Smart RPC-based computing in Grids and on Clouds

Thomas Brady[†], Oleg Girko[‡], Alexey Lastovetsky[‡]

[†]Openet, Dublin, Ireland

[‡]School of Computer Science and Informatics, University College Dublin, Ireland

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].

1.1 GridRPC programming model and API

The aim of the GridRPC model is to provide a standardised, 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).

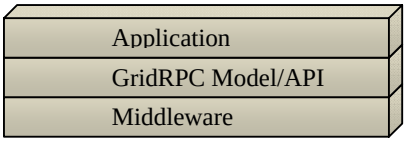

Figure 1: Overview of GridRPC model/API

This standardization 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.

1.1.1 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.

1.1.1.1 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.

1.1.1.2 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.

1.1.2 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.

1.2 GridRPC: A GridRPC application

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

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 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.

---

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

```
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();
}
```

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.

1.3 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 1.1.

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).

1.3.1 GridSolve: Agent discovery

The section outlines the GridSolve implementation of the "Register discovery" part of the GridRPC model. 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 2).

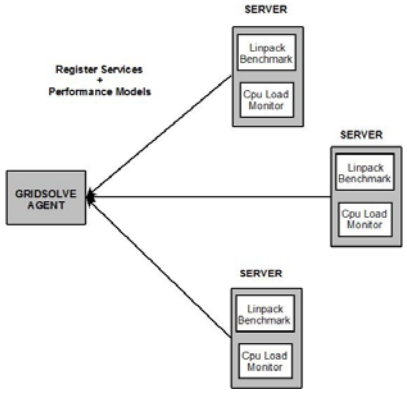

Figure 2: GridSolve – Agent discovery

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.

1.3.2 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.

1.3.2.1 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.

1.3.2.2 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. Also included are a set of mapping heuristics that rely on the other performance model in GridSolve called the Historical Trace Manager (HTM).

1.3.2.3 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.

1.4 GridRPC limitations

GridRPC 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.

2 SmartGridRPC and SmartGridSolve

SmartGridRPC [6] is 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.

2.1 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 3). 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.

2.1.1 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. In the future, SmartGridSolve will implement performance models such as the Functional Performance Model, which is described in [7] [8]. Other possible implementations could include the Network Weather Service [9], the MDS directories (Globus, Ninf) [4] and the Historical Trace Manager (GridSolve) [10]. 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 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 [11] 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.

2.1.1.1 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.

2.1.1.2 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.

2.1.1.3 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.

2.1.1.4 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.

2.1.1.5 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.

2.1.2 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. 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.

2.1.3 A SmartGridRPC application

Table 2 is the SmartGridRPC implementation of the GridRPC application in section 1.2. 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.

Let us consider the same simple scenario as in section 1.2, 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.

```
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();
}
```

2.2 SmartGridSolve: implementing SmartGridRPC in GridSolve

SmartGridSolve [12] is an extension of GridSolve, which makes the GridSolve middleware compliant with the SmartGridRPC model. SmartNetSolve [13] was previously implemented to make the NetSolve [14] 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.

2.2.1 Agent discovery

This section presents the SmartGridSolve implementation of the "register discovery" part of SmartGridRPC model outlined in section 2.1.1. 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.

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.

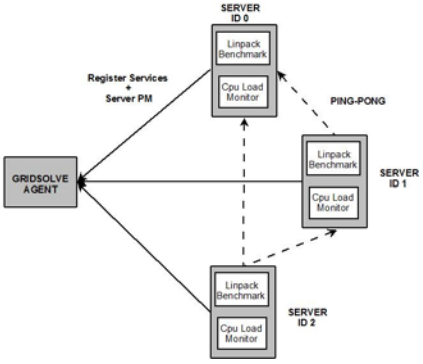

Figure 4: SmartGridSolve - Agent discovery

2.2.2 Run-time of client application

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

2.2.2.1 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.

2.2.2.2 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.

2.2.2.3 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.

2.2.3 Fault tolerance

SmartGridSolve maps tasks to servers collectively and allows server to server communication, so single server's failure can affect not only tasks running on that server, but also tasks dependent on those tasks.

