A Non-Intrusive and Incremental Approach to Enabling Direct Communications in RPC-based Grid Programming Systems

University College Dublin, School of Computer Science and Informatics
Technical Report UCD-CSI-2005-2

Alexey Lastovetsky, Xin Zuo, Peng Zhao

Abstract

This paper advocates a non-intrusive and incremental approach to enabling existing Grid programming systems with new features. In particular, it presents a software component enabling NetSolve applications with direct communications between remote tasks. The software component is a supplementary one working on the top of the basic NetSolve system. Its design also allows remote tasks to be freely mixed in a single application, independent on whether each particular task is enabled for direct communications or not. Experiments with this software are also presented.

1. Introduction

High performance Grid programming systems have reached a certain level of maturity. Two examples are NetSolve [1-3] and Ninf [4] that allow scientific programmers to develop reliable Grid applications. The systems are quite easy to install and use. They also demonstrate high level of stability and reliability achieved over years of testing and maintenance.

On the other hand, the constantly growing number of users and applications results in the need of further development of such systems in terms of functionality and quality. Traditionally, addition of a new feature to a Grid programming system is achieved by changing the code of the system and producing its new version. This new version of the system has to replace the previous one in order to enable Grid applications with the new feature.

This approach to the evolution of Grid programming systems has two serious disadvantages. First of all, the change of the system's code may introduce bugs resulting in the situation when some applications, which have been developed, tested and successfully executed with the previous version of the system, will not run properly with the new one. It can take significant time and efforts to fix all the newly introduced bugs and make all broken applications run in the new environment in the same way they did in the old one.

Secondly, the new version of the system has to replace the old version on *all* computers of the Grid in order to support the development and execution of applications enabled with the new feature. Such simultaneous and total replacement can have very high organizational overhead and sometimes be simply unrealistic as different computers on the Grid are managed and administered by independent and, very often, loosely connected users.

Thus, the traditional approach to enabling the existing Grid programming system with a new feature is:

- *Intrusive*, that is, the code of the

system is to be changed in order to add the feature;

- *Non-incremental*, that is, to make the system functional with the new feature, the modified system has to be installed on all the computers that are supposed to participate in the execution of applications.

The goal of our research is to investigate if an existing Grid programming system can be enabled with new features in a non-intrusive and incremental way.

Non-intrusiveness means that the original system does not change and the new features are provided by a supplementary software component working on the top of the system. Correspondingly, all applications not requiring those new features will only use the basic original software and be developed and executed in the same way both in the original and modified systems.

Increment means that the supplementary software component does not have to be installed on all computers to enable applications with the new features. It can be done incrementally, step by step, and the new features will be enabled in part, with the completeness dependent on how many nodes participating in the execution of the application have been upgraded with the supplementary software component.

In this paper, we use one particular well-established Grid programming system and one particular feature, the necessity of which is well recognized, to demonstrate the feasibility of the non-intrusive and incremental evolution of Grid programming systems. The system is NetSolve, and the feature is direct communication between remote tasks.

The rest of the paper is structured as follows. Section 2 describes in detail the design and implementation of a supplementary software component enabling NetSolve applications with direct communications between remote tasks in a non-intrusive and incremental way. Section 3 presents some experiments with this software. Section 4 outlines related work, and Section 5 concludes the paper.

2. Enabling direct communications in NetSolve

NetSolve is positioned as a programming system for high performance distributed computing on global networks based on GridRPC [5]. It deals with the situation when some components of the application cannot be provided by the user and are only available on remote computers. To program a NetSolve application, the user writes a NetSolve client program, which is any program (say, in C or Fortran) with calls to the NetSolve client interface. Each call specifies the name of the remote task to be performed, pointers to the data on the user's computer required by the task, and pointers to locations on the user's computer where the results will be stored. When the program runs, a NetSolve call will result in a task to be executed on a remote computer. The NetSolve programming system is responsible for selection of the remote computer to perform the task, transferring input data from the user's computer to the remote computer, and delivering output data from the remote computer to the user's one.

Thus, in NetSolve output data of remote tasks are typically sent back to the client upon completion of each remote task even if the data are only needed as input for some other remote tasks, resulting in so-called bride communications when data between remote tasks are sent through the client machine. Such unnecessary bridge communications can significantly increase

the execution time of NetSolve applications. Therefore, the ability of remote tasks to communicate directly is a desirable feature of any RPC-based Grid programming system, including NetSolve, which leads to higher performance of its applications.

One approach to enabling NetSolve with direct communications between remote tasks is presented in [6]. It is based on some changes made in the original NetSolve code to provide the new functionality.

In this paper, we propose a lightweight supplementary software component that enables direct communication between remote tasks in NetSolve in a non-intrusive and incremental way. The main features of this component are as follows:

1. The component is built on top of NetSolve rather than built into the system.
2. To enable direct communications, the software component does *NOT* need recompilation or reinstallation of the NetSolve programming system.
3. It uses the existing programming infrastructure and is functioning non-intrusively on the top of the existing functionality.
4. The software component does not have to be installed on all nodes simultaneously. It can be used to enable direct communications between remote tasks incrementally. It allows for remote calls both to tasks enabled for direct communications and to tasks not enabled, within the framework of the same application. Naturally, direct communications are only possible between enabled tasks.

