GrADSolve - a Grid-based RPC system for Remote Invocation of Parallel Software ⋆

Sathish S. Vadhiyar *, Jack J. Dongarra

Department of Computer Science, University of Tennessee

Abstract

Although the existing Remote Procedure Call (RPC) systems provide adequate support for remote execution of sequential software, the support for remote invocation of parallel software is fairly limited. Some RPC systems support parallel execution of software routines with simple modes of parallelism. Some RPC systems statically choose the configuration of resources for parallel execution even before the parallel routines are invoked remotely by the end user. These policies of the existing systems prevent them from being used for remotely solving computationally intensive parallel applications over the dynamic computational Grid environments. In this paper, we discuss a RPC system called GrADSolve that supports execution of parallel applications over Grid resources. In GrADSolve, the resources used for the execution of parallel application are chosen dynamically based on the load characteristics of the machines. Also GrADSolve stages the user's data to the end resources based on the data distribution used by the end application. Finally, GrADSolve allows the users to store execution traces for problem solving and use the traces for subsequent solutions. Experiments are presented to prove that GrADSolve's data staging mechanisms can significantly reduce the overhead associated with data movement in current RPC systems. Results are also presented to demonstrate the usefulness of utilizing execution traces maintained by GrADSolve for problem solving.

Key words: RPC, Grid, GrADSolve, application-level scheduling, data staging, execution traces

⋆ This work is supported in part by the National Science Foundation contract GRANT #EIA-9975020, SC #R36505-29200099 and GRANT #EIA-9975015

* Corresponding author.
Email addresses: vss@cs.utk.edu (Sathish S. Vadhiyar), dongarra@cs.utk.edu (Jack J. Dongarra).

1 Introduction

Remote Procedure Call (RPC) mechanisms have been studied extensively and have been found to be powerful abstractions for distributed computing [1,2]. In RPC frameworks, the end user invokes a simple routine to solve problems over remote distributed resources. A number of RPC frameworks have been implemented and are widely used [3–11]. In addition to providing simple interfaces for uploading applications into the distributed systems and for remote invocation of the applications, some of the RPC systems also provide service discovery, resource management, scheduling, security and information services.

The role of RPC in Computation Grids [12] has been the subject of recent studies [13–17]. Computational Grids consists of large number of machines ranging from workstations to supercomputers and strive to provide transparency to the end users and high performance for end applications. While high performance is achieved by the parallel execution of applications on large number of Grid resources, user transparency can be achieving by employing RPC mechanisms. Hence Grid-based RPC systems need to be built to provide the end users the capability to invoke remote parallel applications on Grid resources by a simple sequential procedure call.

A number of issues are involved in the remote execution of parallel applications on Grid resources from a sequential environment. First, the number and the location of the resources where the parallel application will be executed have to be chosen dynamically based on the load dynamics of the Grid resources. Secondly, the user's data have to be partitioned and the different partitions of the data have to be scattered among the different resources involved in the execution of parallel application depending on the data distribution used by the end application. Similarly, the different partitions corresponding to the output data have to be gathered and copied to the user's address space. Though there are large number of RPC systems that adequately support the remote invocation of sequential software from sequential environments, the number of RPC systems for supporting invocation of parallel software are relatively few [7,9,10,17–20]. Some of these parallel RPC systems [7,17,18] require invocation of remote parallel services from only parallel clients. Some of the RPC systems [9,10] support only master-slave or task farming models of parallelism. Few RPC systems [10,9] fix the amount of parallelism at the time when the services are uploaded into the RPC system and hence are not adaptive to the load dynamics of the Grid resources. Few RPC systems [18–20] supporting invocation of parallel software are implemented on top of object oriented frameworks like CORBA and JavaRMI and may not be suitable for high performance computing according to a previous study [21].

In this paper, we propose a Grid-based RPC system called GrADSolve [1] that enables the users to invoke MPI applications on remote Grid resources from a sequential environment. In addition to providing easy-to-use interfaces for the service providers to upload the parallel applications into the system and for the end users to remotely invoke the parallel applications, GrADSolve also provides interfaces for the service providers or library writers to upload execution models that provide information about the predicted execution costs of the applications and the data distribution used for the different data in the applications. These information are used by GrADSolve to perform application-level scheduling and dynamically choose the resources for the execution of the parallel applications based on the load dynamics of the Grid resources. GrADSolve also uses the data distribution information provided by the library writers to partition the users' data and stage the data to the different resources used for the application execution. Our experiments show that the data staging mechanisms in GrADSolve helps reduce the data staging times in RPC systems by 20-50%. GrADSolve also uses the popular Grid computing tool, Globus [24] for transferring data between the user and the end resources and for launching the application on the Grid resources. In addition to the above features, GrADSolve also enables the users to store execution traces for a problem run and use the execution traces for the subsequent problem runs. This feature helps in significantly reducing the overhead incurred due to the selection of the resources for application execution and the staging of input data to the end resources.

Thus, the contributions of our research are:

(1) design and development of an RPC system that utilizes standard Grid Computing mechanisms for invocation of remote parallel applications from a sequential environment.
(2) selection of resources for parallel application execution based on load conditions of the resources and application characteristics.
(3) communication of data between the user's address space and the Grid resources based on the data distribution used in the application and
(4) maintenance of execution traces for problem runs.

Section 2 presents a brief overview of the of the GrADSolve system. The various entities in the GrADSolve system and the support for the entities in the GrADSolve system are explained in Section 3. Section 3 also deals with the detailed description of the framework of the GrADSolve system. The support in the GrADSolve system for maintaining execution traces is explained in Section 4. In Section 5, the experiments conducted in GrADSolve are explained and results are presented to demonstrate the usefulness of the data staging

[1] The system is called GrADSolve since it is derived from the experiences of the GrADS [22] and NetSolve [9,23] projects.

mechanisms and execution traces in GrADSolve. Section 6 looks at the related efforts in the development of RPC systems. Section 7 presents conclusions and future work.

