Preliminary Study of A Task Farming API over The GridRPC Framework

Yusuke Tanimura, Hidemoto Nakada, Yoshio Tanaka, and Satoshi Sekiguchi
Grid Technology Research Center
National Institute of Advanced Industrial Science and Technology
{yusuke.tanimura, hide-nakada, yoshio.tanaka, s.sekiguchi}@aist.go.jp

Abstract

In this paper, a middleware suite, which provides a Task Farming API, is studied in use over the GridRPC standard, in order to reduce the complexity of developing task parallel applications for the grid. APIs are proposed and higher functionality in task scheduling and fault tolerance is implemented in the middleware, based on our past experiences with the Ninf-G. Through our study, it is revealed that the Argument Array API needs to provide a means to copy arguments for duplicated task assignment. Timing of data transfer in the non-blocking RPC and a method to retrieve execution information for each RPC are expected to be standardized in the GridRPC. By resolving these three issues in the GridRPC, our Task Farming API library, meeting application requirements, can be fully realized on multiple GridRPC systems, paving the way for other higher functional API libraries to be designed and implemented.

1. Introduction

Both parameter-sweep applications and master-slave applications would achieve their large-scale calculations in a realistically short time on the grid, if the grain size of each sub task were large enough. The GridRPC[8] is a framework to develop such applications easily. The End-User API set of the GridRPC provides an API to call a remote-library function as if the function exists at the local end. The set succeeds in hiding most of the complexity of the grid, such as differences in machine architecture, communication methods, and so on, and in improving the programmability of scientific applications.

We have worked on the Ninf project[6] since 1994, and developed the Ninf-G as a reference implementation of the GridRPC. Collaborative research with application people who study physics or chemistry indicated to us the significance of the GridRPC framework and the Ninf-G software[10]. In particular, the stability of the Ninf-G, and how to implement a fault-tolerant application using the GridRPC are shown in our studies[9, 11], one of which performs a long-term experiment to solve a TDDFT equation[11]. A higher-level functionality should be implemented in order to accomplish the long time execution on several hundreds of nodes on the grid. On the other hand, the End-User API provides only a primitive function according to the design basis of the GridRPC. Application programmers need to specify the destination of the RPC or append codes for fault-tolerance over the primitive function. This is really inconvenient because they must implement some of those functions themselves so that their applications continue to run for a long time on hundreds of nodes to contribute to scientific discovery. Otherwise, their applications will hang up on an unstable network, or will not achieve the high performance they expect.

Based on this background, this paper targets cost reduction of developing a real-science application. We discuss a Task Farming API library that implements common components and hides their complexity from applications. Task Farming runs the same program in parallel while changing input data and parameters. The Task Farming API enables programmers to produce task farming code easily, and to have almost the best performance and stability possible without a great amount of effort. Currently, NetSolve[2] and Ninf[6] projects implement the Task Farming API in their studies. NetSolve has a performance issue because its asynchronous call does not perfectly implement non-blocking data transfer and the farming API is built on that. The NetSolve farming API consists of only one function call and the farming is completed in the function. This forces programmers to wait for any result until all tasks are done. Some applications, such as image processing, which shows a processed image in real time, may want to retrieve a result as soon as the task is finished[3]. Ninf is implemented with a callback mechanism to spend memory capacity frugally[5]. The callback approach is utilized for post-processing of each task. The callback function is executed after one task is done, so that the result data is passed to the user's memory space from the Ninf library. This approach means that the Ninf doesn't have to have the entire memory space to receive results of all tasks at the moment. However, this approach is not friendly for some application programmers, who are not accustomed to C programming rather than Fortran, to write code for the callback method.

In this study, we summarize the user's requirements for the Task Farming API and redesign it with task scheduling and fault-tolerance. Through discussion of our design and its implementation, we will point out what the GridRPC standard should include or modify in order to build a higher-functional API set like that required for Task Farming. Our paper uses the Ninf-G, but the discussion and the results can be applied to other GridRPC systems.

2. GridRPC and Task Farming

In this section, the GridRPC and its current status are introduced, and widely anticipated functions for Task Farming are summarized.

