Design and Implementation of Distributed Task Sequencing on GridRPC

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

Abstract

In the framework of GridRPC, a new function that allows direct data transfer between RPC servers is implemented for efficient execution of a Task Sequencing job in a grid environment. In Task Sequencing, RPC requires dependency between input and output parameters, which means output of a previous RPC becomes the input of the next RPC. In this study, the direct transfer of data is implemented using the grid filesystem without destroying the GridRPC programming model and without changing very many parts of the existing Ninf-G implementation. Our Task Sequencing API library analyzes RPC arguments to detect intermediate data after task submissions, and reports the information to GridRPC servers so that the intermediate data is created on the grid filesystem. Through our performance evaluation on LAN and on the Japan-US grid environment, it was verified that the function achieved performance improvement in distributed Task Sequencing.

1. Introduction

GridRPC[1] is an RPC mechanism tailored for the Grid. The GridRPC offers an attractive programming model of standard RPC plus asynchronous, coarse-grained parallel tasking, while hiding the dynamic, insecure, and unstable aspects of the Grid and software complexities. There are several practical case studies of GridRPC, with scientific applications[2, 3]. In these studies, the performance and stability are analyzed if the GridRPC system meets the requirements of the applications to produce scientific discoveries. Applications reported in these studies are task parallel applications which could be implemented by a simple framework of the client-server model, i.e. multiple independent tasks were processed on distributed computing resources. However, the other types of applications whose tasks have dependency, such as workflow-type execution, are considered attractive applications for the Grid[4].

Task Sequencing function over the GridRPC is expected by applications, however there is a conflict with simple programming of GridRPC, since the client-server style does not suppose inter-server communications. Existing GridRPC implementations, such as Ninf-G[5], NetSolve[6] and OmniRPC[7], provide a functionality of Task Sequencing as an extension outside of GridRPC. Since the implementations of these systems use memory or files for storing intermediate data which would be accessed by more than two tasks, these tasks must be processed on the same node or nodes which share the filesystem. Direct data transmission between servers, which is required by the distributed Task Sequencing, are not allowed in most GridRPC implementations. One possible way is to transfer data through the client, i.e. the data is transferred from one server to the client, then the data is sent from the client to another server. This causes serious performance degradation, in particular in transferring large data of practical applications.

The goal of this study is to achieve efficient implementation of Task Sequencing over distributed computing resources, by developing a function for direct data transmission between servers, without destroying the client-server model as the GridRPC basics. In addition, the paper presents the Task Sequencing API and performance evaluation of the overall implementation of the Task Sequencing function to show the practicality.

Our approach is to implement direct data transfer by combining the GridRPC system and the Grid filesystem. Keeping the simple client-server style programming for users, this approach does not require changing many parts of the existing GridRPC systems. In this paper, Gfarm[8] is used as the Grid filesystem and Ninf-G is as the GridRPC system. First, we enhanced Ninf-G so that users specify a file on the Gfarm filesystem as a GridRPC argument. Then, we implemented the Task Sequencing API library over the Ninf-G. The library analyzes a Task Sequencing job inside the library itself, checks all file type arguments, and creates an intermediate file on the Gfarm filesystem.

2. Related work

There are several studies concerned with efficient execution of Task Sequencing. Ninf[9] and NetSolve, which are

```
Module sample
Define func1(IN filename input, OUT filename result)
Required "sample.o"
Calls "C" func1(input, result);
```

Figure 1. **A sample IDL**

```
void func1(char *input, char *result){
  FILE fp1, fp2;
  fp1 = fopen(input, "r");
  fp2 = fopen(result, "w");
    /* Do work */
  fclose(fp1);
  fclose(fp2); }
```

Figure 2. **An example of a remote program**

primarily GridRPC systems, implement an API to explicitly store a data array on a storage server. For example, the DSI (Distributed Storage Infrastructure)[10] of NetSolve allows the client to save a data array on a storage server using ns_dsi_write_matrix(), in advance. This function returns a DSI object. When the object is specified as an RPC argument, the data array on the storage server will be directly transmitted to the host where the RPC is assigned. The DSI is built on top of the IBP (Internet Backplane Protocol)[11].

