
# Performance of GridRPC-based programming systems for distributed scientific computing: issues and solutions

*Author:*

Michele Guidolin
B.Sc., M.Sc.

The thesis is submitted to
UNIVERSITY COLLEGE DUBLIN

for the degree of
PHD IN COMPUTER SCIENCE

in
COLLEGE OF ENGINEERING, MATHEMATICAL
& PHYSICAL SCIENCES

September 2009

SCHOOL OF COMPUTER SCIENCE & INFORMATICS

*Head of School:*
Dr. Joe Carthy

*Supervisor:*
Dr. Alexey Lastovetsky

*To my love Katie,*

*to my parents Teresio and Silvana,*

*and to my brother Claudio*

# Contents

# List of Tables

# List of Figures

**Acknowledgements**

This thesis would have never been possible if it wasn't for the help of my supervisor Alexey. Thanks to his knowledge, presence, suggestions and guidance, I was able to carry out this research.

I would like to thank the guys in the HCL laboratory, Thomas, Robert, Brett, Vladimir, Maureen, Ravi, and Xin, for the help during this experience. Particularly Thomas, for the work together on the project, and Robert, for all the time he help in problem solving.

During these years, I met a lot of new friends that helped me in this adventure: Davide, Luis, Alfredo, Jose and Andreas for the fun time together; Lisa and Patrizio for the bowling games; Thomas and Francesco that had to endure me at lunch time; Britta, Martin and Paulo for all the good times in the flat; Marco, Claudio, Mattia and Lenka for the lovely pizza nights; and the volleyball people for the endless hours of play. I will be never able to show them how much important they have been for me. By the way, thanks Thomas for the help with the cover letter.

I would like to thank my parents for the continuous drive to finish my thesis ("*work you lazy ...*") and moreover for the endless hours of support by phone during the difficult times. Furthermore, I would also like to thank my brother Claudio for always reminding me of my final goal ("*get a job!*"); It looks like I finally got it.

Last but not least (or least but not last, I'm never sure about these things), a big thanks goes to Katie who, despite not letting me eat her brain, gave me the best reason for finishing this thesis (and she corrected it). Grazie mille Katie.

Michele Guidolin

**Summary**

The GridRPC programming model is a standard designed for easy development of distributed scientific applications for Grid computing, while obtaining high performance. As there are already many specialised systems for embarrassing parallel and stream applications, GridRPC needs to obtain good performance with tightly synchronised applications in order to reach a large audience.

Unfortunately, a comprehensive performance analysis of the GridRPC model for these applications is missing. In this work, we present Hydropad, a real life task parallel tightly synchronised astrophysical simulation, as a method to evaluate the performance of GridRPC systems. The results show that the GridRPC version of Hydropad can achieve faster computation than the sequential code but is limited by the client-to-server connection speed and the client memory. As a result the client-side hardware is a performance bottleneck. GridRPC is therefore not the ideal paradigm for executing tightly synchronised task parallel distributed applications on a Grid environment.

SmartGridRPC aims to overcome the limitations found in GridRPC. This is done by implementing server-to-server communications and mapping of groups of tasks. Our experiments show that the SmartGridRPC version of Hydropad obtains significant performance gains in comparison to both the GridRPC implementation and also the sequential one. Furthermore, these performance gains are not influenced negatively by the client hardware. Our analysis shows that SmartGridRPC is an effective alternative for executing tightly synchronised task parallel distributed applications on a Grid environment.

To get these improvements, a SmartGridRPC middleware needs a task graph that fully represents the application's algorithm. The automatic task graph generation method introduced by SmartGridRPC may not always work for all kinds of algorithms. To overcome this, we developed the Algorithm Description Language (ADL) which allows the explicit generation of a task graph for any given algorithm. The results show that SmartGridRPC with ADL obtains higher performance than without ADL and using only GridRPC.

# Chapter 1

# Introduction

Distributed computing allows a scientific user to connect together the resources of different platforms that are available in an institution in order to execute larger scientific applications. Grid Computing [47] expands this idea by allowing various geographically and managerially separated powerful systems to be grouped together while keeping the access to their resources as simple as possible.

GridRPC [75] is a standard promoted by the Open Grid Forum that allows a scientific user to design an application to interface smoothly with a Grid environment. The motivation that led to the conception of the GridRPC model [74] was the need to have a programming model that simplifies the development of scientific Grid applications, in order to obtain widespread adoption of Grid computing by scientific users that are not grid specialists. Therefore, it was important to develop a programming model that can hide the technicalities and difficulties of interfacing with a Grid environment through a middleware. For these reasons a remote procedure call (RPC) model [16] adjusted for Grid computing has been chosen since this model was already familiar to a large amount of scientific users and it can be easily picked up by new programmers. The GridRPC programming model is easier than the RPC one since the programmer does not need to specify the server to execute the task and the stub for each remote task. Furthermore, GridRPC extends RPC since it adds asynchronous remote task calls. Currently, various Grid middleware systems implement the GridRPC model, such as GridSolve, Ninf-G and DIET.

In this thesis, we found that the GridRPC mapping and communication models have some limitations that can impact heavily on the performance of distributed scientific applications. A GridRPC middleware, due to the GridRPC programming model, works by individually mapping the applications tasks to appropriate servers in the Grid. This model supports minimisation of the execution time of each individual task of the application rather than the minimisation of the execution time of the whole application. Furthermore, the individual mapping of tasks implies that a GridRPC middleware is unaware of any data dependencies between tasks. Therefore, all the data objects used in a remote task can be communicated only between the server and the client machine without direct server- to-server communication. The drawback of this communication model is that for each task there is a high quantity of data communication on the client-to-server network link.

The work in this thesis is part of the SmartGridRPC [17] project, which is a new programming model aimed to overcome the GridRPC limitations by extending its single-task map and client-to-server communication models without changing the base of the GridRPC programming model. The SmartGridRPC model implements server-to-server communications and mapping of groups of tasks while adding only two new methods to the GridRPC API. In order to collectively map a group of tasks and to use a fully connected network, a SmartGridRPC middleware needs a task graph that represents the full knowledge of all the tasks executed in the applications algorithm. The task graph, a direct acyclic graph (DAG) structure, highlights the order of tasks and their synchronisation (whether they are executed in sequence or parallel), the dependencies between tasks, the load of data communication and the task computational volume [18]. An important new feature of SmartGridRPC is the automatic task graph generation method that allows the building of the task graph directly from the application code by iterating twice through the code that contains the task calls to be mapped collectively.

## Problems and Motivations

The main problem of GridRPC, and therefore the main motivation of this thesis, is that there has been no comprehensive performance analysis of the GridRPC model

showing its potential and limitations for tightly synchronised applications. The overwhelming majority of applications chosen, or artificially created, to demonstrate the performance of GridRPC middlewares are pipelined or embarrassing parallel (as analysed in section 2.2 of this thesis). We do not know if the GridRPC model has the potential to achieve good performance for a wide range of scientific applications, since performance evaluations of GridRPC middlewares with tightly synchronised applications are missing and it is difficult to obtain top performance in a distributed environment for such applications because of their high level of data communication between tasks. 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 tightly synchronised application that shows the eventual limits and benefits of the middleware tested.

Furthermore, the new SmartGridRPC programming model suffer the same issue since we also do not know if the new extensions implemented in SmartGridRPC, the mapping of group of tasks and the improved communication model, allow tightly synchronised applications, as well any other scientific distributed applications, to fully take advantage of the Grid environment. Therefore, it is important to evaluate also the performance potential of SmartGridRPC and compare with the GridRPC model.

In this thesis we also analyse that the automatic task graph generation method of SmartGridRPC has some restrictions that can impact the performance of various applications. A task graph may not always be automatically generated for all kinds of algorithms. There are different situations where the automatic task graph generation will not work. A typical example is when, in the code to be mapped, a conditional construct exists that checks a value that cannot be known without executing a remote task call. We show that a technique to avoid this problem is to create the task graph from a smaller block of code, but 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. Since the application programmer usually has an in depth knowledge of the algorithm used inside his application, he or she 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.

It is essential to resolve these issues given that high performance in the execution of any type of distributed scientific application is an important design objective of the GridRPC model, and thus of the SmartGridRPC model, since the target users are computational scientists. If we consider that many available systems for Grid computing are highly specialised in executing embarrassing parallel and pipelined applications and are already widely used; the main aim of GridRPC and SmartGridRPC must be to achieve high performance in the execution of tightly synchronised distributed scientific applications. This is the only way to allow GridRPC and SmartGridRPC to be used by a large audience 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.

## Contributions

This thesis contains two main contributions, the first is the performance evaluations of the GridRPC and SmartGridRPC models using a tightly synchronised real-life astrophysical application, Hydropad. The second is the design and implementation of a specific high level language, Algorithm Description Language (ADL), that can be used by the applications programmer to directly specify a task graph like structure into SmartGridRPC. Broken down further, the main contributions are:

**The design and implementation of Grid-enabled Hydropad [55]:** This is a real-life astrophysical application that was "gridified" in order to be used as a tool for experimental performance evaluation of GridRPC and SmartGridRPC systems. This application is composed of tasks that have a balanced ratio between computation and communication with a high level of data synchronisation between them. Therefore, this application can be classified as a tightly synchronised application. Hydropad requires high processing resources because it has to simulate an area comparable to the dimensions of the universe and simultaneously try to achieve a high enough resolution to show how the stars developed.

**The analysis of the benefits and limitations of GridRPC model:** Where we show that despite the fact that Hydropad is not the most suitable application to be executed on a Grid environment (because of the high magnitude of data communication involved between tasks), the GridRPC version of Hydropad obtains many non performance and performance related benefits, which can be applied to many other scientific applications. Our analysis shows that the GridRPC implementation of Hydropad can achieve better performance than the original sequential code. This is due to the performance advantages of the GridRPC model, namely, the remote execution of tasks on powerful servers and the parallel execution of the tasks. However, as previously mentioned, we show that the mapping and communication models utilised by GridRPC are not optimal and thus the increase in computational performance using GridRPC depends on the client-to-server links speed and specifically the client side hardware.

**The analysis of the benefits and limitations of SmartGridRPC model:** Here we show that the SmartGridRPC implementation of Hydropad does not only keep the non performance related benefits but can significantly improve the performance of the application. This applies even in situations where the GridRPC implementation fails to do so, and we also show that these performance gains are not deteriorated by the client side hardware. However, in order to obtain these performance improvements, SmartGridRPC needs a task graph that is representative of the underlying algorithm of the application and therefore we present three trivial examples of irregular algorithms where the automatic task graph generation method fails to build a representative task graph.

**The development and implementation of the ADL language:** This is a tool to help an application programmer easily specify a task graph for all kinds of algorithms. We demonstrate that the use of ADL in conjunction with SmartGridRPC improves the performance of example applications, that are comprised of irregular algorithms, over the individual use of SmartGridRPC and GridRPC. The ADL language is modular, it has a well-defined structure and its syntax is similar to C language. The objectives of this language are to be easy to use and easy to understand. Its integration with the SmartGridRPC model is straightforward and

it permits a user to select the flow of execution and the relative task complexity dynamically from the client code. The idea behind the development of ADL is to give a powerful tool to the programmer that help him or her implement any SmartGridRPC application with the best mapping and execution possible.

## Structure

The thesis is outlined as follows. Chapter 2 introduces briefly the distributed and Grid computing fields and the RPC model. Then, it presents the GridRPC model and the existing GridRPC middlewares. The last part of the chapter contains a literature review of the different applications used to evaluate the performance of GridRPC middlewares. Chapter 3 presents Hydropad, its implementation in GridRPC and the non-performance related benefits of GridRPC, that can be applied to Hydropad as well as to any other distributed scientific application. Furthermore, this chapter introduces a classification of Grid applications that compares the difference in computation of the tasks, the amount of memory and data used and inter-task communications required between different types of scientific applications. In chapter 4, we outline the performance related benefits and the eventual limits that the GridRPC model delivers to any scientific application by analysing the experimental results obtained using Grid-enabled Hydropad.

Chapter 5 presents the new SmartGridRPC model and the SmartGridSolve middleware, which extends the GridSolve middleware to implement this new model. Furthermore, this chapter shows the SmartGridRPC implementation of Grid-enabled Hydropad. In chapter 6, we use the various implementations of Hydropad to evaluate the performance potential that the new SmartGridRPC model delivers in comparison to the GridRPC model.

Chapter 7 presents three example algorithms where the automatic task graph generation of SmartGridRPC fails. These examples model real-life irregular algorithms that are common in many scientific applications. In this chapter, we analyse the problems behind the automatic task graph generation and we outline the GridRPC and the eventual SmartGridRPC implementations of these example algorithms. In chapter 8, we introduce how ADL can be used to generate a representative task graph for the example irregular algorithms. We also analyse

the performance improvements obtained by the use of ADL in conjunction with SmartGridRPC and compare them with the GridRPC and SmartGridRPC implementations. Furthermore, this chapter contains a literature review of existing languages that are used to generate task graphs or similar structures for systems in the Grid computing field and for GridRPC middlewares. Finally, chapter 9 presents an in depth description of the ADL language and its compiler.

# Chapter 2

# State of the Art

Scientific applications are designed to simulate, analyse and solve problems in many different fields (such as physics, biology, chemistry, etc) through the use of mathematical models. Their internal structure, algorithm and composition vary vastly according to the field or subject considered. These applications usually need high amounts of computational power and memory footprint to solve the mathematical models with high accuracy. Therefore, they are executed in specific systems that have enough hardware resources to handle their complexity. Unfortunately, powerful systems are not easy to access since they are expensive and difficult to manage. Thus, a common situation is that the systems available to a scientific user do not have enough resources to compute the desired large problem.

Instead of utilising a single powerful system, a common solution is to use distributed computing. The idea behind distributed computing is not recent and it is to connect together all the resources of different platforms available in an institution to execute the required scientific application. These distributed resources are accessed by dividing the computational problem of the application into many tasks. These tasks are executed, synchronously or asynchronously, on the various remote platforms and the data is communicated between each task through the use of a common technique or paradigm. Therefore, distributed computing permits a scientific user to easily access all the available resources.

Thanks to its many advantages, the field of distributed computing for computational science has been studied and researched for many years. However, in

the past not all types of scientific applications were executed in a distributed environment, due to the limitations on speed and latency of the network links. The ideal application to get top performance in a distributed environment is composed of coarse-grained tasks. These tasks have high computation and minimal data dependencies between them. Due to these characteristics, scientific applications with this type of task were the most developed and executed in distributed environments since they were the most able to gain from the remote and distributed execution of tasks.

A common technique used to execute coarse-grained scientific applications in a distributed environment is batch processing. In this technique, the various tasks of an application are developed as individual programs, called jobs. These jobs are executed in batches through the use of script languages or remote shells. Another typical approach is the use of batch management systems, such as Condor [66] and PBS [57]. They provide a high level of abstraction to easily manage remote jobs and their executions, such as job queueing, scheduling, workflow definition, resource monitoring and management. These techniques are still used today since they are very effective for this type of scientific application. However, they were not the only distributed computing paradigms developed. An important technique was the remote procedure call (RPC) model [16]. It was developed to achieve remote task execution in an easy way by directly using the high level language of the application.

The RPC model provides a straightforward and simple programming model for executing tasks on remote computers. To execute a task remotely, the application programmer does not need to learn a new programming language but merely uses an RPC method. The main idea behind RPC is that a remote call is as similar as possible to a normal function call of the underlying language. There are many different implementations of the RPC model. However, a typical RPC function call consist of 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, the location on the user's computer where the results will be stored and the encoding and decoding method used during the communication of the data.

The RPC model is based on a client-server architecture. The application, that implements the RPC API, acts as a client while the remote systems run the server

components. At run-time during the RPC method call, the client sends the input data to the given server. Then, the server executes the specific procedure using the communicated data. When the procedure is completed on the server, the output data generated are communicated back to the client. During this process, while the task is being executed remotely, the application is blocked inside the RPC method. Then, when the client receives the data back, the RPC call is finished and thus the application can continue the computation. The various software components, which implement this model of execution (for example, the data communication model, the remote executions, the scheduling, etc), are called the middleware of an RPC implementation. The information used to execute tasks remotely is specified by the application programmer in special files called stubs.

Recently, due to improvements in software tools and communication infrastructures, the concept of distributed computing has been expanded into a new field. The concept is that various geographically and managerially separated powerful systems can be grouped together in a common computational network and the resources of this network can be easily accessed by a scientific user like an electrical grid; this is called Grid computing [47]. The main objective of Grid computing is the ease of use, ease of development and the single point of access to the available resources. Therefore, a Grid environment could have resources located in the same place or scattered around the world and managed by different entities but their access would be through the use of a single login and a common method. Another objective of Grid computing, important for scientific applications, is the ability to reach high computational performance. Thanks to the low latency and fast communication speed of the recent network links, scientific Grid applications can benefit from the use of distributed resources also if they have tasks with finer granularity, thus with lower computation and higher data dependencies than previously possible.

Snavely *et al.* in [76] identify a trend in the types of scientific applications that may benefit from Grid computing. They classified Grid applications in four large groups by comparing the difference in computation of the tasks, the amount of memory and data used and inter-task communications required (see section 3.1 for more details). The first class defined (class I) contains applications that are embarrassing parallel in nature. These applications can be divided into many tasks

where there are no data dependencies between 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 synchronisation 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 synchronised 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.

As previously mentioned, 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 [49]) or Grid middlewares specific for stream processing (e.g. gLite [1]). 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. Typically, tightly synchronised applications are executed in massive parallel systems and they implement data parallelism. However, the opportunity to exploit the natural task parallelism of some mathematical models with the increase in speed of communication links have amplified the possibility for tightly synchronised applications that implement task parallelism to have good performance in a Grid environment.

The vast amount of resources that can be available in a Grid environment, with the possibility to utilise them for accelerating many types of scientific applications, has generated a large amount of research for finding paradigms that permit a user to easily unlock these resources. This situation, and the fact that

Grid computing is still a fairly young research field, has produced many different programming models and paradigms. For example, message passing, shared state model, batch system, workflow system and software toolkit [64]. As in the case of scientific applications for distributed computing, one of the most researched techniques for Grid computing is RPC. This is due to its straightforward programming model, since the main objectives of Grid computing are its ease of use and ease of development. Many projects using the RPC model have been implemented for Grid computing, however in this thesis we focus on GridRPC [75], a standard promoted by the Open Grid Forum [2], that is designed for distributed scientific computing in a Grid environment.