2.1 Overview

We start presenting the software component by a short description of its use.

- **Client side**: The only thing for client programmers to do is to install the wrapper API and Job Name Service on the client side, then compile the client program with the wrapper library. The wrapper API allows the programmers to explicitly specify the dataflow between remote tasks. So they only need to slightly modify their client code. The principle is quite easy: the programmer just replaces the input/output arguments with handlers as the input/output data.
- **Procedure developers**: The procedure programmers should do nothing to enable direct communications. They develop their own procedures as usual. The supplementary software component has no effect on both existing procedures and newly added procedures.
- **Server administrator**: To enable direct communication control on server side, the server administrator needs to register the software component as a new problem file to NetSolve. No re-installation and re-compilation of NetSolve itself are needed.

2.2 A Client-side example

Before we start describing the model and implementation, let us take a look at an example of the use of the software component. In the example, we want two tasks, **A** and **B**, to be performed on remote nodes. The output of task **A** is the input of task **B**. Normally, the NetSolve client code would be written like:

```
errno=netsl("A",inputA,outputA);
errno=netsl("B",outputA,inputB,outputB);
```

The default manner of my*netsl* leaves no options for the programmer to control the dataflow. So the output of procedure **A** has to be sent back to the client machine and stored there in *outputA*, and then

transferred together with *inputB* to **B**. This causes unnecessary communications. To allow the programmer to explicitly specify the optimal dataflow, we extend the API as follows:

mynetsl("A",inputA,hdlA);
mynetsl("B",hdlA,inputB,outputB);

In this modified version, the output of procedure **A** is represented by handler *hdlA*. However the data are still stored on the remote server. Upon invocation of **B**, *hdlA* replaces *outputA* as **B**'s input argument. The use of *hdlA* tells procedure **B** where it can get its input. The advantage of this approach is that it does not change the program's semantics: **A** and **B** are called with the blocking routine, unlike *REDGRID* [6], where one of these two jobs must be altered to use the nonblocking remote procedure call *netslnb*.

The other modification in the above example is the use of *hdlA* instead of *outputA*. This tells procedure **A** not to return its result to the client, but leave it on the remote server for someone to receive. Another option here is to remotely call **A** with the target's handler. In this case, **A** will directly send its output to that job.

Thus, programmers are given the ability to explicitly specify the data flow in their code. Although it may slightly impact the RPC transparency, this will not increase the programming difficulty. Very slight changes to the existing client programs are required to gain the benefit of direct communications.

2.3 Architecture of the software component

As we have described in the introduction, our approach to enabling direct communications is via a software component supplementary to the existing RPC-based Grid programming system, which does *NOT* need recompilation or reinstallation of the original Grid programming system. The proposed software component consists of three parts: *Client API & Argument Parser, Server Connector and Job Name Service (JNS).* Figure 1 depicts the architecture of the software component.

To use the direct communication component, a client NetSolve programmer needs to install the provided library on the client side, use Client API to write his/her own client program and compile the program with the library. To enable direct communications for existing NetSolve client programs, very slight changes are needed to be made as that has been described in Section 2.2 (the changes mainly include the replacement of real data references by handlers). Wrapper functions implementing Client API use calls to the Argument Parser to parse the list of arguments and generate communication information for each argument. The communication information is then passed to the Server Connector, which uses it when enabling direct communications between remote servers.

Enabling direct communication on the server side only includes registration of the Server Connector as yet another NetSolve problem. The Server Connector is mainly responsible for receiving input data from and sending output data to other servers. It re-submits the task to the local server after all the necessary data are successfully received.

JNS is set up on the client side automatically before client program is executed to submit tasks to NetSolve server. It contains all information about every handler. Handler registration and access are done during the execution of the wrapper functions implementing the Client API on the client side. There is no communication between JNS and servers or between JNS and Server Connector.

2.4 Client API and Argument Parser

Client API provides a uniform interface for the client to make remote procedure calls. Despite the modification on the remote side, the wrapper API allows the calls to be made in the same manner. The only difference is in the arguments. Like in NetSolve, we parse the list of argument to construct the handler array.

For each argument, the relevant *communication information* is generated. For each input argument, which is a variable storing real data, the local IP address and the port number are used as such communication info. If this input argument is a handler, then a request is sent to the JNS to get the IP address and the port number of the remote resource and this information is used as communication info for this handler.

For each output argument, which is a variable storing real data, the client wrapper function will set up a socket to download output data from computational servers. If this output argument is a handler, the returned result information from computational servers is sent to JNS and registered there. So, in the future, other computational tasks can require the data source information from JNS and use the obtained information to get real data.

A handler contains the data source/target's IP address and the port number, which will be used to send/receive data. In this sense, upon making a call to NetSolve, this is actually only a handler array which is transferred to the remote server. All the other I/O data transfer is managed by the Server Connector. The pseudo code for our wrapper for *mynetsl()* is in Appendix A.