2 Overview of GrADSolve

The general architecture of GrADSolve is shown in Figure 1. At the core of the GrADSolve system is a XML database implemented with Apache Xindice [25]. Since XML is mostly useful for storing metadata and transferring compatible documents across the network, GrADSolve uses XML as a language for storing information about different Grid entities. This database maintains four kinds of tables - *users*, *resources*, *applications* and *problems*. The *users* table contains information about the different users of the Grid system, namely the home directories of the users on different resources. The *resources* table contains information about the different machines in the Grid, namely the names of the machines, the clusters to which the machines belong, the architecture and the operating system in the machines, the peak performance of the machines etc. The *applications* table contains information about different applications, namely the name and owner of the application, if the application is sequential or parallel, the language in which the application is written, the number of input and output arguments, the data type and size of the arguments, the location of the binaries of the applications on each of the resources etc. Finally, the *problems* table maintains information about the individual problem runs due to the invocation of the remote applications by the end users. All the above mentioned information are stored in the XML database in the form of XML documents. The Xindice implementation of the XML-RPC standard [4] was used for storing and retrieving information to and from the XML database.

The library writer uploads his application into the Grid system specifying the problem description of the application using an Interface Definition Language (IDL). The GrADSolve system creates a wrapper for the application, compiles the wrapper along with the application and transports the executable application to the different resources of the Grid system using the Globus GridFTP mechanisms. The library writer also has the option of adding an execution model for the application. The information regarding the locations of the end applications on the resources are stored in the Xindice XML database.

The end user writes a client program in C or Fortran to execute applications over the Grid. The GrADSolve client accesses the XML database, retrieves the problem specification for the application and matches the user's data with the parameters of the problem. The GrADSolve client also downloads the execution model of the application from a remote resource if the application possesses an execution model. For the resources that contain the application,

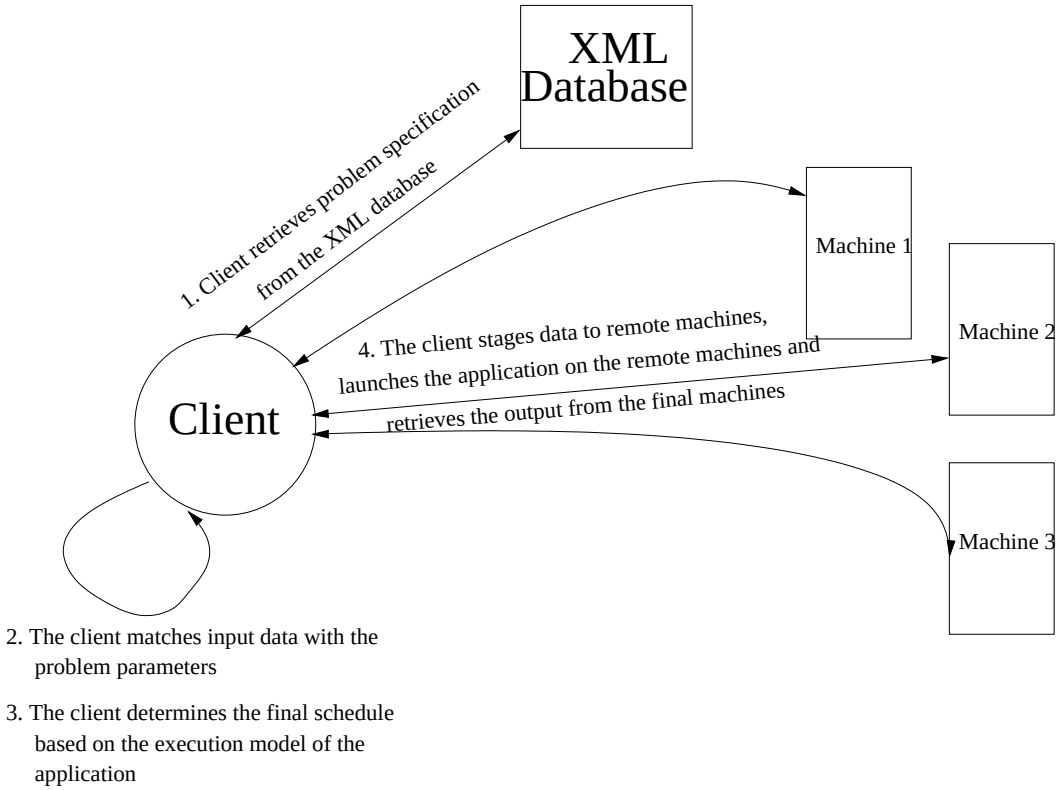

Fig. 1. Overview of GrADSolve system

the GrADSolve client retrieves the various performance characteristics including the peak performance of the resources, the load on the machines, the latency and the bandwidth of the networks between the machines and the free memory available on the machines from the Network Weather Service (NWS) [26].

Based on the resource characteristics of the machines and the execution model of the application, the GrADSolve client determines an application-level schedule for application execution by employing scheduling heuristics [27,28]. The application-level schedule consists of the list of Grid resources for the execution of end application. After determining the final application-level schedule, the GrADSolve client partitions the user's input data and stages the appropriate blocks of data to the different resources using the Globus GridFTP mechanisms. The client then spawns the application on the set of resources using MPICH-G [29]. Similar to the staging of the input data, the client gathers the different blocks of output data from different resources using GridFTP and copies the data to the user's memory.

3 GrADSolve Entities

There are three human entities involved in GrADSolve - *administrators*, *library writers* and *end users*. The role of these entities in GrADSolve and the functions performed by the GrADSolve system for supporting these entities are explained in the following sub sections.

3.1 Administrators

The GrADSolve administrator is responsible for managing the users and resources of the GrADSolve system. The administrator initializes the XML database and creates entities for different users in the XML database by specifying a *user configuration file*. The user configuration file contains information for different users, namely the user account names for different resources and the location of the home directories on different resources in the GrADSolve system. These information are translated into XML documents and stored in the *users* table of the Xindice database. Finally, the administrator creates the *resources* table in the Xindice database and adds entries for different resources in the GrADSolve system by specifying a *resource configuration file*. The various information in the configuration file, namely the names of the different machines, their computational capacities, the number of processors in the machines and other machine specifications, are stored as XML documents. The translation of the configuration files into XML documents are automatically handled by the GrADSolve system.

