性として，タグ，データ型，データの個数を与える．該当データがない場合は，返り値にエラーが設定される．
- int GMWS_w_readFin()
  通信データの受信を終了する．GMWS_w_readInit() を呼び出して，新しいデータが届いていた場合，この関数が呼ばれるまで次の新しいデータを受信することはない．

3.4.3　システム情報の取得関数

現在利用している計算資源の情報は，通信関数のデータ・タグに以下を設定することで取得できる．マスター AppUnit では，すべてのホストの情報を参照できるため，引数にホスト ID も指定する．以下は主な予約データタグである．

- LOADAVG1：過去 1 分間の CPU の平均負荷（ロードアベレージ）を取得する．
- AVAIL_MEMORY：利用可能なメモリ量 [MBytes] を取得する．
- TOP_PROCESS：CPU 使用率が高いプロセス名を取得する．
- TOP_USER：CPU 使用率が高いプロセスを実行しているユーザ名を取得する．
- LATENCY：マスター Agent からワーカ Agent に問合せを開始してから，実際にデータが取得されるまでの遅延時間 [sec] を取得する．

4.　GMWS の基本性能の評価

GMWS の基本性能を評価するために，以下に記す実験環境を構築し，Agent どうしの通信性能を計測する．

4.1　実 験 環 境

実験では，ある 1 台の計算機からインターネットを介して遠隔に存在する計算機群を利用する．手元にある計算機がマスターとなり，遠隔の計算機群がワーカとなる．計算機とネットワークの構成を**図 3** に示し，実験前後のネットワーク・スループットを**表 1** に示す．ネットワーク・スループットは Netperf ベンチマーク[1]を利用して，1KB のメッセージを送りあう TCP Stream 性能を計測した．SSH は Open SSH の

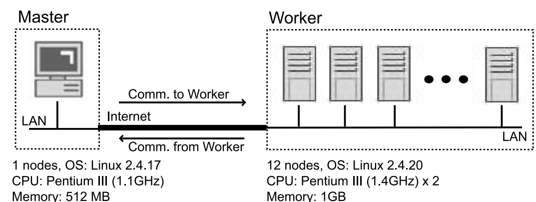

図 3　実験に用いた計算機とネットワーク構成
Fig. 3　System specification of the grid environment used in the experiments.

表 1　サイト間のスループット
Table 1　Througput between Master and Worker sites.

Network	TCP Stream performance
Master → Worker	763 [Kbps]
Worker ← Master	759 [Kbps]

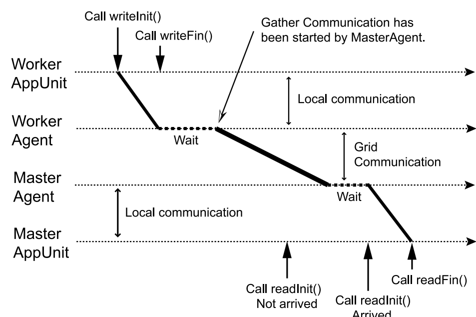

図 4 GMWS の通信遅延の内訳
Fig. 4 Details of the GMWS communication delay.

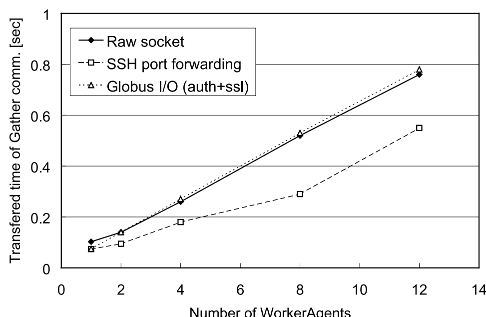

図 5 Gather 通信の計測結果
Fig. 5 Transfered time of Gather communication performance.

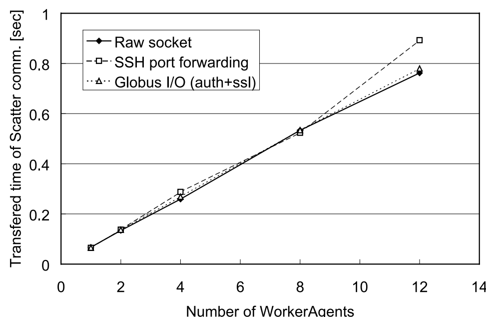

図 6 Scatter 通信の計測結果
Fig. 6 Transfered time of Scatter communication performance.

