# Grid RPCにおける広域データ管理レイヤの利用

相田祥昭[†,☆] 中島佳宏[†,††] 佐藤三久[†]
建部修見[†] 櫻井鉄也[†]

これまで Grid RPC では各ワーカごとに共通に使用されるデータや呼び出しごとに変化するパラメータの転送を同じデータ通信レイヤ上で行ってきた．しかし，RPC モデルでは通信はマスタ/ワーカ間で 1 対 1 で行われるため，ネットワークトポロジを意識した効率的なデータ転送を行うことが難しい．またワーカ間の直接のデータ授受は不可能であるため，マスタを介した RPC 通信が必要である．そこで我々は，データ転送を効率化してスケーラビリティの向上を図るためにワーカ間で共通に使用されるデータと呼び出しごとに異なるパラメータデータの転送を分離するモデルを提案する．そしてこのモデルを用いたデータ管理レイヤ OmniStorage を設計・実装し，性能評価を行った．ネットワークトポロジを考慮しツリー状にデータのブロードキャスト転送を行うシステムと，P2P モデルを用いたファイル転送ソフトウエアである BitTorrent を使うシステム，広域分散ファイルシステムである Gfarm を利用したシステムをそれぞれ実装し，性能評価を行った．その結果，Grid RPC 単体の場合よりも提案システムも併用する方が性能面やスケーラビリティに関して優位性があることが分かった．さらに，Grid RPC アプリケーションでのブロードキャストや one-to-one，all-to-all などの必要な通信パターンを基に，データ転送レイヤを選択することが性能面やスケーラビリティに関して有効なことが分かった．

# Performance Improvement by Distributed Data Management Layer on Grid RPC System

Yoshiaki Aida,[†,☆] Yoshihiro Nakajima,[†,††] Mitsuhisa Sato,[†]
Osamu Tatebe[†] and Tetsuya Sakurai[†]

Grid RPC applications often need large size of shared data to initialize worker programs. Conventional grid RPC model only transfers data by RPC's parameters from a master to a worker for both shared data and RPC's parameters. Since the RPC model supports point-to-point communication between a master and workers, it is difficult to achieve efficient data transfer. Furthermore in the conventional RPC model each worker does not communicate directly with other workers. To address this kind of the issues, we have designed and implemented a data management layer OmniStorage that augments functionality of the grid RPC model. It decouples the data transmission from RPC mechanism aiming to achieve the efficient data transfer fand choose a suitable data transfer method of OmniStorage. We have implemented OmniStorage on three data transfer methods; tree-topology-aware data broadcasting middleware, BitTorrent protocol for P2P file sharing and Gfarm as a distributed parallel file system. We evaluated the basic performance of three implementation using synthetic benchmark programs. The proposed system achieved better performance than the original OmniRPC system in terms of both scalability and efficiency of data transfer. We found that selecting a data transfer method gave an impact on both the performance and scalability of applications with OmniStorage.

## 1. はじめに

近年，広域ネットワークインフラの発達にともなって，広域ネットワーク上での計算機資源の統合やデータの共有を可能にするグリッド技術が注目されるようになっている．この中で，グリッド環境上の複数の計算資源を遠隔手続き呼び出し（Remote Procedure Call）によって利用する Grid RPC が，グリッド環境におけ

† 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
†† 日本学術振興会特別研究員
Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists
☆ 現在，沖電気工業株式会社
Presently with Oki Electric Industry Co., Ltd.

るプログラミングモデルの 1 つとして有望視されている．Grid RPC はパラメータサーチアプリケーションやタスク並列アプリケーションにおいて有効なプログラミングモデルである．我々は，この Grid RPC のためのミドルウェア実装の 1 つとして OmniRPC[1),2)] の研究開発を進めている．

Grid RPC を用いた典型的なプログラムは，遠隔呼び出しを行うマスタと，呼び出される手続を実行する複数のワーカで構成される．このようなアプリケーションでは各ワーカ間で比較的大きなデータを共通に持つ必要がある場合が多い．たとえば，パラメータサーチ型のアプリケーションではそれぞれのワーカが同じデータを持ち，パラメータを変えて並列に同じ遠隔呼び出しを実行する．Persitency を持たない遠隔呼び出しのみの場合は，呼び出しごとに共通のデータを送らなくてはならない．したがって，データのサイズが大きくなった場合はマスタとワーカの間での通信が増えるため，全体の処理に対するオーバヘッドが増大する．

OmniRPC では，このようなプログラムに対してモジュール自動初期化機能を提供している．これは，各ホストにおける初回の遠隔呼び出しの前に一度だけ実行される処理を記述することによって実行ホストの初期化を行う機能である．この機能を用いて，ワーカ起動時に共通データの転送を行えば，パラメータごとの同一ホストに対する遠隔呼び出しでは，データ転送なしでただちにワーカの処理を開始することができる[3)]．

しかし，この初期化機能の処理は通常の遠隔呼び出しの一部として実行されている．すなわち，初期化のためのデータはマスタからそれぞれのワーカに直接転送される．ワーカの数が多い場合，あるいはマスタとワーカの間のバンド幅が低い場合には，初期化処理に長い時間がかかり，大きなオーバヘッドとなる．さらに，RPC の引数として数十 MB に及ぶデータ転送を必要とするアプリケーションは存在しており，このようなアプリケーションでは転送にかかる時間の増大が実行効率の低下を招く場合がある．

たとえば，我々が使用した並列固有値計算プログラム[4)] では，1 個のジョブの実行時間が 1 分程度であるのに対して初期化データのサイズが約 50 MB と大きく，広域ネットワークでのデータの転送時間が性能向上のボトルネックとなっている．

また，アプリケーションにて RPC を用いてパイプライニング処理を行うような場合，データの依存性を持つ 2 つ以上の RPC の処理が必要となり，計算ノード間でのデータ通信にはマスタを介して行わなければならない．ここで，計算ノード間のデータ通信のみで対処可能になれば，不必要なマスタへのデータ通信を省くことができ，効率的な処理が可能になる．

そこで我々は，データの転送を Grid RPC の機構から分離し，データ転送のための別レイヤを用いて，データの転送を効率化するモデルを提案する．Grid RPC アプリケーションのあるプロセスがデータを識別子とともにデータレポジトリに登録し，そのデータを使用するプロセスは，識別子を用いてデータレポジトリよりデータを取得する．遠隔呼び出しの引数として転送する場合には個々のワーカに転送しなければならないのに比べて，データの転送方法を工夫することができ，結果として転送の効率化が可能になる．

我々はこのためのプロトタイプ実装としてデータ管理レイヤ OmniStorage を設計，実装した[5)]．OmniStorage は，Grid RPC ミドルウエアとともに利用し，大規模なデータ転送を RPC から分離し，効率的な転送を行うためのフレームワークである．OmniStorage は，識別子を用いてデータを扱うための API を提供し，ユーザから配送レイヤ内部の動作やデータの所在を隠蔽する．さらに，複数のデータ配送レイヤに対応し，かつ使用されるデータに対してヒント情報を付加させることにより，OmniStorage でそのデータの転送の種類を考慮したデータ転送レイヤを選択可能にする．これによってデータの転送の効率化を目指す．また，複数データ配送レイヤに対応させるべく，ツリートポロジ状にブロードキャストを行うミドルウエア，広域分散ファイルシステムである Gfarm，Peer to Peer を用いたファイル転送ソフトウエアである BitTorrent のそれぞれのミドルウエア上に OmniStorage を実現した．本稿では異なる環境下で，実装した OmniStorage の基本的な性能評価と実アプリケーションでの本システムの有効性を報告する．

次章では OmniRPC の概要，3 章で提案システムの概要と 4 章でその実装についてそれぞれ述べる．5 章でグリッド環境上での性能評価を行い，終章でまとめと今後の課題について述べる．

## 2. 背景：OmniRPC の概要

OmniRPC は，クラスタから広域ネットワークで構成されたグリッドに至る様々な計算機環境において，シームレスなマスタ/ワーカ型の並列プログラミングを可能にする Grid RPC システムである．

OmniRPC で想定しているグリッド環境は，インターネット上で複数の計算機クラスタが接続され，それらのクラスタを相互利用するような環境である．ま

た OmniRPC は，現在のクラスタ環境に多く見られる，クラスタのマスタノードだけがグローバル IP を持ち，スレーブノードはプライベートアドレスを用いた構成も計算機資源として利用可能である．

OmniRPC の API は，基本的に Ninf[6] の API を踏襲しており，さらにワーカプログラム側の計算データの状態を保持する Persistency をサポートしている．この Persistency をサポートした API を利用することにより，効率的なプログラミングが可能となる．また，非同期呼び出しの API を用いることにより並列プログラミングを行うことができる．

また，グリッド環境において典型的な並列アプリケーションであるパラメータサーチなどのアプリケーションを効率的にサポートするために，OmniRPC では，リモート実行モジュールの起動時に実行される初期化手続きを定義することによって，実行モジュールの起動時に自動で初期化する機能を有している．この機能により，リモート実行プログラムの初期化のための大量のデータ転送や計算を，2 回目の RPC 以降省くことができる．

OmniRPC は，エージェントを使用してクライアントプログラムと複数のリモート実行プログラムとの通信を多重化し 1 つのコネクションで行うことができる．この通信の多重化を利用することにより，ユーザは，プライベートアドレスで構築されたクラスタや 1,000 台規模のリモートホストを利用することができる．

## 3. 広域データ管理レイヤ OmniStorage の設計

### 3.1 Grid RPC におけるデータ転送に関する問題点

遠隔手続き呼び出しでは，関数呼び出し側のプロセスであるマスタと，関数を呼び出される側のプロセスであるワーカと，Point-to-Point 通信を必要とする．このため，初期データの転送，RPC のデータ依存性がある場合のワーカ間でのデータ転送，大規模な結果の収集に関して以下のような問題が発生する．

#### 3.1.1 初期データの転送に関わる問題

パラメータサーチのような並列計算において，リモート実行プログラムを実行させるために必要な入力データは，複数のジョブで共通な「初期データ」とジョブごとに異なる「パラメータデータ」に分けることができる．

