高性能GridRPCアプリケーションの開発環境

小林孝嗣[†] 渡邊啓正[†] 本多弘樹[†]

GridRPCを用いることでプログラマはグリッドアプリケーションの作成を容易に行うことができる．しかし，グリッドでは計算資源やネットワーク資源の性能は不均一で変動するため，プログラマにとってアプリケーション実行中に起きた問題の原因究明や，資源を効果的に活用するアプリケーションの作成を行うことは容易ではない．本研究では高性能GridRPCアプリケーションの開発環境の一環としてGridRPCアプリケーションの開発支援ツールの設計および実装を行った．本ツールはアプリケーションのデバッグや性能改善の際のプログラマの手間を軽減すべく，計算資源の負荷情報やRPC実行情報などを視覚的にプログラマに提示する．

Development Environment for a High Performance GridRPC Application

Takatsugu Kobayashi,[†] Hiromasa Watanabe[†] and Hiroki Honda[†]

GridRPC system allows programmers to develop a grid application easily. However, performance of resources of a grid are heterogeneous and changes dynamically. This makes it difficult to specify problems occurred in execution of a GridRPC application, or to develop a GridRPC application that uses resources effectively. We developed development assistant tool for GridRPC application, as a part of development environment for high performance GridRPC application. This tool visually shows programmers information about a load of resources and execution of GridRPC, in order to mitigate programmer's burden for debugging or performance improvement of GridRPC application.

1. はじめに

GridRPC[1]はグリッドにおける有力なプログラミングモデルの1つであり，GridRPCを用いることでグリッド上で動作する様々なアプリケーションを作成できることが報告されている[2),3)]．

グリッドでは計算資源やネットワーク資源の性能は不均一で変動する．そのため高性能なGridRPCアプリケーションを作成する際に，正常な動作の妨げおよび性能低下につながる障害が起きる（**表1**）．表中の1〜4の障害を特定するためにはGridRPCのAPIによるエラーコードやRPC実行の様子などのRPC実行情報を，また表中の5〜7の障害を特定するためには計算資源の性能情報および負荷情報といった計算資源情報を調べなければならない．しかし，これらの情報収集作業は以下の理由によりプログラマにとって非常に大きな手間となる．

- 資源の状態が実行のたびに変化する．
- 大規模な環境では情報量が増大する．
- 情報収集のためにアプリケーションのソースコードを変更しなければならない．

したがってプログラマはGridRPCアプリケーションの動作や性能に影響する障害の特定を行う際に大きな労力を割かなければならない．

本研究では高性能GridRPCアプリケーションの開発環境の一環としてGridRPCアプリケーションの開発支援ツールの設計および実装を行った．本ツールは上記の問題を解決し，プログラマの手間を軽減するために以下の機能を提供する．

- RPC実行情報の収集機能
- 計算資源情報の収集機能
- 収集した情報の可視化機能

なお本研究ではGridRPCシステムの1つであるNinf-G[4)]を用いたGridRPCアプリケーションの開発を支援の対象としている．

本稿の構成は以下のとおりである．2章で典型的なGridRPCアプリケーション開発のシナリオおよびそ

† 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

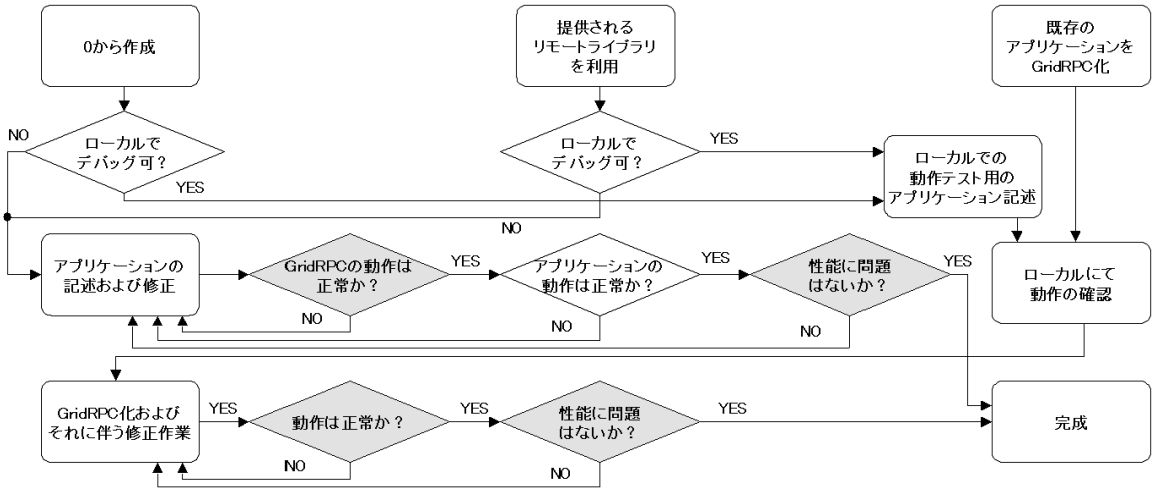


図 1 GridRPC アプリケーションの開発の流れ
Fig. 1 Flow of development of a GridRPC application.

表 1 GridRPC アプリケーション実行中に起きうる障害の分類
Table 1 Classification of fault in execution of a GridRPC application.

1. 指定したホストが動作または存在していない.
2. サーバにリモートライブラリが存在しない.
3. ネットワーク性能が不十分である.
4. ネットワーク障害によりサーバとの通信が途絶える.
5. サーバの負荷が大きい.
6. RPC 実行中の計算資源に障害が起きる.
7. 計算資源の性能が不十分である.