Protocol Version 2 を利用して，対話的なパスフレーズの入力をすることなくログインおよび，RPC コマンドを実行できるように設定した．Globus はマスター，ワーカとも Version 2.4.2 を利用した．

4.2 実 験 方 法

GMWS では，ワーカ AppUnit から送られた情報がどれくらいの遅延でマスター AppUnit に届くか，あるいはその逆の通信遅延が性能を大きく決定する．これは，**図 4** に示すように，Agent と AppUnit 間の通信，Agent どうしの通信，そして通信関数が呼び出されてから定期巡回までの待ち時間の合計となる．Agent と AppUnit 間の通信は通常，同一資源内であるため遅延は小さい．定期巡回までの待ち時間は，アプリケーション・プログラム内で設定する定期巡回間隔に依存する．しかし，これはタイミングの問題であり，待ち時間が小さくなるよう設定することは難しい．そこで，Agent 間の通信性能を GMWS の基本性能を測る指標とする．

本実験では，マスター Agent において，Gather 通信と Scatter 通信に要した時間をそれぞれ測定した．これは，マスターがすべてのワーカにひととおりアクセスして情報を得る，あるいは送るという操作を完了するまでの時間である．本来，Scatter() 通信は送信関数の呼び出しのみであり，実際にデータが送られることを保証しない．しかし，本実験では，ワーカ側がデータの受信を完了した後に ACK をマスターに送り，Scatter() 関数が返る仕組みを特別に用意して実験を行った．その他，通信の相手側にすでに送信した情報がある場合には，実際にデータが送信されない仕組みになっている．これは，マスターやワーカの状況に応じて，通信結果が異なるということである．そこで本実験では，複数試行の結果から最も大きい値を計測値とし，これが全ワーカ Agent との通信において，実際にデータ転送が必要であったことを確認した．マスター Agent からワーカ Agent に送信するデータは倍精度浮動小数点（Double）型で 100 変数，受信するデータは Double 型で 1 変数とした．この測定は単純なソケット通信，SSH のポートフォワーディングによる通信，Globus I/O を利用した通信の 3 種類で行った．

4.3 実 験 結 果

図 5 に Gather 通信，**図 6** に Scatter 通信の結果を示す．Scatter 通信では，Gather 通信に比べて実際のデータ転送量が大きいにもかかわらず，そのままのソケット通信と GlobusI/O を利用した通信では，結果に違いが見られなかった．これは，通信を確立する際の Globus の認証機構や通信時の暗号化のオーバヘッドが小さいことを示している．

それに対して，SSH のポート・フォワーディングを利用した仕組みは，Gather 通信では通常のソケット通信より性能が良く，Scatter 通信では他とほぼ同等の性能か逆に悪い性能となっている．この理由は，SSH を利用した通信が代理ユニットを経由してデータを転送しているためであると考える．代理ユニットは，Agent と独立に動作し，受信 Agent が別の処理をしている間に送信 Agent からデータを受信でき，代

理ユニットはその後，受信 Agent と同一ホスト内の通信をすればよいからである．しかし，転送するデータ量が大きくなった Scatter 通信では，代理ユニットのバッファに収まらないため，転送速度が他と同じ程度となる．

提案システムでは，マスター Agent とワーカ Agent の接続は最初から最後まで維持され，SSH や Globus の認証は接続の際に 1 度だけ行われる．これが，認証によるオーバヘッドを小さく抑えている原因である．また，実験に用いたデータ量程度では，暗号処理による通信速度の低下も小さいといえる．結果として，Gather および Scatter 通信の実行時間は，ワーカ数に比例して伸びている．これについては，今後，より多くのワーカを用いた調査が必要である．

5. GMWS を用いた並列遺伝的アルゴリズムの実装と実行

本章では，GMWS を利用したプログラム例として並列 GA を取り上げ，マスタースレーブモデルと島モデルを実装する方法を示し，それらの実行結果の比較を示す．さらに，島モデル GA の動的な計算モデルとその効果例を紹介する．

5.1 マスタースレーブモデル GA の実装

典型的な GA のマスタースレーブモデルでは，個体評価を分散・並列処理する方法がとられる[10]．これはマスターワーカの構成に沿って，実装すればよい．たとえば 1 個体ずつ処理する場合，まずその個体の遺伝子配列を以下のようにワーカに分配し，結果が返ってくるのを待つ．

```
count = 0;
for(i=0;i<NUM_WORKERS;i++){
  GMWS_m_writeInit(i);
  GMWS_m_write(i,indiv[count].gene_tag,
    gene_length,GMWS_CHAR,indiv[count].gene)
  GMWS_m_writeFin(i);
  submit_list[i] = count;
  count++;
}
```

