情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 2　157–168 (Aug. 2008)

複雑なグリッド環境で柔軟なプログラミングを実現するフレームワーク

弘中　健[†1]　斎藤秀雄[†1]
高橋　慧[†1]　田浦健次朗[†1]

大規模な広域計算機環境での計算支援フレームワークは，接続性の問題（NAT, firewall）やスケーラブルな接続管理のほか，動的なプロセスの参加・脱退，通信・ノードの耐故障性が重要である．本稿では分散オブジェクト指向モデルを拡張することで，従来の言語の柔軟性を保ちながらこのようなフレームワークを実装した．モデルでは非同期なメソッド呼び出しを用いて並列性を表現し，逐次プログラムからの飛躍が小さい．その一方，プロセスの動的参加，非同期呼び出し返り値などの非同期イベント処理で，デッドロックを防ぎつつロックなどのプリミティブを不要とする，オブジェクトへのアクセスを逐次化するセマンティクスを導入する．通信は参加プロセス間で自動的に TCP オーバレイを構築し，接続性，スケーラビリティの問題を解決する．Python のライブラリとして実装することで短期間でのアプリケーション開発を実現する．評価では多様なネットワーク環境を含む 9 クラスタ，合計 900CPU コアを用いて branch-and-bound 探索を使うアプリケーションが容易に開発できることを示した．

A Framework for Flexible Programming in Complex Grid Environments

Ken Hironaka,[†1] Hideo Saito,[†1] Kei Takahashi[†1] and Kenjiro Taura[†1]

Problem-solving frameworks in large-scale and wide-area environments must handle connectivity issues (NAT and firewalls), maintain scalability with respect to connection management, accommodate dynamic processes joining/leaving at runtime, and provide simple means to tolerate communication/node failures. This paper designs and implements such a framework by minimally extending distributed object-oriented models for maximum *generality* and *flexibility*. In the framework, parallelism is expressed via asynchronous method invocations to allow a natural transition from sequential programs. To cope with asynchronous events such as dynamic process joins and asynchronous method invocation returns, we introduce an implicit serialization semantics on objects to relieve programmers from explicit synchronization primitives while avoiding recursion deadlock problems. In our implementation, participating nodes automatically construct a TCP overlay so as to address connectivity and scalability issues. We have implemented our framework as a library for Python to allow rapid development of complex workflows and to maximally leverage the richness of its libraries. For evaluation, we show on over 900 cores across 9 clusters with complex networks (involving firewall and NATs) and process managements (involving SSH, torque, and SGE) configurations, how a simple branch-and-bound search application can be expressed simply and executed easily.

1. はじめに

グリッド環境でのプログラミングは非常に複雑である．計算資源は大規模なうえ，複数のクラスタにまたがる．資源間の通信は NAT/firewall などによって妨げられ，各クラスタはそれぞれ異なったポリシで管理されており，利用可能な資源は随時変動する．また大規模で広域な環境では，利用中に計算資源，ネットワークの障害に遭遇する確率は高い．したがって，グリッド環境でのプログラミングを支援するフレームワークは以下の要件を満たす必要がある．

- 下層の複雑なネットワーク環境を隠蔽し，グリッド上に分散するプロセスの通信を実現する．
- 新たに利用可能になった計算資源の進行中計算への参加およびその簡便なハンドリングを実現する．
- 計算資源やネットワークの障害に対応する．

以上の要点を満たしながら，ユーザ，またはプログラマに大きな負担を課さない簡潔なフレームワークが必要である．

多くのグリッド環境用フレームワークが提案されているが，これらはプログラミングの柔軟性を犠牲にしている．たとえば，バッチスケジュラと呼ばれるシステム[1)–4)] では，サブタスク間の依存性がないアプリケーションを容易に並列実行できる．これをタスク間での

†1 東京大学
The University of Tokyo

依存性の記述を可能にするよう拡張[5),6)] したモデルも存在する．これらのアプローチでは，逐次プログラムをそのまま分散環境で実行できる．より一般的なアプローチとして，ある種類のアプリケーションに特化したプログラミングフレームワーク[7),8)] などもある．これらのフレームワークは単純で，グリッド環境にかかわる上記の問題をユーザやプログラマにまったく透過的に見せることが利点である．しかしこれらのアプローチの場合，サブタスク間の協調，通信は非常に限られた方法と局面のみ可能である．フレームワークが提供する方法以外のタスク間の通信を実現したい場合，ユーザは独自にソケット，ファイルなどを用いた ad-hoc な解決法を迫られる．

一方で，分散オブジェクト指向言語と RMI（Remote Method Invocation）というプロセス間通信機構を用いるフレームワークは，記述の柔軟さは実現されているものの，同期機構の管理などにおいてプログラマに大きな負担を課してしまう．また，大規模なグリッド環境で展開するときのスケーラビリティおよび NAT/firewall への対応を考慮した実装は数少ない[9)]．

我々は，逐次のオブジェクト指向プログラミング言語が持つ表現の柔軟性を保ちつつ，グリッドで高性能な並列・分散処理を可能にするプログラミングフレームワークを提案する．これは Python のライブラリとして実装されており，具体的には以下のような特徴を持つ．

- プログラム内の並列性は非同期呼び出しと *futures*[10),11)] と呼ばれる機構によって実現される．多数のプログラミングモデル[12),13)] に採用された経緯があるほか，逐次プログラミングからの飛躍が少ないという利点がある．
- 多くの場合で排他制御を必要としない新たなオブジェクトモデルを提案する．このモデルは *passive object* により実現され，*active object* モデル[9),14)] に付随する再帰呼び出しによるデッドロックなどの問題を回避する．
- オブジェクトへの非同期イベントの通知という概念を導入することにより，動的資源増加にともなうプロセスの参加を自然に記述することが可能である．したがって，既存に非同期イベントのためのイベント処理ループなどといった低レベルの処理を排除する．
- RMI にともなう通信は，参加プロセス間で TCP のオーバレイネットワークを構築することにより，NAT/firewall を含むシステム内でもスケーラブルかつ透過的に実現する．特に SSH のみでアクセス可能な資源も導入するために，SSH portforwarding も用いることが可能である．

