GridRPCを用いたタスクファーミング APIの試作

中田秀基[†,††] 田中良夫[†]
松岡聡[††,†††] 関口智嗣[†]

タスクファーミングとは、大量の同じタスクを異なるパラメータで、複数の計算機で実行することである。このタスクファーミングの実行手段のひとつとしてGridRPCがある。GridRPCはグリッド上のミドルウェアであり、いくつかのアプリケーションの実装を通してその有効性が確認されている。しかし、GridRPC APIは簡潔さを旨に設計されているため、耐故障性やスケジューリングなどの機能を持たず、アプリケーションプログラマの負担となっている。本稿では、GridRPC API上に構築したタスクファーミング APIの設計と実装について述べる。本APIはマスタ・ワーカ型の計算を支援することを意図して設計されており、再実行による耐故障性とセルフスケジューリングを実現する。また、タスクファーミング APIを用いたプログラム例を示す。

A Task-Farming API on GridRPC and its implementation

Hidemoto Nakada [†,††] Yoshio Tanaka [†,††]
Satoshi Matsuoka [††,†††] and Satoshi Sekiguchi [†]

Task-farming means that to compute huge number of tasks for different parameter on large number of computers. Grid RPC is a kind of middleware on the Grid, that is considered to be suitable for task-farming, While Grid RPC provides basic functionarity for task-farming, it lacks high-level features such as scheduling or fault tolerance, due to its design principle, and burdens application programmer to implement them. In this paper we describe Task-farming API implemented on the GridRPC API. It is designed to ease the burden and to support master-worker computation. We also show the implementation of the API and a sample program which uses the API.

1. はじめに

タスクファーミングとは、大量の同じタスクを異なるパラメータで、複数の計算機で実行することである。さまざまな問題がタスクファーミングという形式で実行できることがわかっている。このタスクファーミングの実行手段のひとつとしてGridRPCがある。

GridRPC[1]はグリッド上のミドルウェアのひとつで、これを用いるとクライアントからサーバ上の関数を実行することができる。いくつかのタスクファーミングアプリケーションがGridRPCを用いて実装されており、その有効性が確認されている[2),3)]。しかし、GridRPC APIは簡潔さを旨に設計されているため、耐故障性やスケジューリングなどの機能を持たない。このため、タスクファーミングを実行するためには、アプリケーションプログラマは独自にこれらの部分を実装しなければならず、プログラマにとって大きな負担となっている。そこで、われわれはGrid RPC API上に タスクファーミング ライブラリを作成することでプログラマの負担の提言を図った。

本稿では、GridRPC API上に構築したタスクファーミング APIの設計と実装について述べる。本APIはマスタ・ワーカ型の計算を支援することを意図して設計されており、再実行による耐故障性とセルフスケジューリングを実現する。処理系の実装には、GridRPC Ninf-G[4]を用いた。

本稿の構成は次のとおりである。2節でGridRPCとタスクファーミングに関して概説する。3節、4節で本タスクファーミング APIの設計と実装を述べる。さらに本タスクファーミング APIの使用例を示す。5節で議論を行い、6節で結論と今後の課題を述べる。

2. GridRPCとタスクファーミング

2.1 GridRPC

GridRPCは、クライアント・サーバ型の通信を行うグリッド上のミドルウェアで、クライアントから

† 産業技術総合研究所 National Institute of Advanced Industrial Science and Technology (AIST)
†† 東京工業大学 Tokyo Institute of Technology
††† 国立情報学研究所 National Institute of Information

サーバ上の計算ルーチンを起動することを可能にする。GridRPC は現在 GGF(Global Grid Forum)[5] の GridRPC-WG において標準化作業が進んでいる。GridRPC の仕様はまだ確定していないが、仕様のドラフトが GridRPC-WG の Web サイト [6] から入手可能である。現在、われわれの Ninf-G を含めて 4 つの処理系がこの仕様を実装している。以下簡単に GridRPC の API を紹介する。

2.1.1 GridRPC API

GridRPC API の中心となるのは関数ハンドル構造体である。関数ハンドルは、特定のサーバ上の関数を抽象化したものであり、RPC を行う際には、まずこの関数ハンドルを取得し、次に関数ハンドルに対して呼び出しを行うことになる。関数ハンドルを取得するための関数を図 1(A) に示す。ホスト名とポート番号、関数名を指定している。