一般的な Grid RPC システムにおいては，前回使用したデータを利用するようなことはできず，RPC ごとに初期データとパラメータデータを転送しなくてはならない，これに対して，OmniRPC では，ワーカプログラムで共通に使用される初期データを，RPC ごとに送らずにワーカプログラムの初回起動時に転送し，2 回目以降の RPC でそのデータを再利用できるモジュール自動初期化機能を持つ．しかし，ワーカプログラム起動とモジュール初期化処理はシリアライズされており，すべてのワーカの起動が完了するまでにワーカ 1 台の起動時間の台数倍の時間がかかってしまう．さらに，転送データ量の増加に従い，計算ノードにジョブを割り当てられない場合も発生し，その結果計算能力を有効に活用することができず，結果として性能向上が得られない．

#### 3.1.2 RPC のデータに依存性がある場合におけるワーカ間でのデータ転送に関わる問題

ある RPC の計算結果のデータを用いて別な RPC で計算を行うような場合，つまり 2 つ以上の RPC を用いて計算を行うような場合でかつ複数の RPC においてデータの依存性があるような場合を考える．基本的に RPC システムにおいては，直接ワーカ間でのデータ交換を行う方法は提供されていないため，ワーカ間でのデータ交換を行う場合には，マスタを介してデータの転送を行わなくてはならない．特に広域ネットワーク環境を考慮すると，このマスタへのネットワーク通信はスケーラビリティを妨げる原因となる可能性が高い．このような場合に，マスタへのデータ転送をバイパスしワーカ間で直接データ転送が可能になれば，不必要なデータ転送を省くことができ，全体の処理の効率を上げることができる．

#### 3.1.3 計算結果のデータの収集に関わる問題

大規模な計算実行を考慮すると，RPC 呼び出しを用いて計算した結果のデータやファイルのサイズが膨大することが考えられる．これらのデータやファイルをマスタに収集する場合，複数の計算ノードからのデータ転送がマスタへ集中してしまい，ネットワークの輻輳が発生して効率良くデータの収集を行うことができない．転送データの増加にともなってさらにこの状況は悪化することが考えられる．

### 3.2 広域データ管理レイヤ OmniStorage の設計

前節の問題を解決するために，データの転送を Grid RPC の機構から分離し，効率的なデータの転送を可能にする広域データ管理レイヤのプロトタイプとして，我々は OmniStorage を設計・実装した．

OmniStorage は，Grid RPC アプリケーションのための，データ通信を RPC から分離し，効率良くデータ管理を行うためのデータレポジトリとして働くフレームワークである．OmniStorage の概要を**図 1** に示す．

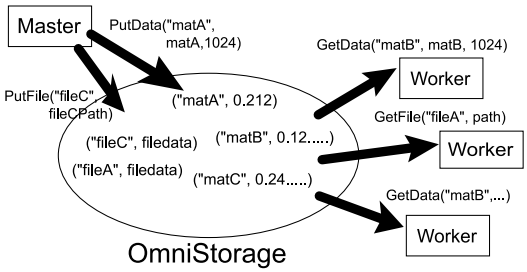


**図 1** OmniStorage の概要
Fig. 1 Overview of OmniStorage system.

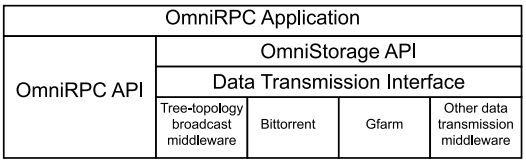


**図 2** OmniStorage を用いた OmniRPC アプリケーションのソフトウエアスタック
Fig. 2 Software stack of OmniRPC applications with OmniStorage system.

OmniStorage のデータレポジトリにおいて，データは “(id, value)” というように，識別子とデータの組で管理される．また，アプリケーション側からは，データのロケーションを意識せずに登録・取得ができるように，ユーザが定義したユニークな識別子を用いてデータを管理する．ここで，OmniStorage で管理されるデータのネームスペースに関しては，OmniStorage ではアプリケーションごとに管理することとする．これは，OmniStorage のプロセスは OmniRPC アプリケーションの開始とともに起動され，またアプリケーションの実行終了とともにプロセスを終了させることを想定しているためである．さらに，識別子のネームスペースにおいて同じ識別子の登録は想定しない．これは実装を簡易にするためである．OmniStorage におけるデータに関しては，作成したアプリケーションのみがアクセス可能で，ほかのアプリケーションからは利用しない．

OmniStorage を用いた OmniRPC アプリケーションのソフトウエアスタックを**図 2** に示す．OmniRPC アプリケーションにおいて，ワーカ間で自由にやりとりを行うようなデータに関しては OmniStorage で管理し，ワーカごとのデータやワーカ側での処理ためのデータ管理については OmniRPC の API を用いて実現する．また複数のデータ転送レイヤを活用可能なように，OmniStorage API の下のレイヤ Data Transfer API では，OmniStorage API と各データ転送レイヤをつなぐためのグルーコードを記述するようにした．

```
/* master program */
int main(){
 double sd[LEN];
 ...
 for(i = 0; i < n; i++)
  req[i] = OmniRpcCallAsync("foo", LEN, sd, i);
 ...
 OmniRpcWaitAll(n, req);

}

/* worker's IDL */
Define foo(IN int size, IN double data[size],
           IN int iter){
 ...
 /* main calculation */
 ...
}
```

**図 3** OmniRPC のみを用いたプログラミング例
Fig. 3 An example code of OmniRPC program.

OmniStorage においてデータを登録したいプロセスは，データを識別子とともに OmniStorage に登録する．そのデータを使用したいプロセスでは，識別子を用いて OmniStorage にデータ取得要求を転送する．OmniStorage は，データ転送に最適な方法を選択し，要求に従いデータを転送する．OmniStorage 側でこのデータ管理レイヤを用いることによって，結果として効率の良いデータの取得が可能となる．

OmniRPC を用いたパラメータサーチ向けアプリケーションの基本的なプログラム例を**図 3** に，そのプログラム例を OmniStorage を用いたものに変更したコードを**図 4** にそれぞれ示す．マスタ側プログラムでは OmniRPC の非同期呼び出しの前に `OmstPutData()` を呼び出し，ワーカ側プログラムでは RPC の冒頭で `OmstGetData()` を呼び出している．データを識別するために，マスタ側とワーカ側の両方で OmniStorage における識別子 “`MyData`” をもとにして，配列 *sd* のデータ登録・取得を行う．データサイズの指定も行うが，これは両者で同じ値を指定する．また，`OmstPutData()` で登録するデータについての転送のパターンや転送されたときのデータの扱われ方を指定する．このヒント情報により，OmniStorage 内部で転送のパターンに適したデータ転送レイヤを選択する．このプログラム例では，各ワーカに同じファイルを転送することを想定しており，ブロードキャストのためのヒントを設定している．

OmniStorage ではクライアントからクラスタのマスタノードへの通信とマスタノードからクラスタ内部の各計算ノードへの通信を分離しデータ転送の最適化を行う．クラスタのマスタノードと各計算ノードに

```
/* master program */
int main(){
 double sd[LEN];
 ...
 req = OmstPutData("MyData", sd, OMST_DOUBLE * LEN,
                   OMST_BROADCAST);
 OmstWait(req);
 for(i = 0; i < n; i++)
  req[i] = OmniRpcCallAsync("foo", i);

 OmniRpcWaitAll(n, req);
 ...
}

/* worker's IDL */
Define foo(int IN iter){
 double sd[LEN];
 ...
 req = OmstGetData("MyData", sd, OMST_DOUBLE * LEN);
 OmstWait(req);
 ...
}
```

**図 4** OmniRPC と OmniStorage を用いたプログラミング例
Fig. 4 An example code of OmniRPC program with OmniStorage system.

キャッシュを持たせて通信の局所化を行わせることによりこの機能を実現している．データのキャッシングを行うことによって，データサイズが大きくなったときにボトルネックとなるクライアントとクラスタ間の通信を 1 回で済ませることができる．一方でプログラムでは API を用いるだけで OmniStorage の機能を利用できるため，最適化を意識することなくプログラミングが可能になる．

### 3.3 OmniStorage API

OmniStorage が管理しているデータ空間には，OmniStorage の API を通してアクセスする．OmniStorage の API は，データを登録する関数，データを取得する関数と，データの取得を待つ関数から構成されている．以下で各 API の説明を行う．

- `OmstPutData(id, data, size, hint)`
  OmniStorage のシステムに対して，ポインタで示された開始アドレスおよびサイズを持つデータを登録する．さらにデータを一意に識別する名前を指定する．また，登録するデータの転送パターンやデータの扱われ方をヒント情報として指定する．
- `OmstPutFile(id, path, hint)`
  ファイルパスで示されたファイルを登録する以外は前項と同様である．
- `OmstGetData(id, data, size)`
  指定された名前のデータを取得し，ポインタで示されたメモリアドレスに格納するためのリクエストを作成して返す．この関数が終了した時点ではまだデータはメモリに格納されていない．
- `OmstGetFile(id, path)`
  指定された名前のデータを取得し，ファイルパスで示されたファイルに格納するためのリクエストを作成して返す．
- `OmstWait(req)`
- `OmstWaitAll(reqs, noreqs)`
  `OmstGetData()`/`OmstGetFile()` で作成したリクエスト (`req`,`reqs`) を受け取り，データの取得を待つ．すべてのデータの取得が完了すると関数から戻る．

OmniStorage に対してデータを登録する際には，そのデータがどのように，またはどの転送パターンが使用されるのかについてのフラグ，もしくは使用するデータ通信レイヤのフラグを論理和で，`OmstPutData()`，`OmstPutFile()` のヒント情報に指定する．以下でそのフラグの説明を行う．

- データ転送パターンの違いによる属性の指定
  - `OMST_BROADCAST`
    複数のプロセスにこのデータがブロードキャストされる．
  - `OMST_GATHER`
    複数のワーカから，マスタにデータを収集させる．
  - `OMST_POINT2POINT`
    ワーカ間でのデータ通信を行う．
  - `OMST_VOLATILE`
    取得されたデータはリモートで保存されない．
- データ配送レイヤを直接指定する
  - `OMST_BT`
    データ配送レイヤに Bittorrent を使用する．
  - `OMST_TREE`
    データ配送レイヤにツリートポロジ状にデータ配送を行うミドルウエアを使用する．
  - `OMST_GF`
    データ配送レイヤに Gfarm を使用する．

これらの情報を活用することにより，OmniStorage がデータ通信のパターンに適したデータ配信レイヤを選択し，さらに効率的な通信が可能になることが考えられる．

## 4. OmniStorage の実装

データ管理レイヤの OmniStorage の実装手法について述べる．また，あわせて OmniStorage のプロトタイプ実装についても述べる．OmniStorage では，データ転送レイヤとして以下の 3 つのシステムを利用する．

- 我々が開発したツリー構造のネットワークトポロジに基づきデータを配送するブロードキャストのみに適したレイヤ.
- アップロード帯域を有効に利用して大容量ファイルの配布効率を高める目的で作成した高速ファイル交換ネットワークプロトコルである BitTorrent を用いたレイヤ.
- グリッド環境での広域分散ファイルシステムを提供する Gfarm を用いたレイヤ.