我々が用いた実験環境は 9 クラスタにまたがる 900CPU コアからなり，多くのクラスタは IP filtering やプライベート IP アドレス空間を持つ．一部のクラスタにアクセスするには，多段の SSH に加え，バッチスケジューラを用いてプロセスを起動する必要がある．我々は提案するフレームワークを用いた組合せ最適化問題のソルバを実装し，これら 900CPU コアを用いて実行した．ソルバの本体は C++で記述された，第 3 回 Grid Plugtest 大会[15)] 優勝プログラムである．我々の Python フレームワークは実験環境でのプログラムを起動，およびソルバ間の暫定解の交換に使われている．この繋ぎ合わせコードはわずか 250 行で，その骨組を 3 章に示す．この実行において複雑な事前設定などは必要なく，容易に全コアを使うことができた．

以降，2 章で関連研究を紹介し，3，4 章で提案するモデルと実装を記す．5 章で評価を示し，6 章でまとめる．

2. 関 連 研 究

いくつものグリッド用フレームワークが提案されてきたが，グリッドにかかわる接続性，資源の増減といった問題をふまえながら，アプリケーションの構築を柔軟にするシステム・プログラミングモデルを提供することは容易ではない．

2.1　資源間の柔軟な通信と協調の実現

ともに計算をする資源があらゆる場面で通信と協調ができることは，効率な計算を実現するうえで重要である．ユーザが記述した個々のサブタスクをスケジューリングするものとしてバッチスケジューラ[2),4)] が有名で，なかにはサブタスク間の依存関係の定義を可能にする拡張もある[5)]．しかし，サブタスク間の協調手段が非常に限られており，柔軟性も応用アプリケーションの範囲も狭い．通信の媒体としてファイルへの入出力が多く用いられ，それがゆえに順番に実行されるタスクがファイルを介して入出力データを受け渡すにとどまっている．

マスタ・ワーカモデルでは，マスタ・ワーカ間の通信を可能にし，より密なサブタスク間の協調を可能にする．理解しやすいモデルとして広く受け入れられており[16)]，このモデルに特化したフレームワークも少なくない[17)–20)]．しかし，これらの手法においてもマスタとワーカが通信をする機会がタスクの開始と終了時に限られており，ワーカがマスタと頻繁に通信したい場合，細粒度にタスクを分割する必要がある．なかには，定期的にマスタとワーカがメッセージを用いた通信を可能にするフレームワークもあるが，各ノードは 1 つ 1 つのメッセージごとに処理を分岐することが必要になり，煩雑になる．

分散環境で分割統治法を用いたフレームワークとして Satin[13)]，distributed-Cilk[12)] などがある．しかし問題解法が分割統治法という大きな束縛があり，タスクの分割，統合時以

外の通信は不可能で，適用できる問題集合に限られている．

それに対して，我々は分散オブジェクト指向のプログラミングモデルを採用する．通信はRMI という形で隠ぺいされる一方，任意のタイミングで任意の分散オブジェクトどうしが互いにを呼び出し合うことができる．future を用いて容易に非同期なメソッド呼び出しができ，資源間で自由な協調と通信が可能である．

2.2　並列 RMI における排他制御の最小化

既存のオブジェクト指向プログラミング言語をベースとし，分散環境で並列計算を行うフレームワークがある．Java では Ibis RMI[21] や ProActive[9]，Python では DisPyte[22] などによって代表される．これらの手法では，分散オブジェクトに対してスレッドによるRMI 呼び出しや非同期 RMI を用いて並列計算を実現する．しかし，この方法では複数スレッドの競合，非同期実行によるレースコンディションが多発し，クリティカルなデータへのアクセスはロックを用いて排他制御する必要がある．なかには 1 オブジェクトに対して 1 つの占有スレッドを割り当て，排他制御を不要にする active object[9] モデルがある．しかし，この方法では逆に再帰呼び出しなどによるデッドロックを招く．

それに対して，我々は future を用いた新しい同期モデルを提案する．これでは 1 度にオブジェクトを占有できるスレッドを 1 つに限っているが，そのスレッドがブロックする際には他のスレッドが占有可能になるため，デッドロックが生じない．

2.3　低オーバヘッドな資源の参加・脱退への対応

グリッド環境では計算資源が計算途中に参加，脱退することを考慮する必要があるが，それがユーザに大きな負荷になってはならない．Satin[13] では，プログラミングモデルが分割統治法であることを利用し，失われた部分問題は透過的に再実行することでノードの故障に対応し，負荷分散には Random Work-Stealing[23] を用いている．しかし，この方法は分割統治法以外のモデルで適用することは難しい．

Jojo2[17] のようなマスタ・ワーカ型フレームワークでは，ワーカ計算ノードの参加，脱退に対する処理を記述することができるが，これはユーザに非常に低レベルなイベント処理（ノードの参加，脱退）を自分で排他制御を行いながらすることを迫り，コードが複雑化する．

本提案では，非同期に発生するノードの参加のようなイベントの処理をオブジェクトへのシグナルという形で記述することが可能である．また分散環境でも，故障による RMI の失敗を例外という形で検出する．これらは我々が提案する，明示的な排他制御を最小限にとどめるプログラミングモデルと合わせて活用することができ，資源の参加脱退を考慮することによるコードの煩雑化を防ぐ．

2.4　グリッド環境における NAT/firewall 越え

グリッド環境で RMI などの通信を実現するうえで，NAT/firewall による計算資源間の接続性の障害へのスケーラブルな対応が非常に重要である．分散プログラミングで多様なネットワークハードウェア上での動作を可能にするものとして，CORBA を使った PadicoTM[24] があるが，我々が想定する NAT/firewall などのネットワーク環境での起動には対応していない．ProActive[9] ではグリッド環境での通信にはユーザが接続可能点を記述する設定ファイルをもとに接続を確立し，グリッド計算を実現する．その一方，複雑な設定ファイルで計算・ネットワーク資源を記述する必要があり，大きな問題になる．我々の提案では，このような複雑設定ファイルを避け，ユーザレベルのオーバレイを構築し，自動的にグリッド上の接続性の障害を解消しており，各計算資源は少数の接続を確立するスケーラブルな提案をする．

3.　プログラミングモデル

我々はグリッド環境の NAT/firewall による接続性，計算資源の増減の問題をふまえた分散オブジェクト指向プログラミングモデルを提案する．他の分散オブジェクトフレームワークと同様に，通信はオブジェクトに対する RMI として行われ，記述の柔軟性を保つ．一方，単純な RMI では解決されないグリッド環境への対処もモデルレベルで導入している．任意のオブジェクトへの RMI を場所透過に実現し，明示的なロックを必要としないオブジェクトセマンティクス，動的に参加する資源用のブートストラップ機構，故障セマンティクスも提供する．