こにおける方法論について述べる．3 章で提案するツールの設計について述べ，4 章でツールの実装について述べる．5 章で本ツールの利用手順と動作概要について述べ，6 章でツールの評価について述べる．7 章で関連研究について述べ，8 章で結論と今後の課題について述べる．

2. GridRPC アプリケーション開発のシナリオ

GridRPC アプリケーション開発のシナリオは大きく分けて以下の 3 つに分類できる（**図 1**）．

(1) 新規に GridRPC アプリケーションを作成する．
(2) 既存のアプリケーションを GridRPC 化する．
(3) 提供されているリモートライブラリを利用する GridRPC アプリケーションを作成する．

(1) の場合は，アプリケーションのアルゴリズムや GridRPC の利用方法などをすべてプログラマが新規に設計しなければならない．そのためアプリケーションが正常に動作しなかった場合，プログラマはバグがクライアントプログラムとリモートライブラリのどちらに存在しているのか，GridRPC に関連するバグなのかそれ以外のアルゴリズムなどによるバグなのかといったいくつもの要因を調べなければならない．

(2) の場合は，既存のアプリケーションは正常に動作するものであるはずなので，バグは GridRPC 化の際に発生すると考えられる．そのためプログラマは GridRPC 周りを中心にデバッグを行えばよい．

(3) の場合は，通常提供されるリモートライブラリにバグはないと仮定できる．そのためプログラマはバグが存在するのはクライアントプログラム内のみであると限定してデバッグを行うことができる．

以下では GridRPC アプリケーションの開発において，どのように GridRPC アプリケーションのデバッグおよび性能改善を行えばよいかを考察する．

2.1 GridRPC アプリケーションのデバッグ

GridRPC アプリケーションの開発においては，GridRPC に関連するバグとそうでないバグが混在している場合のデバッグが，プログラマにとって大きな手間となる．

発生しているバグが GridRPC に関連するものであれば，プログラムの記述ミス以外にも利用した計算機が動作していなかったなど，アプリケーションが動作する環境に問題が発生していたということも考えられる．そのほかにも，利用する計算機の負荷が高くて正常に動作しなかったなどといった再現性のない問題が起きることも考えられる．このような GridRPC に関連するバグとそうでないバグが重なった場合は，アプリケーションの不具合の原因がどこにあるのかを調べるのが困難であり，どこからアプリケーションの修正

を行えばよいかという判断が難しい．

そこで本研究では上記の問題に対する 1 つの解決方法として，以下の方法による GridRPC 関連のバグとそうでないバグの分離を行うことを前提とする．

(a) ローカルでの動作テスト可能な場合
GridRPC の部分を通常の関数呼び出しに置き換えて動作テストが可能な場合，既存のデバッガやデバッグ支援ツールを利用することで容易にローカルで動作するアプリケーションを作成することができる．ローカルでテストを行った後に関数呼び出しの部分を RPC 呼び出しに変更することで，アプリケーションに発生するバグは GridRPC に関連するものであると限定することができる．

(b) ローカルでの動作テストが不可能な場合
特定の場所でしか利用できないリモートライブラリが必要な場合や計算量の関係で複数台の計算機の利用が必要な場合など，(a) のようにローカルでの動作テストが不可能な場合もある．このような場合には，まず GridRPC が正常に動作していることを確認し，そのうえでアプリケーション全体のデバッグを行う．

2.2 GridRPC アプリケーションの性能改善

GridRPC アプリケーションの性能改善の例として以下のものがあげられる．

(1) RPC 部分の並列化によるチューニング
(2) 逐次部分のアルゴリズムのチューニング
(3) スケジューリングのチューニング
(4) 耐故障性の実現

この中で (1)，(3)，(4) は GridRPC の動作する回数やタイミングなどに影響を及ぼす．そのため，意図したとおりのチューニングができているかの判断には，性能改善後のアプリケーションにおける GridRPC の動作内容を確認しなければならない．これには RPC 実行のタイミングや所要時間などが含まれる．

2.3 GridRPC アプリケーション開発の手間の軽減

2.1 節で触れた GridRPC が正常に動作していることの確認および 2.2 節で触れたアプリケーション内の GridRPC 実行の様子の確認は，1 章で述べたとおり手間のかかる作業となる．よってその作業の支援を行うことで，プログラマのアプリケーション開発の手間を大きく軽減することが可能となる．

3. ツールの設計

本章では GridRPC アプリケーションの開発を支援するために本ツールが提供する機能（**図 2** 網掛け部）の設計について述べる．

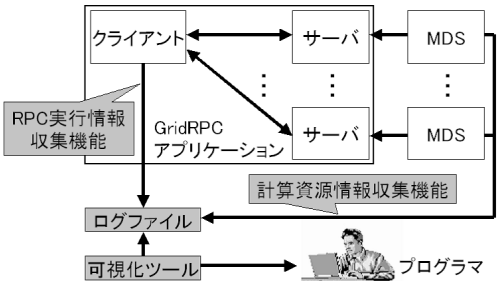


図 2 本ツールの機能の構成
Fig. 2 Outline of the behavior of our tool.