3.2 Library Writers

The library writer uploads his application into the GrADSolve system by specifying an Interface Definition Language (IDL) file for the application. The Backus Normal Form (BNF) of the GrADSolve IDL is given in Figure 2.

In the IDL file, the library writer specifies the name of the problem suite and the description of the problem suite. A problem suite consists of a set of functions that the user can invoke remotely. For each function, the library writer specifies in the IDL file, the programming language in which the function is written, the name of the function, the set of input and output arguments in the function, the description of the function, the names of the object files and libraries needed for linking the function with other functions, if the function is sequential or parallel etc. GrADSolve supports the library functions to be written in the popular C or Fortran languages. For each input and output arguments, the library writer specifies the name of the argument, if the ar-

$$
\begin{aligned}
\langle \text{PROBLEMSTART} \rangle &\longrightarrow \langle \text{PROBLEMDESC} \rangle \, \langle \text{FUNCTION} \rangle \\
\langle \text{PROBLEMDESC} \rangle &\longrightarrow \mathit{PROBLEM} \, \langle \text{PROBLEMNAME} \rangle \\
\langle \text{FUNCTION} \rangle &\longrightarrow \langle \text{LANGUAGE} \rangle \, \mathit{FUNCTION} \, \langle \text{FUNCDEFN} \rangle \, \langle \text{FUNCDESC} \rangle \\
&\quad \langle \text{FUNCLIB} \rangle \, \langle \text{FUNCTYPE} \rangle \\
\langle \text{LANGUAGE} \rangle &\longrightarrow \mathit{C} \mid \mathit{FORTRAN} \\
\langle \text{FUNCDEFN} \rangle &\longrightarrow \langle \text{FUNCNAME} \rangle \; (\langle \text{ARGLIST} \rangle) \\
\langle \text{FUNCDESC} \rangle &\longrightarrow '' \, \langle \text{STRING} \rangle '' \\
\langle \text{FUNCLIB} \rangle &\longrightarrow \mathit{LIBS} = '' \, \langle \text{STRING} \rangle '' \\
\langle \text{FUNCTYPE} \rangle &\longrightarrow \mathit{TYPE} = \langle \text{TYPESTRING} \rangle \\
\langle \text{ARGLIST} \rangle &\longrightarrow \langle \text{ARGUMENT} \rangle \mid \langle \text{ARGLIST} \rangle \, , \langle \text{ARGUMENT} \rangle \\
\langle \text{ARGUMENT} \rangle &\longrightarrow \langle \text{INOUTSTRING} \rangle \, \langle \text{DATATYPE} \rangle \, \langle \text{VARNAME} \rangle \\
&\quad \mid \langle \text{INOUTSTRING} \rangle \, \langle \text{DATATYPE} \rangle \, \langle \text{VARNAME} \rangle \\
&\quad \langle \text{VACTORATTR} \rangle \\
&\quad \mid \langle \text{INOUTSTRING} \rangle \, \langle \text{DATATYPE} \rangle \, \langle \text{VARNAME} \rangle \\
&\quad \langle \text{MATRIXATTR} \rangle \\
\langle \text{VECTORATTR} \rangle &\longrightarrow [\langle \text{DIMENSIONEXPR} \rangle] \\
\langle \text{MATRIXATTR} \rangle &\longrightarrow [\langle \text{DIMENSIONEXPR} \rangle] \; [\langle \text{DIMENSIONEXPR} \rangle] \\
\langle \text{DIMENSIONEXPR} \rangle &\longrightarrow \langle \text{NUMBER} \rangle \mid \langle \text{VARNAME} \rangle \\
\langle \text{PROBLEMNAME} \rangle &\longrightarrow \langle \text{IDENTIFIER} \rangle \\
\langle \text{FUNCNAME} \rangle &\longrightarrow \langle \text{IDENTIFIER} \rangle \\
\langle \text{TYPESTRING} \rangle &\longrightarrow \mathit{sequential} \mid \mathit{parallel} \\
\langle \text{INOUTSTRING} \rangle &\longrightarrow \mathit{IN} \mid \mathit{OUT} \mid \mathit{INOUT} \\
\langle \text{DATATYPE} \rangle &\longrightarrow \mathit{INT} \mid \mathit{FLOAT} \mid \mathit{DOUBLE} \mid \mathit{CHAR} \\
\langle \text{VARNAME} \rangle &\longrightarrow \langle \text{IDENTIFIER} \rangle
\end{aligned}
$$

Fig. 2. BNF of GrADSolve IDL

gument is an input or output argument, the datatype of the argument, the number of elements of the argument if the argument is a vector, the number of rows and columns of the argument if the argument is a matrix etc. The number of elements in the vector arguments and the number of rows and columns of the matrix arguments can be constants or expressed in terms of the other input arguments. An example of a IDL file written for a ScaLAPACK QR factorization problem is given in Figure 3.

After the library writer submits the IDL file to the GrADSolve system, GrADSolve translates the IDL file to a XML document. The XML document generated for the IDL file in Figure 3 is shown in Figure 4.

The GrADSolve translation system also generates a wrapper program. This wrapper program is a driver and acts as an entry point for remote execution of

```
PROBLEM qrwrapper
C FUNCTION qrwrapper(IN int N, IN int NB, INOUT double A[N][N],
                     INOUT double B[N][1])
``a version of qr factorization that works with
  square matrices.''
LIBS = ``/home/grads23/GrADSolve/ScaLAPACK/pdgeqrf_instr.o \
        /home/grads23/GrADSolve/ScaLAPACK/pdscaex_instrQR.o \
        ...
        ...
        ...''
TYPE = parallel
```

Fig. 3. An example GrADSolve IDL for a ScaLAPACK QR problem

the actual function. The wrapper program when compiled and executed performs certain important functions. The wrapper performs the necessary initialization if the end application is a parallel application. The wrapper program then retrieves the problem description from the XML database, initializes the input and output arguments, and reads the input data from the appropriate files into the input arguments. It then invokes the actual function specified in the IDL file with the input and output arguments. Once the actual problem is solved by the execution of the actual function, the wrapper program stores the output arguments to files. It finally performs finalization routines for deregistering from the parallel execution environment.