2.1. Status of GridRPC

The GridPRC is one of the programming models for a grid application. An application program using the GridRPC consists of main code and a segment of stub code. When the main program invokes an RPC, the corresponding stub program runs on a remote machine. The stub program must be prepared on the remote before the RPC is invoked or be shipped to the remote at the same time as the RPC invocation. A task-parallel program can be written with asynchronous RPC.

The GridRPC API is being standardized at the working group of the GGF[4]. At this time, in August, 2005, the End-User API is in the final phase, and is going to be released as a recommendation document. The End-User API is designed to provide a primitive API set, and then some issues, such as scheduling and fault-tolerance, should be considered in the application program or in the higher middleware over the primitive set. There are two major higher API sets in consideration. One is the Task-Farming that this paper targets, and the other is Task-Sequencing[1] which enables programmers to write a single execution for grouped tasks. Output data of one task becomes input data of another task in grouped tasks. The middleware that provides the Task Sequencing API implements data persistency on the remote for avoiding unnecessary data communication.

Currently, the GridRPC working group is engaged in discussion about the Middleware API sets to support implementation of those higher APIs over the GridRPC framework. Especially, the Argument Array API set to handle arguments of the RPC, and the Data Handle API to treat data persistency, have been discussed.

2.2. Requirements for a Task Farming API

- Design for making a Task Farming tool over APIs

  A Task Farming API set should allow users to implement other tools over itself. For example, application people expect to submit a task with interactive tool like Matlab or other scripting languages[3]. Some application people may want to concentrate on an algorithm for generating parameters and post-processing of each task.

- Programmability of the C-based GridRPC API

  The End-User API of the GridRPC abstracts a remote library with a function handle. Each handle corresponds to a unique function on a specific remote machine. The function handle must be indicated in invoking each RPC. However, it is convenient for application people if the GridRPC system allocates the RPC to an appropriate server. Moreover, users can benefit from a performance improvement by the sophiscated implementation of the middleware. In this case, every task should be watched by the middleware. An application program only has to wait for the completion of some or all tasks, using "wait any" or "wait all," like an API. Some application users prefer using the qsub-like semantics of the batch system in front of remote libraries. Their main reason is because the interface is friendly to most scientific application users.

  Another requirement is to provide remote initialization for the Task Farming. The initialization should be performed with either uniform or non-uniform parameters. The registration mechanism of an initialization method and its arguments, and an automatic invocation mechanism for the method should be implemented inside of the Task Farming library. In some applications, post-processing must be executed immediately after the task is completed.

- Automatic task allocation

  The middleware should implement efficient task assignment by monitoring the status of computers and networks. A round-robin task assignment method should be implemented as basic policy, and pooled servers are arranged in the order of performance or stability. This self-scheduling, based on real-time monitoring, seems to be useful. Of course, the number of task submissions should be limited or blocked for a while by considering memory or disk capacity on the client node.

- Fault-tolerant mechanism

  From most application users, the application program must be kept running despite occurrence of faults. The failed task should be resubmitted to another live server without returning an error to the application. After the server node is back online from the fault, the remote program should be restarted, and a certain number of servers should be maintained as a computational environment. In order to improve task processing throughput, duplicated task assignment is expected. The logging function will also be useful for both users and system administrators to find the cause of the fault.

3. Design and implementation of a Task Farming API

In this section, design of the Task Farming API is described, and implementation of it using the latest Ninf-G as a GridRPC system, is introduced.

3.1. Design

- As a pre-process of the Task Farming, remote initialization of each server program is supported. When the initialization method is prepared on a remote, a client program can register the method as the initialization method. The parameters for the method are also registered at the same time.

- Each remote program can be given an ID by the client program. The client program can specify the destination of the RPC with this ID. This function is left for some applications to take care of their RPC destinations by themselves.

- By watching waiting tasks and monitoring the availability of servers, tasks are assigned to an appropriate destination. However, the range of the task assignment is limited so as not to use up all memory space, and to achieve the best performance.

- In case a fault happens, a new task is not assigned to the affected server, and the failed task is resubmitted to another live server automatically.

- When a node is back, the remote program is automatically restarted and initialized by the initialization method, which is registered in the client library.

- When the number of unprocessed tasks is few, a function to assign one task to multiple servers is supported.

