SmartGridRPC: The new RPC model for high performance Grid computing

Thomas Brady[†], Jack Dongarra[‡], Michele Guidolin[†], Alexey Lastovetsky[†], Keith Seymour[‡]

[†]School of Computer Science and Informatics, University College Dublin, Ireland
‡Department of Electrical Engineering and Computer Science, University of Tennessee, USA

Technical Report UCD-CSI-2009-10
October, 2009

ABSTRACT

The paper presents the SmartGridRPC model, an extension of the GridRPC model, which aims to achieve higher performance. The traditional GridRPC provides a programming model and API for mapping individual tasks of an application in a distributed Grid environment, which is based on the client-server model characterised by the star network topology. SmartGridRPC provides a programming model and API for mapping a group of tasks of an application in a distributed Grid environment, which is based on the fully connected network topology. The SmartGridRPC programming model and API, its implementation in SmartGridSolve and its performance advantages over the GridRPC model are outlined in this paper. In addition, experimental results using a real-world application are also presented.

1. INTRODUCTION

The remote procedure call (RPC) paradigm [1] is widely used in distributed computing. It provides a straightforward procedure for executing parts of an application on a remote computer. To execute a RPC, the application programmer does not need to learn a new programming language but merely uses the RPC API. Using the API the application programmer specifies the remote task to be performed, the server to execute the task, the location of the input data on the user's computer required by the task and the location on the user's computer where the results will be stored. The execution of the remote call involves transferring input data from the user's computer to the remote computer, executing the task on the remote server and delivering output data from the remote computer to the user's one.

GridRPC [2] is a standard promoted by the Open Grid Forum, which extends the traditional RPC. GridRPC differs from the traditional RPC in that the programmer does not need to specify the server to execute the task. When the programmer does not specify the server, the middleware system, which implements the GridRPC API, is responsible for finding the remote executing server. When the program runs, each GridRPC call results in the middleware mapping the call to

a remote server and then the middleware is responsible for the execution of that task on the mapped server. Another difference is that GridRPC is a stubless model, meaning that client programs do not need to be recompiled when services are changed or added. This facilitates the creation of interfaces from interactive environments like Matlab, Mathematica, and IDL. A number of Grid middleware systems have recently become GridRPC compliant including GridSolve [3], Ninf-G [4] and DIET [5].

This simple extension to the RPC however has some limitations affecting the performance of Grid applications. When using the traditional GridRPC to execute tasks remotely, the mapping and execution of the task is one atomic operation, which cannot be separated. As a result, each task is mapped separately and independently of other tasks of the application.

Another important aspect of the GridRPC model is its communication model. The communication model of GridRPC is based on the client-server model or star network topology. This means that a task can be executed on any of the servers and inputs/outputs can only traverse the client-server links.

Mapping tasks individually on to the star network results in mapping solutions that are far from optimal. If tasks are mapped individually, the mapping heuristic is unable to take into account any of the tasks that follow the task being mapped. Consequently, the mapping heuristic does not have the ability to optimally balance the load of computation and communication. Another consequence of mapping tasks in this way is that dependencies between tasks are not known at the time of mapping. Therefore this approach forces bridge communication. Bridge communication occurs when the output of one task is required as an input to another task. In this case, using the traditional GridRPC model, the output of the first task must be sent back to the client and the client then subsequently sends it to the server executing the second task when it is called.

Also, since dependencies are not known and the network is based on the client-server model, it is impossible to employ any parallelism of communication between the tasks in the group. For example, this can be implemented if there is a dependency between two tasks and the destination task is not executed in parallel or immediately after the source task. In theory, this dependent data could be sent to the destination task in parallel with any computation or communication on any other machine (client or other servers) which happens in the intervening time. However, since tasks are mapped individually on to a star network, this parallelism of communication cannot be realized using the GridRPC model.

In this paper, we propose an enhancement of the traditional GridRPC model, which would allow a group of tasks to be mapped collectively on to a fully connected network. This would remove each of the limitations of the GridRPC model already described. The SmartGridRPC model has extended the GridRPC model to support collective mapping of a group of tasks by separating the mapping of tasks from their execution. This allows the group of tasks to be mapped collectively and then executed collectively.

In addition, the traditional client-server model of GridRPC has been extended so that the group of tasks can be collectively executed on to a network topology, which is fully connected. This is a network topology where all servers can communicate directly or servers can cache their outputs locally.

There are a number of advantages of mapping tasks collectively on to a fully connected network. When mapping tasks individually, the communication and computation load of a single task are only considered. However, when tasks are mapped collectively, the

communication and computation load of multiple tasks can be considered together and therefore this load can be better distributed over the fully connected network. In addition, the relationships between tasks can also be considered such as the data dependencies between tasks. This allows bridge communication to be eliminated by mapping these dependencies on to virtual links connecting servers. As a result, servers can send data directly to other servers and therefore do not need to send it via the client. Eliminating bridge communication can significantly decrease the overall communication time of an application and hence improve the overall performance of the application because communication is often the more time consuming phase in a RPC context.

In addition, this may also eliminate memory paging on the client, which would otherwise occur when storing intermediate data. Also, since dependencies between tasks are known, it means that remote communication of one task can be parallelized with other computation and communication in the group.

The client-server model of GridRPC results in a communication network, which has a star topology. Therefore, in this case for any given mapping of a group of tasks to remote servers there will be only one communication path between any pair of servers that could be considered when mapping. This path consists of two communication links connecting the servers with the client machine. Any other path connecting the two servers obviously results in higher communication cost.

However, if the communication network is fully connected, then there will be multiple independent paths connecting the servers and each of these paths can be considered when mapping. In other words, for each mapping of a group of tasks to remote servers in a star communication network there is only one fixed communication scheme that can be employed. However, when a group of tasks are mapped on a fully connected network there are many communication schemes to choose from. These communication schemes may employ direct communication, server broadcast, client broadcast or caching. Therefore, the mapping of a group of tasks on a fully connected network not only involves the mapping of tasks to servers but also the mapping of the dependencies between tasks on to the communication paths of the network. This increases the mapping solution space and allows for further optimization to be achieved by choosing the optimal paths for data to traverse between servers. This increase of the solution space means that the mapping heuristics implemented in the SmartGridRPC model have more potential of finding a more optimal solution than the mapping heuristics inherent in implementations using the standard GridRPC model.

GridSolve is a middleware system that implements the GridRPC model. It enables users to solve complex scientific tasks remotely on distributed resources. GridSolve emphasises ease-of-use for the user and includes resource monitoring, mapping and service-level fault tolerance. In addition to providing Fortran and C clients, GridSolve enables SCEs (such as Matlab) to be used as clients, so domain scientists can use Grid resources from within their preferred environments.

SmartGridSolve [6] is an extension of GridSolve, which makes the GridSolve middleware compliant with the SmartGridRPC model. SmartNetSolve [7] was previously implemented to make the NetSolve [8] middleware, which was the predecessor of GridSolve, compliant with the SmartGridRPC model.

The SmartGridSolve extension is interoperable with GridSolve. Therefore, if GridSolve is installed with the SmartGridSolve extension, the user can choose whether to implement an application using the standard GridRPC model or the extended SmartGridRPC model. In

addition, SmartGridSolve is incremental to the GridSolve system. Therefore, if the SmartGridSolve extension is installed only on the client side, the system will be extended to allow for collective mapping. If SmartGridSolve is installed on the client side and on only some of the servers in the network, the system will be extended to allow for collective mapping on a partially connected network. If it is installed on all servers, the system will be extended to allow for collective mapping on the fully connected network.

The paper is outlined as follows. Section 2 gives the motivation of GridRPC and SmartGridRPC models. Section 3 outlines research papers, which are related to the SmartGridRPC model. The GridRPC programming model and API are described in Section 4. The SmartGridRPC model and API are described in Section 5. Section 6, describes the implementation of the GridRPC model in GridSolve and Section 7 describes the implementation of the SmartGridRPC model in SmartGridSolve. Section 8 outlines the Hydropad application, which is an astrophysics application we used to benchmark the GridRPC model and the SmartGridRPC model. Section 9 gives experimental results, which compare the GridRPC model with the SmartGridRPC model using the Hydropad application as a benchmark. The paper concludes with Section 10.

2. MOTIVATION

2.1. Motivation: GridRPC model

The following are some of the key benefits of implementing GridRPC enabled applications:

- Improved performance of applications.
- Solution of larger scale applications.
- More control over applications.
- Solution of hardware specific applications.
- Portability.
- Easy and powerful development of applications.

Improved performance of applications: The performance related benefits include the potential for faster solution of a problem of a given size and solution of problems of larger sizes. There are two main reasons for this. Firstly, if parts of the code can be executed in parallel on remote servers then the GridRPC model allows us to implement their parallel execution on remote servers. This parallelisation will decrease the computation time of the application.

Secondly, if the Grid environment contains machines more powerful than the client machine, then remote execution of the tasks of this application on these more powerful machines will also decrease the computation time of the application.

However, this decrease of the computation time does not come for free. The application will pay the communication cost due to remote execution of the tasks. If communication links connecting the client machine and the remote servers are relatively slow, than the acceleration of computations may be compensated by the communication cost resulting in the total execution time of the application being higher than its sequential execution on the client machine.

Solution of large scale applications: The GridRPC model provides a solution for applications, which cannot be executed on a client machine due to their strong demands on the

resources (memory, disk space etc.) of the client machine. In this case, GridRPC provides a means to allocate these demands to remote servers in the Grid environment. For example, the execution of parts of a memory intensive application on remote servers could eliminate heavy paging that would otherwise occur on the client machine.

More control over the application: In some cases, applications that could be executed in a Grid environment could potentially be executed in a high performance computer (HPC) system. Unfortunately, in a HPC system, where applications are executed in batch mode, the user will not have much control over the execution. Grid-enabled applications allow the user to have a high control over its execution because, although the tasks are being computed in remote servers, the main component of the application is running on the client machine. This can be important for applications that need a direct interaction with the data produced. For a given application it would be possible for a user to see intermediate results of the application. Furthermore, while the user/client is checking these results, they could decide on the fly to change some parameters of the application or restart the application.

Solution of hardware-specific applications: Some applications have a task that is inherently remote. For example, a task could be a proprietary pre-compiled binary, which has been compiled for a specific architecture, or a task may be tuned or tweaked to execute more efficiently on a specific type of hardware such as an FPGA.

Furthermore, a task could require interaction with a resource that can only interface with a particular machine such as a telescope, video camera, microscope etc. In such cases, an environment that allows the resources (including software) to be used on a particular computer is needed.

Portability: Since a Grid-enabled application comprises of a client application and server-side compiled executables, the client application can be easily ported, compiled and executed on a new machine in the Grid environment. This does not require the recompilation of server-side task executables, which could make up a large proportion of the application. Also, client implementations can be ported to different environments and languages easily since language-specific stub generators are not required. This also enables many effortless cross-language calling scenarios (e.g. C to Fortran, Matlab to C, etc.).

An easy and powerful development paradigm: Any task, which has been developed for remote execution on GridRPC enabled application can be easily reused in other Grid applications. This situation can reduce the programmer's effort on developing a Grid application. For example, the programmer can use already existing tasks that they would not have the time or skill to write.

2.2. Motivation: SmartGridRPC model

SmartGridRPC model share the same benefits of the GridRPC model. In addition, the following are the key benefits of implementing a SmartGridRPC enabled application over a GridRPC enabled application:

- Improved balancing of computation load.
- Reduced volume of communication.
- Improved balancing of communication load.
- Increased parallelism of communication.
- Reduced memory usage and paging.

Improved mapping of computation load: In GridRPC, tasks get mapped individually on to a client-server network. This could result in poor load balancing of computation. Since tasks get mapped individually, it is impossible to balance the load of computation of a group of tasks, which are executing in parallel. If tasks are mapped individually, each task will be mapped without the knowledge of any of the subsequent parallel tasks. This means that if a large task follows a smaller task, the mapping heuristic will give priority to the smaller task over the larger task.

This is because when the smaller task is mapped, the mapping heuristic cannot take into account that a larger task will be executing in parallel. Therefore, it maps the smaller task to the faster server as this will yield the lowest execution time for this individual task. When the mapping heuristic maps the larger task, it will assign it to the next fastest server as the fastest server is busy executing the previous task. This is poor load balancing of computation.

However, if you implement the collective mapping of the SmartGridRPC model, then the computation load can be better distributed over the network. In this case, if both tasks can be mapped collectively then the larger task would be mapped to the faster server and the smaller to the slower server. This improved balancing of the computation load will increase the performance of the execution of these parallel tasks.

Reduced volume of communication: Since the GridRPC model maps tasks individually on to a client-server network, the model forces bridge communication between tasks. This occurs because dependencies are not known between tasks and data can only traverse the client-server links. As a result, the source task can only send the dependent data to the destination task via the client. This requires two communication steps: - the first from the source task to the client and the second from the client to destination task. However, this can be eliminated with the SmartGridRPC model, where tasks can be mapped collectively on to a network, which is fully connected. Since tasks are mapped collectively, dependencies between tasks are known. These dependencies can then be mapped on to virtual links connecting the source server to the destination server, which is only one communication step. Therefore, the overall volume of communication over the network will be reduced, which would result in improved application performance.

Moreover, if the source task and destination task are both executing on the same server then this output could be cached to the local file system or cached in memory, which would further reduce the overall communication on the network and increase the performance of the application.

Improved mapping of communication load: Since the GridRPC model is based on the client-server model, communication can only be mapped to client server links. This may result in the client links becoming heavily loaded.

SmartGridRPC can increase the performance of an application by better distribution of communication load over the network. When tasks are mapped collectively, the volume of communication of each task in the group of tasks is known. Since the sizes of inputs and outputs of each task are known and this communication is mapped onto a network, which is fully connected, this communication can be better distributed over the fully connected network. This improved load balancing of communication will result in improved overall communication times and hence improved application performance.

Increased parallelism of communication: In much the same way that the GridRPC model improves on the RPC model with the parallelism of computation; the SmartGridRPC improves on the GridRPC model with the parallelism of communication.

With the GridRPC model, the parallelism of communication is limited to the sending of inputs to a non-blocking task, which is an asynchronous operation.

With the SmartGridRPC model any communication on one machine can be done in parallel with computation or communication on any other. This asynchronous communication is only achievable since the dependencies between tasks are known prior to the execution of the group due to the collective mapping.

This parallelism of communication can be advantageous if a task executing on one server has a dependency on another task, which will be executed on another server, and the destination task is not executed immediately after the source task. In this case, this communication can be done asynchronously. This means that the server initiates the communication but does not wait for it to finish. Therefore, this communication can be done in parallel with any other computation or any communication on any other machine (client or servers), which happens in the intervening time.

In addition, this parallelism of communication can be beneficial if the client broadcasts an argument to more than one task, which is to be executed on different machines. If any of the tasks are not executing immediately after the communication, then the communication to these tasks can be done in parallel with any computation on the client machine and computation or communication on any other server, which happens in the intervening time. The same is true for server broadcast communication.

This parallelism of communication reduces overall communication times and thus improves the overall performance of the group of tasks executing on the fully connected network.

Reduced memory usage and paging: The direct communication between servers and the data caching that SmartGridRPC model implements mean that intermediate results do not have to be sent back to the client. This minimizes the amount of memory used on the client. This could eliminate any paging on the client, which would otherwise occur. This elimination of paging would considerably increase the performance of an application.

3. RELATED RESEARCH

This section examines those systems, which implement the GridRPC model (i.e. GridSolve, Ninf-G, DIET) and their predecessors (i.e. NetSolve and Ninf), and will focus on the papers, which mostly relate to our research, specifically those papers, which fall into the following categories:

- Papers presenting extensions, which extend the client-server model to implement direct communication between servers or data persistence.
- Papers presenting extensions, which extend the system so that a group of tasks can be collectively mapped.

These papers will be presented in chronological order and we will outline the limitations of each approach in comparison with the SmartGridRPC model.