In the wrapper function for *mynetsl()*, if the client cannot find any server, which has both Server Connector and the requested remote procedure, it will still run properly by using calls to the original *netsl()* functions. In particular, the data transfer between the client and the server is performed with help of the client-side JNS. The algorithm of selection of the fastest server among all available servers is the smae as the one implemented in the NetSolve agent program.

2.5 Servers-side Connector

On the server side, a proxy program called Server Connector is responsible for interacting with clients and other Server Connectors to enable direct communications. The Server Connector has two main functions. The first one is to pass handler information between clients and servers. This allows servers to know how to get the data without bridge communication.

The second function is the extraction of the handlers' information and using it to download needed data through direct communication. After all needed data have been acquired, the Server Connector calls the procedure to re-submit to the local host to perform computations that the user exactly requested for.

There is no difference in the way the client and computational servers download the result of the computations. The Server Connector firstly returns the result's communication information to the client. Then it sets up a socket waiting for the client or the server to connect in to download the result of computations. The pseudo code for *connector()* is in Appendix B.

2.6 Job Name Service

In the example given in Section 2.2, procedure **B** is given a handler to locate **A**. However, the handler is only meaningful on the client side. It contains no information about the job's network address and communication port number. Therefore, we introduce an external name service to register/search remote jobs.

Any procedure registers itself on a dedicated *Job Name Service* (JNS) upon its invocation. Other procedures may send requests to the JNS to search for this registered procedure. JNS is set up on the client side automatically. During the execution of the application, it contains all information about every handler. Only client has the permission to register or access a handler on the JNS. There is no communication and interaction between JNS and computational servers.

Handler publication is made by calling *jobPublish(Handler, dataInfo)*, and *jobQuery(Handler, dataInfo)* is used for searching. In the prototype version of the system, we use the following format to label a specific job:

<jobAddr>
Handler= "hdlA"
dataInfo[0] = "csa004b3pc2.ucd.ie" // ip
dataInfo[1] = 2919 // port number
dataInfo[2] = 100 // matrix size
dataInfo[3] = 2 // requested times
dataInfo[4] = 0 // broadcast type
</jobAddr>

In this example, *Handler* contains the name of the handler used in the function prototype. The array *dataInfo* specifies the data's location, data's format details and transaction mode. This information allows the job to be uniquely identified in the network. Different jobs can use the JNS to publish themselves, search others, and exchange data. Also, the JNS is designed as a system-independent system on the client side, so that it can be applied to different RPC-based systems and not influenced by any fault or crash on the server side.

3. Implementation and experiments

3.1 Implementation

Currently, we have a prototype implementation of the software component. Figure 2 depicts how this component works. To connect **A**'s output with **B**'s input, two calls of wrapper API are made. **A** registers the contact address on the JNS, where **B** gets this information. Then these two jobs set up a connection and pass through it the intermediate results.

Note that despite NetSolve was assumed as the enabled system, nothing in the implementation is specific for this particular system. This makes the proposed approach applicable to other Grid RPC systems.

Since the inter-job communication is provided in the form of external function, it is possible for the client to connect calls of different Grid RPC systems (for example, feeding the input of a Ninf call with the output of a NetSolve call).

3.2 Experiments

For our experiments we choose the same remote computational task that has been used in experiments with REDGRID presented in [6], namely, matrix multiplication. Experiments in [6] used 2 remote servers to perform 3 matrix multiplications, and the client, agent and servers all were in the same Ethernet segment.

In our experiments, we used 8 remote servers to perform 8 matrix multiplications. The interconnecting network is based on 100 Mbit Ethernet with a switch enabling parallel communications between computers. Specification details of computational nodes are given in Table 1.

Table 1. Installation and specifications of computational nodes

Name	Architecture	Cpu Mhz	Main Memory (mb)	Cache (kb)
Pg1cluster01	Linux 2.6.8 - 1.521 smp Intel(R) EON™	2048	1024	512
Pg1cluster02	Linux 2.6.8 - 1.521 smp Intel(R) EON™	2048	1024	512
Pg1cluster03	Linux 2.6.8 - 1.521 smp Intel(R) EON™	2048	1024	512
Csultra01	SunOS 5.8 sun4u sparc SUNW, Ultra-5_10	440	512	2048
Csultra02	SunOS 5.8 sun4u sparc SUNW, Ultra-5_10	440	512	2048
Csultra03	SunOS 5.8 sun4u sparc SUNW, Ultra-5_10	440	512	2048
Csultra04	SunOS 5.8 sun4u sparc SUNW, Ultra-5_10	440	512	2048
Csultra05	SunOS 5.8 sun4u sparc SUNW, Ultra-5_10	440	512	2048