After generating the wrapper program, the GrADSolve system compiles the wrapper program with the object files and the libraries specified in the IDL file and with the appropriate parallel libraries if the application is specified as a parallel application in the IDL file. The result of the compilation process is an executable file suitable for remote execution on the GrADSolve resources. The application is finally uploaded into the GrADSolve system. The uploading process consists of two phases. In the first phase, the executable file is staged to the remote machines in the GrADSolve system and stored in the user's accounts on the machines. The machines available in the system are determined by querying the *resources* table in the XML database and the user's home directories on the machines to which the executable file is stored are determined by querying the *users* table in the XML database. In the second phase of the uploading process, the XML document for the application generated from the IDL file is stored in the XML database keyed by the problem name. Also, stored in the XML database for the application is the information regarding the location of the executable files for the application on the remote resources.

If the library writer wants to add an execution model for his application, he executes the *getperfmodel_template* utility, specifying the name of the application. The utility retrieves the problem description of the application from the XML database and generates a performance model template file. The template

```
<?xml version="1.0"?>
  <function name="qrwrapper">
    <user>grads23</user>
    <description>a version of qr factorization that works
                 with square matrices.
    </description>
    <type>parallel</type>
    <language>C</language>
    <continue>1</continue>
    <reconfigure>1</reconfigure>
    <call>
      <argCount>4</argCount>
      <arg>
        <inout>IN</inout>
        <datatype>int</datatype>
        <objecttype>scalar</objecttype>
        <name>N</name>
      </arg>
      <arg>
        <inout>IN</inout>
        <datatype>int</datatype>
        <objecttype>scalar</objecttype>
        <name>NB</name>
      </arg>
      <arg>
        <inout>INOUT</inout>
        <datatype>double</datatype>
        <objecttype>matrix</objecttype>
        <name>A</name>
        <rowExpression>N</rowExpression>
        <colExpression>N</colExpression>
      </arg>
      <arg>
        <inout>INOUT</inout>
        <datatype>double</datatype>
        <objecttype>matrix</objecttype>
        <name>B</name>
        <rowExpression>N</rowExpression>
        <colExpression>1</colExpression>
      </arg>
    </call>
  </function>
```

Fig. 4. XML document generated for the IDL in Figure 3

```
int areResourcesSufficient(int N, int NB, double* A,
                           double* B,
                           RESOURCEINFO* resourceInfo,
                           SCHEDULESTRUCT* schedule){

}

int getExecutionTimeCost(int N, int NB, double* A,
                         double* B,
                         RESOURCEINFO* resourceInfo,
                         SCHEDULESTRUCT* schedule,
                         double* cost{

}

int mapper(int N, int NB, double* A, double* B,
           RESOURCEINFO* resourceInfo,
           SCHEDULESTRUCT* inputSchedule,
           SCHEDULESTRUCT* mapperSchedule){


}
```

Fig. 5. A Performance Model template generated by the GrADSolve system for the QR problem

file contains the definitions of the execution model routines. The library writer fills in the execution model routines with the appropriate code for predicting the execution cost of his application. The performance model template file generated by the *getperfmodel_template* for the ScaLAPACK QR problem is shown in Figure 5.

The performance model template file contains definitions for three functions. The first function *areResourcesSufficient* takes as input the problem parameters, a given set of machines for problem execution, *schedule* and the resource capabilities of the machines, *resourceInfo*. The library writer fills the function such that the function will return 1 if the machines have adequate capacities for solving the problem and 0 otherwise. The second function *getExecutionTimeCost* takes as input the problem parameters, the given set of machines, the resource capabilities and returns as output, the predicted execution cost, *cost*, of the application if the application were to run on the given set of machines. The third function, *mapper* is an optional function. It is used for specifying the data distribution of the different data used by the application. The mapper can also change the order of the machines in the given set of machines represented by *inputSchedule* and return the new order of machines

in the *mapperSchedule*. The execution model for the ScaLAPACK QR application filled with the code written by the library writer is shown in the Figure 6.

The library writer uploads his execution model by executing the *add_perfmodel* utility. After performing certain error checking mechanisms, the *add_perfmodel* utility generates a wrapper program for the execution model. This wrapper program contains functions that act as entry points for the execution model. The functions in the wrapper program initialize certain parameters with default values and invoke the functions in the execution model. The *add_perfmodel* utility finally uploads the execution model for the application by storing the location of the wrapper program and the execution model to the XML database corresponding to the entry for the application.

3.3 End Users

The end users solve problems over remote GrADSolve resources by writing a client program. This client program can be written in C or Fortran. The client program includes an invocation of a routine called *gradsolve()* passing to the function, the name of the end application and the input and output parameters needed by the end application. An example of a GrADSolve client program written in C is shown in Figure 7.

The invocation of the *gradSolve()* routine triggers the execution of the GrADSolve *Application Manager*. As a first step, the Application Manager verifies if the user has credentials to execute applications on the GrADSolve system. GrADSolve uses Globus Grid Security Infrastructure (GSI) [30] for the authentication of users. The Application Manager then queries the XML Database to verify if the application had been previously uploaded by the library writer. If the application had not been uploaded, the Application Manager displays an error message to the user and aborts operation. If the application exists in the GrADSolve system, the Application Manager registers the problem run in the *problems* table of the XML database. The Application Manager then retrieves the problem description from the XML database and matches the user's data with the input and output parameters required by the end application.