Both NetSolve and Ninf, the predecessors to the GridSolve and Ninf-G system, were started at roughly the same time. The projects were both started in 1994 and were first released in 1995.

These systems were designed to resolve the difficulty of performing computational science problems over loosely connected geographically disperse networks. The computational libraries, which the most common computational problems use, may be highly optimized for only certain platforms and do not provide a convenient interface to other computer systems. Other libraries demand considerable programming effort from the user, who may not have the time to learn the required programming techniques. The resolution of these issues was the motivation behind both projects. These systems were called Network Enabled Server (NES) or Problem Solving Environment (PSE) systems and employed a RPC-style model to perform remote computations.

In 1999, task farming [9] was introduced to NetSolve. The farming feature of NetSolve allowed a certain class of tasks, called farming jobs, to be processed collectively. A farming job fell into the class of embarrassingly parallel programs, for which it is very clear how to partition the jobs for parallel programming environments. While these tasks were processed collectively, they were not mapped collectively. Each task was individually mapped but computation loads of subsequent tasks were dynamically adjusted at run-time based on previous task response times.
The limitations of task farming are:

- It can only be implemented for a certain class of application.
- Tasks are mapped individually and therefore the mapping heuristic cannot take advantage of characteristics of the group such as data dependencies.
- Conditional statements cannot exist in the scope of the task farming job.
- Client computation cannot exist in the scope of the task farming job
- The group of tasks is called as one atomic call, therefore intermediate results cannot be viewed or analysed.

In 2000, task sequencing [10] was introduced to NetSolve. Using the task sequencing API a group of tasks could be processed collectively so that data dependencies could be analysed. This group of task is subsequently mapped on to a single server and if any data dependencies exist, the data would be stored locally and not sent back to the client. Therefore, using this API data persistency could be implemented and therefore if dependencies exist, bridge communication could be eliminated.
The limitations of task sequencing are:

- The group of tasks can only be mapped to a single server.
    - This computation load could be better distributed over a number of servers.
    - There may not be a server in the environment that can execute all tasks.
- Conditional statements, such as for, if, while, are forbidden between tasks.
- Client computation cannot exist in the scope of the task sequencing job.

In 2001, data transfers between servers were introduced to NetSolve [11]. This was achieved with an added function to the API, which allowed the user to explicitly outline data dependencies. If there are two tasks, which have a data dependency and are executing on different servers, this data would be stored in the source server when it finished execution and then the destination server would pull the argument from that server when it is called for execution.
The limitations to this approach are:

- Tasks are mapped individually.
- Push communication cannot be implemented when tasks are mapped individually.

    - Increased communication times since communication cannot be done in parallel with computation or other communication.
- The user has to explicitly specify dependencies.
    - More labour intensive.
    - More prone to error.

This feature was later implemented in the GridRPC model in the GridSolve system [12], DIET [13] and NINF-G [14] and had the same limitations.

In July 2002, the DIET system was launched, which implemented an architecture where the scheduler/agent is scattered across a hierarchy of Local Agents and Master Agents. The motivation for this architecture was that it was more scalable and solved the problem of bottlenecks in a centralised agent/scheduler when many clients try to access several servers. In addition, the DIET system employed direct communication between servers and data persistency. Where a dependency existed between tasks, this output would remain on the source server. When the destination task is called for execution, this data would be pulled from the source server. If the source server is the same as the destination server this output would be stored and retrieved locally (data persistency).

The limitations of this approach are:

- Tasks are mapped individually.
- Push communication cannot be implemented when tasks are mapped individually
    - Increased communication times since communication cannot be done in parallel with computation or other communication.

In 2002, Distributed Storage Interface (DSI) [15] was implemented in NetSolve. The DSI was another feature that attempted to minimize data movement in the NetSolve middleware. With DSI, data could be stored in the storage depots, which are close to servers, which require the data. Instead of having multiple transmissions of the same data, DSI allows the transfer of data once from the client to storage depot. A data handle is then used to retrieve only the relevant portions of the stored data when running computations. This reduced communication times but again did not change how tasks are fundamentally mapped.

Also in 2002 the Global Grid Forum (now known as the Open Grid Forum) standardized the RPC mechanism for Grid computing with the GridRPC programming model and API [2]. This was implemented in NetSolve [16] and Ninf-G [4]. Ninf-G is the second generation of Ninf and was implemented on top of the Globus toolkit [17]. The Globus toolkit provides a reference implementation of standard protocols and it deploys Globus Security Infrastructure so that all the components of Ninf-G are protected properly. In 2003, GridSolve, which is the second generation of NetSolve, was released and provided full support for the GridRPC model.

In 2005, the GridRPC model was implemented in DIET [5]. This paper also introduced the Data Tree Manager (DTM). The DTM allows data to be left on a server after computation and then retrieved from another server during its computation. This paper described how JUXMEM (Juxtaposed memory) could be used in the DIET system to allow servers to share memory data. Both the DTM and JUXMEM avoided multiple transmissions of the same data from a client to a server but again tasks could only be processed and mapped individually.

The limitations to this approach are:

- Tasks are mapped individually.

- Push communication is not implemented.

SmartNetSolve was designed in 2004, implemented in 2005 and was first presented in [7] in April 2006. SmartNetSolve is the predecessor to SmartGridSolve. SmartNetSolve allowed a group of tasks to be collectively mapped and collectively executed on a fully connected network. The initial design allowed the user to give a description of the group of tasks and then at run-time this description would be used to generate a task graph. This task graph and a graph of the network were used to generate a mapping solution, which was then in turn used to execute the tasks on the fully connected network. Initially, the description of the task graph was given using an XML file, which was read at run-time. A new language, Application Definition Language (ADL) [18], was also being designed to make this more user friendly for the user.

In September 2006, distributed task sequencing was developed for the GridRPC model [19]. A new function was introduced that allows direct data transfer between servers when executing a task sequencing job in a Grid environment. This meant that multiple servers could be used and not just a single server as was originally a restriction of task sequencing.
The limitations of this approach are:

- Tasks were not mapped collectively.
- Conditional statements cannot exist in the scope of the task sequencing job.
- Client computation cannot exist in the scope of the task sequencing job.
- Push communication is not implemented.

In October 2006, the special agent called MA_{DAG} was implemented in DIET, which handled workflow submissions [20]. The user gives a description of this workflow using an XML file including the values of any arguments (i.e. element values of vectors, matrices, etc). Using the DIET API, the user references the file, which has the DAG description. This is used to create a DAG or task graph, which is submitted to the MA_{DAG} agent, which is responsible for scheduling the DAG. This is implemented for the DIET API and has not yet been implemented for the GridRPC model and API. This implementation does not follow the RPC style of calling each task in the application. Instead, the application calls a function that submits the entire task graph as a single entity.
The limitations to this approach are:

- Since this approach does not follow the GridRPC model, intermediate results cannot be sent back to the client.
- The task graph has to be known at compile time. Therefore, no conditional statements can exist and initial values of input matrices, vectors etc. have to be known before run-time.
- Client computation cannot exist between tasks.
- It is not user friendly as it can be difficult and time consuming to write the XML description of a task graph.
- Writing XML files to generate task graphs is more prone to error than if the task graphs were automatically generated by the system.

Since this initial design of MA_{DAG} system, a GUI has been developed, which has made it more user friendly [21]. However, this is also prone to error due to the fact that the application programmer has to outline the task graph and in addition it is still more labour intensive than if the task graph was automatically generated.

In late 2006, work began on SmartGridSolve and the SmartGridRPC model to address the limitations described above. This was presented in 2008 [6].

4. GRIDRPC PROGRAMMING MODEL AND API

The aim of the GridRPC model is to provide a standardized, portable and simple programming interface for Remote Procedure Call. It intends to unify client access to existing Grid computing systems (such as GridSolve, Ninf-G, DIET and OmniRPC). This is done by providing a single standardized, portable and simple programming interface for Remote Procedure Call (figure 1).

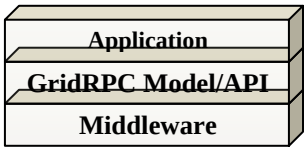

Figure 1: Overview of GridRPC model/API

This standardisation provides portability of the programmers' source code across all GridRPC implemented platforms. Since the GridRPC model specifies the API and the programming model but does not dictate the implementation details of the servers, which will execute the remote procedure call, there may be multiple different middleware implementation of the GridRPC model, in which the source code could be executed on.

4.1. Design of the GridRPC programming model

The functions presented in this section are shared by all the implementations of the GridRPC model. However the mechanics of these functions differ in each implementation.

Register discovery: The servers of the Grid environment register the tasks, which they can execute with a "registry". This involves sending information such as how the client should interface with the task and what type of arguments the server expects when the task is called (the calling sequence). In this paper, the registry will be an abstract term for the entity/entities, which stores the information about the registered tasks and the underlying network. This may be a single entity, such as the Agent in GridSolve, or several entities such as the MDS (or LDIF), running on servers in Ninf-G, or the Global Agents and Local Agents, running in the DIET system.

Run-time of client application: When the GridRPC call *grpc_function_handle_default* is invoked, the client contacts the registry to look-up a desired task and receives a handle, which is used by the client to interface with the remote task. A task handle is a small structure that describes various aspects of the task and its arguments such as:

- The task name (dgesv, dgemm etc.)
- The object types of the arguments (scalars, vectors, matrices etc.)
- The data type of the arguments (integer, float, double, complex etc)
- Whether the arguments are inputs or outputs.

The client then uses the handle to call the task, which eventually returns the results. Each GridRPC call gets processed individually, where each task is discovered (task look-up) and executed separately from all the other tasks in the application.

Currently a task is discovered by explicitly asking the registry for a known function through a string look-up. For applications, which are run using the GridSolve middleware, the discovery mechanism is done via the GridSolve agent. In Ninf -G, discovery is done via the Globus MDS, which runs on each server, and in DIET discovery is done via the Global Agent. The GridRPC model does not dictate the mechanics of resource discovery since different underlying GridRPC implementations may use vastly different protocols.

GridSolve and DIET are GridRPC systems that can perform dynamic mapping of tasks. Discovery for dynamic mapping also involves discovery of performance models, which are used by the mapping heuristics. The performance models for DIET are the FAST prediction tool [5], CORI [21] and NWS [5]. The performance models for GridSolve are described in section 6.3.

4.2. GridRPC: API and semantics

Now we will introduce the fundamental objects and functions of the GridRPC API and explain their syntax and semantics.

The two fundamental objects in the GridRPC model are the task handles and the session IDs. The task handle represents a mapping from a task name to an instance of that task on a particular server.

Once a particular task-to-server mapping has been established by initializing a task handle, all GridRPC calls using that task handle will be executed on the server specified in that binding. In GridRPC systems, which perform dynamic resource discovery and mapping, it is possible to delay the selection of the server until the task is called. In this case, resource discovery and mapping is done when the GridRPC task call is invoked with this initialized handle. In theory, there is more chance to choose a “better” server in this way, since at the time of invocation more information regarding the task and network is known, such as the size of input/outputs, complexity of task and dynamic performance of client-server links.

The two types of GridRPC task call functions are blocking calls and non-blocking calls. The *grpc_call* function makes a blocking remote procedure call with a variable number of arguments. This means the function does not return until the task has completed and the client has received all outputs from the server.

The *grpc_call_async* function makes a non-blocking remote procedure call with a variable number arguments. When this call is invoked, the remote task and data transfer of the input are initiated and the function returns. This means that either the client computation or server computation can be done in parallel with the *grpc_call_async* call.

The *grpc_wait* function waits for the result of the asynchronous call with the supplied session ID. The *grpc_wait_all* function waits for all preceding asynchronous calls.

4.3 GridRPC: A GridRPC application

Table 1 is a simple application, which uses the GridRPC API.

Table 1: GridRPC model – Example application.

```
main()
{
    int N;
    int M;
    double A[N*N], B[N*N], C[N*N];
    double D[M*M], E[M*M], F[M*M], G[M*M];

    grpc_function_handle_t h1, h2, h3;
    grpc_session_t s1, s2;
    grpc_initialize(argv[1]);

    /* initialize */
    char * hndl_str= “bind_server_at_call_time”;

    grpc_function_handle_init(&h1, hndl_str,"mmul/mmul");
    grpc_function_handle_init(&h2, hndl_str, "mmul/mmul");
    grpc_function_handle_init(&h3, hndl_str, "mmul/mmul");

    N=getNSize();
    initMatA(N, A);  initMatB(N, B);
    if(grpc_call_asnc(&h1,&s1, N, A, B, C)!= GRPC_NO_ERROR) {
        fprintf(stderr, "Error in grpc_call\n");
        exit(1);
    }

    M=getMSize();
    initMatD(M, D);  initMatD(M, E);
    if(grpc_call_async(&h2, &s2, M, D, E, F)!=GRPC_NO_ERROR){
        fprintf(stderr, "Error in grpc_call\n");
        exit(1);
    }

    grpc_wait(s1);
    grpc_wait(s2);

    if (grpc_call(&h3, M, C , F, G) != GRPC_NO_ERROR) {
        fprintf(stderr, "Error in grpc_call\n");
        exit(1);
    }

    grpc_function_handle_destruct(&h1);
    grpc_function_handle_destruct(&h2);
    grpc_function_handle_destruct(&h3);
    ...
    grpc_finalize();
}
```

It comprises of three task handles and three corresponding remote calls. The task handles are set up so that the remote call is bound to a server at call time by passing “bind_server_at_call_time” [1] as a parameter. This string could be substituted with a server host name or the user could assign it to the default server by calling *grpc_function_handle_default*.

The task “mmul” takes four arguments: - the size of the matrices, the two input matrices and the one output matrix. In this application, the size of the matrices are not known prior to run-time as they can only be established by executing the local functions (initMatA and initMatB). Therefore, it is impossible for a user to decide which servers to assign which tasks since the size of inputs and outputs and complexity are not known until the application is run. This is a difficult decision even if the sizes of the matrices are known before run-time as the performance of underlying networks are dynamic and difficult to predict in Grid environments.

It is also impossible for a dynamic GridRPC system such as GridSolve, which can discover resources and map tasks at run-time, to optimally map the tasks in this application. This is due to the current GridRPC model only permitting a single task to be processed at any time. Therefore, when the system maps the GridRPC task call executing handle h1, it has no knowledge of what

[1] This special string is a GridSolve-specific workaround to enable lazy binding in GridRPC.

tasks are executing in parallel with this task and the computation load of the tasks executing in parallel.

Consider the following scenario - M is initialized to 1000 and N is initialized to 100. Therefore, the computational load of the first task will be far less than that of the second task. In this circumstance, when the system maps the function handle h1, it will map this to the fastest server as this will yield the lowest execution time for this task. Then, when the system maps the function executing handle h2, it will map it to the second fastest server as the fastest server is currently heavily loaded with the first task. This is poor load balancing of computation and will affect the overall performance of the parallel execution of both tasks.

In addition, since tasks are processed individually in the GridRPC model, it is impossible for systems, which implement this model, to know the dependencies between tasks. Since dependencies between tasks are not known and the communication model of GridRPC model is based on the client-server model, bridge communication between remote tasks is forced. With the GridRPC model, this dependent argument would have to be sent from the source task to the destination task via the client, which is two communication steps. This necessity for the client to buffer intermediate data may also cause memory paging on the client. In this application, the third task, h3, is dependent on argument F from the second task h2 and argument C from task h1. In this case, the only way to send F from the server executing h2 and C from the server executing h1 to the server executing h3 is via the client, which is two communication steps. Mapping tasks individually in this application has forced bridge communication and increased the amount of memory used on the client. This will affect the overall volume of communication and may cause paging on the client, which would significantly affect the performance of the application. In addition, since tasks are mapped individually on to a star network, parallelism of remote communication cannot be employed. In this case, if dependencies were known, argument C could be sent from the server executing h1 to the server executing h3 in parallel with computation and communication of task h2 (permitting that task h2 has been assigned a different server than h3).

From this application, it is evident that the potential for higher performance applications would be increased if we could map tasks collectively as a group on to a network, which is fully connected. This is the premise of the SmartGridRPC model.