以降，ツリー構造のネットワークトポロジに基づくデータ配送ミドルウエア，BitTorrent，Gfarm を用いた OmniStorage の実装をそれぞれ，Omst/Tree，Omst/BT，Omst/GF と呼ぶ.

Omst/Tree，Omst/BT，Omst/GF のローカルキャッシュの実体はファイルである．つまり，OmniStorage において，利用するデータ登録先のプロセスは配列データをファイルに書き込む処理を行う．そして，ホスト間のキャッシュファイルの転送に，Tree トポロジ状にブロードキャストを行うミドルウエアや BitTorrent を使用することとした．Gfarm に関しては，キャッシュファイルへのアクセスをするために，リモート読み込み・書き出しを行ったりファイルのレプリケーションを行ったりするようにしている．このときに転送したキャッシュファイルはホストに保存されることもあり，以後キャッシュとして扱われる．さらに，データ取得先のプロセスではローカルにあるキャッシュファイルから配列データを読み込む処理を行う．このキャッシュサイズはユーザ定義であり，登録するデータ・ファイルサイズによって決まる．使用できるキャッシュのサイズは，各ホストのストレージ容量に依存する．また，現在の実装では，各ホストは使用する全キャッシュを保持する以上のストレージ容量を有していると仮定しており，キャッシュの削除については行っていない.

### 4.1 Omst/Tree

Omst/Tree はツリー構造のネットワークトポロジに基づいて効率的なデータ転送を行うレイヤ上に OmniStorage を実装したものである．ただし，Omst/Tree はデータのブロードキャストの通信のみに特化しており，One-to-One のなどの他の通信へのデータ転送レイヤとして利用はできない.

Omst/Tree は，各ホストに配置されてデータの中継を行うサーバと，ユーザプログラムから呼ばれる API で構成されている．以降，それぞれ Omst-server，Omst-api と呼ぶ．Omst-server はユーザプログラムとは別プロセスで動作する．Omst-api は共有ライブラリとして設計されている．また，各ホストはキャッシュを持ち，Omst-server と Omst-api の両方からアクセスすることができる.

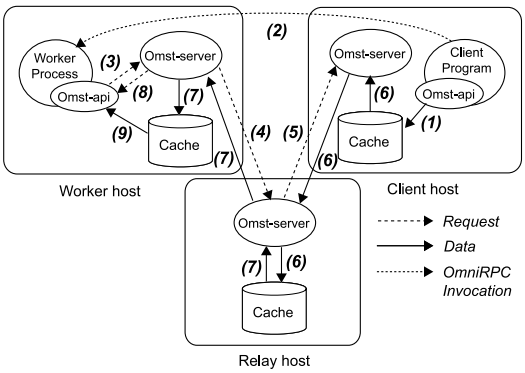


**図 5** Omst/Tree の実装
Fig. 5 Omst/Tree architecture.

**図 5** に Omst/Tree の動作の概要を示す．以下，括弧で囲んだ数字は図中の番号に対応する．図中の破線は制御の流れ，実線はデータの流れを表す.

(1) クライアントプログラムが API を通してクライアントのキャッシュに送信するデータを登録する（`OmstPutData()`）.
(2) OmniRPC でリモート実行プログラムを起動する（`OmniRpcModuleInit()`，`OmniRpcCallAsync()`）.
(3) 起動されたリモート実行プログラムがローカルのキャッシュファイルを調べ，必要なキャッシュファイルがなければワーカホストの Omst-server にキャッシュファイルの要求をする（`OmstGetData()`）．そして `OmstWait()` でキャッシュデータの取得を待つ.
(4) 要求を受けたワーカホストの Omst-server は上流のリレーホストにデータを要求する.
(5) 要求を受けたリレーホストの Omst-server はローカルのキャッシュファイルを調べ，要求されたデータがなければさらに上流のクライアントホストにキャッシュファイルを要求する.
(6) 要求を受けたクライアントホストの Omst-server はローカルのキャッシュファイルから要求されたデータを返す．データはデータ要求を出したリレーホストのキャッシュファイルに格納される.
(7) リレーホストの Omst-server がデータ要求を出したワーカホストに対してローカルのキャッシュファイルからデータを返す．データはワーカホストのキャッシュファイルに格納される.

(8) API にレスポンスを返す.
(9) API がキャッシュファイルからデータを読み込み，メモリに格納した後，`OmstWait()` が返る.

(1) から (9) までの一連の動作がすべて実行されるのはワーカホストとリレーホストのいずれにもデータがなかった場合である．ワーカホストのキャッシュファイルにデータがあった場合は (3) から (9) に飛ぶ．同様にワーカホストのキャッシュファイルになくリレーホストのキャッシュファイルにデータがあった場合は (5) から (7) に飛ぶ．すなわち，あるアプリケーションにおいて一番最初に起動されたジョブは (1) から (9) までのすべてのステップをたどるが，2 番目以降のジョブでは途中にあるキャッシュファイルが利用されるため最上位のホストまでたどることはない．特に同一ワーカ上で 2 回目のジョブが実行されるときは API が直接ローカルのキャッシュファイルを読みにいくため，Omst/Tree システムへのアクセスは発生しない．

一方，2 つの Omst-server 間のコネクションは通常ワーカホスト側から行うが，Omst-Server の設定により，逆向きからコネクションを行うように切り替えることが可能である．この機能により，クライアントホストがプライベート IP アドレスしか持たない場合でも，グローバル IP アドレスを持つホストを中継することで問題なく動作することができる．

### 4.2 Omst/BT

Omst/BT は BitTorrent をキャッシュファイルのデータ転送レイヤとして OmniStorage を実現したシステムである．

BitTorrent [7] は，Bram Cohen が開発した P2P ファイル共有プロトコルである．BitTorrent は巨大なデータを多数のピアに対して効率的に配布することに特化して設計されている．BitTorrent の各ピアは，ファイルのダウンロードが終わり次第データのアップロードを開始し，サーバとして機能することができる．したがってデータの配布の進捗に従って同じデータを持っているピア数が増加し，ピア 1 個あたりの負荷が低下する．さらにピア間の通信は全対全で行われるため，局所的なネットワーク負荷の増大が起こりにくい．BitTorrent では，転送するデータを「ピース」と呼ばれる一定の長さを持った小さな単位に分割する．データの転送はこのピース単位で行う．

Omst/BT の構成を**図 6** に示す．Omst/BT でのキャッシュファイルのノード間の転送に，BitTorrent を用いて行っている．そのため，OmniStorage に登録したデータを使用するホストすべてに，キャッシュファイルが作成される．ただし，通常の BitTorrent を使用する際には，メタ情報ファイル（`torrent` ファイル）の作成，公開，ダウンロードなどの手続きはユーザが手作業で行う．そこで，BitTorrent をデータ転送レイヤとして利用するために，Omst/BT ではこれらの処理をすべて自動化した．また，BitTorrent のクライアントは libtorrent [8] を用いて作成した．libtorrent は BitTorrent クライアントの実装の 1 つで，C++で記述された複数の API によって BitTorrent のクライアントの動作をユーザプログラムから制御することが可能である．以下に Omst/BT の動作の概要を示す．以下，括弧で囲んだ数字は図 6 中の番号に対応する．

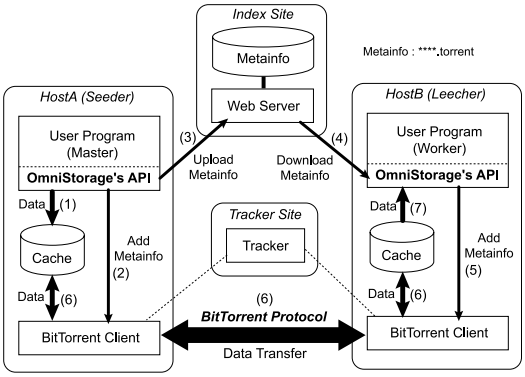


**図 6** Omst/BT の実装
Fig. 6 Omst/BT architecture.

(1) マスタ側プログラムが Omst/BT のデータ登録 API（`OmstPutData()`）を呼び出す．API はこのデータをキャッシュファイルに書き込み，さらにデータを解析して torrent ファイルを作成.
(2) torrent ファイルをインデックスサイトにアップロードする.
(3) torrent ファイルをローカルの BitTorrent クライアントに登録する．以上でデータ配布の準備はすべて完了する.
(4) ワーカ側プログラムが Omst/BT のデータ取得 API（`OmstGetData()`）を呼び出す．API はローカルのキャッシュファイルを調べ，指定されたデータがあればそのデータを返すが，ない場合はデータ取得のために torrent ファイルをインデックスサイトからダウンロードする.
(5) torrent ファイルをローカルの BitTorrent クライアントに登録する.
(6) BitTorrent のピース交換が開始される．ワーカ側プログラムは `OmstWait` でデータの取得完了を待つ.
(7) データの取得が終わると API がキャッシュファ