3.1 GridRPC アプリケーション開発における本ツールの位置づけ

本ツールでは GridRPC の動作に関連する情報の調査作業において，プログラマの手間を軽減するための機能を提供する．これらの機能は GridRPC アプリケーション開発において，GridRPC に関連した情報が必要な場面（図 1 網掛け部）において利用される．

3.2 RPC 実行情報収集機能

GridRPC に関する処理の開始時刻や所要時間などの情報を収集する．

これらの情報を収集する処理はプログラマがアプリケーションプログラムの内容を大きく変更することなく行えるようにしなければならない．

3.3 計算資源情報収集機能

GridRPC アプリケーションが利用した計算資源の性能情報および負荷情報を収集する．

これらの情報も，RPC 実行情報の収集と同様に，プログラマに余計な手間をかけさせることなく，自動で収集ができるようにしなければならない．

3.4 アプリケーション実行情報可視化機能

収集した RPC 実行情報および計算資源情報から，GridRPC アプリケーションのデバッグや性能改善に有用な情報を可視化し，プログラマが効率良く把握可能とする．

以下では可視化ツールで提供する機能について述べる．

アプリケーション実行環境の情報提示機能
実行環境の情報を把握しやすい形でプログラマに提示する．グリッドは性能の不均一な計算資源から構成されていることが多いため，アプリケーションがどの資源を利用したのかという情報が GridRPC アプリケーションのデバッグおよび性能改善に必要である．

RPC 実行状況の可視化機能

RPC の実行や同期などの処理がどのように行われていたのか，どの程度の処理時間がかかっていたのかなどの情報を把握しやすい形でプログラマに提示する．GridRPC アプリケーションにおいては RPC 実行がアプリケーションの大部分を占めているため，RPC 実行の状況を容易に把握可能とする機能が必要である．

計算資源情報の提示機能

計算資源の負荷情報や性能情報といった計算資源情報を把握しやすい形でプログラマに提示する．計算資源の性能や負荷状態が RPC に関連する処理の動作に大きく影響するため，計算資源の情報が GridRPC アプリケーションのデバッグおよび性能改善に必要である．

詳細情報の提示機能

RPC のハンドル初期化や同期処理などの処理それぞれに関する詳細な情報を表示する．処理の詳細な情報が障害の特定および性能改善に必要となる．

フィルタ表示機能

すべての処理の中から特定の計算資源上での処理だけを表示するなど，指定した条件を満たす情報のみを表示する．大規模な環境においては GridRPC アプリケーション実行に関する情報の量が増加するため，プログラマが必要とする情報のみに絞って表示する機能が必要である．

4. ツールの実装

4.1 RPC 実行情報収集機能

GridRPC の API の実行開始時刻および終了時刻を gettimeofday() 関数を用いて取得し，GridRPC の API からエラーコードやセッション情報を取得する．取得した情報はログファイルに出力する．そしてプリプロセッサを用いて GridRPC の API を前述の処理を付加した API に置換する．

プログラマはクライアントプログラムにヘッダファイルのインクルード文を追加するだけで RPC 実行情報の収集を行うことができる．ヘッダファイル内においては，GridRPC の API を情報収集機能を追加した API に置換するためのコードが記述されている．

また，以下の本ツールが提供する時刻計測用 API をクライアントプログラムの開始部分および終了部分に挿入することで，アプリケーション全体の実行時間を計測する．

- begin_monitor()
  アプリケーションの開始時に挿入する必要がある API．情報収集の初期化処理を行う．引数として Ninf-G 用の設定ファイルへのパス，資源情報収集の有無を示すフラグなどをとる．
- end_monitor()
  アプリケーションの終了時に挿入する必要がある API．情報収集の終了処理を行う．

4.2 計算資源情報収集機能

計算資源の性能情報および負荷情報を Globus Toolkit [5] の資源情報管理システムである MDS を利用して収集し，ログファイルに出力する以下の API をプログラマに提供する．

- check_resource()
  計算資源の負荷情報を収集する API．プログラム中の任意の時点で実行可能．

プログラマはこの API をクライアントプログラムに挿入することで任意の時点での計算資源情報を収集できる．

MDS で収集可能な情報は，CPU やメモリの性能情報および負荷情報などの単純な情報のみである．また，MDS では資源情報を収集する周期は標準で 30 秒であり，設定によって 1 秒まで短くできる．よって実行時間が 1 秒以下の RPC に対して適用ができない．しかし，グリッド上で実行される RPC の実行時間は比較的長いことが予想されるため，GridRPC アプリケーションのデバッグおよび性能改善に必要な情報は，MDS から得られる情報で十分であるといえる．

4.3 アプリケーション実行情報可視化機能

ログファイルに出力された RPC 実行情報および計算資源情報を Java を用いて実装した GUI ツールによって可視化してプログラマに提示する．

4.1 節および 4.2 節で述べたとおり，本ツールではログファイルとして情報を出力する方式を採用した．これは再現性のない問題の原因究明および性能改善前後のアプリケーション実行情報の比較には，実行のたびに情報を保存する必要があるためである．そのため，リアルタイムでアプリケーションの動作を追うことはできないが，アプリケーション実行途中においても，その時点のログファイルを読み込むことで途中経過を可視化することは可能である．