5. SMARTGRIDRPC PROGRAMMING MODEL AND API

The aim of the SmartGridRPC model is to enhance the GridRPC model by providing functionality for collective mapping of a group of tasks on a fully connected network.

The SmartGridRPC programming model is designed so that when it is implemented it is interoperable with the existing GridRPC implementation (figure 2). Therefore, if any middleware has been extended to be made SmartGridRPC compliant, the application programmer has the option whether their application is implemented for the SmartGridRPC model, where tasks are mapped collectively on to a fully connected network or for the standard GridRPC model, where tasks are mapped individually on to a client-server star network.

In addition, the SmartGridRPC model is designed so that when it is implemented it is incremental to the GridRPC system. Therefore, if the SmartGridRPC model is installed only on the client side, the system will be extended to allow for collective mapping. If the SmartGridRPC model is installed on the client side and on only some of the servers in the

network, the system will be extended to allow for collective mapping on a partially connected network. If it is installed on all servers, the system will be extended to allow for collective mapping on the fully connected network.

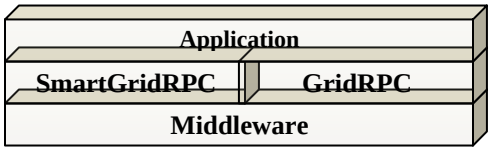

Figure 2: Overview of SmartGridRPC model/API

5.1. SmartGridRPC programming model

The SmartGridRPC model provides an API, which allows the application programmer to specify a block of code, in which a group of GridRPC task calls should be mapped collectively. Then, when the application is run, the specified group of tasks in this block of code is processed collectively and each operation in the GridRPC call is separated and done collectively for all tasks in the group. Namely, all tasks in the group are discovered collectively, mapped collectively and executed collectively on the fully connected network. In the discovery phase, performance models are generated for estimating the execution time of the group of tasks on the fully connected network. In the mapping phase, the performance models are used by the mapping heuristic to generate a mapping solution for the group of tasks. In the execution phase, the group of tasks is executed on the fully connected network according to the mapping solution generated.

In the context of this paper, a performance model is any structure, function, parameter etc., which are used to estimate the execution time of tasks in the distributed environment. The SmartGridRPC performance model refers to performance models, which are used to estimate the time of executing a group of tasks on the fully connected network. The GridRPC performance model refers to performance models, which are used to estimate the execution time of an individual task on a star network. A mapping heuristic is an algorithm, which aims to generate a mapping solution that satisfies a certain criterion, for example, minimum completion time, minimum perturbation etc. The SmartGridRPC mapping heuristics refer to mapping heuristics, which map a group of tasks on to a fully connected network. The GridRPC mapping heuristics refer to mapping heuristics which map an individual task on to a client-server network. Furthermore, a mapping solution is a structure, which outlines how tasks should be executed on the distributed network. The SmartGridRPC mapping solution outlines both a task-to-server mapping of each task in the group to a server in the network and the communication operations between the tasks in the group. The GridRPC mapping solution outlines the server list, which specifies where the called task should be executed, and the backup servers which should execute the task should the execution fail.

The collective mapping of the SmartGridRPC model allows the mapping heuristics to estimate the execution time of more mapping solutions than if these tasks were mapped individually and therefore have higher potential of finding a more optimal solution.

The job of generating the performance models is divided between the different components of GridRPC architecture (i.e. client, server and registry). The components may only be capable of constructing part of the performance model required to estimate the groups' execution time.

Therefore, the registry accumulates these parts from the different components and generates the required performance models.

There are numerous methods for estimating the execution time of the group of tasks on a fully connected network so the implemented performance models are not specified in the SmartGridRPC model. Examples of performance models would be the ones currently implemented in SmartGridSolve, which have extended the performance models used in GridSolve (section 6.3). In the future, SmartGridSolve will implement performance models such as the Functional Performance Model, which is described in [22][23]. Other possible implementations could include the Network Weather Service [24], the MDS directories (Globus, Ninf) [4] and the Historical Trace Manager (GridSolve) [25]. In general, in the SmartGridRPC model, the performance models are used to estimate:

- The execution time of a task on a server.
- The execution time of multiple tasks on a server and the affect the execution each task has on the other (perturbation).
- The communication time of sending inputs and outputs between client and server.
- The communication time of sending inputs and outputs between different servers.

Mapping heuristics implement a certain methodology that uses these performance models to generate a mapping solution, which satisfies a certain criterion. Examples of mapping solutions include the greedy mapping heuristic and the exhaustive mapping heuristics, which have been currently implemented in SmartGridSolve. There has been extensive research done in the area of mapping heuristics [26] so this is not the focus of our study.

The following sections describe the programming model of SmartGridRPC in the circumstance where the performance models are generated on the registry and the group of tasks is mapped by a mapping heuristic on the registry. However, the SmartGridRPC model could have an alternative implementation. These performance models could be generated on the client and the group of tasks could also be mapped by a mapping heuristic on the client. This may be a more suitable model for systems, such as Ninf-G, which have no central daemon like the GridSolve Agent or the DIET Global Agent.

The SmartGridRPC map function separates the GridRPC call operations into three distinct phases so they can be done for all tasks collectively:

- Discovery phase – The registry discovers all the performance models necessary for estimating the execution time of the group of tasks on a fully connected network.
- Mapping phase – The mapping heuristic uses the performance models to generate a mapping solution for the group of tasks.
- Execution phase - The group of tasks is executed on the fully connected network according to the mapping solution.

Register discovery: The servers provide the part of the performance model, which would facilitate the modeling of the execution of its available tasks on the underlying network. This partial model can either be automatically generated by the server or has to be explicitly specified or both. This partial model will be referred to as the *server PM*.

As previously mentioned, the SmartGridRPC model does not specify how to implement the *server PM* as there are many possible implementations. Exactly when the *server's PM* is sent to

the registry is also not specified by the SmartGridRPC model as this would depend on the type of performance model implemented.

But for example, the *server PM* could be sent to the registry upon registration and then updated after a certain event has occurred (i.e. when the CPU load or communication load has changed beyond a certain threshold) or when a certain time interval has elapsed. Or it may be updated during the run-time of the application when actual running times of tasks are used to build the performance model. It is suffice to say that the *server PM* is updated on the registry and is stored there until it is required during the run-time of a client application.

Client application run-time: The client also provides a part of the performance model, which is sent to the registry during the run-time of the client application. This will be referred to as the *client PM*. This part of the performance model is application-specific such as the list of tasks in the group, their order, the dependencies between tasks and the values of the arguments in the calling sequences. In addition, the *client PM* specifies the performance of the client-server links.

In order to determine the parts of the performance model of the group of tasks, which are application-specific, each task, which has been requested to be mapped collectively, will be iterated through twice. On the first iteration, each GridRPC task call is discovered but not executed. This is the *discovery phase*. After all tasks in the group are discovered, the client determines the performance of the client-server links and sends the *client PM* to the registry. The registry then generates the performance models based on the stored *server PM* and the *client PM*. Based on these performance models, the mapping heuristic generates a mapping solution. This is the *mapping phase*. On the second iteration through the group of tasks, each task is then executed according to the mapping solution generated. This is the *execution phase*. This approach of iterating twice through the group tasks to separate the discovery, mapping and execution of tasks into three distinct phases is the basis that allows the SmartGridRPC model to collectively map and then collectively execute a group of tasks.

The run-time map function, *grpc_map*, is part of the SmartGridRPC API and allows the application programmer to specify a group of GridRPC calls to map collectively.

This is done by using a set of parenthesis, which follows the map function, to specify a block of code, which consists of the group of GridRPC task calls that should be mapped collectively: -

```
grpc_map(char * mapping_heuristic_name){
  ...
  //group of GridRPC calls to map collectively
  ...
}
```

When this function is called, the code and GridRPC task calls within the parenthesis of the function are iterated through twice as previously described.

Discovery phase: On the first iteration through the group of tasks, each GridRPC task call within the parenthesis is discovered but not executed so therefore all tasks in the group can be discovered collectively. This is different to the GridRPC model, which only allows a single task to be discovered at any one time. The client can therefore look up and retrieve handles for all tasks in the group at the same time. In addition to sending the handles, the registry also sends back a list of all the servers that can execute each task. The client then determines the performance of the client-server links to the servers in the list. The client may only determine

the performance of some of these links, depending on how many servers are in this list, or may not determine the performance of any of the links if the arguments being sent over the links are small. Exactly how the client determines the performance of these links is not specified by the SmartGridRPC model. This could be implemented using NWS sensors, ping-pong benchmarks, MDS directory or any other conceivable method for determining the performance of communication links.

The client now sends the *client PM* to the registry. The *client PM* specifies the order of tasks in the group, their dependencies and the values of each argument in the calling sequence of each task and the performance of the client-server links. This does not involve sending non-scalar arguments, such as matrices or vectors, but just the pointer value as this will be used to determine the dependencies between tasks. The registry then uses the *server PM* and *client PM* to generate the performance models for estimating the time of executing a group of tasks on the fully connected network. These performance models are then used in the mapping phase to generate a mapping solution.

Mapping Phase: Based on the performance models, the mapping heuristic then produces a mapping solution, which satisfies a certain criterion, for example, minimizing the execution time of tasks. The implemented mapping heuristic is chosen by the application programmer using the SmartGridRPC API.

There is an extensive number of possible mapping heuristics that could be implemented and therefore the mapping heuristics implemented are not bound by the SmartGridRPC model. However, the SmartGridRPC framework allows different mapping heuristics and different performance models to be added and therefore provides an ideal framework for testing and evaluating these performance models and mapping heuristics.

Execution Phase: The execution phase occurs on the second iteration through the group of tasks. In this phase, each GridRPC call is executed according to the mapping solution generated by the mapping heuristic on the previous iteration. The mapping solution not only outlines the task-to-server mapping but also the remote communication operations between the tasks in the group.

5.2. SmartGridRPC: API and semantics

The SmartGridRPC API allows a user to specify a group of tasks that should be mapped collectively on a fully connected network. The SmartGridRPC map function is used for specifying the block of code, which consists of the group of GridRPC tasks calls that is to be mapped collectively.

When the *grpc_map* function is called, the code within its parenthesis will be iterated through twice as previously described in section 5.1. After the first iteration through the group of tasks, the mapping heuristic specified by the parameter “mapping_heuristic_name” of the *grpc_map* function generates a mapping solution.

The mapping solution outlines a task to server mapping and also the communication operations between tasks. These communication operations include:

- Client-server communication
    - Standard GridRPC communication
- Server-server communication
    - Server sends a single argument to another server
- Client broadcasting

    - Client sends a single argument to multiple servers.
- Server broadcasting
    - Server sends a single argument to multiple servers.
- Server caching
    - Server stores an argument locally for future tasks.

As a result, the network may have:

- A fully connected topology - where all the servers are SmartGridSolve enabled servers (SmartServers), which can communicate directly with each other.
- A partially connected topology – where only some of the servers are *SmartServers*, which can communicate directly. The standard servers can only communicate with each other via the client.
- A star connected topology – where all servers are standard servers and they can only communicate with each other via the client.

During the second iteration through the code, the tasks will be executed according to the generated mapping solution.

The SmartGridRPC model also requires a method for identifying code that will be executed on the client. There are many possible approaches, which could be implemented to identify client code. For example, a preprocessor approach could be used to identify the client code transparently. Where the client code cannot be identified, we provide a *grpc_local* function call, which the application programmer can use to explicitly specify client computation: -

```
grpc_map(char * mapping_heuristic_name){

   //reset variables which have been updated
   // during the discovery phase

   grpc_local(list of arguments){
    //code to ignore when generating task graph
   }
   ...
   // group of tasks to map collectively
   ...
}
```

The *grpc_local* function is used to specify the code block that should be ignored during the first iteration through the scope of *grpc_map*. The function is also used to specify remote arguments that are required locally. This information is used to determine when arguments will be sent back to client and also facilitates the generation of the task graph.

Any segment of client code that is not part of the GridRPC API should be identified using this function. There is one exception to this rule, when the client code directly affects any aspect of the task graph. For example, if a variable is updated on the client that determines which remote tasks get executed or the size of inputs/outputs of any task, then the operations on this variable should not be enclosed by the *grpc_local* function. If any variables or structures are updated during the task discovery cycle then they should be restored to their original values before the execution cycle begins.

5.3. SmartGridRPC: A SmartGridRPC application

Table 2 is the SmartGridRPC implementation of the GridRPC application in section 4.3. There is only one extra call required to make this application SmartGridRPC enabled, which is the *grpc_map* function. In this example, the user has specified that all three tasks should be mapped collectively.

Table 2: SmartGridRPC model – Example application.

```
main()
{
    int N=getNSize();
    int M=getMSize();

    double A[N*N], B[N*N], C[N*N];
    double D[M*M], E[M*M], F[M*M], G[M*M
    grpc_function_handle_t h1, h2, h3;
    grpc_session_t s1, s2;
    grpc_initialize(argv[1]);

    /* initialize */
    initMatA(N, A);  initMatB(N, B);
    initMatD(M, D);  initMatD(M, E);


    grpc_function_handle_default(&h1, "mmul/mmul");
    grpc_function_handle_default(&h2, "mmul/mmul");
    grpc_function_handle_default(&h3, "mmul/mmul");

    grpc_map("greedy_map"){
      if(grpc_call_asnc(&h1,&s1,N,A,B,C)!= GRPC_NO_ERROR) {
        fprintf(stderr, "Error in grpc_call\n");
        exit(1);
      }
      if(grpc_call_async(&h2, &s2,M,D,E,F)!=GRPC_NO_ERROR){
        fprintf(stderr, "Error in grpc_call\n");
        exit(1);
      }
      grpc_wait(s1);
      grpc_wait(s2);

      if (grpc_call(&h3,M,C ,F,G) != GRPC_NO_ERROR){
        fprintf(stderr, "Error in grpc_call\n");
        exit(1);
      }
    }

    grpc_function_handle_destruct(&h1);
    grpc_function_handle_destruct(&h2);
    grpc_function_handle_destruct(&h3);
    ...
    grpc_finalize();
}
```

Let us consider the same simple scenario as in section 4.3, where task h2 has a larger computational load than h1 and the underlying network consists of two servers, which have different performances. In this case, since all tasks are mapped together, the SmartGridRPC model will improve the load balancing of computation by assigning task h2 to the faster server and h1 to the slower.

In addition, task h3 has a dependency on the argument F, which is an output of task h2, and argument C, which is an output of task h1. Since the tasks are mapped as a group and therefore dependencies can be considered, this dependency can be mapped on to the virtual link connecting the servers executing both tasks, which will reduce the communication load. Or if the tasks are executing on the same server, then the output can be cached and retrieved from the same server, which would further reduce the communication load and further increase the overall performance of the group of tasks.

Also, since no intermediate results are sent back to the client, the amount of memory utilised on the client will be reduced and this will reduce the risk of paging on the client. This prevention of paging could also considerably reduce the overall execution time of the group of tasks.

In addition, since dependencies are known and the network is fully connected, the remote communication of argument C from server, executing task h1, to server executing task h2, could be done in parallel with the communication and computation of h2.

6. IMPLEMENTING THE GRIDRPC MODEL IN GRIDSOLVE

The GridSolve agent, which is the focal point of the GridSolve system, has the responsibility of performing discovery and mapping of tasks. The GridSolve agent is implementation of the registry entity, which was outlined in the section 4 of the paper.

In order to map a task on the client-server network, the agent must discover performance models, which can be used to estimate the execution time of individual tasks on different servers on the network. These performance models include functions for each task, which calculate the computation/communication load of tasks, and parameters, which specify the dynamic performance of the network. These performance models are sent from each server in the network to the agent before run-time of the client application (Agent discovery).