NetSolve provides the Request Sequence API[12], too. RPC requests can be written between the begin function and the end function of the Request Sequence. Pointers to the data to be returned to the client node should be passed to the end function. NetSolve analyzes the input/output mode of each RPC argument inside of the client library. Thus, arguments are not transmitted to the client in vain.

Ninf-G and OmniRPC support the remote-object mechanism. Once a remote program is launched as an object, the program holds data on the memory or disk of the remote node. This allows the RPC to access the data used in the previous RPC. The remote-object is useful because sharable data is not sent back to the client needlessly.

DIET[13] proposes the Data Management API to treat data persistency on remote servers. In creating a data handle, users can set a parameter to specify the case: the data is left on the server, the data should be sent back to the client, or the data is movable to other servers. The DIET system implements Data Tree Manager (DTM), which supports direct data transmission between servers.

Our study follows the APIs discussed in the above studies and implements direct data transmission between servers. The difference from the DIET approach is that we use a Grid filesystem in the data transmission layer. Our approach makes the implementation simpler, and provides functionality for data replication in case of a fault.

3. Extension of the file transmission in Ninf-G

Ninf-G implements the file transmission with GASS[14] of Globus, and a file location (including the host name and the path) is reported by the client to the remote program.

```
[Original description]
grpc_function_handle_init(&handle, "func1", "host1");
grpc_call(&handle, "exp1/input.dat", "result.dat");

[Extended description to specify a file on the Gfarm
 filesystem]
grpc_function_handle_init(&handle, "func1", "host1");
grpc_call(&handle,
       "GFS:LTMP:GFILE:exp1/input.dat","result.dat");
```

Figure 3. **Sample client code**

This implementation assumes that the file is either on the client or on the server. If a grid filesystem is available, however, it will be acceptable to pass a universal file path to the remote program. For example, the file "test.dat" can be accessed by the path "gfarm:test.dat" from anywhere on the Gfarm environment, and the path is independent of the storage host. This allows us to not change current Ninf-G implementation very much to support direct transmission. Therefore, the present study uses Gfarm version 1.3 as a grid filesystem, and modified the file transmission code of Ninf-G so that the Gfarm file path is passed to Ninf-G as an RPC argument. In addition, our implementation allows the client to set a temporary directory on the Gfarm filesystem.

3.1. Design of the extension

Our approach is to implement a transmission function for the file stored on the Gfarm filesystem, by modifying the Ninf-G implementation as little as possible. In the conventional Ninf-G, the transmission of the file type argument is described as in Figures 1, 2 and 3. Figure 1 is an IDL (Interface Definition Language) file for the server program in Figure 2. [Original description] in Figure 3 is client code to invoke an RPC. A file transferred from the client to the compute node, is stored in the directory which is specified in the configuration file on the server. The file is named by the system and treated as a temporary file. The file name is passed to the remote function as a string argument.

In the extension using Gfarm, a Gfarm file path can be set in the GridRPC argument of the calling functions. The argument type in IDL is still a file type (filename), which does not require users to modify the remote program. The Ninf-G protocol is also the same as those of Ninf-G version 2.3 and 2.4 for compatibility.

The temporary file on the compute node is deleted after the RPC, if the remote program is not launched as a remote object. This rule is applied to the extension. An output file on the Gfarm filesystem should be taken care of by higher layers of the Ninf-G, such as the applications or the Task Sequencing library.

The file cache is handled by Gfarm. When a remote program reads a file on the Gfarm filesystem, the file is replicated on the local node. The local node must be a storage node of Gfarm in this case. Once one of replicas is updated, Gfarm automatically deletes the rest.