If an execution model exists for the end application, the Application Manager downloads the execution model from the remote location where the library writer had previously stored the execution model. The Application Manager compiles the execution model programs with algorithms for scheduling heuristics and starts the application-specific *Performance Modeler* service. The Application Manager then retrieves the list of machines in the GrADSolve system from the *resources* table in the XML database, and retrieves resource charac-

```
int areResourcesSufficient(int N, int NB, double* A,
                           double* B,
                           RESOURCEINFO* resourceInfo,
                           SCHEDULESTRUCT* schedule){
  memAvailable = 0.0;
  for(i=0; i<schedule->count; i++){
    memAvailable +=
       resourceInfo->meminfo[schedule->indices[i]];
  }
  memNeeded = (double)N *( (double)N + (double)NB + 1.0)
              * sizeof(double);
  if(memNeeded > memAvailable){
    return 0; /*resources not sufficient */
  }
  return 1; /*resources sufficient */
}
int getExecutionTimeCost(int N, int NB, double* A,
                         double* B,
                         RESOURCEINFO* resourceInfo,
                         SCHEDULESTRUCT* schedule,
                         double* cost{
  for(j=0; j<N; j+=NB){
    trun += t1+t2; /* t1 and t2 are times for different
                      phases in the iteration */
  }
  *cost = trun;
  return 0;
}
int mapper(int N, int NB, double* A, double* B,
           RESOURCEINFO* resourceInfo,
           SCHEDULESTRUCT* inputSchedule,
           SCHEDULESTRUCT* mapperSchedule){
  setBlockCyclicDistribution("A", mapperSchedule, N*NB);
  B_distribution =
        (int*)malloc(sizeof(int)*mapperSchedule->count);
  B_distribution[0] = N;
  for(i=1; i<mapperSchedule->count; i++){
    B_distribution[i] = 0;
  }
  setDistribution("B", mapperSchedule, B_distribution);
  free(B_distribution);
  getExecutionTimeCost(N, NB, A, B, resourceInfo,
                       mapperSchedule,
                       &(mapperSchedule->cost));
  return 0;
}
```

Fig. 6. A QR Performance Model filled with library writer code

```
#include "gradsolve.h"

int main(int argc, char** argv){
int N, NB;
double* A, * B;
int i;

  N = 1000;
  NB = 40;

  A = (double*)malloc(sizeof(double)*N*N);
  B = (double*)malloc(sizeof(double)*N);

  srand48(time(0));

  for(i=0; i<N*N; i++){
    A[i] = drand48();
    if(i < N){
      B[i] = drand48();
    }
  }

  gradsolve("qrwrapper", N, NB, A, B);

  free(A);
  free(B);

  return 1;


}
```

Fig. 7. GrADSolve C client code for the QR problem

teristics of the machines from the NWS. The Application Manager passes the list of machines, along with the resource characteristics to the Performance Modeler service to determine if the resources are sufficient to solve the problem. If the resources are sufficient, the Application Manager proceeds to the *Schedule Generation* phase.

In the Schedule Generation phase, the Application Manager first determines if the end application has an execution model. If an execution model exists, the Application Manager contacts the Performance Modeler service and passes the problem parameters and the list of machines with the machine capabilities. The Performance Modeler service uses the execution model supplied by the

library writer along with the scheduling heuristics [27,28] to determine a final schedule for application execution and returns the final list of machines to the Application Manager. Along with the final list of machines and the predicted execution cost for the final schedule, the Performance Modeling service also returns information about the data distribution for the different data in the end application. If an execution model does not exist for the end application, the Schedule Generation phase adopts default scheduling strategies to generate the final schedule for end application execution. At the end of the Schedule Generation phase, the GrADSolve Application Manager receives a list of machines for final application execution. The Application Manager then stores the status of the problem run and the final schedule in the *problems* table of the XML database corresponding to the entry for the problem run.

The Application Manager then creates working directories on the remote machines of the final schedule for end application execution and enters the *Application Launching* phase. The Application Launching phase consists of several important functions. The Application Launcher stores the input data to files and stages these files to the corresponding remote machines chosen for application execution. If data distribution information for an input data does not exist, the Application Launcher stages the entire input data to all the machines involved in end application execution. If the information regarding data distribution for an input data exists, the Application Launcher stages only the appropriate portions of the data to the corresponding machines. This kind of selective data staging significantly reduces the time needed for the staging for entire data especially if large amount of data is involved. Apart from staging the input data, the Application Launcher also stages the information regarding data distribution to the remote machines.

After the staging of input data, the Application Launcher launches the end application on the remote machines chosen for the final schedule using the Globus MPICH-G [29] mechanism. The end application reads the input data that were previously staged by the Application Launcher and solves the problem. The end application then stores the output data to the corresponding files on the machines in the final schedule. If the end application finished execution, the Application Launcher copies the output data from the remote machines to the user's memory space. The staging in of the output data from the remote locations is a reverse operation of the staging out of the input data to the remote locations. The GrADSolve Application Manager finally returns success state to the user client program.

4 Execution Traces in GrADSolve - Storage, Management and Usage

One of the unique features in the GrADSolve system is the ability provided to the users to store and use execution traces of problem runs. There are many applications in which the outputs of the problem depend on the exact number and configuration of the machines used for problem solving. For example, considering the problem of adding large number of double precision numbers, one of the parallel implementations of the problem is to partition the list of double precision numbers among all processes of the parallel application, compute local sums of the numbers in each process and compute the global sum of the local sums computed on each process. The final sum obtained for the same set of double precision numbers may vary from one problem run to another depending on the number of elements in each partition, the number of processes used in the parallel application and the actual processes used in the computation. This is due to the impact of the round off errors caused by the addition of double precision numbers. In general ill-conditioned problems or unstable algorithms can give rise to vast changes in output results due to small changes in input conditions. For these kinds of applications, the user may desire to use the same input environment for all problem runs. Also, during testing of new numerical algorithms over the Grid, different groups working on the algorithm may want to ensure that same results are obtained when the algorithms are executed with same input data on the same configuration of resources.