6.1. GridSolve: Agent discovery

The section outlines the GridSolve implementation of the "Register discovery" part of the GridRPC model outlined in section 4.1. The agent maintains a list of all available servers and their registered tasks. This list is incremented when each new server registers with the agent. In addition, the agent stores performance models required to estimate the execution time of available tasks on the servers. This includes the dynamic performance of each server and functions/parameters, which are used to calculate the computational/communication load of tasks. These performance models are implemented by executing the LINPACK benchmark on each server when they are started, running the CPU load monitor on the server and using descriptions of the task provided by the person that installed the task to generate functions for calculating the computation and communication load of the tasks (figure 3).

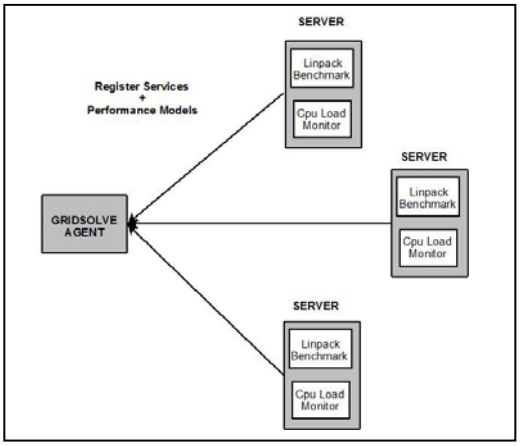

Figure 3: GridSolve – Agent discovery

The server may optionally be configured to maintain a history of execution times and use a non-negative least squares algorithm to predict future execution times. At run-time of the client

application, when each GridRPC task call is invoked, these performance models are used to estimate the execution time of the called task on each server.

6.2. Run-time GridRPC task call

In practice, from the user's perspective the mechanism employed by GridSolve makes the GridRPC task call fairly transparent. However, behind the scenes a typical GridRPC task call involves the following operations:

- The discovery operation.
- The mapping operation.
- The execution operation.

The discovery operation: When the GridRPC call is invoked, the client queries the agent for an appropriate server that can execute the desired function. The agent returns a list of available servers, ranked in order of suitability.

This ranked list is sorted based only on task computation times. Normally, the client would simply submit the service request to the first server on the list, but if specified by the user it is resorted according to its overall computation and communication time. If this is specified, the bandwidth from the client to the top few servers is measured. This is done using a simple 32KB ping-pong benchmark. The time required to do the measurement will depend on the number of servers, which have the requested task, and the bandwidth and latency from the client to those servers. When the data is relatively small, the measurements are not performed because it would take less time to just send the data than it would take to do the measurements. Also, since a given service may be available on many servers, the cost of measuring network speed to all of them could be prohibitive. Therefore, the number of servers to be measured is limited to those with the highest computational performance.

The mapping operation: As previously described, the agent sends a server list, which is ordered according to their estimated computation time.

In GridSolve, there is a number of mapping heuristics, which can be employed to generate the mapping solution. Among the mapping heuristics is the minimum completion time (MCT) mapping heuristic, which bases its execution time on the performance models and the dynamic network performance of each server outlined in section 6.3. Also included are a set of mapping heuristics that rely on the other performance model in GridSolve called the Historical Trace Manager (HTM).

The execution operation: The client attempts to contact the first server from the list. It sends the input data to the server, the server then executes the task on behalf of the client and returns the results. If at any point the execution fails, the client automatically moves down the list of servers.

6.3. GridSolve: Performance models

The performance models of GridSolve are used by the mapping heuristics to estimate the execution time of individual tasks on a client-server network. In GridSolve, the performance models can be used to estimate:

- The execution time of a task on a server.

- The communication time of sending inputs and outputs between client and server.
- The perturbation that one task has on another.

The mapping heuristics use these performance models to estimate the time of different possible task-to-server mapping solutions and choose the mapping solution, which most satisfies a certain criterion.

The performance models of GridSolve specify the dynamic performance of each server and the dynamic performance of the client-server links. They also specify the computation load and communication load of the called task. When a task is called for execution, the computation load and dynamic performance of a server is used to estimate the task's computation time. The communication load and dynamic performance of the client-server links are used to estimate the task's communication time.

The dynamic performance of a server is parameterized by the number of floating point operations per second (flop/sec) that the server can perform. It is obtained by first determining the static performance of each server by running a sequential benchmark on each server (figure 3). This sequential benchmark is the LINPACK benchmark, which is executed on each server when it is started. The benchmark times the execution of a routine, which solves a dense system of linear equations. This benchmark is close to the peak performance rate of the server. There is a "CPU load monitor" on each server, which continually monitors the CPU load. When this CPU load changes beyond a certain threshold or if a certain time interval has elapsed (~5mins), then this CPU load is sent to the agent. To get the dynamic performance (p) of the server, the agent uses this updated CPU load to scale the value for the server's peak performance (P). The dynamic performance of a server is calculated as follows:

$$p = \frac{P * n}{\frac{w}{100} + 1}$$

- where p is the dynamic performance of the server, w is the current CPU load, P is the peak performance (benchmark) of the server and n is the number of processors on the server.

The dynamic performance of a client-server link is parameterized by its bandwidth (bw), which is the number of bytes per second (bytes/sec) that can traverse the link. It is obtained using the ping-pong benchmarks. Using these benchmarks the bandwidth of the link can be calculated as follows:

$$bw = \frac{PING\_PACKET\_SIZE}{PING\_TIME}$$

- where the PING_PACKET_SIZE is 32 KB and PING_TIME is the time it takes to send the packet between the client and server.

In addition, the performance model includes functions for calculating computation and communication load of tasks. These functions are generated from the task description of each task, which is provided by the person who writes or installs a task. They are written in a language called the GridSolve Interface Definition Language (IDL). With this language, the task writer/installer provides a specification of the calling sequence of the task. This specification describes the data type of each argument (integer, float, double etc.), the object type of each argument (scalar, vector, or matrix) and whether each argument is an input, an output or

an input-output. Table 3 shows the IDL description of the DGESV task, which is a LAPACK routine that solves $A * X = B$. The IDL description specifies that the first two arguments of the calling sequence are input scalar integers. The third argument of the calling sequence is an input-output, which is a matrix of doubles. The fourth argument is an output scalar argument. The fifth argument is an output vector of integers and the sixth is an input-output matrix of doubles. The seventh is a scalar integer input and the eight is a scalar output integer. This specification of the calling sequence is used to generate functions for calculating both the computation load and the communication load of the task.

Table 3: IDL specification of DGESV task

```
SUBROUTINE dgesv(
  IN int N,
  IN int NRHS,
  INOUT double A[LDA][N],
  IN int LDA,
  OUT int IPIV[N],
  INOUT double B[LDB][NRHS],
  IN int LDB,
  OUT int INFO)

"This solves Ax=b using LAPACK"
LANGUAGE = "FORTRAN"
LIBS = "$(LAPACK_LIBS) $(BLAS_LIBS)"
COMPLEXITY = "2.0*pow(N,3.0)*(double)NRHS"
MAJOR="COLUMN
```

Included in the IDL description of the task is a "string" formula that is used in conjunction with the specification of the calling sequence to generate a function for calculating the computation load of a task. This formula is denoted in the IDL file as the "COMPLEXITY" parameter. The string formula for the DGESV task is:

$$dgesv_{flop} = 2 * N^3 * NRHS$$

This formula in conjunction with the specified calling sequence in table 3 generates a function that describes the computational load as a multiplication of 2 by the first argument cubed multiplied by the second argument of the calling sequence.

At run-time, when the values of these arguments are known, this function can be used to calculate the computation load of task. The computation load is measured in the number of floating operations (flop), which the task will execute. An example of the calling sequence of the DGESV task could be as follows:

grpc_call(&degesv_handle,400,100, A, 800, IPIV, B, 400,INFO)

For this calling sequence the computation load would be:

$$dgesv_{flop} = 2 * 400^3 * 100 = 12800 * 10^6 Flop = 12800 MFlop$$

The communication load of a task can be calculated using the following formulas in conjunction with the specified calling sequence in the IDL specification:

$$arg\_size\ (matrix) = \quad rows * cols * get\_elem\_size(arch, DATA\_TYPE)$$
$$arg\_size\ (vector) = \quad rows * get\_elem\_size(arch, DATA\_TYPE)$$

In these formulas, the DATA_TYPE variable specifies whether the argument type is a double, integer, float etc. And the rows and cols variables are the dimensions of the matrix/vector. The get_elem_size function returns the size of bytes of the specified DATA_TYPE (double, integer etc.).

The formula for calculating matrix argument size in conjunction with the specification of the calling sequence in table 3 would generate a function that outlines that the communication load of *argument A* of the DGESV task can be calculated by multiplying the fourth argument (LDA) in the calling sequence by the first argument (N) in the calling sequence by the size of a double (e.g. 8 bytes). At run-time, when the values of these arguments are known, this function can be used to calculate the communication load of argument A of the DGESV task.

However, in some instances, the platform of the sending machine must be known to determine this communication load. One of the problems with C and C++ is that the built in data types such as *int* and *long int* are platform dependent. There is nothing in the standard to say how many bytes each data type occupies beyond some basic ordering. For example, long int must use at least as many bytes as int (but could be the same). Table 4 outlines the number of bytes of different data types on different platforms.

Table 4: Size of datatypes on different platforms

OS	ARCH	Size of int	Size of long int	Size of double
LINUX	x86	4 bytes	4 bytes	8 bytes
LINUX	x86-64	4 bytes	8 bytes	8 bytes
Windows	x86	4 bytes	4 bytes	8 bytes
Windows	x86-64	4 bytes	4 bytes	8 bytes
MAC OS X	x86	4 bytes	4 bytes	8 bytes
MAC OS X	x86-64	4 bytes	8 bytes	8 bytes

For this reason the *get_elem_size* function also takes an architecture identifier as a parameter. For the calling sequence outlined previously for the DGESV task, the communication load of non-scalar arguments on a 32bit Intel machine running LINUX OS would be:

$$A_{bytes} = LDA * N * get\_elem\_size(LINUX_{x86}, \quad DOUBLE)$$

$$A_{bytes} = 800 * 400 * 8 = 256 * 10^4 Bytes = 2.44\ MBytes$$

$$IPIV_{bytes} = N * get\_elem\_size(LINUX_{x86}, \quad INTEGER)$$

$$IPIV_{bytes} = 400 * 4\ Bytes = 1600 Bytes = 0.00153\ MBytes$$

$$B_{bytes} = LDB * NRHS * get\_elem\_size(LINUX_{x86}, \quad DOUBLE)$$

$$B_{bytes} = 400 * 100 * 8 = 320000\ Bytes = 0.30518\ MBytes$$

From these performance models, it is possible to estimate both the communication time and computation time of individual tasks on the client-server network. These performance models are used by the mapping heuristics to generate a mapping solution.

6.4. GridSolve: Mapping heuristics

There have been several mapping heuristics implemented in GridSolve. Each task is mapped when it is called for execution and therefore each task is mapped individually on the client-server network. The following mapping heuristics have been implemented:

- Minimum Completion Time.
- HTM - Minimum Completion Time.
- HTM - Minimum Perturbation.
- HTM - Minimum Sum Flow.
- HTM – Minimum Length.

The Minimum Completion Time (MCT) maps the individual task based on the performance models described in 6.3.

All the HTM mapping heuristics generate mapping solutions based on the Historical Trace Manager (HTM) performance model. When a new task arrives, the HTM simulates the execution of the task on each server. Using the HTM information, the heuristic has an estimation of finishing time of each task running on each server. This is used to consider the perturbation that tasks induce on each other and compute the 'best' server according to the main objective of that heuristic.

When a task has completed the server sends a message to the agent that the task has completed and this information is used by the HTM to correct what has been simulated and improve the quality of future predictions.

7. SMARTGRIDSOLVE: IMPLEMENTING SMARGRIDRPC IN GRIDSOLVE

7.1. SmartGridSolve: Agent discovery

This section presents the SmartGridSolve implementation of the "register discovery" part of SmartGridRPC model outlined in section 5.1. In addition to registering services, the servers also send the *server PM*. The *server PM* makes up part of the performance model used for estimating the execution time of the server's available tasks on the fully connected network. This along with the *client PM* is used to generate a performance model, which is used by the mapping heuristics to produce mapping solutions.

Currently, the *server PM* of SmartGridSolve extends that of GridSolve, which comprises of functions for calculating the computation load and communication load and parameters for calculating the dynamic performance of the servers and client-server links. This is described in section 6.3.

However, the network discovery of GridSolve is extended to also discover the dynamic performance of each link connecting *SmartServers*. These are those servers, which can communicate directly with each other or store/receive data from their local cache. The dynamic

performance of the server-server links are taken periodically using the same 32KB ping-pong technique used by GridSolve.

To achieve backward compatibility and to give server administrators full control over how the server operates a server, which has the SmartGridSolve extension enabled, may be also started as a standard GridSolve server.

As a result, the network may have:

- A fully connected topology.
- A partially connected topology.
- A star connected topology.

Also to minimize the volume of data transferred around the network, each *SmartServer* is given an ID. Each *SmartServer* then only sends ping-pong messages to those *SmartServers* that have an id that is less than their own. This prevents the performance of the same communication link being measured twice. Once determined, these values are sent to the agent to update the *server PM*. The *server PM* is stored on the registry and updated either periodically (every 5 minutes) or when the CPU load monitor records a change, which exceeds a certain threshold. This *server PM* is then used to generate the performance models during the run-time of a client application.

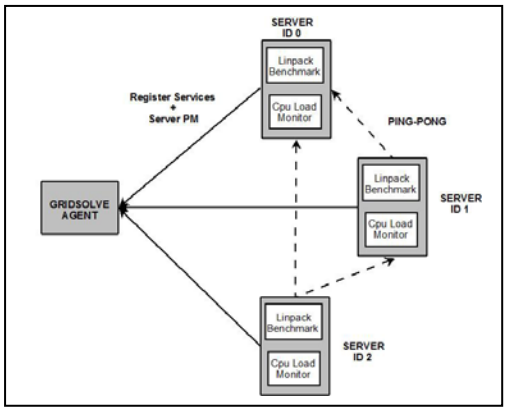

Figure 4: SmartGridSolve - Agent discovery

7.2. Run-time of client application

This section presents the SmartGridSolve implementation of the "Client application run-time" part of SmartGridRPC model outlined in section 5.1. Each phase of the SmartGridRPC run-time map function (*grpc_map*) will be described.

Discovery phase: On the first iteration through the group of tasks, each GridRPC task call (*grpc_call*) within the parenthesis is discovered but not executed. This involves discovering the name of each task and the calling sequence of each task, which involves discovering the pointers to the non-scalar arguments (such as matrices, vectors etc.) and the values of the scalar arguments.

After the first iteration through the group, the client contacts the agent and looks up the group of tasks, which involves sending the agent a list of the task names.

The agent then creates a handle for each task. The agent sends back the group of handles, one for each task. In addition, for each handle it sends a list of servers, which can execute each task.

The client then uses the list of servers to perform the ping-pong benchmark on each of the links from the client to each server that can execute a task in the group of tasks. Subsequent to this, the client will send the *client PM*, which is a structure that specifies application-specific information such as the list of tasks, the calling sequence and the dependencies between the tasks. In addition it specifies the performances of each client-server link.

The agent can now generate all the performance models necessary for estimating the execution time of the group of tasks on the fully or partially connected network. In SmartGridSolve, these performance models consist of a task graph, a network graph and functions for estimating computation and communication times.

The task graph specifies the order of tasks, their synchronisation (whether they are executed in sequence or parallel), the dependencies between tasks, the load of computation and communication of each task in the group.

The network graph specifies the performance of each server in the network and the communication links of the fully connected, partially connected or star network. These performance models will be used by the mapping heuristics in the mapping phase to generate a mapping solution for the group of tasks.

Mapping Phase: The mapping heuristic produces a mapping solution graph based on the task graph, the network graph and the functions for estimating computation and communication time. The mapping heuristics currently implemented in SmartGridSolve are:

- Exhaustive mapping heuristic.
- Random walk mapping heuristic.
- Greedy mapping heuristic.

The mapping solution generated by these heuristics is then used in the execution phase to determine how the group of tasks should be executed on the network.