3.1　排他制御・非同期イベント通知

並列処理を行ううえで非同期 RMI を使う必要があるが，それの効率的な処理は依然として未解決である．RMI の返り値が戻ると callback 関数が呼ばれるようにする手法もある．しかし，callback は別スレッドで実行されるため，プログラマはロックを使った排他制御をする必要がある．この解決として返り値が必要なときにブロックする future という機構がある．これには，返り値が自動的に設定され，アクセス時にまだない場合に初めてブロックする．呼び出しスレッドは待っている間に別処理をすることができる．利点は単一スレッドで主な処理が行われるうえ，逐次のプログラムからの飛躍も小さいという点である．

その一方，これは RMI 処理側からの問題解決になっていない．RMI の処理は独自のスレッドで処理される必要があるので，RMI で呼び出される可能性があるオブジェクトはや

はりロックで排他制御を行う必要がある．そのため，一部のモデル[9] のように各オブジェクトに専用スレッドがある *active object* を採用することも可能である．しかし，排他制御の必要はなくなる一方，再帰呼び出しで簡単にデッドロックするという問題がある．現実的なマスタ・ワーカのグリッドアプリケーションでも，マスタとワーカが互いに呼び合うというフローはきわめて自然であるが，active object を採用する場合，ユーザの意思に反してデッドロックしてしまう．

別の要件として，計算機資源をまたぐ非同期なイベント処理も重要である．この場合，OS のシグナル機構は使えず，RMI 自身も解決策にならない．具体的な例として，RMI の返り値を求めてブロックしている間に，新しいノードの参加という非同期なイベントに対する処理を行う必要がある場合がある．これは広く受け入れられているマスタ・ワーカモデルでも頻出するようなシナリオである．新たなスレッドを用いたイベント処理を行えば，以前の排他制御問題が生じる．従来のイベント指向プログラミングモデルを用いて RMI の返り値の処理，ノードの参加，ノードの脱退などとすべてのイベントを 1 つのイベントループで処理することも可能である．しかしこれはプログラムの意味的フローを小さく分断し，逐次プログラムからの隔たりも大きい．

我々は従来の RMI 以外にも分散オブジェクト指向モデルに必要な性質をまとめる．

- 並列計算を可能にするために future 機構を提供すること
- オブジェクトをアクセスするときに排他制御を必要としないこと
- 非同期なイベント処理を全体のフローを分断せずに可能にすること

我々が提案するモデルでは，future を提供し，スレッドはそれを使い，ブロックして結果を待つことができる．排他制御を解決するため，オブジェクトを 1 度にアクセスできるスレッドは最大 1 つである．しかし，オブジェクトのスコープ中でブロックした場合，他のスレッドがアクセスすることが許される．最後に，オブジェクトへの非同期イベントの通知機構を備え，スコープ内でブロックしているスレッドをアンブロックする機構を備えている．

非同期な RMI を発行した場合，future をブロックせずに返す．ある `foo` というオブジェクトに `fib` という RMI を発行するとき，以下を実行する．

```
future = foo.fib.future(args)
```

結果を取得するには以下を発行する．

```
result = future.get()
```

結果がまだない場合，呼び出しはブロックする．また，コモンケースとして複数の future に対して結果待ちをする必要がある場合がある．そのために，future のリストを引数にとる `wait` というプリミティブを提供する．最低でも 1 つの future の結果が返るまでブロックし，結果が返った future のリストを返す．

```
readys = wait(futures)
```

最後に，オブジェクトにイベントを通知する機構を備え，あるオブジェクト `obj` に以下のように通知することができる．

```
obj.signal()
```

この結果，`obj` のスコープ内でブロックしているスレッドを最大 1 つ強制アンブロックさせる．その瞬間アンブロックするスレッドがない場合，次回オブジェクトのスコープ内でブロックしたスレッドがアンブロックする．ブロッキングプリミティブが強制アンブロックされた場合，返り値 None を返す．ブロックするときにイベントの通知を受けたくない場合，イベント通知をマスクすることも可能である．

提案するオブジェクトをアクセスできるスレッド数は最大 1 つというモデルは *active object* に類似する．しかしオブジェクトスコープ内でスレッドがブロックする処理（同期 RMI，future に対する結果待ち，`wait`）を行う場合，他スレッドが走行することを許している．具体的には，オブジェクトのスコープに入るときにロックを陰に取得し，スコープを出るとき（他のオブジェクトへの呼び出し）や，future，`wait` に対する処理を行うときに陰にロックを解除する．こうすることで *active object* モデルに付随するデッドロックを回避している．したがって，オブジェクトのアトミックブロックはスコープ内のブロック処理間であり，この間での状態の操作にはロックを必要としない．

アトミックブロックがメソッドの開始から終わりまでの間でないため，一見扱いにくいプログラミングモデルとも思える．しかし，現実で扱うクリティカルセクションのブロックは非常に短く，コモンプラクティスとしてブロックする可能性がある I/O などを挟んで設定することは望ましくない．また，ブロック可能な呼び出し（同期 RMI，future へのアクセス，`wait` の呼び出し）は明白に分かっているので判別はユーザにとっても容易である．デバッグに関しても，active object のように複数の呼び出しの連鎖からなるデッドロックのようなことはなく，データのアクセスを個々のメソッド単位で検査することで済むという意味で優位である．

各オブジェクトに備わっている `signal` のセマンティクスは UNIX シグナルの I/O システムコール（e.g.，read）でブロックしているスレッドをアンブロックするというものに類似する．オブジェクトスコープ内でブロックしているスレッドをアンブロックすることによりイベントを通知し，対処することができる．

ここで，ブロックしているスレッドではなく，それがコンテキスト内にいるオブジェクトへイベントを通知する理由として，以下がある．各オブジェクトへの RMI ごとに別のスレッドが生成されることを考慮すると，個別のスレッドを指名して通知をするということは稀である．イベントを発行する側からは，オブジェクト内で走っているどのスレッドに通知をするかよりも，どのオブジェクトに通知するかということに関心がある．この選択は，オブジェクトのモジュラリティを保つという意味でも望ましい．

3.2 故障セマンティクス

実行時に通信路，計算資源に故障が生じることを想定することはグリッド環境ではきわめて大切である．我々のモデルでは例外というセマンティクスで RMI のエラーに対応している．RMI の実行中で何らかのエラーが発生した場合，`RemoteException` が呼び出し側でレイズされる．エラーには 3 つのシナリオがある．

(1)　RMI 処理側で例外がレイズされる．
(2)　RMI 呼び出し側と RMI 処理側間の通信路が切れる．
(3)　RMI 処理側が故障する．

シナリオ (1) の場合レイズされた例外が，シナリオ (2)，(3) の場合ハードウェアのエラーを通知する例外が呼び出し側に転送され，そこでレイズされる．通信，資源の故障も通常の Python の例外セマンティクスに沿った処理で扱える．

