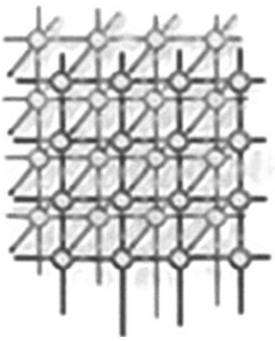
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2010)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1574



SmartGridRPC: The new RPC model for high performance Grid computing

Thomas Brady[1,*,†], Jack Dongarra[2], Michele Guidolin[1], Alexey Lastovetsky[1] and Keith Seymour[2]

[1]*School of Computer Science and Informatics, University College Dublin, Dublin, Ireland*
[2]*Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, U.S.A.*

SUMMARY

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 characterized 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 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. Copyright © 2010 John Wiley & Sons, Ltd.

Received 7 April 2009; Revised 21 January 2010; Accepted 30 January 2010

KEY WORDS: high performance computing; GridRPC; Grid; programming model; middleware

1. INTRODUCTION

GridRPC [1] is a standard promoted by the Open Grid Forum, which extends the traditional RPC for the Grid environment. A number of Grid middleware systems are GridRPC compliant, including GridSolve [2], Ninf-G [3], and DIET [4].

A GridRPC system processes each GridRPC task call by first performing dynamic resource and task discovery, then mapping the task to a server and then executing the task on the mapped server. The important restriction of the GridRPC model is that for each GridRPC call these three operations

*Correspondence to: Thomas Brady, School of Computer Science and Informatics, University College Dublin, Dublin, Ireland.
†E-mail: thomasbrady@ucd.ie

(discovery, mapping, and execution) are executed atomically and cannot be separated. As a result, each task will be mapped separately and independently of other tasks of the application. Therefore, the model can only support the minimization of the execution time of each individual task of the application rather than the minimization of the execution time of the whole application.

Another important aspect of the GridRPC model is its communication model. The communication model of GridRPC is based on the pure client-server model resulting in a star network topology. This means that inputs/outputs for remote tasks executed on servers can only traverse the client-server links.

Mapping tasks individually onto a star network results in mapping solutions that may be 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 loads of computation and communication. Another consequence of mapping tasks in this way is that dependencies among 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.

In this paper, we propose an enhancement of the traditional GridRPC model, called SmartGridRPC, which would allow a group of tasks to be mapped collectively onto 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 was extended so that the group of tasks can be collectively executed on 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 in mapping tasks collectively onto a fully connected network. The mapping heuristics which are implemented under the SmartGridRPC model can improve the performance of that group by:

- more effectively balancing the load of computation of the group of tasks;
- more effectively balancing the load of communication of the group of tasks;
- reducing the overall volume of communication of the group by eliminating bridge communication either by caching or direct data transfers among servers;
- increasing the parallelism of communication;
- reducing memory usage and paging.

These are the key benefits of executing an application using the SmartGridRPC model over the GridRPC model and they will be described in more detail in Section 2.

The paper is outlined as follows. The motivation and key benefits of SmartGridRPC are described in Section 2. The SmartGridRPC model and API are described in Section 3. Section 5 describes Hydropad, the astrophysics application we used to evaluate the performance of the SmartGridRPC model. Section 6 gives the experimental results that compare the GridRPC model with the SmartGridRPC model using the Hydropad application to evaluate the performance of both models. The paper concludes with Section 8.

DOI: 10.1002/cpe

2. SMARTGRIDRPC MODEL: MOTIVATION

This section describes each of the key benefits of SmartGridRPC outlined in the Introduction.

2.1. Improved balancing of computation load

In GridRPC, tasks get mapped to servers individually. This could result in poor load balancing of computation. As tasks get mapped individually, it is impossible to balance the load of computation of a group of tasks that are executed in parallel. If tasks are mapped individually, each task will be mapped without the knowledge of any of the subsequent parallel tasks. For example, this means that if two tasks of different computational loads are executed in parallel and if the larger task follows the smaller task, the mapping heuristic will give the smaller task priority 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 executed in parallel. Therefore it maps the smaller task to the faster server as this will yield the lowest execution time for this individual task. And 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 of both tasks are considered collectively and can therefore be better distributed and balanced 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.

2.2. Reduced volume of communication

As the GridRPC model maps tasks individually onto a client-server network, the model forces bridge communication among tasks. This occurs because dependencies are not known among 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 the destination task. However, this can be eliminated with the SmartGridRPC model, where tasks can be mapped collectively onto a network which is fully connected. As tasks are mapped collectively, dependencies among tasks are known. These dependencies can then be mapped onto virtual links connecting the source server to the destination server, which is only one communication step. Therefore, the overall volume of communication required to be sent over the network and the client-server links will be reduced which would result in improved application performance.

Moreover, if the source task and destination task are both executed 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.

2.3. Improved balancing of communication load

As the GridRPC model is based on the pure client-server model, communication can only be mapped to client-server links. This may result in the client-server links becoming heavily loaded.

DOI: 10.1002/cpe

SmartGridRPC can increase the performance of an application by better distribution of the communication load over the network. When tasks are mapped collectively, the volume of communication of each task in the group of tasks is known. As 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 and balanced over the fully connected network. This improved load balancing of communication will result in improved overall communication times and hence improved application performance.

2.4. 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 among 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 are to be executed on different machines. If any of the tasks are not executed 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 the overall impact that these communication transactions have on the overall performance of the application.