そして，結果を返してきたワーカに対して，新しい個体の評価計算を割り付ける処理を以下のように記す．ワーカ側のプログラムは，与えられたデータを受け取り，結果を返す単純な記述になる．

```
finished = 0;
while(finished < num_indiv){
  for(i=0; i<NUM_WORKERS; i++){
    if(GMWS_m_readInit(i)!=GMWS_PROT_UPDATED)
      continue;
    GMWS_m_readInit(i,result_tag,1,
      GMWS_DOBULE,&result)
    GMWS_m_readFin(i);
    indiv[submit_list[i]].fitness=result;
    finished++;
    if(count<num_indiv) { 送信処理を記述 }
}
```

5.2 島モデル GA の実装

島モデルは，母集団と呼ばれる GA の個体集合を複数に分割し，サブ母集団（島）ごとに GA の操作を適用する手法である．各島は独立に探索を進めるが，適切な間隔でいくつかの個体を交換する．これは移住と呼ばれる操作である．島モデルを採用することにより，全体として多様性を維持して探索が行えるようになり，探索性能が高くなる[14]．

一般に，島モデルでは並列に動くプロセスの 1 つが 1 つの島に割り当てられる．そこで，GMWS を用いた実装では各ワーカに 1 つの島を担当させる．グリッド環境を考慮すると，高速なワーカに対してより多くの計算をさせ，障害の発生したワーカは切り離し，ある時点において最良のトポロジを動的に作成していく必要がある．GMWS では，ワーカ間通信をマスターで管理できるため，たとえば次のように実装できる．

マスターは，あるタイミングで以下を実行し，ワーカから届いている情報が更新されているか確認し，更新されていたワーカだけでトポロジを作成する．dest_list[] 配列には，各ワーカの個体送信先が記される．トポロジに参加しないワーカに対しては，送信不要を示す値が設定される．

```
for(i=0;i<NUM_WORKERS;i++){
  if(GMWS_m_readInit(i)!=GMWS_PROT_UPDATED)
    continue;
  active_list[count]=i;
  count++;
}
create_topology(active_list,count,dest_list);
```

そして，dest_list[] 配列に合わせて個体情報を複製し，最終的に GMWS_m_writeFin() 関数を呼ぶことで，それぞれのワーカに新しい個体を送信する手続きをとる．

```
for(i=0;i<NUM_WORKERS;i++){
  src=i; dest=dest_list[i];
  GMWS_m_read(src,gene_tag,gene_length,
    GMWS_CHAR,gene);
  GMWS_m_write(dest,gene_tag,gene_length,
    GMWS_CHAR,gene);
}
```

ワーカ側では，交換対象となる個体をマスターに送り，マスターから受け取った個体を母集団に組み込む操作を記述する．交換がどの島となされるかは，ワーカのプログラムでは考慮しないことになる．

5.3 実行結果と並列化効率

マスタースレーブモデル GA と島モデル GA を実

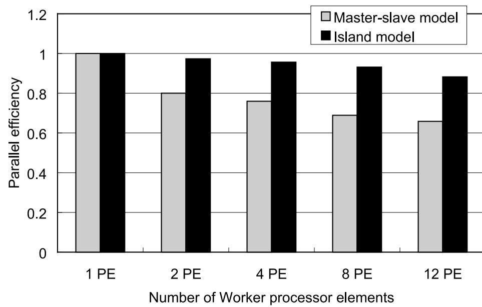

図 7 並列 GA の計算結果
Fig. 7 Parallel efficiency of parallel GAs implemented with GMWS.