3.3 ブートストラップ

動的に資源が計算に参加することを想定すると，プロセスがブートストラップするすべが必要である．特に分散オブジェクト指向の場合，これは「最初のオブジェクトの参照を得る」ということを意味する．我々のモデルでは，いかなるオブジェクトも任意の文字列の名前でネットワーク全体に「公開」することができる．

```
object.register("any_name")
```

プロセスは公開されたオブジェクトの参照を以下のように取得することができる．

```
ref = RemoteRef("any_name")
```

新たに参加したプロセスはこのようにすでに存在するオブジェクトへの参照を得，これをもとに参加をオブジェクトにイベント通知することができる．たとえば，マスタ・ワーカモデルのアプリケーションではマスタが自分のオブジェクトを公開し，ワーカがマスタへの参照を得ることができる．

3.4 サンプルコード

提案するプログラミングモデルを用いて容易にプロセスが増減する環境でのアプリケーションを記述することができる．その一例としてワーカが増減する環境のマスタ・ワーカアプリケーションの骨組を**図 1** に示す．マスタは `node.work.future()` という非同期呼び出しで，ワーカ `node` に `work` というメソッドでジョブを割り振る．新たなワーカは `signal` を使って新たな参加をマスタに通知する．このとき，`wait` のブロックは解かれ，None が返り，loop に先頭に戻って新しいワーカの追加，ジョブの割振りをすることが可能である．故障には future に返る例外を処理することで対応できる．また，一連の処理にロックをいっさい必要としないこと，future を使うことでマスタがワーカに仕事を配るというセマンティクスがきれいに表現されることに注意されたい．このような記述をロックを不要とする active object モデルで行った場合，マスタは常時メソッド `run` 内を loop しているため，いつまで経ってもワーカはマスタのメソッド `nodeJoin` を実行することができないので，実現不可能である．したがって，提案するモデルが優位であることも分かる．

従来のマスタ・ワーカ型フレームワークの記述で行う場合，擬似コードは**図 2** のようになる．しかし，コードからうかがえるように，1 つ 1 つのノードの参加，ジョブ終了，ノードの脱退においてイベントの種類で分岐をし，それに対応する処理を行う必要がある．また，イベントがいつ発行するか分からず，ぞれぞれ別のスレッドが `handleEvent` を実行することを考慮すると，ロックを使った排他制御が必要になる．したがって，ジョブを投げ，結果を得るという一連の流れが失われてしまう．それに対して提案するモデルでは，1 つのジョブの実行がワーカへの非同期 RMI という形で表現され，関数呼び出しという本来のフローを保っている．

このようなマスタ・ワーカ型のプログラム以外にも，RMI の柔軟性を生かして多種多様なアプリケーションの記述ができる．たとえば，分散環境でのアイランド型 GA が考えられる．このアプリケーションでは，各計算ノードで通常の GA を実行し，ときどき隣接する計算ノードと通信をし，交叉をするというものである．一般に，このように等価な計算ノード達が互いに通信をし，データの共有・交換をするアプリケーションは，各ノードの分散オブジェクトが互いに RMI を呼び出し合うというモデルで非常によく表現することが可能である．

4. 実　　装

グリッド環境での分散オブジェクト指向プログラミングに付随する問題をいかに解決するかを述べる．特に以下の 3 点を述べる：広域環境下でのスケーラブルなポイント–ポイント通信，動的なプロセスの参加，RMI エラー検出．

```
class Master:
  def __init__(self):
    self.nodes = []
    self.jobs = []

  def addJob(self, job):
    self.jobs.append(job)
    self.signal() # notify new job

  def nodeJoin(self, node):
    self.nodes.append(node)
    self.signal() # notify join

  def run(self):
    assigned = {}
    while True:
      # dispatch work to available workers
      while len(self.nodes)>0
            and len(self.jobs)>0:
        node = self.nodes.pop()
        job = self.jobs.pop()
        # asynchrous RMI to worker
        f = node.work.future(job)
        assigned[f] = (node, job)

      # wait for any results
      readys = wait(assigned.keys())
      # if got signal, loop back
      if readys == None: continue

      #read ready results
      for f in readys:
        node, job = assigned.pop(f)
        try:
          print "done:", f.get()
          self.nodes.append(node)
        except RemoteException, e:
          # in case of a fault, rerun job
          self.jobs.append(job)

class Worker:
  def work(self, job):
    #do work on job...

    return results

  def run(self, mastername):
    #obtain reference to master and join
    master = RemoteRef(mastername)
    master.nodeJoin(self)
```

図 1　マスタ・ワーカプログラムのコアを示す．それぞれのクラスの run メソッドを実行することでマスタ・ワーカはそれぞれ初期化される

Fig. 1　The core for a simple Master-Worker Program. Classes for the Master and the Worker are shown. Both the master and the worker initialize by invoking its run method.

```
class Master:
  def __init__(self):
     self.jobs = []
     self.workers = []
     self.tabs = {}
     self.lock = Lock() # for mutex

  #invoked on new event with arg. e
  def handleEvent(self, e):
     #need mutual exclusion
     self.lock.acquire()
     try:
       #give job to new node
       if e.type == NEW_NODE:
           node = e.node
           #give new job, if any
           if len(self.jobs) > 0:
              job = self.jobs.pop()
              self.tabs[node] = job
              self.giveJob(node, job)
           else:
              self.workers.append(node)

       #give new job to master
       elif e.type == NEW_JOB:
           job = e.job
           self.jobs.append(job)
           #assign to master, if any
           if len(self.workers) > 0:
              node = self.workers.pop()
              job = self.jobs.pop()
              self.giveJob(node, job)

       #handle result and give-out a new job
       elif e.type == JOB_DONE:
           print "done:", e.result
           node = e.node
           #give new job, if any
           if len(self.jobs) > 0:
              job = self.jobs.pop()
              self.tabs[node] = job
              self.giveJob(node, job)
           else:
              self.workers.append(node)

       #re-enque lost job on node failure
       #do not re-enque worker
       elif e.type == FAILURE:
           node = e.node
           job = self.tabs.pop(node)
           self.jobs.append(job)
     finally:
       self.lock.release()
```

図 2　マスタ・ワーカ型プログラムで記述した場合のマスタオブジェクトの主な処理を示す擬似コード

Fig. 2　Pseudo-code for when written in a typical Master-Worker programming framework.