イルからデータを読み込み，`OmstWait` が返る．

### 4.3 Omst/GF

Omst/GF は分散ファイルファイルシステムのミドルウエア Gfarm 上に OmniStorage を実現したシステムである．

Gfarm システムは複数組織で高信頼なデータ共有，高速なデータアクセスを実現し，大規模データ処理サービス，データインテンシブコンピューティングをグリッド上で可能とするためのシステムソフトウェアである[9]．データ参照局所性を利用し，データのネットワーク移動を抑えることにより，高速入出力，高速並列処理を実現する．Gfarm システムでは，耐故障性を備えた広域仮想ファイルシステムの Gfarm ファイルシステムと，そのファイルシステムと計算グリッドを連携させた分散並列処理環境を提供する．

Gfarm システムは，以下の 3 つの種類のホストから構成される．

- クライアント
  ユーザが Gfarm を利用するホスト．
- ファイルシステムノード
  CPU と，Gfarm ファイルシステムのディスクを提供するホスト群．
- メタデータサーバ
  Gfarm ファイルシステムのメタデータと，ユーザがリクエストした並列プロセスを管理するホスト．

Gfarm ファイルシステムでは，ファイルは任意のファイルシステムノードに配置され，またファイルは複製され，異なるノードに配置されることもある．さらに，ファイルはファイルの集合として構成することも可能であり，個々の構成ファイルは特にファイル断片と呼ばれ，任意のファイルシステムノードに配置される．

Omst/GF ではキャッシュファイルを Gfarm で管理し，キャッシュファイルへのアクセスに関しては，Gfarm で管理されたキャッシュファイルへのリモート読み出し・書き込みを行うことで実現している．つまり，Omst/GF では，キャッシュファイルの作成やデータのキャッシュファイルへの書き込みや読み込みを，Gfarm のファイル I/O を行う関数を使うことによって実現している．ファイルを作成する場合やファイルを読み込む・書き込む場合のホストの選択については，Gfarm のスケジューラの機能を用いることとした．ただし，データの転送パターンによってキャッシュファイルを作成する際に，リモートホストにキャッシュファイルを複製することにより，効率の良いデータ転送を実現する可能性がある場合がある．このような場合には，Omst/GF でヒント情報を解釈し，リモートノードにキャッシュファイルを複製するようにした．今回の実装では，データ転送のパターンがブロードキャストの場合に，2 つ以上の複製を作成することとした．

## 5. 性能評価

提案システムであるデータ管理レイヤ OmniStorage を用いた際の，Grid RPC アプリケーションに対するデータ転送の効率改善について性能評価を行う．RPC アプリケーションをモデル化したプログラムと実アプリケーションを用いて性能評価を行う．

ここで RPC アプリケーションをモデル化したプログラムでは，RPC アプリケーションで必要とされるデータ転送のパターンに基づいて，OmniStorage の基本的な性能を使用するデータ転送レイヤを変えて評価した．また，OmniStorage を使用した例として実 Grid RPC アプリケーションである並列固有値計算プログラムを用いた性能評価を行う．ただし，OmniStorage 内部で使用しているデータ転送レイヤの BitTorrent や Gfarm システム自身の性能はそれぞれ報告があるためそちらを参照されたい[10)~12)]．

### 5.1 実験環境

実験に際して，異なるネットワークで接続された 3 クラスタを使用した．またマスタプログラムは，筑波大学にある計算機上で実行した．計算機環境を，**表 1** に示す．また，**図 7** と**表 2** に各ホスト間のネットワークバンド幅の実測値を示す．

使用したソフトウエアは，OmniRPC version 1.2, libtorrent version 0.9.1, Azureus version 2.5.0.2, Gfarm version 1.4, Boost C++ Library version 1.33.1, Java2 Runtime Environment, Standard Edition version 1.4.2 である．

以降の各計測結果は，5 回測定した結果からの平均値を用いている．

### 5.2 RPC アプリケーションをモデル化したベンチマークによる OmniStorage の基本的な性能評価

OmniStorage の基本性能の評価を，RPC アプリケーションをモデル化し，3 つのデータ通信パターンに特化したベンチマークプログラムを用いて行う．

- 1 ワーカから 1 ワーカにファイル転送．
- 初期化ファイルをマスタから全ワーカに転送．
- 全ワーカにおいて各ワーカの保持するデータを自分以外の全ワーカにファイル転送．

OmniStorage の基本的な性能評価として，2 つのクラスタを結ぶネットワークの構成が異なる 2 つの環境

**表 1** グリッドテストベットの計算機環境
Table 1 Machine configuration on grid testbed.

| Site | Cluster Name | Machine | Memory | Network | Storage | Node 数 |
|---|---|---|---|---|---|---|
| omni.hpcc.jp | ClusterA | Dual Xeon 2.4 GHz | 1 GB | 1 Gb Ethernet | 80 GB 7200 rpm IDE HDD | 16 |
| AIST | ClusterB | Dual Xeon 3.2 GHz | 1 GB | 1 Gb Ethernet | 3ware RAID controller 750 GB | 8 |
| 筑波大学 | ClusterC | Dual Xeon 3.2 GHz | 2 GB | 1 Gb Ethernet | 147 GB 10025 rpm SCSI HDD | 64 |
| | cTsukua | Dual Xeon 2.4 GHz | 1 GB | 1 Gb Ethernet | 40 GB 7200 rpm IDE HDD | 1 |

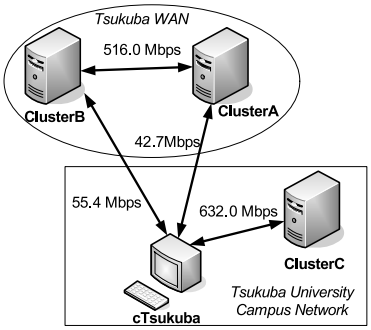


**図 7** 使用したクラスタのネットワーク
Fig. 7 Overview of grid testbed network.

**表 2** ホスト間のネットワークバンド幅
Table 2 Network bandwidth between two nodes.

| ノード間 | | | Network Bandwidth (Mbps) |
|---|---|---|---|
| cTsukuba | – | ClusterA | 42.7 |
| cTsukuba | – | ClusterB | 55.4 |
| ClusterA | – | ClusterA | 898 |
| ClusterA | – | ClusterB | 516 |
| ClusterB | – | ClusterB | 752 |
| cTsukuba | – | ClusterC | 632 |
| ClusterC | – | ClusterC | 892 |

を想定し，以下の 2 つの計算機環境を設定した．

(1) 2 つのクラスタが高速なネットワークによって接続されている場合
ClusterA クラスタを 8 ノードずつ 2 つに分け，仮想的に 2 つのクラスタとして使用する．

(2) 異なるネットワークにある 2 つのクラスタを使用する場合
ClusterA クラスタの 8 ノードと ClusterB クラスタの 8 ノードを使用する．

ただし，マスタプログラムは cTsukuba にあるノード上で実行することとした．また各ノードで動作させるワーカは 1 とし，ノードごとに別のワーカを動作させることにする．

#### 5.2.1 1 ワーカから 1 ワーカへのファイルの転送

ある RPC の計算結果のデータファイルを用いて別な RPC で計算を行うようなアプリケーションを想定し，OmniStorage を介してワーカとワーカ間でファイルを授受するベンチマークを作成した．このベンチマークでは，あるノードで動作するワーカがデータファイルを OmniStorage に登録し，別のノードで動作するワーカがそのデータファイルを取得する際のデータファイル取得にかかる時間をデータのサイズを変化させて測定した．

```
/* master program */
int main(){
 ...
 t_start = getTime();
 OmniRpcCallByHandle(nodeA, "file_get", path);
 OmniRpcCallByHandle(nodeB, "file_put", path);
 t_end = getTime(); calcExecTime(t_start,t_end);
 ...
}

/* worker's IDL */
Define file_get(OUT filename fp){
 fp = "bdata.dat";
}
Define file_put(IN filename fp){
  /* nothing to do */
}
```

**図 8** OmniRPC のみを用いた 2 ワーカ間のファイル転送プログラム
Fig. 8 OmniRPC program code that transfers a file from a worker to another worker.

OmniRPC のみを用いたときと OmniStorage を用いたときのベンチマークのコードの断片をそれぞれ，**図 8**，**図 9** に示す．OmniRPC のみの場合には，あるワーカからファイルを同期 RPC（`file_get`）の引数としてマスタに転送し，その後同期 RPC（`file_put`）の引数としてファイルをワーカに送る．OmniStorage を利用する際には，マスタが同期 RPC（`p2pt_wput`）を用いてあるワーカ上で転送するファイルを識別子（`omst_id`）で OmniStorage に登録し，その識別子をワーカに RPC の引数として渡す．その後，同期 RPC（`p2pt_wget`）でファイルの識別子を別なワーカに渡し，そのワーカが OmniStorage の API を用いてファイルを取得する．このベンチマークにおける実行時間は，マスタ側で 2 つの同期 RPC にかかる時間（t_start, t_end の差分）を測定している．

まず，ベンチマークの実行の前に，Omst/BT の各データサイズにおけるメタデータの登録のために必要

```
/* master program */
int main(){
 ...
 t_start = getTime();
 OmniRpcCallByHandle(nodeA, "p2pt_wput", omst_id);
 OmniRpcCallByHandle(nodeB, "p2pt_wget", omst_id);
 t_end = getTime(); calcExecTime(t_start,t_end);
 ...
}

/* worker's IDL */
Define p2pt_wput(OUT string omst_id){
 asprintf(&omst_id, "bcastfile");
 r = OmstPutFile(omst_id, "bdata.dat", OMST_POINT2POINT);
 OmstWait(r);
}
Define p2pt_wget(IN string omst_id){
 r = OmstGetFile(omst_id, path);
 OmstWait(r);
}
```

**図 9** OmniStorage を用いた 2 ワーカ間のファイル転送プログラム

Fig. 9 OmniRPC program code using OmniStorage system that transfer a file from a worker to another worker.