行し，その並列化効率を**図 7** に示す．実験環境は 4.1 節と同様のものを用い，通信には SSH のポート・フォワーディング方式を利用した．島モデルは並列数に依存して探索性能が変化するため，図 7 では時間経過に対する平均世代数から並列化効率を求めた．すなわち，並列化効率の式は，単一母集団の GA を逐次実行したときの単位時間あたりの処理世代数が分子になり，各モデルにおける単位時間あたりの処理世代数と PE 数を掛け合わせたものが分母となる．また，1 個体ずつ評価計算を分配するマスタースレーブモデルは，より粒度の大きい計算を要求することから，1 個体あたり平均 10 秒，島モデルでは平均 0.48 秒の計算を設定した．

島モデルは高い効率を維持できているのに対し，マスタースレーブモデルはそうではない．これは，図 4 に示したように，通信データは Agent によって転送されるのを待つ必要があり，低いレイテンシで転送されないからである．しかし，前章の結果から，通信データサイズの違いがわずかであることが分かっているため，個体数をまとめて送ることで通信頻度を減らし，この効率の低下を抑えることができると考えられる．

5.4 動的にモデルを変更する例

GMWS を利用すると，システムが提供する情報を利用して，動的な計算モデルを作成できる．たとえば，島モデル GA は 5.2 節で示したように実装を行うことができるが，GA を実行しているワーカの計算機で，他のユーザが別の計算を開始したら，島を消去して資源を解放する操作を加えることができる．これは，島モデルではいくつかの島の探索が遅くなると，移住時に遅い島の個体が早い島に移住して，解への収束を過度に遅らせる作用があるためである．資源を解放すれば，他のユーザはその資源を占有することもできる．

GMWS では，その API を用いてワーカのロードアベレージを取得できる．1 つのプロセスが実行されていればロードアベレージ値は 1.0 程度になり，2 つのプロセスが実行されていれば 2.0 程度になる．これを利用して，本論文では，ワーカの 1 分間のロードアベレージが 1.5 を超えた時点で，マスターから該当のワーカに通知を行い，島を消去して GA を停止する島モデル GA を作成し，先の実験環境で実験を行った．GA で解く対象問題は，テスト関数の 1 つである Ridge 関数の最小化であり，10 次元のものを用いた．Ridge 関数の計算負荷は非常に小さいため，実問題を想定して関数の計算に重みをつけ，1 個体あたりの計算負荷を 0.19 秒と設定した．以下に，用いた Ridge 関数の式を示し，**表 2** に GA のパラメータを示す．下記の式は結果表示の便宜のために，もとの Ridge 関数の式に -0.001 を加えている．

表 2 島モデル GA のパラメータ
Table 2 Parameters of Island model GA used in the experiments.

Number of islands (or workers)	8
Number of indivisuals (per island)	18
Chromosome length	100
Crossover method (Rate)	1 point crossover (1.0)
Mutation method (Rate)	Bit complement (1 / Chromosome length)
Migration topology	Ring generated randomly each time
Migration interval	20 generations
Number of migrants	2

$$f = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2 - 0.001 \quad -5.12 \leq x_i \leq 5.12$$

図 8，**図 9** に実験結果を示す．いずれのグラフも，3 試行の結果，経過時間に対する全探索点の中の最良解を示している．Normal は 5.2 節のモデルを環境の変化なく実行した結果であり，Overload は 8 ワーカで GA を開始し，400 秒経過後に，そのうちの 2 ワーカで他のユーザの計算が始まっても，そのまま計算を続行した結果である．図 3 に示すように．ワーカ側の計算機はデュアルプロセッサであるため，他のユーザのプロセスを 2 つ走らせ，その計算機のロードアベレージが 3.0 となるよう負荷を加えた．Release は，400 秒後に負荷が高くなると島消去機能が働くようにしたモデルの結果である．

図 8 より，400 秒以降，2 つのワーカに負荷が加えられた Overload の探索速度が低下していることが分かる．それに対して図 9 では，島消去機能が働く Release が探索速度を低下させることなく探索を続けて

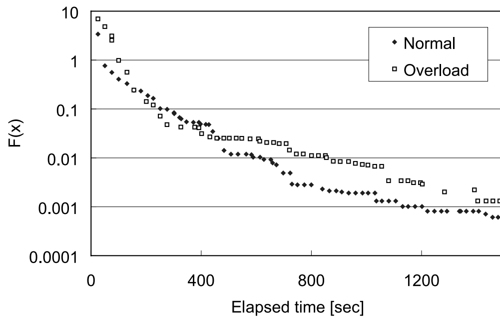

図 8 負荷増加の影響
Fig. 8 Affect by increase of load.

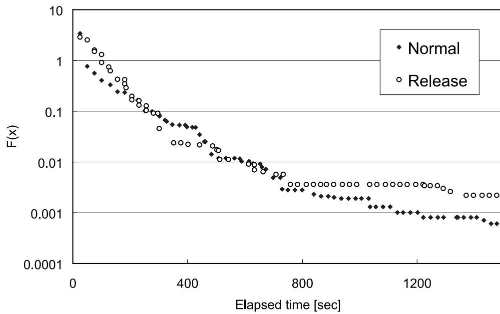

図 9 動的な島消去の影響
Fig. 9 Affect by dynamic reduction of some islands.