In this chapter, we first introduce the GridRPC project by showing its programming and communication model. Then, we present the GridRPC methods that can be used in order to interface a scientific application to a Grid environment. Finally, we introduce three middlewares that implement the GridRPC model and we present a different programming model that is similar to GridRPC. In the second part of this chapter, we show that the overwhelming majority of applications chosen, or artificially created, to demonstrate the performance of GridRPC middlewares are of class I and II.

## 2.1 GridRPC

GridRPC provides a simple and portable programming interface that permits an easy development of applications which can simply execute tasks remotely in a Grid environment. A number of Grid middleware systems are GridRPC compliant including GridSolve, Ninf-G and DIET.

GridRPC extends the traditional RPC model 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 stub-less client model, meaning that the programmer does not need to specify a stub for each remote task. There-

fore, client programs are not required to be recompiled when tasks are changed or added. This facilitates the creation of interfaces to scientific computing environments (SCEs) such as Matlab and Mathematica.

These fundamental characteristics of GridRPC are possible due to the use of the function handle and section ID objects in the GridRPC API. The function handle associates a task name with the respective task implementation in a specific server. This association is performed automatically by the underlying GridRPC middleware. The session ID represents a particular non-blocking RPC call. These two objects are the only extra information that is used during a remote task call in comparison to a local subroutine call of the applications language. Therefore, the development of a GridRPC application from an existing scientific code is straightforward.

GridRPC standard does not dictate the underlying structure of the middleware, since different GridRPC implementations may use different mechanisms, but indicates only the API and the programming model [74]. This model is designed to hide the complexity of the interaction with a Grid environment from the programmer and consequently to simplify the development of a Grid-enabled application.

### 2.1.1 GridRPC Model

The GridRPC model is based on a client-server architecture and its basic structure is illustrated in figure 2.1. The functionalities presented in this section are shared by all the implementations of the GridRPC model [67].

**Function handle** In order for a task to be available on a server, a programmer has to define the specific information that describes various aspects of the remote task. The information used, and the method to define it, differs in each GridRPC implementation. Typically the various aspects described are:

- 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).

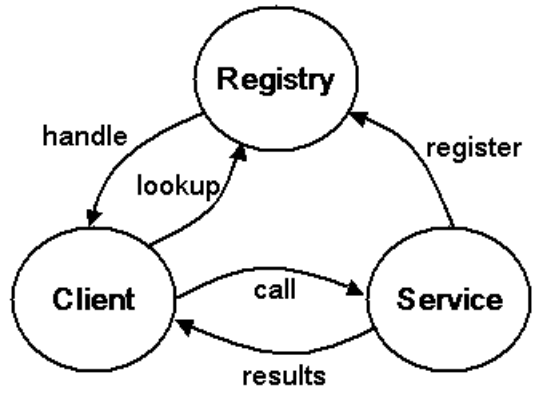



**Figure 2.1:** *The basic GridRPC model*

- Whether the arguments are inputs or outputs.

Common in all the implementations is the use of the function handle to associate a task name to the respective task implementation in a specific server. Through the function handle, the task's information is retrieved by the client and is used by the middleware to execute the task on the server. This technique eliminates the need for client-side stubs for each task in the Grid environment.

**Resource discovery** Each server of the Grid environment registers its tasks available in a *registry*. This involves the servers sending the task's information to the registry. The registry is an abstract term that could indicate a single entity or several entities, which works as a resource discovery. Its mechanism varies between implementations. Different implementations of the registry may store further information, such as data describing the underlying Grid, i.e. the speed of the client-to-server network links and the performance of the servers. These various different types of information can be utilised to generate *performance models* which are used to estimate the possible execution time of a task in the Grid environment.

**Client application run-time** The function handle is retrieved at run time by using the GridRPC method *grpc_function_handle_default*. When this method is invoked without a specific server, the server is dynamically chosen by the re-

source discovery mechanism. Once a particular task-to-server mapping has been established by initialising a task handle, all GridRPC task calls using that function handle will be executed on the server specified in that binding. Each GridRPC task call gets processed individually, where each task is discovered (task look-up) and executed separately from all the other tasks in the application. A GridRPC system, which performs dynamic resource discovery and mapping, can delay the selection of the server until the task is called. 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.

### 2.1.2 GridRPC Communication Model

Another important aspect of GridRPC is its communication model. Since the GridRPC model maps tasks individually, the data dependencies between tasks are not known during run-time. This model forces bridge communication between tasks because the underlying GridRPC middleware does not know if an output data object is an input in an another task. Thus, output objects need to be sent to the client and input objects need to be received from it. Therefore, the communication model of GridRPC is based on the client-server model or star network topology. This means that input/output objects can only traverse the client-server links even though the server machines are connected directly by a network link. Figure 2.2 show the structure of the GridRPC communication model.

### 2.1.3 GridRPC API

The standard API includes a large number of methods, which can be utilised in various ways, from initialising and finalising the middleware to error management. However, the main methods used in an application's algorithm are the remote task call functions and the asynchronous tasks control functions.

The task call functions are divided into two types, blocking and non-blocking. The method *grpc_call* is used to execute a remote task synchronously, i.e. the function call waits until the end of the remote task computation before returning to the caller. While the method *grpc_call_async* executes the task asynchronously,

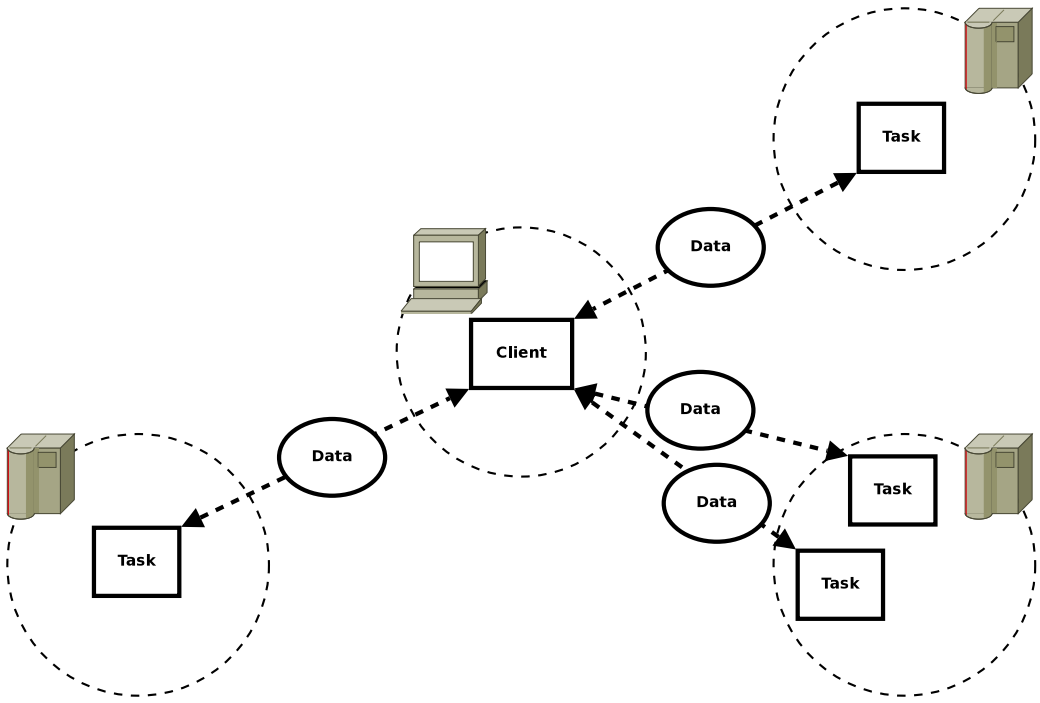



**Figure 2.2:** *GridRPC communication model*

i.e. the function call returns immediately without waiting for the end of the task execution, thus permitting tasks parallelism.

Table 2.1 shows an example application of these functions. The first argument of both methods is the function handle of the task executed. The second argument of the non-blocking call is the session ID of the remote call. The following arguments are the data objects used by the remote task. The session ID (SID) is used by the method *grpc_wait* to block the client computation until the remote task, that generates the SID, is concluded. The function *grpc_wait_all* blocks the execution until any previously issued asynchronous request has completed.

### 2.1.4 GridRPC Middlewares

A GridRPC middleware is responsible for:

- Giving a method to define the task's information.
- Defining performance models of tasks and the network.

**Table 2.1:** *Example of GridRPC methods*

```
1 // Blocking call
2 grpc_call(&handle,obj1,obj2,...);
3
4 // Non-blocking call
5 grpc_call_async(&handle,&sid,obj1,obj2,...);
6
7 // Asynchronous tasks control
8 grpc_wait(sid)
9 grpc_wait_all();
```

- Managing the scheduling of tasks.
- Directing the movement of task data.
- Executing the task on the remote server.

In this section, the three major GridRPC middlewares are introduced.

#### 2.1.4.1 GridSolve

The GridSolve middleware [84, 42] is the evolution of the NetSolve [25] system that implements the GridRPC model. It enables users to solve complex scientific tasks remotely on distributed resources. GridSolve emphasises ease-of-use for the user and includes resource monitoring, mapping and service-level fault tolerance. In addition to providing Fortran and C clients, GridSolve enables SCEs to be used as clients, so domain scientists can use Grid resources from within their preferred environments. The GridSolve system consists of three entities: the client that calls the remote task in its algorithm, the server that executes the tasks requested by the client and the agent that acts as a registry and can perform dynamic discovery. GridSolve, to generate the performance model, uses the LINPACK benchmark and the ping pong benchmark for calculating respectively the servers' performances and client-to-server links speeds. A programmer, that includes tasks in a server, has to provide the description of the task by using the

GridSolve Interface Definition Language (gsIDL). This language allows an easy implementation of existing functions as tasks in a GridSolve server [41].

#### 2.1.4.2 Ninf-G

Ninf-G [79] is a redesign of the Ninf [68] system that fulfils the GridRPC API and programming model. The focal point in the Ninf-G development is simplicity. Thus, Ninf-G does not implement its own protocols but it is designed to be a small RPC layer on top of the Globus toolkit [46]. This permits a high level of interoperability with other Globus-based Grid systems. The Globus toolkit provides different components to Ninf-G, such as GRAM to invoke remote executable and MDS (Monitoring and Discovery Service) for distributed resource discovery. However, Ninf-G does not directly provide fault recovery, load-balancing or scheduling. Client programs that interface with Ninf-G can be written in C, C++, Fortran and Java; while server tasks can be written in C, C++ and Fortran. Furthermore, Ninf-G can execute remote tasks that use MPI. To include functions from existing libraries in a server, a programmer can utilise Ninf-IDL (Interface Description Language). This language, that is similar to the GridSolve one but with a different semantic, is used to describe the interface of a task.

#### 2.1.4.3 DIET

DIET (Distributed Interactive Engineering Toolbox) [23] is a set of tools designed to build and manage a scalable Grid middleware with a highly hierarchical structure that is accessible through native and GridRPC API. The resource discovery and scheduler in DIET are scattered across a hierarchy of Local Agents and Master Agents. The motivation for this architecture is that it is more scalable and solves the problem of bottlenecks in a centralised agent/scheduler when many clients try to access several servers. DIET uses Corba as a communication layer. The user can use different types of client interfaces to access DIET middleware: web portals, SCEs and C or C++ applications. DIET is a GridRPC system that can perform dynamic mapping of tasks, i.e. delay the selection of the server during the remote task call. Furthermore, DIET dynamic mapping also involves discovery of performance models, which are used by the mapping heuristics. The perfor-

mance models for DIET are the FAST prediction tool, Network Weather Service (NWS) [23] and CoRi [7]. In DIET, the information needed by a task to be executed in a server is not added by the programmer with a specific language as for GridSolve and Ninf-G. Instead, the task information is retrieved directly from the task code thanks to specific methods included in it. This approach has the disadvantage that the task code has to be heavily modified by the programmer [38].

### 2.1.5 Alternative RPC model for Grid: OmniRPC

As previously mentioned, GridRPC is not the only programming model developed to implement the remote procedure call paradigm for Grid computing. An important alternative model is the OmniRPC [72] programming model. OmniRPC is designed to allow easy development and implementation of parallel scientific applications for distributed and Grid environments.

OmniRPC is an evolution of Ninf, since it inherits the API and basic structure from it, and thus the OmniRPC programming model is very similar to the GridRPC one. The central difference is that OmniRPC is mainly designed for multi-threaded clients that have a master-worker structure. This design is achieved by implementing a remote procedure call system that is thread-safe.

The architecture of OmniRPC is similar to the one of GridRPC. It is composed of a client application and various remote computational hosts, which execute the remote procedures. Remote locations can be connected via a local area network or over a wide-area network. The client application can be written in various different languages, such as FORTRAN, C and C++, and the parallel execution in the client can be obtained by using direct thread libraries, such as the POSIX thread, or the OpenMP API. The interface to a remote function is described by the Ninf IDL. In OmniRPC, the remote executions are managed by the use of remote shell (rsh) for local distributed environments and by the use of Globus and ssh for Grid environments. This techinique fulfills another objective of OmniRPC; that is the possibility to seamless switch from a distributed environment to a Grid one.

## 2.2 GridRPC Performance Analysis: Related Work

The GridRPC model is designed to permit a typical scientific application to gain many non-performance related benefits, such as ease of development and control of the application. However, an important goal of the GridRPC model is to achieve high performance in execution of various types of scientific applications. Thus, an investigation into the performance of the execution model of GridRPC is necessary to fulfil its design. Furthermore, as indicated in section 2.1.4, the various GridRPC middlewares have different structures and implementations despite having a standard model in common. Therefore, it is difficult to understand which GridRPC middleware has better performance for a wide range of applications or which particular implementation component is a performance bottleneck for a specific type of applications.

A popular approach is to utilise a singular or a set of different programs to experimentally analyse the performance of a system. This allows the comparison of different designs and implementations of a particular execution model. As it has developed in the High Performance Computing (HPC) field, the existing tools for performance analysis in Grid computing appear to be in two categories:

1. Low level *probes*, which are used to measure the performance of single component of a system. For example, the one proposed by Chun *et al.* [29].

2. *Benchmark applications*, which are used to evaluate the overall performance of a tested system and its execution model such as the GridNPB (NAS Grid Benchmarks) [50] suite.

Since benchmark applications permit to expose the eventual limits and benefits of Grid middlewares, they are often used to show the performance of new Grid implementations. This is the case for GridRPC systems as well. Unfortunately, applications of class I and II, which are best suited to run on a Grid environment, are typically chosen to analyse and present the performance of a GridRPC middleware system. This induces the situation where it is difficult to differentiate the performance of the various GridRPC implementations or to expose problems in the execution model. Therefore, this nullifies the benefits that the experimental investigation of the performance brings.

Another different approach in selecting a good application for analysing the performance of a Grid system is the possibility to choose an artificially created application or a real-life one. The former is designed from the ground up to identify the bottleneck of a system and to anticipate the requirements of the applications that will run on the system. These modelled applications have the advantage that they are easy to manage, such as choosing the quantity of data computed or the numbers of tasks executed. However, there is always the risk, with artificial applications, that the internal structure, tasks, and algorithm chosen are not representative of the future real-life applications executed on the tested system. It is challenging for the tester to predict all the characteristics of an application and their impact on performance. There is always the chance of a misunderstanding that may compromise the results obtained. The second possibility is to utilise a real-life application as a performance analysis tool with the same characteristics of the future applications to be executed in the system. This has the advantage that the application represents exactly the kind of problem that the system tested has to face at production time. However, real-life applications have some disadvantages as well. They may not be flexible enough, perhaps only allowing particular input data sizes or having a limited number of parallel tasks available. Furthermore, implementing a real-life application with a particular set of API can be challenging.

An additional distinction to make when choosing an application for performance analysis is about task parallelism and data parallelism executions. As previously mentioned, a task parallel application divides the computation between different tasks. Each task executes the same computation or different computations on the same or different data. A data parallel application divides the data between different processes that execute the same computation. The dissimilarities between the two modes are subtle. However the main difference is that task parallelism is usually coarse-grained while data parallelism is usually fine-grained. Therefore, tightly synchronised data parallel applications have a high amount of data communication for each computation block and they are typically executed on a system with very fast network links, such as massive parallel systems. Thus, they are developed using message passing or shared memory paradigms, such as MPI or OpenMP, and not using remote procedure call methods. Consequently, data parallel applications are not representative of the types of applications devel-

oped for RPC.

In the following sections we examine the various applications, frameworks and algorithms used to evaluate the performance of a generic Grid system and of different GridRPC middlewares.

### 2.2.1 NAS Grid Benchmarks

GridNPB suite is a pencil-and-paper specification, i.e. it provides only reference code not real implementation, which aims to give to the developers and users a uniform tool to analyse the performance of a Grid systems [50]. It is based on the NAS Parallel Benchmarks (NPB) suite [13] that is largely used to analyse the performance of HPC systems. GridNPB defines four families of problems that represent workloads of Grid applications. Each problem is defined by a workflow graph and each node in the graph is an instance of an NPB code. The four GridNPB problems are: Embarrassingly Distributed (ED), Helical Chain (HC), Visualisation Pipeline (VP), and Mixed Bag (MB). The ED problem is related to class I of Grid applications, while HC and VP are related to class II. The MB problem can be affiliated to class III applications. At the moment of writing, a GridRPC version of NAS Grid Benchmarks has not yet been implemented.

The Mixed Bag problem is the more interesting since it represents a model of a tightly synchronised application. Figure 2.3 shows the workflow graph of the MB problem, where boxes represent the tasks executed and arrows the data communications between tasks. The Mixed Bag tasks perform operations that symbolise typical scientific computations, such as Fourier transform (FT), Multigrid (MG) and LU solver. The computational granularity of a task can be controlled by varying the number of iterations using the task's parameters.