The client code *WITH* bridge communications looks as follows:

```
/* Compute matrix multiplications */
mynetsl("matmul()", matA, matB, matC, n);
mynetsl("matmul ()", matC, matD, matE, n);
mynetsl("matmul ()", matE, matF, matG, n);
mynetsl("matmul ()", matG, matH, matI, n);
mynetsl("matmul ()", matI, matJ, matK, n);
mynetsl("matmul ()", matK, matL, matM, n);
mynetsl("matmul ()", matM, matN, matO, n);
mynetsl("matmul ()", matO, matP, matQ, n);
```

The client code with direct communications is as follows:

```
/* Compute matrix multiplications */
mynetsl("matmul()", matA, matB, hdlC, n);
mynetsl("matmul ()",hdlC, matD, hdlE, n);
mynetsl("matmul ()",hdlE, matF, hdlG, n);
mynetsl("matmul ()",hdlG, matH, hdlI, n);
mynetsl("matmul ()",hdlI, matJ, hdlK, n);
mynetsl("matmul ()",hdlK, matL, hdlM, n);
mynetsl("matmul ()",hdlM, matN, hdlO, n);
mynetsl("matmul ()",hdlO, matP, matQ, n);
```

Parameter *n* is the dimension of matrices. *matA*, *matB*, *matC*, *matD*, *matE*, *matF*, *matG*, *matH*, *matI*, *matJ matK*, *matL*, *matM*, *matN*, *matO*, *matP* and *matQ* are matrix data. h*dlC*, *hdlE*, *hdlG*, *hdlI*, *hdK*, *hdlM* and *hdlO* are handlers, which are used to eliminate bridge communication. In the experiments, we only measure the communication time of trails.

We select 3 trails for each matrix size. Experiment results are presented in Table 2. The average execution time of the two applications (with bridge and direct communications) is calculated for each set of trails.

Figure 3 shows the communication time as a function of matrix size. Figure 4 shows the speedup of the application with direct communications over the one with bridge communications. As expected, the communication cost is visibly reduced

Table 2 Comparison of different communication Approaches (bridge and direct)

Size	Trail 1		Trail 2		Trail 3		Average		Speedup
	B	D	B	D	B	D	B	D	
1000	38.3	28.7	39.5	29.2	38.6	29.1	38.8	29	25.2%
2000	155.5	115.7	151.2	113	153.4	110	153.4	112.9	26.4%
3000	342.9	238	345	255	340.8	260	342.9	251	26.8%
4000	607	428	604	436	611	450	607	438	27.8%
5000	920	691	923	671	908	636	917	666	27.4%
6000	1354	901	1379	1005	1402	1094	1378	1000	27.4%
7000	1840	1391	1810	1392	1895	1321	1848	1368	26.0%
8000	2460	1773	2395	1810	2453	1853	2436	1812	25.6%
9000	3069	2349	3095	2298	3023	2205	3062	2284	25.4%
10000	3563	2670	3810	2894	3750	2845	3708	2803	24.4%

B - Bridge Communication, D - Direct Communication; Time(second)

Table3. Speedup for different ratios of eliminated bridge communications

Ratio of Bridge Communication Cut (Theoretical Speedup)		3/12 (25.0%)	4/15 (26.7%)	5/18 (27.8%)	6/21 (28.6%)	7/24 (29.2%)	8/27 (29.6%)	9/30 (30.0%)	10/33 (30.3%)
Average Value	B	1992	2502	3007	3518	4037	4546	5061	5581
	D	1550	1907	2255	2620	2963	3331	3702	4043
Speedup		22.2%	23.8%	25.0%	25.5%	26.6%	26.7%	26.9%	27.6%

B - Bridge Communication, D - Direct Communication; Time(second)

by using direct communications. In the experiments, seven communication bridges were eliminated among twenty four communications. So, the theoretical speedup is 7/24 = 29.2%. The obtained experimental speedup ranges from 24% to 27%, which is close to the theoretical value. We can also see that the experimental results are similar to the REDGRID ones, which range from 18% to 28%.

The speedup depends on the ratio of the number of eliminated bridge communications and the total number of communications. Table 3 shows speedups obtained for various ratios for the same matrix size, 10000. The result of experiments shows that the speedup due to elimination of bridge communications increases with the increase of the ratio. If communication links between the computers are of the same bandwidth, the upper bound on the speedup is as follows:

$$\lim_{n->\infty}(\frac{n-1}{n\times 3}) = 1/3 = 33.3\%$$

If communication links connecting remote computers are much faster than communication links connecting the remote computers and the client computer, the speedup due to elimination of bridge communications will be much higher. To corroborate it, we design another experiment. We manually make all bridge communications be performed at the rate of 10 Mbit per second. For the direct communications between remote servers, we still use 100 Mbit Ethernet interconnecting network. Figure 5 shows the communication time for this configuration of the communication network. Figure 6 presents the speedup of the application with direct communications over the one with bridge communications in this case. The experimental speedup is around 54% when the ratio of eliminated bridge communications is 2/9. Thus, much higher speedup can be achieved in heterogeneous communication networks, which are typical for real-life Grid environments,

than in artificially designed homogeneous ones.