2.5. Reduced memory usage and paging

The client machine is a real bottleneck of the GridRPC model as it needs to store all intermediate data and communicate with all servers. A low-specification client machine can significantly deteriorate the overall performance of GridRPC-based applications. The direct communication among servers and the data caching that the SmartGridRPC model implements mean that intermediate results do not have to be sent back and stored on the client. This minimizes the amount of memory used on the client and the volume of communication necessary between the client and servers. Technically speaking, this could eliminate paging on the client that would otherwise occur. This elimination of paging would considerably increase the performance of an application. In terms of the overall

DOI: 10.1002/cpe

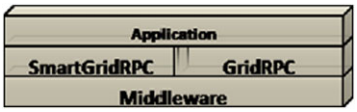


Figure 1. Overview SmartGridRPC model and API.

design, this makes the performance of a SmartGridRPC application much less dependent on the capacity of the client machine.

3. 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 1). Therefore, if any middleware has been upgraded 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 onto a fully connected network or for the standard GridRPC model, where tasks are mapped individually onto 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, where only SmartGridRPC-enabled servers (*SmartServers*) can communicate directly. If it is installed on all servers, the system will be extended to allow for collective mapping on the fully connected network.

3.1. SmartGridRPC: API and semantics

3.1.1. The grpc_map() function

The *grpc_map()* function of 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 task calls that is to be mapped collectively.

In addition, the application programmer specifies the mapping heuristic to implement by passing it in as a parameter to the *grpc_map()* function.

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


DOI: 10.1002/cpe

The *grpc_map()* 'function' is in fact a macro that inserts a while loop around the code block enclosed by the parentheses. When the *grpc_map()* function is called the code within the parentheses of the function is iterated through twice and during these iterations the discovery, mapping and execution of tasks are performed for all tasks collectively.

3.1.1.1. Discovery phase. On the first iteration through the group of tasks, the task graph for the group of tasks and the performance model of the fully or partially connected network are discovered and generated. During this first iteration, each GridRPC task call within parentheses is discovered but not executed, hence all tasks in the group can be discovered collectively. This is different from the GridRPC model, which only allows a single task to be discovered at any one time.

This allows for the discovery of the computation load, the communication load and the dependencies of all tasks in the group, which is used to generate the task graph. The dependencies among tasks are discovered by analyzing the pointer values of the input and output arguments. The computation and communication loads are discovered using the parameters passed into the calling sequence of the tasks and the performance models of the tasks. In addition during this phase, the performance of the servers on the network, the performance of the client-server links and the performance of links connecting *SmartServers* to other *SmartServers* are discovered. This information is used to generate the performance model of the fully or partially connected network.

Possible implementations for generating the task graph and the network performance model are described in Section 3.2.

3.1.1.2. Mapping phase. Based on the task graph and the performance models of the network, the mapping heuristic produces a mapping solution, which satisfies a certain criterion, for example minimizing the execution time of tasks. The mapping heuristic is specified by the application programmer using the SmartGridRPC API.

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

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

- Client-server communication—which is standard GridRPC communication.
- Server-server communication—when the server sends a single argument to another server.
- Client broadcasting—when the client sends a single argument to multiple servers.
- Server broadcasting—when the server sends a single argument to multiple servers.
- Server caching—when the server stores an argument locally for future tasks.

As a result, the network may have:

- A fully connected topology—where all the servers are *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.

DOI: 10.1002/cpe

- A star connected topology—where all servers are standard servers and they can only communicate with each other via the client.

3.1.1.3. 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 among the tasks in the group.

3.1.2. The grpc_local() function

The SmartGridRPC model also requires a method for identifying the client code that has no influence on the task graph and therefore should not be executed at the discovery phase. This can be achieved using many possible approaches. For example, a preprocessor approach could be used to identify the client code transparently. Where the client code cannot be automatically 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 data objects which have been updated
 // during the discovery phase

 grpc_local(){
 //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 will return false during the task discovery phase and true during the execution phase.

Thus, any segments of client code that are 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 data object is updated on the client that determines which remote tasks get executed or determines the size of inputs/outputs of any task, then the operations on this data object should not be enclosed by the *grpc_local()* function. For example, this should be considered if there are remote task calls within a while, for or if statement. Then local operations on any data object that affects the results of these conditional statements should not be enclosed in the *grpc_local()* function. As a result these operations will be executed during the discovery phase. If any data object is updated during the task discovery cycle it should be restored to its original value before the execution cycle begins. If a data object that affects the results of these conditional statements is updated remotely, then the *grpc_map()* function should be placed inside

DOI: 10.1002/cpe

the conditional statement. In that way, the number of tasks getting mapped is not affected by the conditional statement.

3.2. Performance models

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 context of this paper, a performance model is any structure, function, parameter, tool etc. which is used to estimate the execution time of tasks in the distributed environment. The job of generating the performance models is divided among the different components of the GridRPC architecture (i.e. client, server, and registry). In this paper, the registry is an abstract term for the entity or entities, which stores information about the registered tasks and the underlying network.

Each component in the architecture may only be capable of constructing parts of the performance model. Therefore, the registry accumulates these parts from the different components and generates the required performance models.

There are numerous methods for implementing performance models, hence they are not specified in the SmartGridRPC model. Examples of performance models are the ones currently implemented in SmartGridSolve [5], which have extended the performance models used in GridSolve.

In the future, SmartGridSolve will implement performance models such as the Functional Performance Model, described in [6,7]. Other possible implementations could include the Network Weather Service [8], FAST (DIET) [4], the MDS directories (Globus, Ninf) [3], and the Historical Trace Manager (GridSolve) [9]. Implementing these performance models under the SmartGridRPC model would allow for collective discovery of the task graph for a group tasks and the discovery of the performance model of the fully or partially connected network. The task graph and performance model of the network are used by the mapping heuristics to estimate the execution time of the group on the fully or partially connected network.