以下では可視化ツールを構成する主要なコンポーネントについて述べる．

アプリケーション全体の情報表示部（図 3 の①）

アプリケーションの実行時間，RPC 実行回数，利用したサーバの数などのアプリケーション全体の情報を表示する．

実行環境の略図表示部（図 3 の②）

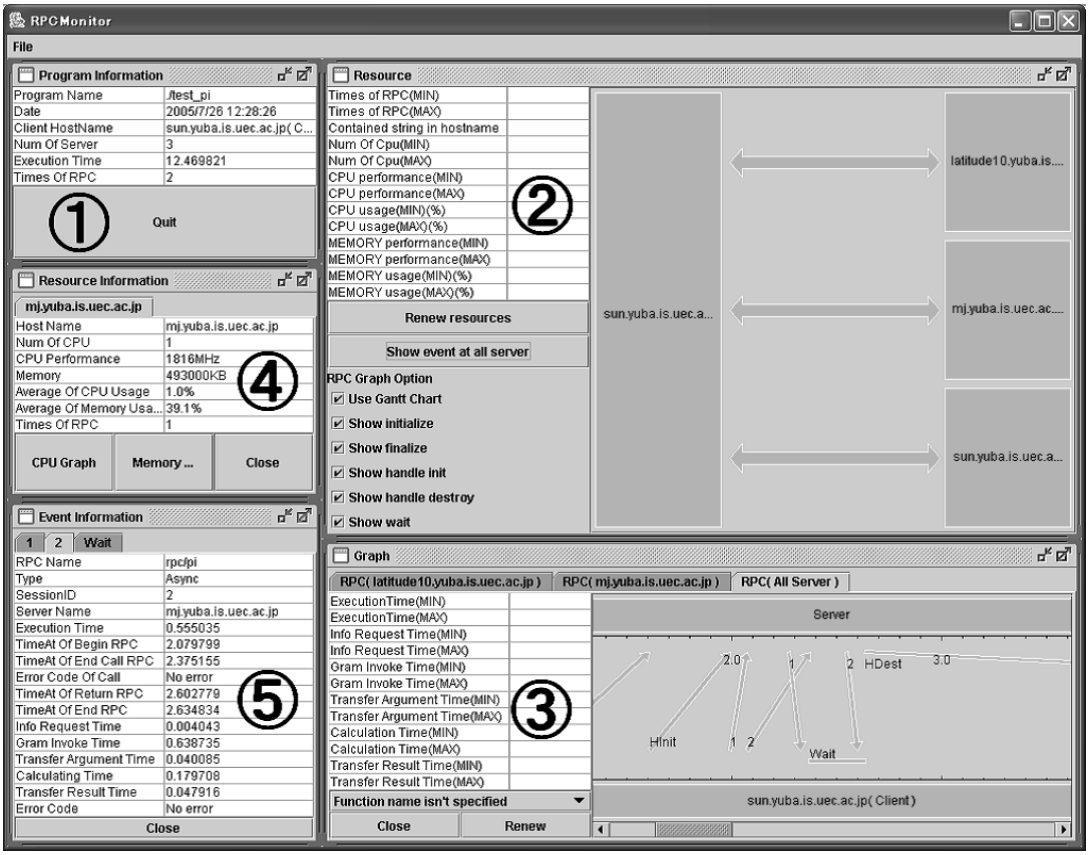


図 3 可視化ツールの動作画面
Fig. 3 Screenshot of our visualization tool.

アプリケーションにおいて RPC の実行に利用された計算機の情報を視覚的に表示する．本コンポーネントは性能低下やエラーの原因になっている計算資源の目処をつけやすくするために，負荷の高かった計算資源，エラーの起きた計算資源は異なった色で表示する．またこの部分において「平均 CPU 負荷が 80%以上の計算資源」などの条件を指定することにより多数の計算資源を利用していた場合でもプログラマが必要な情報のみを表示させることができる．

RPC 実行状況のグラフ表示部（図 3 の③）
アプリケーションにおける RPC 実行の様子をグラフ形式で視覚的に表示する．プログラマは本コンポーネントのグラフを見ることで，計算や通信に長時間を要した RPC などの特定を容易に行うことができる．この部分においても実行環境の略図表示部と同様に RPC の計算時間などの条件を指定して一致するものだけを表示することができる．このグラフでは横軸が時間，下から上への矢印がクライアントからサーバへ対する処理，上から下への矢印がサーバからクライアントへの処理を表している．

計算資源の負荷状態のグラフ表示部（図 3 の③）
各計算資源ごとの CPU およびメモリの負荷の変動を折れ線グラフで表示する．本コンポーネントは計算資源の負荷変動のグラフを RPC 実行状況のグラフと同じ部分に表示する．これによりプログラマは容易に双方を比較することができる．

計算資源の詳細情報表示部（図 3 の④）
各計算資源ごとの性能情報などの詳細な情報を表示する．本コンポーネントは各計算資源ごとに性能情報，負荷情報，RPC 実行回数などの情報を一括してプログラマに提示する．

各処理の詳細情報表示部（図 3 の⑤）
GridRPC に関連する処理の詳細な情報を表示する．本コンポーネントは各 GridRPC の API の処理開始時刻，所要時間，エラーコードなどを一括してプログラマに提示する．

5. ツールの利用手順と動作概要

本ツールおよび Ninf-G を用いてアプリケーション開発を行う場合，プログラマは通常の Ninf-G アプリケーションの開発に加えて以下の作業が必要となる．

(1) クライアントプログラムに本ツールの提供するヘッダファイルのインクルードなどのコードを数行追加する．
(2) 生成されたログファイルを GUI ツールで可視化する．

可視化ツールの大まかな利用手順は以下のとおりである．

(1) ログファイルを開くと GridRPC アプリケーションの実行環境の状態を表す図が表示される（図 3 の②）．
(2) 実行環境の状態を表す図から調べたい計算資源を選択するとその計算資源における負荷情報および RPC 実行状況のグラフが表示される（図 3 の③，④）．
(3) RPC 実行状況のグラフの各処理を表す図形をクリックするとその処理に関する詳細な情報が表示される（図 3 の⑤）．