3.2. GridRPC API provided by Ninf-G

The details of the Ninf-G are introduced before explaining our implementation of the Task Farming API. The Ninf-G has been developed by AIST (National Institute of Advanced Industrial Science and Technology, Japan) and TITECH (Tokyo Institute of Technology, Japan). The Ninf-G is a reference implementation of the GridRPC. Several kinds of applications have been implemented with Ninf-G since November 2002, when version 1.0 was released. Through those evaluations, performance, scalability and stability have been improved[9, 11]. Moreover, extended APIs and functions, which are not defined in the GridRPC, are added to the Ninf-G to meet requirements from the application side.

By specifying an option in the configuration file, a Ninf-G application programmer can handle timing of data transfer in an asynchronous function, enable compressed communication, and use a heartbeat function to achieve the best performance and stability. Normally, a default value is set, but the value just helps users to run their applications with more stability. Since the user who needs the best performance must care about the configuration, it is expected that these functions be hidden inside of the higher functional library.

As for the others, the Ninf-G original APIs are provided with an "_np" suffix. There are 3 major extensions. 1) A function to start multiple remote programs by one call, 2) A function to share a variable between more than 2 RPCs in the remote program (Remote-object function), 3) A function to stack the arguments of an RPC and to execute the RPC with them. A main reason to have to use 1) is that Ninf-G is built on the Globus Toolkit. On the other hand, 2) and 3) are issues related to other GridRPC systems. 2) and 3) are implemented differently on each of them.

3.3. Implementation

An overview of our implementation is shown in Figure 1. Because Ninf-G works on Globus, the Globus gatekeeper starts a remote program, which is called a Ninf-G server, through a local batch system. In consideration of the performance and the stability of Globus version 2.x/3.x, we use a tentative method to boot multiple remote programs by one function call of the GRAM (Grid Resource Allocation Manager) client API. This is implemented in the Ninf-G as function 1), which is explained in the previous subsection. In calling the booting function, a function handle array is returned from the Ninf-G. Each function handle that belongs to the same function handle array is taken to pieces and put into a server status pool. A handle of the server that has just started is stored in the Down pool. During execution of the initialization method on the server, the handle is in

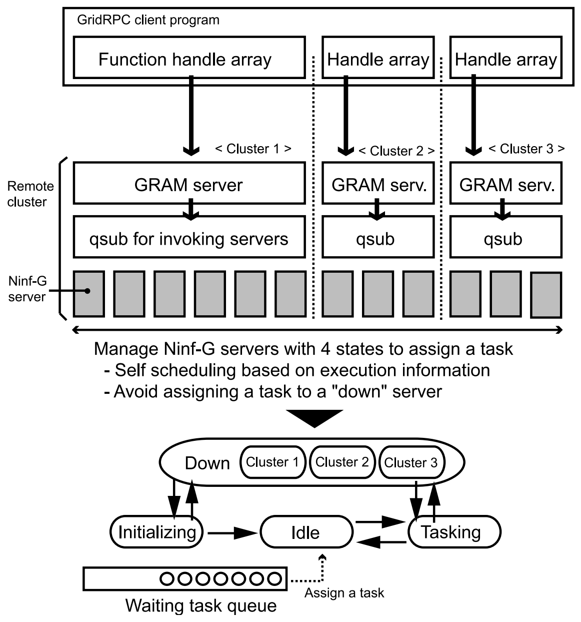

Figure 1. An overview of the Task Farming API

```
/* Initialization of an argument array */
grpc_error_t grpc_arg_array_init(
                grpc_function_handle_t * handle,
                grpc_arg_array_t * array);

/* Initialization of an argument array from va_list */
grpc_error_t grpc_arg_array_init_with_va_list(
                grpc_function_handle_t * handle,
                grpc_arg_array_t * array,
                va_list list);

/* Put an argument pointer in the array */
grpc_error_t grpc_arg_array_put(
                grpc_arg_array_t * array,
                int index,
                void * item);

/* Get an argument pointer from the array */
grpc_error_t grpc_arg_array_get(
                grpc_arg_array_t * array,
                int index,
                void ** item_ptr);

/* Asynchronous RPC with the argument array */
grpc_error_t grpc_call_arg_array_async(
                grpc_function_handle_t * handle,
                grpc_sessionid_t * sessionId,
                grpc_arg_array_t * array);
```