4. Related work

To enable direct communications, NetSolve introduces an original mechanism called Request Sequencing [7]. The mechanism imposes a number of restrictions on the sequence of remotely called tasks, the most restrictive of which is that all the tasks have to be performed on the same computing node. Another effort to reduce the overhead of bridge communications in NetSolve is the Logistical Computing and Internetworking (LoCI) [8]. LoCI provides facility to schedule the data storage at a place 'close' to the receiver. The mechanism is mainly aimed at replicating data in order to keep them even in the case of crash of some of the computers. Although it is sufficient for enabling direct communications, the goal of building a complete network storage system makes LoCI over-heavy for enabling just this particular feature.

The REDGRID project [6] is closest to our approach sharing the similar idea behind its design. The main difference is that REDGRID is built into NetSolve and difficult to be migrated to other GridRPC-based systems. The REDGRID project uses an intrusive and non-incremental approach and requires re-compilation and re-installation of the modified NetSolve on all involved computing nodes to enable direct communication. Also the REDGRID's design is not extendable and relies on the NetSolve architecture. A certain amount of work is needed to port REDGRID to other GridRPC-based systems.

5. Conclusion

In this paper, we have presented an approach to reducing unnecessary bridge communications in RPC-based Grid programming systems. The main advantage of the approach is that it is non-intrusive, requiring no changes in the enabled programming system. It does *NOT* need recompilation or reinstallation of the Grid programming system. The approach is incremental by nature allowing remote tasks both enabled for direct communication and not, to be freely mixed in a single application. It can be applied to different RPC-based Grid programming systems. Finally the experimental results have shown that the performance of Grid applications can be significantly improved by using our supplementary software component.

References

[1] http://icl.cs.utkedu/netsolve/

[2] H.Casanova, J.Dongarra. "NetSolve: A Network Server for Solving Computational Science Problems." The International Journal of Supercomputer Applications and High Performance Computing, Volume 11, Number 3, pp. 212-223, 1997.

[3] D. Arnold, H. Casanova, J. Dongarra. "Innovation of the NetSolve Grid Computing System." Concurrency: Practice and Experience, Volume 14, numbers 13-15, pp. 1457-1479, 2002.

[4] Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., 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] Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., Casanova, H.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing, Proceedings of the Third International Workshop on Grid Computing, 2002, 3-

540-00133-6, 274–278, Springer-Verlag

[6] Desprez, F., Jeannot, E.: Improving the gridrpc model with data persistence and redistribution. In: International Symposium on Parallel and Distributed Computing in association with HeteroPar (2004) 193–200.

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

[8] Beck, M., Arnold, D., Bassi, A., Berman, F., Casanova, H., Dongarra, J., Moore, T., Obertelli, G., Plank, J., Swany, M., Vadhiyar, S., Wolski, R.: Middleware for the use of storage in communication. Parallel Computing Volume 28, Issue 12 (December 2002)

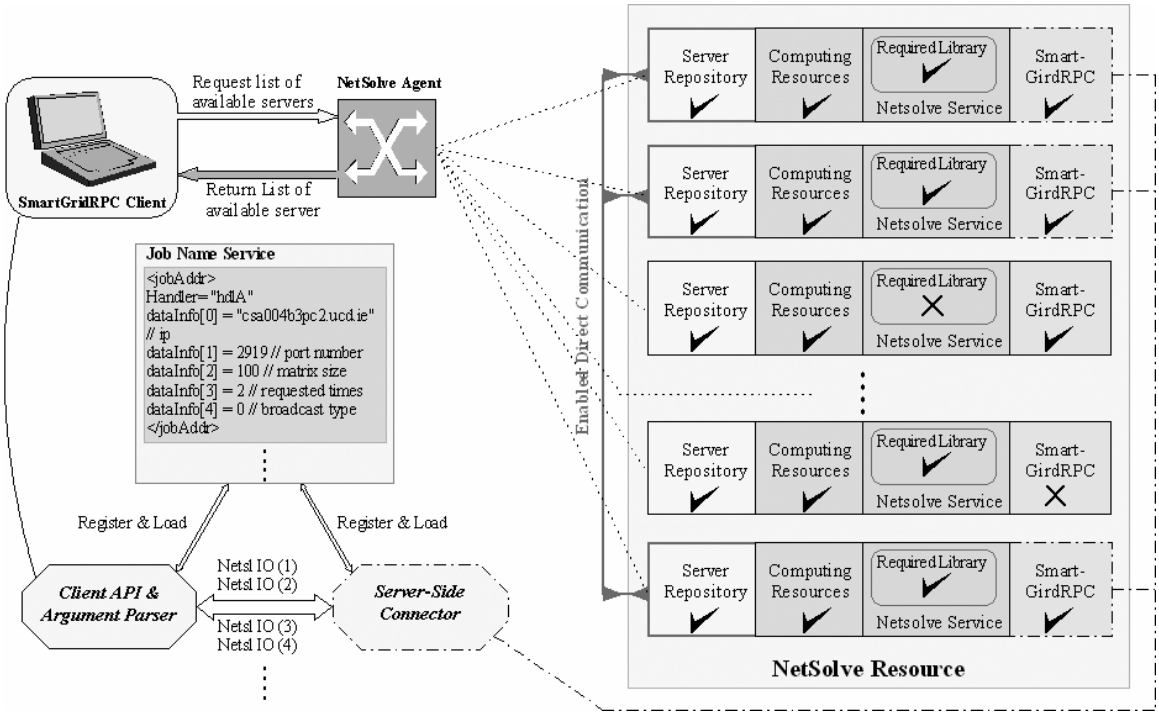


Figure 1: Architecture of the supplementary software component enabling direct communications in NetSolve applications