To guarantee reproducibility of numerical results in the above situations, GrADSolve provides capability to the users to store *execution traces* of problem runs and use the execution traces during subsequent executions of the same problem with the same input data. For storing the execution trace of the current problem run, the user executes his GrADSolve program with a configuration file called *input.config* in the working directory containing the following line:

$$\text{TRACE\_FLAG} = 1$$

During the registration of the problem run with the XML database, the value of the TRACE_FLAG variable is stored. The GrADSolve Application Manager proceeds to other stages of its execution. After the end application completes its execution and the output data are copied from the remote machines to the user's memory, the Application Manager, under default mode of operation, removes the remote working directories used for storing the files containing the input data for the end application. But when the user wants to store the execution trace of the problem run, i.e. when the *input.config* file contains

"TRACE_FLAG = 1" line, the Application Manager retains the input data used for the problem run in the remote machines. At the end of the problem run, the Application Manager generates an output configuration file called *output.config* containing the following line

TRACE_KEY = <key>

The value *key* in the *output.config* is a pointer to the execution trace stored for the problem run.

When the user wants to execute the problem with the execution trace previously stored, he executes his client program specifying the line,

TRACE_KEY = <key>

in the *input.config* file. The value *key* in the input.config, is the same value previously generated by the GrADSolve Application Manager when the execution trace was stored. The Application Manager first checks if the TRACE_KEY exists in the *problems* table of the XML database. If the TRACE_KEY does not exist, the Application Manager displays an error message to the user and aborts operation. If the TRACE_KEY exists for an execution trace of a previous problem run, the Application Manager registers the current problem run with the XML database and proceeds to the other stages of its execution. During the Schedule Generation phase, the Application Manager, instead of generating a schedule for the execution of the end application, retrieves the schedule used for the previous problem run corresponding to the TRACE_KEY, from the *problems* table in the XML database. The Application Manager then checks if the capacities of the resources in the schedule at the time of trace generation are comparable to the current capacities of the resources. If the capacities are not comparable, the Application Manager displays an error message to the user and aborts the operation. If the capacities are comparable, the Application Manager proceeds to the rest of the phases of its execution. During the Application Launching phase, the Application Manager, instead of staging the input data to remote working directories, copies the input data and the data distribution information, used in the previous problem run corresponding to the TRACE_KEY, to the remote working directories. The use of the same number of machines and the same input data used in the previous schedule also guarantees the use of the same data distribution for the current problem run. Thus GrADSolve guarantees the use of the same execution environment used in the previous problem run for the current problem run, and hence guarantees reproducibility of numerical results.

To support the storage and use of execution traces in the GrADSolve system, two trigger functions are associated with the XML database. One trigger

function called *trace_usage_trigger* updates the last usage time of an execution trace when the execution trace is used for a problem run. Another trigger function called *cleanup_trigger* is used for periodically deleting entries in the *problems* table of the XML database thereby maintaining the size of the *problems* table in the database. The *cleanup_trigger* is invoked whenever a new entry corresponding to a problem run is added to the *problems* table. The *cleanup_trigger* first deletes those entries for which the execution traces were not stored if the entries existed in the database for more than 10 minutes. The *cleanup_trigger* then deletes those entries for which the execution traces were stored, if the time of last usage of the execution trace is greater than 30 days. Thus by using longer duration for those problem runs for which execution traces were stored, the *cleanup_trigger* function provides greater opportunity for the usage of the execution traces for the subsequent problem runs. If no entries meet the above criteria for deletions and the number of entries for which the execution traces were stored is greater than 100, the *cleanup_trigger* function orders the entries based on completion times and deletes the first few entries in the list till the number of entries in the database decreases to less than 100.

5 Experiments and Results

The GrADS testbed consists of about 40 machines from University of Tennessee (UT), University of Illinois, Urbana-Champaign (UIUC) and University of California, San Diego (UCSD). For the sake of clarity, our experimental testbed consists of 4 machines:

- a 933 MHz Pentium III machine with 512 MBytes of memory located in UT,
- a 450 MHz Pentium II machine with 256 MBytes of memory located in UIUC and
- 2 450 MHz Pentium III machines with 256 MBytes of memory located in UCSD.

The 2 UCSD machines are connected to each other by 100 Mb switched Ethernet. Machines from different locations are connected by Internet. In the experiments, GrADSolve was used to remotely invoke ScaLAPACK driver for solving the linear system of equation, $AX = B$. The driver invokes ScaLAPACK QR factorization for the factorization of matrix, A. Block cyclic distribution was used for the matrix, A and the right-hand side vector, B. A GrADSolve IDL was written for the driver routine and an execution model that predicts the execution cost of the QR problem was uploaded into the GrADSolve system. The GrADSolve user invokes the remote parallel application by passing the size of the matrix, the matrix, A and the right-hand side vector, B to the

gradsolve() call.

GrADSolve was operated in 3 modes. In the first mode, the execution model did not contain information about the data distribution used in the ScaLAPACK driver. In this case, GrADSolve transported the entire data to each of the locations used for the execution of the end application. This mode of operation is practiced in RPC systems that do not support the information regarding data distribution. In the second mode, the execution model contained information about the data distribution used in the end application. In this case, GrADSolve transported only the appropriate portions of the data to the locations used for the execution of end application. In the third mode, GrADSolve was used with an execution trace corresponding to a previous run of the same problem. In this case, data is not staged from the user's address space to the remote machines, but temporary copies of the input data used in the previous run are made for the current problem run.

Figure 8 shows the times taken for data staging and other GrADSolve overhead for different matrix sizes and for the three modes of GrADSolve operation. Since the times taken for the execution of the end application are same in all the three modes, we focus only on the times taken for data staging and possible Grid overheads. The machines that were chosen by the GrADSolve application-level scheduler for the execution of end application for different matrix sizes are shown in Table 1. The UT machine was used for smaller problem sizes since it had larger computing power than other machines. For matrix size, 5000, UIUC machine was also used for the execution of parallel application. For matrix sizes, 6000 and 7000, the available memory in the UT machine at the time of the experiments was less than the memory needed for the problems. Hence UIUC and UCSD machines were used. For matrix size, 8000, all 4 machines were needed to accommodate the problem. All the above decisions were automatically made by the GrADSolve system taking into account the size of the problems and the resource characteristics at the time of the experiments.