4.1　オーバレイ構築

あるノードのオブジェクトから別のノードのオブジェクトへ RMI をするとき，そのノードと通信をする必要があり，これを実現するには接続をノード間で確立する必要がある．しかし，グリッド環境では任意のノード間で接続が張れる機会が稀になりつつある．多くのクラスタはゲートウェイで NAT/firewall を導入しており，外部の計算機からクラスタ内部に直接アクセスすることが難しくなっている．なかにはクラスタ内部の計算機すら外部に通信することが許されない環境もある．したがって，グリッド環境に対応したフレームワークはこうした厳しい環境にも耐えられる必要がある．広域環境での TCP 接続 NAT/firewall 越えに関しては，接続をつなぎたい 2 ノードが別々の NAT/Firewall 環境に置かれても接続を成功させる手法もあるが，成功には 2 ノードの同期が重要で，成功の確率も高くない[25)–27)]．また，大規模な計算で通信する相手が多くなる場合，それぞれへの接続を確立することはスケーラブルではない．

本実装では，各プロセスが最初の分散オブジェクトへの参照を得ると同時に，自動的に TCP 接続のオーバレイを構築する．こうすることで，任意のノード間での通信を実現し，任意のノード間の RMI を実現する．各ノードはランダムに選択された少数のノードと接続を試みる（〜15 接続）．この手法はスケーラブルな接続管理を実現するほか，解析によってこの手法は高い確率で全ノードを含む連結グラフを構築することが示されている[28)]．この際，ユーザによる設定が必要なく自動的にオーバレイを構築し，ノード間の RMI にはこのオーバレイが利用される．しかし，一部のクラスタは内外の双方向通信を禁じている．このような例外的なケースにおいて，SSH portforwarding と TCP 接続を駆使し無理やり接続を張ることを実現している．この場合，ユーザに SSH portforwarding を行うホスト間を指定する必要がある．このケースに限ってこの設定をファイルに記述することが必要である．ただし，これは非常に稀であり，わずか数行の設定で実現される．実際の portforwarding，TCP 接続の確立は自動的に実現される．

構築されたオーバレイでモバイル Ad-hoc ルーティングプロトコル，AODV[29)] から取り込まれたルーティングレイアを実装した．このプロトコルは動的なグラフの変化への対応に適しており，我々が想定する資源が増減するシナリオに適合し，ブロードキャストストームを避けることができる．このプロトコルを用いて任意のノード間で透過的にポイント–ポイントの通信を実現している．

4.2　動的プロセス参加

プロセスが計算に参加するにはオーバレイに接続する必要がある．つまり，ブートストラップ用の接続可能なエンドポイントを取得する必要がある．特に TCP オーバレイの場合，（IP，port）のペアが必要である．我々の実装では *endpoint server* を使った参加を実現している．各ノードはサーバをアクセスし，接続可能なエンドポイントを取得すると同時に自分のエンドポイントを通知する．サーバにはいくつかのオプションを提供している．1 つは HTTP サーバで，もう 1 つは GXP[30)] ⋆1 上に構築されたサーバである．GXP は SSH でのみノードを取得するグリッドシェルである．GXP を使うことで数百にものぼる計算資源に SSH でログインすることができる．いったんログインしたノードたちでは並列にプロセスの起動，コマンドに実行をすることができる．また，GXP では並列実行されたプロセスグループでローカルなファイルディスクリプタに読み書きすることで互いに通信することが可能であり，プロセスグループに動的にプロセスを追加させることも可能である．我々はこのメカニズムをサーバの機能のみで採用し，SSH 以外の手段ではアクセスできない資源もエンドポイントを取得できるようにした．

4.3　RMI エラー検出

特に我々が想定する環境における「通信路」は単一の TCP 接続ではなくオーバレイ上の一連の TCP 接続をなしていることが特徴である．2 プロセス間の RMI は 2 種類のメッセージで実現されている．RMI Request Message と RMI Return Message である．さらにすべての RMI 呼び出しはグローバルにユニークな識別子 *RMIID* が割り振られている．実装では次のようなときに RMI が失敗したと判断する：RMI を処理するプロセスが消滅するか，RMI Request Message がたどった TCP 接続が 1 つでも切れる．

RMI が呼び出されたとき，RMI Request Message がオブジェクトをホストするプロセスに向けてオーバレイ上で転送される．各メッセージは RMIID および今までたどったホップを記憶する．メッセージを転送するとき，各プロセスは転送先のプロセスへの *path pointer* および今までたどったホップを RMIID で保持する．

RMI 処理後，結果は RMI Return Message によって RMI Request Message がたどった道を逆に呼び出しプロセスに向けて転送される．RMI Request Message が来たパスを逆にたどる過程で各プロセスは記憶した *path pointer* を削除する．

あるプロセスが消滅した，もしくは TCP 接続が切断されたとき，接続が切断されたことをつながっていた各プロセスが検知する．検知した各ノードは切断された接続先を指す *path pointer* がある場合，エラーを知らせる例外を返り値とした RMI Return Message を

⋆1 http://www.logos.ic.i.u-tokyo.ac.jp/gxp/

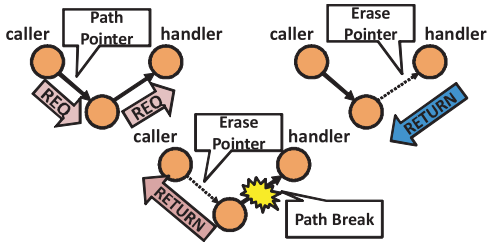


図 3　左・右の図はそれぞれ RMI Request，RMI Return のメッセージがいかにコネクションに沿って伝搬されるかを示している．中央の図は RMI Request メッセージの中継に使われる接続が切断された場合に起こる様子を示している

Fig. 3　The left and right figures show how the RMI Request and the RMI Return messages respectively, are forwarded along the connections. The center figure depicts what would happen if an intermediate connection is lost.