Table 1. File transfer patterns

No.	Input	Temporary	Output	Implementation
1	–	Local	–	No transfer
2	–	Local	Local	Implemented
3	–	Local	GFS	Case 2
4	Local	Local	–	Implemented
5	Local	Local	Local	Implemented
6	Local	Local	GFS	Case 2
7	GFS	Local	–	Case 1
8	GFS	Local	Local	Case 1
9	GFS	Local	GFS	Case 1, 2
10	–	GFS	–	No transfer
11	–	GFS	Local	Case 4
12	–	GFS	GFS	Case 6
13	Local	GFS	–	Case 3
14	Local	GFS	Local	Case 3, 4
15	Local	GFS	GFS	Case 3, 5
16	GFS	GFS	–	Case 5
17	GFS	GFS	Local	Case 4, 5
18	GFS	GFS	GFS	Case 5, 6

The extension is implemented in the remote-side library of Ninf-G. Therefore, the client library and system call hook library of Gfarm should be installed on the compute node, in order to link the Ninf-G remote executable with them.

3.2. Implementation

Table 1 shows patterns of file transmission, including conventional patterns and transmission with Gfarm. The patterns are categorized by the input, the temporary file on the remote, and the output. When the following 6 use cases are newly implemented, all of the transmission patterns are realized. In this paper, it is defined that a GFS file means a file on the Gfarm filesystem, and that an LFS file means a file on the common UNIX filesystem.

- Case 1 (Input/GFS → Temporary/Remote)
  The remote library of Ninf-G opens a GFS file by using the C API of Gfarm and copies the context to the temporary LFS file on the remote. The GFS file is specified in the GridRPC argument of the calls from the client-side.
- Case 2 (Temporary/Remote → Output/GFS)
  The remote library of Ninf-G opens a temporary LFS file and copies the context to a GFS file specified in the GridRPC call. If the remote node is a Gfarm storage node, the GFS file will be created in the spool of the Gfarm storage on the local host. This is a selection mechanism of the storage node in Gfarm.
- Case 3 (Input/Client → Temporary/GFS)
  An LFS file on the client is transmitted to a temporary GFS file, using GASS transmission which is implemented in Ninf-G, version 2.3. The physical location of the GFS file is determined according to the Gfarm storage selection mechanism, as in Case 2.
- Case 4 (Temporary/GFS → Output/Client)
  A temporary GFS file is transmitted to the client, using the GASS transmission function of Ninf-G. The physical location of the GFS file follows the Gfarm storage selection mechanism, as in Case 2.
- Case 5 (Input/GFS → Temporary/GFS)
  Because the source and destination is the same, it is not necessary to transmit the file. When the compute node is a storage node of Gfarm, however, the GFS file is replicated on the spool of the node.
- Case 6 (Temporary/GFS → Output/GFS)
  Because the source and destination is the same, it is not necessary to transmit the file. The file is not replicated in this case, which is different from Case 5.

The temporary file on the Gfarm filesystem is created in the directory "gfarm:ng/tmp/." The file path of the GridRPC argument is transformed to "/gfarm/ng/tmp/⟨filename⟩." However, this is only available when the system call hook library is installed on the compute node.

3.3. Use and limitation

In order to use the present function, the client must describe the following in the GridRPC argument.

GFS:$[LTMP|GTMP]$:$[LFILE|GFILE]$:⟨file path⟩
[:$[LFILE|GFILE]$:⟨file path⟩]

"GFS:" is a reserved word indicating that additional parameters of our extension follow. $[LTMP|GTMP]$ is a user's option specifying whether the temporary file is created on the local filesystem or on the Gfarm filesystem. $[LFILE|GFILE]$ indicates whether the next path after the colon specifies an LFS file or a GFS file. For instance, when the file "gfarm:exp1/input.dat" is transmitted to the compute node as in No.7 in Table 1, the input argument is "GFS:LTMP:GFILE:exp1/input.dat", shown in the [Extended description] part of Figure 3. The second $[LFILE|GFILE]$ is for the INOUT mode and can be omitted in other modes. In transmission pattern Nos. 6, 8, 15, and 17, the filesystem of the input file and the output file may possibly be different; one file is on the local filesystem and the other is on the Gfarm filesystem.