いるのが分かる．ただし，Release は探索点数を削減しているため，探索の後半で多様性を保つことができなくなっている．ユーザが計算を継続し，精度の高い解を得ることを望むのであれば，多様性が失われる前に探索点を別のワーカで確保するような操作を適用する必要がある．あるいは，ユーザが 800 秒前後で計算を停止し，その時点での近似解で満足するかもしれない．このように，資源と計算の状況を見ながら，ユーザの目的に従ってアプリケーションのレベルで動的にモデルを変更する検討ができることが GMWS の特徴の 1 つである．

6. 関 連 研 究

Condor MW [8]や AMWAT（The AppLeS Master/Worker Application Template）[13]は，マスターワーカ型のアプリケーションをグリッド上で実行するためのシステムである．テンプレートとなるプログラムが用意されており，マスターとワーカ側の動作および相互に送る通信データを記述することでアプリケーションを作成する．そのため，GOCA で定義されるモデルのうち，独立に実行可能な分散計算プログラムを記述するのに適している．さらに，両システムともにスケジューリングに注力しており，グリッド上の処理能力を最大限に生かせるようワーカ計算を効率良く実行したり，障害時の対応を行ったりする機能を持つ．それに対して，提案する GMWS は，グリッド資源の利用状況や障害などを考えた対応を，アプリケーションレベルで記述できる仕組みを提案している．たとえば，5.4 節で示したように，島モデルで動的に島を消去する操作を組み込むことができ，アプリケーションのとりうるモデルの幅を広げることができる．同様のことを上記のシステムで行うことは容易ではない．

GridRPC に基づくシステムである NetSolve [3]，Ninf [11]，OmniRPC [17]は，遠隔に存在する計算サービスを簡単に呼び出す仕組みとインタフェース，計算サービスを用意する方法を提供する．GridRPC の枠組みでは，1 つのタスクに対して 1 つの RPC 要求がなされ，複数のタスクにまたがる仕事の継続性やタスクどうしの通信についての明確な規定がない．そのため，NetSolve や Ninf では，継続した仕事の実行はオーバヘッドが大きくなる．OmniRPC では，単純な遠隔呼び出しを提供するだけでなく，そうした実行モジュールの再初期化や持続性の機能を持ち，OpenMP のプログラミングをサポートするなど，より上位のインタフェースが考えられている．しかし，先に述べた MW の仕組みと同様に設計方針が GMWS と異なるために，アプリケーション・プログラムが資源情報を取得する仕組みはなく，動的な変化はシステムに吸収され，GOCA を満たすグリッド計算モデルは作りにくいといえる．

MPICH-G2 [9]は，Globus で構築されるグリッド上で動作する MPI の実装である．従来の MPI で記述されたプログラムをそのままグリッド上で動かすことができ，今まで同様，自由度の高い並列プログラミングが可能である．Cluster of Cluster のグリッド資源も利用でき，WAN 接続と LAN 接続を自動的に発見し，利用環境に適したグループ通信を選択するような仕組みも備えている．しかしながら，MPI は，グリッドのような計算環境を考慮したものではなく，その枠組みでグリッド環境に適したプログラムを開発することは労力をともなう．すなわち，プログラマ自身がシステム情報を取得したり，障害時の対応を記したりするための仕組みを作成する必要が生じる．

7. 結 論

本論文では，グリッド環境に適用しやすい計算アプリケーションについて分析し，そのようなアプリケーションを開発・実行するための枠組みの構築を目指して GMWS を提案した．GMWS はマスターワーカの仕

組みで動作し，マスターからワーカを操作できる API，マスターとワーカが通信するための API と，ワーカが動作する資源情報を取得する API を提供する．アプリケーション開発者は，取得した情報を利用して動的に変化するモデルや障害に対して強いモデルを作ることができる．本論文では，このような GMWS の設計目標やシステム・アーキテクチャ，動作，API について述べ，実際の環境での基本的な性能評価を示した．そして，遺伝的アルゴリズムを用いて，GMWS のプログラム作成例として，マスターワーカ方式に沿ったモデルを従来どおり作成できるほか，グリッド環境の特性を生かしたモデルの構築が可能であることを示した．