There are two fault tolerance modes in SmartGridSolve.

2.2.3.1 Advanced mode

The advanced mode is the default mode implemented in `grpc_map()`function in SmartGridSolve. If there is a problem communicating with a server running a task, or the server reports a failure during `grpc_call()` or `grpc_call_async()`, the library enters the fault mode. The agent is notified that the server is unavailable and should be removed from the list of available servers. All remaining `grpc_call()` and `grpc_call_async()` calls do nothing, `grpc_local()` blocks are being skipped, and the `grpc_map()` loop is started over again in mapping phase.

The disadvantage of this mode is that even a single server's fault is expensive: it leads to all tasks inside `grpc_map()` block to be mapped and started over again. All results of computation done already inside `grpc_map()` block are lost.

The advantage of this mode is that it has no performance impact if there are no faults.

If several servers became unavailable during execution of a single `grpc_map()`, only the first fault is detected, all subsequent calls to `grpc_call()` and `grpc_call_async()` do nothing, and the remapping is performed before the next loop pass. At the moment of the next remapping the agent may have noticed some of the other failed servers, but it may take several mapping and execution attempts before the agent notices all failed servers.

2.2.3.2 Simple mode

The simple mode is implemented in `grpc_map_ft()` function. This function works the same way as `grpc_map()`, but with server-to-server communication disabled. As a result, a failure of a single GridRPC call does not affect other calls. The fault tolerance inside `grpc_map_ft()` is implemented the same way as in GridSolve: if there is any problem communicating with a server running a task, or the server reports a failure of that task, another server is selected, and the task is run again on that server.

The advantage of this approach is that each fault is relatively cheap: a server's fault affects only those tasks which are running on this server at the moment of the fault.

The disadvantage is that although tasks are still mapped to servers collectively, the communication optimization is sacrificed: all communication goes through the client. This has a significant negative impact on performance in absence of faults.

If several servers became unavailable during execution of a single `grpc_map_ft()`, each fault is handled independently, and each failed task is re-tried on a different server.

2.2.3.3 Additional constraints for fault tolerance in advanced mode

The current implementation of SmartGridSolve places additional constraints on the code inside `grpc_map()` block to work correctly in case of a task failure. The existing guideline is that all side effects should be done in `grpc_local()` block, which is skipped during discovery phase and is run only during the actual task execution phase (the last pass of the

loop). However, in case of task failure, the task execution phase is not the the last: the loop will start over again for the mapping phase. This means that if there is `grpc_call()` or `grpc_call_async()` after the `grpc_local()` block, and this call has failed, `grpc_local()` block preceding it has been already executed, producing side effects, and there is no way to roll those side effects back. Hence, here is the additional guideline for `grpc_local()` block to work correctly in case of failure: if there are more remote task calls after the `grpc_local()` block inside `grpc_map()` loop, the code inside this `grpc_local()` block should be idempotent: produce exactly the same results if it is run more than once.

3 Making SmartGridSolve smarter

3.1 SmartGridSolve approach to smart mapping and its limitations

The current approach to smart mapping implemented by SmartGridSolve is the following. A block containing an algorithm to apply smart mapping to is marked by the `grpc_map()` directive before the block. So, if the original algorithm is looking as follows:

```
grpc_call(task1, ...);
grpc_call(task2, ...);
grpc_call(task3, ...);
```

after converting it to use smart mapping it will look like this:

```
grpc_map("heuristics_name") {
    grpc_call(task1, ...);
    grpc_call(task2, ...);
    grpc_call(task3, ...);
}
```

The `grpc_map()` directive is actually a C preprocessor macro implemented as a `while` loop. The first pass of this loop discovers all GridRPC calls, but instead of executing them it just stores the information about calls and their arguments for building dependency graph. Before the second loop pass the dependency graph is built and sent to the agent for smart mapping. The agent responds with the suggested mapping, and the tasks are run on servers suggested by the agent during the second loop pass. If a fault happens during the execution of a task, the faulty server is excluded from the list of available servers, all subsequent task runs are being skipped, and the loop is run once again, asking for the agent for mapping and then running the tasks inside the loop pass.

This approach is very simple, but it has the following limitations.