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 effect the execution of 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 among different servers.

The following paragraphs 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 servers provide the part of the performance model that would facilitate the estimation of the execution time 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. The partial model

DOI: 10.1002/cpe

will be referred to as the *server PM*. This part of the performance model is server specific such as the performance of the server and its communication links and a partial model of its available tasks.

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 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. 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. Alternatively, it may be updated during the run-time of the application when actual running times of tasks are used to build the performance model. Suffice it 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.

The client also provides a part of the performance model that 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 specifies the list of tasks in the group, their order, the dependencies among tasks and the values of the scalar arguments in the calling sequences. This discovery of the group of tasks is enabled by the *grpc_map()* function, which is part of the SmartGridRPC API. In addition, the *client PM* also specifies the performance of the client-server links.

At the end of the discovery phase, the *client PM* and the *server PM* are combined on the registry to generate a task graph representation of the group of tasks and the performance model of the fully or partially connected network. These are then used by the mapping heuristic to generate a mapping solution for the group of tasks.

4. IRREGULAR ALGORITHMS

There are certain situations where the automatic task graph generation will not work. A typical example is when a conditional construct exists that checks a value that cannot be known without executing a remote task call. A technique to avoid this problem is to create the task graph from a smaller block of code within the construct whereas the resulting group of tasks to be mapped generates a less optimal execution. A comprehensive solution to this problem is to permit the application programmer to explicitly specify a task graph that best represents the run-time execution of the irregular algorithm. As the application programmer usually has an in-depth knowledge of the algorithm used inside his application, he can generate the most representative task graph possible in the situation where the output of a remote task call can change the flow of execution. Therefore, a SmartGridRPC middleware that uses an explicitly generated representative task graph can achieve high performance also for applications with irregular algorithms. This permits any type of scientific distributed application to obtain high performance using a GridRPC-based programming system. A common method to directly specify a task graph-like structure is to use a specific high level language.

The Algorithm Definition Language (ADL) was designed as a tool to help an application programmer to easily specify a task graph for all kinds of algorithms [10]. It is demonstrated in [10] that ADL in conjunction with the SmartGridRPC model improves the performance of the

DOI: 10.1002/cpe

example applications, that comprise irregular algorithms, over the individual use of SmartGridRPC and GridRPC.

5. EXPERIMENTAL COMPARISON OF THE SMARTGRIDRPC AND GRIDRPC

Snavely *et al.* [11] identify a trend of the types of scientific applications that may benefit from Grid computing. They classified Grid applications in four large groups by comparing the difference in the computation of the tasks, the amount of memory and data used and the inter-task communications required. The first class defined (class I) contains applications that are embarrassingly parallel in nature. These applications can be divided into many tasks where there are no data dependencies among them. Thus, these tasks can easily run simultaneously on many systems of the Grid environment. The applications of second class (class II) compute continuous streams of data. Thus, they are called pipeline or stream applications. The tasks that compose an application of this class are data intensive and, while there is parallelism between them, there are minimal data dependencies between tasks. Class III applications have tasks with a high level of data synchronization between them. Thus, data dependencies and inter-task communication have an important impact on the performance and possible task parallelism of the application. The applications of this class are called tightly synchronized applications. Finally, class IV contains applications that perform data-related workloads, such as search or distributed database applications. The tasks employed by these applications are usually not computational and memory intensive.

Classes I and II of distributed scientific applications are ideal to get top performance in a distributed environment and therefore in a Grid environment. This happens since their tasks have high computation and minimal data dependencies and they can be easily executed in parallel on many different remote systems. In fact, these two classes of applications are still mainly developed and executed using batch management systems specific for Grid computing (e.g. Condor-G [12]) or Grid middlewares specific for stream processing (e.g. gLite [13]). On the other hand, the class III group is more important since it contains a large amount of scientific applications that are largely used and that are difficult to execute in a distributed environment, for example climate, astrophysics, aerodynamic, and molecular simulations.

High performance is an important design objective of the GridRPC model, since its target users are computational scientists. If we consider that many available systems for Grid computing are highly specialized in executing class I and II applications and are already widely used, the main aim of GridRPC must be to achieve high performance in the execution of class III distributed scientific applications.

This is the only way that GridRPC will attract interest from a large number of scientific users, since an easy to use and easy to develop paradigm that obtains high performance for such applications in Grid computing is still missing. The main problem of GridRPC is that there has been no comprehensive performance analysis showing its potential and limitations for executing task parallel tightly synchronized applications. The overwhelming majority of applications chosen, or artificially created, to demonstrate the performance of GridRPC middlewares are of class I and II.

We believe that to justify the use of GridRPC, we should not use an extremely suitable application to demonstrate the performance potential of GridRPC systems but a real-life application of class III that shows the eventual limits and benefits of the GridRPC middleware tested. We propose Hydropad

DOI: 10.1002/cpe

[14,15], a real-life astrophysical application, to be used as a tool for experimental performance evaluation of GridRPC and SmartGridRPC models. This application is composed of tasks that have a balanced ratio between computation and communication with a high level of data synchronization between them. Therefore, this application can be classified as class III.

The Hydropad application simulates the evolution of clusters of galaxies in the Universe. Hydropad requires high processing resources because it has to simulate an area comparable with the dimensions of the Universe.

The cosmological model, which this application is based on, assumes 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 theorized 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.