今後は，提案システムのスケーラビリティの調査と性能向上を図ると同時に，グリッドの特性を生かして，既存の並列アプリケーションの計算モデルをどのように変更していく，あるいはしないのがよいかを個々のアプリケーションごとに検討していく予定である．

参 考 文 献

1) Netperf benchmark. http://www.netperf.org
2) Alba, E. and Troya, J.M.: A Survey of Parallel Distributed Genetic Algorithms, *Complexity*, Vol.4, No.4, pp.10–11 (1999).
3) Casanova, H. and Dongarra, J.: Netsolve: A Network Server for Solving Computational Science Problems, *The International Journal of Supercomputer Applications and High Performance Computing*, Vol.11, No.3, pp.212–223 (1997).
4) Daniel, J.B. and Richard, E.S.: *SSH, The Secure Shell: The Definitive Guide*, O'Reilly (2001).
5) Foster, I. and Kesseleman, C.: *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann (1998).
6) Foster, I. and Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit, *The International Journal of Supercomputer Applications and High Performance Computing*, Vol.11, No.2, pp.115–128 (1997).
7) Goldberg, D.E.: *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley (1989).
8) Goux, J., Kulkarni, S., Linderoth, J. and Yoder, M.: An Enabling Framework for Master-Worker Applications on the Computational Grid, *Proc. HPDC-9*, pp.43–50 (2000).
9) Karonis, N., Toonen, B. and Foster, I.: MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, *Journal of Parallel and Distributed Computing* (2003).
10) Levine, D.: A parallel genetic algorithm for the set partitioning problem, Technical Report ANL-94/23, Algonne National Laboratory, Mathematics and Computer Science Division (1994).
11) Nakada, H., Sato, M. and Sekiguchi, S.: Design and Implementations of Ninf: toward a Global Computing Infrastructure, *Future Generation Computing Systems, Metacomputing Issue*, Vol.15, pp.649–658 (1999).
12) Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C. and Casanova, H.: GridRPC: A Remote Procedure Call API for Grid Computing, Technical Report ICL-UT 02-26, Innovative Computing Laboratory (2002).
13) Shao, G., Berman, F. and Wolski, R.: Master/Slave Computing on the Grid, *Proc. HPDC-9* (2000).
14) Tanese, R.: Parallel Genetic Algorithms for A Hypercube, *Proc. 2nd ICGA*, pp.177–183 (1987).
15) Thain, D., Tannenbaum, T. and Livny, M.: Condor and the Grid, *Grid Computing: Making The Global Infrastructure a Reality, Berman, F., Hey, A.J.G. and Fox, G.* (*Eds.*), John Wiley (2003).
16) 夏目　亘，合田憲人，二方克昌：階層的マスタワーカ方式による bmi 固有値問題の grid 計算，情報処理学会研究報告 HPC-91-13 (2000).
17) 佐藤三久，朴　泰祐，高橋大介：OmniRPC：グリッド環境での並列プログラミングのための Grid RPC システム，情報処理学会論文誌：コンピューティングシステム，Vol.44, No.SIG11 (2003).

(平成 15 年 10 月 10 日受付)
(平成 16 年 2 月 20 日採録)

谷村　勇輔（学生会員）

1976 年生．2001 年同志社大学大学院工学研究科修士課程修了．同年同志社大学大学院工学研究科博士課程入学．クラスタや広域環境における並列・分散計算，進化的最適化手法に興味を持つ．IEEE-CS 学生会員，超並列計算研究会会員．

廣安　知之（正会員）

1997年早稲田大学大学院理工学研究科後期博士課程修了．2003年より同志社大学工学部助教授．進化的計算，最適設計，並列処理，設計工学等の研究に従事．IEEE，電気情報通信学会，計測自動制御学会，日本機械学会，超並列計算研究会，日本計算工学会各会員．

三木　光範（正会員）

1950年生．1978年大阪市立大学大学院工学研究科博士課程修了，工学博士．大阪市立工業研究所研究員，金沢工業大学助教授を経て，1987年大阪府立大学工学部航空宇宙工学科助教授，1994年同志社大学工学部教授．進化的計算手法とその並列化，および知的なシステムの設計に関する研究に従事．著書は『工学問題を解決する適応化・知能化・最適化法』（技法堂出版）等多数．IEEE，米国航空宇宙学会，人工知能学会，システム制御情報学会，日本機械学会，計算工学会，日本航空宇宙学会各会員．超並列計算研究会代表．