Execution Phase: This execution phase occurs on the second iteration through the group of tasks. In this phase, each called GridRPC call is executed according to the mapping solution generated by the mapping heuristic. The mapping solution not only outlines the task-to-server mapping but also the communication operations between the tasks in the group.

In addition to the standard GridRPC communication, the mapping solution can use the following communication operations:

- Server-server communication.
- Client broadcasting.
- Server broadcasting.
- Server caching.

7.3. SmartGridSolve performance models

This section presents the performance models, which are currently implemented in SmartGridSolve. The performance models are used by the mapping heuristics to estimate the execution times of different mappings of the group of tasks on the network. This involves both estimating the computation time of tasks of the application on the servers of the network and also estimating the communication time of sending inputs/outputs over the network. The accuracy of these performance models affects the ability of the mapping heuristics to generate optimal mapping solutions.

7.3.1. Network graph

The network graph is a representation of the performance of the servers and communication links of the fully connected, partially connected or star network. If SmartGridSolve is installed only on the client side, this structure will represent a star network where no servers can communicate directly. With this network topology, the application programmer may only benefit from improved mapping of tasks to servers. If some of the servers are *SmartServers* then this structure will represent a partially connected network. With this network topology, the application programmer may also benefit from improved mapping of communication. If all servers are *SmartServers*, the network will be fully connected. With this network topology, the application programmer will benefit from the full potential of improved mapping of communication.

The graph specifies the performance of each server and also the performance of each link connecting it with the client. Where there are two or more servers in the network that are *SmartServers,* the graph will include links which specify the performance of the link between these servers.

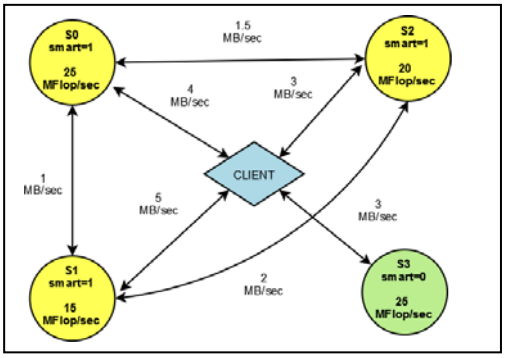

Figure 5: SmartGridSolve – The network graph

Figure 5 illustrates a network graph, which represents three SmartServers and one standard server. Each circle node in the graph represents a server and is weighted by its dynamic performance, which is measured in floating point operations per second (flop/sec). The single diamond shaped node represents the client. Each link connecting nodes represents a "virtual communication link" and is weighted by its dynamic performance (bandwidth), which is measured in the number of bytes per second (bytes/sec), which can traverse the link. All four servers have links connecting them to the client but only *SmartServers* have links connecting them to other *SmartServers*. In figure 5, the servers S0, S1 and S2 are *SmartServers* and therefore have links connecting them with each other and also to the client. Server S3 is a server, which was started without direct communication enabled. If GridSolve has been compiled with the SmartGridSolve extension, a server administrator has the option whether the server is started with direct communication enabled or disabled.

The performance of the servers and communication links are calculated using the equations outlined in section 6.3.

7.3.2. Task graph

The task graph is the representation of the mapped group of tasks. The task graph specifies the order of tasks, their synchronisation (whether they are executed in sequence or parallel), the dependencies between tasks, the load of computation and communication of each task in the group.

Figure 6 illustrates a task graph, which represents 5 tasks, where *task 0* and *task 1* are executed in sequence and then *task 2*, *task 3* and *task 4* are executed in parallel. The task graph has three sets of nodes, the task nodes, the client node and the argument nodes. Each task node is represented by a rectangle node and is weighted by its computation load (flop). Each input and output non-scalar argument (matrix, vector etc.) is represented by a circle shaped node and is weighted by communication load (bytes).

The functions for calculating the computation load of each task in the group are generated using the formulas specified by the person that wrote or installed the task in conjunction with the specification of the tasks calling sequence in the IDL description (table 3). This is part of the *server PM*, which is sent from each server to the agent prior to the execution of the client application.

Then at run-time, the calling sequence of each task in the group is discovered collectively and these calling sequences are sent as part of the *client PM* to the agent. The functions of the *server PM* and the calling sequences of the *client PM* can be used to determine the computation load of each task in the group.

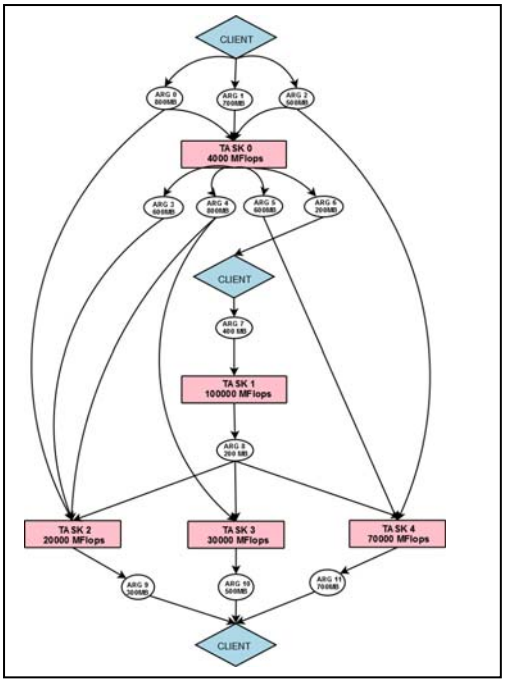

Figure 6: The task graph

The functions for calculating the communication load of each non-scalar argument in the group are generated using the functions for calculating argument sizes in section 6.3 in

conjunction with information on the tasks calling sequence in the IDL description (table 3). This is part of the *server PM*.

At run-time, the calling sequences of the tasks in the group are discovered collectively and are sent to the agent as part of the *client PM*. Then, the communication load functions of the *server PM* and the calling sequence of the *client PM* are used to determine the communication load of each non-scalar argument in the group.

The dependencies between tasks are determined by examining the pointers of non-scalar arguments of the calling sequence of each task (which is outlined in the *client PM)* and using the IDL description (which is outlined in the *server PM*) to determine whether they are inputs or outputs.

The links in the graph represent the data-flow between tasks. There are two types of data-flow dependencies, the input data-flow dependency and output data-flow dependency.

Input dependencies occur when a task has a dependency on an input of another task. This is specified in the task graph by a link from an input argument node of one task pointing to another task node. If multiple tasks require the same input argument then a link will emanate from this argument node to each dependant task node. In this case the mapping heuristics can choose a mapping solution, which broadcasts the input argument of the source task from the client to each of the servers of the destination tasks.

Output dependencies occur when a task has a dependency on an output of another task. This is specified in the task graph by a link from an output argument node of one task pointing to another task node. If multiple tasks require the same output argument, then a link will emanate from this argument to each dependant task. In this case, mapping heuristics can choose a mapping solution, which broadcasts the argument from the source task to each of the servers of the destination tasks.

7.4. Mapping heuristics

The problem of optimally mapping a group tasks on to a fully connected network has been proved to be NP-complete, thus requiring the development of heuristic techniques for practical usage. SmartGridSolve currently has a number of mapping heuristics implemented to optimize searching through possible mapping solutions (the solution space).

In order to reduce mapping times, the current mapping heuristics implemented in SmartGridSolve do not consider all possible mapping solutions due to large number of possible solutions in the solution space. Instead, for each task-to-server mapping, the mapping heuristics consider only a single communication scheme, the communication scheme that has the lowest number of communication steps. In the future with improved optimization heuristics a greater number of communication schemes could be considered.

When considering only a single communication scheme, the number of possible mapping solutions can be significant even when there are only a few tasks and servers. For a given set of tasks and servers, the number of task-to-server mappings when only a single communication scheme is considered will be:

$$task.mappings = servers^{tasks}$$

For each task-to-server mapping, the mapping heuristic can consider the following number of communication schemes.

$$comm.schemes = \frac{(servers-1)!}{2} * num.dep$$

This represents the number of communication schemes when there are more than one server in the network (*servers*) and at least one dependency in the group of tasks (*num.dep*). Otherwise, there is only a single communication scheme that needs to be considered.

Therefore, the total number of solutions for a given group of tasks with one or more dependencies on a fully connected network with more than one server will be:

$$solutions = task.mappings + \sum_{i=0}^{task.mappings} comm.schemes_i$$

This sums the number of task-to-server mappings and the number of possible communication schemes for each task to server mapping.

To demonstrate the significant increase when considering multiple communication schemes, consider the following example. A group of 10 tasks, which have 10 dependencies between the tasks, are mapped on to a network of 5 servers. In this circumstance, the number of possible task-to server mappings would be nearly 1 billion when considering only a single communication scheme. Then, when considering the number of possible communication schemes, the number of possible mapping solutions would increase to in excess of 1 trillion. Considering that it roughly takes one second to simulate 10,000 solutions on Pentium 4 3.2 GHz CPU it is unfeasible to consider multiple communication schemes for each task-to-server mapping.

Therefore, for each possible task-to-server mapping the current mapping heuristics implemented in SmartGridSolve consider a single communication scheme. This is the communication scheme, which minimizes the number of communication steps between tasks.

If there is a dependency between two tasks, which have been assigned to different *SmartServers*, the only communication path that is considered is the one, which connects both servers directly. If there is a dependency between two tasks and both have been assigned the same *SmartServer*, then the communication scheme will outline that this output should be cached locally on the server. If there is a dependency between two tasks and either has been assigned to a server that is not a *SmartServer*, the only communication path that will be considered is the one that connects both servers via the client.

Therefore, the mapping heuristics of SmartGridSolve will generate at most $servers^{tasks}$ mapping solutions graphs.

7.4.1. Mapping solution graph

A mapping solution graph is a structure, which outlines both the task-to-server mapping of the group of tasks and the communication scheme between the tasks in the group. In addition, the mapping solution graph outlines the estimated computation time of each task on their assigned server and the estimated communication time of each task dependency on their assigned communication path.

Since only a single communication scheme is considered, a mapping solution can be generated from the task graph, the network graph and only a single task-to-server assignment vector (table 5). The index of each vector element corresponds to a task in the task graph and each vector element corresponds to a server in the network graph.

The task-to-server vector in table 5 specifies that *task 0* of the task graph is assigned to *server 0* of the network graph, *task 1* is assigned *server 0, task 2* is assigned to *server 1, task 3* is assigned to *server 2* and *task 4* is assigned to *server 0*.

Table 5 - Task-to-server assignment vector

0	0	1	2	0

Figure 7 shows a mapping solution graph for the task-to-server assignment vector in table 5 and the task graph in figure 6 and the network graph in figure 5.

When the mapping solution graph for this task-to-server vector is generated, the communication scheme chosen would be the one which minimizes the number of communication steps between the tasks. The communication scheme in figure 7 minimizes the communication steps for the given assignment vector in table 5.

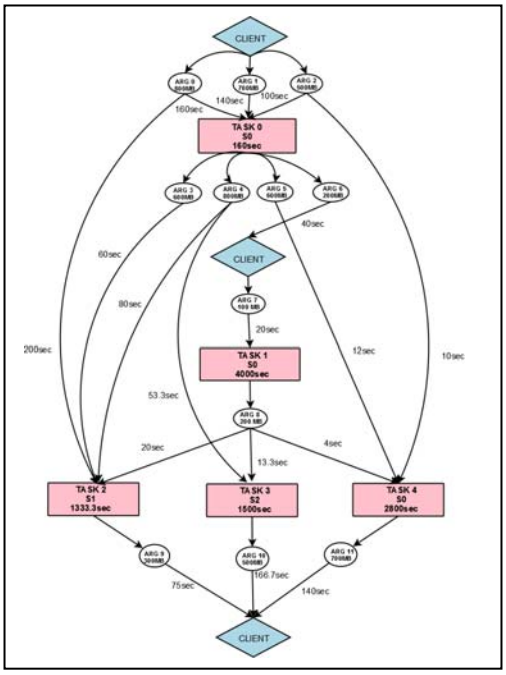

Figure 7: The mapping solution graph

This communication scheme implements each type of communication transactions, which can be employed in the SmartGridRPC model:

- Direct server-server communication.
- Client broadcasting.
- Server broadcasting.
- Server caching of inputs.

- Server caching of outputs.

The mapping solution outlines **direct server-server communication** of *argument 3* from *server 0* to *server 1* after *task 0* has executed. This argument is subsequently used on *server 1* for the execution of *task 2*. It outlines **server broadcast communication** of *argument 4* from *server 0* to *server 1* and *server 2* after *task 0* has executed. This argument is subsequently used on *server 1* for execution of *task 2* and on *server 2* for the execution of *task 3*. It outlines **client broadcast communication** of *argument 0* from the client to *server 0* and *server 1* before the execution of *task 0*. It outlines the **server caching of input** *argument 2* on *server 0* before the execution of *task 0* as this argument. This argument is subsequently used on the same server by *task 4*. It outlines the **server caching of output** *argument 5* on *server 0* after the execution of *task 0*. This argument is subsequently used on the same server by *task 4*.

The estimated time of each of these remote communication transactions is calculated by dividing the communication load of the argument outlined in the task graph in figure 6 and the bandwidth of the communication link outlined in the network graph in figure 5. For example the direct server-server communication of argument 3 is estimated to take 60 seconds, which is calculated by dividing the communication load of 600MB by the link speed, which is 10MB/sec.

The estimated time of the caching transactions are based on a naïve assumption that disk speed is 50MB/sec. For example, the caching of input argument 2 takes 10 seconds, which is calculated by dividing the argument size of this argument which is outlined in the task graph which is 500MB by the disk speed which is 50MB/sec. In the future, benchmarks could be used to determine a more accurate disk speed for each machine.

The estimated time for computation is calculated by dividing the computation load of the task, outlined in the task graph, and the server performance speed, outlined in the network graph. For example, the computation time for *task 0* is 160seconds, which is calculated by dividing the computation load of 4000MFlops by the server speed, which is 25MFlops/second.

However, not every task in the group contributes to the overall execution time of the group of tasks. Parallelism of computation has been employed between tasks 2, task 3 and task 4 and therefore only the task that takes the longest time of all three contributes to the total execution time of all three. In this mapping solution, the time saved due to parallelism of computation is:

$$par.comp.time = t(t2, s1) + t(t3, s2) = 1333.3 + 1500 = 2833.3\ seconds$$

The SmartGridRPC model also permits the parallelism of communication. Any communication may be done in parallel with other computation/communication in the group. This is advantageous when there is a dependency between two tasks and the destination task is not executed in parallel or immediately after the source tasks. In this case, the dependent data can be sent to the destination task in parallel with any computation or communication on any other machine (client and servers), which happens in the intervening time.

For example, each of the communication transactions from *server 0* after the execution of *task 0* can be done in parallel with other computation and communication. This is because the tasks that require the arguments, *task 2*, *task 3* and *task 4*- are not executed in parallel or immediately after *task 0*. Therefore, these communication transactions can be done in parallel with the computation of *task 1* on *server 0* or any computation on the client.

Moreover, broadcast communication from the client can also be done in parallel with other computation/communication that happens in the intervening time. The sending of *argument 0* from the client to *server 1* can be done in parallel with:

- The computation of *task 0* on *server 0.*
- All the communication transactions from *server 0* that happen after *task 0* has executed.
- The computation of *task 1* on *server 0.*
- The broadcast of *argument 6* after the execution of task 1.
- Any computation on the client which happens in the intervening time.