An important characteristic of Hydropad is the difference in the computational and memory load of its tasks. The computational load of the baryonic matter component is far greater than the dark matter, $C_{\mathrm{bm}} \gg C_{\mathrm{dm}}$, when the number of particles is equal to the number of cells in the grid, $N_p = N_g$. Furthermore, the quantity of data used by the dark matter computation is much greater than the baryonic matter, $D_{\mathrm{dm}} \gg D_{\mathrm{bm}}$.

Hydropad was executed using GridSolve in order to evaluate middleware systems that implement the GridRPC model and SmartGridSolve in order to evaluate systems that implement the SmartGridRPC model.

5.1. GridRPC implementation of Hydropad

Table I 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.

5.2. SmartGridRPC implementation of Hydropad

The code in Table II shows the modifications required to implement Hydropad for the SmartGridRPC model. In Table II, the *grpc_map()* and *grpc_local()* functions have been added to the GridRPC code in Table I to make Hydropad SmartGridRPC enabled.

On the first iteration through the map block, each task is discovered. The specified mapping heuristic, in this case the greedy mapping heuristic, generates a mapping solution for this group of tasks. 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 identify code that should not be executed during the discovery phase. Any local computations that do not affect the generation of the task should be encapsulated by the *grpc_local()*. This prevents the encapsulated code being

DOI: 10.1002/cpe

Table I. Hydropad implementation in GridRPC.

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

      /* wait for non blocking calls */
      grpc_wait (sid_dark);
      grpc_wait(sid_bary);
      print_info (xl, n,...);
      t_sim+=t_step;
}
```

Table II. Hydropad implementation in SmartGridRPC.

```
t_sim_old=t_sim;
grpc_map("greedy_map"){
      t_sim=t_sim_old;
      while(t_sim<t_total){
            grpc_call(grav_hndl, phiold,...);
            if(t_sim=0){
                  grpc_call(intvel_hndl, phi, ...);
            }
            grpc_call_async(dark_hndl,&sid_dark,xl,...);
            grpc_call_async(bary_hndl,&sid_bary,n,...);

            /*waitfornonblockingcalls*/
            grpc_wait(sid_dark);
            grpc_wait(sid_bary);
            grpc_local(){
                  print_info(xl,n,..);
            }
            t_sim+=t_step;
        }
}
```

executed during the discovery phase, which happens during the first iteration through the group of tasks.

However, if a local computation directly affects the task graph , then the *grpc_local()* function should not be used. This would be the case if a local computation affects whether certain remote tasks get executed or affects the size of the computation of tasks. For example, in Table II the *t_sim* is updated within the scope of the *grpc_map* function and determines the number of remote tasks that will be executed.

DOI: 10.1002/cpe

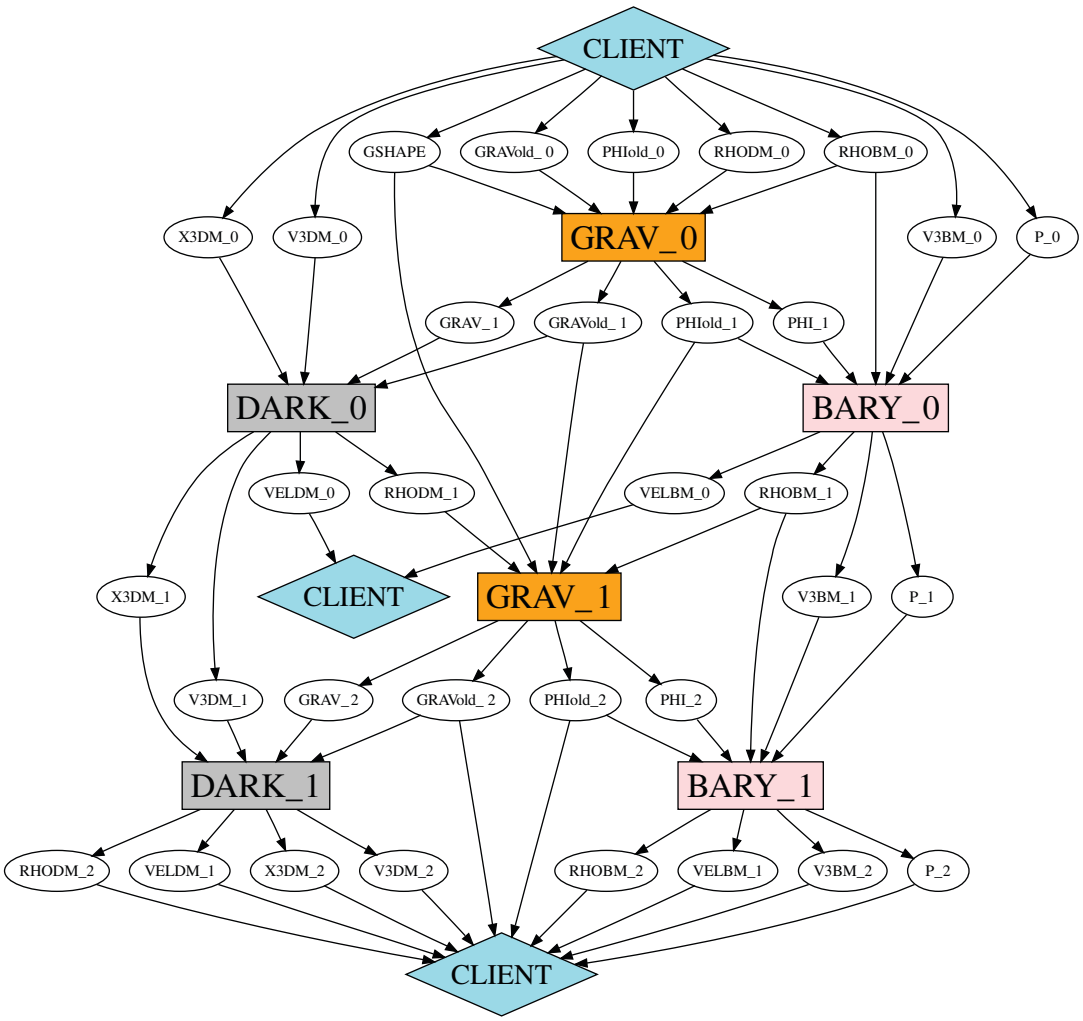


Figure 2. Task graph for two evolution steps.

Local operations on the *t_sim* variable are therefore not enclosed by the *grpc_local()* function. In addition this variable is reset to its original value at the beginning of the scope of the *grpc_map()* function. This is required as all data objects updated during the discovery phase must be reset to their original values before the beginning of the execution phase.

Once the discovery phase is completed, a task graph is generated. Figure 2 is a task graph generated for only two cycles of the evolution step. The mapping heuristic then generates a mapping solution based on this task graph and the performance model of the network. On the second iteration, during the execution phase all the code in *grpc_map()* function is executed normally (i.e. the local

DOI: 10.1002/cpe

Table III. Dynamically specify the size of the group of mapped tasks.