一方，GridRPC アプリケーション実行における本ツールの動作概要は以下のとおりである（図 2）．

- アプリケーション開始時，終了時に MDS に問い合わせて計算資源情報を取得し，ログファイルに出力する．
- RPC の実行，同期などの処理が起きるとその内容をログファイルに出力する．
- プログラマが本ツールの提供する計算資源情報収集の API をクライアントプログラムに挿入していた場合はその部分で計算資源情報を取得し，ログファイルに出力する．

6. ツールの評価

6.1 評価環境

評価環境は**表 2** のとおりである．ホスト 0〜12 はそれぞれクラスタを構成する計算ノードであり，本評価ではそれぞれのノードを 1 つの計算機と見立てて利用している．すべてのホストで OS として Redhat Linux9 を，グリッドミドルウェアとして Globus Toolkit 2.4.3 および Ninf-G 2.3 を用いた．実行時オーバヘッドの評価ではホスト A をクライアント，ホスト B をサーバとして利用し，ツールの動作検証においてはホスト 0 をクライアント，ホスト 1〜12 の 12 台をサーバとして利用した．

表 2 評価環境
Table 2 Evaluation environment.

ホスト名	CPU	ネットワーク
ホスト A	Pentium4 M 1.8(GHz)	100 BASE-T
ホスト B	Pentium4 1.8(Ghz)	100 BASE-T
ホスト 0〜12	Pentium4 1.4(GHz) × 2	1000 BASE-SX

表 3 GridRPC 実行情報収集によるオーバヘッド
Table 3 Overhead to collect information about execution of GridRPC.

RPC 実行回数（回）	情報収集なし（秒）	情報収集あり（秒）	オーバヘッドの割合（%）
1	0.41	0.42	2.4
4	2.43	2.49	2.0
16	8.46	8.53	0.8
64	34.61	34.81	0.6

表 4 計算資源情報収集によるオーバヘッド
Table 4 Overhead to collect information about resources.

収集回数（回）	オーバヘッド（ミリ秒）
1	4.0
4	10.6
16	60.8
64	245.8

6.2 実行時オーバヘッドの評価

本評価ではツールを利用して情報を収集することによりどの程度の実行時オーバヘッドが生じているのかを計測した．RPC 実行情報を収集する際のオーバヘッドは，RPC の実行回数を変えながら，情報を収集した場合としなかった場合での実行時間を比較することによって求めた．計測にはモンテカルロ法により円周率を求めるプログラムを利用した．

計算資源情報を収集する際のオーバヘッドは，単純に指定した回数計算資源情報を収集するプログラムを用いて計測した．今回の評価では 1 回の収集につき計算資源 1 つの情報を取得している．

RPC 実行回数に対してのアプリケーション実行時間，アプリケーション実行時間に対する RPC 実行情報収集のオーバヘッドの割合は**表 3** のとおりである．計算資源情報の収集回数に対するオーバヘッドの変化は**表 4** のとおりである．

表 3 から実行時オーバヘッドは最大でプログラム実行時間の 2.4%程度であり，アプリケーションの性能にはほとんど影響しないことが分かる．表 4 から計算資源情報収集によるオーバヘッドは 1 回につき 4 ミリ秒程度で，通常のアプリケーションの実行時間から考えれば無視してもかまわない程度であると分かる．

実際のグリッド環境においては，利用する計算資源の規模の大きさ，ネットワーク性能の不均一さなどから，オーバヘッドの値は本評価の結果より大きくなると考えられる．その結果，本ツールを適用した場合のアプリケーションの実行時間が大きくなってしまい，適切なデバッグおよび性能改善が行えなくなる．

これに対しては MDS の設定の最適化などを行い，情報収集のオーバヘッドを小さくすることによる解決方法が考えられる．実際に短時間処理向けに最適化を行った結果，64 台の資源情報を 500 ミリ秒で収集できたという報告があり[6]，ツール利用による情報収集のオーバヘッドを小さくするために，このような技術を利用することが考えられる．

6.3 ログファイルの容量の測定

ログファイルの容量はアプリケーション中の RPC に関するイベントの増加に比例して大きくなる．RPC システムの初期化や終了および RPC の同期などの回数は，アプリケーションによらずほぼ固定であると考えられる．よって，本評価では RPC の実行回数を増やしながらログファイルの容量を計測した（**表 5**）．また計算資源の情報を収集する際にもログファイルの容量は増加するので，計算資源の情報収集回数および情報収集の対象となる計算資源数を増やすことによる，ログファイル容量の変化の計測も行った（**表 6**）．計測に用いたのはモンテカルロ法により円周率を求めるプログラムである．

計測の結果より，RPC 実行 1 回につき 500 Byte 程度，1 つの計算資源の情報の収集 1 回につき 300 Byte 程度の容量が必要であると分かった．RPC 実行回数，計算資源情報収集の回数，情報収集の対象となる計算資源の数に対して，ログファイル容量はリニアに増加している．

6.4 デバッグにおけるツールの利用

6.4.1 RPC 関連の記述ミスの特定

実際にツールを利用することで，プログラムの記述ミスにより正常に RPC が動作していない場合において，その原因を容易に特定できるかを検証した．

本評価では，5 回の RPC を順次実行するプログラムを想定した．このプログラムにおいては，RPC を 5 回実行するが，同期部分において 4 回目の RPC が終了した時点でプログラムが次へと進み，5 回目の RPC の同期を行わずに終了するというバグが含まれている．

ここで，本ツールによってこのバグを容易に発見することができるかを検証する．本評価における原因究明の流れは以下のとおりである．

(1) プログラムに対して本ツールの適用を行う．

表 5 RPC 実行回数によるログファイル容量の変化
Table 5 Size of a log file according to the number of times of RPC.