このようにして取得した関数ハンドルを用いて、RPC を行う。最も基本的な関数は図 1(B) に示す可変引数関数`grpc_call` である。

この`grpc_call` はブロッキング呼び出しを行うため、マルチスレッドでクライアントプログラムを記述しない限り、並列に実行することができない。これを解決するために、ノンブロッキング版の`grpc_call_async` が用意されている (図 1(C))。この関数は、サーバでの関数実行の終了を待たずにリターンし、その関数実行に対応するセッション ID とよぶ正整数を返す。

`grpc_call_async` で起動した RPC の終了を待つには一連の wait 系関数を用いる。もっとも単純なのは図 1(D) に示す`grpc_wait` である。この関数ではひとつのセッション ID に対する待ち受けしかできない。既存のセッションすべて、もしくはいずれかに対する待ち受けは図 1(E) に示す関数`grpc_wait_all` および`grpc_wait_any` で行う。

`grpc_call`, `grpc_call_async` のバリエーションとして、引数を`grpc_arg_stack` という構造体で受け渡す関数も用意されている (図 1(F))。

これらの関数は、GridRPC をミドルウェアとして使用し、上位により高レベルなプログラムを記述するために用意されたもので、プログラム中で任意の呼び出し引数列を構築できる。`grpc_arg_stack` に対しては、引数をプッシュ、ポップする関数が定義されている (図 1(G))。

2.2 タスクファーミング

タスクファーミングとは、大量の同じタスクを異なるパラメータで、複数の計算機で実行することである。

タスクファーミングで実行できる計算の典型的な例が、パラメータサーベイと呼ばれる計算である。これは特定の関数がパラメータ変動に対してどのように挙動するかを求めることを目的とする計算で、さまざまなパラメータ値にたいして、独立に関数の計算を行うことで実行できる。

```
int grpc_function_handle_init(
    grpc_function_handle_t * handle,
    char * host_name,
    int    port,
    char * func_name);
                          (A)

int grpc_call(
    grpc_function_handle_t * handle,
    ...);
                          (B)

int grpc_call_async(
    grpc_function_handle_t * handle,
    ...);
                          (C)

int grpc_wait(int sessionID);
                          (D)

int grpc_wait_all();

int grpc_wait_or(int * idPtr);

                          (E)

int grpc_call_arg_stack_async(
    grpc_function_handle_t * handle,
    grpc_arg_stack         * stack
);
                          (F)

int    grpc_arg_stack_push_arg(
       grpc_arg_stack * stack,
       void *           data
);
void * grpc_arg_stack_pop_arg (
       grpc_arg_stack * stack
);
                          (G)
```

図 1 GridRPC API の概要

類似した例でモンテカルロ法の計算がある。モンテカルロ法は乱数で生成した値に対して、同じ計算を繰り返し行うことで統計量を産出する計算である。

また、マスタ・ワーカ型計算の一部もタスクファーミングで実行することができる。マスタ・ワーカ型計算の応用範囲は広大である。

2.3 GridRPC API によるタスクファーミングの記述

GridRPC API を用いてタスクファーミングを行うには、前述の`grpc_wait_or` を使用して、スケジューリングを行わなければならない。モンテカルロ法で円周率を求めるトイプログラムのコア部分を図 2 に示す。簡便化のためエラーハンドル関連の文は割愛してある。

まず、サーバに対応する関数ハンドルを作成する。次に、各サーバに対して`grpc_call_async` で初期タスクを投入する。その後、無限ループを回りながら、`grpc_wait_any` を行い、いずれかのタスクの終了を待ち、そのタスクに対応するサーバに対して、新たなタスクを割り当てる。処理するタスクがなくなると、タスクの割り当ては中止する。