Comparing the first two modes in Figure 8, we find that for smaller problem sizes, the times taken for data staging in both the modes are the same. This is because only one machine was used for problem execution and the same amount of data are staged in both the modes when only one machine is involved for problem execution. For larger problem sizes, the times for data staging with distribution information is less than 20-55% of the times taken for staging the entire data to remote resources. Thus the use of data distribution information in GrADSolve can give significance performance benefits when compared to staging the entire data that is practiced in some of the RPC systems. Data staging in the third mode is basically the time taken for creating temporary copies of data used in the previous problem runs in remote resources. We find this time to be negligible when compared to the first two

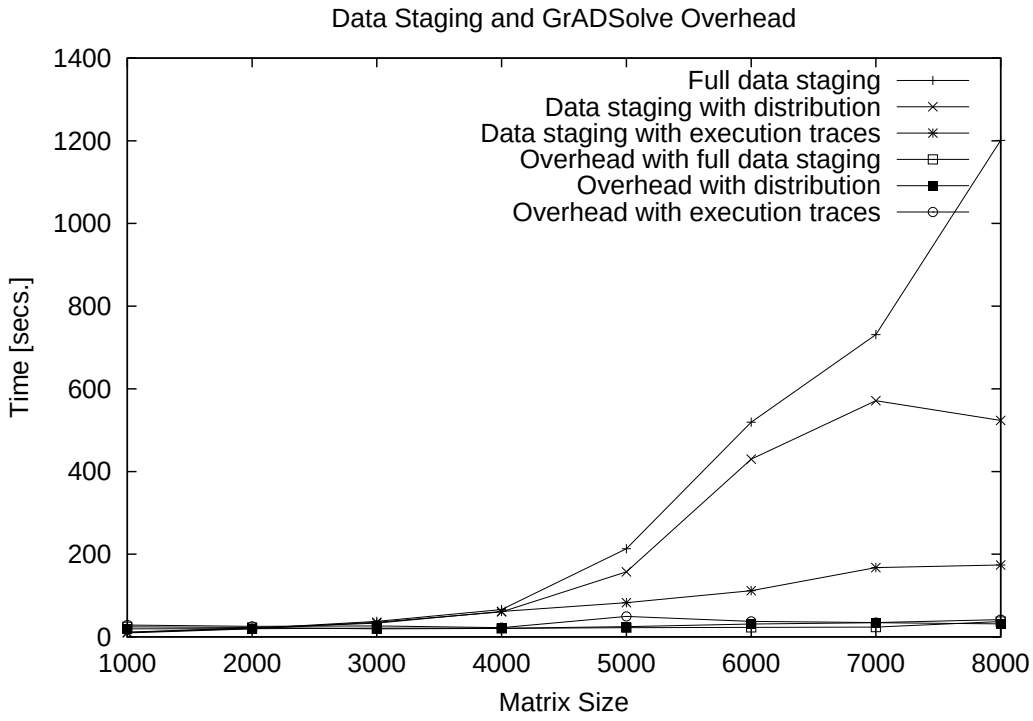

Fig. 8. Data staging and other GrADSolve overhead

Matrix size	*Machines*
1000	1 UT machine
2000	1 UT machine
3000	1 UT machine
4000	1 UT machine
5000	1 UT, 1 UIUC machines
6000	1 UIUC, 1 UCSD machines
7000	1 UIUC, 1 UCSD machines
8000	1 UT, 1 UIUC, 2 UCSD machines

Table 1
Machines chosen for application execution

modes. Thus execution traces can be used as caching mechanisms to use the previously staged data for problem solving. The GrADSolve overheads for all the three modes are found to be the same. This is because of the small number of machines used in the experiments. For experiments when large number of machines are used, we predict that the overheads will be higher in the first two modes than in the third mode. This is because in the first two modes, the application-level scheduling will explore large number of candidate schedules to determine the machines used for end application while in the third mode,

a previous application-level schedule will be retrieved from the database and used.

6 Related Work

Few RPC systems contain mechanisms for the parallel execution of remote software.

MRPC [7] is a RPC system tuned for providing high performance for MPMD applications on homogeneous clusters. The RPC communications are implemented on top of Active Messages (AM) [31] and the user's client programs are written in Compositional C++ (CC++). The work by Maassen et. al [18] extends Java RMI for efficient communications in solving high performance computing problems. Both MRPC [7] and the Java RMI extension [18] requires the end user's programs to be parallel programs.

NetSolve [9], Ninf [10], RCS [11] and DFN-RPC [8] support task parallelism by the asynchronous execution of number of remote sequential applications. OmniRPC [17] is an extension of Ninf and supports asynchronous RPC calls to be made from OpenMP programs. But similar to the approaches in NetSolve, Ninf, RCS and DFN-RPC, OmniRPC supports only master-worker models of parallelism. NetSolve, Ninf and RCS also support remote invocation of MPI applications, but the amount of parallelism and the locations of the resources to be used for the execution are fixed at the time when the applications are uploaded to the systems and hence are not adaptive to dynamic loads in the Grid environments.

The efforts that are very closely related to GrADSolve are PaCO [19,32] and PaCO++ [14,20] from the PARIS project in France. The PaCO systems are implemented within the CORBA [5] framework to encapsulate MPI applications in RPC systems. The data distribution and redistribution mechanisms in PaCO are much more robust than in GrADSolve and support invocation of remote parallel applications either from sequential or parallel client programs. Recently, the PARIS project has been investigating coupling multiple applications of different types in Grid frameworks [15,16]. Although the PARIS project aims to improve the performance of CORBA for high performance computing, the RPC mechanisms provided in CORBA by the use of client stubs and server skeletons have not found to be favorable for high performance computing according to a previous study [21]. Also, the PaCO projects do not support dynamic selection of resources for application execution as in GrADSolve. Also, GrADSolve supports Grid related security models by employing Globus mechanisms. And finally, GrADSolve is unique in maintaining execution traces that can help bypass the resource selection and data staging

phases.