Therefore, the time saved due parallelism of communication will be:

$$par.comm.time = t(cl \rightarrow a0 \rightarrow s1) + t(s0 \rightarrow a2 \rightarrow s0) + t(s0 \rightarrow a0 \rightarrow s1) + t(s0 \rightarrow a4 \rightarrow s1) + t(s0 \rightarrow a4 \rightarrow s2) + t(s0 \rightarrow a5 \rightarrow s0)$$

$$par.comm.time = 200 + 10 + 60 + 80 + 53.3 + 12 = 415.3 seconds$$

$$= 415.3\ seconds$$

In addition to specifying a more advantageous communication scheme, the mapping solution will outline a more advantageous computation scheme (i.e. task-to-server mapping). Since the mapping heuristics can consider all tasks in the group collectively, it can better distribute the load of parallel computation over the servers. Because the computations of all tasks in the group are considered collectively, the mapping heuristic is able to balance the load of the computation of the three parallel tasks. It therefore assigns task 4, which has the highest computation load, to the fastest sever (*server 0*) and *task 2*, which has the lowest computation load, to the second slowest server *(server 1)*. If these were mapped individually in the GridRPC model, they could be mapped in reverse order as individual mapping gives priority to tasks in the order they are mapped.

In this example, the amount of time saved by employing parallelism of computation is $2833.3\ seconds$ and the amount of time saved by employing parallelism of communication is 415.5 seconds.

The time saved due to this parallelism of computation and communication does not contribute to the overall group time and therefore would not be included in the calculation for the total execution time for the group.

$$group.time = t(cl \rightarrow a0 \rightarrow s0) + t(cl \rightarrow a1 \rightarrow s0) + t(cl \rightarrow a2 \rightarrow s0) + t(t0, s0) + t(s0 \rightarrow a6 \rightarrow cl) + t(cl \rightarrow a7 \rightarrow s0) + t(t1, s0) + t(s0 \rightarrow a8 \rightarrow s1) + t(s0 \rightarrow a8 \rightarrow s2) + t(s0 \rightarrow a8 \rightarrow s0) + t(t4, s0) + t(s1 \rightarrow a9 \rightarrow cl) + t(s2 \rightarrow a10 \rightarrow cl) + t(s0 \rightarrow a11 \rightarrow cl)$$

$$group.time = 160 + 140 + 100 + 160 + 40 + 20 + 4000 + 20 + 13.3 + 4 + 2800 + 75 + 166.7 + 140$$

$$group.time = 7839\ seconds$$

This example has outlined that the mapping heuristics of the SmartGridRPC model have more potential of finding a better mapping solution due to collective mapping and employing the SmartGridRPC communication model, which permits parallel remote communication.

7.5. Communication model

The communication model of SmartGridSolve is based on the fully connected network. This extends the GridRPC communication model, so that in addition to client-server communication, the following communication transactions can be employed:

- Server-server communication
    - Server sends a single argument to another server.
- Server broadcasting
    - Server sends a single argument to multiple servers.
- Client broadcasting
    - Client sends a single argument to multiple servers.
- Server caching
    - Server stores an argument locally for future tasks.

To apply a communication scheme, which employs any of these transactions, the client and the servers must be able to identify where to send and receive arguments of each task. To achieve this functionality, the communication scheme of the group of tasks is stored in communication structures, which specify the communication required for each task in the group. When each task is called for execution, a communication structure is created for that task, which is based on the mapping solution outlined by the mapping heuristic. They are subsequently used by the client to determine where to send the inputs of each task and where to receive outputs of each task. In addition, the client sends the communication structure to other servers if they are involved in any remote communication. The servers use the structure to determine whether to send their inputs/outputs to remote destinations, to cache them locally or to send them back to the client. If arguments are sent remotely, the structure specifies which servers to send it to and the filename of where arguments should be stored. If the argument is received remotely it specifies the filename where the argument should be read from. These filenames are unique for each argument that is sent remotely.

To demonstrate how the communication model of SmartGridSolve operates using these communication structures, we will consider what communication operations occur if task 0 of the mapping solution in figure 7 was called for execution.

Firstly, the communication transactions, which are initiated before the execution of task 0, will be described. These communication transactions are illustrated in figure 8. When task 0 is called for execution, the client generates a communication structure for this task based on the mapping solution. The client then interprets this communication structure, which specifies that argument 0, argument 1 and argument 2 should be sent to server 0. In addition to sending these arguments to the server, the client also sends the communication structure.

The server interprets it and deciphers what to do with the inputs arguments prior to the execution of the task and the output arguments after the execution of the task. In this case, it outlines that input argument 2 should be cached locally in a specified file as it is required by task 4. This operation is done asynchronously, which means that remote computation/communication (on the client or other server) can be done in parallel with this operation.

The communication structure also outlines that argument 0 should be sent to server 1. This is again done asynchronously and so computation on the client or communication/computation on any other server can be done in parallel with this communication. In addition to sending the

argument, the client also sends the communication structure for this argument, which outlines that the argument should be stored locally in the file specified.

It should be noted that arguments can also be stored to memory and it is the server administrator's responsibility to choose which method of storage is implemented on the server. If this is the case, every argument is stored in a buffer/array, which is given a unique identifier similar to that of the filename if it was stored in cache.

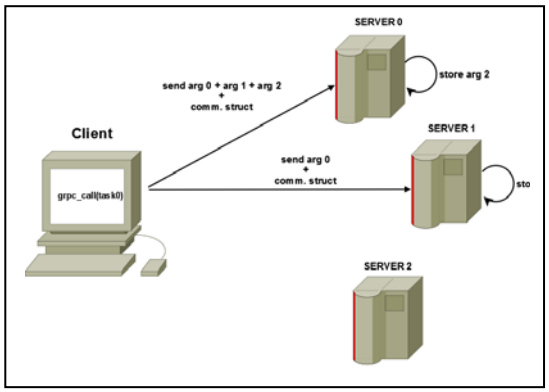

Figure 8: Communication transactions of task 0 of the mapping solution in figure 7, which happen prior to the execution of task 0.

After task 0 has finished executing, the communication structure is used by the server to determine what should be done with the output arguments (figure 9).

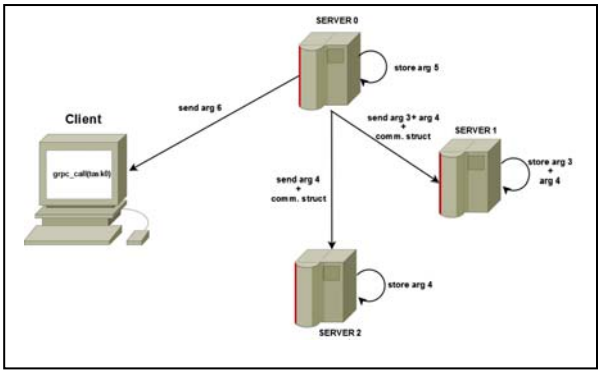

Figure 9: Communication transactions of task 0 of the mapping solution in figure 7, which happen subsequent to the execution of task 0.

The communication structure would outline that argument 5 should be cached locally in a specified file as it is required by task 4. In addition, argument 3 and argument 4 are sent to server 1 and argument 4 is sent to server 2. In these transactions, the communication structure is also sent so the destination servers know the files names to store the arguments.

Once again, these remote transactions happen asynchronously and therefore if there is any computation/communication on any machine (client or server), then this will be done in parallel with this communication.

When the destination tasks, which require these remote arguments, are called for execution, the client will send a communication structure outlining that arguments should be received locally from a file and it will specify the file name where the argument is stored.

7.6. Fault tolerance

The *grpc_map_ft* function in SmartGridSolve is a fault tolerant version of the *grpc_map* function:

```
grpc_map_ft(char * mapping_heuristic_name){
  ...
  // group of tasks to map collectively
  ...

}
```

This is the same as *grpc_map* function, except that the mapping solution generated does not implement server-server communication. The mapping solution outlines a task to server mapping and a communication scheme, which only implements communication between client and server. The communication scheme may implement:

- Client-server communication
    - Standard GridRPC communication.
- Client broadcasting
    - Client sends a single argument to multiple servers.

If any server that is part of the mapping solution fails, the tasks mapped to that server will be mapped to the next server, which is estimated to give lowest execution time for that task.

Although, no direct server communication or caching is implemented when this function is called, the performance of a group of tasks can be increased due to improved load balancing of computation and client broadcast.

In the future, we plan to introduce a fault-tolerant method of mapping a group of tasks, which will remove this restriction on remote communication.

8. SMARTGRIDRPC BENCHMARK APPLICATION: THE EVOLUTION OF A CLUSTER OF GALAXIES

A typical numerical simulation needs a lot of computational power and memory footprint to solve a physical problem with a high accuracy. A single hardware platform that has enough computational power and memory to handle problems of high complexity is not easy to access. Grid computing provides an easy way to gather computational resources, whether local or geographically separated, that can be pooled together to solve large problems.

A scientific application that obviously benefits from the use of GridRPC consists of tasks with high computational loads and low communication loads. These applications, which are the best suited to run on a Grid environment, are not representative of many real-life scientific applications. Unfortunately they are typically chosen, or artificially created, to test and show the

performance of GridRPC middleware systems. We believe that to justify the use of GridRPC for a wide range of applications, we should not use an extremely suitable application as a benchmark but a real life application that shows the eventual limits and benefits of the GridRPC middleware systems tested.

In this section, we present Hydropad [27][28], a real-life astrophysics application that simulates the evolution of clusters of galaxies in the Universe. This application is composed of tasks that have a balanced ratio between computation and communication. Hydropad requires high processing resources because it has to simulate an area comparable to the dimension of the Universe.

The cosmological model, which this application is based on, has the assumption that the universe is composed of two different kinds of matter. The first is baryonic matter, which is directly observed and forms all bright objects. The second is dark matter, which is theorised to account for most of the gravitational mass in the Universe. The evolution of this system can only be described by treating both components at the same time, looking at all of their internal processes, while their mutual interaction is regulated by a gravitational component. Figure 10 shows an example of a typical output generated by Hydropad.

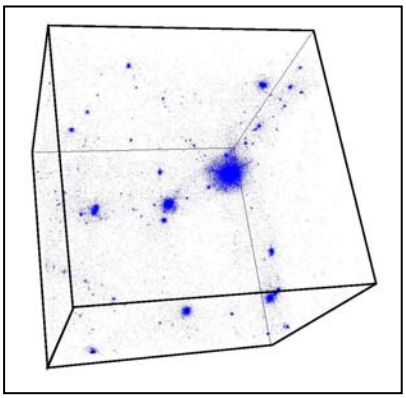

Figure 10: Example of Hydropad output

The dark matter computation can be simulated using N-Body methods [29]. This method utilises the interactions between a large number, Np, of collision-less particles. These particles, subjected to gravitational forces, can simulate the process of the formation of galaxies. The accuracy of this simulation depends on the quantity of particles used. Hydropad utilises a Particle-Mesh (PM) N-Body algorithm, which has a linear computational cost and depends on the number of particles, O(Np). In the first part this method transforms the particles, through an interpolation, into a grid of density values. Afterwards the gravitational potential is calculated from this density grid. In the last part the particles are moved depending on the gravitational forces of the cell where they were located.

The baryonic matter computation utilises a Piecewise-Parabolic-Method (PPM) Hydrodynamic algorithm [30]. This is a higher order method for solving partial differential equations. PPM reproduces the formation of pressure forces and the heating and cooling processes generated by the baryonic component during the formation of galaxies. For each time step of the evolution, the fluid quantities of the baryonic matter are estimated over the cells of the grid by using the gravitational potential. The density of this matter is then retrieved and used to calculate the gravitational forces for the next time step. The accuracy of this method depends on the

number of cells of the grid used, Ng, and its computational cost is linear O(Ng). The application computes the gravitational forces, needed in the two previous algorithms, by using the Fast-Fourier-Transform (FFT) method to solve the Poisson equation. This method has a computational cost of O(Ng logNg). All the data, used by the different components in Hydropad, are stored and manipulated in three-dimensional grid-like structures. In the application, the uniformity of these base structures permits easy interaction between the different methods.

Figure 11 shows the workflow of the Hydropad application. It is composed of two parts: the initialisation of the data and the main computation. The main computation of the application consists of a number of iterations that simulate the discrete time steps used to represent the evolution of the universe from the Big Bang to present time. This part consists of three tasks: the gravitational task (FFT method), the dark matter task (PM method) and the baryonic matter task (PPM method). For every time step in the evolution of the universe, the gravitational task generates the gravitational field using the density of the two matters calculated in the previous time step. Hence the dark and baryonic tasks use the newly produced gravitational forces to calculate the movement of the matter that happens during this time step. Then the new density is generated and the lapse of time in the next time step is calculated from it. It is possible to see in figure 11 that the dark matter task and baryonic matter task are independent of each other.

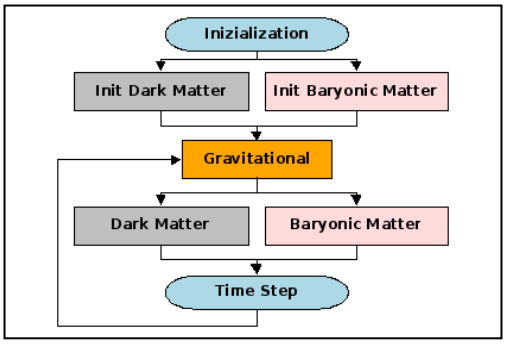

Figure 11: Overview of the Hydropad application

The initialisation part is also divided in two independent tasks. The main characteristic of dark matter initialisation is that the output data is generated by the external application *grafic*, a module of the package COSMICS [31]. *Grafic*, given the initial parameters as an input, generates the position and velocity of the particles that will be used in the N-Body method. The output data is stored in two files which information has to be read by the application during the initialisation part. Like the main application, *grafic* has a high memory footprint.

An important characteristic of Hydropad is the difference in computational and memory load of its tasks. Despite both algorithms being linear, the computational load of the baryonic matter component is far greater than the dark matter one, Cbm >> Cdm, when the number of particles is equal to the number of cells in the grid, Np = Ng. Furthermore, the quantity of data used by the dark matter computation is greater than the baryonic matter one, Ddm >> Dbm.

As previously indicated, Hydropad utilises three dimensional grid structures to represent the data. In the application code, these grids are represented as vectors. In the case of the dark matter component, the application stores the position and velocity in three vectors for each particle, one for each dimension. The size of these vectors depends on the number of particles, Np, chosen to run on the simulation. For the gravitational and baryonic components, the different

physical variables, such as force or pressure, are stored in vectors, with the size depending on the given number of grid cells, Ng. In a typical simulation the number of particles is of the order of billions, while the number of cells in a grid can be over 1024 for each grid side. Given that for the values of $Ng = 128^3$ and $Np = 10^6$ the total amount of memory used in the application is roughly 500MB, the memory demand to run a typical simulation is very high.

8.1. GridRPC implementation of Hydropad

Hydropad was originally a sequential Fortran code, which we upgraded this program to take advantage of the GridRPC API and to work with the GridSolve middleware. Table 6 shows the original Hydropad code of the main loop, written in the C language. Three functions, *grav*, *dark*, and *bary*, are called in this loop to perform the three main tasks of the application. In addition, at the first iteration of this loop, a special task, *initvel* is called to initialise the velocities of the particles. The dark and baryonic tasks compute the general velocities of the respective matter. At each iteration, these velocities are used by a local function, *timestep*, to calculate the next time step of the simulation. The simulation will continue until this time becomes equal to the present time of the universe, tsim = tuniv .

Table 6: Hydropad evolve loop

```
t_sim=0
while(t_sim<t_univ) {
grav(phi,phiold,rhoddm,rhobm,...);
if(t_sim==0){
initvel(phi,...);
}
dark(xdm,vdm,...,veldm);..
bary(nes,phi,...,velbm);
timestep(veldm,velbm,...,t_step);
t_sim+=t_step;
}
```

The GridRPC implementation of Hydropad application uses the APIs *grpc_call* and *grpc_call_async* to execute respectively a blocking and an asynchronous remote call of the Fortran functions. The first argument of both APIs is the handler of the task executed; the second is the session ID of the remote call while the following arguments are the parameters of the task. Furthermore, the code uses the method *grpc_wait* to block the execution until the chosen, previously issued, asynchronous request has completed. When the program runs, the GridSolve middleware maps each *grpc_call* and *grpc_call_async* functions singularly to a remote server. Then, the middleware communicates the data from the client computer to the chosen server and then executes the task remotely. At the end of the task execution, the data is communicated back to the client. In the blocking call method, the client cannot continue the execution until the task is finished and all the outputs have been returned. Instead, in the asynchronous method, the client does not wait for the task to finish and proceeds immediately to execute the next code. The output of the remote task is retrieved when the respective wait call function is executed.