In the transmission patterns in Table 1, Nos. 2, 4, and 5 have already implemented in the current Ninf-G. Nos. 9 and 18 are used for a scenario where the required data is on the Gfarm filesystem. Nos. 11 and 13 are useful for the fault tolerance feature of the remote program, because the temporary file is created on the Gfarm filesystem. The other transmission patterns are used for the implementation of Task Sequencing, which is described in the next section.

4. Design and implementation of a Task Sequencing API library

4.1. Task Sequencing API

In this study, the Task Sequencing API shown in Figure 4 is implemented, referencing the APIs of Ninf and NetSolve. grpc_begin_sequence() passes the beginning of the sequence to the library. grpc_submit() is a function used to invoke GridRPC with the name of the remote function and some arguments. grpc_submit() is actually a wrapper function of grpc_call_async() which is defined in the GridRPC API standard, and it does not require the function handle as

```
grpc_begin_sequence(TMP_ON_GFS, DUPLICATION_ON);
grpc_submit("func1", A, B);
grpc_submit("func2", B, C);
grpc_end_sequence();
```

Figure 4. Task Sequencing API

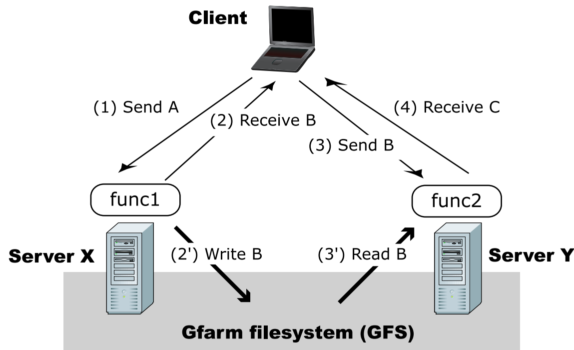

Figure 5. Intermediate data transfer using Gfarm

its argument. When grpc_end_sequence() is called, the sequence is finalized. Then the sequence is analyzed in the library and the destination of each RPC is automatically determined with optimized data logistics. In the example in Figure 4, when *func1* is executed with input data A, the output data is B. Because data B becomes the input data of *func2*, those two functions, *func1* and *func2*, are expected to be processed on the same node. When users cannot avoid running *func2* on another node, however, data B should be transmitted directly from the node which services *func1* to the node which services *func2*. This is achieved in the Task Sequencing API library we implemented with the extension in the last section. As shown in Figure 5, data B is transmitted by way of (2) and (3) with the previous Ninf-G. The extended Ninf-G can transmit data B by way of (2') and (3'), through the Gfarm filesystem. The final result C is sent back only to the client. Determination of the RPC destination and data transmission of the intermediate data are hidden in our library, so that the application program does not have to care about that level of complexity.

In addition, our API accepts two options for convenience and fault tolerance. The options are specified as the argument of grpc_begin_sequence(). When TMP_ON_LOCAL is given to the first argument, the intermediate data file is transmitted through the client. When TMP_ON_GFS is given, the intermediate file is transmitted though the Gfarm filesystem. When DUPLICATION_ON is given to the second argument, with TMP_ON_GFS to the first argument, the intermediate file is replicated on another storage node. This becomes a backup in an emergency if the storage node which stores the intermediate data becomes unavailable.