なオーバヘッドの詳細を測定した．ただし，Omst/Tree と Omst/GF に関してはメタデータの登録はほとんど無視できる時間であるためオーバヘッドの測定については行わない．Omst/BT において，データファイルを OmniStorage に登録する際に以下の 3 つの処理が必要である．

(1) torrent ファイルの作成
(2) インデックスサイトに torrent ファイルを登録する処理
(3) ピアにデータファイルを転送を開始するための処理

Omst/BT のメタデータの登録に必要なオーバヘッドの詳細を**図 10** に示す．torrent ファイルをインデックスサイトに転送する際の時間はほぼ無視できるほどであるが，Bittorrent データサイズの増大に従い，torrent ファイルを作成するための時間が増加していることが分かる．これはファイルをあるサイズのピースに分割し各ピースでハッシュ値を計算してるためである．さらにピアにデータファイル転送を開始するために必要な処理時間は，データファイルのサイズが 64 MB 以下の場合はほぼ一定であり．256 MB 以上の場合で緩やかに増加する．

高速なネットワークで接続された 2 ワーカ間（ClusterA-node0 と ClusterA-node8）のデータファイル転送にかかる実行時間と，異なるネットワークにあるクラスタの 2 ワーカ間（ClusterA-node0 と ClusterB-node0）のデータファイル転送にかかる実行時間をそれぞれ，**図 11** と**図 12** に示す．性能向上の比較として，OmniRPC のみを用いてワーカ間のデータファイルを交換したときの実行時間を加えた．

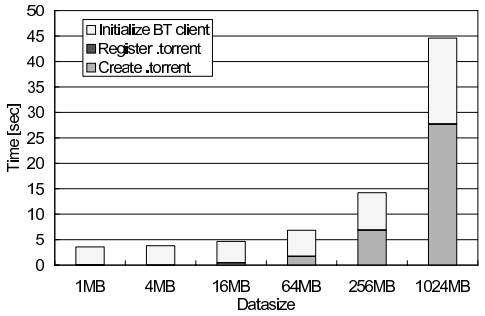


**図 10** Omst/BT の各データファイルサイズにおける前処理にかかる実行時間

Fig. 10 Execution details of pre-processing of Omst/BT according to the data size.

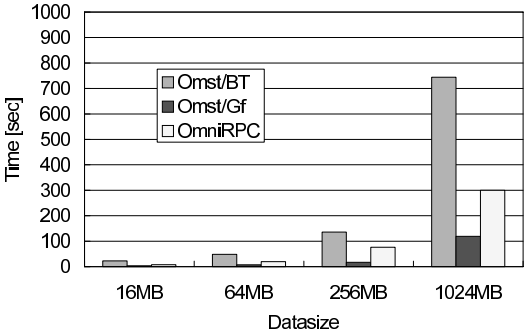


**図 11** 高速なネットワークで接続された 2 ワーカ間（ClusterA-node0 と ClusterA-node8）のデータファイル転送にかかる実行時間

Fig. 11 Elapsed time of one file transfer between two workers by higher performance network.

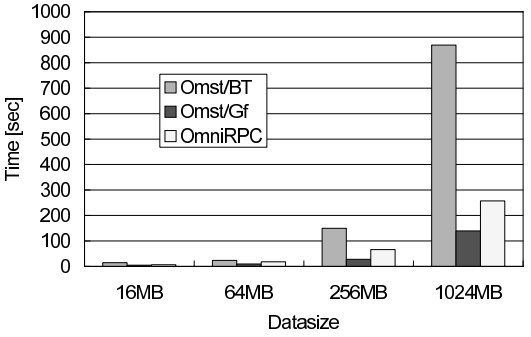


**図 12** 異なるネットワークにあるクラスタにある 2 ワーカ間（ClusterA-node0 と ClusterB-node0）のデータファイル転送にかかる実行時間

Fig. 12 Elapsed time of one file transfer between two workers in connected by lower performance network.

2 つの計算機構成にかかわらず，Omst/GF が効率良くワーカ間でのデータファイル通信が行えている．Omst/GF は，OmniRPC 単体でワーカ間のデータファイル通信を実現する場合よりも約 2 倍効率良く転送ができている．Omst/GF や Omst/BT ではデー

**表 3** ClusterA-node0 と ClusterA-node8 間のデータファイル転送にかかる実行時間（sec）の詳細
Table 3 Details of elapsed time of one file transfer between ClusterA-node0 and ClusterA-node8.

| 使用したノード | データサイズ | 16 MB | | 64 MB | | 256 MB | | 1024 MB | |
|---|---|---|---|---|---|---|---|---|---|
| | | OmstPutData() | OmstGetData() | OmstPutData() | OmstGetData() | OmstPutData() | OmstGetData() | OmstPutData() | OmstGetData() |
| ClusterA – ClusterA | Omst/BT | 6.95 | 15.8 | 9.88 | 38.64 | 22.79 | 112.82 | 130.16 | 613.90 |
| | Omst/GF | 1.16 | 2.33 | 2.37 | 4.71 | 4.22 | 12.87 | 58.68 | 60.37 |
| | OmniRPC のみ | 3.32 | 4.25 | 6.57 | 13.24 | 24.87 | 51.57 | 97.67 | 202.33 |
| ClusterA – ClusterB | Omst/BT | 5.71 | 8.89 | 10.75 | 13.01 | 24.73 | 124.86 | 121.49 | 747.93 |
| | Omst/GF | 1.55 | 3.24 | 2.40 | 7.21 | 6.61 | 21.29 | 57.60 | 81.83 |
| | OmniRPC のみ | 2.69 | 3.77 | 7.00 | 10.83 | 25.18 | 40.62 | 97.66 | 159.32 |

タファイル転送を必要とするワーカどうしの通信のみでデータファイル通信を実現が可能であるため，効率的に転送ができている．しかし，OmniRPC のみの利用によるワーカ間のデータファイル通信は，あるワーカからマスタプログラムにデータファイルを転送し，その後別のワーカにデータファイルを転送することにより実現している．この部分が性能ボトルネックとなる．本実験では 2 ワーカのみで実験を行ったが，複数のワーカ間でのデータファイル通信が発生するような場合にはマスタへの通信が集中し，ネットワークの輻輳が発生するため，さらにデータファイル転送の効率が悪くなると考えられる．このような通信パターンには Omst/GF の方が効率が良くデータ転送ができる．

ただし，Omst/BT は OmniRPC のみでワーカ間のデータファイル通信を実現するよりも効率が悪い．これは，Bittorrent は多数のピアが集合した際には性能良く転送ができるが，プロトコル性質上少数のピア間でデータファイル通信を行う際には効率が良い転送を実現することができないためである．また，BitTorrent は全体のピアにピースが平均的に行きわたった状態になるまではピースの交換が活発に行われず，ネットワーク全体のバンド幅が生かしきれない．さらに，BitTorrent は接続先のピアを発見するまでにある程度タイムラグがあるため，データサイズが小さい場合他のデータ管理レイヤよりも不利となる．

それぞれ使用するデータ転送レイヤの違いによって，2 ワーカ間のデータファイル転送に関わる `OmstPutData()` と `OmstGetData()` にかかった実行時間の詳細を**表 3** に示す．Omst/GF は，`OmstPutData()` と `OmstGetData()` にかかる時間はほぼ同じであり，ほかの方式よりもそれぞれ短時間で実行できていることが分かる．Omst/BT では `OmstPutData()` にかかる時間がほかのデータ転送レイヤに比べて長い，これは Bittorrent の torrent ファイルの作成時間があるためである．また，`OmstGetData()` が長時間かかっていることから少数のピアどうしでは効率良くデータ転送ができていないことが分かる．

### 5.2.2 マスタから全ワーカへのファイルのブロードキャスト

Grid RPC アプリケーションによく見られる，全ワーカが同じ初期化用のファイルを必要とし，マスタから全ワーカへのデータファイル転送を行うプログラムをモデル化したベンチマークを作成して性能評価に用いた．使用したベンチマークプログラムは，マスタ側で保持する 1 つのファイルを 2 つのクラスタにある全 16 ワーカに転送させるプログラムである．転送するファイルのサイズを変化させ，マスタがデータファイルの OmniStorage への登録を始め全 16 ノードへのデータファイル転送が終了するまでの実行時間を測定した．

```
/* master program */
int main(){
 ...
 t_start = getTime();
 for(i = 0; i < noNodes; i++)
  r[i] = OmniRpcCallAsyncByHandle(h[i], "bcast_omrpc",
                                                  "bdata.dat");
 OmniRpcWaitAll(noNodes, r);
 t_end = getTime(); calcExecTime(t_start,t_end);
 ...
}

/* worker's IDL */
Define bcast_omrpc(IN filename name){
  /* nothing to do */
}
```

**図 13** OmniRPC のみを用いたファイルのブロードキャストプログラム
Fig. 13 OmniRPC program code that broadcasts a file to all workers.

使用したベンチマークを，OmniRPC のみを用いた場合と，OmniStorage を用いた場合のコードの断片をそれぞれ，**図 13**，**図 14** に示す．OmniRPC のみを用いる場合には，複数の非同期 RPC（`bcast_omrpc`）の引数としてファイルをマスタからワーカに転送している．非同期 RPC をマスタで待ち合わせることによって，ファイルのブロードキャストの終了を検知する．OmniStorage を用いる際には，マスタ側で OmniStorage にファイルを登録し，非同期 RPC（`bcast_omst`）を用いて全ワーカにそのファイルに対応する OmniStorage の識別子（"bcast_file"）; を渡し，ワーカ側で OmniStorage API を用いてファイルの取得を行う．マスタ側でその RPC の終了を待ち合わせすることによって，全ワーカへのファイルのブロード

```
/* master program */
int main(){
 ...
 t_start = getTime();
 r = OmstPutFile("bcast_file", "bdata.dat",
                    OMST_BROADCAST);
 OmstWait(r);
 for(i = 0; i < noNodes; i++)
  r[i] = OmniRpcCallAsyncByHandle(h[i], "bcast_omst",
                                      "bcast_file");
 OmniRpcWaitAll(noNodes, r);
 t_end = getTime(); calcExecTime(t_start,t_end);
 ...
}

/* worker's IDL */
Define bcast_omst(IN string omst_id){
 r = OmstGetFile(omst_id, path);
 OmstWait(r);
}
```

**図 14** OmniStorage を用いたファイルのブロードキャストプログラム
Fig. 14 OmniRPC program code with OmniStorage system that broadcasts a file to all workers.