```
while(t_sim<t_total){
     t_sim_old=t_sim;
     grpc_map("greedy_map"){
          t_sim=t_sim_old;
          for(i=0;(i<nb_evol)&(t_sim<t_total);i++){
               grpc_call(grav_hndl,phiold,...);
               if(t_sim==D){
                   grpc_call(intvel_hndl,phi,...);
               }
               grpc_call_async(dark_hndl,&sid_dark,xl,...);
               grpc_call_async(bary_hndl,&sid_bary,n,...);

               /* wait for nonblocking calls */
               grpc_wait(sid_dark);
               grpc_wait(sid_bary);
               grpc_local(){
                   print_info(xl,n,...);
               }
               t_sim+=t_step;
         }
         grpc_local(){
               end_time=get_time();
               exec_time=start_time-end_time;
               nb_evol=calc_nb_evol(exec_time,nb_evol,prev_evol_times);
               start_time=get_time();
         }
     }
}
```

computation is also executed) and remote tasks and remote communication transactions are executed according to the mapping solution generated.

The mapping in the code of Table II is performed for all iterations of the main while loop. This type of coarse mapping would be more favorable on a highly stable distributed environment, for example a distributed environment that consisted of dedicated servers or servers that are idle. However, Grids are usually dynamic in nature, consisting of machines that are not dedicated to a single application and therefore it may be more favorable to generate more frequent mapping solutions. To generate more frequent mapping solutions in Table II, the *grpc_map* could be placed inside the while loop. This would generate a mapping solution for each evolution step.

Furthermore, since Grids are usually dynamic in nature it may be favorable to make the frequency in which mapping solutions are generated more dynamically adaptive. In Table III, the value assigned to the variable *nb_evol* 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 collectively at the various stages of the execution of the application. In this example, the value for *nb_evol* is updated and fine-tuned using an evaluation function. In this example, we use *calc_nb_evol()* function to dynamically determine the number of evolution steps to map collectively after each mapping solution is executed.

DOI: 10.1002/cpe

This function changes the value of the variable *nb_steps* based on an evaluation of the performances of previous executions of collective mappings. However, this is just given as an example of a possible implementation as there are many conceivable implementations of this type of evaluation function.

6. EXPERIMENTAL RESULTS

In the experiments performed in this section, we use three different implementations of Hydropad: the original sequential implementation, a GridSolve and a SmartGridSolve implementation. For each implementation, we present the average computation time of one evolution step of the Hydropad application. The average time is calculated from five separate executions. In the SmartGridSolve version the ten evolution cycles are mapped collectively. A more comprehensive analysis of the time spent on communication/computation/paging is given in [16].

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, 498 and 531 MFlops, respectively, and they have equal amount of main memory, 1GB each. The bandwidth of the communication link between the two servers is 1 Gb/s. The client machine, C, is a computer with low hardware specifications, 248MFlop of performance. The experiments were conducted for two different hardware setups, where the client-server link speed and the client memory size were varied. In the first configuration setup (**C100-256**), the client-server link is 100 Mb/s and the client memory size is 256 MB. In the second configuration setup (**C1-1**), the client-server link is 1 Gb/S and the memory size is 1 GB. For each experimental setup, the evolution steps were calculated for increasing number of simulated particles. In these experiments, the amount of memory required to run the application (data or problem size), increases proportionally with the number of particles simulated.

In Section 6.1, we compare the local sequential, the GridSolve and the SmartGridSolve implementation of Hydropad executed on the low-specification client machine. Then in Section 6.2 we compare the local sequential, the GridSolve and the SmartGridSolve implementation of Hydropad executed on high-specification client machine. Finally, in Section 6.3 we compare the performance of the GridSolve and SmartGridSolve implementations on both configuration setups.

6.1. Experiments with the low-specification client machine

Figure 3 shows the results obtained by the local, GridSolve and SmartGridSolve implementations of Hydropad when the client machine has a slow client-to-servers connection of 100 Mb/s and only 256 MB 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.

When the local sequential implementation of Hydropad was run, paging occurred when the problem size was equal to or greater than the main memory on the client machine which was 256 MB.