回数（回）	容量（Byte）
0	1,325
1	1,840
4	3,022
16	7,852
64	27,169

表 6 計算資源情報収集回数および計算資源数によるログファイル容量の変化
Table 6 Size of log file according to the number of times of resource information collection and the number of resources.

回数（回）	1 台（Byte）	4 台（Byte）	8 台（Byte）
0	1,325	1,325	1,325
1	1,604	2,315	3,687
4	2,445	6,015	10,815
16	5,805	20,027	39,014
64	21,165	76,814	148,925

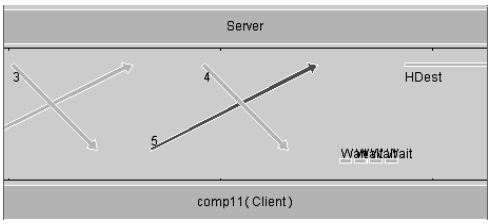


図 4 RPC 実行異常の色分け
Fig. 4 Classification of failed RPC by color.

(2) 生成されたログファイルを可視化ツールで開く．
(3) 可視化結果（**図 4**）より正常に動作していない RPC が見つかる．
(4) 異常のあった RPC を示す矢印をクリックすることで，その RPC の詳細な情報が表示され，同期が行われていなかったことが分かる．
(5) プログラムのバグは RPC の同期が正常に行われていなかったためであると判明する．

以上により，プログラムの記述ミスにより RPC が正常に動作していなかったという問題の解決を，本ツールを利用することにより容易に行えていることが確認できた．

このようなバグを従来の手法で修正する場合，RPC に関連する処理のどの部分にバグがあるか分からないため，すべての RPC に関連する処理において，GridRPC の API および printf() を用いて実行内容を出力するようにプログラムを変更する必要がある．しかし，本ツールを利用した場合には，わずかなプログラムの変更（5 章）で情報の収集を行うことができ，

可視化によりどこにバグが発生しているのかも容易に特定することができる．

6.4.2 実行環境の障害の特定

実際にツールを利用することで，問題の原因となっている障害を特定する手間を軽減できているかを検証した．評価に用いたアプリケーションは，モンテカルロ法を用いて円周率を求めるものである．今回の例では，利用するサーバに対してそれぞれ RPC を 1 回ずつ実行した．

本評価では上記のプログラムを 1 つの計算資源の負荷を意図的に高くした状態で実行する．この場合，通常なら 10 秒程度で終わるはずのプログラムが終了までに 60 秒近くかかる．これはアプリケーションの内容からして明らかに実行時間が長すぎるといえる．

ここで 1 つの計算資源の負荷が高かったことが異常の原因であることを突き止めることができるかを検証する．

本評価における原因究明の流れは以下のとおりである．なおアプリケーションの動作検証に先立って本ツールが適用済みであると仮定する．

(1) 正常に動作しなかった際のログファイルを可視化ツールで開く．
(2) アプリケーションの実行環境の略図のウィンドウからある計算資源の負荷が高かったことが分かる（**図 5**）．
(3) RPC 実行状況のグラフ（図 3 の③）を用いて，負荷の高かった計算資源上の RPC 実行状況を他の計算資源上のものと比較する．
(4) 負荷の高かった計算資源上の関数ハンドル生成や RPC 実行などが他の計算資源上のものより大幅に時間がかかっていると分かる（**図 6** 下部）．
(5) 正常に動作した場合の RPC 実行状況と比較するとどの程度の遅延が起きていたのかが分かる（図 6）．
(6) 異常の原因は 1 つの計算資源の負荷が高く，そこで実行された処理が全体の実行時間に影響を与えたためであると分かる．

以上により，利用した計算資源の中に高負荷なものが含まれていたという，グリッド特有の再現性のない問題についても原因究明が行えていることが確認できた．

今回の場合，従来の手法で原因を究明するには，プログラマが自分で MDS などのモニタリングシステムを検索するなどして各計算資源の負荷状況を調べる必要があった．しかし利用した計算資源が多い場合には，その分だけ計算資源情報の検索を行わなければならず手間が大きい．また今回のような問題は，再現性がないために後から調べて原因を究明することが難しい．モニタリングシステムによっては一定時間前までの資源情報しか保持していないものもあり，その場合には GridRPC アプリケーションの実行直後にプログラマが自分で資源情報を別途保存しておかなければならない．これらのことからアプリケーションの動作検証を行う際にあらかじめ本ツールを適用しておくことが有効であると分かる．

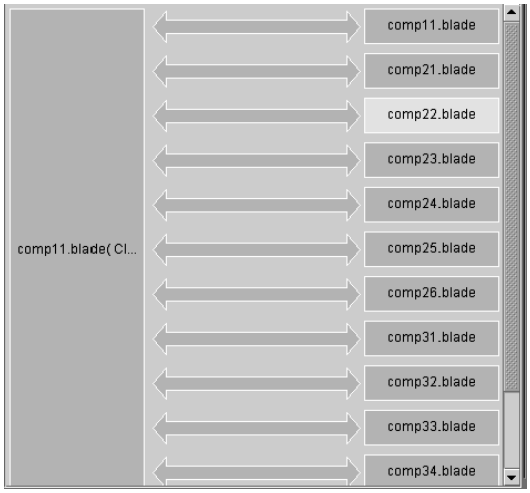


図 5 高負荷な資源の色分け
Fig. 5 Classification of a high-load resource by color.