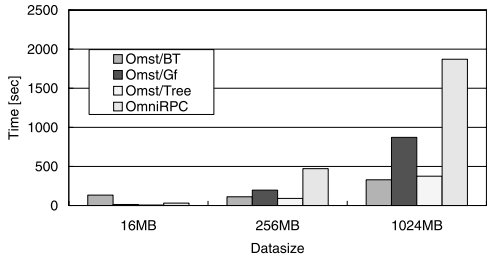


**図 15** マスタから ClusterA で動作する 16 ワーカへのデータファイルのブロードキャストする実行時間
Fig. 15 Elapsed time of one file broadcast from a master to 16 workers in ClusterA.

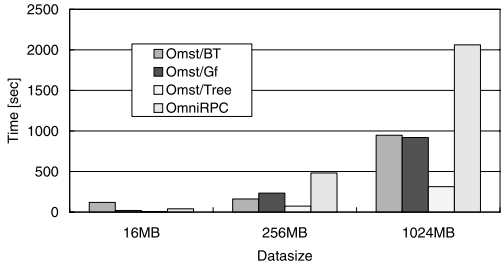


**図 16** マスタから ClusterA と ClusterB で動作する 16 ワーカへのデータファイルのブロードキャストする実行時間
Fig. 16 Elapsed time of one file broadcast from a master to both 8 workers in ClusterA and 8 workers ClusterB.

キャストを完了させる．このベンチマークの実行時間は，マスタ側で非同期 RPC を始めてから全非同期 RPC の終了までの時間（t_start，t_end の差分）を測定している．

ClusterA で動作する全 16 ワーカ（node0 - node16）へのデータファイルのブロードキャストにかかる時間と，ClusterA と ClusterB でそれぞれ動作する全 16 ワーカ（ClusterA-node0 から Cluster-node7 と ClusterB-node0 から ClusterB-node7）へのデータファイルのブロードキャストにかかる時間を，それぞれ**図 15** と**図 16** に示す．性能評価の比較としてワーカから全ノードに OmniRPC のみを用いてデータファイル転送した際の実行時間もあわせて示す．

実験の結果より，ClusterA で動作する 16 ワーカへのデータファイルのブロードキャストの場合，OmniRPC のみを用いて行う場合よりも，Omst/Tree は最大 5.2 倍，Omst/GF は最大 2.4 倍，Omst/BT は最大倍 5.7 高速に実行できている．また，ClusterA と ClusterB で動作する 16 ワーカへのデータファイルのブロードキャストの場合，OmniRPC のみを用いて行う場合よりも，Omst/Tree は最大 6.6 倍，Omst/GF は最大 2.2 倍，Omst/BT は最大 3.0 倍高速に実行できている．

データサイズの大きさによらず，ほぼ Omst/Tree の方が高速にデータファイルをワーカにブロードキャストできていることが分かる．ただし，性能の高いネットワークで 2 つのクラスタが接続されている場合には，Omst/BT はほぼ Omst/Tree と同じような性能でデータファイルのブロードキャストが行えている．特にデータサイズが 1024 MB の場合，Omst/BT は Omst/Tree よりも高速にデータファイルのブロードキャスの実行ができている．これは，Omst/BT では転送するデータファイルは分割されその分割されたデータごとに他のランダムなピアに転送されるため効率が良い転送が実現できていると考えられる．また，Omst/GF は他の OmniStorage の実装よりも性能が悪い場合がある．これは Gfarm 側のファイルアクセスのスケジューリングによって，よりバンド幅の狭いマスタ側のプロセスとワーカプロセス間のネットワーク通信回数が他の実装よりも多いためである．

#### 5.2.3 全ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャスト

全ワーカにおいて他のワーカでの計算結果を収集し次の結果で使用する場合を想定したベンチマークを作成して性能評価を行う．このベンチマークプログラムは，全ワーカが保持している 1 つのあるサイズのデータファイルを他の全ワーカに対してデータファイルを転送させる．いっせいにデータ通信を始めてそれぞれのワーカのデータファイル取得が終了する時間を測定する．

```
/* master program */
int main(){
 ...
 t_start = getTime();
 for(i = 0; i < noNodes; i++)
  r[i] = OmniRpcCallAsyncByHandle(h[i], "a2at_file_get"
                                        filepath[i]);
 OmniRpcWaitAll(noNodes, r);

 for(i = 0; i < noNodes; i++)
  for(j = 0, k = 0; j < noNodes; j++){
   if(i != j)
    r[k++] = OmniRpcCallAsyncByHandle(h[j], "a2at_file_put",
                                           filepath[i]);
   OmniRpcWaitAll(noNodes-1,r);
  }
 t_end = getTime(); calcExecTime(t_start,t_end);
 ...
}

/* worker's IDL */
Define a2at_file_get(OUT filename fp){
  fp = "owndata.dat";
}
Define a2at_file_put(IN filename name){
  /* nothing to do */
}
```

**図 17** OmniRPC のみを用いた全ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストするプログラム
Fig. 17 OmniRPC program code that makes each worker exchange their own file each other.

```
/* master program */
int main(){
 ...
 t_start = getTime();
 for(i = 0; i < noNodes; i++)
  r[i] = OmniRpcCallAsyncByHandle(h[i], "a2at_wput",
                                        i, omst_id[i]);
 OmniRpcWaitAll(n, r);

 for(i = 0; i < noNodes; i++)
  r[i] = OmniRpcCallAsyncByHandle(h[i], "a2at_wget",
                                        noNodes, omst_id);
 OmniRpcWaitAll(n, r);
 t_end = getTime(); calcExecTime(t_start,t_end);
 ...
}

/* worker's IDL */
Define a2at_wput(IN int id, OUT string omst_id){
 my_id = id; asprintf(&omst_id, "file-%d", id);
 r = OmstPutFile(omst_id, "bdata.dat", OMST_BROADCAST);
 OmstWait(r);
}
Define a2at_wget(IN int n, IN string omst_id[n]){
 for(i = 0, j = 0; i < n ; i++)
  if(my_id != i)
   r[j++] = OmstGetFile(omst_id[i], path[i]);
 OmstWaitAll(n-1, r);
}
```

**図 18** OmniStorage を用いた全ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストするプログラム
Fig. 18 OmniRPC program code with OmniStorage that makes each worker exchange their own file each other.

OmniRPC のみを用いたときと OmniStorage を用いたときの使用したベンチマークコードの断片をそれぞれ，**図 17**，**図 18** に示す．OmniRPC のみを用いる場合には，マスタで非同期 RPC（a2at_file_get）を行い，引数としてファイルを全ワーカから取得する．その後に取得したファイルを複数の非同期 RPC（a2at_file_put）を用いてワーカに転送する．OmniStorage も用いる際には，マスタから非同期 RPC（a2at_wput）を実行することにより，ワーカにノード番号を設定しそのノード番号を基にした識別子でワーカが持つファイルを OmniStorage に登録する．また RPC の引数としてその識別子をマスタに渡す．非同期 RPC の待合せによって全ワーカの OmniStorage へのファイルの登録を完了させる．次に非同期 RPC（a2at_wget）で，各ワーカにそれぞれが取得するファイルの識別子の配列を渡し，全ワーカがほぼ同時に OmniStorage に登録された他ワーカが保持するファイルの取得を行う．マスタで非同期 RPC の終了を待つことにより，全ワーカにおけるファイルの取得を完了させる．このベンチマークの実行時間は，マスタ側で各ワーカのデータの登録と取得を行う非同期 RPC の処理の時間（t_start，t_end の差分）を測定している．

ClusterA で動作する全ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストするための時間と，ClusterA と ClusterB でそれぞれ動作する全ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストするための時間を，それぞれ**図 19** と**図 20** に示す．性能評価の比較として OmniRPC のみを用いてデータファイル転送した際の実行時間もあわせて示す．

実験結果より，ClusterA の全 16 ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストする場合，OmniRPC のみを利用するときよりも，Omst/GF は最大 21.7 倍，Omst/BT は最大 7.0 倍高速に実行できている．また，ClusterA の ClusterB で動作する全 16 ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストする場合，Omst/GF は最大 16.4 倍，Omst/BT は最大 7.0 倍高速にできている．また，いずれのデータサイズ，クラスタ構成においても，Omst/GF の方が Omst/BT より 2 倍程度高速であることが分かった．

OmniRPC のみの場合は，マスタプログラムのデータ転送がボトルネックとなり，さらに，マスタプログラムからのデータ転送量を大量に必要になるため，性能向上が妨げられた．BitTorrent の方が遅くなった原因としては，各ピアに対してリクエストされたデータの所在を知らせるトラッカのピア選択アルゴリズム，または各ピアのチョーキングアルゴリズムに問題があるか，トラッカとピアの動作を設定するためのパラメータの設定が不適切であることが原因であると考

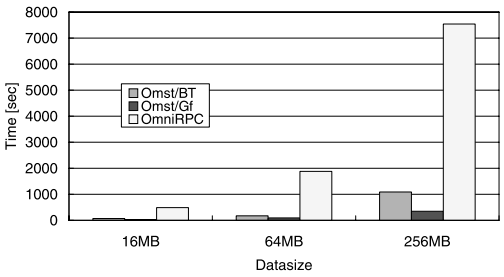


**図 19** ClusterA で動作する全ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストする実行時間

Fig. 19 Elapsed time of program execution that makes each worker exchange their own file each other in ClusterA.

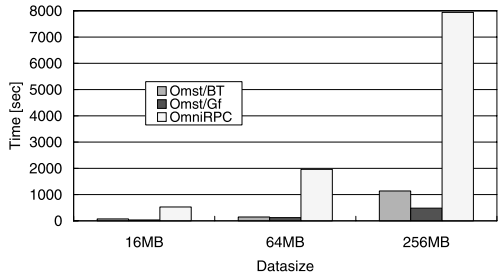


**図 20** ClusterA と ClusterB で動作する全ワーカが自分が保持するデータファイルを他の全ワーカにブロードキャストする実行時間

Fig. 20 Elapsed time of program execution that makes each worker exchange their own file each other in ClusterA and ClusterB.