- The current implementation of the `grpc_map()` directive puts significant restrictions on a code inside the bock following it. Although the fact that `grpc_map()` is implemented as a loop should be hidden as implementation detail, a user still has to keep this in mind. In particular, all statements besides GridRPC calls should be written to yield the same result when run during the first and second loop pass. User should keep in mind that the loop is run at least twice and avoid the statements with side effect inside `grpc_map()` controlled block. A special `grpc_local()` block was introduced to allow all statements which should not be run during the discovery phase to be put there. However, this does not fully protect those statements from being run twice in case of GridRPC call failure. This means that all statements inside `grpc_map()` block should be idempotent: either having no side effects, or if they have one, the side effect should be exactly the same when the statement is performed for the second time (except statements inside `grpc_local()` blocks not followed by any GridRPC calls). See section 2.2.3.3 for details. It's very easy to violate this restriction and there is no easy

way to detect if that happened. A simple mistake can lead to unpredictable results. Or even worse, a program can silently produce incorrect results only if a server fault has happened during algorithm execution.

- The current implementation does not handle branching inside `grpc_map()` block properly. If there is an `if` statement, the rules of C programming language dictate that only one branch is executed during the discovery phase. This means that no GridRPC calls in another branch will be taken into account when building the dependency graph.
- The implementation of `grpc_map()` block and GridRPC calls uses global state to determine which loop pass is taking place and what to do now. This makes GridRPC implementation non re-entrant and thread unsafe.
- If a fault happens during GridRPC call, all the mapping and the discovery, mapping and execution stages are performed again on the whole algorithm inside `grpc_map()` block, not taking into account that some tasks could have already successfully completed and their results are not lost.

3.2 Better approach to smart mapping

In order to achieve more correct smart mapping which puts less constraints on statements used inside the block to be mapped, other approaches are needed. Very few programming languages have facilities for introspection on statement level, and those languages are not widely used for grid computing anyway, so in order to discover GridRPC calls dependency graph correctly without executing the block in a “special mode”, some kind of additional tool is needed.

In order to discover dependency graph, the algorithm should be analysed somehow before it is run. As the algorithm does not contain self-modifying code, the dependency graph is the same during different algorithm runs, so it does not make sense to build it every time it's run. The better way to build dependency graph is to analyse the algorithm during the compilation phase or even before. Hence, the preprocessor approach (discovery before compilation) is proposed.

The preprocessor approach can be implemented by a program which reads a program written in a programming language, analyses it, finds all blocks marked for smart mapping, finds all GridRPC calls inside these blocks, builds dependency graph for all those blocks, and then outputs the program with blocks marked for smart mapping modified to contain dependency graph table, a call to the agent to get task-to-server mapping in the beginning of the block, and GridRPC calls which use this mapping.

The preprocessor approach can lift some restrictions on code inside `grpc_map()` block because there will be no discovery phase during runtime, just execution phase:

- there will be no need in `grpc_local()` blocks to keep some code from being executed during the discovery phase;
- all GridRPC calls in all branches of `if` statements will be taken into account when building the dependency graph, not just ones which happen to be executed during the discovery phase.

However this approach makes the solution for fault tolerance used in the current implementation of SmartGridSolve even less optimal: placing restrictions on code inside `grpc_map()` block just for the rare case of task failure. Better approach to lift remaining restrictions on code is proposed in section 3.3.1.

3.3 Better approaches to fault tolerance

The current implementation of fault tolerance in SmartGridSolve is very simple, and it's far from optimal. It has no performance overhead in case there are no faults, but if the fault happens the overhead is huge. All algorithm inside `grpc_map()` block is being essentially restarted in this case.

There are three main directions to improve fault tolerance in SmartGridSolve:

- lifting the restrictions on code inside `grpc_map()` block caused by fault handling implementation;
- reducing set of tasks to be restarted to only those tasks that have their results being lost;
- reducing set of tasks to be restarted by reducing likelihood of their results being lost;
- improving reliability of mapping and graceful handling of mapping failures.