4.2. Implementation

Inside of our Task Sequencing library, the Argument Stack API provided by Ninf-G creates an argument stack

1) Allocate a resource to each task (from $Task_1$ to $Task_N$).
2) Set value i to 1.
3) Analyze arguments of $Task_i$ and $Task_{i+1}$ in the following loop.
3-1) Find the next OUT-mode argument for $Task_i$. If no more arguments, go to 3-5).
3-2) Find the next IN-mode argument for $Task_{i+1}$. If no more arguments, go to 3-1).
3-3) Compare the two arguments found at 3-1) and 3-2) by their datatypes and pointers. When those arguments indicate the same data, the data must be intermediate data which is the output of $Task_i$ and the input of $Task_{i+1}$.
3-4) Go back to 3-2).
3-5) Increment value i by 1. If value i is $N-1$ or less, go back to 3-1). Otherwise, exit the loop.

Figure 6. The process flow of the Task Sequence

when grpc_submit() is called. When grpc_end_sequence() is called, the argument stack is analyzed and modified for optimized data logistics. Then each task is sequentially executed on the assigned node.

Figure 5 shows the internal process flow of grpc_end_sequence() during execution of a Task Sequencing job that consists of N RPCs. The candidates for the execution node are selected by the function name of the first RPC request in 1), and the RPC is assigned to the top of the list. If the second RPC can be processed on the same node as the previous RPC, the second RPC is assigned to the node. If not, the second RPC is assigned to another node at the top of the candidate list. This operation is applied repeatedly.

After all of the RPC requests are assigned to appropriate nodes, two arguments next to each other are compared in order to detect the intermediate data in 3). First, the output mode argument is picked up in the proper order, from the first argument of the first RPC. The data type of the argument is compared with that of the input mode argument of the second RPC. If the data type is the same, the pointers of the arguments are also compared to conclude whether or not the arguments represent the same data, which means the argument is the intermediate data of the sequence. The intermediate data should be created on the Gfarm filesystem, according to the given parameter of grpc_begin_sequence().

Our implementation supports Task Sequencing for the file type argument of GridRPC. In the example in Figure 4, the output file of *func1* and the input file of *func2* are the same GFS file. The transmission patterns used for Task Sequencing are categorized by the location of the temporary file and the input/output mode of *func1* or *func2*. In this study, our library is implemented using a set made up of Nos. 3 and 7, and a set made up of Nos. 6 and 8. The Task Sequencing library internally modifies user's file path parameter to the GFS file path, in order to create the intermediate file on the Gfarm filesystem in execution. After result C is sent back to the client, the library deletes the intermediate GFS file.

Table 2. Computational environment

Site (role)	CPU	OS kernel	Globus (flavor)	I/O for Local FS		I/O for Gfarm FS	
				Read	Write	Read	Write
AIST (client)	PentiumIII 1.4GHz × 2	Linux 2.4.20	2.4.3 (gcc32)	144 (46.0)	33.3	–	–
AIST (server)	Xeon 2.8GHz ×2	Linux 2.6.9	3.2.1 (gcc32)	122 (98.2)	112	72.6	92.7
NCSA (server)	Xeon 2.0GHz × 4	Linux 2.4.21	2.4.3 (gcc32pthr)	991 (71.8)	22.5	91.9	21.8

I/O performance unit: Mbytes/sec

Table 3. Network performance

From	To		
	AIST (client)	AIST (server)	NCSA (server)
AIST (client)	–	59.59 [MB/sec]	Not measured
AIST (server)	51.46 [MB/sec]	86.13	Not measured
NCSA (server)	0.19	0.35	82.39

Table 4. Performance of data transfer when both functions are serviced at the AIST site

Transfer type	10 MB data		50 MB data	
	(Total)−(Func)	(2)+(3)	(Total)−(Func)	(2)+(3)
Gfarm	2.10 [sec]	0.68 [sec]	7.08 [sec]	2.09 [sec]
GASS	2.51	1.10	16.0	8.41
Protocol	1.10	0.470	4.45	1.99
Remote object	1.33	0.00955	4.71	0.0467

Table 5. Performance of data transfer when both functions are serviced at the NCSA site