すべてのタスクが終了するとループから抜け、関数ハンドルを破棄している。

このように、GridRPC API によってタスクファー

```
...
/* 関数ハンドルの初期化 */
for (i = 0; i < NUM_HOSTS; i++)
  grpc_function_handle_init(&handles[i], hosts[i],
    port, "pi/pi_trial");

/* サーバの数だけタスクを投入 */
for (i = 0; i < NUM_HOSTS; i++)
  ids[i] =  grpc_call_async(&handles[i], i,
                            times, &count[i]);

/* サーバを使いまわしてスケジューリング */
while (1) {
  int i = 0, id, code;

  code = grpc_wait_any(&id);
  if (code == GRPC_OK && id == GRPC_OK)
    break;                              /* 終了 */

  if (code == GRPC_ERROR)
    continue;

  for (i = 0; i < NUM_HOSTS; i++)  /* FIND HOST */
    if (ids[i] == id) break;

  sum += count[i];
  done += times;

  if (done >= whole_times)
    continue;
  ids[i] = grpc_call_async(&handles[i], i,
                           times, &count[i]);
}

/* 関数ハンドルの破棄 */
for (i = 0; i < NUM_HOSTS; i++)
  grpc_function_handle_destruct(&handles[i]);

/* 結果の計算表示 */
pi = 4.0 * ( sum / ((double) done));
printf("PI = %f\n", pi);

...
```

図 2 GridRPC API によるタスクファーミング

ミングを記述することは可能ではあるが、相当に煩雑である。また、計算が失敗した際に再実行するロジックはこのプログラム断片には含まれていない。このロジックを記述すると、プログラムはさらに複雑になり、アプリケーションロジックに集中するべきプログラマにとっては重い負担となる。

2.4 NetSolve のタスクファーミング API

GridRPC システムのひとつである NetSolve[7] は、request farming と呼ぶタスクファーミング API を、システム API のひとつとしてサポートしている。

この API は、各タスクに対する引数をあらかじめ配列として用意しておき、この配列から iterator オブジェクトを作成し、関数`netsl_farm` に渡して実行するものである。この際に、配列に対するレンジを指定することができる。

iterator オブジェクトの作成の際には、引数配列とともに、インデックス変換文字列を指定することができる。これによって、柔軟な引数配列からのマッピングが実現されている。

request farming API の使用例を図 3 に示す。このプログラムは、`iqsort` という関数を、200 回呼び出している。その際、i 番目のイタレーションでは、`size_array[i]`, `ptr_array[i]`, `sorted_array[i]`,

```
int size_array[200];
void *ptr_array[200];
void *sorted_array[200];

/* 配列のセットアップ */
size_array[0] = size1;
ptr_array[0] = ptr1;
sorted_array[0] = sorted1;
...

/* request farming の呼び出し */
status_array =
        netsl_farm("i=0,199","iqsort",
                   ns_int_array(size_array,"$i"),
                   ns_ptr_array(ptr_array,"$i"),
                   ns_ptr_array(sorted_array,"$i"));
```

図 3 NetSolve の Request farming API

を引数となる。

3. タスクファーミング API の設計

3.1 要　　請

タスクファーミング API に対しては以下が要求される。

- 耐故障性
- スケジューリングとスロットリング
- ファーミング実行中のプログラム実行

3.1.1 耐故障性

大量のサーバ群を用いる大規模計算においては、すべてのサーバが計算の終了まで稼動し続けることを期待すべきではない。したがって、計算実行中に特定のサーバがダウンした際には、ダウンを安全に検出し、そのサーバをサーバプールから除外しなければならない。さらに、ダウンしたサーバ上で実行中であったタスクを、他のサーバ上で再実行しなければならない。

3.1.2 スケジューリングとスロットリング

サーバ群に対して大量のタスクを実行する際には、タスクにたいして適切なサーバを割り当てなければならない。タスクの数がサーバの数に対して十分大きいことを仮定すれば、サーバの選択自体は問題にならないため、選択手法はラウンドロビンなどの単純な手法で十分である。

もうひとつ重要なことは、ひとつのサーバに対して複数のタスクを割り当てることでサーバの速度低下を引き起こすことのないようにすることである。

3.1.3 ファーミング実行中のプログラム実行

タスクファーミングで重要なことのひとつが、タスクのセットアップとクリーンアップである。これらをタスクファーミングの前後にまとめて行うと、すべてのタスクをメモリ上に展開することになり、物理メモリ実装量を超えるタスクの実行が難しくなる。したがって、タスクファーミングの実行中にも、タスクのセットアップやクリーンアップが実行可能なインターフェイスを提供する必要がある。