3.3.1 Recovery from task failures

When the task-to-server mapping is found, everything else looks simple on the first glance: just execute the algorithm, making GridRPC calls according to the mapping. However, things become complicated if there is a GridRPC call which has failed. The problem is what to do when this happened. In a common case it's not enough to re-run the failed GridRPC call on a different server: some of the previous GridRPC calls results are lost if server-to-server communication is involved. This means that a new mapping should be built and those GridRPC calls which have results lost have to be re-run as well.

All GridRPC tasks have an interesting property: they are independent on each other, return the same results when called with the same arguments, and have no side effects. This means that in case of GridRPC failure, instead of rolling back the whole algorithm to a checkpoint, we can re-run all GridRPC calls which have results lost with exactly the same arguments as before, and we'll have exactly the same result.

In this approach, each GridRPC call should store its arguments in some kind of redo log. Also each GridRPC call should contain logic for fault recovery: ask agent for another task-to-server mapping using the same dependency graph and re-run the redo log or its part in case of failure.

The redo approach has the advantage that it completely solves the problem with side effects inside and outside of program state: nothing has to be rolled back, no program state need to be restored, no side effects need to be discovered. Combined with the preprocessor approach to discovering dependency graph described in section 3.2, it allows to completely lift the constraints on code inside `grpc_map()` block described in sections 2.2.3.3 and 3.1.

3.3.2 Restarting only relevant tasks

3.3.2.1 Not remapping completed tasks

The current SmartGridSolve implementation remaps and executes again even those tasks that were successfully completed, and results were successfully retrieved by the client, even if there are no more tasks which need those results.

The simplest and and most obvious improvement is to take completed tasks into account and exclude those of them which have final results retrieved by the client from the remapping in case of other task's failure. It's necessary to remap only those tasks which were not completed or have other tasks which are dependent (directly or indirectly) on their results.

The correct implementation of this improvement decreases performance penalty in case of a

failure and has negligible performance impact if there are no failures.

3.3.2.2 Individually restarting tasks not dependent on other ones

The most difficult problem to solve in case of a fault during collective mapping is the loss of results which were produced by other tasks. But the simple case if there is no data from other tasks needed by the failed task can be handled without a complete remapping. If there are no data loss from other tasks, the failed task can be restarted individually.

The correct implementation of this improvement decreases performance penalty in case of a failure and has negligible performance impact if there are no failures.

3.3.3 Losing less results

3.3.3.1 Keeping task results until all tasks dependent on them finished

Another direction of improvement of fault tolerance is to prevent losing the results of completed tasks. The current SmartGridSolve sends the results of tasks to the tasks dependent on those results directly. The sending server removes the results when the transmission is complete, so the results are lost in case of the receiving server's fault after the results were transmitted. This means that the failed task can not be restarted individually: the data needed for this task are lost and have to be recalculated by running all the tasks necessary to produce those data again.

Recalculating lost results can be avoided if they were stored somewhere else, not only on failed server. It can be implemented by caching data on the server which produced the results until all tasks which need those results are completes and passed their results to the client or to other servers.

This approach has no additional overhead besides using storage for the results for longer period of time. However, this still results in high performance penalty if the server which keeps the results and the server running a task using those results failed simultaneously. In this case the results are lost and have to be recalculated by rerunning all tasks the lost data depend on.

3.3.3.2 Redundant storage for intermediary results

To even more decrease probability of data loss, more generic approach can be used for storing task results redundantly. For example, task results can be not only kept on the server where those results were produced and on the servers which run tasks which need those results, but also on some servers which are not supposed to run tasks which need those results. The agent can keep track of the servers where results are stored and instruct them to send the data to the servers where remapped tasks will be run in case of a failure. The number of servers to store the same data redundantly should be configurable.

This approach can be optimised further by sending results for redundant storage to those servers which are the most likely candidates for running tasks which use those results in case of a server failure, and by taking into account where data stored when remapping tasks.

3.3.4 More reliable mapping

3.3.4.1 Redundant agents

There is a problem in the current SmartGridSolve implementation: the agent is a single point of failure, and there is no recovery in case of the agent's failure. Servers are redundant, and tasks running on a failed server can be retried on another server. However, the failed agent can make the whole grid unusable by making all clients to be unable to map or remap all further tasks.