The GridSolve implementation is slower than the local computation when the client machine is not paging. This happens because there is a large amount of data communication among tasks. So for

DOI: 10.1002/cpe

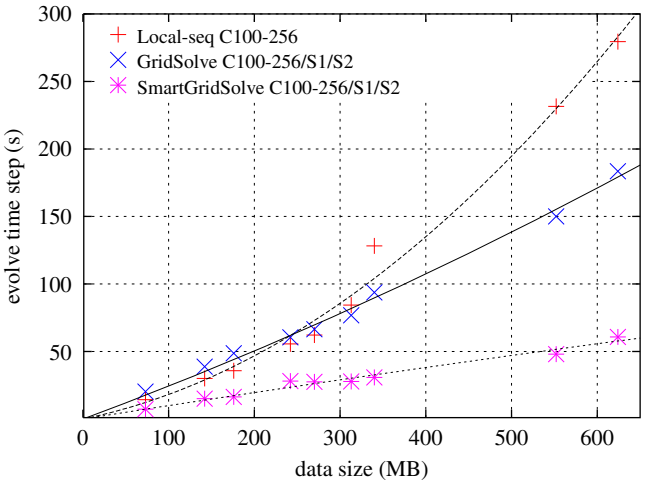


Figure 3. Evolution time step on the low-specification client machine.

this configuration, the time gained by executing the computation in parallel on faster remote servers does not compensate for the time sent communicating the data to those servers. However when the problem size passes 256 MB, the speedup of GridSolve over the local implementation increases due to the increased paging on the client during the execution of the local implementation. This speedup is achieved despite the slow communication links between the client and server machines.

Figure 3 also shows that the SmartGridSolve implementation is much faster than the GridSolve and the local sequential implementation. The speedup is around four times that of GridSolve and the speedup versus the local sequential implementation is nearly six in the case of larger problems. These performance improvements are due to the key features of the SmartGridRPC model: improved balancing of computation load, reduced volume of communication, improved balancing of communication load, increased parallelism of communication and reduced memory usage on the client.

SmartGridSolve improved the mapping by improving the load balancing of computation of baryonic matter and dark matter tasks. The baryonic task is computationally far larger than the dark matter, $C_{\mathrm{bm}} \gg C_{\mathrm{dm}}$. When a GridRPC system maps these two tasks, it does so without the knowledge that they are a 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 (S2). With the fastest server occupied, the larger baryonic task will then be mapped to a slower server (S1) and the overall execution time of the group of tasks will be sub-optimal.

In the SmartGridSolve implementation because the computation load of all tasks in the group were considered collectively as a group, the baryonic matter task was assigned to the faster server (S2) and to baryonic matter the slower server (S1). This improves the load balancing of computation and the execution of both tasks executing in parallel will be decreased. Secondly whenever there were dependencies among tasks that reside on different servers, this dependency was mapped onto the link connecting the source server to the destination server. This eliminates bridge communication

DOI: 10.1002/cpe

and reduces the overall volume of communication on the network and on the client-server links. Whenever there were dependencies among tasks that reside on the same server the output was cached to the local file system and retrieved from the file system by the subsequent destination task. This further decreases the overall volume of communication and increases the performance of the application. In addition, all remote communication and caching was performed asynchronously, hence whenever it can be done in parallel with other computation or communication in the group, these operations were overlapped. Furthermore, since intermediate data are not sent back to the client and temporarily stored there, this decreases the amount of memory used on the client and would eliminate paging that would otherwise occur. This significantly increases the performance of SmartGridSolve over GridSolve when the GridSolve implementation begins to page at around 310 MB.

6.2. Experiments with the high-specification client machine

Figure 4 shows the results obtained by the local sequential, the GridSolve and the SmartGridSolve implementations of Hydropad on the second network configuration where the client machine has a fast network connection and a large quantity of memory.

One can see that the GridSolve implementation is faster than the local sequential computation. The speedup obtained is over 1.50 for all problem sizes. As the communication links are faster, the speedup due to the parallel execution of the two tasks on faster remote machines is increased compared with the experiments on the low-specification client machine. The fluctuation in speedup obtained by GridSolve depends on the varying ratio of the data size used by the two parallel tasks for different problem sizes.

For the GridSolve implementation, the paging again occurs later than for the local implementation, when the problem size is around 310 MB, whereas paging occurs at 256 MB in the local implementation. The GridRPC implementation can save memory due to the temporary data allocated remotely in the tasks and consequently increasing the problem size will not cause the significant

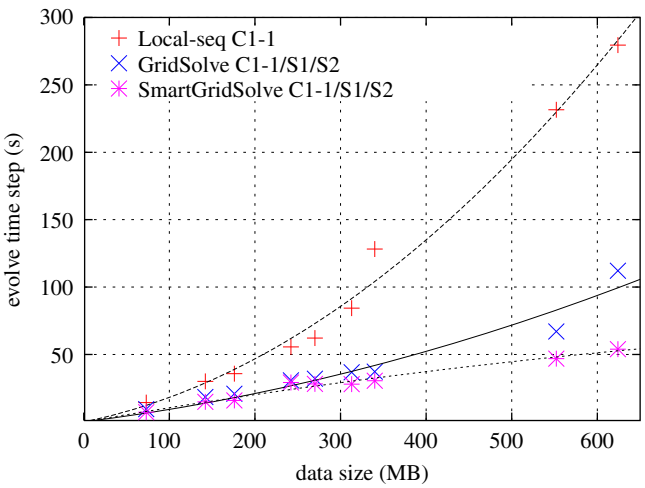
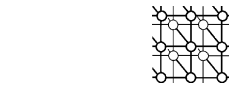


Figure 4. Evolution time on the high-specification client machine.

DOI: 10.1002/cpe