Transfer type	10 MB data		50 MB data	
	(Total)−(Func)	(2)+(3)	(Total)−(Func)	(2)+(3)
Gfarm	127 [sec]	4.26 [sec]	601 [sec]	9.07 [sec]
GASS	233	107	1151	577
Protocol	274	128	1362	680
Remote object	115	0.0316	585	0.114

Table 6. Performance of data transfer when func1 is serviced at the NCSA site and func2 is serviced at the AIST site

Transfer type	10 MB data		50 MB data	
	(Total)−(Func)	(2)+(3)	(Total)−(Func)	(2)+(3)
Gfarm	111 [sec]	23.7 [sec]	497 [sec]	64.6 [sec]
GASS	116	28.6	592	138
Protocol	150	54.1	691	260

5. Evaluation

Our Task Sequencing API library over a Ninf-G extension using Gfarm was evaluated on the testbed, in terms of the usefulness of direct data transmission. The testbed consisted of one client node and two servers at AIST, and two servers at NCSA (National Center for Supercomputing Applications). The specifications of those machines are shown in Table 2. The I/O columns display read and write performance on the local filesystem and on the Gfarm filesystem. In the I/O measurement for Gfarm, a benchmark program read and wrote data through the Gfarm spool on the local host. The buffer size was set as 8 KB. 1 GB of data was written into a file and read from the file. Because a cache effect was observed in the local access, the raw speed of the disk access is displayed in the parenthesis. The AIST (server) uses a RAID system and NCSA (server) accesses the file on a RAID system via NFS. A Gfarm metadata server runs on a node at the AIST site.

Table 3 shows TCP throughput between servers, the value of which is the average of the 1-minute-measurement by Netperf, at 3 different moments. The route from AIST to NCSA was not measured due to firewall restrictions.

5.1. Experiments

In the experiments, *func1* and *func2* are serviced on remote, but different, nodes. Each function requires two file type arguments, input and output. The submitted Task Sequencing job is to execute *func1* and then *func2*. The Ninf-G client ran on AIST throughout our entire experiments. Table 4 shows the experiment results when both the *func1* and *func2* serviced nodes were at AIST. Table 5 shows the experiment results when both functions were serviced at NCSA nodes. Table 6 shows the experiment results when *func1* was serviced on the NCSA node, and *func2* was serviced on the AIST node. The size of the input file and the output file are the same in the experiment, and we had two cases, one each of a 10 MB file and a 50 MB file. In the tables, the value (Total)-(Func) is obtained when the execution time of *func1* and *func2* is subtracted from the total execution time that is measured from the time grpc_begin_sequence() starts, to the time grpc_end_sequence() ends. The value (2)+(3) is the time of the intermediate data transmission, which is shown as via (2) and (3), or via (2') and (3') in Figure 5.

In the results in Tables 4 $\sim$ 6, three file transmission methods are compared in terms of performance. Gfarm in the table represents the transmission through the Gfarm filesystem. Shared key authentication is used to access the storage node when both the client and the server are at AIST. GSI authentication is used in the case where both the client and the server are at NCSA, and in the case where the client gets access to a storage server on another site. We launched gfarm_agent on each cluster to improve the access speed to the metadata server. GASS in the table represents GASS transmission integrated into Ninf-G. The HTTP protocol was used in the experiments. Protocol represents the file transmission over the Ninf-G protocol which has been implemented with Globus XIO since Ninf-G version 2.4. GASS and Protocol always transfer data from one server to another server, through the client. Remote-object represents an implementation using the remote-object function of Ninf-G. In this case, *func1* and *func2* are implemented in a single program and serviced on a single host. The intermediate data is just stored on the node.

5.2. Results and Discussions

The Gfarm transmission was faster than the GASS transmission but slower than the Protocol transmission, as shown in Table 4. There are two main reasons for this result. Authentication at the storage server costs 0.2 seconds on average. Also, access to the metadata server generates overhead.