This problem can be solved by running multiple synchronised instances of the agent. A server can register with any agent instance, and this information will be propagated among all instances. The protocol of server-agent and client-agent communication can be extended by allowing them to reconnect to another agent instance in case of communication error.

3.3.4.2 Improved mapping error handling

There is another issue which is not directly related to performance in case of server failure, but it is related to the handling of agent failure. The current SmartGridSolve implementation has no way to handle task mapping failure gracefully. This is because the condition inside `while()` loop in plain C has no way to communicate additional information besides being true or false. If there is an error during task mapping, the client falls back to individual mapping for all further tasks, and there is no way to switch to smart mapping for further task groups, even if the agent became available again.

The handling of mapping failure can be improved in preprocessor approach by introducing syntax to handle such kind of failure.

4 Smart RPC-based computing on Clouds: Adaptation of SmartGridRPC and SmartGridSolve to Cloud computing

4.1 Introduction

Cloud computing allows users to have easy access to large distributed resources from data centers from anywhere in the world. Moreover, by renting the infrastructure on a pay per use basis, they can have immediate access to required resources without any capacity planning and they are free to release them when resources are no longer needed. These resources can be dynamically re-configured to adjust to a users needs or the variable load of applications allowing for an optimum resource utilization. Hence, the computing infrastructure can be scaled up and down according to the application requirements or the budget of the user or a combination of both.

SmartCloudSolve (SCS) is a Platform as a Service (PaaS). PaaS are platforms which facilitate deployment of cloud applications without the cost and complexity of buying and managing the underlying hardware and software layers. SCS is based on SmartGridSolve [6] which is an extension to GridSolve [3], the middleware platform for Grids and Clusters. SCS allows application programmers to easily convert existing applications into high performance scalable applications that will run on a number of cloud infrastructures Other examples of PaaS include, Google AppEngine [15], MicroSoft Azure [16] and Force.com [17].

The following are the advantages of using SCS over other PaaS:

- Increased performance – SCS can automatically generate a task graph for an application and this task graph is used to improve the mapping of the application on the cloud infrastructure and thus increase the performance of the application.
- Automation – SCS can improve the scaling up and scale down of the cloud infrastructure by using the task graph. SCS can determine future levels of parallelism of the application from its task graph and therefore can scale the infrastructure up ahead of time to maximize performance. Since future levels of parallelism can be determined, SCS can also terminate servers if they are not required for future tasks and therefore reduce costs.
- Ease of use – With a few simple steps an application programmer can change their existing serial applications into high performance scalable applications that can execute in a cloud environment.

- Flexibility – The application programmer can choose from a number of programming languages and their application is not restricted to run on a single public or private cloud.

4.2 Cloud Computing

In this section we will describe the three major catogories of cloud computing. These are: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) which is the catogory the SCS platform falls under and Software as a Service (SaaS).

4.2.1 Infrastructure as a Service

Infrastructure as a Service (IaaS) or Hardware as a Service (HaaS) are terms that refer to the practice of delivering IT infrastructure based on virtual or physical resources as a commodity to customers. These resources meet the end user requirements in terms of memory, CPU type and power, storage, and, in most of the cases, operating system. Users are billed on a pay per use basis and have to set up their system on top of these resources that are hosted and managed in data centers owned by the vendor. Amazon is one of the major players in providing IaaS solutions. Other providers include Rackspace [18], GoGrid [19] and FlexiScale [20].

4.2.2 Platform as a Service

Cloud systems can offer an additional abstraction level: instead of supplying a virtualized infrastructure, they can provide the software platform where systems run on. The sizing of the hardware resources demanded by the execution of the services is made in a transparent manner. This is denoted as Platform as a Service (PaaS). Platform as a Service solutions provide an application or development platform in which users can create their own application that will run on the cloud. PaaS implementations usually provide users with an application framework and a set of API that can be used by developers to program or compose applications for the cloud. The two major players adopting this strategy are Google and Microsoft. Google AppEngine is a platform for developing scalable web applications that will be run on top of server infrastructure of Google. It provides a set of APIs and an application model that allow developers to take advantage of additional services provided by Google such as Mail, Datastore, Memcache, and others. By following the provided application model, developers create applications in Python. These applications will be run within a sandbox and AppEngine will take care of automatically scaling when needed. Azure is the solution provided by Microsoft for developing scalable applications for the cloud. It is a cloud service operating system that serves as the development, run-time, and control environment for the Azure Services Platform. By using the Microsoft Azure SDK, developers can create services that leverage the .NET Framework. These services have to be uploaded through the Microsoft Azure portal in order to be executed on top of Windows Azure. Additional services, such as workflow execution and management, web services orchestration, and access to SQL data stores, are provided to build enterprise applications.