NAS Grid Benchmarks, and specifically Mixed Bag, can be a good tool for performance analysis since it aims to provide a standard set of tasks and problems that can be used as a meaningful comparison. However, the issue of GridNPB is that the four problems are artificial. For example, the Mixed Bag problem is designed from the ground up to represent a highly synchronised application with asymmetrical data communication. Therefore, despite the use of tasks that are realistic, the workflow itself is only a representation of what might be a real-life

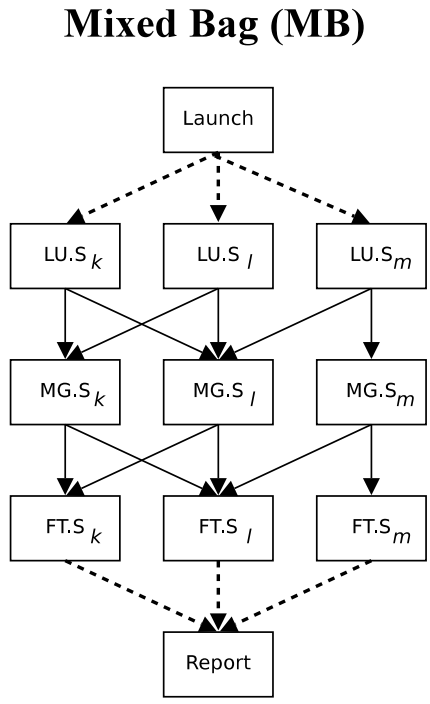



**Figure 2.3:** *Workflow graph of the NAS Grid Benchmarks Mixed Bag problem.*

application. Consequently, using this benchmark on a Grid system may not highlight performance problems that could arise from production time applications. Furthermore, the possibility to change the computation granularity of the tasks could result in the tester involuntarily setting a favourable amount of workload and consequently invalidating the results.

### 2.2.2 Evaluation Applications used in GridSolve

As previously mentioned (section 2.1.4.1), GridSolve is an evolution of NetSolve, which started in 1996. Therefore, during its evolution many new components or models were developed to improve its performance (see section 5.6). Naturally, different applications were used to analyse the performance of the various implementations of the components and GridSolve itself.

MCell is a real-life application used in the neuro-science field. It uses a Monte Carlo method as the main algorithm. Casanova *et al.* [26] utilise this application to analyse the performance of a scheduling algorithm and the task farming compo-

nent in NetSolve. A Monte Carlo method consists of many identical, independent computations of different values generated randomly. Thus, MCell is composed of large numbers of parallel tasks and can be considered a class I application.

Arnold *et al.* [11] utilise two sensing/image processing domain applications to benchmark the request sequencing component in NetSolve. The first application performs the Principle Component Analysis (PCA) transformation to an image. This application is composed of data intensive sequential tasks and can be categorised as a class II. The second application performs several classification steps to an image and the results of these steps are combined by a fusion module. This application is of class II as well because is composed of two data intensive parallel tasks with minimal synchronisations.

Desprez and Jeannot [39] analyse the performance of their implementation of a data persistence component in NetSolve by using two applications. The first one consists of three matrix multiplication tasks where the first two tasks are executed in parallel. This application can be classified as a class II. The second application performs a singular matrix multiplication using a parallel block algorithm. Thus, this application is composed of fine-grained parallel tasks with a high level of data synchronisation and can be considered class III. Parallel block matrix multiplication applications can be treated as typical examples of data parallel applications. This type of fine-grained data parallelism application is rarely computed on a Grid environment instead of a parallel supercomputer due to the high amount of data communication in relation to the computation. Furthermore, how this application's data is usually divided to be computed in parallel and the relative data communication are specifically well-suited to being developed with a message passing paradigm rather than a remote procedure call programming model.

To illustrate the use of Internet Backplane Protocol (IPB) in NetSolve, Beck *et al.* [14] implement an application that performs two sequential tasks. The first one executes a matrix multiplication, the second task reuse the output of the first task to solve a complex system of equations. This can be categorised as an artificial application of class II.

YarKhan *et al.* [84] create two artificial applications to benchmark the load balancing and the request sequencing components of a newly implemented Grid-

Solve. The first one is composed of 16 parallel isolated tasks of matrix multiplication while the second application uses three matrix multiplication sequential tasks. These two applications can be roughly classified as class I and II respectively.

To test the integration between GridSolve and gLite [1], Hardt *et al.* [56] use an artificial application composed of many parallel tasks, each one made of a CPU intensive loop. These tasks do not communicate between them, thus this benchmark application can be considered class I.

Yinan *et al.* [65] analyse the performance of the GridSolve request sequencing component with two applications. The first application calculates a matrix multiplication using the Strassen's Algorithm. This application is composed of different layers of fine-grained parallel tasks with a high level of data synchronisation between tasks of different layers. Therefore, this benchmark can be classified as a data parallel class III application since is similar to the parallel block matrix multiplication previously discussed.

The second application is taken from an image processing toolkit. The workflow of the original real-life application consists of a series of data intensive sequential tasks. This workflow was modified to have a variable number of independent parallel tasks before the sequential tasks. Therefore, this application can be considered class I in the modified part and class II in the rest of the workflow.

It is important to notice that Yinan *et al.* [65] using the data parallel class III application, the Strassen's Algorithm, detect that the performances obtained in the experiments using more than one computer are not satisfactory in comparison to the executions with a single computer. They emphasise that the improvement generated by the parallel execution and the new request sequencing component is offset by the overhead of the data communication caused by the fine-grained highly synchronous parallel tasks. While this conclusion is highlighted as well by the experiment results of this thesis (see section 4), we consider that the use of a tightly synchronised data parallel application as a tool to analyse the performance of GridRPC is out of scope since this type of application is mainly developed using the message passing programming paradigm on massive parallel systems. Therefore, an analysis of the performance potential and limits of GridSolve with task parallel tightly synchronised scientific applications is missing.

### 2.2.3 Evaluation Applications used in Ninf-G

Ninf-G is an evolution of the Ninf middleware that started around the same time as the NetSolve project. In the initial articles produced to present Ninf, three applications were used to analyse the performance of the various components of the new middleware, such as overhead of remote executions, overhead of multiple simultaneous client executions and load balancing techniques. The three applications are: an artificial application composed of two sequential remote calls of Linpack benchmark [73], the embarrassingly parallel problem of NAS Parallel Benchmark [77] and the density of states calculation of a large molecule [69] that can be divided in many parallel individual computations. The first application can be roughly considered as a class II application, while the other two applications belong to the class I group.

Tanaka *et al.* [79] present the newly implemented Ninf-G middleware and its performance by using an embarrassingly parallel class I Monte Carlo application. Furthermore, a weather forecasting application is used as well to analyse the performance of Ninf-G. While the internal details of this application are not given, the authors indicate that the client executes altogether many simulations on various servers. Therefore, it can be considered as a class I application.

A phylogenetic tree contains the information of evolutionary relationships between various biological species. It is used to determine how different species have evolved and are related to each other. Yamamoto *et al.* [83] introduce an application that parallelises the searches on a phylogenetic tree by splitting the main tree into sub-trees. A part of the various subtree computations is parallelised using Ninf. These computations are independent of each other, therefore the application can be considered as a class I.

A second version of Ninf-G is presented by Tanaka *et al.* [80] and to evaluate the performance of this new version of the middleware a weather forecasting application is used. This application is designed to predict short to middle term global weather changes and the main simulation routine is implemented as a remote task. The client code performs many parallel independent remote simulations, therefore this application is embarrassingly parallel and of class I.

Osawa *et al.* [71] present a technique to speed-up the finding of an optimal

batting order in a baseball team through the use of a Markov chain method implemented in Ninf-G. This is a class I application, since the calculation for each batting order is independent.

Takemiya *et al.* [78] research the possibility to execute a multi-scale simulation of a semiconductor processing using a Grid environment composed of systems scattered across the pacific. Furthermore, the authors test the feasibility and performance of using a hybrid GridRPC/MPI application with Ninf-G in such an unstable Grid system. The application is composed of a molecular dynamic (MD) simulation that performs many quantum mechanical (QD) simulations that are computational intensive . The MD simulation is implemented in MPI code and it acts as a GridRPC client where the various QD simulations are executed remotely in different cluster systems using the Ninf-G middleware. The QD task is implemented in MPI as well and the various QD simulations are independent each other. The authors take advantage of the natural embarrassingly task parallelism of the application to analyse the performance of various components of the project. Therefore, this application belongs to the class I group.

One interesting application that is used to analyse the performance of Ninf and Ninf-G is an application that implements a parallel branch and bound algorithm to solve the BMI eigenvalue problem. This application appears in various works with slightly different internal structure and implementation. However, the method to "gridify" the algorithm is the same in all the various guises [4]. The branch and bound algorithm repeats the following procedures until the gap between the lower and upper bounds of the solution space is less than a threshold; decomposing a master problem into two sub-problems (branching operation), computing the lower and upper bounds, and the solution for each sub-problem. The sub-problem is then pruned if its lower bound is greater than the temporary upper bound (bounding operation), and a sub-problem with the lowest lower bound is selected as the next master problem to restart the cycle. This application uses a Master-Worker paradigm, where a master (GridRPC client) dispatches sub-problems to multiple workers (GridRPC servers), and the worker performs computations on the sub-problem (GridRPC tasks). The computation granularity of a task can be changed by part of the branching being performed in the task itself instead of the client. However, the task has a really fine granularity, in fact

the computation can take less than a second on modern hardware.

The algorithm generates a search tree where the root is the initial problem and the nodes represent sub-problems. The amount of data communicated between tasks in comparison to the computation is minimal, but there is a good level of synchronisation between tasks of different levels in the search tree. However, the computations of tasks in the same level are independent of each other and after a few iterations there are thousands of parallel tasks computed simultaneously in a level of the search tree. Therefore, this application is embarrassingly parallel and of class I. The characteristic of this algorithm, which generates thousands of independent parallel and really fine-grained tasks, lets Aida and Osumi [5] consider this application as a good performance evaluation tool for Ninf/Ninf-G middlewares. In this work, the authors realise that the computation of the various tasks is too fine for the middleware overhead. Therefore, they implement a hierarchy structure in the application to be able to enlarge sufficiently the task granularity. Despite this application being of class I, it is a good tool to evaluate the performance of a Grid system since it shows the limits of the middleware. In this particular case, it is the fine granularity of the tasks. However, also in the case of Ninf-G, a performance evaluation of the middleware and the GridRPC model limits and benefits of class III applications has not been done.

### 2.2.4 Evaluation Applications used in DIET

As previously mentioned in section 2.1.4.3, one important characteristic of DIET is its hierarchical structure of agents, where they can be local or master agents that perform the resource discovery and scheduler functionalities. Therefore, the existing literature of the DIET project is notably focused on analysing and testing various algorithms for this hierarchical structure and thus different applications were used as performance evaluation tools to analyse these algorithms. Furthermore, as it was in the case of GridSolve, during the existence of DIET various functionalities were included to improve the performance of the middleware (see section 5.6), and therefore different applications were used to evaluate the performance of these new functionalities.

Dail and Desprez [32] introduce an extension of the DIET scheduler that in-

creases its ability to support a high flow of requests through various levels of the agents hierarchical structure. Two usage scenarios are presented to test this scheduler extension, a sequential user model and a batch user model. The application used in these two scenarios is composed of a single task or ten sequential tasks of a matrix-matrix operation. While the application itself is class II, it is executed multiple times in parallel for the batch user scenario. Therefore, it can be considered as an embarrassingly parallel experiment.

The problem of optimally deploying a scheduler for hierarchically organised systems is analysed by Chouhan *et al.* [27] using DIET as a testbed. The application used to evaluate the deployment model analysed is composed of a single matrix multiplication task call and therefore is class II.

Jannot and Monard [60] utilise DIET to execute an application that calculates the potential energy of a chemical system (a molecule or a set of interacting molecules). Since this potential energy is calculated for each atoms' conformation in the system and this calculation is independent between different conformations, this computation is highly parallel and thus the application is class I.

In order to minimise the amount of data communicated during a remote task execution, DIET introduces a new functionality, DTM (Data Tree Manager), that permits the application programmer to directly specify data persistence between tasks and between different executions of an application. This new functionality is tested with various different applications during its development. At first, Caron *et al.* [22] uses an application composed of three sequential matrix-matrix operations to analyse the performance of DTM. Therefore, this application is class II. Then, a more detailed analysis of different DTM attributes is made by Del Fabbro *et al.* [34] using various applications: (1) an application composed of 10 synchronously remote tasks, which is executed simultaneously many times to test the overhead and scalability of DTM; (2) a linear algebra application composed of a single remote task, where the computation time is independent of the data size, to evaluate the benefits of data persistence between various executions; (3) an application with various synchronous calls of a task, which computes the number of occurrences of a letter in a file, to analyse the advantage of data replication; and finally (4) to test DTM under realistic conditions, an application that extracts and displays an isosurface from a 3D medical image, which is composed of three

sequential remote tasks. All these applications are class II.

Ramses is an astrophysical application that simulates the evolution of the dark matter in the universe using an N-Body algorithm. Caniou *et al.* [20] present a Grid version of Ramses that uses DIET. This application executes a "zoom simulations" technique that is composed of two steps: a fast simulation of the universe with low accuracy to identify dark-matter halo regions and re-simulations of these regions using a high amount of particles for more accuracy. Therefore, the flow of tasks execution is made of a singular remote task, which executes the initial low resolution simulation, followed by many parallel tasks that compute the re-simulations. Since these parallel tasks are independent each other, Ramses is a class I application.

JUXMEM is a software platform that permits a Grid application to share data between different servers in a Grid environment. These servers can be divided in various communication groups that are part of a hierarchy. Antoniu *et al.* [9] introduce a performance analysis of the JUXMEM integration in the DIET middleware. This analysis is done by using MUMPS, a sparse parallel solver, as a remote task and by using an application that is composed of 32 sequential GridRPC calls to this task. Therefore, this application is class II.

This analysis shows that in the case of DIET, all the applications used to evaluate the middleware performance are either class I or II.

## 2.3 Summary

In this chapter, we have presented the GridRPC programming model, its communication model and its API. Furthermore, we have introduced the various middlewares that implement the GridRPC model. We have also analysed all the applications used to evaluate the performance of GridRPC middlewares. This analysis shows that typically the applications used belong mainly in the class I or II groups. These types of applications are extremely suitable for execution in a Grid environment, therefore they do not emphasise the eventual performance problems or advantages of a Grid system. We believe that to justify the use of GridRPC for a wide range of applications, we should not use an extremely suitable application to demonstrate the performance potential of GridRPC systems but a real life ap-

plication of class III that shows the eventual limits and benefits of the GridRPC middlewares tested.

The idea of using tightly synchronised Grid applications to research the performance of a GridRPC programming system is a valid one since this approach can be seen also in NAS Grid Benchmarks. This suite uses four different types of problems as performance analysis tools; these problems are not only representative of class I and II applications but also of class III, as in the case of Mixed Bag workflow. In fact, when a class III application is used as a benchmark for GridRPC system, like Yinan *et al.* [65] with the Strassen's Algorithm, the eventual limits of the studied component are recognised. Therefore, it is possible to understand which component of a middleware or part of an execution model represents the bottleneck in term of performance. Unfortunately, the few class III applications used as performance analysis tools in GridRPC belong to the data parallelism field, i.e. applications of which the fine-grained highly synchronised parallel computation is more suitable to be handled using message passing paradigms like MPI. Since the RPC programming model, and thus GridRPC, is mainly used for task parallel applications, the use of data parallel applications for performance analysis of GridRPC is not representative.

Another problem of the existing applications used to analyse the performance of GridRPC systems is that they are created artificially for this purpose. The use of artificially designed applications is good to stress test one or many components of a GridRPC system. However, they are not good to research the overall performance since there is the chance that the engineered algorithm, task computations and data communications are not representative of the future real applications executed on a system. Therefore, real life scientific applications used as performance analysis tools allow a scientific user to research, not only the performance of particular components, but also the limits and benefits that a particular class of application will get when executed using the specifically programming model and middleware tested.

# Chapter 3

# A Scientific Application as a Tool for Analysing the Performance of GridRPC Systems

As it was shown in section 2.2, the overwhelming majority of applications used to demonstrate the performance potential of GridRPC middlewares are either embarrassing parallel (class I) or pipelined (class II). Furthermore, these applications are usually artificially created. Therefore, a performance analysis of GridRPC programming systems with real life tightly synchronised (class III) applications is missing.

In this work, we designed and implemented Grid-enabled Hydropad [55], a real-life task parallel tightly synchronised application, which we propose to use as a method to evaluate the performance of GridRPC systems. Hydropad as a benchmark application shows the eventual limits and benefits of the GridRPC middleware systems tested and it justifies the use of a GridRPC system for a wide range of applications. Grid-enabled Hydropad can be used as a stand alone performance analysis tool or can be used in association with other different applications in a benchmark framework. Hydropad is an astrophysical application that simulates the evolution of clusters of galaxies in the universe. This application is composed of a natural task parallelism and its tasks have a balanced ratio between computation and communication and a high level of data synchronisation between them.

Hydropad requires high processing resources because it has to simulate an area comparable to the dimension of the universe and simultaneously try to achieve a high enough resolution to show how the stars developed.

In the first section of this chapter, we present in more detail the classification of Grid applications introduced by Snavely *et al.* [76]. In the following sections, we introduce Hydropad, its internal structure and the type of computations of its task. Then, we show how Grid-enabled Hydropad was implemented using GridRPC. In the last section of the chapter, we introduce the non-performance related benefits of using GridRPC in Hydropad, which can then be applied to any other scientific application.

## 3.1 Classification of Grid Applications

The four classes of scientific application that may benefit from grid Computing identified by Snavely *et al.* are:

**I** *Loosely Coupled.* Applications that are embarrassingly parallel in nature, thus it is very easy to separate the computation in parallel tasks. Each task is composed of a computational intensive algorithm with low memory requirement that does not need to exchange information with other tasks. Therefore, there is not data dependencies between tasks. As previously mentioned, this type of application are ideally suited to achieve high performance in Grid computing. Some examples can be found in the image processing, bioinformatics, molecular biology and cryptography fields.

**II** *Pipelined.* Applications that have to compute a continuous stream of data, which can be generated in real-time by data acquisition devices. The tasks that composed this class of application are often very memory and data intensive and there is a highly level of parallelism between them. However, in comparison to class I applications, data communication exists between tasks. This type of application highly benefits from Grid computing as well because the data steams processed may be generated in different places geographically separated. Example applications can be real-time signal processing, remote sensors such as satellite, microscope etc. Another important

examples of this class are the particles physics Grid applications that compute the stream of data generated by the particle generators in CERN.

**III** *Tightly Synchronised*. Applications in this class have tasks with a high level of data synchronisation between them. Thus, data dependencies and inter-task communication have an important impact on the performance and possible task parallelism of the application. The ratio between task computation and data communication depends on the application. Therefore, the communication infrastructure and its model can be the performance bottleneck of these applications. Typically, these types of applications are execute in HPC systems and they exploit data parallelism. The possibility to use these applications in a Grid environment depends on their computation-communication ratio. Hoverer, the opportunity to exploit natural task parallelism, other than data parallelism, can increase an application's chance to have good performance in a Grid environment. Applications that adopt iterative method as main algorithm are examples of this class of applications. They are used in climate, astrophysics, aerodynamic and molecular simulations.