保存された RMI Request が来たパスを逆にたどって呼び出しプロセスに返す．この過程で RMI Request Message が築いた *path pointer* も削除される．全体の過程のイメージを**図 3** に示す．

5. 実　　験

図 1 に示すプログラムをベースに，我々はいくつかのマスタ・ワーカモデルのアプリケーションを実装した．この章ではそれらが資源が増減するグリッド環境で有効に動作することを検証する．まず，設定環境を**表 1**，**図 4** に示す．kyoto，imade クラスタは NAT 環境であるため外部からの接続を受け付けない．kototoi クラスタはグローバル IP を持つが，ゲートウェイスイッチの設定で外部からの接続を受け付けない．また，istbs，tsubame クラスタは外部とのネットワーク設定でほとんどのポートのパケットがフィルタされている．istbs，tsubame クラスタのゲートウェイノードから hongo クラスタの 1 ノードに向けて SSH portforwarding を使った TCP 接続を行う設定にした．このため簡単な設定ファイルを設け，6 行程度でこの設定を記述した．その他のノードは 4.1 節の手法により，自動的に全資源を連結につなぐオーバレイグラフを構築することができた．

5.1 耐 故 障 性

この実験では動的に計算資源の追加，突然の故障に対する我々のフレームワークの性能を評価した．1 つのマスタオブジェクトから 10,000 個のサブタスクを各プロセス 1 つのワーカオブジェクトに分散処理をさせる．プロセスの追加には 4.2 節の手法を用い，プロセスの

表 1　各クラスタの設定

Table 1　Specifications of each cluster.

Name	CPU	Nodes (Cores)	Network
chiba	Pentium M 1.86 GHz Core2 Duo 2.13 GHz	70 (70) 58 (116)	global IP
hongo	Pentium M 1.86 GHz Core2 Duo 2.13 GHz	70 (70) 14 (28)	global IP
kototoi	Xeon 2.33 GHz	22 (88)	global IP (firewall)
imade	Core2 Duo 2.13 GHz	30 (60)	private IP
okubo	Core2 Duo 2.13 GHz	14 (28)	global IP
suzuk	Core2 Duo 2.13 GHz	36 (72)	global IP
kyoto	Core2 Duo 2.13 GHz	35 (70)	private IP
istbs	Xeon 2.40 GHz	158 (316)	global IP (blocked)
tsubame	Dual Core Opteron 2.40 GHz	- (64)	global IP (blocked)

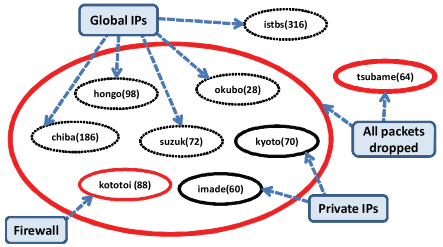


図 4　実験に用いた各クラスタのネットワーク設定

Fig. 4　The network settings for each cluster in our experiment.

削除には単純にプロセスを kill するということを行った．計算にどのタイミングで計算資源の追加，削除を行ったかを下に示す．

0 sec:　hongo，chiba，suzuk，okubo，kyoto，kototoi でブートストラップ（510 workers）．

60 sec:　istbs クラスタ追加（203 workers）．

280-830 sec:　tsubame クラスタ追加（65 workers），しかし通信不具合で失われる．

980 sec:　tsubame 再追加（65 workers）．

1320 sec:　chiba クラスタ全ワーカ kill（−169 workers）．通信に chiba クラスタを経由し

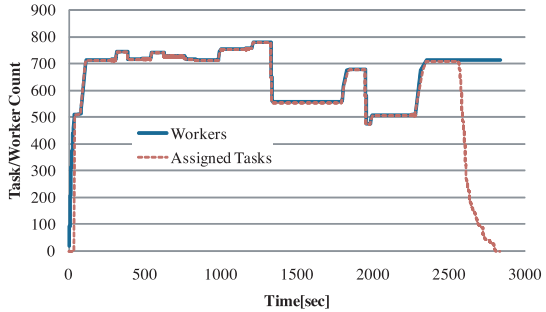


図 5　ノードが追加・削除される間のマスタが認知している参加ワーカ数とマスタに割り当てられたタスク数の時系列
Fig. 5　Time series of active workers and assigned tasks as nodes are added and removed.

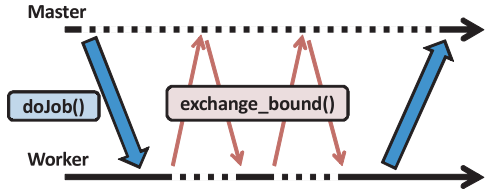


図 6　P-FSP ソルバにおいて，マスタとワーカが互いに呼び出す RMI の流れを示す．マスタは doJob でワーカにタスクを割り振る．ワーカは exchange_bound を定期的に呼び出し，最新の暫定解の交換を行い，doJob の返り値として探索の結果をマスタに返す
Fig. 6　Master-Worker RMI interaction for the Permutation Flowshop Solver. The master assign tasks via the doJob method. Workers invoke exchange_bound periodically to exchange the latest bounds, and return search results as the return value for doJob.

ていたワーカも失われる．

1800 sec:　chiba クラスタ再追加（+169 workers）．

1950 sec:　istbs クラスタの全ワーカ kill（−202 workers）．

2350 sec:　istbs クラスタ再追加（+202 workers）．

この実験の間 1 つのタスクも失われることはなかった．その裏付けとして，マスタが認知しているワーカ数と割り当てられているタスク数の時系列を**図 5** に示す．図からワーカ数と割り当てられているタスク数が連動していることが見受けられる．ワーカが削除されるとき，そのワーカが実行していたタスクも失われるが，ワーカ数と実行中のタスクが連動していることから，どのタスクが失われているか認知していることも分かる．失われたタスクを再投入することで 1 つのタスクも失われなかったことになった．RMI エラーに対する処理はすべて図 1 に示すプログラムレベルで行われている．4.3 節に示した機構により，全プロセスが直接互いに接続していなくてもエラー検出ができている．また物理的に障害を与えてから [ms] オーダの時間でマスタは障害を検知した．

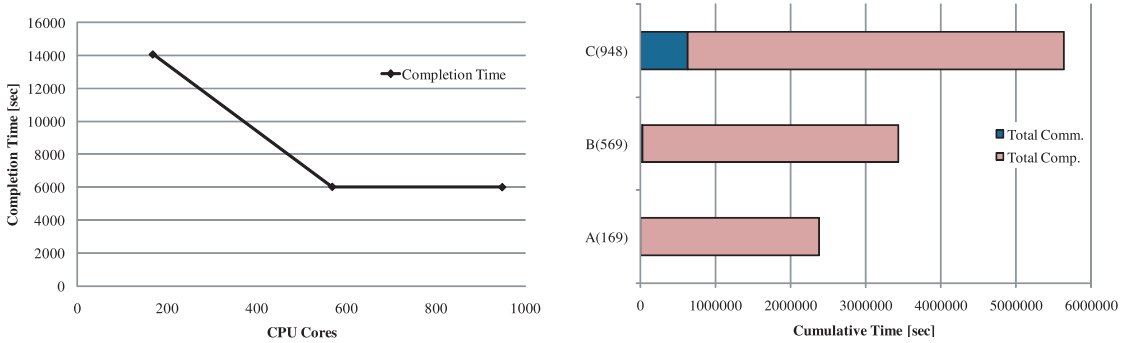
5.2　実アプリケーション

実アプリケーションではタスク間の通信を必要とするものが数多くある．その一例として branch-and-bound を用いるソルバがあり，この応用では並列に走る各ソルバが最新のバウンド情報を共有することは必須である．このような定期的な通信をともなうアプリケーションはバッチスケジューラや，分割統治法フレームワークでは対応することは不可能である．文献 16) にあるようにこのような問題はマスタ・ワーカ型で記述できるが，我々の使う厳しい実験環境（NAT，firewall，IP フィルタリング）で動作するフレームワークは存在しない．