また、ファーミング実行中に、実行終了したタスクの結果から新たなタスクを生成して実行したい場合が

```
int ng_group_init(
    ng_group_t              * group,
    char                    * grpc_config_file,
    char                    * machine_file,
    char                    * function_name,
    ng_group_arg_cleanup_f    arg_cleanup,
    ng_group_result_consume_f result_consume
);

int ng_group_set_property(
    ng_group_t              * group,
    int                       key,
    void                    * val
);

int ng_group_fin(
    ng_group_t              * group
);
```

図 4 タスクファーミング API の関数群 1

ある。これを実現するためにも、ファーミング実行中にユーザプログラムからなんらかの制御ができなければならない。

3.2 タスクファーミング API

タスクファーミング API は、ng_group_t という typedef された構造体と一連の関数群によって構成される。図 4 に、ng_group_t を操作する関数群を示す。

ng_group_init はng_group_t を初期化する関数である。第 2 引数に下位システムとして使用する Grid RPC システムの設定ファイルを、第 3 引数にマシンファイル、第 4 引数に関数名を指定する。第 5、第 6 引数には、送信引数のクリーンアップ関数およびタスクの結果を処理するコールバック関数のポインタを指定する。これらのコールバック関数に関しては後述する。

この関数は、第 2 引数で渡される設定ファイルを用いて下位の Grid RPC システムを初期化し、マシンファイルに記述されているサーバに対して、関数名で指定された関数のハンドルを作成する。マシンファイルの書式は、一行ごとに計算記名とポート番号を羅列したものとなっている。

ng_group_set_property は、プロパティを設定する汎用関数である。現在用意されているプロパティは下記の 2 つである。

- NG_GROUP_USE_ARG_STACK
  コールバック関数に 2.1.1 で述べた、arg_stack を渡すかどうかを指定する。コールバック関数は、このスタックから引数を取り出し処理を行う。処理が必要なければ、このプロパティを 0 にセットすればよい。デフォルトでは渡すよう設定されている。
- NG_GROUP_AUTO_REINVOKE
  タスクの実行が何らかの理由で失敗した際に、再実行を行うかどうかを指定する。デフォルトでは行わない。

ng_group_fin は終了関数である。内部で使用したメモリを解放するとともに、使用したハンドルをすべて解放する。さらに、下位の Grid RPC システムを終了する。

```
int ng_group_call(
    ng_group_t              * group,
    void                    * user_data,
    ...
);

int ng_group_call_arg_stack(
    ng_group_t              * group,
    void                    * user_data,
    grpc_arg_stack          * args
);

int ng_group_wait_done(
    ng_group_t              * group
);
```

図 5 タスクファーミング API の関数群 2

```
typedef void (* ng_group_arg_cleanup_f) (
  grpc_arg_stack * args,
  int              serial,
  void           * user_data
);

typedef void (* ng_group_result_consume_f) (
  grpc_arg_stack * args,
  int              serial,
  void           * user_data,
  int              session_id
);
```

図 6 コールバック関数の定義

図 5 にタスク呼び出しと終了待ちうけに使用する関数群を示す。ng_group_call が可変引数版の呼び出し関数。ng_group_call_arg_stack は、GridRPC API のgrpc_arg_stack 型を用いた呼び出し関数である。第二引数で指定するユーザデータは、後述するコールバック関数に渡されるデータであり、任意のデータを指定することができる。

ng_group_wait_done はすべてのタスクの終了が完了するのを待つ関数である。

ng_group_t の初期化時に指定するコールバック関数は図 6 に示すように定義されている。

双方の第一引数には、前述したgrpc_arg_stack が渡される。第二引数は、このgrpc_group_t を用いた何番目の呼び出しであるかを示すシリアル番号が渡される。第三引数には、RPC 実行時にユーザが指定したデータが渡される。ng_group_result_consume_f の第 4 引数は、そのタスクを実行した Grid RPC のセッション ID である。

4. タスクファーミング API の実装

タスクファーミング API の実装は、Grid RPC システム Ninf-G をターゲットとして行った。Ninf-G が使用する Globus のライブラリに、non thread 版と pthread 版があるため、タスクファーミング API の実装もこの両者を用意した。

4.1 逐次版の実装

逐次版は、関数ハンドルのプールとgrpc_wait_or で実現されている。

ng_group_call では、関数ハンドルのプールからハンドルを取り出し、RPC を実行する。空いたハンドルがない場合には、grpc_wait_or でいずれかのハンドルの終了を待ってブロックする。