4.2.3 Software as a Service

Software as a Service solutions are at the top end of the cloud computing stack and they provide end users with an integrated service comprising hardware, development platforms, and applications. Examples of the SaaS implementations are the services provided by Google for office automation, such as Google Document and Google Calendar, which are delivered for free to the Internet users and charged for professional quality services. An example of a commercial solution is Salesforce.com [21], which provides an on line customer relation management service.

4.3 SmartCloudSolve (SCS)

4.3.1 Overview

SCS is a PaaS that allows programmers to easily develop applications or convert their existing applications into high performing scalable applications that will execute on a cloud infrastructure. SCS will give the application programmer the freedom to develop their application using any one of a number of languages and execute that application on any one of a number of cloud infrastructures. Therefore, programmers can focus on software development in their language of choice as opposed to focusing on managing the underlying cloud infrastructures (servers, network, storage etc) and determining how the application should be executed on that infrastructure.

By using the simple SmartGridRPC API, applications written in a number of languages can be easily converted into high performing scalable cloud applications. Applications written using the SmartGridRPC API are interpreted by SCS, which then generates a task graph automatically for the application. The task graph is a directed acyclic graph representing the order, computational load, communication load of each task in the application and the data dependencies between these tasks. This task graph is used to determine a close to optimal solution for executing the application on the cloud. The mapping outlines how tasks are mapped to sever and how communication will be sent between these tasks (inter-server communication). SCS design also takes advantage of the processing power of each individual CPU core in the underlying infrastructure by using the task graph to leverage message passing, process forking and threading. In addition the task graph is used to determine ahead of time, when to automatically scale up and scale down the cloud infrastructure and can thus minimize time and cost.

4.3.2 Advantages of the SCS Platform

The main advantages of SCS over existing cloud PaaS is that you can implement applications in any number of languages and these application can potentially achieve higher performance. The application programmer creates an application using their chosen language and the SmartGridRPC API. SCS applications can be developed in Fortran, C, Matlab, Mathematica, and Octave. The platform is extensible to include other languages. Therefore, the application programmer will not be locked-in to a specific development language, which is the case with most other PaaS. With Google, for example programmers must write their applications only in the Python programming language to Google specific APIs. “Regular” applications which have been implemented to run on a single machine can be implemented for SCS to execute on a cloud infrastructure with only a few simple changes. In addition higher performance can be achieved since the task graph of the application is automatically generated and is known prior to execution. This can be used to determine the scaling of the infrastructure and to implement a close to optimal execution of the application on that infrastructure.

The other advantage of SCS is there is no infrastructure lock-in. When an application is developed for SCS it will not be locked in to any one infrastructure. The application is “cloud neutral” which is not the case with other PaaS. For example, if you develop an application using Force.com, which is the SalesForce PaaS or Google AppEngine, it is impossible to move these applications to other cloud infrastructures. SCS will be developed so that applications can be easily moved from one cloud provider to another or even to a private cloud.

4.3.3 High Level Design of the SCS Platform

Figure 5 shows an high level view of the design of SCS. In the software layer, the application programmer creates an application using their chosen language and the

SmartGridRPC API. In the platform layer, SCS automatically generates a task graph when the application is run, which is then used to optimally map the application to the cloud infrastructure. The infrastructure is scaled up and down based on the level of parallelism in the task graph. In addition, using this graph the load of computation and communication is balanced over the network and inter-server communication is implemented to further increase the performance. The task graph could also be used to execute parallel fine grain tasks on multiple processors on the same machine (OpenMP [22] /Pthreads [23]) or on multiple processors on different machines (MPI [24]).