paging that happens during the execution of the location implementation. Furthermore, in the local sequential execution, the paging takes place during a local task computation, whereas for the GridSolve implementation the paging occurs during a remote task data communication. Hence, for the GridSolve implementation of Hydropad, the remote paging on the server machines does not negatively affect the execution time of the application.

The speedup of SmartGridSolve over GridSolve in Figure 3 is not as significant as the first configuration shown in Figure 4. This is because the penalty for bridge communication in the GridSolve implementation is not as severe as when the client-server links are fast in this case. In addition the penalty for temporary allocation of intermediate data on the client is not as severe as there is 1 GB of memory capacity on the client. Thus in this case, the client will not page in the GridSolve implementation of Hydropad. However, SmartGridSolve still achieves over two times speedup over GridSolve.

6.3. GridSolve vs SmartGridSolve: influence of the client machine on the performance

The new features of SmartGridRPC also have a secondary benefit. In these experiments, SmartGridSolve implementation obtains similar results for both configurations 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 5 shows this trend.

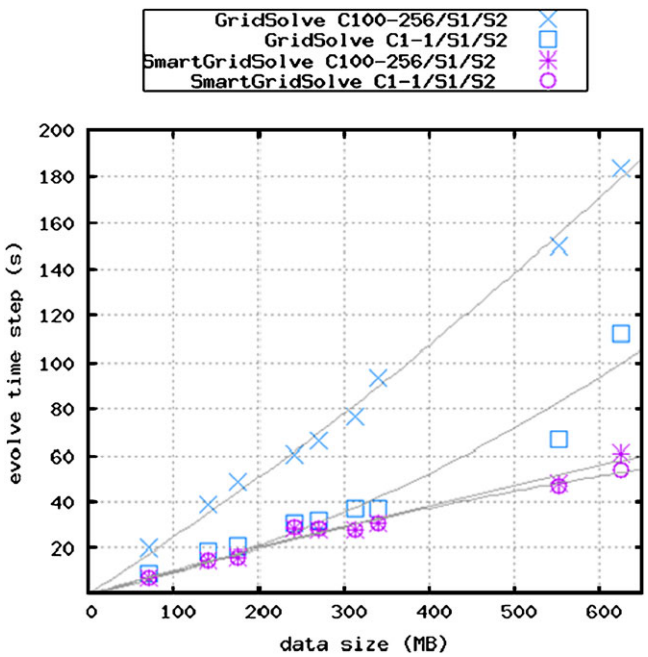


Figure 5. Execution times of the GridSolve and SmartGridSolve implementation of Hydropad when the client machines are C1-1 and C100-256.

DOI: 10.1002/cpe

It is possible to see that in the case of GridSolve the performance changes dramatically depending on the hardware used whereas for SmartGridSolve the performance is similar. This is because the SmartGridSolve implementation is less dependent on the performance of the client-server links and the memory capacity on the client.

Additionally this shows that the SmartGridSolve implementation is more scalable than the GridSolve implementation. This is because the major limitation when scaling up applications is the client-server bottleneck. Typically, the more an application is scaled up the more the client-server links will be used and the larger the amount of memory capacity that will be required on the client to store the intermediate data. But this is not the case with the SmartGridRPC model, where even when the problem size increases, the performance of the SmartGridSolve implementation remains linear. However, in the GridSolve implementation the performance degrades at an increasing rate due to the client-server links being heavily loaded and paging occurring on the client. These experiments show that the SmartGridRPC model is a more scalable solution than the GridRPC model.

7. RELATED WORK

A significant body of research has been done in the area of workflow management systems in Grid computing [17–21]. In addition, the inspector-executor [22,23] approach for run-time parallelization has some similarities to the application discovery of the SmartGridRPC model. However, little has been done in the area of workflow management and application discovery in the context of the GridRPC programming model.

In DIET, which is a GridRPC-based middleware, a special agent called MADAG was implemented to handle workflow submissions [24]. However, the task graph was not automatically discovered and had to be explicitly specified in an XML file. This approach of using XML files to handle task graph submissions is similar to the approach of SmartNetSolve [25] which was the predecessor of SmartGridSolve. In addition, the DIET approach does not adhere to the GridRPC model as the task graph submission is done instead of executing RPC calls.

In GridSolve, distributed task sequencing was developed for the GridRPC model [26]. This allows data dependencies to be discovered among a set of tasks and whenever data dependencies exist the data are sent directly among servers executing those tasks. However, the SmartGridRPC model provides the advantage of also determining the computation and communication loads of tasks in the group by using the performance models described in Section 3.2. Consequently, the SmartGridRPC model can achieve improved optimization as it has a more comprehensive representation of the group of tasks.

The original design of SmartGridRPC was inspired by the mpC [27] and HeteroMPI [28] programming systems for heterogeneous parallel computing based on the optimal mapping of the implemented parallel algorithm on the executing heterogeneous platform.

8. CONCLUSION

In this paper, we have presented the SmartGridRPC model, which is an extension to the GridRPC model that aims to achieve higher performance.

DOI: 10.1002/cpe

Using the Hydropad application for performance analysis, it was shown that the SmartGridRPC model reduces the influence that the client machine and the client links have on the performance of the executing application. This removes the client machine as the major bottleneck when executing GridRPC applications. In summary, the experimental results have shown that the SmartGridRPC model can increase the performance of an application by:

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