RPC の実行が終了した際にはコールバック関数を適切に呼び出した後、使用したハンドルをハンドルプールに戻す。RPC の実行に失敗した場合には、ハンドルをハンドルプールに戻さずに破棄する。この際、自動再実行のフラグがセットされている場合には、プールから新たにハンドルを取り出して再実行を行う。

4.2 pthread 版の実装

pthread 版の実装は逐次版のそれとはまったく異なり、grpc_wait_or を用いない。各サーバに対してひとつのスレッドを割り当て、ブロッキング呼び出しで RPC を行う。ユーザプログラムは独立したスレッドで実行され、サーバに対応するスレッドと、キューを介して通信を行う。コールバック関数は個々のスレッドで独立して実行されるので、排他制御が必要な変数に関してはユーザが排他制御を記述する必要がある。

ユーザプログラムはキューにタスクを投入する。サーバスレッドはこのキューからタスクを取り出して RPC を実行する。キューには長さの制約が設けてあり、制約を超えてタスクを投入しようとするとブロックする。これは、過度にタスクを生成することでメモリを圧迫しないようにするためである。

4.3 使 用 例

タスクファーミング API を使用したプログラム例を図 7 に示す。このプログラムは、モンテカルロ法で円周率を求める非常に単純なプログラムである。cleanup ファンクションは使用せず、consume ファンクションのみを使用している。

冒頭で定義されている関数consume が、各呼び出しの終了時に呼び出される関数である。この関数では、まず、呼び出しが正常に終了したかどうかを判断し、正常に終了した際には、user_data から返り値が収められているポインタをとりだし、そこを参照することで返り値を取得して、グローバル変数を更新している。

メインルーチンでは、まず ng_group 構造体を初期化し、次に、オプションのアトリビュートを設定している。ここでは、arg_stack の保存を無効化、自動再実行を有効化している。

つづくメインループで、関数の呼び出しを行っている。このループでは単純に ng_group_call を呼び出しているだけだが、すべての使用サーバに対してタスクが割り振られている際には、自動的にng_group_call の内部で、いずれかのタスクが終了してサーバがあくのを待つことになる。

続くng_group_wait_done ですべてのタスクの終了を待つ。タスクが終了したら結果を表示して、ng_group_fin で後始末を行っている。

```
#include "ng_group.h"

long trial;
long in;
long count;

/* consume ファンクション */
void consume(grpc_arg_stack * stack,
     int              serial,
     void           * user_data,
     int              session_id
){
  int error_code = grpc_get_error(session_id);
  printf("consuming ... \n");
  if (error_code != GRPCERR_NOERROR){
    fprintf(stderr, "%s\n",
            grpc_error_string(error_code));
    trial -= count;
  } else {
    in += *((long *)user_data);
    free(user_data);
  }
}

/* メイン プログラム */
int main(int argc, char * argv[]){
  ng_group_t group;
  long no_of_trials;
  int div;
  int counter = 0;

  no_of_trials = atol(argv[3]);
  div = atoi(argv[4]);
  count = no_of_trials / div;

  /* 構造体の初期化 */
  if (ng_group_init(&group, argv[1], argv[2],
                    "pi/pi_trial",
    NULL, consume) != GRPC_OK){
    fprintf(stderr, "failed to group init\n");
    exit(2);
  }

  /* オプションのセットアップ */
  {
    int use_arg_stack = 0;
    int auto_reinvoke = 1;
    ng_group_set_property(&group,
                          NG_GROUP_USE_ARG_STACK,
                          &use_arg_stack);
    ng_group_set_property(&group,
                          NG_GROUP_AUTO_REINVOKE,
                          &auto_reinvoke);
  }

  /* メインループ */
  while (no_of_trials > 0){
    no_of_trials -= count;
    long * out = (long *) malloc(sizeof(long));

    fprintf(stderr, "calling %d\n", counter++);
    if (ng_group_call(&group, out, random(),
                      count, out)
        != GRPC_OK)
      break;
    trial += count;
  }

  /* 終了待ち */
  if (ng_group_wait_done(&group) != GRPC_OK){
    fprintf(stderr, "failed to group wait\n");
    exit(2);
  }

  /* 結果出力 */
  fprintf(stdout, "PI =:= %f\n",
          4.0 * (in / (double)(trial)));

  /* 構造体のクリーンアップ */
  if (ng_group_fin(&group) != GRPC_OK){
    fprintf(stderr, "failed to group fin\n");
    exit(2);
  }

  return 0;
}
```