In the infrastructure layer, SCS executes the application on the target infrastructure. The original target infrastructure for SmartGridSolve which SCS is based on is Grids and clusters. SCS supports many different architectures/platforms such as Linux, Solaris, BSD, Windows and Windows Compute Cluster. This means that SCS can easily be extended so that applications can be executed across a number of public clouds or even private clouds. Namely, SCS could be extended for other public clouds such as Rackspace, GoGrid, Flexiscale and private clouds such as the Eucalyptus cloud [25].

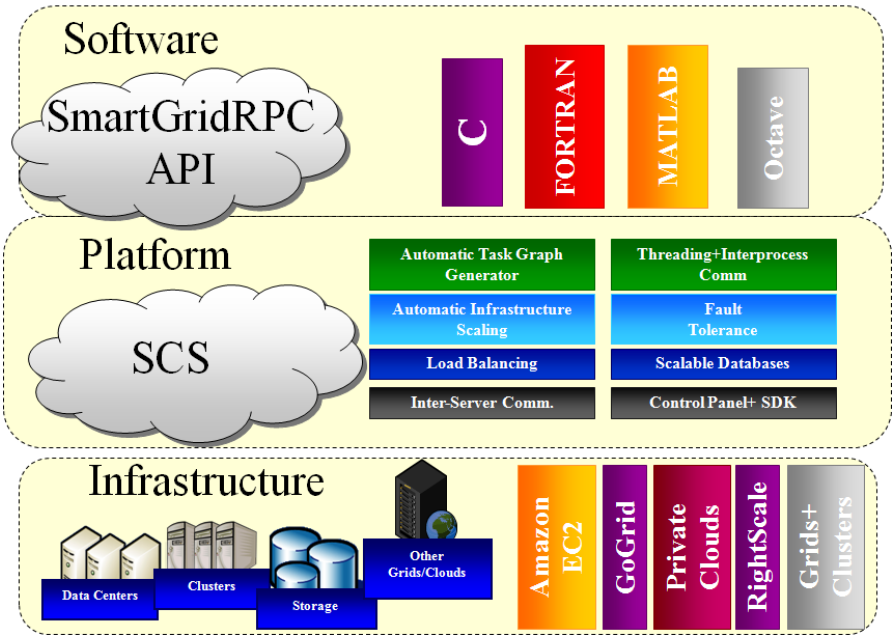

Figure 5: Overview of SmartCloudSolve

4.3.4 SCS API and Application Implementation

This section will describe how a programmer can easily convert their serial application into a high performance scalable application using the SmartGridRPC APIs. SmartGridRPC model is the programming model that the SCS platform is based on which was described in section 2.1.

Hydropad [26] is a real-life astrophysics application that simulates the evolution of clusters of galaxies in the Universe. Hydropad requires high processing resources because it has to simulate an area comparable to the dimensions 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 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. Figure 6 shows the serial implementation of the main loop of the Hydropad that simulates the evolution of the

Universe. At each iteration of the loop, the *grav* function calculates the gravitational effect the matter has on each other. The *dark* function and the *bary* functions then use the result of the *grav* function to calculate their new respective positions. The *grav* function then uses the positions of the particles in the previous evolution step to calculate the gravitational result in this evolution step. This is done cyclically for each evolution step of the Universe that is simulated.

Table 3 shows the same Hydropad application implemented as a serial application. Table 4 shows the same application implemented for SCS. When implementing a SCS application, there are three areas that the application the programmer needs to identify. These are the remote computation part of the application, the local computation part of the application and the part of the application that will be mapped collectively.

```
t_sim=0;
while(t_sim < t_total){
   grav(phiold, ….);
   if(t_sim==0){
      initvel(phi, …);
   }
   dark(x1, ..);
   bary(n, …);
   t_sim+=t_step;
}
```

Table 3: Hydropad serial application

```
#include “gridrpc.h”
t_sim=0;
gridrpc_map(){
   while(t_sim < t_total){
      gridrpc_call (grav , phiold, ….);
      if(t_sim==0){
         gridrpc_call (initvel , phi, …);
      }
      gridrpc_call (dark, x1, ..);
      gridrpc_call (bary , n, …);
      gridrpc_local(){
         t_sim+=t_step;
      }
   }
}
```