**IV** *Widely Distributed*. 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. However, they work on large data-set (databases, files, etc.) that can be everywhere on the Grid. These types of applications are especially used in commercial environments and they can be considered as part of cloud computing and "software as service" applications.

## 3.2 Hydropad: a Simulator of Galaxies' Evolution

Hydropad is a cosmological application, originally written by Claudio Gheller, which simulates the evolution of clusters of galaxies in the universe [51]. The cosmological model that this application is based on has the assumption that the universe is composed of two different kinds of matter. The first is baryonic matter, which is directly observed and forms all bright objects. The second is dark matter,

which is theorised to account for most of the gravitational mass in the Universe. The evolution of this system can only be described by treating both components at the same time, looking at all of their internal processes, while their mutual interaction is regulated by a gravitational component. Figure 3.1 shows an example of a typical output generated by Hydropad.

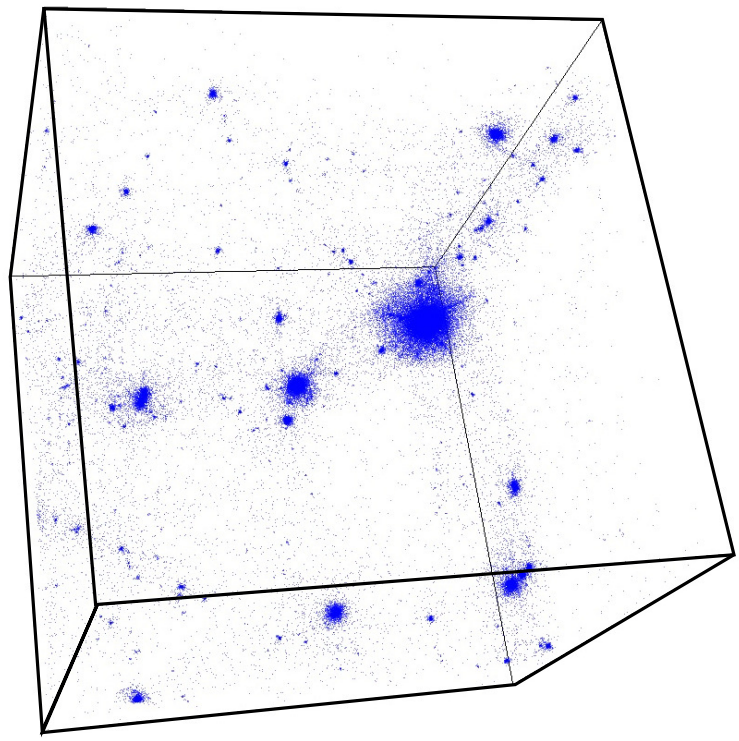


**Figure 3.1:** *Example of Hydropad Output*

The dark matter computation can be simulated using *N-Body* methods [59]. These methods utilise the interactions between a large number, $N_p$, of collisionless particles. These particles, subjected to gravitational forces, can simulate the process of the formation of galaxies. The accuracy of this simulation depends on the quantity of particles used. Hydropad utilises a *Particle-Mesh* (PM) N-Body algorithm, which has a linear computational cost and depends on the number of particles $O(N_p)$. In the first part this method transforms the particles, through an interpolation, into a grid of density values. Afterwards the gravitational potential is calculated from this density grid. In the last part the particles are moved

depending on the gravitational forces of the cell where they were located.

The baryonic matter computation uses a *Piecewise-Parabolic-Method* (PPM) hydrodynamic algorithm [31]. This is a higher order method for solving partial differential equations. PPM reproduces the formation of pressure forces and the heating and cooling processes generated by the baryonic component during the formation of galaxies. For each time step of the evolution, the fluid quantities of the baryonic matter are estimated over the cells of the grid by using the gravitational potential. The density of this matter is then retrieved and used to calculate the gravitational forces for the next time step. The accuracy of this method depends on the number of cells of the grid used, $N_g$, and its computational cost is linear $O(N_g)$. The application computes the gravitational forces, needed in the two previous algorithms, by using the *Fast-Fourier-Transform* (FFT) method to solve the Poisson equation. This method has a computational cost of $O(N_g \log N_g)$. All the data, used by the different components in Hydropad, are stored and manipulated in three-dimensional grid-like structures. In the application, the uniformity of these base structures permits easy interaction between the different methods.

Figure 3.2 shows the work-flow of the Hydropad application. It is composed of two parts: the initialisation of the data and the main computation. The main computation of the application consists of a number of iterations that simulate the discrete time steps used to represent the evolution of the universe from the Big Bang to present time. This part consists of three tasks: the gravitational task (FFT method), the dark matter task (PM method) and the baryonic matter task (PPM method). For every time step in the evolution of the universe, the gravitational task generates the gravitational field using the density of the two matters calculated in the previous time step. Hence the dark and baryonic tasks use the newly produced gravitational forces to calculate the movement of the matter that happens during this time step. Then the new density is generated and the lapse of time in the next time step is calculated from it. It is possible to see in figure 3.2 that the dark matter task and baryonic matter task are independent of each other.

The initialisation part is also divided into two independent tasks. The main characteristic of dark matter initialisation is that the output data is generated by the external application *grafic*, a module of the package COSMICS [15]. *Grafic*, given the initial parameters as an input, generates the position and velocity of the

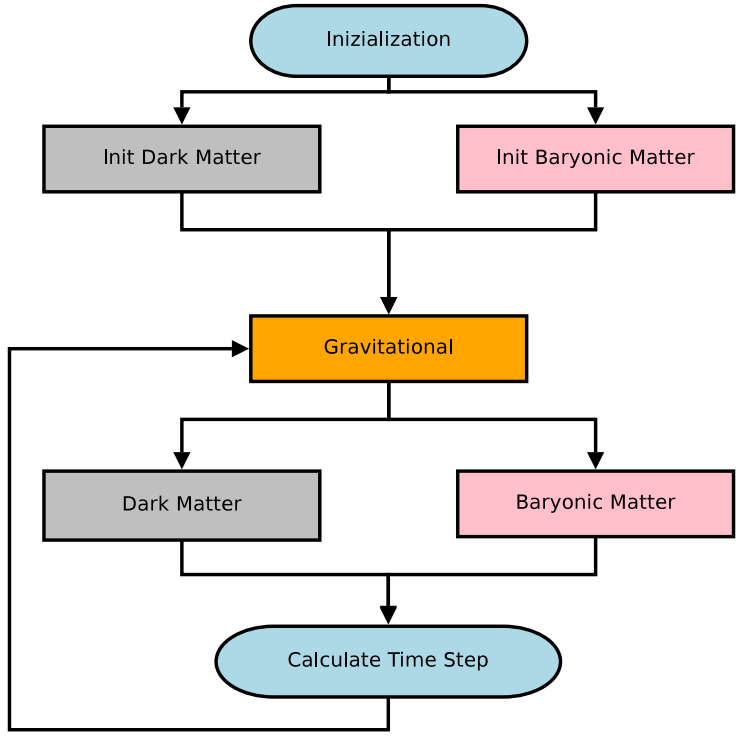



**Figure 3.2:** *Internal structure of Hydropad*

particles that will be used in the N-Body method. The output data is stored in two files, the information within this has to be read by the application during the initialisation part. Like the main application, *grafic* has a high memory footprint.

An important characteristic of Hydropad is the difference in computational and memory load of its tasks. Despite both algorithms being linear, the computational load of the baryonic matter component is far greater than the dark matter one, $C_{bm} \gg C_{dm}$, 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 greater than the baryonic matter one, $D_{dm} \gg D_{bm}$.

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

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

## 3.3 GridRPC Implementation of Hydropad

Hydropad was originally a sequential *Fortran* code, we upgraded this program to take advantage of the GridRPC API and to work with the GridSolve middleware. Table 3.1 shows the original Hydropad code of the main loop, written in the C language. Three functions, *grav*, *dark*, and *bary*, are called in this loop to perform the three main tasks of the application. In addition, at the first iteration of this loop, a special task, *initvel* is called to initialise the velocities of the particles. The dark and baryonic tasks compute the general velocities of the respective matter. At each iteration, these velocities are used by a local function, *timestep*, to calculate the next time step of the simulation. The simulation continues until this time becomes equal to the present time of the universe, $t_{sim} = t_{univ}$.

**Table 3.1:** *Hydropad universe evolution loop*

```
1  t_sim=0;
2  while(t_sim<t_univ) {
3
4     grav(phi,phiold,rhoddm,rhobm,...);
5
6     if(t_sim==0){ initvel(phi,...); }
7
8     dark(xdm,vdm,...,veldm);
9     bary(nes,phi,...,velbm);
10
11    timestep(veldm,velbm,...,t_step);
12    t_sim+=t_step;
13 }
```

The GridRPC implementation of Hydropad application utilises the functions *grpc_call* and *grpc_call_async* to execute respectively a blocking and an asynchronous remote call of the Fortran routines. The first argument of both methods is the function handle of the task executed (section 2.1.1), the second is the session ID of the remote call while the following arguments are the parameters of the task (section 2.1.3). Furthermore, the code uses the method *grpc_wait* to block the execution until the chosen, previously issued asynchronous request has completed. When the program runs, the underlying GridRPC middleware maps each *grpc_call* and *grpc_call_async* functions singularly to a remote server. Then, the middleware communicates the data from the client computer to the chosen server and then executes the task remotely. At the end of the task execution, the data is communicated back to the client. In the blocking call method, the client cannot continue the execution until the task is finished and all the outputs have been returned. Instead, in the asynchronous method, the client does not wait for the task to finish and proceeds immediately to execute the next code. The output of the remote task is retrieved when the respective wait call function is executed.