Figure 2. Major Argument Array APIs

the Initializing state. The handle is stored in a global Idle pool after initialization. All handles are arranged by the order that initialization or task execution is done. Based on the user's preference, the handles are rearranged by the order of the performance achieved according to the RPC execution information. A handle indicating which server is tasking is stored in the Tasking pool. A task is also managed with a single queue and the task information is placed in the queue in the order that the task is submitted from an application program. The first task is assigned to the server whose handle is first in the Idle pool. Initialized data is utilized in the task execution, which is realized by function 2) in the previous subsection. A handle of a server with a fault goes back to the Down pool. When the number of down servers increases over a certain ratio, the function tries to restart all the servers that belong to the same function handle array.

Additionally, a higher API, such as the Task Farming API, might provide some functions that have an arbitrary number of arguments. This is not possible to implement with the End-User API, but the GridRPC working group has started discussion about standardizing the Argument Array API set, as shown in Figure 2. The Argument Array API set is a redesign of function 3) in the previous subsection. Since the API set allows programmers to create RPC arguments from a va_list, the Task Farming middleware separates task submission from an application program and task execution on the server, in order to control task execution inside of the Task Farming API library.

3.4. API

Figure 3 shows our proposed API set. First, a client program calls grpc_init() with a configuration file that describes the hostname of the front-end of the cluster to be used, the full path to a remote executable, and the number of servers to be started in the cluster. Parameter sched is a structure which includes task assignment policy, and parameter ft is another structure which includes a policy for the fault-tolerant mechanism. Next, grpcg_remote_init() should be called to boot the remote programs. The number of remote programs is specified by num_pe and the cluster to be used is given in the configuration file. In grpcg_remote_init(), func is the name of the initialization method, the rest are arguments of the method. This is an overview of registration of the initialization method in start/restart of the servers. The initialization method is processed inside of the library. The client program can call grpcg_remote_init_n() instead of grpcg_remote_init(). This function sets an ID for each server and registers a different initialization method to be assigned to each server. The ID is an integer and is given as server_id in Figure 3.

There are 4 kinds of task submission APIs in our set. The most common API is grpcg_submit(), in which programmers only have to specify a main calculation method of the task and the arguments. The submitted task will be assigned to an appropriate destination by the Task Farming library. When programmers want to specify the destination, they can use grpcg_submit_n() to indicate the destination with an ID that is assigned at grpcg_remote_init_n(). Some applications want to give a reference pointer to each task and to

```
/* Initialization and finalization of the Task Farming
    library */
int grpcg_init(char *conf, sched_attr_t *sched,
               ft_attr_t *ft);
int grpcg_fin();

/* Invocation and halt of a remote program
   (Ninf-G server) */
int grpcg_remote_init(int num_pe, char * func, ...);
int grpcg_remote_init_n(int server_id, int num_pe,
                        char * func, ...);
int grpcg_remote_fin(int num_pe);
int grpcg_remote_fin_n(int server_id, int num_pe);

/* Task submission */
int grpcg_submit(char * func, ...);
int grpcg_submit_n(int server_id, char * func, ...);
int grpcg_submit_r(void * ref, char * func, ...);
int grpcg_submit_nr(int server_id, void * ref,
                    char * func, ...);

/* Wait and cancellation of a submitted task */
int grpcg_wait_all();
int grpcg_wait_any(int * task_id, void ** ref);
int gprcg_cancel(int task_id);
```

Figure 3. Our proposed Task Farming API

use it in the post-process. grpcg_submit_r() provides a function to store the pointer for that purpose. grpcg_submit_nr() is a combination of grpcg_submit_n() and grpcg_submit_r(). There are 2 APIs used to wait for a task. grpcg_wait_all() is used to wait for all tasks. grpcg_wait_any() is used to wait for the task that is completed the earliest. Server rearrangement based on the RPC execution information only operates at the end of grpcg_wait_all(). This will have an effect on the next farming step in the client program.

grpcg_remote_fin() is used to finalize the remote program and grpcg_fin() is used to finalize the Task Farming library.