我々はケーススタディとして，Permutation Flowshop Scheduling Problem（P-FSP）に取り組んだ．P-FSP は m マシンでの処理を必要とする n ジョブを考える．総実行時間を最短とするような n 個のジョブの順列の最適解を探索空間を網羅することで発見する．m マシンの順列は考えない．我々の並列ソルバでは各ソルバに分断した探索空間を与え，branch-and-bound 法で探索する．

ワーカが参加したとき，マスタは図 1 に示すように，メソッド呼び出し `doJob` でワーカにタスクを割り振る．branch-and-bound 法では，効率的に探索空間の枝切りが進むために各ワーカができるだけ最新の暫定解を共有することが大切である．したがって，定期的に（60 秒ごと）ワーカはマスタに対するメソッド呼び出し `exchange_bound` で暫定解を交換する．ワーカは自分のタスクが終了すると，マスタに結果を `doJob` の返り値として返す．この様子を**図 6** に示す．この実例にあるように，2 つのオブジェクト間でメソッド呼び出しをすることはグリッド計算でも自然であり，active object のようなモデルではデッドロックによって実現できない例である．

このアプリケーションの狙いはまだ最適解が見つかっていない Taillard 問題集[31] の 1 問を解くことであった．計算は数カ月にも及ぶ可能性があるので対故障性は重要な要件であった．マスタ・ワーカのプログラムは第 3 回 Grid Plugtest 大会[15] のための C++実装を用い，我々のフレームワークはそれらをグリッド環境で走らせるつなぎ合わせとして用いた．Python と C++ソルバのつなぎ合わせには UNIX の `popen` を用いた stdin/stdout の読み書きで行った．しかし，Python から C/C++を呼び出すライブラリも備わっており，より自然な関数呼び出しという形で 2 つをつなぎ合わせることも容易である．Python でのコー



a. Computation time with respect to CPU cores　　b. Cumulative computation/communication time in seconds

図 7　ランダム生成された P-FSP（n: 28，m: 20）に対する並列ソルバの評価
Fig. 7　Evaluation of the Permutation Flowshop Solver for a randomly generated instance (n: 28, m: 20).

ドはわずか 250 行程度である．

3 つの資源の組合せで実行をした．

A:　chiba クラスタのみ：168 cores

B:　chiba，hongo，kototoi，okubo，suzuk，imade，kyoto クラスタ：569 cores

C:　chiba，hongo，kototoi，okubo，suzuk，imade，kyoto，istbs，tsubame クラスタ：948 cores

時間上の制約により，評価にはランダムに生成された比較的小さい問題（$n = 28$，$m = 20$）を解くことにした．結果を**図 7** (a)，(b) に示す．計算はコア数に対して線形にはスピードアップしていないが，これはタスクの重複投入があるからである．この要因として，ソルバの動的負荷分散アルゴリズムがある．問題と branch-and-bound 法の性質上，各タスクの大きさには大きな差がある．したがって，1 回実行してしまったタスクをより小さなサブタスクに展開する必要がある．この目安として，マスタは未割当てのタスクがなくなると，すでに割り当てたタスクをより小さなサブタスクに展開し，配布するということをしている．展開されたタスクを実行していたワーカの仕事は重複に実行され，無駄になってしまい，次に `exchange_bound` を実行するまで気づくことはない．このタスクの重複実行はワーカ数が多くなるほど顕著になる．図 7 (b) を見ると，組合せ C の累計実行時間は組合せ A，B のそれぞれ 2.11 と 1.47 倍である．この重複実行を考慮する場合，組合せ C の A に対するスピードアップは 4.94 である（組合せ C は A の 5.64 倍のコア数である）．したがって，スピードアップが伸び悩むことは主にタスク分散アルゴリズムが要因である．ここでの新規性は，我々が提供する柔軟なプログラミングモデルを用い，多種多様なグリッド環境で最小限の設定でアプリケーションが実行できたことである．

6. ま と め

我々は広域環境の接続性の問題，動的な資源の増減をはらむグリッド環境上のプログラミングフレームワークを提案した．プログラミングの柔軟性を損なわないよう，既存のオブジェクト指向言語に並列分散計算用の拡張として設計した．その一方，RMI のみでは解決できない排他制御，非同期イベント処理，透過的な通信，動的なプロセスの参加・脱退を解決するために，グリッド環境において並列計算を行うプログラミングモデルを提示した．オブジェクトへのアクセスはデッドロックが起きないように陰に逐次化し，オブジェクトへのイベント通知機構とイベント通知のセマンティクスを定義することで非同期なイベントも低レベルのイベントループを使わずに処理できる．また，TCP オーバレイとルーティングを用いることで，NAT，firewall が混在する環境での RMI 通信を透過的に扱い，スケーラブルな接続管理を実現することができた．プロセスの参加にともなって必要なブートストラップ機構を用意し，脱退時の障害は RMI エラーというセマンティクスで対応している．

実験では NAT，firewall，SSH でのアクセスが必須なクラスタを含む 9 クラスタにまたがる 900 コアを用い，branch-and-bound を用いた最適化問題の並列ソルバを実行した．そ