**Table 3.2:** *Hydropad implementation using GridRPC API*

```
t_sim=0;
while(t_sim<t_total) {

  grpc_call(grav_hndl,phiold,...);

  if(t_sim==0){ grpc_call(initvel_hndl,phi,...); }

  grpc_call_async(dark_hndl,&sid_dark,x1,...);
  grpc_call_async(bary_hndl,&sid_bary,nes,...);

  grpc_wait(sid_dark); /*wait for non blocking*/
  grpc_wait(sid_bary); /*calls to finish*/

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

Table 3.2 outlines the GridRPC implementation of the main loop of Hydropad that simulates the evolution of 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.

## 3.4 Non Performance Related Benefits of GridRPC

The use of GridRPC for scientific applications does not only bring performance related advantages. The main idea that led to the conception of GridRPC was the need to have a programming model that simplifies the development of Grid applications, in order to obtain a widespread adoption of Grid computing. Therefore, in this section we analyse the non-performance related benefits that we experienced from using GridRPC in the Grid-enable Hydropad project which can be applied to any other scientific applications.

**An easy and powerful development paradigm** As shown in table 3.1, the Hydropad code was already logically divided into tasks by the original author. He used different procedures for each component of the cosmological model since each specific component solves a different mathematical model and thus algorithm. To "gridify" this application, we had only to identify these logical divisions and the relative code and then create for each procedure the gsIDL stub file; thus enabling the remote execution of the tasks. Then, thanks to the simplicity of the RPC programming model, we easily included the remote task execution in the main loop of the application using the *grpc_call* method in the place of the original procedures.

**The possibility to simply reuse code and algorithms** One of the few problems that we came across during the development of GridRPC implementation of

Hydropad was to identify all the global variables used in the various Hydropad procedures. A numerical method, to be executed remotely, has to avoid internal state changes, like a function with isolated computation and no global variable. This method of development creates tasks that have a specific interface for input/output values. Therefore, the GridRPC tasks and their respective algorithms can be easily reused in other Grid applications because their execution with the same input always produces the same output. This situation can reduce the programmer's effort in developing a Grid application. For example, the programmer can use already existing tasks that he/she would not have the time or skill to write.

**Exploiting the natural task parallelism of scientific problems** The idea of solving a scientific problem using different complex mathematical components and thus algorithms is a desired one. Unfortunately, multiple powerful systems able to compute various computational intensive algorithm of desired size have never been easy to find. Therefore, the opportunity created by Grid computing to gather multiple powerful systems together in addition to the one created by the ease of development and code reuse of GridRPC programming model is an unique one. It permits a scientific user to solve easily and quickly a composed and complex scientific problem by exploiting its possible task parallelism. For example, in the GridRPC implementation of Hydropad, the baryonic matter task with its PPM method can be computed remotely in a parallel cluster using the MPI paradigm while simultaneously the dark matter task with its PM N-Body method can be solved in a different cluster using the openMP paradigm.

**Portability** Since Grid-enabled Hydropad comprises of a client application and server-side compiled executables, the client application can be easily ported, compiled and executed on a new machine. This does not require the recompilation of server-side task executables, which could make up a large proportion of the application.

**More control over the application** Hydropad potentially can be executed not only in a Grid environment but also in a high performance computer (HPC) system. Unfortunately, in a HPC system, where applications are executed in batch

mode, the user will not have much control over the execution. Grid-enabled Hydropad allows the user to have a high control over its execution because, although the tasks are being computed in remote servers, the main component of the application is running on the client machine. This can be important for many types of applications, some examples are:

- Applications that need a direct interaction with the data produced. For example, the user could visualise directly in the client machine the evolution of the universe, while Hydropad is running on the Grid. Furthermore while the user is checking the simulation evolution, he or she could decide on the fly to change some parameters of the simulation or restart the application. This is possible since in Grid-enabled Hydropad the main data and the main execution are on the client machine.

- Applications that have a task that is inherently remote. For example in the case of Hydropad, if *grafic* cannot be executed on the client machine because it needs a specific hardware, the user has to generate the initial data on the remote server and then manually retrieve it. The use of GridRPC can simplify this situation by allowing a special task to interface with *grafic* directly on the remote server. This task can communicate immediately the initial data generated by *grafic* to the application.

## 3.5 Summary

In this chapter, we have introduced Hydropad, a real-life astrophysical application, that we propose should be used as a tool for experimental performance evaluation of GridRPC systems. This application is composed of tasks that have a balanced ratio between computation and communication with a high level of data synchronisation between them. Therefore, this application can be classified as a class III. Hydropad requires high processing resources because it has to simulate an area comparable to the dimensions of the universe and simultaneously try to achieve a high enough resolution to show how the stars developed. Despite the fact that these types of tasks are not the most suitable to be executed on a Grid because of the high magnitude of communication involved, we have shown that

Hydropad obtains many benefits from being Grid-enabled. These benefits are related to performance gains and also the management and development aspects of the application.

# Chapter 4

# Experimental Analysis of Performance Potential and Limits of GridRPC Model

In this chapter, we research and present the performance related benefits and the eventual limits that the GridRPC model delivers to any scientific applications by analysing the experimental results obtained using Grid-enabled Hydropad. The chapter is divided into two sections where each section is composed of an introduction to a possible GridRPC benefit and the experimental results to test it.

In these experiments, we compare the execution times and memory footprints of the GridSolve implementation of Hydropad against its sequential execution on the client machine. For each version, we present the average computation time of one evolution step and the memory footprints of the application on the client machine. The average time is calculated from five separate executions, where each execution is composed of ten evolution steps. The hardware configuration used in the experiments consists of three machines: a client and two remote servers, S1 and S2. The two servers are heterogeneous; however, they have similar performance, respectively 498 and 531 MFlops, and they have an equal amount of main memory, 1GB each. The bandwidth of the communication link between the two servers is 1Gb/s. The client machine, C, is a computer with low hardware specifications, 248MFlops of performance. The client to server connection

varies depending on the experimental setup. We use two setups, C1 with a 1Gb/s connection and C100 with a 100Mb/s communication link. For each experiment conducted, table 4.1 shows the initial problem parameters, the corresponding data sizes and the total memory used during the execution of Hydropad on a single machine. The quantity of memory available in the client machine varies as well depending on the experimental setup. We use two configurations: C-1 with 1GB of memory, which is large enough to avoid paging, and C-256 with 256MB of memory, that undergoes paging for larger problems.

**Table 4.1:** *Input values and problem sizes for the Hydropad experiments*

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

## 4.1 Faster Solution of a Given Problem.

Grid-enabled Hydropad has the potential to perform the simulations of the same given size faster than the original Hydropad on the client machine. There are two main reasons for this:

- The Hydropad application includes two independent tasks, the baryonic matter task and the dark matter task, that can be executed in parallel. The non-blocking GridRPC task call function allows the implementation of parallel execution on remote servers of the Grid environment. This parallelisation decreases the computation time of the application.

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

However, this decrease of the computation time does not come for free. The application will pay the communication cost due to remote execution of the tasks. If communication links connecting the client machine and the remote servers are relatively slow, than the acceleration of computations will be compensated by the communication cost resulting in a higher total execution time of the application than in the case of its sequential execution on the client machine. On the other hand, the experimental results in the next section show that the GridRPC version of Hydropad can achieve faster execution times with larger problem sizes also when there is a relatively slow link speed, thanks to reduced use of the client machine memory of GridRPC remote computations.

**Table 4.2:** *Experimental results with GridSolve using client C1-1 that has 1Gb/s network link to the servers and 1GB of memory*

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

Table 4.2 shows the results obtained by local computations and by the GridSolve version of Hydropad using C1-1 as the client machine which has a fast network connection and large quantity of memory. In this table and in the following ones, the symbol $S_p$ stands for speed-up. It is possible to see that the GridSolve version is faster than the local sequential computation. The speedup obtained is

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

## 4.2 Reduced Client Memory Use and Paging

Grid-enabled Hydropad has the potential to perform larger simulations faster than in the case of sequential execution, thus resulting in higher accuracy. Indeed, the baryonic and dark matter tasks allocate temporary memory during their execution. Remote execution of these tasks will decrease the amount of memory used on the client machine as the temporary memory is now allocated on remote machines. Therefore, within the same memory limitations on the client machine (such as the amount of memory that can be used by the application without heavy paging), Grid-enabled Hydropad will achieve high performance for larger simulations.

Table 4.3 shows the results obtained by the GridSolve version when the client machine used, C100-256, has a slow client-to-servers connection of 100Mb/s and the quantity of memory available is only 256MB. 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 for performing large simulations, and wants to use a powerful Grid environment through a slow network link. Table 4.3 also presents the scale of paging that occurs in the client machine during the executions. It is possible to see that for the local computation the paging is taking place when the problem size is equal to or greater than the machine memory of 256MB.

In these experiments *light* paging means that paging is occurring only in some task calls and the amount of paging is approximately 10% of the main memory (approx. 25MB). *Normal* paging means that paging is occurring on almost every task call and the amount of paging is approximately 40% of the main memory

**Table 4.3:** *Experimental results with GridSolve using client C100-256 that has 100Mb/s network link to the servers and 256MB of memory*

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

(approx. 100MB). *Heavy* paging means that all task calls cause a memory page and almost 100% of the main memory is paged (approx. 256MB).

The GridSolve version is slower than the local computation when the client machine is not paging. This is happening because there is a large amount of data communication between tasks. So for this configuration, the time spent communicating the data compensates for the time gained by computing tasks remotely. However as the problem size gets larger and the client machine starts paging, the GridSolve version becomes faster than the local computation, even in the case of slow communication between the client and server machines. This trend is also in figure 4.1 on page 49. In the GridSolve version the paging is occurring later than the local version, when the problem size is around 310MB, as shown in table 4.3. The GridRPC implementation can save memory due to the temporary data allocated remotely in the tasks and consequently increase the problem size that will not cause the paging. Furthermore in the sequential local execution the paging is taking place during a task computation, while for the GridSolve version the paging occurs during a remote task data communication. Hence for the GridSolve version of Hydropad the paging on the client machine does not negatively affect the execution time of the experiments.

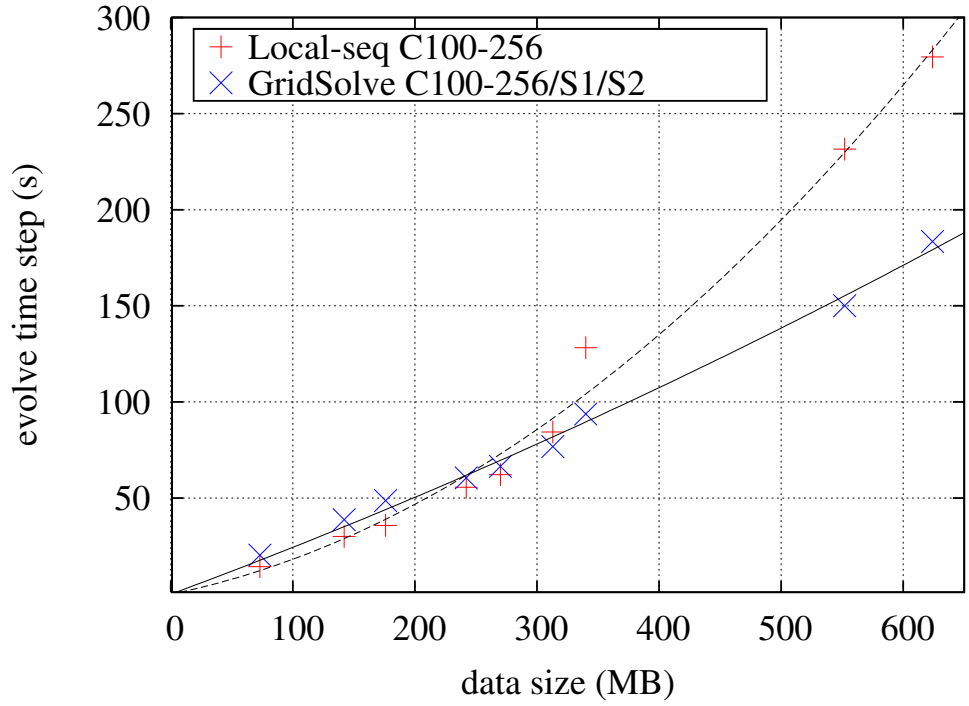



**Figure 4.1:** *Execution time of the evolution step, with varying problem sizes, for the local and GridSolve versions of Hydropad using client C100-256*

## 4.3 Summary

The use of Grid-enabled Hydropad as a performance analysis tool shows that tightly synchronised applications can also achieve many benefits from the use of a GridRPC system (as shown in section 3.4, not only performance related). Furthermore, the experiments, performed using GridSolve, show that the GridRPC version of Hydropad is faster than a local computation when there is a 1Gb/s client-to-server link speed connection. This is due to the remote execution of tasks on powerful servers and the parallel execution of the baryonic and dark matter tasks. Thus Hydropad, despite being a class III application, can achieve good performance with a GridRPC system. However, this performance improvement depends on the client-to-server link speed and generally on the client side hardware. Regardless of this limitation, the experimental results show that the GridRPC programming model allows faster execution of large computations than

would be possible using only a local machine. The benefits of GridRPC permit a programmer to have an alternative option when executing scientific applications.

The GridRPC implementation of Hydropad has some advantages over the sequential local computation. However, it is evident that the model of execution utilised by GridRPC is not optimal. In a GridRPC system all tasks are mapped individually. The mapper will always choose the fastest available server at the instant that a task is called, regardless of the computational size of the task and regardless of whether the task is to be executed sequentially or in parallel. A drawback of this behaviour is highlighted by Grid-enabled Hydropad. The parallel tasks in Hydropad are not computationally balanced. The baryonic task is computationally far larger than the dark matter one, $C_{bm} \gg C_{dm}$. When a GridRPC system goes to map these two tasks, it does so without the knowledge that they are part of a group to be executed in parallel. Its only goal is to minimise the execution time of individual tasks as it is called by the application. If the smaller dark matter task is called first it will be mapped to the fastest available server. With the fastest server occupied, the larger baryonic task will then be mapped to a slower server and the overall execution time of the group of tasks will be sub-optimal.

Another constraint of the GridRPC model, which influences the performance of Hydropad as with any other application, is that all the data objects have to pass through the client machine and thus the client hardware acts as a performance bottleneck. Various servers computing tasks with data dependencies on each other cannot communicate with each other directly. It is possible for the application programmer to avoid this issue by implementing data caching in remote tasks. However it requires the programmer to make heavy modification to the tasks and this is a clear drawback. It also means that remote tasks passing data to each other must all run on the same server, where the data they need is cached. Another possibility for the application programmer is to store the main data in different files and then use the file name as task arguments instead of data objects. However, for this method to work it needs a global file system that is deployed through the various servers and heavy modification to the tasks and client codes as well.

# Chapter 5

# SmartGridRPC: Overcoming the Limitations of GridRPC

SmartGridRPC [17] is an extension of GridRPC that has been designed to bypass the limitations of the GridRPC model of execution. It expands the individual task map and client-server communication model of GridRPC by implementing:

- The mapping of groups of tasks.
- The server to server communication.
- The broadcast communication.
- The automatic data caching on servers.

Collective mapping of groups of tasks, using a fully connected network, allows a SmartGridRPC middleware to find an optimal mapping solution for an application that fully exploits a Grid environment. Furthermore, the direct server to server communication, broadcast communication and automatic data caching minimise the amount of memory used on the client and the volume of communication necessary between client and server. The data objects can reside only on the servers where they are needed and they can be moved directly between servers without having to pass through the client.

The main goal of the SmartGridRPC model is to provide these functionalities to the user in a practical and simple way. To achieve this, it provides only minor changes and an addition to the API of GridRPC. The SmartGridRPC model

is designed so that when it is implemented it is interoperable with the existing GridRPC implementation. An application programmer can take advantage from the improved performance of using SmartGridRPC by making only minor modifications to any application that is already GridRPC enabled.

In this chapter, we introduce the SmartGridRPC programming model and its differences to the GridRPC one. Then, we show how SmartGridRPC allows the mapping of a group of tasks and the server-to-server communication while adding only two new methods in the GridRPC API. We also introduce the internal structure of the SmartGridSolve middleware, which implements the SmartGridRPC model, and we present the structure of the task graph used by this middleware. Furthermore in this chapter, we show how the SmartGridRPC model can be easily implemented in Hydropad. Finally, we examine the various components and functionalities implemented into GridRPC systems that are related to new features of the SmartGridRPC project.

## 5.1 SmartGridRPC Model

The SmartGridRPC model is similar to the GridRPC one (see section 2.1.1), with the same structure, client-server architecture and functionalities. The main difference is that the SmartGridRPC model defines:

1. New API to specify a block of code, in which a group of GridRPC task calls should be mapped collectively.

2. A specific performance model, which permits the estimation of the execution time of a group of tasks on a fully connected network instead of a single task on a star network.

3. A modification to the client application run-time, which adds the generation of part of the performance model and the processing of the group of tasks.

**Performance model** The SmartGridRPC model does not specify the structure, internal data and implementation of the performance model since there are many numerous methods for estimating the execution time of the group of tasks on

a fully connected network. Furthermore, it does not indicate where the performance model is generated and stored. However, SmartGridRPC defines that the performance model has to contain the necessary information to estimate:

- The execution time of a task on a server.
- The execution time of multiple tasks on a server and the effect that 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 between different servers.

Additionally, the SmartGridRPC model defines the logical structure of the performance model. The approach chosen in SmartGridRPC is similar to the one implemented in mpC [61] and HeteroMPI [62] that use the performance model of the application and the performance model of the heterogeneous network to optimally map the application on the underlying network. Thus, the performance model in SmartGridRPC is logically divided in two sub parts:

1. *Task Graph*: This part is application specific, such as the list of tasks in the group, their order, the dependencies between tasks and the values and sizes of the arguments in the calling sequences. These arguments can be used to calculate the size of the data objects and thus the computational weight of the remote tasks.
2. *Network Performance Model*: This part is server and network specific, such as the dynamic performance of the various servers on the Grid and the communication links between them and the client. Furthermore, it contains the performance model of each task available on a Server.

**Client application run-time** The client application provides the data needed to generate the task graph part of the performance model. This information is generated automatically at run-time by using the SmartGridRPC method *grpc_map*.

Furthermore, this method allows the application programmer to specify the group of GridRPC calls to be mapped collectively, see example of table 5.1. This is done

**Table 5.1:** *Example of a group of task calls specification in SmartGridRPC*

```
1 grpc_map(...) {
2   ...
3   //group of GridRPC calls to map collectively
4   ...
5 }
```

by using a set of parenthesis, which follows the map function, to specify a block of code. This code contains the group of GridRPC task calls that should be mapped collectively. When this function is executed at run-time, the code and GridRPC task calls within the parenthesis of the function are iterated through twice and three phases are performed:

- *Discovery phase*: On the first iteration through the group of tasks, each GridRPC task call within the parenthesis is discovered but not executed, thus 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 a time. Then, the client can look up and retrieve handles for all tasks in the group at the same time.

- *Mapping phase*: After the discovery phase, the task graph can be generated, on the client or on a different entity, using the information contained in the handle and the other data retrieved during the discovery, such as the order of tasks in the group, their dependencies and the values of each argument in a task calling sequence. Then the task graph, in conjunction with the network performance model, is used in the mapping phase to generate a mapping solution. An application programmer can specify the mapping heuristic using the SmartGridRPC map function. The SmartGridRPC model does not define any specific mapping heuristic implementations but it allows different ones to be added. Therefore, SmartGridRPC provides an ideal framework for testing and evaluating different mapping algorithms.

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

## 5.2 SmartGridRPC Communication Model

The SmartGridRPC communication model differs from the GridRPC one because it is based on a fully connected network topology (figure 5.1), instead of a star network topology (figure 2.2). Since tasks are mapped collectively in SmartGridRPC, dependencies between tasks are known during the mapping phase. This knowledge permits an underlying SmartGridRPC middleware to identify which are the source tasks and the destination tasks of each data object. Thus, an object can be send directly from the server of the source task to the server of the destination task without passing through the client. Furthermore, if the source task and destination task are both executing on the same server then this output could be cached on the local file system or on memory. Finally, if an object has more than one destination, this object can be broadcast to multiple servers.

## 5.3 SmartGridRPC API

SmartGridRPC introduces two new methods in addition to the one of GridRPC, the *grpc_map* and *grpc_local* functions. The SmartGridRPC map method is used to specify the block of code that contains the group of GridRPC tasks calls to be mapped collectively. This function is also used to automatically retrieve the information needed to build the task graph. The method *grpc_local* is used by the application programmer to indicate a block of code that contains local computation. Table 5.2 shows an example containing the two new methods. The first argument of the map function is a string that specifies the mapping heuristic used during the mapping phase. There is also a fault tolerant version of this function,

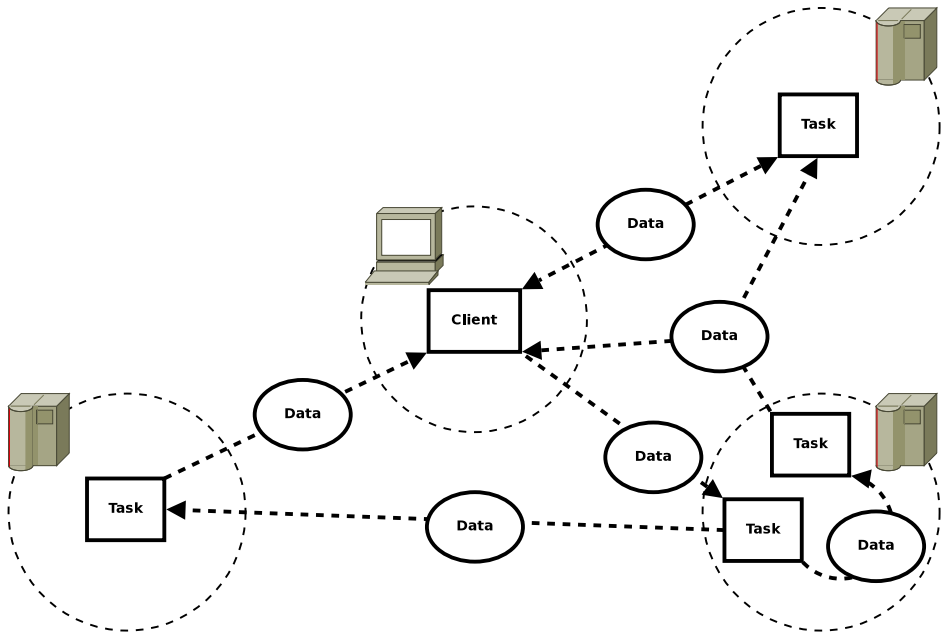



**Figure 5.1:** *SmartGridRPC communication model*

*grpc_map_ft*, where the mapping solution generated does not implement server-to-server communication.

When the *grpc_map* function is called, the code within its parenthesis will be iterated through twice as previously described in section 5.1. The first iteration does not execute the remote task calls; however the non GridRPC code is executed normally. Therefore, the method *grpc_local* is used to indicate the part of the code that cannot be executed twice, like a function executed locally that computes some variables. Thus, the code inside the block is executed only during the second iteration. The arguments passed to the *grpc_local* method are the data objects, used or generated by remote tasks, that are required locally. This information is used to determine when data objects will be sent back to client and it is essential to generate the task graph.

The procedure of iterating the code twice, performed by the *grpc_map* and *grpc_local* methods, is called automatic task graph generation. This technique permits an application programmer to easily add the mapping of a group of tasks in a pre-existing GridRPC code without the need to add extra information of the underlying algorithm.

**Table 5.2:** *Example of SmartGridRPC methods*

```
1  grpc_map("map heuristic") {
2    // Code iterated twice
3
4    grpc_call(&handle,obj1,obj2,...);             // Blocking
5
6    grpc_call_async(&handle,&sid,obj3,obj4,...); // Non-blocking
7
8    grpc_local(obj1,obj2,...)
9      {
10       // Code ignored during the first iteration
11     }
12
13   // Asynchronous tasks control
14   grpc_wait(sid)
15   grpc_wait_all();
16 }
```

## 5.4 SmartGridRPC Middleware

SmartGridSolve [18] is an extension of the GridSolve middleware that implements the SmartGridRPC model. The first version of SmartGridSolve was called SmartNetSolve [19] and it was based on NetSolve. 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. In this work, the GridSolve middleware with the SmartGridSolve extension activated will be referred to as simply SmartGridSolve.

### 5.4.1 SmartGridSolve Internals

The GridSolve performance model includes functions for calculating the computation load and communication load of the called task. Furthermore, it contains methods for calculating the dynamic performance of the servers and client-server links. The SmartGridSolve network performance model extends this GridSolve performance model by including the dynamic performance of each server-to-server link. These performances are taken periodically utilising the same techniques used by GridSolve. In the future, SmartGridSolve will implement performance models such as the *Functional Performance Model*, which is described in [63, 58]. Other future possible implementations could include the performance models used in DIET and Ninf-G, such as NWS and MDS. The SmartGridSolve task graph implementation is detailed in section 5.4.2.

In GridSolve, the agent entity works as the GridRPC resource discovery functionality. Thus, the agent maintains a list of all available servers and their registered tasks. After being registered, a server sends to the agent the information about its computational performance and the speed of client-to-server link. This information is update dynamically because of the continuously changing status of the server and network link. When a task is added to the agent, the server sends also the task description retrieved from the respective gsIDL provided. This information contains the computational complexity of the task in the form of a mathematical formula that uses the sizes of data-objects. The agent uses all these data to generate the performance model.

The SmartGridSolve agent performs the same work of the GridSolve one. In addition, it receives from the servers the dynamic performance of each connection link between them. Another job of the SmartGridSolve agent is to generate at run-time the task graph and the network performance model. At the end of the discovery phase, the client sends the information retrieved about the group of tasks to the agent. This information plus the computational complexity of each task, already present on the agent, are used to generate the task graph. After the performance model is built on the agent from the two parts, it is sent to the client where it is used to perform the collective mapping of the group of tasks.

### 5.4.2 Task Graph

The task graph in SmartGridSolve is implemented as a direct acyclic graph (DAG) structure. It is used to fully represent the group of tasks, and thus the underlying algorithm, defined by the *grpc_map* method. The task graph specifies the order of tasks execution and their synchronisation (whether they are executed in sequence or in parallel), the data dependencies between tasks, the load of data communication and the task's computational volume. This information is essential to choose the best server to execute a task and to minimise the amount of data movement in the Grid network.

Figure 5.2 on page 60 shows an example of a task graph. The example contains five remote tasks, with two parallel executions composed of two tasks, and a local computation. The rectangles in the graph represent remote tasks, the diamonds represent the client computation and the circles represent the data objects. The incoming arrows of these circles indicate their source, whether it is the client or another remote task and the outgoing arrows indicate their destination. If multiple tasks require the same input object then more than one link will emanate from this object. This indicates data broadcast. The dotted arrows highlight the order of task calls and if the tasks are executed in sequence or parallel. The values inside the circles and rectangles are respectively the size of a data object and the computational complexity of a task. These are correlated to the values of the respective task arguments. In a task graph all the input data objects that are not generated by a remote task are retrieved from the client. All the output objects that are not used by a task call are sent back to the client. In the current implementation, only non-scalar arguments (matrix, vector etc.) are represented.

The dependencies between tasks are determined by examining the pointers of non-scalar arguments of the calling sequence of each task. Furthermore, the gsIDL description is used to determine whether the objects are inputs or outputs. The functions for calculating the computation load of each task in the group are generated using the mathematical formulas specified by the programmer in the gsIDL description. The information retrieved from the gsIDL is also used to generate the functions for calculating the communication load of each non-scalar data objects in the group.

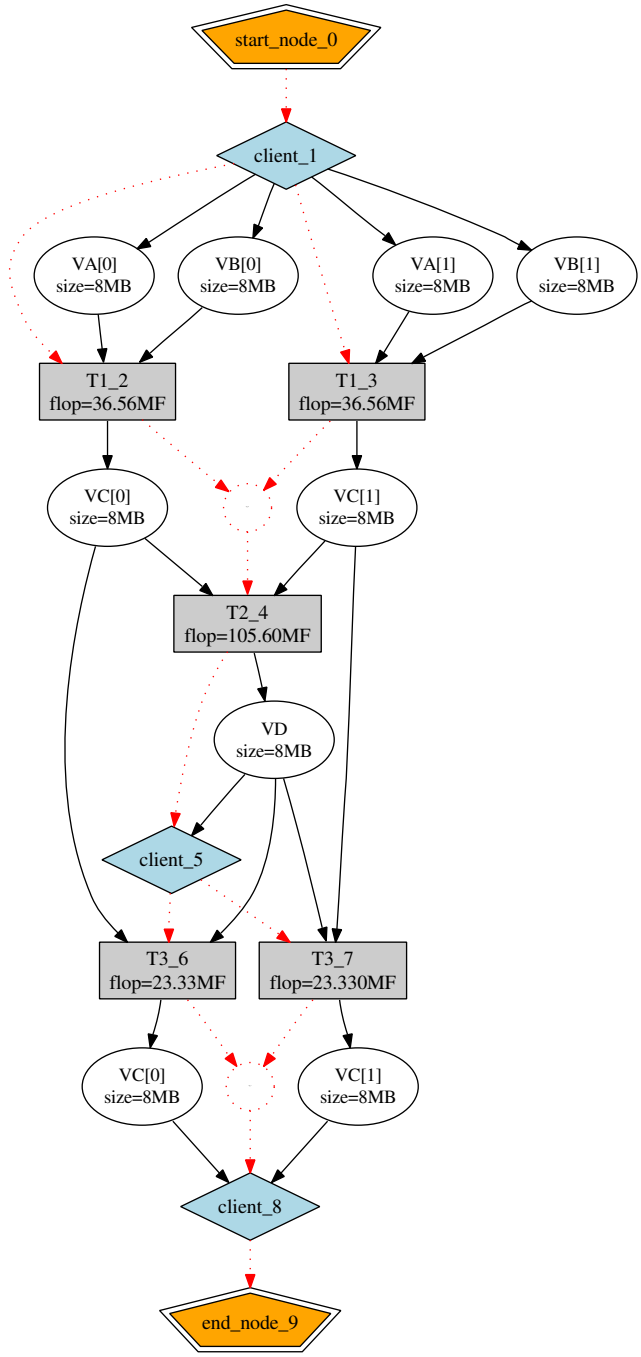



**Figure 5.2:** *Example of SmartGridSolve task graph*

## 5.5 SmartGridRPC Implementation of Hydropad

In this section, we introduce how the SmartGridRPC API can be used in Hydropad. The code in table 5.3 shows the changes required to use the SmartGridRPC model in Hydropad, in contrast to those shown in table 3.2 where we illustrate the changes required for GridRPC. The only differences between the two examples are the minor additions of the *grpc_map* and *grpc_local* methods. This simple implementation permits Hydropad to work with SmartGridSolve and thus to be able to map one or many evolution cycles collectively in a fully connected network.

**Table 5.3:** *Hydropad implementation using SmartGridRPC API*