3.5. Sample code from the Task Farming API

A code sample for using our proposed Task Farming API is shown in Figure 4. This code is a reimplementation of the ED (Embarrassingly Distributed) benchmark, which is defined in the NAS Grid Benchmark[7]. The ED doesn't require an initialization method, and so the second argument of grpcg_remote_init() is null. In the Task Farming part, the S-class SP is a task and its task number is given with an "&i" argument that becomes a parameter used to execute the SP. Other arguments of the task in submission are static values. The sample program submits "numTask" tasks with those arguments and just waits until all of them are completed.

Compared with End-User API programming, Task Farming programming is simple because the destination of the task, resource management, and error handling are completely hidden from the application code. grpcg_submit() actually calls grpc_call_async() of the End-User API inside the Task Farming library, but grpcg_submit() saves the RPC request once and invokes grpc_call_async() with the proper destination at the appropriate time. Similarly, grpcg_wait_all() calls grpc_wait_any() inside the library, but grpcg_wait_all() only returns after all tasks are completed or a critical error happens, while grpc_wait_any() returns upon detecting any RPC error. Thus our Task Farming library allows programmers to concentrate on parameter generation or post-processing of the farming.

```
  :
rc = grpcg_init("servers.list", &sched, NULL)
if(rc != GRPCX_OK){
  perror("grpcg_init() failed.");
  exit(1);
}
grpcg_remote_init(NUM_PES, NULL);

for(i=0; i<numTask; i++){
  grpcg_submit("SP.S", "SP", &npbClass, &ngbClass,
               &ascii, "ED", &i, &width, &depth,
               &pid, &verbose, &filter,
               &numInput, &numOutput, &ngbBin,
               &report[i]);
}
rc = grpcg_wait_all();

grpcg_remote_fin(NUM_PES);
grpcg_fin();
  :
```

Figure 4. A sample application code of the Task Farming API

Our proposed API set can be applied to many Task Farming applications, whose each task has no dependencies, except one dependency between initialization and main calculation. Some applications, such as the branch and bound method, however, might not fit into our API framework if data communication for bounding is required outside of the task execution. Because the proposed APIs insist on hiding task destination and timing of data transfer, it is hard to apply the APIs to a program that wants to identify them. It is obviously better for those applications to use the End-User API set.

4. Issues for developing a higher functional middleware suite

In this section, several issues for developing a higher functional middleware suite such as the Task Farming software and possible solutions involved with the GridRPC standardization, are pointed out.

4.1. Ninf-G extensions

The Ninf-G-specific APIs are shown in Figure 5, except the Argument Array API set. First, a remote object to share data between multiple RPCs is utilized to implement a remote initialization. The remote object achieves this by storing initialized data inside of the remote server and allowing the next RPC to refer to the same data.

```
/* Multiple starts and halts of a remote program that
   implements a remote object function */
grpc_object_handle_array_init_np();
grpc_object_handle_array_destruct_np();

/* Retrieval of the data transfer time and remote
   calculation time of the RPC */
grpc_get_info_np();

/* Error ouput */
grpc_perror_np();
```

Figure 5. Ninf-G's extended GridRPC APIs that are utilized for implementation of our Task Farming API library

Second, the multiple-booting method is utilized to achieve high performance. As previously mentioned, the main cause of low performance is that the Globus doesn't show enough performance and stability. Keeping performance up, however, is important for practical execution of scientific applications. In calling an asynchronous RPC, the "transfer_argument" parameter of the Ninf-G is configured as "nowait." This option keeps the asynchronous RPC from blocking until all of the RPC arguments are transferred to the remote. In the current GridRPC, this option depends on implementation of each GridRPC system. For example, NetSolve provides one option such that the asynchronous RPC function returns after data transfer is completed. Because of that, the Task Farming feature of NetSolve cannot achieve sufficient performance. In this study, the RPC arguments were managed inside of the middleware, and blocking/non-blocking was appropriately selected by the middleware.

The ordering of the remote server in the Idle pool makes use of the execution information of the last RPC. grpc_get_info_np() is used for retrieving the information measured by the Ninf-G. The function returns the transfer time of the input data and the output data of the RPC, and the execution time of the RPC on the remote, by the session ID that is assigned to the RPC.