Table 7 outlines the GridRPC implementation of the main loop of Hydropad that simulates the evolution of the universe. At each iteration of the loop, the first *grpc_call* results in the gravitational task being mapped and then executed. When this task is completed, the client proceeds to the next call, which is a non-blocking call of the dark matter task. This call returns after the task is mapped and its execution is initiated. Then, the baryonic matter call is executed in the same way. Therefore, the baryonic and dark matter tasks are executed in parallel. After this, the client waits for the outputs of both these parallel tasks using the *grpc_wait* calls.

Table 7: Hydropad implementation in GridRPC

```
t_sim=0;
while(t_sim<t_total) {
  grpc_call(grav_hndl,phiold,...);
  if(t_sim==0){
    grpc_call(initvel_hndl,phi,...);
  }
  grpc_call_async(dark_hndl,&sid_dark,x1,...);
  grpc_call_async(bary_hndl,&sid_bary,n,...);

  /* wait for non blocking calls */
  grpc_wait(sid_dark);
  grpc_wait(sid_bary); /* to finish */
  timestep(t_step,...);
  t_sim+=t_step;
}
```

8.2. SmartGridRPC implementation of Hydropad

The code in table 8 shows the modifications required to implement Hydropad for the SmartGridRPC model. The only minor difference between the GridRPC code in table 7 and the SmartGridRPC code in table 8 is the addition of: the *grpc_map* block and *grpc_local* condition. These belong to the SmartGridRPC API.

Table 8: Hydropad implementation in SmartGridRPC

```
t_sim=0;
while(t_sim<t_univ) {
  grpc_map("greedy_map"){
    grpc call(grav_hndl,phiold,...);
    if(t_sim==0){
      grpc_call(initvel_hndl,phi,...); }
      grpc_call_async(dark_hndl,&sid_dark,x1,.);
      grpc_call_async(bary_hndl,&sid_bary,n,..);

      /* wait for non blocking call */
      grpc_wait(sid_dark);
      grpc_wait(sid_bary);

      grpc_local(){
        timestep(t_step,...);
        t_sim+=t_step;
      }
    }
  }
}
```

The specified mapping heuristic, in this case the greedy mapping heuristic, generates a mapping solution for this group of tasks based on these performance models. On the second iteration through the group of tasks the group is executed according to the mapping solutions generated.

The *grpc_local* function is used by the application programmer to indicate when a local computation is executed. At run time, on the first discovery iteration, the code within this conditional statement is not executed. This is to avoid computing local executions when generating a performance model for the group of remote tasks. However, if a local computation directly affects the performance model of the group of remote tasks, then the *grpc_local* function should not be used. This would be the case a local computation affects whether certain remote tasks get executed or affects the size of computation of tasks. If this were the case, then the local computation should be executed during discovery and any structures, variables etc. that have changed values should be reset back to their original values before the beginning of execution.

On the second iteration, during the execution phase all the code in *grpc_map* function is executed normally (i.e. the local computation is also executed). The mapping in the code of table 8 is performed at every iteration of the main loop; this can generate a good mapping solution if the Grid environment is not a stable one. This would be the case if there are other applications' tasks running on the Grid servers. If the Grid environment is dedicated, where only one application executes at a time, a better mapping solution may be generated if the area to map contains more tasks, i.e. two or more loop cycles. A simple solution could be including an inner loop within the *grpc_map* code block (table 9). The application programmer could increase the number of tasks mapped together by increasing the number of iterations of the inner loop.

Figure 12 is a task graph generated for only two cycles of the evolution step. It is also possible to map a significantly larger number of evolution steps, by increasing the value of the *nb_evolutions* variable in table 9.

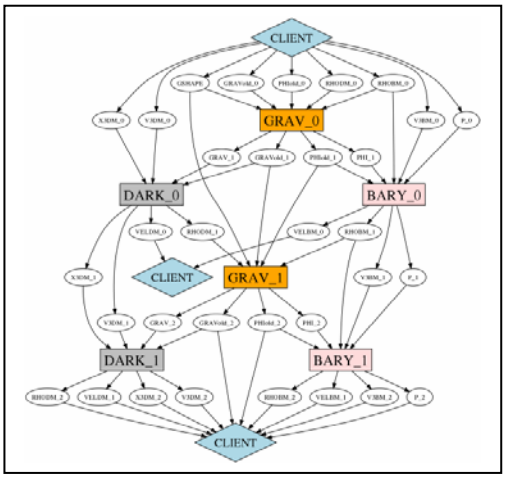

Figure 12: Task graph for two evolution steps

This type of coarse mapping would be more favourable on a distributed environment, which is highly stable, for example, a distributed environment that consisted of dedicated servers or servers that are idle. However, if the environment is highly changeable, which would be the case if the distributed environment consisted of workstations currently being used, then it might be

more advantageous to have a higher frequency of mappings. It may also be necessary to increase the frequency of mappings, if the task graph is altered as a result of the execution of one of the remote tasks in the task graph. For example, this may be the case if there is a conditional statement in the group of tasks that is based on an output of a remote task in the group (task A). If this conditional statement determines whether another remote task (task B) gets executed then the shape of the task graph depends on the output of task A. When the shape of a task graph is determined by the outputs of a remote task in the group then it is important to increase the frequency of mappings and perform mappings whenever the task graph is altered. To ensure the shape of the task graph is accurate in the aforementioned case, the task graph should be generated and mapped every time task A is executed.

It is also possible to make this mapping frequency more dynamically adaptive. In table 9, the value assigned to the variable *nb_steps* indicates how many evolution steps should be mapped collectively at the next point of execution of the application. This value can be fine-tuned during the execution of the application to determine the optimal number of evolutions to map as a group. In this example, the value for *nb_steps* is updated and fine-tuned using an evaluation function *func*. This may be a function that changes the value of the variable *nb_steps* based on an evaluation of the performances of previous executions of collective mappings.

Table 9: Dynamically determining the optimal group size to map

```
t_sim=0;
while(t_sim<t_univ) {
  nb_steps=func(..); //assign dynamically
  for(i=0;(i<nb_steps)&(t_sim<t_univ);i++){
    grpc_map("greedy_map"){
      grpc call(grav_hndl,phiold,...);
      if(t_sim==0){
        grpc_call(initvel_hndl,phi,...); }
        grpc_call_async(dark_hndl,&sid_dark,..);
        grpc_call_async(bary_hndl,&sid_bary,..);

        /* wait for non blocking call */
        grpc_wait( sid_dark);
        grpc_wait( sid_bary);

        grpc_local(){
          timestep(t_step,...);
          t_sim+=t_step;
        }
      }
    }
  }
}
```

This approach can be used to find the optimal mapping for an application on any given distributed environment. Once determined, this optimal number can then be assigned statically for each subsequent execution of the application on this environment without the need for an evaluation function.

However, in the case, where the environment is highly changeable, this optimal number of evolutions may vary throughout the execution of the application and therefore it may be more beneficial to maintain this dynamic update of *nb_steps* variable at run-time.

9. EXPERIMENTAL RESULTS

In the experiments performed in this section, we use three different implementations of Hydropad: the original sequential version, the GridSolve version and the SmartGridSolve version. For each version, we present the average computation time of one evolution step and the memory footprints of the application on the client machine. In the first part of this section, we compare the GridSolve version versus the local sequential version. Then, in the second part we compare the SmartGridSolve version of Hydropad versus both GridSolve and the local one. Furthermore, in the second part we will focus on the performance improvement of each of the key benefits of the SmartGridRPC model over the GridRPC model, which were introduced in section 2.2.

The hardware configuration used in the experiments consists of three machines: a client and two remote servers, S1 and S2. The two servers are heterogeneous, however, they have similar performance, respectively 498 and 531 MFlops, and they have equal amount of main memory, 1GB each. The bandwidth of the communication link between the two servers is 1Gb/s. The client machine, C, is a computer with low hardware specifications, 248MFlops of performance. The client to server connection varies depending on the experimental setup. We use two setups, C1 with a 1Gb/s connection and C100 with a 100Mb/s communication link. For each conducted experiment, table 10 shows the initial problem parameters and the corresponding data sizes (the total memory used during the execution of Hydropad on a single machine). The quantity of memory available in the client machine varies as well depending on the experimental setup. We use two configurations: C-1 with 1GB of memory, which is large enough to avoid paging, and C-256 with 256MB of memory, that undergo paging for larger problems.

Table 10: Input values and problem sizes for the Hydropad experiments

Problem ID	N_p	N_g	Data Size
P1	120^3	60^3	73MB
P2	140^3	80^3	142MB
P3	160^3	80^3	176MB
P4	140^3	100^3	242MB
P5	160^3	100^3	270MB
P6	180^3	100^3	313MB
P7	200^3	100^3	340MB
P8	220^3	120^3	552MB
P9	240^3	120^3	624MB

9.1. Experiments with the GridSolve version of Hydropad

Table 11 shows the results obtained by local computation and by GridSolve version of Hydropad using C1-1 as the client machine which has a fast network connection and large quantity of memory.

Table 11: Experimental results using client C1-1 that has 1Gb/s network link to the servers.

	Local	GridSolve	
P. ID	Time Step	Time Step	S_p v Local
P1	14.12s	9.40s	1.50
P2	29.90s	18.38s	1.63
P3	34.84s	20.82s	1.67
P4	52.04s	30.81s	1.69
P5	54.06s	32.00s	1.69
P6	58.56s	36.81s	1.59
P7	66.29s	37.22s	1.78
P8	102.03s	67.04s	1.52
P9	114.83s	112.05s	1.02

One can see that the GridSolve version is faster than the local sequential computation. The speedup obtained is constantly over 1.50, which is due to the parallel execution of the two tasks and the use of servers with greater performance than the client machine. The fluctuation in speedup obtained by GridSolve depends on the varying ratio of data size used by the two parallel tasks for different problem sizes. Furthermore, it should also be noted, that the speedup achieved on P9 is significantly lower due to paging on the server. This is caused by the fact that the GridRPC model maps both tasks to the same server and therefore causes paging on it.

Table 12 shows the results obtained by the GridSolve version when the client machine used, C100-256, has a slow client-to-servers connection of 100Mb/s and only 256MB of memory available. This hardware configuration simulates a common situation that can happen in real life. A user has access only to a slow client machine with low hardware specification, which is not suitable to perform large simulations, and wants to use a powerful Grid environment through a relatively slow network link. Table 12 also presents the scale of paging that occurs on the client machine during the executions. One can see that for the local computation the paging is taking place when the problem size is equal or greater than the machine main memory, 256MB.

Table 12: Experimental results using client C100-256 that has 100Mb/s network link to the servers and 256MB of memory.

	Local		GridSolve		
PD	Time Step	Paging	Time Step	Paging	S_p v Local
P1	14.32s	No	20.26s	No	0.71
P2	30.05s	No	38.75s	No	0.78
P3	35.78s	No	48.65s	No	0.74
P4	55.57s	Light	60.48s	No	0.92
P5	62.13s	Light	66.43s	No	0.94
P6	84.33s	Yes	76.76s	Light	1.10
P7	128.22s	Yes	93.74s	Yes	1.37
P8	231.56s	Heavy	150.03s	Heavy	1.54
P9	279.52s	Heavy	183.45s	Heavy	1.52

The GridSolve version is slower than the local computation when the client machine is not paging. This is happening because there is a large amount of data communication between tasks.

So for this configuration, the time spent communicating the data compensates for the time gained by computing tasks remotely. However, as the problem size gets larger and the client machine starts paging, the GridSolve version becomes faster than the local computation, even in the case of slow communication between the client and server machines. This trend is also seen in figure 13. For the GridSolve version, the paging is occurring later than for the local version, when the problem size is around 310MB, as shown in table 12. The GridRPC implementation can save memory due to the temporary data allocated remotely in the tasks and consequently increase the problem size that will not cause the paging. Furthermore, in the sequential local execution, the paging is taking place during a task computation, while for the GridSolve version the paging occurs during a remote task data communication. Hence, for the GridSolve version of Hydropad, the paging on the client machine does not negatively affect the execution time of the application.

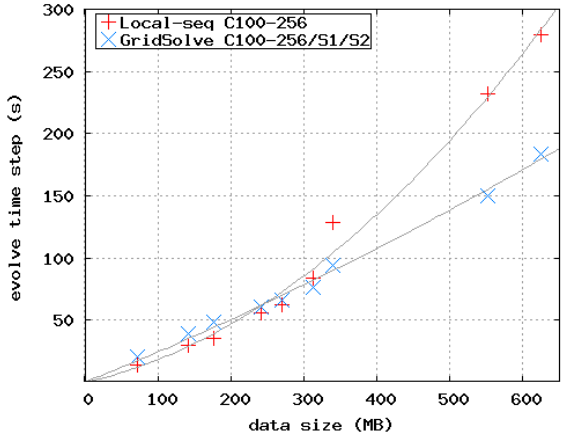

Figure 13: Evolution time step of the local and GridSolve computation on client C100-256

In these experiments, "light" paging means that paging is occurring only in some task calls and the amount of paging is approximately 10% of the main memory (approx. 25MB). "Normal" paging means that paging is occurring on almost every task call and the amount of paging is approximately 40% of the main memory (approx. 100MB). "Heavy" paging means that all task calls cause a memory page and almost 100% of the main memory is paged (approx. 256MB).

9.2. Experiments with the SmartGridSolve version of Hydropad

In the first experiment of this section, we use the same hardware configuration of table 12. The client machine used, C100-256, has a slow client-to-servers connection of 100Mb/s and only 256MB of memory available. As previously mentioned, this is a common situation. Table 13 shows the results obtained by the SmartGridSolve version for this configuration. This table shows that the SmartGridSolve version is much faster than the GridSolve and the sequential versions. The speedup is around three times that of GridSolve, figure 14, and the speedup versus the local sequential version is over 4 in the case of larger problems.

Table 13: Experimental results using client C100-256 that has 100Mb/s network link to the servers and 256MB of memory

	Local		GridSolve			SmartGridSolve			
PD	Time Step	Paging	Time Step	Paging	S_p v Local	Time Step	Paging	S_p v Local	S_pvGS
P1	14.32s	No	20.26s	No	0.71	7.31	No	1.96	2.77
P2	30.05s	No	38.75s	No	0.78	15.06	No	2.00	2.57
P3	35.78s	No	48.65s	No	0.74	16.36	No	2.19	2.97
P4	55.57s	Light	60.48s	No	0.92	28.06	No	1.98	2.16
P5	62.13s	Light	66.43s	No	0.94	27.54	No	2.26	2.41
P6	84.33s	Yes	76.76s	Light	1.10	27.78	No	3.04	2.76
P7	128.22s	Yes	93.74s	Yes	1.37	30.81	Light	4.16	3.04
P8	231.56s	Heavy	150.03s	Heavy	1.54	48.04	Light	4.82	3.12
P9	279.52s	Heavy	183.45s	Heavy	1.52	60.74	Light	4.60	3.02

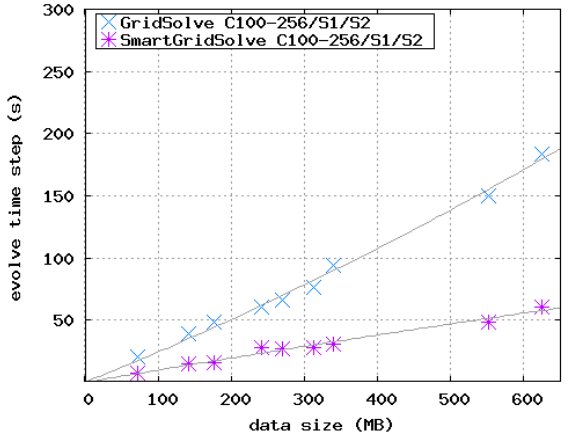

Figure 14: Execution times of the GridSolve and SmartGridSolve version of Hydropad on client C100-256

These performances improvements are due to the key features of the SmartGridRPC model: improved mapping, improved data movement and reduced memory usage. In the next experiments of this section, we show the benefits introduced by each feature by using specific hardware configurations and setup.