```
1  t_sim=0;
2  while(t_sim<t_univ) {
3    grpc_map("ex_map") {
4
5      grpc_call(grav_hndl,phiold,...);
6
7      if(t_sim==0){ grpc_call(initvel_hndl,phi,...);}
8
9      grpc_call_async(dark_hndl,&sid_dark,x1,...);
10     grpc_call_async(bary_hndl,&sid_bary,nes,...);
11
12     grpc_wait(sid_dark); /*wait for non blocking*/
13     grpc_wait(sid_bary); /*calls to finish*/
14
15     grpc_local() {
16       timestep(t_step,...);
17       t_sim+=t_step;
18     }
19   }
20 }
```

The code enclosed in the *grpc_map* block will be iterated through twice. On the first iteration, each *grpc_call* and *grpc_call_async* is discovered but not executed. At the beginning of the second iteration, when all the tasks within the scope of the block have been discovered, the respective task graph is generated.

Then, the second iteration is performed and the tasks are executed remotely using the task graph to aid their mapping. The *grpc_local* function is used by the application programmer to indicate when a local computation is executed. At run time on the first discovery iteration, the code within the parenthesis after this method is not executed. This is to avoid computing local executions when generating the task graph for the group of remote tasks.

The mapping in the code of table 5.3 is performed at every iteration of the main loop. This can generate a good mapping solution if the Grid environment is not a stable one. For example, where there are other applications' tasks running on the Grid servers. If the Grid environment is dedicated, where only one application executes at a time, a better mapping solution may be generated if the area to map contains more tasks, i.e. two or more loop cycles. A simple solution could be including an inner loop within the *grpc_map* code block. The application programmer could increase the number of tasks mapped together by changing the number of iterations of the inner loop.

**Table 5.4:** *Dynamic selection of the number of evolution cycles included in the group of tasks to map collectively*