Table 4: Hydropad cloud application

To convert a serial application in to a cloud application the application programmer must first identify functions for remote execution using the *gridrpc_call* function. In this example, the programmer has identified the *grav*, *initvel*, *dark* and *bary* functions for remote execution. The SCS system may now execute these tasks locally or remotely in the cloud depending on which gives higher performance. Secondly, the parts of the application that is executing local computation need to be identified. In this case, the local operation on t_*sim* has been identified using the *grpc_local* function. The region of the application which will be mapped and executed collectively is identified using the parenthesis of the *gridrpc_map* function.

This application is now a high performance scalable cloud application. SCS can now generate a task graph for this application and can therefore scale up and down the cloud infrastructure, balance the load of computation and communication and perform inter-server communication to maximize the performance of the application.

5 Acknowledgement

This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/IN.1/I2054.

6 Bibliography

[1] A. Birrell and B. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 2, pp. 39-59, 1984.

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

[3] A. YarKhan, K. Seymour, K. Sagi, Z. Shi and J. Dongarra, "Recent Developments in GridSolve," *International Journal of High Performance Computing Applications*, vol. 1, no. 20, p. 131, 2006.

[4] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura and S. Matsuoka, "Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing," *Journal of Grid Computing*, vol. 1, no. 1, pp. 41-51, 2003.

[5] E. Caron and F. Desprez, "DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid," *International Journal of High Performance Computing Applications*, vol. 3, no. 20, p. 131, 2006.

[6] T. Brady, J. Dongarra, M. Guidolin, A. Lastovetsky and K. Seymour, "SmartGridRPC: The New RPC Model for High Performance Grid Computing," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 22, pp. 2467-2487, 2010.

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

[8] A. Lastovetsky, R. Reddy and R. Higgins, "Building the Functional Performance Model of a Processor," *Proceedings of the 21st Annual ACM*

Symposium on Applied Computing (SAC 2006), pp. 746-753, 2006.

[9] R. Wolski, N. Spring and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Journal of Future Generation Computing Systems*, vol. 5, no. 15, pp. 757-768, 1999.

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

[11] T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen and R. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 6, no. 61, pp. 810-837, 2001.

[12] T. Brady, M. Guidolin and A. Lastovetsky, "Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model," *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid2008)*, no. 29, 2008.

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

[14] H. Casanova and J. Dongarra, "NetSolve: A Network Server for Solving Computational Science Problems," *In Proceedings of High Performance Computing Applications*, vol. 3, no. 11, p. 212, 1997.

[15] E. Ciurana, Developing with Google AppEngine, Apress, 2009.

[16] R. Jennings, Cloud Computing with the Windows Azure Platform, Wrox, 2009.

[17] J. Ouellette, Development with the Force.com Platform: Building Business Applications in the Cloud, Addison-Wesley, 2009.

[18] [Online]. Available: http://www.rackspace.com/.

[19] [Online]. Available: http://www.gogrid.com/.

[20] [Online]. Available: http://www.flexiscale.com/.

[21] C. McGuire, C. Roth, D. Carroll and N. Tran, SalesForce.com Fundamentals: An Introduction to Custom Application Development in the Cloud, SalesForce.com, 2008.

[22] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, no. 5, pp. 46-55, 1998.

[23] B. Nichols and D. Buttlar, Pthreads programming, O'Reilly Media, 1996.

[24] W. Gropp, E. Lusk and A. Skjellum, Using MPI: portable parallel programming with the message-passing interface, The MIT Press, 1999.

[25] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," in

International Symposium on Cluster Computing and the Grid, 2009.

[26] M. Guidolin and A. Lastovetsky, "Grid-Enabled Hydropad: a Scientific Application for Benchmarking GridRPC-Based Programming Systems," in *International Parallel and Distributed Symposium*, 2009.

[27] D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih and T. Lewis, "A programming environment for heterogeneous distributed memory machines," in *6th IEEE Heterogeneous Computing Workshop (HCW`97)*, Geneva, 1997.