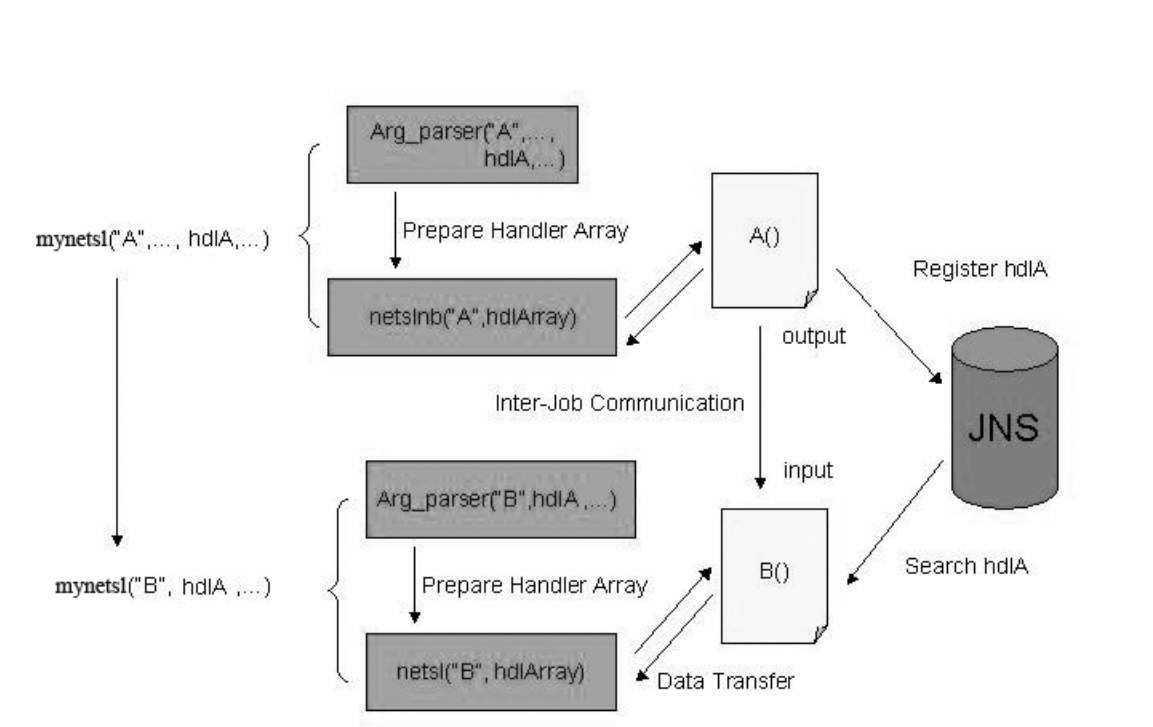
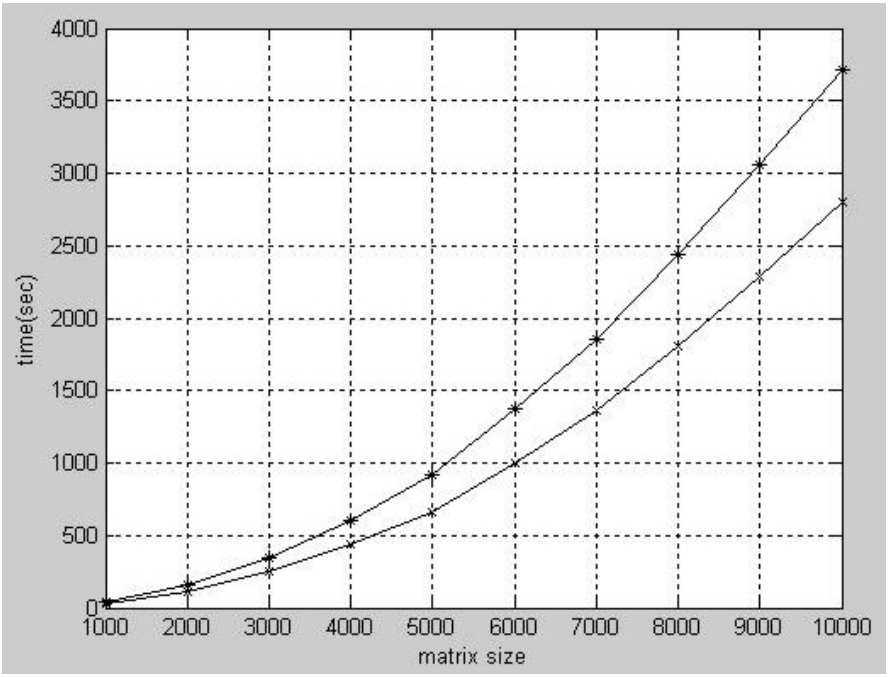
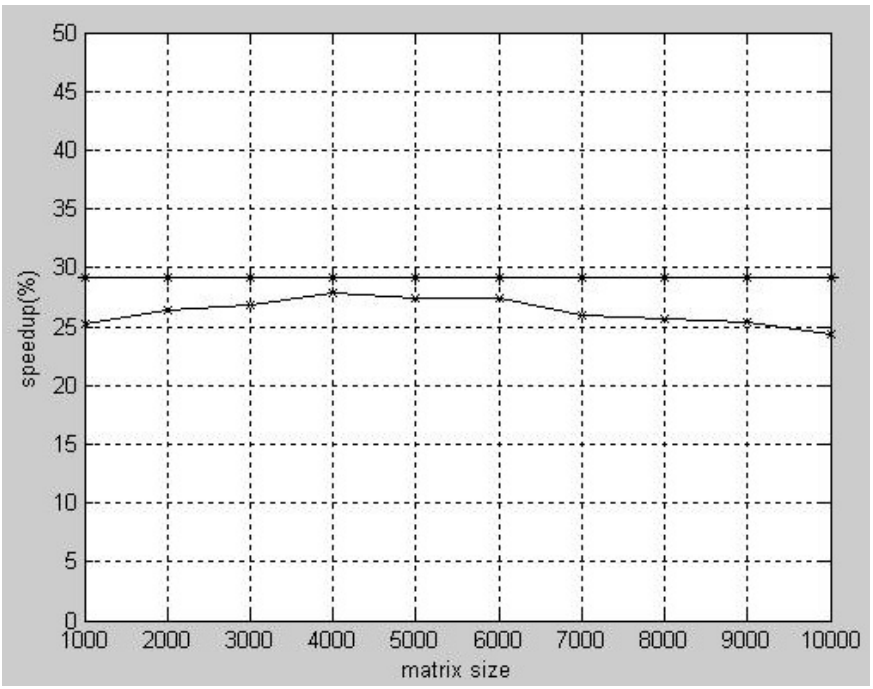