These extensions are necessary to implement our proposed Task Farming API set. The remote object might be implemented another way and should be discussed along with the Data Handle API set. Booting multiple servers with one call depends on the implementation of Globus. Those two issues are beyond the scope of this paper and they should be studied and discussed more. On the other hand, standardization of non-blocking data transfer and a method to retrieve execution information would be widely useful.

4.2. Requirements for an Argument Array API

This paper assumed that the Argument Array API is available as a Middleware API set of the GridRPC. The proposed Argument Array API stores only pointers for the arguments and it does not copy the data itself. This is inconvenient for developing higher functional middleware, for two reasons.

The first reason is that a higher API should not require application programmers to rewrite the data that is passed to the library by the Argument Array API. Because the End-User API forces programmers to manage a session of the RPC, they can avoid rewriting data consciously. Task Farming, however, hides the session and it is inappropriate that programmers need to care about a covered session's status.

The second reason is that it is hard to implement duplicated task assignment. If arguments can be copied inside of the library, output arguments from the remote will also be duplicated. Each task can just receive the result at grpcg_wait*(), store the result on the appropriate memory space, and the result that was completed earlier is simply returned to the application. Otherwise, another method or API to lock the data receive should be included in the Argument Array API set.

Therefore, the copy function for the RPC arguments is expected to be added to the Argument Array API set. Of course, the client node must have enough memory space to store a copy of the arguments and this might cause insufficient memory capacity. Our proposed API, however, has a function to limit the number of task submissions, and it is possible to incarnate automatic tuning of limiter values. Besides, some scientific applications treat the RPC arguments as a file. In our results we found, it is much more advantageous to provide the copy function in the Argument Array API set.

5. Summary and future work

In this paper, the Task Farming API set was designed as a higher functional middleware suite over the GridPRC. The API set is implemented with the End-User API of the GridRPC, some Ninf-G extensions, and the Argument Array API. In order to meet the requirements of the Task Farming API, a method to retrieve execution information for the RPC and a data transfer option are expected to be standardized in the GridRPC. Furthermore, it was found that the copy function of the RPC arguments is necessary as part of the requirements for the Argument Array API. We plan to implement and evaluate a full functional Task Farming API with an advanced Argument Array API, and would like to gather feedback from scientific application users.

Acknowledgements

We would like to thank to Dr. Ikegami (AIST) and Mr. Takemiya (AIST) who gave us many comments on our work.

A part of this research was supported by a grant from the Ministry of Education, Sports, Culture, Science, and Technology (MEXT) of Japan through the NAREGI (National Research Grid Initiative) Project.

References

[1] D. Arnold, D. Bechmann, and J. Dongarra. Request Sequencing: Optimizing Communication for the Grid. *Lecture Notes in Computer Science: Proceedings of 6th International Euro-Par Conference*, 1900:1213–1222, 2000.

[2] D. Arnold and et.al. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, 2002.

[3] H. Casanova, M. Kim, J. S. Plank, and J. J. Dongarra. Adaptive Scheduling for Task Farming with Grid Middleware. *High Performance Computing Applications*, 13(3):231–240, 1999.

[4] GGF. http://www.gridforum.org/.

[5] H. Nakada, Y. Tanaka, S. Matsuoka, and S. Sekiguchi. A Task-Farming API on GridRPC and its implementation (in Japanese, abstract in English). *IPSJ SIG Technical Report*, 2003(102):61–66, 2003.

[6] Ninf project. http://ninf.apgrid.org/.

[7] Rob F. Van der Wijngaart and M. Frumkin. NAS Grid Benchmarks Version 1.0. Technical Report NAS-02-005, NASA Ames Research Center, 2002.

[8] K. Seymour and et al. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar, editor, *Proceedings of the 3rd International Workshop on Grid Computing*, pages 274–278, 2002.

[9] H. Takemiya, K. Shudo, Y. Tanaka, and S. Sekiguchi. Constructing Grid Applications Using Standard Grid Middleware. *Grid Computing*, 1:117–131, 2003.

[10] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Grid Computing*, 1(1):41–51, 2003.

[11] Y. Tanimura, T. Ikegami, H. Nakada, Y. Tanaka, and S. Sekiguchi. Implementation of Fault-Tolerant GridRPC applications. In *Workshop on Grid Applications: from Early Adapters to Mainstream Users*, 2005.