図 7 タスクファーミング API を用いたプログラム例

5. 議　　論

5.1 NetSolve request farming API との比較

2.4 で示した NetSolve の request farming API は、本 API と同様にタスクファーミングを行うことを目的で設計されているが、以下のような相違点がある。

本 API は大容量のデータを動的に生成、解放しつつ実行することを想定しているのに対し、requst farming API は比較的小規模のデータを静的に生成しておいて、実行することを想定している。このため、request farming API を用いる場合、引数データの総量の規模が使用可能メモリに制約を受ける。これに対して、本 API では、実行中の呼び出しに関する引数データのみがメモリ上にあればよいので、メモリからの制約を受けにくい。

また、request farming API では、静的に定めた回数の実行を行うため、動的な状況変化に応じて実行回数を変更することができない。本 API ではコールバック関数の中からの RPC 呼び出しも可能であるため、RPC 呼び出しの結果に応じて、次の RPC 呼び出しを行うかどうかを決定したり、呼び出しをの引数を変更することも可能である。このためより広い範囲の問題を容易に記述することができる。

5.2 GridRPC API の問題点

本 API の実装過程で、現在 GGF の GridRPC-WG で策定中の GridRPC API が提供する関数群では実装できない部分が存在することが判明した。具体的には、`grpc_arg_stack` に関する部分で、`va_list` から `grpc_arg_stack` を作成することと `grpc_arg_stack` の複製を作成することが、現在の GridRPC API ではできない。このため、現在の実装では、これらの部分を Ninf-G での `grpc_arg_stack` の実装に依存して作成している。

しかし、これらの機能は十分に一般的であり、他の Grid RPC API を用いたアプリケーションでも必要になることが考えられる。追加する API の機能とインターフェイスを十分に整理、検討したうえで、これらの関数の追加を GGF の GridRPC-WG において提案していく予定である。

6. おわりに

本稿では GridRPC Ninf-G 上に構築した、タスクファーミング API の設計と実装について述べた。さらに、本タスクファーミング API を用いたプログラムの記述例と実行例を示した。

今後の課題としては以下が挙げられる。

- タスクファーミング API の妥当性の評価
  大規模なプログラムを実装して、API の妥当性を確認する必要がある。同時に今回実装したライブラリの頑健性を評価していく。また、解探索のアプリケーションでは、計算をキャンセルする機構が必要になる可能性がある。キャンセルを実現する手法についても検討を進める。
- サーバ計算機の追加、削除
  現在のタスクファーミング API は、サーバ計算機群は初期化時にファイルから読み込むだけであり、実行中に増減することができない。これは実行が長期にわたる場合には不便である。このため、サーバを動的に登録削除する機能の追加を検討している。
- ログ機能とログの視覚化
  タスクファーミング API によってプログラムの記述は容易になったが、大規模な実行に不可欠の実行のモニタおよび解析を行うにはユーザがプログラミングをする必要がある。タスクファーミング機構の内部にログを出力する機能を設け、別プログラムで視覚化機能を提供することで、モニタと解析を容易にすることを検討している。

謝　　辞

本 API の策定に協力いただいた産総研グリッド研究センターの武宮博氏に感謝する。

参 考 文 献

1) Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C. and Casanova, H.: GridRPC: A Remote Procedure Call API for Grid Computing, submitted to Grid2002.
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, SACSIS2003 論文集, pp. 251–258 (2003).
3) 池上努, 武宮博, 長嶋雲兵, 田中良夫, 関口智嗣: Grid: 広域分散並列処理環境での高精度分子シミュレーション — C20 分子のレプリカ交換モンテカルロ, SACSIS2003 論文集, pp. 243–250 (2003).
4) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *Journal of Grid Computing*, Vol. 1, No. 1, pp. 41–51 (2003).
5) Global Grid Forum. http://www.gridforum.org.
6) GridRPC-WG. http://graal.ens-lyon.fr/GridRPC.
7) 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).