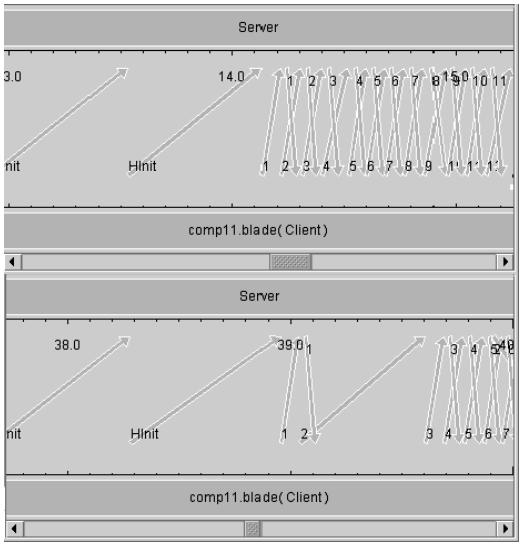


図 6 GridRPC 実行フローの比較
Fig. 6 Comparison of the flows of execution of GridRPCs.

6.5 Grid アプリケーションの性能改善におけるツールの利用

前節で取り上げた問題点に対しては RPC にタイムアウト時間を設定するなどの対応が考えられる．適切なタイムアウト時間を設定するには，プログラマは RPC が正常に動作した場合，計算資源の負荷が高くて処理時間が長かった場合の RPC の動作内容を比較し，タイムアウト時間の設定によりどのように RPC の振舞いが変化するかを検討しなければならない．そのような作業を行う際には本ツールの RPC 実行状況のグラフ表示機能などが役に立つと考えられる．手作業で RPC の実行状況を調べようとした場合には GridRPC の API 呼び出しごとに printf() 文などをソースコードに追加しなければならないが，本ツールを利用すれば 3 行程度のコードを追加するだけでよい．

7. 関 連 研 究

MPI や OpenMP などの並列プログラムに対するデバッグ支援ツールや可視化ツールに関する研究は多数行われている[7)~9)]．しかし，これら既存のツールでは以下の点を考慮していない．

- 計算資源およびネットワーク資源の性能が不均一であること．
- 計算資源およびネットワーク資源の負荷状況が外乱により変動すること．
- 利用する資源の数が増えて出力される情報の量が膨大になること．
- 実行環境を構成する計算資源の数が動的に変化すること．

したがって，既存のツールの方式をグリッドに適用した場合，実行環境の状態によるアプリケーション性能の低下や大規模環境の利用によるアプリケーション実行情報量の肥大化などのグリッド特有の問題点を解決することができない．

グリッド上の計算資源情報をモニタリングするためのツールとして，Ganglia[10)] や SCMSWeb[11)] など様々なツールが存在する．しかし，GridRPC アプリケーションのデバッグや性能改善において，それらを利用する場合，プログラマにはアプリケーション実行中の負荷情報などを調べるために直接モニタリングツールを操作しなくてはならない．一方，本ツールにおいては，利用されているモニタリングツールから，デバッグに必要な計算資源情報を自動で収集し，GridRPC に関する処理と関連付け，利用しやすい形で提供している．

一方，グリッドにおいてアプリケーションの開発や動作検証に利用可能なツールの研究も行われている．GXP[12)] では分散環境における各計算資源上でのコマンド実行を一括して行うことができる．たとえば GXP を用いて ps コマンドを実行すれば各計算資源の CPU 負荷を調べることができる．しかし，利用する計算資源の数が多い場合にはテキストでたくさんの情報が出力されるため，必要な情報をその中から選び出すのは大変である．また，デバッグにおいては計算資源情報単体ではなく，負荷の高かった時点でどのような処理が行われていたのかなど，資源情報と RPC 実行情報が関連して必要となることが多い．

この点に対して，本ツールでは必要な情報を抽出するだけではなく，関連する情報とともにプログラマに提示している．このように，GridRPC アプリケーションのデバッグおよび性能改善の支援を行うには，プログラマが必要な情報のみを提示する機能や，可視化することにより情報を分かりやすく表示し，即座に関連する情報を呼び出して比較できる機能が有効となる．

8. お わ り に

本研究では GridRPC アプリケーションの開発におけるプログラマのデバッグおよび性能改善の際の手間を軽減すべく，GridRPC アプリケーションの開発支援ツールの設計および実装を行った．

評価実験によりツールによる実行時オーバヘッドは無視してもかまわない程度であると分かった．

有用性の検証によってグリッド特有の再現性のない問題に対する原因究明が可能であり，同時にその結果をアプリケーションの性能改善にも利用できることを示した．

今後の課題は以下のとおりである．

(1) 様々な障害に対する本ツールの有用性の検証
本稿では負荷の高い計算資源が全体の性能低下を引き起こしたという障害に対しての本ツールの有用性を示した．今後は表 1 に示した各障害に対しても本ツールが有用であることを検証する．

(2) 大規模なグリッド環境におけるツールの評価
本稿では，クラスタの各計算ノードをグリッドを構成する 1 つの計算機と見立ててオーバヘッドの評価を行った．しかし，実際のグリッド環境においては，規模の大きさやネットワークの不均一性などから，今回の評価より大きなオーバヘッドの値が出ると考えられる．よって，実際のグリッド環境における，本ツールの利用によるアプリケーション実行時間への影響を確認