えられる．今回の実験では，クライアントの実装として libtorrent，トラッカの実装として azureus を利用しており，そのなかの BitTorrent のパラメータはデフォルト値を使用した．これらの値を適切にチューニングすることによって現状よりも性能は向上すると考えられる．さらに，BitTorrent では，配布データを多数のノードに速やかに分散させるために，特定のピアに帯域を独占させず，ピアごとに一定転送量に達すると帯域制限（チョーキング）を行っている．しかし，すべてのネットワーク構成に対応した最適なチョーキングアルゴリズムは現状存在しない．BitTorrent が想定するネットワーク構成はインターネット上に非常に多数（数百から数万ノード）のピアが存在するような環境であり，本稿における実験環境とはかなりかけ離れているため，本来の転送性能が発揮できていない可能性が考えられる．今回の実験では，各ノードの転送量はネットワークの最大バンド幅をかなり下回っていることが分かっている．

### 5.3 実アプリケーションを用いた OmniStorage の性能向上への寄与

Grid RPC の実アプリケーションを用いて，OmniStorage のアプリケーションへの性能向上への寄与とスケーラビリティについて調べるために，並列固有値計算プログラムを OmniStorage も用いるように変更し性能評価を行った．

この実験については，OmniStorage のアプリケーションのスケーラビリティに対する効果を見るために，台数の多い ClusterC クラスタを使用して性能評価を行っている．

並列固有値計算プログラム[4] を利用して，計算ノード数の増加におけるスケーラビリティの評価を行った．このプログラムは，大規模な一般固有値問題を解くもので，複素空間上の円周上の点に対応する方程式を並列に解くことにより，その円周内にある固有値を効率的に求めるプログラムである．詳しくは，櫻井らの論文[4] を参照されたい．なお本実験における固有値計算ジョブの総数は 80 個である．また，マスタからワーカに転送される初期化データの大きさは約 50 MB である．基本的に初期化データを全ワーカにブロードキャストすることを行うため，データ転送レイヤとしてこの通信パターン向けのツリー上にブロードキャスト Omst/Tree を使用することとする．

OmniRPC を用いたオリジナルのプログラムコードから，OmniStorage を利用するために変更した点は，RPC で全ワーカに転送されていた 7 つのデータを OmniStorage で管理するようにしたことである．このためのプログラムコードの変更に関しては，マスタ側では 7 行，ワーカ側では 8 行である．

**図 21** に，1 つのクラスタ内で使用する計算ノードの数を変化させたときの全実行時間を比較した結果を示す．クライアントホストには cTsukuba，クラスタは ClusterC を用いた．広域データ管理レイヤとして Omst/Tree を使用した．折れ線グラフは 1 ノードの実行時間に対する性能向上比を表す．これより，OmniStorage を併用することにより 64 台程度までの台数において性能向上が得られることを確認できた．これにより，OmniStorage は並列固有値計算プログラムへの有効性を確認できた．また，OmniRPC のみでは全 64 ノードを有効に活用できなかったことに対し，OmniStorage を併用することによって 64 ノードを有効に活用できていることが分かった．つまり，OmniStorage はアプリケーションのスケーラビリティを向上させることに成功している．ただし，64 ノードのときに性能向上比の傾きが鈍っている．これは固有

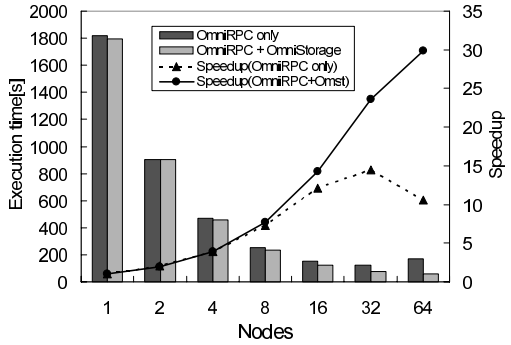


**図 21** ノード数による並列固有値計算プログラムの実行時間の変化
Fig. 21 Execution time of parallel eigen solver program and its scalability.

値計算ジョブの実行時間のばらつきと，使用したデータの並列度が 80 しかなく，64 ノードを有効に活用するためのジョブを各ノードに割り当てられなかったためである．

## 6. 議　　論

OmniStorage の基本的な性能評価から，Grid RPC アプリケーションにおいて使用されるデータの通信パターンから最適なデータ配送レイヤの選択は以下のように結論できる．

- 1 ワーカから 1 ワーカにデータ転送
  ワーカ間での 1 対 1 のデータの授受を行うデータ通信のパターンに関しては，Omst/GF を選択した方が良いと結論できる．Omst/BT は 1 ピア対 1 ピアの場合 Bittorrent プロトコルがネットワーク通信の効率化のためにはうまく働かず，効率の良いデータ通信ができない．そのため Omst/GF の方が Omst/BT の約 7 倍，OmniRPC 単体だけの場合より約 2 倍効率的にデータ通信ができている．
- 初期化ファイルをマスタから全ワーカに転送
  使用するクラスタへの途中経路や中間ノード構成などネットワークトポロジが既知であるような場合に関して，Omst/Tree は他のデータ配送レイヤよりも約 3 倍高速に転送できている．このような場合に関しては Omst/Tree を選択した方がよいと考えられる．しかし，現在の Omst/Tree の実装ではファイルを分割して転送していないため，パイプライン的にデータ転送ができないような場合にはデータ転送の効率が悪化する．また，ネットワークトポロジの推定が不可能であるような場合，最適なツリートポロジを生成できないため，データ通信を効率良くできない可能性がある．このような場合には，Peer-to-Peer 環境を想定して作成された Bittorrent をデータ配送レイヤとして利用する Omst/BT を選択した方が良いことも考えられる．さらに，1,000 台規模の計算機環境を考慮すると，Omst/Tree はネットワークトポロジを推定するための設定が煩雑になり，設定に関する作業が増えることが考えられる．このような場合に，この煩雑な設定が不要である Bittorrent を利用した Omst/BT を選択した方がよいであろう．
- 全ワーカにおいて各ワーカの保持するデータを自分以外の全ワーカにデータ転送
  全ワーカにおいて各ワーカの保持するデータを自分以外の全ワーカにデータ転送する通信パターンに適したデータ配送レイヤは Omst/GF である．Omst/GF は Omst/BT よりも 2 倍ほど効率良く通信ができていることが分かった．5.2.3 項で述べたように，Omst/BT の性能が良くなかったのは Bittorrent のプロトコルもしくは実装に起因するものだと考えられる．ただし，本来ファイル交換にパラメータがチューニングされているため，この Bittorrent のパラメータを調節することにより大幅に改善できる可能性もある．

OmniStorage に登録するデータのヒント情報を活用することによる，データ転送レイヤの選択とデータの転送の効率化について議論を行う．ヒント情報を登録するデータに付加させることにより，OmniStorage 側にデータ通信パターンの情報を伝えることができる．このためパターンの種類にあったデータ配送レイヤを選択することが可能になり，さらに広域環境においてのネットワークトポロジを考慮した通信が可能となり，効率の良いデータ通信が可能になる．仮に，データレポジトリにデータに関するヒント情報を用いないことを想定すると，そのデータの扱いや転送パターンなどの情報を使用することができない．単純に Point-to-Point の通信となってしまい，プロセスが協調しあって効率的なデータ通信を実現することは難しい．つまり，登録するデータにヒント情報を指定することにより効率的なデータ配送ができるのではないかと考えられる．ただし，登録するデータに対するヒントから最適なデータ転送レイヤの自動選択にはネットワークトポロジの推定や自動設定などを解決する必要がある．

## 7. 関 連 研 究

Grid RPC システムにおいて，マスタとワーカプロセス間で共通に使用または巨大なデータを効率良く管

理しアプリケーションの性能向上をねらう試みがいくつかある．

Grid RPC の一実装である NetSolve[13] では，分散ストレージインフラストラクチャである Internet Backplane Protocol（IBP）[14] を用いることで，マスタと計算ノード間で共通に使用されるデータを IBP 側で管理し，またはデータのキャッシュを行いデータ通信量の削減とデータ通信の最適化を行っている[15]．また，1 つの計算ノードでの実行結果のデータを別の計算ノードでも使用するような処理の場合，結果のデータを一度マスタに送り返すことをせずに，ワーカ間でデータ通信を行うようなことも可能である．OmniStorage では，NetSolve と比較して，複数のデータ転送レイヤを利用可能である，またさらに使用されるデータがどのように転送されるかについてのヒント情報を用いることにより，効率の良いデータ転送レイヤを選択することが可能である．また OmniStorage では，ある名前空間に対しデータやファイルを Put/Get するように簡便にプログラミングできるようなのアプローチをとっている．しかし，NetSolve と IBP を用いる場合は，一般的なファイルの操作を行うように，ホスト名の指定やファイルをパスを指定し，ファイルポインタになどの低レベルなプログラミングを行うアプローチをとっている．

また NetSolve の Request Sequencing では，データ依存性のある複数の RPC を 1 つの RPC 群としてまとめ，その RPC のデータ依存をライブラリ内部で解析を行いデータの DAG を作成，計算ノード間でデータ転送を行うようにしている．これによって，不要なデータをマスタに戻さず，効率的なデータのステージングを可能としている[16]．

DIET[17] でも同じように，計算ノード間でのデータ通信を行い，余計なマスタへの通信を省くデータ管理レイヤ Data Tree Manager を実装している．さらに，遠隔地でのデータの永続性を扱うための機構も有しており，操作するデータに永続性の属性をつけることによりシステム内でのデータの扱いの処理を変化させている．また，別のデータ管理レイヤとして，P2P 環境向の通信ライブラリである JXTA[18] 上に DSM 空間を構築した Juxmem[19] を用いる試みが行われている[20]．DIET では，DIET の機能でデータ転送効率良く行うようなアプローチをとっているが，OmniStorage では，複数のデータ転送レイヤのなかからデータ転送パターンによって効率の良いレイヤを選択できるようなアプローチをとっている．