Figure 2: Implementation of transferring the output of procedure A to procedure B as its input



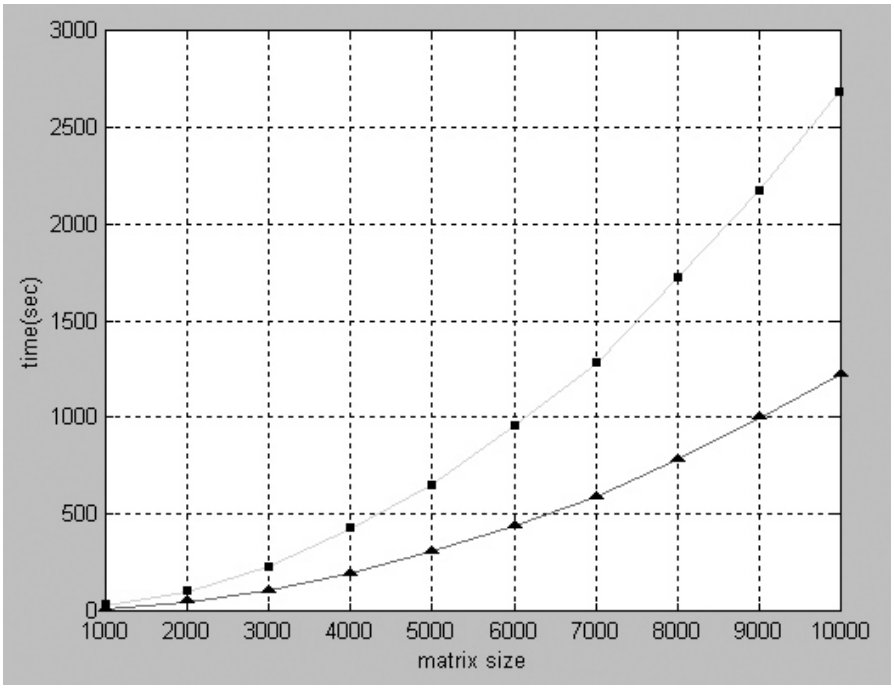
*** bridge communication x direct communication**

Figure 3: Time elapsed for both communication types when all communication links have the same bandwidth, 100Mb per sec.