する必要がある．

(3) 可視化結果をもとにしたアプリケーションの修正を支援する機能の実装
アプリケーションの修正を支援する機能として，可視化ツールから得られた情報をもとに，利用する計算資源，RPC のタイムアウト時間などを記述した Ninf-G 用環境設定ファイルを自動で生成する機能の実装を行う．

(4) 本ツールでは対応していない Ninf-G の API への対応
現在では grpc_call_arg_stack() など，いくつかの Ninf-G の API に対して RPC 実行情報収集機能の実装を行っていないので，それらの API に対しても実装を行う．

(5) ツールのインタフェースおよび機能の改善
現在の可視化ツールの機能に加えて，計算機ごとの処理状況を並べて表示するなどの新しいインタフェースを検討している．

(6) ローカルスケジューラによって割り当てられるジョブへの対応
現在の実装では，単純に計算資源に割り当てられた RPC に関する情報の収集しか行うことができない．しかし，グリッドにおいてはクラスタに対してジョブを実行し，そのジョブがローカルスケジューラによって計算ノードへと割り当てられることが多い．よって，そのような場合にも RPC に関する情報を収集できるようにしなければならない．

(7) Ninf-G 以外の GridRPC システムへの対応の検討
Ninf-G 以外の GridRPC システムにおいても，GridRPC 利用のためのプログラム記述方法に大きな違いはない．よって，本質的な部分では，他の GridRPC システムにおいても本ツールの適用は可能である．また GridRPC の API に関しては標準化が進められており，Ninf-G をはじめとして様々な GridRPC システムがその標準にあわせて実装されている．本ツールもその標準に合わせて実装を行うことで，様々な GridRPC システムにおいて利用可能になると考えられる．一方で，GridRPC の詳細なセッション情報の表示は，Ninf-G および Globus Toolkit の仕様に合わせて実装しているため，その部分は各 GridRPC システムの仕様に合わせて実装する必要がある．

参 考 文 献

1) Seimour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C. and Casanova, H.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing, *Proc. Grid Computing — (Grid 2002)*, pp.274–278 (2002).
2) 武宮 博, 首藤一幸, 田中良夫, 関口智嗣：Grid 環境上における気象予報シミュレーションシステムの構築, 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), pp.155–160 (2003).
3) 谷村勇輔, 池上 努, 中田秀基, 田中良夫, 関口智嗣：耐障害性を考慮した Ninf-G アプリケーションの実装と評価, 情報処理学会論文誌コンピューティングシステム，Vol.46, No.SIG 7, pp.18–27 (2005).
4) 田中良夫, 中田秀基, 朝生正人, 関口智嗣：Ninf-G2: 大規模環境での利用に即した高機能，高性能 GridRPC システム，情報処理学会研究報告 2003-HPC-95, pp.89–94 (2003).
5) Globus Toolkit. http://www.globus.org/
6) 村木健介，川崎康博，水谷奏治，伊野文彦，荻原健一：短時間処理向け資源管理を実現するための Globus Toolkit の性能評価，情報処理学会研究報告 2005-HPC-101, pp.79–84 (2005).
7) 丸山真佐夫, 津邑公暁, 中島 浩：データ再演法による並列プログラムデバッキング, 先進的計算基盤システムシンポジウム SACSIS2005, pp.61–70 (2005).
8) 上島 明, 小畑正貴, 金田悠紀夫：Omni OpenMP コンパイラ用並列プログラム可視化ツール，先進的計算基盤システムシンポジウム SACSIS2005, pp.53–60 (2005).
9) Trace Analyzer. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/index.htm
10) Ganglia. http://ganglia.info
11) SCMSWeb. http://www.opensce.org/components/SCMSWeb/
12) Taura, K.: Grid Explorer: A Tool for Discovering, Selecting and Using Distributed Resources Efficiently, *Summer United Workshops on Parallel, Distributed and Cooperative Processing 2004 (SWoPP2004)*, pp.235–240 (2004).

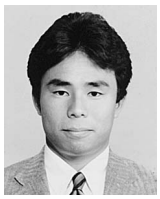
(平成 18 年 1 月 26 日受付)
(平成 18 年 4 月 30 日採録)

小林　孝嗣（学生会員）

2005 年電気通信大学情報工学科卒業．同年より同大学大学院情報システム学研究科博士前期課程在学中．GridRPC アプリケーションの開発支援環境に関する研究に従事．

渡邊　啓正（学生会員）

2003 年電気通信大学情報工学科卒業．2004 年同大学大学院情報システム学研究科博士前期課程修了．同年より同大学大学院情報システム学研究科博士後期課程在学中．計算グリッドのプログラム開発支援環境に関する研究に従事．情報処理学会第 65 回全国大会にて学生奨励賞受賞．



本多　弘樹（正会員）

1984 年早稲田大学理工学部電気工学科卒業．1991 年同大学大学院理工学研究科博士課程修了．1987 年より同大学情報科学研究教育センター助手．1991 年より山梨大学工学部電子情報工学科専任講師．1992 年より同助教授．1997 年より電気通信大学大学院情報システム学研究科助教授．並列処理方式，並列化コンパイラ，並列計算機アーキテクチャ，グリッド等の研究に従事．工学博士．電子情報通信学会，IEEE-CS，ACM 各会員．平成 15 年度山下記念研究賞受賞．