谷村らは Ninf-G において，データの依存関係を持つ 2 つ以上の RPC を，マスタを介さずに RPC の実行ノード間で直接データ転送をする Task Sequencing を，分散ファイルシステムである Gfarm をデータレポジトリとして利用することにより実現している[21]．対象とするデータはファイル型のみで，Task Sequence としてまとめた RPC 群の中からデータファイルの依存を解析し，計算ノード間での依存するファイルの授受を Gfarm を介して行うようにしている．OmniStorage では，ファイル型のデータだけではなく配列も扱うことが可能である．さらに，データの授受の種類によって複数のデータレイヤのミドルウエア中から最適なミドルウエア選択できる．さらに Task Sequence 内部だけのデータを扱っているだけで，柔軟性は低い．

Wei らは，マスタからファイルを全ワーカへブロードキャストを行う際に転送に用いるプロトコル（FTP と Bittorrent）を変更して，ブロードキャストにかかる実行時間を比較している[22]．ノード数が 15 台以上かつ転送されるファイルサイズが 50 MB 以上の場合，ブロードキャストに関して BitTorrent よりも FTP の方が効率良く実行できると報告している．しかし，FTP と BitTorrent との比較で終始しており，他のプロトコルとの比較については記述がない．

## 8. まとめと今後の課題

Grid RPC アプリケーションのために RPC レイヤからデータ通信とプロセスのコントロールのためのデータ通信を分離し，アプリケーションに共通に使用可能，かつ効率的なデータ配送を行うデータ管理レイヤ OmniStorage を設計・実装した．ネットワークトポロジを考慮しツリー状にデータのブロードキャスト転送を行うミドルウエア，Peer to Peer を用いたファイル転送ソフトウエアである BitTorrent と広域分散ファイルシステムである Gfarm 上に実装した．Grid RPC アプリケーションをモデル化したベンチマークを用いて，各データ転送レイヤ上に実装した OmniStorage の基本的な性能評価を行った．さらに，実 Grid RPC アプリケーションを OmniStorage を併用するように変更し性能評価を行ったところ，RPC 単体の機能を利用するときよりも提案システムも併用することにより性能面やスケーラビリティに関して優位性があることが分かった．Grid RPC アプリケーションでのブロードキャストや one-to-one，all-to-all などの必要な通信パターンを基に，データ転送レイヤを選択することが性能面やスケーラビリティに関して有効なことが分かった．

今後の課題としては，広域データ管理レイヤのセ

キュリティの強化があげられる．現在の実装ではネットワーク経路で通信の暗号化は行っておらず，またOmniStorage のネットワーク接続に関しては認証を行っていないため，これらの改善が課題としてあげられる．また，自動的な広域ネットワークのトポロジ推定があげられる．さらに効率的なデータ転送を行うためには，ネットワークトポロジの推定は必須であり，かつ，経路選択についても何らかの機構を入れる必要がある．さらに，各データ転送レイヤのチューニングが考えられる．現在のところ各データ転送レイヤのデフォルトの設定を使用しているが，同時コネクション数などのパラメータを設定することにより，より効率的なデータ転送が可能になる可能性がある．

**謝辞** 本研究の一部は，文部科学省科学研究費補助金基盤研究（A）課題番号 17200002，特別研究員奨励費 課題番号 177324，特定領域研究課題番号 19024009 および，日仏共同研究プログラム（SAKURA）による．

## 参 考 文 献

1) 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 (ACS 3), pp.34–45 (2003).
2) Nakajima, Y., Sato, M., Boku, T., Takahashi, D. and Gotoh, H.: Performance Evaluation of OmniRPC in a Grid Environment, *2004 Symposium on Applications and the Internet Workshops*, pp.658–665 (2004).
3) Nakajima, Y., Sato, M., et al.: Implementation and performance evaluation of CONFLEX-G: grid-enabled molecular conformational space search program with OmniRPC, *Proc. 18th Annual International Conference on Supercomputing*, pp.154–163 (2004).
4) 櫻井鉄也，多田野寛人，早川賢太郎，佐藤三久，高橋大介，長嶋雲兵，稲富雄一，梅田宏明，渡邊寿雄：大規模固有値問題の master-worker 型並列解法，情報処理学会 ACS 論文誌，Vol.46, No.10, pp.1–8 (2005).
5) 相田祥昭，中島佳宏，佐藤三久，櫻井鉄也，高橋大介，朴 泰祐：グリッド RPC における広域データ管理レイヤの利用，先進的計算基盤システムシンポジウム SACSIS 2006, pp.85–92 (2006).
6) Ninf Project. http://ninf.apgrid.org/
7) BitTorrent. http://www.bittorrent.com/
8) libtorrent. http://libtorrent.sf.net/
9) 建部修見，森田洋平，松岡 聡，関口智嗣，曽田哲之：ペタバイトスケールデータインテンシブコンピューティングのための Grid Datafarm アーキテクチャ，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，Vol.43, No.SIG 6 (HPS 5), pp.184–195 (2002).
10) Qiu, D. and Srikant, R.: Modeling and performance analysis of BitTorrent-like peer-to-peer networks, *SIGCOMM '04: Proc. 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, pp.367–378, ACM Press (2004).
11) Bharambe, A.R., Herley, C. and Padmanabhan, V.N.: Some observations on bitTorrent performance, *SIGMETRICS '05: Proc. 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, pp.398–399, ACM Press (2005).
12) Ogura, S., Matsuoka, S. and Nakada, H.: Evaluation of the inter-cluster data transfer on Grid environment, *CCGRID '03: Proc. 3rd International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, p.374, IEEE Computer Society (2003).
13) Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Seymour, K., Sagi, K., Shi, Z. and Vadhiyar, S.: Users' Guide to NetSolve V1.4.1, Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN (2002).
14) Bassi, A., Beck, M., Moore, T., Plank, J., Swany, M., Wolski, R. and Fagg, G.: The Internet Backplane Protocol: A study in resource sharing, *Future Generation Computer Systems*, Vol.19, No.4, pp.551–561 (2003).
15) Beck, M., Dongarra, J., Huang, J., Moore, T. and Plank, J.: Active logistical state management in gridsolve, *4th International Symposium on Cluster Computing and the Grid* (*CCGrid 2004*), IEEE Computer Society (2003).
16) Arnold, D., Bachmann, D. and Dongarra, J.: Request Sequencing: Optimizing Communication for the Grid, *Proc. 6th European Conference on Parallel Computing* (*Euro-Par 2000*), pp.1213–1222 (1900).
17) Del-Fabbro, B., Laiymani, D., Nicod, J.-M. and Philippe, L.: Data Management in Grid Applications Providers, *DFMA '05: Proc. 1st International Conference on Distributed Frameworks for Multimedia Applications* (*DFMA'05*), pp.315–322 (2005).
18) Gong, L.: Industry Report: JXTA: A Network Programming Environment, *IEEE Internet Computing*, Vol.5, No.3, pp.88– (2001).
19) Antoniu, G., Bougé, L. and Jan, M.: JuxMem:

An adaptive supportive platform for data sharing on the grid, *Scalable Computing: Practice and Experience*, Vol.6, No.3, pp.45–55 (2005).
20) Antoniu, G., Caron, E., Desprez, F. and Jan, M.: Towards a Transparent Data Access Model for the GridRPC Paradigm, Technical Report RR-6009, INRIA (2006). Also available as IRISA Research Report PI1823.
21) 谷村勇輔，中田秀基，田中良夫，関口智嗣：GridRPC における複数ノードにまたがる Task Sequencing の実現，先進的計算基盤システムシンポジウム SACSIS 2006, pp.75–84 (2006).
22) Wei, B., Fedak, G. and Cappello, F.: Scheduling Independent Tasks Sharing Large Data Distributed with BitTorrent, *The 6th IEEE/ACM International Workshop on Grid Computing*, pp.219–226 (2005).

(平成 19 年 1 月 22 日受付)
(平成 19 年 5 月 16 日採録)

**相田　祥昭**

昭和 57 年生．平成 17 年筑波大学第三学群情報学類卒業．平成 19 年同大学大学院システム情報工学研究科修士課程修了．同年沖電気工業株式会社入社，現在に至る．

**中島　佳宏**（学生会員）

昭和 55 年生．平成 15 年筑波大学第三学群情報学類卒業．現在，同大学大学院システム情報工学研究科在学中．グリッドコンピューティングに関する研究に従事．

**佐藤　三久**（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年通産省電子技術総合研究所入所．平成 8 年新情報処理開発機構並列分散システムパフォーマンス研究室室長．平成 13 年より，筑波大学システム情報工学研究科教授．同大学計算科学研究センター勤務．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術，グリッドコンピューティング等の研究に従事．IEEE，日本応用数理学会各会員．

**建部　修見**（正会員）

昭和 44 年生．平成 4 年東京大学理学部情報科学科卒業．平成 9 年同大学大学院理学系研究科情報科学専攻博士課程修了．同年電子技術総合研究所入所．平成 17 年独立行政法人産業技術総合研究所主任研究員．平成 18 年筑波大学大学院システム情報工学研究科助教授．博士（理学，東京大学）．超高速計算システム，グリッドコンピューティング，並列分散システムソフトウェアの研究に従事．日本応用数理学会，ACM 各会員．

**櫻井　鉄也**（正会員）

昭和 36 年生．昭和 61 年名古屋大学大学院工学研究科博士課程前期課程修了．同年名古屋大学工学部助手．平成 5 年筑波大学電子・情報工学系講師．平成 8 年同大学助教授．平成 17 年より，筑波大学大学院システム情報工学研究科教授．東京大学，慶應義塾大学非常勤講師．独立行政法人産業技術総合研究所客員研究員．博士（工学）．大規模固有値問題の並列解法，方程式の反復解法と精度保証，数理ソフトウエアの利用支援環境の研究に従事．平成 8 年日本応用数理学会論文賞受賞．日本応用数理学会，日本数学会各会員．