のプログラムは簡潔にかつ柔軟に記述でき，複雑なネットワーク環境であるにもかかわらず動作した．

参 考 文 献

1) OpenPBS. http://www-unix.mcs.anl.gov/openpbs/
2) Ayyub, S., Abramson, D., Enticott, C., Garic, S. and Tan, J.: Executing Large Parameter Sweep Applications on a Multi-VO Testbed, *CCGRID '07: Proc. 7th IEEE International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, pp.73–82, IEEE Computer Society (2007).
3) Singh, G., Deelman, E., Mehta, G., Vahi, K., Su, M.-H., Berriman, G.B., Good, J., Jacob, J.C., Katz, D.S., Lazzarini, A., Blackburn, K. and Koranda, S.: The Pegasus Portal: Web based Grid Computing, *SAC '05: Proc. 2005 ACM symposium on Applied computing*, New York, NY, USA, ACM, pp.680–686 (2005).
4) Thain, D., Tannenbaum, T. and Livny, M.: Distributed computing in practice: the Condor experience, *Concurrency — Practice and Experience*, Vol.17, No.2-4, pp.323–356 (2005).
5) DAGMan. http://www.cs.wisc.edu/condor/dagman/
6) Taverna Project Website. taverna.sourceforge.net/
7) Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI*, pp.137–150 (2004).
8) Egner, M.T., Lorch, M. and Biddle, E.: UIMA Grid: Distributed Large-scale Text Analysis, *CCGRID '07: Proc. 7th IEEE International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, pp.317–326, IEEE Computer Society (2007).
9) Huet, F., Caromel, D. and Bal, H.E.: A High Performance Java Middleware with a Real Application, *Proc. Supercomputing conference*, Pittsburgh, Pennsylvania, USA (2004).
10) Halstead, J.R.H.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Trans. Program. Lang. Syst.*, Vol.7, No.4, pp.501–538 (1985).
11) Taura, K., Matsuoka, S. and Yonezawa, A.: *ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language: Its Design and Implementation* (1994).
12) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada, pp.212–223 (1998). Proceedings published *ACM SIGPLAN Notices*, Vol.33, No.5 (May 1998).
13) Wrzesinska, G., van Nieuwpoort, R.V., Maassen, J., Kielmann, T. and Bal, H.E.: Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments, *International Journal of High Performance Computing Applications* (*IJHPCA*), Vol.20, No.1 (2006).
14) Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, USA (1986).
15) 3rd Grid Plugtest Report. http://www-sop.inria.fr/oasis/plugtest2006/plugtests_report_2006.pdf
16) Aida, K. and Osumi, T.: A Case Study in Running a Parallel Branch and Bound Application on the Grid, *SAINT '05: Proc. 2005 Symposium on Applications and the Internet* (*SAINT'05*), Seattle, WA, USA, pp.164–173, IEEE Computer Society (2005).
17) Aoki, H., Nakada, H., Tanaka, K. and Matsuoka, S.: A Programming Environment with Dynamic Node Configuration for Hierarchical Grid: Jojo2, *IPSJ-SACSIS* (2006).
18) Linderoth, J., Goux, J.-P. and Yoder, M.: Metacomputing and the Master-Worker Paradigm, Technical Report ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory (2000).
19) Nakada, H. and Matsuoka, S.: A Java-based programming environment for hierarchical Grid: Jojo, *CCGRID*, pp.51–58 (2004).
20) Shudo, K., Tanaka, Y. and Sekiguchi, S.: P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources, *CCGRID '05: Proc. 5th IEEE International Symposium on Cluster Computing and the Grid* (*CCGrid'05*), Vol.1, Washington, DC, USA, pp.259–266, IEEE Computer Society (2005).
21) van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T. and Bal, H.E.: Ibis: A Flexible and Efficient Java-based Grid Programming Environment, *Concurrency and Computation: Practice and Experience*, Vol.17, No.7-8, pp.1079–1107 (2005).
22) Fuhner, Tim, Popp, Stephan, Jung and Thomas: A novel framework for distributing computations DisPyTE distributing Python tasks environment, *Journal of Computational Electronics*, Vol.5, No.4, pp.349–352 (2006).
23) van Nieuwpoort, R.V., Kielmann, T. and Bal, H.E.: Efficient Load Balancing for Wide-area Divide-and-conquer Applications, *PPoPP '01: Proc. 8th ACM SIGPLAN symposium on Principles and practices of parallel programming*, New York, NY, USA, pp.34–43, ACM (2001).
24) Denis, A., Pérez, C. and Priol, T.: PadicoTM: An open integration framework for communication middleware and runtimes, *Future Gener. Comput. Syst.*, Vol.19, No.4, pp.575–585 (2003).

25) Ford, B., Srisuresh, P. and Kegel, D.: Peer-to-peer communication across network address translators, *ATEC'05: Proc. USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, Berkeley, CA, USA, p.13, USENIX Association (2005).
26) Guha, S., Takeda, Y. and Francis, P.: NUTSS: A SIP-based approach to UDP and TCP network connectivity, *FDNA '04: Proc. ACM SIGCOMM workshop on Future directions in network architecture*, New York, NY, USA, pp.43–48, ACM (2004).
27) Maassen, J. and Bal, H.E.: Smartsockets: Solving the connectivity problems in grid computing, *HPDC '07: Proc. 16th international symposium on High performance distributed computing*, New York, NY, USA, pp.1–10, ACM (2007).
28) Horita, Y., Taura, K. and Chikayama, T.: A Scalable and Efficient Self-Organizing Failure Detector for Grid Applications, *GRID '05: Proc. 6th IEEE/ACM International Workshop on Grid Computing*, Washington, DC, USA, pp.202–210, IEEE Computer Society (2005).
29) Perkins, C.: Ad-hoc On-demand Distance Vector Routing, *MILCOM '97 panel on Ad Hoc Networks, Nov. 1997* (1997).
30) Taura, K.: GXP: An Interactive Shell for the Grid Environment, *IWIA*, Vol.00, pp.59–67 (2004).
31) Taillard Homepage. http://mistic.heig-vd.ch/taillard/

(平成 20 年 1 月 29 日受付)
(平成 20 年 4 月 25 日採録)

弘中　健（学生会員）
1983 年生．2007 年東京大学工学部電子情報工学科卒業．同年より同大学大学院情報理工学系研究科電子情報学専攻修士課程在学中．ACM 会員．

斎藤　秀雄（学生会員）
1981 年生．2006 年東京大学大学院情報理工学系研究科電子情報学専攻修士課程修了．同年より同博士課程在学中．ACM 会員．

高橋　慧
1983 年生．2008 年東京大学大学院情報理工学系研究科電子情報学専攻修士課程修了．現在，ソニー（株）．

田浦健次朗（正会員）
1969 年生．1997 年東京大学大学院理学博士（情報科学専攻）．1996 年より東京大学大学院理学系研究科情報科学専攻助手．2001 年より東京大学大学院情報理工学系研究科電子情報学専攻講師．2002 年より同助教授．2007 年より同准教授．日本ソフトウェア科学会，ACM，IEEE-CS 各会員．