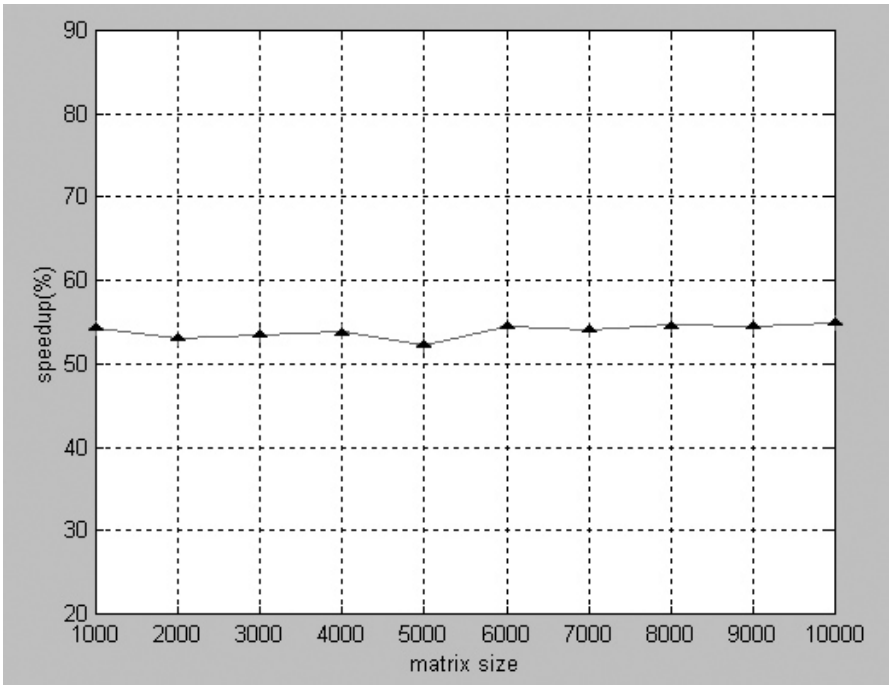
*** theoretical speedup x experimental speedup**

Figure 4: Speedup due to the use of direct communications for the homogeneous communication network.



■ bridge communication ▲ direct communication

Figure 5: Time elapsed for both communication types when communication client and servers is at the rate of 10 Mb per sec, and communication between servers is at the rate of 100 Mbit per sec.



▲experimental speedup

Figure 6: Speedup due to the use of direct communications for the heterogeneous communication network.

Appendix A. Pseudo code of the wrapper for my*netsl()*.

```
int mynetsl(ProcName,ArgList) {
// get list of Netsolve servers
server_list = my_NS_config();
for (i=0;i<number_of_servers;i++) {
  server_info[i]=get_info(servers_list);
}

// get list of problems for each server
for (i=0;i<number_of_servers;i++) {
   prob_list[i]=my_NS_problmes(server_info[i]);
}

// select servers which have registered both
// Connector and the problem "ProcName"
servers_available= myselect(server_info,
prob_list);

// if there is at least one server which has
// registered both problem and Connector
if (servers_available != NULL)
  {
   // select fastest server from available servers
   server_best=select_fastest(servers_available);

   // generate communication information
   // for each argument in Arglist by parsing
   // the arguments
   for (i=0;i<num_input_arg;i++) {
   if LOCAL RESOURCE {
    // allocate local IP and port number
      local ip and port -> ArgList_info[i];
      }
   else if HANDLER {
      // get IP and port number from JNS
       ArgList_info[i]= myRequest(handler);;
      }
   }

   // make NetSolve non-blocking assignment call
   // to invoke Server Connector
   err=netslnb_assignment("server_best:connecto
                          r", ProcName,
                          ArgList_info);

   // set up socket waiting for computational
   // servers to connect in to download local
   // input data described by Arglist_info.
   for (i=0;i<num_input_arg;i++) {
      mysocket_wait( data_input[i] );
      }

   // wait until result info is returned
   result_info = mysocket_wait();

   // receive results data from Server Connector,
   // or submit this info to JNS
   for (i=0;i<num_output_arg;i++) {
     if LOCAL RESULT {
        result[i] = mysocket_get(result_info[i]);
        }
    else if HANDLER {
        myRegister(result_info[i]);
       }
     }
    }
//  if there is no available server
//  which has registered both
// the problem and Connector
else
  {
   // get address of variables which store the result
   // of computation from JNS.
   addr_info=myRequest(ArgList);

   // create a new ArgList by replacing handlers
   // with address of variables which store the
result
   // of computation
   new_ArgList = mycreate(ArgList, getPDF());

   // use original netsl to submit task
  err=netsl(ProcName, new_ArgList);

 // register address of variables storing the result
 // of computation  to JNS
 myRegister(new_ArgList);
 }
}
```

Appendix B. Pseudo code for connector*()*.

```
int connector(ProcName, ArgList_info) {
// check the ArgList_info

// data by using input source information
for (i=0;i<ArgNum;i++) {
   mysocket_get(source_info[Arg_num]);
   }

// Re-Submit our computational function which
// user want to compute result
err=netslnb_assignment("localhost:ProcName"
                              ,input1,input2,…);

// fill result_info with server's ip and port
// number
local ip and port -> result_info;
// return result_info to client
mysocket_send(result_info);
// set up socket waiting for client or
// another computational server to down
// -load result
mysocket_wait(result);
}
```

April 2005.

```
// get all input source information by
// extracting ArgList_info
source_info = extract(ArgList_info);

// set up sockets to download all input
```