Computational load experiments. One important feature of SmartGridRPC is the superior mapping system that permits to have an improved balancing of computational load of tasks compared to standard GridRPC. In the underlying experiments, we compare the average computation time of one evolution step achieved by the GridSolve version versus the SmartGridSolve version of Hydropad, where SmartGridSolve is set up to utilize the same network topology of GridSolve (star-network), i.e. without direct server-to-server communication and server-caching. Consequently, the performance gains obtained by the SmartGridSolve version are due only to the improved mapping method. In these experiments, we use C1-1 as the client machine. This machine has a high speed network connection of 1Gb/s to the servers. Table 14 shows that the SmartGridSolve version of Hydropad is faster than the GridSolve version.

Table 14: Experimental results using only star-network topology (i.e. no direct server-to-server communication) and client C1-1 that has 1Gb/s network link to the servers

	GridSolve	SmartGridSolve	
P. ID	Time Step	Time Step	S_p v GS
P1	9.40s	7.09s	1.33
P2	18.38s	15.27s	1.20
P3	20.82s	16.17s	1.29
P4	30.81s	29.02s	1.06
P5	32.00s	28.99s	1.10
P6	36.81s	29.88s	1.23
P7	37.22s	30.88s	1.21
P8	67.04s	52.05s	1.29
P9	112.05	53.35s	2.10

Despite Hydropad having only two parallel tasks, the collective mapping of SmartGridRPC can produce a faster execution time than the individual task mapping of GridRPC. The baryonic task is computationally far larger than the dark matter one, Cbm≫Cdm. When a GridRPC system goes to map these two tasks, it does so without the knowledge that they are part of a group to be executed in parallel. Its only goal is to minimize the execution time of an individual task as it is called by the application. If the smaller dark matter task is called first, it will be mapped to the fastest available server. With the fastest server occupied, the larger baryonic task will then be mapped to a slower server and the overall execution time of the group of tasks will be sub-optimal. As previously mentioned, in some cases, both tasks will be mapped to the same server, which would also increase the total execution time and would cause paging on the server, as happened for problem P9.

Communication load experiments. As mentioned before, another primary improvement of SmartGridSolve is its communication model, use of which minimizes the amount of data movement between the client and servers. This advantage is most prominent when the client connection to the Grid environment is slow. Table 15 shows the results obtained by the SmartGridSolve version of Hydropad using C100-1 as the client machine which has a slow network connection of 100Mb/s. One can see that the SmartGridSolve version is much faster than the GridSolve versions. The increase of speed is over twice that of GridSolve, which is primarily due to the improved communication model of SmartGridSolve.

Furthermore, one can see that the timing results obtained by SmartGridSolve in table 15 are similar to those obtained in table 14. This shows that when the client-server links are slow and there is direct communication (table 15) it is similar to when the client links are fast and there is no direct communication (table 14). This shows that the SmartGridRPC model allows the mapping heuristic to generate solutions, which effectively minimize the communication load on the networks link.

Table 15: Experimental results using client C100-1 that has 100Mb/s network link to the servers

	GridSolve	SmartGridSolve	
P. ID	Time Step	Time Step	S_p v GS
P1	19.97s	7.24s	2.76
P2	38.73s	15.17s	2.55
P3	48.20s	16.24s	2.97
P4	61.59s	29.42s	2.09
P5	66.26s	28.91s	2.29
P6	78.16s	29.73s	2.63
P7	93.20s	31.25s	2.99
P8	140.53s	50.20s	2.80
P9	174.14	53.02s	3.28

Memory usages experiments. In the following experiments, we utilize the client machine C1-256, that has a high speed network connection of 1Gb/s to the servers and has 256MB of main memory. Table 16 shows the average computation time of one evolution step achieved by the local computation, by the GridSolve version and by the SmartGridSolve version of Hydropad. Table 16 also presents the scale of paging that occurs on the client machine during the various executions.

Table 16: Experimental results using client C1-256 that has 1Gb/s network link to the servers and 256MB of memory

	Local		GridSolve			SmartGridSolve			
PD	Time Step	Paging	Time Step	Paging	S_p v Local	Time Step	Paging	S_p v Local	S_pvGS
P1	14.3s	No	8.6s	No	1.67	7.0s	No	2.02	1.21
P2	30.0s	No	18.4s	No	1.63	14.4s	No	2.08	1.27
P3	35.7s	No	20.1s	No	1.77	15.8s	No	2.26	1.27
P4	55.5s	Light	31.3s	No	1.77	27.5s	No	2.02	1.14
P5	62.1s	Light	33.7s	No	1.84	28.1s	No	2.21	1.20
P6	84.3s	Yes	42.3s	Light	1.99	28.8s	No	2.92	1.47
P7	128s	Yes	63.1s	Yes	2.03	30.0s	Light	4.27	2.10
P8	231s	Heavy	109.3s	Heavy	2.12	46.6s	Light	4.96	2.34
P9	279s	Heavy	144.3s	Heavy	1.94	55.13	Light	5.07	2.62

One can see that for the SmartGridSolve experiments the paging on the client machine is less penalizing than in the GridSolve and local experiments. A secondary advantage of the direct server to server communication implemented in SmartGridSolve is that the quantity of memory used on the client machine is lower than that of the GridSolve version. Furthermore, in SmartGridSolve, the memory paging is happening only when data has to be sent to the server. Hence, it happens only at the beginning and at the end of a group of tasks execution. This minimizes the impact of paging on the overall execution of the group of tasks.

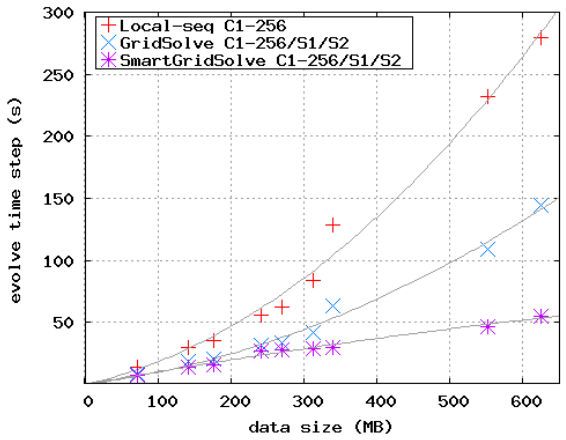

Figure 15: Execution times of the GridSolve and SmartGridSolve version of Hydropad when the client machine C1-256 has 256MB of memory

Therefore, the SmartGridSolve version of Hydropad can execute larger problems without the paging having a serious impact on the execution time. One can see that the computation time of the evolution steps in table 16 is similar to that of table 14 and 15. The speedup of SmartGridSolve over GridSolve, is increasing as the problem gets larger due to paging on the client. This trend is also seen in Figure 15.

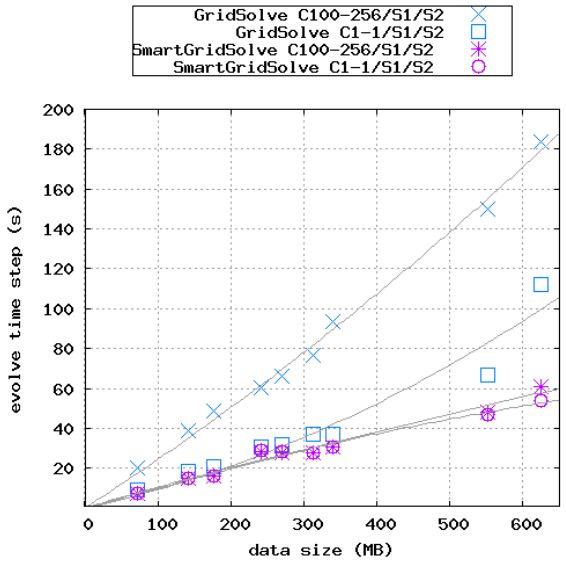

Figure 16: Execution times of the GridSolve and SmartGridSolve version of Hydropad when the client machines are C1-1 and C100-256

The new features of SmartGridRPC have also a secondary benefit. As previously mentioned, SmartGridSolve obtains similar results when the client memory and the client-to-server link are largely different. Consequently, the hardware configuration of the client has less impact on the

application performance than in the case of GridRPC. Figure 16 shows this trend. We compare the results obtained by GridSolve and SmartGridSolve version of Hydropad when the two configurations of the client used are the optimal one, C1-1, and the worst one, C100-256. It is possible to see that in the case of GridSolve the performance change dramatically depends on the hardware used while for SmartGridSolve the performance is similar.

10. CONCLUSION

In this paper, we have presented the SmartGridRPC model, which is an extension to the GridRPC model which aims to achieve higher performance. The SmartGridRPC model extends the GridRPC model, which maps tasks individually on to a star network, to provide functionality for collective mapping of tasks on a fully connected network. This functionality can be achieved using only two simple calls which are part of the SmartGridRPC API. The SmartGridSolve model has shown that mapping heuristics can improve the performance of an application by:

- Improving the load balancing of computation
- Improving the load balancing of communication
- Reducing the overall volume of communication
- Reduced memory usage on the client (reduce paging)
- Parallelism of communication.

We also outlined an implementation of the SmartGridRPC model in SmartGridSolve which is an extension to the GridSolve middleware which implements the GridRPC model. It described a possible implementation of the performance models which are used to simulate the different executions of the group of tasks on the fully connected network.

We also gave an experimental evaluation of the SmartGridRPC model in comparison with the GridRPC model using a real-life astrophysics application called Hydropad. This application simulates the evolution of clusters of galaxies in our universe from the beginning of time till present. The reason this application was chosen to benchmark both models was that it is an application which is not well suited to be implemented in Grid environments and consequently it can show the eventual limits and benefits of the two models tested. The experiments show a significant speedup when the application was executed using the SmartGridRPC model over the GridRPC model. The experiments section demonstrated the performance increase achieved by using the SmartGridRPC model and highlights the key benefits of the SmartGridRPC model. A speedup of 1.29 was achieved due to improved mapping of computation on to servers of the network. A speedup of 2.89 was achieved due to improved mapping of communication on to servers of the network. And a speedup of 2.62 was achieved due a decrease of memory usage and paging on the client.

ACKNOWLEDGEMENTS

This work was supported by the Science Foundation Ireland and in part by the IBM Dublin CAS.

REFERENCES

[1] Birrell A, Nelson B. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*; **2**(1): 39-59. 1984.

[2] Seymour K, Nakada H, Matsuoka S, Dongarra J, Lee C, Cassanova H. Overview of GridRPC: A Remote Procedure Call API for Grid Computing, *Lecture notes in computer science*, 274-278, 2002.

[3] YarKhan A, Seymour K, Sagi K, Shi Z, Dongarra J. Recent Developments in GridSolve. *International Journal of High Performance Computing Applications*, **20**(1), 131, 2006.

[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*, **1**(1), 41-51, 2003.

[5] Caron E, Desprez F. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, **20**(3), 335-352, 2006.

[6] Brady T, Guidolin M, Lastovetsky A. Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model, in *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid2008)*, Tsukuba, Japan, **29**.

[7] Brady T, Konstantinov E, Lastovetsky A, SmartNetSolve: High Level Programming System for High Performance Grid Computing, in *Proceedings of the 20th International Parallel and Distributed Symposium (IPDPS2006)*, IEEE Computing Society, 2006.

[8] Casanova H, Dongarra J, NetSolve: A Network Server for Solving Computational Science Problems, *In Proceedings of High Performance Computing Applications*, **11**(3), 212, 1997.

[9] Casanova H, Kim M, Plank J, Dongarra J. Adaptive Scheduling for Task Farming with Grid Middleware, *International Journal of High Performance Computing Applications*, **13**(3), 231, 1999.

[10] Arnold D, Dongarra J, The NetSolve Environment: Progressing Towards the Seamless Grid. In *Proceedings of the International Conference on Parallel Processing (ICPP2000)*, 199-206, 2000.

[11] Desprez F, Jeannot E. Adding Data Persistence and Redistribution to NetSolve. *LIP, ENS Lyon, Tech. Rep*, 2002.

[12] Desprez F, Jeannot E, LIP I, Lyon F. Improving the GridRPC Model with Data Persistence and Redistribution. in *Proceedings of the Third International Symposium on Parallel and Distributed Computing (ISPDC2004)*, 193-200, 2004.

[13] Del-Fabbro B, Laiymani D, Nicod J, Philippe L. Data management in grid applications providers. in *Distributed Frameworks for Multimedia Applications (DFMA2005)*, 315-322, 2005.

[14] Caron E, Del-Fabbro B, Desprez F, Jeannot E, Nicod J. M. Managing Data Persistence in Network Enabled Servers, *Scientific Programming Journal*, **13**(4), 333-354, 2005.

[15] Arnold D, Casanova H, Dongarra J. Innovations of the NetSolve Grid Computing System, *Concurrency and Computation: Practice and Experience*, **14**(13), 1457-1479, 2002.

[16] Seymour K, YarKhan A, Agrawal S, Dongarra J. NetSolve: Grid Enabling Scientific Computing Environments, *Grid Computing and New Frontiers of High Performance Processing*, 2005.

[17] Foster I, Kesselman C, Globus: A metacomputing infrastructure toolkit, *International Journal of High Performance Computing Applications*, **11**(2), 115, 1997.

[18] Guidolin M, Lastovetsky A. ADL: An Algorithm Definition Language for SmartGridSolve, *The 9th IEEE/ACM International Conference on Grid Computing*, 322-327, 2008.

[19] Tanimura Y, Nakada H, Tanaka Y, S. Sekiguchi. Design and implementation of distributed task sequencing on GridRPC, *Proceedings of the Sixth IEEE International Conference on Computer and Information Technology* (CIT06), 67, 2006.

[20] Amar A, Bolze R, Bouteiller A, Chouhan P.K, Chis A, Caniou Y, Caron E, Dail H, Depardon B, Desprez F, Gay J-S, Mahec G. Le, Su A. DIET: New developments and recent results, In *CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar2006)*, 2006.

[21] Caron E, Desprez F, Loureiro D. All-in-one Graphical Tool for the management of DIET a GridRPC Middleware, In *CoreGRID Workshop on Grid Middleware (in conjunction with OGF'23)*, 2008.

[22] Higgins R, Lastovetsky A, Managing the Construction and Use of Functional Performance Models in a Grid Environment, *Proceedings of the 23rd International Parallel and Distributed Symposium (IPDPS2009), 2009.*

[23] Lastovetsky A, Reddy R, Higgins R. Building the Functional Performance Model of a Processor, *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006)*, 746-753, 2006.

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

[25] Caniou Y, Jeannot E. Study of the behaviour of heuristics relying on the Historical Trace Manager in a (multi)client-agent-server System, *Technical Report 5168, LORIA*, 2004.

[26] Braun T, Siegel H, Beck N, Boloni L, et al., A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, *Journal of Parallel and Distributed Computing*, **61**(6), 810-837, 2001.

[27] Gheller C, Pantano O, Moscardini L. A cosmological hydrodynamic code based on the Piecewise Parabolic Method, *Royal Astronomical Society, Monthly Notices*, **295**(3), 519-533, 1998.

[28] Guidolin M, Lastovetsky A. Grid-Enabled Hydropad: a Scientific Application for Benchmarking GridRPC-Based Programming Systems, *Proceedings of the 23rd International Parallel and Distributed Symposium (IPDPS2009)*, 2009.

[29] Hockney R, Eastwood J. Computer Simulation Using Particles, *Institute of Physics Publishing*, 1988.

[30] Colella P, Woodward P. The piecewise parabolic method (PPM) for gas-dynamical simulations, *Journal of Computational Physics*, **54**, 174–201, 1984.

[31] Bertschinger E. COSMICS: Cosmological Initial Conditions and Microwave Anisotropy Codes, *ArXiv Astrophysics e-prints*, 1995.