7 Conclusions and Future Work

In this paper, an RPC system for efficient execution of remote parallel software was discussed. The efficiency is achieved by dynamically choosing the machines used for parallel execution and staging the data to remote machines based on data distribution information. The GrADSolve RPC system also supports maintaining and utilizing execution traces for problem solving. Our experiments showed that the GrADSolve system is able to adapt to the problem sizes and the resource characteristics and yielded significant performance benefits with its data staging and execution trace mechanisms.

Interfaces to the library writers for expressing more capabilities of the end application are currently being designed. These capabilities include the ability of the application to be preempted and continued later with different processor configuration. These capabilities will allow GrADSolve to adapt to changing Grid scenarios. Remote execution of non-MPI parallel programs and applications with different modes of parallelism are also being considered. Support for remote invocation in different programming languages including MATLAB are also part of our future efforts.

References

[1] A. Birrell, B. Nelson, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems 2 (1) (1984) 39–59.

[2] B. Bershad, T. Anderson, E. Lazowska, H. Levy, Lightweight Remote Procedure Call, ACM Transactions on Computer Systems (TOCS) 8 (1) (1990) 37–55.

[3] Simple Object Access Protocol (SOAP) `http://www.w3.org/TR/SOAP`.

[4] XML-RPC `http://www.xmlrpc.com`.

[5] CORBA `http://www.corba.org`.

[6] Java Remote Method Invocation (Java RMI) `java.sun.com/products/jdk/rmi`.

[7] C.-C. Chang, G. Czajkowski, T. von Eicken, MRPC: A High Performance RPC System for MPMD Parallel Computing 29 (1) (1999) 43–66.

[8] R. Rabenseifner, The dfn remote procedure call tool for parallel and distributed applications, in: In Kommunikation in Verteilten Systemen - KiVS 95. K.

Franke, U. Huebner, W. Kalfa (Editors), Proceedings, Chemnitz-Zwickau, 1995, pp. 415–419.

[9] H. Casanova, J. Dongarra, NetSolve: A Network Server for Solving Computational Science Problems, The International Journal of Supercomputer Applications and High Performance Computing 11 (3) (1997) 212–223.

[10] H. N. M. Sato, S. Sekiguchi, Design and Implementations of Ninf: towards a Global Computing Infrastructure, Future Generation Computing Systems, Metascomputing Issue 15 (5-6) (1999) 649–658.

[11] P. Arbenz, W. Gander, M. Oettli, The remote computation system, Parallel Computing 23 (1997) 1421–1428.

[12] I. Foster, C. K. eds., The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

[13] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, H. Casanova, Overview of gridrpc: A remote procedure call api for grid computing, in: M. Parashar (Ed.), Lecture notes in computer science 2536 Grid Computing - GRID 2002, Vol. Third International Workshop, Springer Verlag, Baltimore, MD, USA, 2002, pp. 274–278.

[14] A. Denis, C. Prez, T. Priol, Portable Parallel CORBA Objects: an Approach to Combine Parallel and Distributed Programming for Grid Computing, in: Proc. of the 7th International Euro-Par'01 Conference (EuroPar'01), Springer, 2001, pp. 835–844.

[15] A. Denis, C. Prez, . Priol, Towards High Performance CORBA and MPI Middlewares for Grid Computing, in: C. A. Lee (Ed.), Proc. of the 2nd International Workshop on Grid Computing, no. 2242 in LNCS, Springer-Verlag, 2001, pp. 14–25.

[16] C. Prez, T. Priol, A. Ribes, A Parallel CORBA Component Model for Numerical Code Coupling, in: C. A. Lee (Ed.), Proc. of the 3nd International Workshop on Grid Computing, LNCS, Springer-Verlag, 2002.

[17] M. Sato, M. Hirano, Y. Tanaka, S. Sekiguchi, OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP, in: In Workshop on OpenMP Applications and Tools (WOMPAT2001), 2001.

[18] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for Parallel Programming, ACM Transactions on Programming Languages and Systems (TOPLAS) 23 (6) (2001) 747–775.

[19] C. René, T. Priol, MPI Code Encapsulation using Parallel CORBA Object, in: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, IEEE, 1999, pp. 3–10.

[20] A. Denis, C. Pérez, T. Priol, Achieving Portable and Efficient Parallel CORBA Objects, Concurrency and Computation: Practice and Experience .

[21] T. Suzumura, T. Nakagawa, S. Matsuoka, H. Nakada, S. Sekiguchi, Are Global Computing Systems Useful? - Comparison of Client-Server Global Computing Systems Ninf, Netsolve versus CORBA, in: In Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS '00, 2000, pp. 547–559.

[22] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, R. Wolski, The GrADS Project: Software Support for High-Level Grid Application Development, International Journal of High Performance Applications and Supercomputing 15 (4) (2001) 327–344.

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

[24] I. Foster, C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, Intl J. Supercomputer Applications 11 (2) (1997) 115–128.

[25] Apache Xindice `http://xml.apache.org/xindice`.

[26] R. Wolski, N. Spring, J. Hayes, The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, Journal of Future Generation Computing Systems 15 (5-6) (1999) 757–768.

[27] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, S. Vadhiyar, Numerical Libraries and the Grid: The GrADS Experiments with Scalapack, Journal of High Performance Applications and Supercomputing 15 (4) (2001) 359–374.

[28] A. Yarkhan, J. Dongarra, Experiments with Scheduling Using Simulated Annealing in a Grid Environment, in: M. Parashar (Ed.), Lecture notes in computer science 2536 Grid Computing - GRID 2002, Vol. Third International Workshop, Springer Verlag, Baltimore, MD, USA, 2002, pp. 232–242.

[29] I. Foster, N. Karonis, A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems, In Proceedings of SuperComputing 98 (SC98) .

[30] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, V. Welch, A National-Scale Authentication Infrastructure, IEEE Computer 33 (12) (2000) 60–66.

[31] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, Active Messages: A Mechanism for Integrated Communication and Computation, in: 19th International Symposium on Computer Architecture, Gold Coast, Australia, 1992, pp. 256–266.

[32] C. René, T. Priol, MPI Code Encapsulating using Parallel CORBA Object, Cluster Computing 3 (4) (2000) 255–263.