In Table 5, the transmission cost through the client was obviously large. Task Sequencing using Gfarm was processed in half the time of Task Sequencing using GASS. The results in Table 6 show the performance difference between Gfarm transmission and GASS transmission in the case where data is transmitted over a wide-area network. Protocol transmission was fast when the client and the servers were at AIST. However, Protocol transmission was much slower than GASS transmission because Protocol does not configure a large socket buffer for the wide-area network as GASS does.

The performance of Remote-object is shown at the bottom of Table 4. This is the case where the cost of intermediate data transmission is almost zero. The execution time of Remote-object is longer than Protocol's, however, because the data transmission of input and output are performed by GASS transmission.

It is revealed that the proposed Task Sequencing API library performs efficient execution by taking advantage of Gfarm's capability. Users can benefit from more time reductions when the data transmission cost is large on cross-sites. In the intra-site, the execution time of Gfarm transmission is shorter than GASS transmission of the conventional Ninf-G. Although Protocol transmission is faster than Gfarm's, users benefit from a backup function of the intermediate file by Gfarm. If disk space on the client is very limited, the proposed library will be useful because the intermediate data is not necessarily sent back to the client. Another advantage of the Task Sequencing API library is that users do not have to worry about the physical location of the intermediate data and the data transmission, but can concentrate on application development.

6. Conclusion

Direct data transmission between servers was designed and implemented on the GridRPC system. The Task Sequencing API was proposed for describing a sequence job which consists of several tasks. Our implementation does not require a lot of modification into Ninf-G. The file type argument of Ninf-G is simply extended to accept the Gfarm file path. Reading, writing and copying data on the Gfarm filesystem is completely hidden in the remote-side library of Ninf-G. The Task Sequencing API library is implemented on top of the extension, and evaluated on the grid testbed between Japan and the United States. The results indicate that direct data transfer is significantly effective in executing a Task Sequencing job, on a wide-area network.

Future works are to implement the direct transmission of the non-file type argument, and to improve the task assignment algorithm in the evaluation with practical applications.

Acknowledgements

We would like to thank Dr. Tatebe (Tsukuba University), who gave us many comments on our work.

We are grateful for resource contributions by the National Center for Supercomputing Applications.

References

[1] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar, editor, *Proceedings of the 3rd International Workshop on Grid Computing*, pp. 274–278, 2002.

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

[3] Y. Tanimura, T. Ikegami, H. Nakada, Y. Tanaka, and S. Sekiguchi. Implementation of Fault-Tolerant GridRPC Applications. In *GFD-I.68 (Workshop on Grid Applications: From Early Adopters to Mainstream Users)*, pp. 38–49. Global Grid Forum, 2006.

[4] A. Mayer, S. McGough, N. Furmento, W. Lee, M. Gulamali, S. Newhouse, and J. Darlington. Workflow Expression: Comparison of Spatial and Temporal Approaches. In *Workflow in Grid Systems Workshop, GGF-10*, 2004.

[5] Y. Tanaka and et al. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Grid Computing*, Vol. 1, No. 1, pp. 41–51, 2003.

[6] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, 2002.

[7] M. Sato, T. Boku, and D. Takahashi. OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment. In *Proceedings of CCGrid 2003*, pp. 206–213, 2003.

[8] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *Proceedings of CCGrid 2002*, pp. 102–110, 2002.

[9] H. Nakada and et al. Design and Implementation of Ninf: toward a Global Computing Infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, Vol. 15, pp. 649–658, 1999.

[10] D. Arnold, S. Vadhiyar, and J. Dongarra. On The Convergence of Computational and Data Grids. *Parallel Processing Letters*, Vol. 11, No. 2–3, pp. 187–202, 9 2001.

[11] J. S. Plank, A. Bassi, M. Beck, T. Moore, M. Swany, and R. Wolski. Managing Data Storage in the Network. *IEEE Internet Computing*, Vol. 5, No. 5, pp. 50–58, 2001.

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

[13] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. Technical Report RR-5601, The French National Institute for Research in Computer Science and Control, 2005.

[14] I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.