```
1  t_sim=0;
2  while(t_sim<t_univ) {
3    grpc_map("ex_map") {
4      nsteps = number_of_steps_to_map(...);
5      for(i=0;i<nsteps;i++)
6        grpc_call(grav_hndl,phiold,...);
7        ...
8        grpc_local() {
9          timestep(t_step,...);
10         t_sim+=t_step;
11       }
12     }
13   }
14 }
```

This type of coarse mapping would be more favourable on a Grid environment that is highly stable. For example, a Grid that consists of dedicated servers or

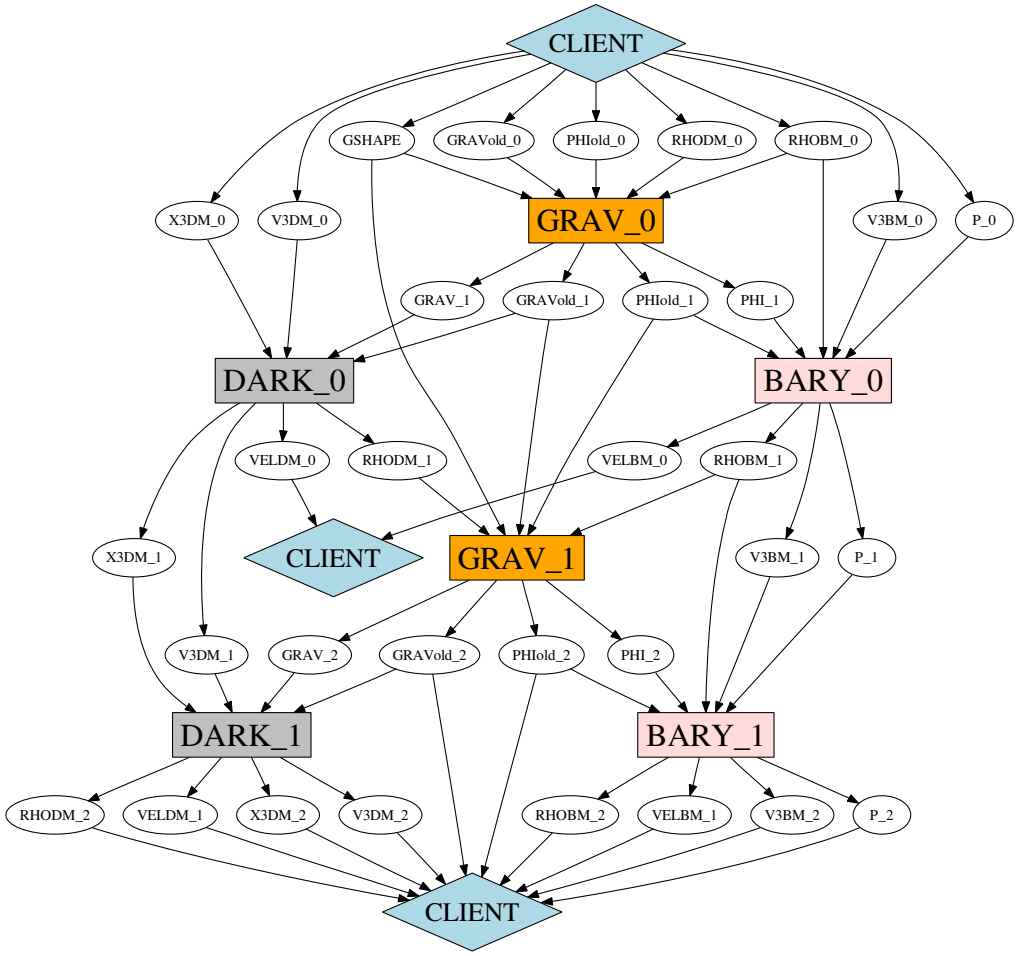



**Figure 5.3:** *Task graph for two evolution cycles*

servers that are idle. However, if in a Grid the computation and communication loads are highly changeable then it might be more advantageous to adjust the frequency of mappings depending on these loads. The code of table 5.4 shows an example where the number of evolution cycles to map, *nsteps*, is dynamically chosen thanks to a special function. This method may change its return value by evaluating the performances of previous executions of collective mappings. Figure 5.3 shows an example of a task graph generated from the SmartGridRPC implementation of Hydropad when two cycles of the evolution step are grouped together ($nstep = 2$).

## 5.6 Related Work

In this section, we examine various components and functionalities implemented into GridRPC systems that are related to new features of SmartGridRPC project. Therefore, we focus on papers that introduce in GridSolve, Ninf-G and DIET the features: direct server-to-server communication, data persistence and collective mapping of a group of tasks. These works are presented in chronological order.

In 1999, task farming [26] functionality was implemented into NetSolve and it was designed for embarrassingly parallel applications. It allows a certain class of tasks, called farming jobs, to be processed collectively by using specific methods. However, these tasks are not mapped collectively but individually and the computation loads of subsequent tasks are dynamically adjusted at run-time based on previous task response times. Therefore the mapping cannot take advantage of characteristics of the group, such as data dependencies. Furthermore, the group of tasks is called with one atomic call and therefore intermediate results cannot be viewed or analysed. Another limitation of this approach is that conditional statements and client computations cannot exist in the scope of the group of tasks.

2000, task sequencing [12] was added to the NetSolve middleware. Using the task sequencing method a group of tasks is processed collectively and thus data dependencies can be analysed. This group of tasks is subsequently mapped onto a single server and if any data dependencies exist, the data is stored locally and not sent back to the client. The limitations of task sequencing are that the group of tasks can only be mapped to a single server and that conditional statements and client computations cannot exist in the scope of the group of tasks.

In 2001, data persistence between servers was implemented into NetSolve [39] and it was later implemented in the GridSolve system [40]. This is achieved by adding a new method that allows the user to explicitly outline in the client code data dependencies between tasks. The limitations of this approach are that the user has to explicitly specify data dependencies and that tasks are mapped individually.

In 2002, Distributed Storage Interface (DSI) [10] was added to NetSolve. DSI is another feature that attempts to minimise data movements in the Grid network. With DSI, data can be stored in storage depots closer to servers where the data is required. The depots are implemented using Internet Backplane Protocol

(IPB) [14]. This save multiple transmissions of the same data, since DSI permits the data to be communicated once from the client to the storage depot. A data handle is then used by the server to retrieve the stored data when executing a task. This reduced communication times, however there is not direct server-to-server communication and the user has to explicitly specify data dependencies.

In 2005, DTM (Data Tree Manager) [22] was implemented into DIET. DTM allows data to be left on a server after a task and then retrieved by another server during a different task computation. Furthermore, the data can be part of different application's executions. In DTM, data persistency is identified using data handle with specific new methods, therefore the user has to explicitly specify these data in the client code. Around the same time, also JUXMEM [23] was introduced. It permits data to be shared between DIET servers using a peer-to-peer method and a hiearchy of communication groups. In JUXMEM data sharing has to be directly specified by the programmer in the client code and also in task code.

In 2006, SmartNetSolve, the predecessor of SmartGridSolve, was introduced. SmartNetSolve was already implementing the direct server-to-server communication, data caching and collective mapping of a group tasks features. However, in SmartNetSolve, the automatic generation of task graph was not yet implemented, therefore the task graph was explicity specified by the user using a XML file, see subsection 8.4.2 for more details.

In 2006, the NetSolve task sequencing was expanded to be distributed and to be used with the GridRPC model [81]. This new functionality allows direct data transfer between servers when executing a task sequencing. Therefore, multiple servers can be used for a sequence of tasks instead of a single server as it is in the case of the original NetSolve task sequencing. However, all the other limitations of the original version persist also with this extended version.

In 2006, the special agent called MADAG was implemented in DIET which handled workflow submissions, where a workflow is similar to a task graph. This workflow is directly specified by the user using a XML file and it is executed with one atomic call. The limitations of this approach are that intermediate results cannot be viewed or analysed and conditional statements and client computations cannot exist in the scope of the group of tasks. A more detail analysis is presented in subsection 8.4.2.

## 5.7 Summary

In this chapter, we have presented the SmartGridRPC model, which is an extension to the GridRPC model that aims to achieve higher performance. SmartGridRPC expands GridRPC, which maps tasks individually on to a star network, by providing the collective mapping of a group of tasks on a fully connected network. These functionalities are achieved by adding only two new simple methods into the GridRPC API and we have shown how easy it is to use these two methods for implementing the SmartGridRPC model into the Hydropad application. The impact in performance that the SmartGridRPC extension can have to Hydropad, as well as to any other distributed scientific application, can be great and they are evaluated in the next chapter.

# Chapter 6

# Experimental Analysis of Performance Potential and Limits of SmartGridRPC Model

In this chapter, we analyse the performance related benefits and the eventual limits that the new SmartGridRPC model delivers in comparison to the GridRPC model. We compare the experimental results obtained by the SmartGridRPC implementation of Hydropad using the SmartGridSolve middleware to those obtained by the GridRPC implementation using GridSolve and the sequential execution of Hydropad. The problem sizes utilised in the experiments (table 4.1 on page 45) and the hardware configurations are the same as in previous experiments shown in chapter 4. In these experiments, we present the average computation time of one evolution step and the memory footprints of the application on the client machine. The average time is calculated from five separate executions, where in the SmartGridSolve version the *grpc_map* block contains ten evolution cycles (ex. $nstep = 10$ in code of table 5.4).

In the first experiment, we use the same hardware configuration of table 4.3. The client machine used, C100-256, has a slow client-to-servers connection of 100Mb/s and only 256MB of memory available. As previously mentioned, this configuration simulates the situation where a user has only access to a slow client machine with low hardware specification, which is not suitable to perform large

**Table 6.1:** *Experimental results with SmartGridSolve, GridSolve and local execution using client C100-256, which has 100Mb/s network link to the servers and 256MB of memory*

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

| | SmartGridSolve | | | |
|---|---|---|---|---|
| P. ID | Time Step | Paging | $S_p$ vs Local | $S_p$ vs GS |
| P1 | 7.31s | No | 1.96 | 2.77 |
| P2 | 15.06s | No | 2.00 | 2.57 |
| P3 | 16.36s | No | 2.19 | 2.97 |
| P4 | 28.06s | No | 1.98 | 2.16 |
| P5 | 27.54s | No | 2.26 | 2.41 |
| P6 | 27.78s | No | 3.04 | 2.76 |
| P7 | 30.81s | Light | 4.16 | 3.04 |
| P8 | 48.04s | Light | 4.82 | 3.12 |
| P9 | 60.74s | Light | 4.06 | 3.02 |

simulations, and wants to use a powerful Grid environment through a slow network link. Table 6.1 shows the results obtained by the SmartGridSolve version for this configuration. In this table and in the following one, the symbol $S_p$ stands for speed-up, while GS is the abbreviation for GridSolve. This table shows that the SmartGridSolve version is much faster than the GridSolve and the sequential versions. The speed is around three times that of GridSolve (see figure 6.1 for the trend) and the speedup versus the local sequential version is over 4 in the case of larger problems.

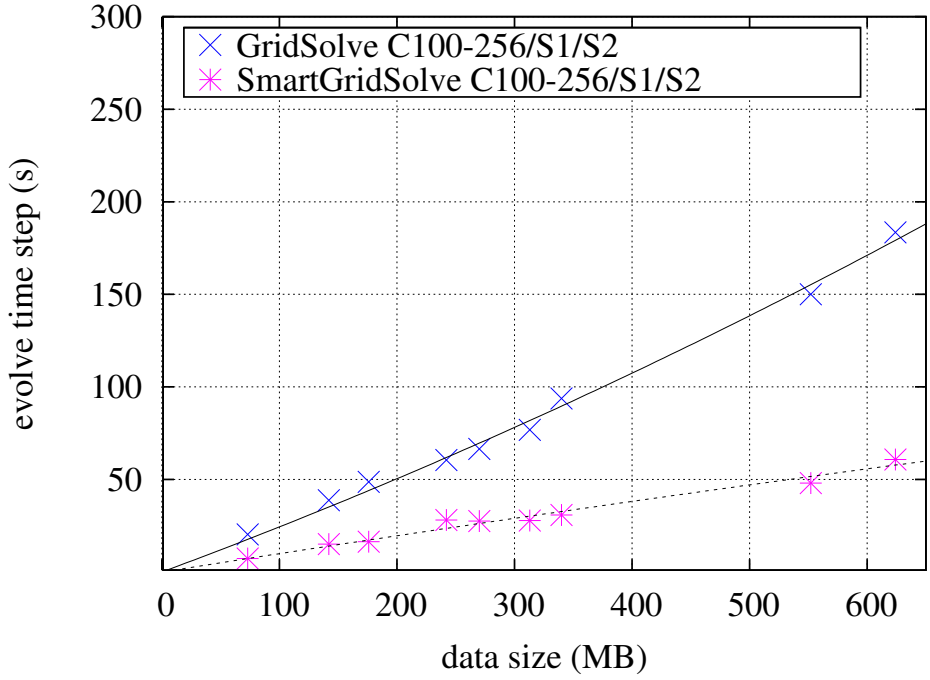



**Figure 6.1:** *Execution time of the evolution step, with varying problem sizes, for the GridSolve and SmartGridSolve versions of Hydropad using client C100-256*

These performance improvements are due to the new features of the SmartGridRPC model; the mapping of groups of tasks and the fully connected network topology. These features involve performance related benefits to Grid-enabled Hydropad, as well as other class III applications, such as improved mapping, improved data movement and further reduced memory usage. In the following sections, we study these new benefits by using specific hardware configurations and setup for the experiments

## 6.1 Improved Computation Load

One important feature of SmartGridRPC is the possibility to have full knowledge of the underlying algorithm and its remote task executions during the mapping phase. This permit a SmartGridRPC middleware to easily implement superior mapping heuristics in comparison to a standard GridRPC middleware. Thus, the SmartGridRPC implementation of Hydropad has the potential to have an improved computational load of tasks compared to the GridRPC version. For example, a GridRPC middleware does not have the knowledge that the baryonic and dark matter tasks are executed in parallel. Therefore its mapper could map the smaller task to the fastest server if it is executed first. Occasionally, a GridRPC middleware could map both tasks to the same server because its mapping heuristic or its performance model fail to consider the possible perturbation between the two tasks. Instead, a SmartGridRPC middleware knows about the parallel execution of the dark matter and baryonic matter tasks and their respective computational weight. Thus, its mapper can produce a better result and consequently an improved load balancing.

In the computational load experiments, we compare the average computation time of one evolution step achieved by the GridSolve version versus the SmartGridSolve version of Hydropad where SmartGridSolve is setup to utilise the same network topology of GridSolve (star-network); i.e. without direct server-to-server communication, server-caching and broadcast communication. Consequently, the performance gains obtained by the SmartGridSolve version are due only to the improved mapping method. In these experiments, we use C1-1 as the client machine. This machine has a high speed network connection of 1Gb/s to the servers and 1GB of main memory. Table 6.2 shows that the SmartGridSolve version of Hydropad is faster than the GridSolve version. The collective mapping of SmartGridRPC, despite Hydropad having only two parallel tasks, can produce a faster execution time than the individual task mapping of GridRPC. As previously mentioned, the GridSolve middleware maps the smaller dark matter task to the fastest server or occasionally both tasks to the same server. Instead, in SmartGridSolve version of Hydropad the larger baryonic matter task is always mapped to the faster server and the smaller dark matter task to the slower one.

**Table 6.2:** *Experimental results using only star-network topology (i.e. no direct server-to-server communication) and client C1-1 that has 1Gb/s network link to the servers and 1GB of memory*

| | GridSolve | SmartGridSolve | |
|---|---|---|---|
| P. ID | Time Step | Time Step | $S_p$ vs GS |
| P1 | 9.40s | 7.09s | 1.33 |
| P2 | 18.38s | 15.27s | 1.20 |
| P3 | 20.82s | 16.17s | 1.29 |
| P4 | 30.81s | 29.02s | 1.06 |
| P5 | 32.00s | 28.99s | 1.10 |
| P6 | 36.81s | 29.88s | 1.23 |
| P7 | 37.22s | 30.88s | 1.21 |
| P8 | 67.04s | 50.05s | 1.29 |
| P9 | 112.05s | 53.35s | 2.10 |

## 6.2 Improved Communication Load

The SmartGridRPC implementation of Hydropad, due to its communication model, minimises the amount of data that pass through the various client-to-sever links. If all these network links are slow, then the SmartGridRPC version obtains faster execution than the GridRPC one. Another common situation, where the SmartGridRPC model has a performance advantage over GridRPC model, is if the client-to-server links and the server-to-server links speeds vary widely. A SmartGridRPC middleware knows the amount of data that each task need and the speed of each network link, due to the task graph and network performance model. Therefore, it can minimise the amount of data moved through the slowest links. For example, considering the situation where the Grid environment is composed of two very fast servers connected to the client machine with slow links and a very slow server connected with a fast link. The SmartGridRPC implementation of Hydropad knows that the dark matter task uses more data than the baryonic matter task while being less computational intensive. Thus, the mapper could optimally decide if the amount of time wasted during the computation of the dark matter task on the slow server is balanced by the time gained by the fast communication link. Furthermore, the mapper could optimally decide where to execute

the gravitational task in order to minimise the data movement caused by the data dependencies between the tasks.

In the next experiments, we set-up the client connection to the Grid environment to be slow since in this situation the improved communication load of SmartGridRPC model is most prominent. Table 6.3 shows the results obtained by the SmartGridSolve version of Hydropad using C100-1 as the client machine which has a slow network connection of 100Mb/s. One can see that the SmartGridSolve version is much faster than the GridSolve versions. The increase of speed is over twice that of GridSolve, which is primarily due to the improved communication model of SmartGridSolve.

**Table 6.3:** *Experimental results using client C100-1 that has 100Mb/s network link to the servers and 1GB of memory*

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

Furthermore, one can see that the timing results obtained by SmartGridSolve in table 6.2 are similar to those obtained in table 6.3. This shows that when the client-server links are slow and there is direct communication (table 6.3), the results are similar to when the client links are fast and there is no direct communication. The SmartGridRPC model allows the mapping heuristic to generate solutions, which effectively minimise the communication load on the networks link.

## 6.3 Further Reduced Client Memory Use and Paging

In section 4.2, we discussed that the dark matter and baryonic matter tasks allocate temporary data during their execution. This permits the GridRPC implementation of Hydropad to cause less paging on the client machine for problems with large computations during the remote execution of these tasks. The SmartGridRPC implementation of Hydropad, decreases further the amount of paging on the client due to the direct server-to-server communication and the data caching on the server. These two features allow a SmartGridRPC middleware to minimise the amount of intermediate objects that cross the client machine. Therefore in the SmartGridRPC implementation of Hydropad, the amount of active memory used on the client machine is far less than the GridRPC one. Hence, the paging on the client machine is minimal for computations of large problems. In figure 5.3 on page 63 (the task graph for two evolution cycles), it is possible to see that the majority of the data objects are communicated to the client only at the beginning and the end of the group of tasks execution.

In the memory usage experiments, we utilise as a client the machine C1-256, which has a high speed network connection of 1Gb/s to the servers and has 256MB of main memory. The client machine (with this quantity of main memory) experiences paging during the execution of larger problems. Table 6.4 on page 74 shows the average computation time of one evolution step achieved by the local computation, by the GridSolve version and by the SmartGridSolve version of Hydropad. Table 6.4 also presents the scale of paging that occurs on the client machine during the various executions.

One can see that for the SmartGridSolve experiments the paging on the client machine is less penalising than in the GridSolve and local experiments since the quantity of memory used on the client machine is lower than that of the GridSolve version. Furthermore, in SmartGridSolve the memory paging is happening only at the beginning and at the end of a group of tasks execution. This minimises the impact of paging on the overall execution of the group of tasks. Therefore, the SmartGridSolve version of Hydropad can execute larger problems without the paging having a serious impact on the execution time. One can see that the com-

**Table 6.4:** *Experimental results using client C1-256 that has 1Gb/s network link to the servers and 256MB of memory*

| | Local | | GridSolve | | |
|---|---|---|---|---|---|
| P. ID | Time Step | Paging | Time Step | Paging | $S_p$ vs Local |
| P1 | 14.32s | No | 8.59s | No | 1.67 |
| P2 | 30.05s | No | 18.41s | No | 1.63 |
| P3 | 35.78s | No | 20.19s | No | 1.77 |
| P4 | 55.57s | Light | 31.34s | No | 1.77 |
| P5 | 62.13s | Light | 33.75s | No | 1.84 |
| P6 | 84.33s | Yes | 42.32s | Light | 1.99 |
| P7 | 128.22s | Yes | 63.12s | Yes | 2.03 |
| P8 | 231.56s | Heavy | 109.33s | Heavy | 2.12 |
| P9 | 279.52s | Heavy | 144.31s | Heavy | 1.94 |

| | SmartGridSolve | | | |
|---|---|---|---|---|
| P. ID | Time Step | Paging | $S_p$ vs Local | $S_p$ vs GS |
| P1 | 7.08s | No | 2.02 | 1.21 |
| P2 | 14.47s | No | 2.08 | 1.27 |
| P3 | 15.84s | No | 2.26 | 1.27 |
| P4 | 27.51s | No | 2.02 | 1.14 |
| P5 | 28.17s | No | 2.21 | 1.20 |
| P6 | 28.88s | No | 2.92 | 1.47 |
| P7 | 30.02s | Light | 4.27 | 2.10 |
| P8 | 46.65s | Light | 4.96 | 2.34 |
| P9 | 55.13s | Light | 5.07 | 2.62 |

putation time of the evolution steps for the SmartGridSolve version in table 6.4 is similar to that of tables 6.3 and 6.2. The speedup of the SmartGridSolve version over the GridSolve and local versions of Hydropad increases as the problem gets larger due to paging on the client. This trend is also seen in figure 6.2.

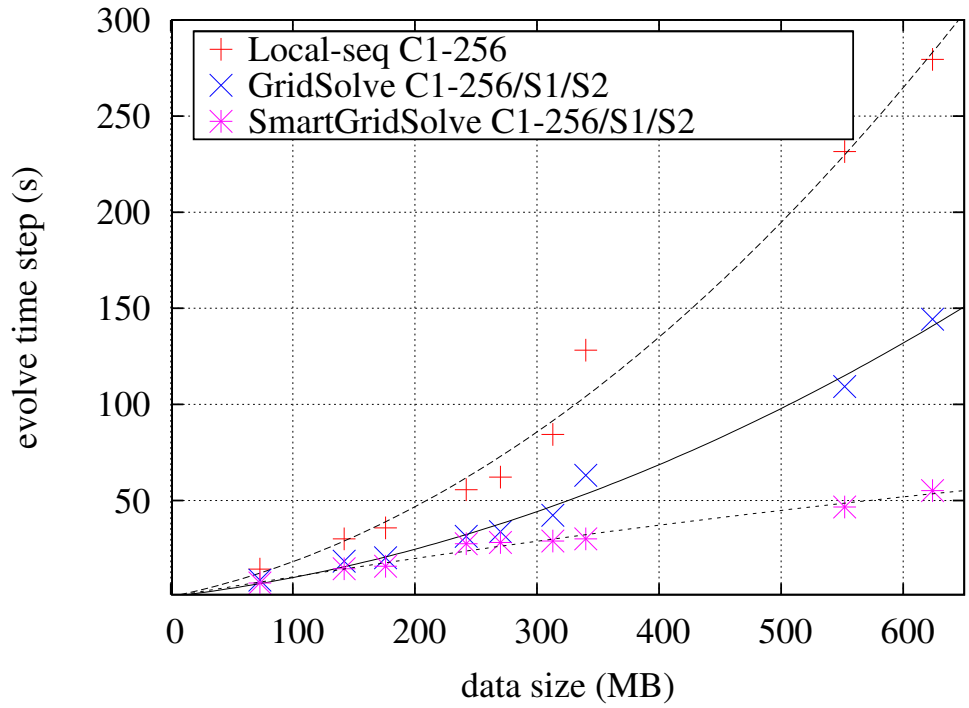



**Figure 6.2:** *Execution time of the evolution step, with varying problem sizes, for the local, GridSolve and SmartGridSolve versions of Hydropad using client C1-256*

## 6.4 Minimal Performance Influence by the Client-Side Hardware

The advantage discussed in this section is a combination of the previous two benefits. The SmartGridRPC implementation of Hydropad achieves higher performance than the GridRPC one due to the improved communication load and the reduced client paging. These two benefits appear when the client machine has a

low amount of memory and a slow client-to-server network link. Therefore, the client-side hardware heavily influences the performance of the GridRPC implementation of Hydropad. Instead, the SmartGridRPC implementation performance is influenced minimally by the client-side hardware. Hence, the SmartGridRPC model permits a distributed scientific application to fully take advantage of the Grid environment without being penalised by the client-side hardware. Figure 6.3 shows this trend. We compare the results obtained by the GridSolve and SmartGridSolve versions of Hydropad when the two configurations of the client used are the optimal one, C1-1, and the worst case, C100-256. It is possible to see that in the case of GridSolve the performance change depends dramatically on the hardware used, and thus the client-side hardware is a performance bottleneck, while for SmartGridSolve the performance is similar and therefore there is not a large overhead from the client hardware.

## 6.5 Summary

The experiments performed in this chapter show that the SmartGridRPC model allows Hydropad to obtain quite significant performance gains in comparison to the GridRPC implementation and to the sequential one. These further performance benefits of SmartGridRPC over GridRPC are identified as the improved computation load, improved communication load and further reduced client memory usage and paging. Furthermore the experiment shows that these performance benefits are not influenced negatively by the bottleneck of a potential slow client-servers connection and low amount of memory in the client machine as much as with the GridRPC model. Therefore, SmartGridRPC is more scalable and permits a tightly synchronised Grid application, as well any other scientific distributed applications, to fully take advantage of the Grid environment; while keeping the non-performance related benefits of the RPC programming model.

The gains obtained by the SmartGridRPC implementation are due to its two new features; the collective mapping of a group of tasks and the use of a fully connected network. As previously discussed, the direct server-to-server connections, data broadcast and improve mapping items are the consequence of full knowledge of the group of tasks algorithm. In SmartGridRPC this knowledge is represented

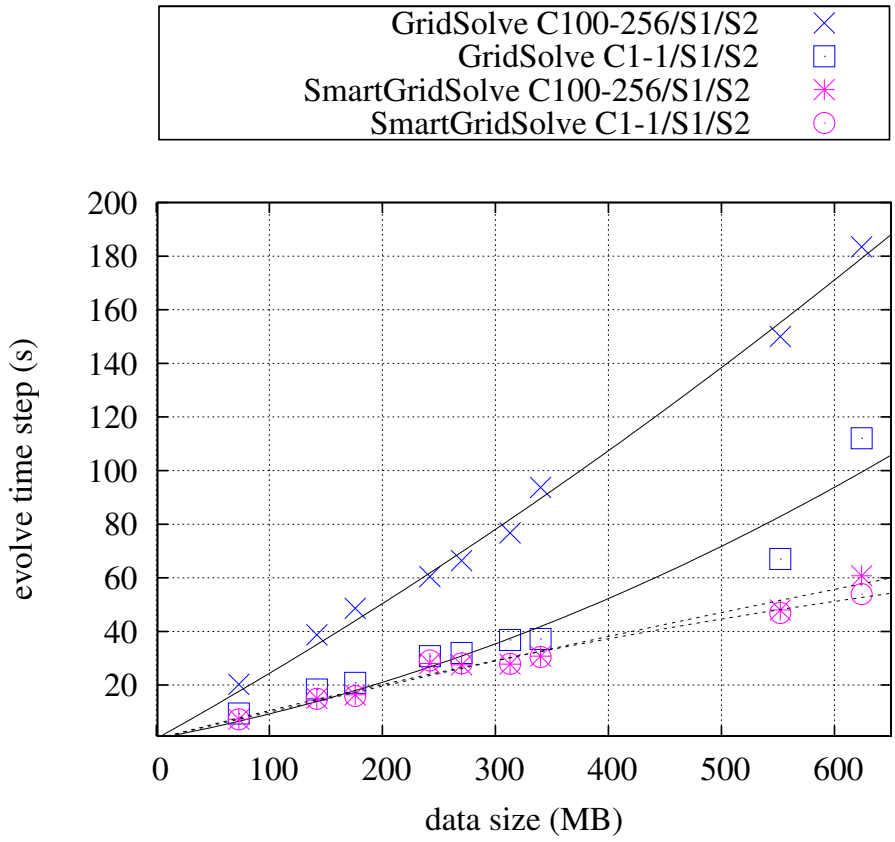



**Figure 6.3:** *Execution time of the evolution step, with varying problem sizes, for the GridSolve and SmartGridSolve versions of Hydropad using clients C1-1 and C100-256*

by the use of a task graph as a performance model and without it the SmartGridSolve middleware would not be able to achieve the discussed improved performances. Therefore, a task graph that wholly represents the underlying algorithm is of primary importance.

An important issue to consider when analysing the performance of Grid applications using the SmartGridRPC model is that a representative task graph may not be always generated for every kind of algorithm by its automatic task graph method. A typical example is when, in the code to be mapped, a conditional construct exists that checks a value that cannot be known without executing a remote task call. A detailed description of this problem, with various examples, and a propose solution are discussed in the following chapters.

# Chapter 7

# The Automatic Task Graph Generation Issue: Irregular Algorithms

In order to optimally minimise the total execution time of an application, the SmartGridRPC model uses the task graph of the group of tasks to be mapped collectively as a part of the performance model. This task graph represents part of the application's algorithm and the full knowledge of all the tasks executed in it. This information is essential to choose the best server to execute a task and to minimise the amount of data movement in a Grid.

The task graph can be generated automatically from the application code by using the SmartGridRPC API. This works by iterating twice through the code that contains the task calls to be mapped collectively. On the first iteration through the code, each task call is discovered but not executed. Then, when the last call in the group of tasks is reached, the task graph is generated. On the second iteration, after producing the mapping by using the new task graph, the code is normally executed and the task calls are performed according to the newly mapping solution.

The automatic construction of the task graph works flawlessly for many regular or static algorithms, i.e. algorithms where the execution is not influenced by the inputs, because the flow of task calls is known at run-time before their execution. Thus, the task graph generated for such an algorithm accurately represents

its run-time execution. This permits a SmartGridRPC middleware to obtain an optimal mapping for this kind of algorithm and therefore to obtain better results than a GridRPC one.

Unfortunately, the automatic approach has the restriction that a representative task graph may not always be automatically generated for every kind of code. The automatic construction of task graphs may not work for irregular or dynamic algorithms, i.e. algorithms where the execution changes depending on the inputs. A typical example is when, in the code, a conditional construct checks a value that cannot be known without executing a remote task call. To apply collective mapping in this case, the application programmer can choose to create task graphs for smaller blocks of code. However, the resulting groups of tasks to be mapped will generate a less optimal execution.

In this chapter, we present three example algorithms that model real-life irregular algorithms that are common in many scientific applications. These algorithms are outlined using three trivial applications implemented in GridRPC and SmartGridRPC. Then, we show the restrictions of the automatic task graph generator for these algorithms and we present some possible solutions.

## 7.1 Examples of Irregular Algorithms

Irregular algorithms, where the flow of task calls change dynamically, are important because they are common in many scientific simulations. In this section, we introduce three trivial examples of irregular algorithms. These examples are models of typical algorithms used to solve real-life problems.

**Iterative Algorithm** An iterative algorithm executes a sequence of computations to approximate a problem solution until the solution reaches a desired accuracy. This algorithm is a general model of so called iterative methods. They are used for solving linear and non-linear algebraic equations that are the base of many numerical simulations. Algorithm 7.1 shows a pseudocode of the example iterative algorithm. This type of algorithm is used in the main evolution loop of the Hydropad application.

**Algorithm 7.1** Iterative

**while** Error $\varepsilon$ is bigger than threshold $t_\varepsilon$ **do**
  Compute Solution
  Compute Error $\varepsilon$
**end while**

**Conditional Algorithm** A common situation in a numerical computation arises when the flow of execution in an algorithm depends on a conditional statement. Algorithm 7.2 shows a pseudocode of the conditional algorithm. One can see that the computation performed at the end of the algorithm depends on the previously calculated error value. This situation can happen in many types of algorithms, even in the iterative methods previously discussed.

**Algorithm 7.2** Conditional

Compute Solution
Compute Error $\varepsilon$
**if** Error $\varepsilon$ is bigger than threshold $t_\varepsilon$ **then**
  Correct Solution by $\varepsilon$
**else**
  Correct Solution by $t_\varepsilon$
**end if**

**Adaptive Algorithm** Typically an algorithm executes its computation on a specific data structure. Some algorithms dynamically change their internal data structure, and consequently their behaviour, depending on the data processed. These algorithms are called adaptive algorithms. The pseudocode of algorithm 7.3 shows a trivial example. In the first step of this example, the solution of the total compu-

**Algorithm 7.3** Adaptive

Compute Solution in Domain $D$
Compute Error $\varepsilon$
Find Sub-domains $S$ of $D$ where $\varepsilon > t_\varepsilon$
**for all** $S^i$ in $S$ **do**
  Compute Solution in Sub-domain $S^i$
**end for**

tational domain is calculated. Then, this domain is divided into many sub-domains by finding the location where the solution error is greater than a threshold. In the next step, the solution is refined by further computation in each new sub-domain. This pseudocode represents a sketch of the AMR (Adaptive Mesh Refinement) method, a real-life example of the adaptive algorithm.

## 7.2 GridRPC and SmartGridRPC implementations of the irregular algorithms

In this section, we present for each irregular algorithm its GridRPC and SmartGridRPC implementations. The SmartGridRPC implementations are then used to show the restrictions of the automatic task graph generation method for these types of algorithms. We also present some programming techniques that help mitigate these restrictions. As previously mentioned, the main evolution loop of the Hydropad application is an iterative algorithm. Thus, the GridRPC and SmartGridRPC implementations and the technique to use the automatic task graph generation for this type of algorithm are already discussed respectively in section 3.3 and section 5.5. This information is further analysed in this section with additional examples.

The following GridRPC examples use *grpc_call* and *grpc_call_async* methods to execute blocking and asynchronous remote calls respectively. Furthermore, the codes use the method *grpc_wait_all* to block the execution until any previously issued asynchronous request has been completed. Additionally, the SmartGridRPC examples use the *grpc_map* method to define a specific area of code. All the task calls contained in this area are mapped as a group of tasks in a fully connected network. These examples utilise also the *grpc_local* method to identify the area of code that contains local computations. The following examples consist of many tasks (*T1*, *T2*, ...), where all the parameters are input objects except the last one that is an output object. The same is applied to the functions (*F1*, *F2*, ...), which represent local computations. The objects (*A*, *B*, ...) used in the examples are all vectors of double precision numbers. All the examples have these characteristics except when indicated differently.

### 7.2.1 Iterative Algorithm

Table 7.1 shows the GridRPC implementation of a trivial application that uses an iterative algorithm. At the beginning, two parallel remote *T1* task calls are executed. These tasks compute a new solution of the objects *A0* and *A1* from inputs *B0* and *B1*. Then, the output objects (*A0* and *A1*) are used as inputs of the remote task *T2*. The output *D* of the latter task is then used as an input to a local function *F1*. The function returns a scalar value *E* that is compared to a threshold value, *tE*. When the returned value is lower than the threshold, the algorithm stops.

**Table 7.1:** *Example of a GridRPC implementation of an iterative algorithm*