REFERENCES

1. Seymour K, Nakada H, Matsuoka S, Dongarra J, Lee C, Casanova H. Overview of GridRPC: A remote procedure call API for Grid computing. *Lecture Notes in Computer Science* 2002; **2536**:274–278.
2. YarKhan A, Seymour K, Sagi K, Shi Z, Dongarra J. Recent developments in GridSolve. *International Journal of High Performance Computing Applications* 2006; **20**(1):131.
3. 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* 2003; **1**(1):41–51.
4. Caron E, Desprez F. DIET: A scalable toolbox to build network enabled servers on the Grid. *International Journal of High Performance Computing Applications* 2006; **20**(3):335.
5. Brady T, Guidolin M, Lastovetsky A. Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model. *International Conference on Grid Computing*, vol. 29, 2008; 49–56.
6. Higgins R, Lastovetsky A. Managing the construction and use of functional performance models in a grid environment. *International Parallel and Distributed Symposium*, Rome, Italy, 2009; 1–8.
7. Lastovetsky A, Reddy R, Higgins R. Building the functional performance model of a processor. *ACM Symposium on Applied Computing*, 2006; 746–753.
8. Wolski R, Spring N, Hayes J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 1999; **15**(5–6):757–768.
9. Caniou Y, Jeannot E. Multi-criteria scheduling heuristics for GridRPC systems. *International Journal of High Performance Computing* 2006; **20**(1):61–76.
10. Guidolin M, Lastovetsky A. ADL: An algorithm definition language for SmartGridSolve. *International Conference on Grid Computing*, Tsukuba, Japan, 2008.
11. Snavely A, Chun G, Casanova H, Van der Wijngaart RF, Frumkin MA. Benchmarks for grid computing: A review of ongoing efforts and future directions. *ACM SIGMETRICS Performance Evaluation Review* 2003; **30**(4):32.
12. Frey J, Tannenbaum T, Livny M, Foster I, Tuecke S. Condor-G: A computation management agent for multi-institutional grids. *International Conference on Cluster Computing*, Chicago, IL, vol. 5(3), 2002; 237–246.
13. Laure E, Fisher S, Frohner A, Grandi C, Kunszt P, Krenek A, Mulmo O, Pacini F, Prelz F, White J, Barroso M, Buncic P, Byrom R, Cornwall L, Craig M, Di Meglio A, Djaoui A, Giacomini F, Hahkala J, Hemmer F, Hicks S, Edlund A, Maraschini A, Middleton R, Sgaravatto M, Steenbakkers M, Walk J, Wilson A. Programming the Grid with gLite. *Computational Methods in Science and Technology* 2006; **12**(1):33–45.
14. Ryu D, Ostriker JP, Kang H, Cen R. A cosmological hydrodynamic code based on the total variation diminishing scheme. *Astrophysical Journal* 1993; **414**(1):1–19.
15. Guidolin M, Lastovetsky A. Hydropad: A scientific application for benchmarking GridRPC-based programming systems. *International Parallel and Distributed Symposium*, Rome, Italy, 2009; 1–8.
16. Brady T, Dongarra J, Guidolin M, Lastovetsky A, Seymour K. SmartGridRPC: The new RPC model for high performance Grid computing. *Technical Report UCD-CSI-2009-10*, University College Dublin, 2009; 1–55.
17. Cao J, Jarvis SA, Saini S, Nudd GR. Gridflow: Workflow management for Grid computing. *International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, 2003; 198.
18. Yu J, Buyya R. A taxonomy of workflow management systems for Grid computing. *International Journal of Grid Computing* 2005; **3**(3):171–200.
19. Spooner DP, Cao J, Jarvis SA, He L, Nudd GR. Performance-aware workflow management for Grid computing. *The Computer Journal* 2005; **48**(3):347.

DOI: 10.1002/cpe

20. Deelman E, Blythe J, Gil Y. Pegasus: Mapping scientific workflows onto the Grid. *Lecture Notes in Computer Science* 2004; **3165**:11–20.
21. Lovas R, Dozsa G, Kacsuk P, Podhorszki N, Drtos, D. Workflow support for complex Grid applications: Integrated and portal solutions. *Lecture Notes in Computer Science* 2004; **9999**:129–138.
22. Saltz J, Berryman H, Wu J. Multiprocessors and runtime compilation. *Concurrency Practice and Experience* 1991; **3**(6):573–592.
23. Koelbel C, Mehrotra P, Van Rosendale J. Supporting shared data structures on distributed memory architectures. *ACM SIGPLAN Symposium on Principles*, 1990; 186.
24. Amar A, Bolze R, Bouteiller A, Chis A, Caniou Y, Caron E, Chouhan P, Le Mahec G, Dail H, Depardon B, Desprez F, Gay JS, Su A. Diet: New developments and recent results. *Lecture Notes in Computer Science* 2007; **4375**:150–170.
25. Brady T, Konstantinov E, Lastovetsky A. SmartNetSolve: High level programming system for high performance Grid computing. *International Parallel and Distributed Symposium*, Rhodes Island, Greece, 2006.
26. Li Y, Dongarra J, Seymour K, YarKhan A. Request sequencing: Enabling workflow for efficient problem solving in GridSolve. *International Conference on Grid and Cooperative Computing*, Lanzhou, China, 2008; 449–458.
27. Lastovetsky A. Adaptive parallel computing on heterogeneous networks with mpC. *Parallel Computing* 2002; **28**(10):1369–1407.
28. Lastovetsky A, Reddy R. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers *Journal of Parallel and Distributed Computing* 2006; **66**(2):197–220.

DOI: 10.1002/cpe