```
1 while(E>tE){
2   grpc_call_async(T1_hnd,&id1,A0,B0,A0);
3   grpc_call_async(T1_hnd,&id2,A1,B1,A1);
4   grpc_wait_all();
5   grpc_call(T2_hnd,&id3,A0,A1,D);
6   F1(D,E);
7 }
```

When the application is executed, a GridRPC middleware maps each *grpc_call* and *grpc_call_async* functions individually to a server in the Grid environment. Then, the data is communicated from the client computer to the chosen server and the task is executed remotely. At the end of the task execution, the data is communicated back to the client.

Table 7.2 shows how the SmartGridRPC API can be used in the previous application to map a group of tasks. This example uses the automatic task graph generator to build the task graph. At run-time, when the *grpc_map* method is executed, the code within its parenthesis will be iterated through twice. On the first iteration, both *grpc_call* and *grpc_call_async* calls are discovered but not executed. At the beginning of the second iteration, the task graph and the mapping solution are generated using the task information from the previous discovery. On the second iteration, the task calls are executed through the SmartGridRPC middleware on the respective servers specified by the mapping solution. The block of code defined by *grpc_local* is not executed during the discovery phase, which is

**Table 7.2:** *Example of a SmartGridRPC implementation of an iterative algorithm*

```
1  while(E>tE){
2    grpc_map("ex_map"){
3      for(i=0;i<nloops;i++){
4        grpc_call_async(T1_hnd,&id1,A0,B0,A0);
5        grpc_call_async(T1_hnd,&id2,A1,B1,A1);
6        grpc_wait_all();
7        grpc_call(T2_hnd,&id3,A0,A1,D);
8        grpc_local(D){
9          F1(D,E);
10       }
11     }
12   }
13 }
```

done on the first iteration, but only on the execution phase, which is done on the second iteration. The argument in method *grpc_local* is utilised by the programmer to inform the task graph generator that the object *D* is needed by the client machine. The use of *grpc_local* method in this code is necessary because during the first iteration the object *D* does not contains any valid data since the task *T2* is not executed. Therefore, the local function *F1* could fail or generate wrong results if executed in the first iteration because of the invalid data in *D*.

In table 7.2 it is possible to see that for this example the *grpc_map* method is used inside the while loop of line code 1. A straightforward SmartGridRPC implementation would be to apply the *grpc_map* to the whole loop, letting the middleware find the optimal mapping solution for the group of remote tasks. Unfortunately, the automatic task graph generator will not be able to build a representative task graph for this straightforward implementation because the number of iterative cycles executed is unpredictable during the discovery phase. A solution to this problem is to use the SmartGridRPC function inside the while loop in conjunction with a for loop statement. This implementation prevents the undefined behaviour of the algorithm during the discovery phase. Furthermore, the programmer can choose the number of iterations (*nloops*) to map simultaneously. Although the code in table 7.2 executes more iterations than the code of table 7.1 when there is

convergence and *nloops* is greater than one, the SmartGridSolve execution of the SmartGridRPC implementation will usually outperform the GridSolve execution of the GridRPC one by virtue of the better mapping.

### 7.2.2 Conditional Algorithm

Table 7.3 shows the GridRPC implementation of a trivial application that uses a conditional algorithm. The peculiarity of this application is that the local function *F1* is used in the conditional statement to choose which data objects will be used by the following task *T3*. If the value returned by this local function is greater than a given threshold, objects *A0* and *A1* will be processed, otherwise it will be objects *B0* and *B1*.

**Table 7.3:** *Example of a GridRPC implementation of a conditional algorithm*

```
1  grpc_call_async(T1_hnd,&id1,A0,B0,C0);
2  grpc_call_async(T1_hnd,&id2,A1,B1,C1);
3  grpc_wait_all();
4  grpc_call(T2_hnd,&id3,C0,C1,D);
5  if(F1(D)>tE){
6    grpc_call_async(T3_hnd,&id4,C0,A0,A0);
7    grpc_call_async(T3_hnd,&id5,C1,A1,A1);
8    grpc_wait_all();
9  }
10 else{
11   grpc_call_async(T3_hnd,&id4,C0,B0,B0);
12   grpc_call_async(T3_hnd,&id5,C1,B1,B1);
13   grpc_wait_all();
14 }
```

The ideal task graph, to be used to map this application, would represent the exact run-time execution of the remote tasks. Thus, the ideal location for the *grpc_map* method would be at the beginning of the code to contain all the task calls of the algorithm. As in the case of the previous example, this ideal task graph cannot be generated by the automatic method because the application's execution is uncertain in the conditional statement. A technique, that allows the application

programmers to still avail themselves of the SmartGridRPC group mapping, is to break the whole code into smaller blocks suitable for the automatic task graph generation as shown in table 7.4. This solution however produces many small groups of tasks. Therefore the SmartGridRPC middleware will minimise the execution time of these small groups rather than the whole algorithm, thus producing less optimal mapping. Additionally, the data objects used between groups instead of being communicated directly between servers, will be communicated through the client machine. Figure 7.1 on page 86 shows the three task graphs generated from the example of table 7.4. This graph illustrates all the data objects that have to pass through the client machine before and after the conditional statement and shows all the data dependencies between the tasks executed that are missed by the code in table 7.4.

**Table 7.4:** *Example of a SmartGridRPC implementation of a conditional algorithm*

```
1  grpc_map("ex_map"){
2    grpc_call_async(T1_hnd,&id1,A0,B0,C0);
3    grpc_call_async(T1_hnd,&id2,A1,B1,C1);
4    grpc_wait_all();
5    grpc_call(T2_hnd,&id3,C0,C1,D);
6  }
7  if(F1(D)>tE) {
8    grpc_map("ex_map"){
9      grpc_call_async(T3_hnd,&id4,C0,A0,A0);
10     grpc_call_async(T3_hnd,&id5,C1,A1,A1);
11     grpc_wait_all();
12   }
13 }
14 else{
15   grpc_map("ex_map"){
16     grpc_call_async(T3_hnd,&id4,C0,B0,B0);
17     grpc_call_async(T3_hnd,&id5,C1,B1,B1);
18     grpc_wait_all();
19   }
20 }
```

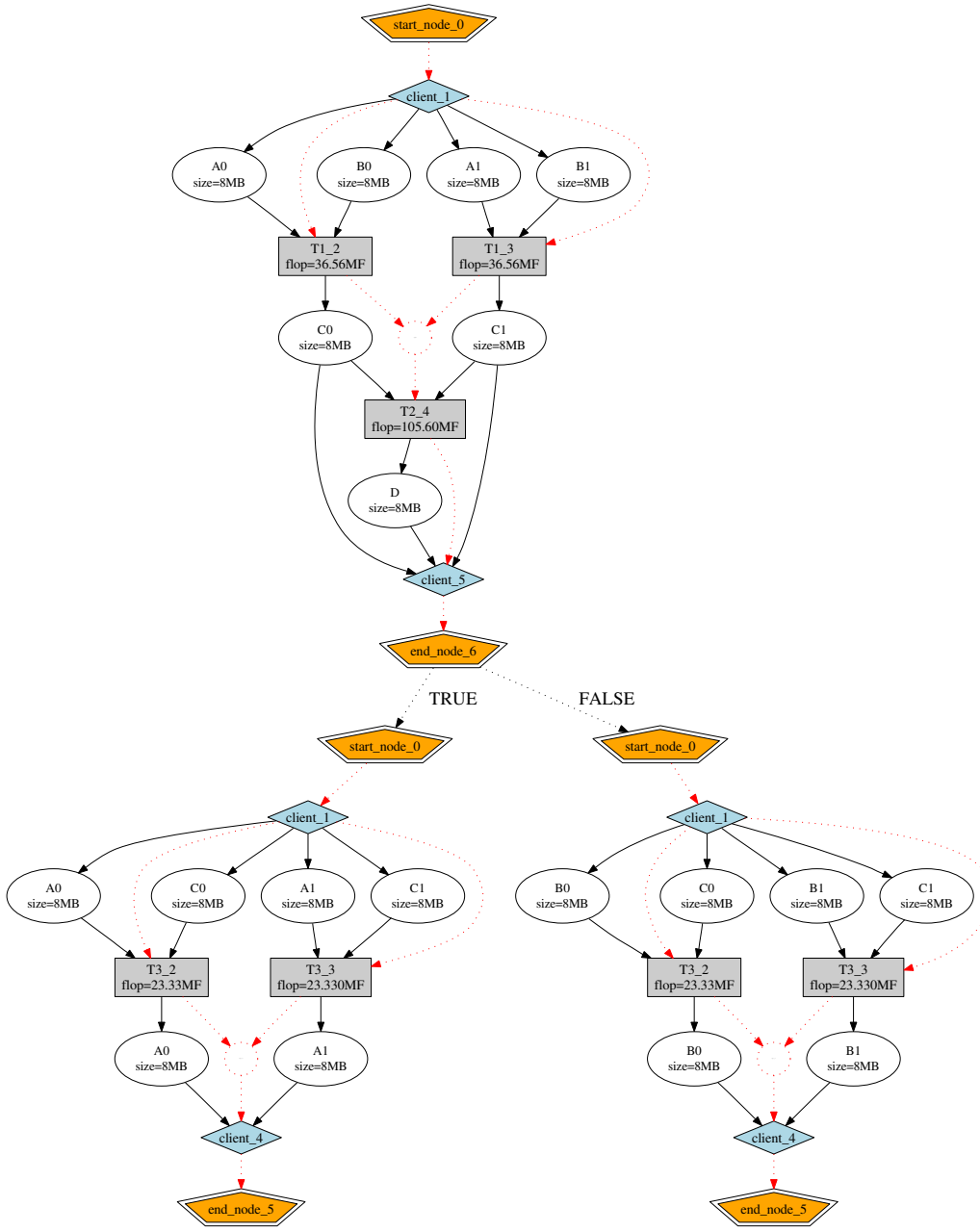



**Figure 7.1:** *Three task graphs generated from the SmartGridRPC implementation of the conditional algorithm example*

### 7.2.3 Adaptive Algorithm

Table 7.5 shows the GridRPC implementation of a trivial application that uses an adaptive algorithm. The task *T1* calculates the solution *C*. Then this data, together with the threshold *tE*, are passed to the task *T4*. Task *T4* checks the solution *C* and outputs a vector of areas, *S*, and the number of such areas, *n*. The areas in the vector *S* represent the location where the solution error is greater than the threshold *tE*. In the next step, task *T5* outputs two sub-vectors, *AS* and *BS*. These sub-vectors are an interpolation, of higher resolution, of the main vectors (*A*, *B*) for each area in *S* previously found. The sub-vectors are then used to calculate a more accurate solution (*CS*) through task *T1*.

**Table 7.5:** *Example of a GridRPC implementation of an adaptive algorithm*

```
1  grpc_call(T1_hnd,&id1,A,B,C);
2  grpc_call(T4_hnd,&id2,C,tE,S,n);
3  for(int i=0;i<n;i++){
4    grpc_call_async(T5_hnd,&id3,i,S,A,B,AS[i],BS[i]);
5    grpc_wait_all();
6  }
7  for(int i=0;i<n;i++){
8    grpc_call_async(T1_hnd,&id4,AS[i],BS[i],CS[i]);
9    grpc_wait_all();
10 }
```

In this example, the outputs of task *T4* not only change the flow of execution of the algorithm but also change the sizes of the objects computed by the following tasks. This is one of the worst case scenarios for the automatic task graph generation. On the discovery phase, the data objects *n* and *S* are unknown. Therefore, not only will the flow of execution be unpredictable but also the data objects' sizes. As in the previous example, a solution is to apply *grpc_map* to smaller blocks of code, as shown in table 7.6.

Figure 7.2 on page 89 shows the two task graphs generated from the Smart-GridRPC example code when the output of *n* is three and thus there are three sub-vectors. The main problem of having two task graphs instead of a fully complete

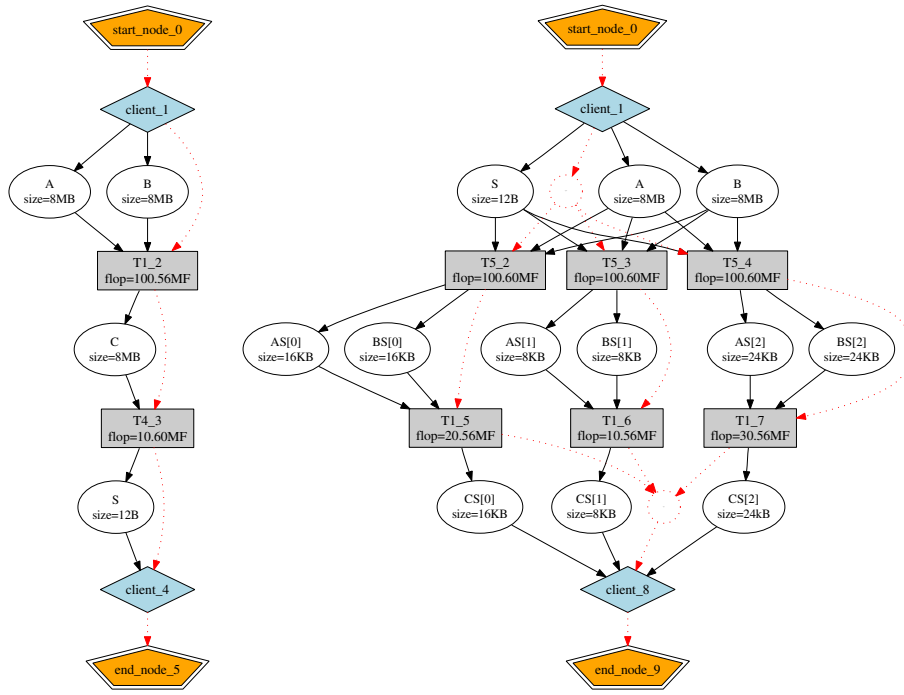
**Table 7.6:** *Example of a SmartGridRPC implementation of an adaptive algorithm*

```
1  grpc_map("ex_map"){
2    grpc_call(T1_hnd,&id1,A,B,C);
3    grpc_call(T4_hnd,&id2,C,tE,S,n);
4  }
5  grpc_map("ex_map"){
6    for(int i=0;i<n;i++){
7      grpc_call_async(T5_hnd,&id3,i,S,A,B,AS[i],BS[i]);
8      grpc_wait_all();
9    }
10   for(int i=0;i<n;i++){
11     grpc_call_async(T1_hnd,&id4,AS[i],BS[i],CS[i]);
12     grpc_wait_all();
13   }
14 }
```

one in this case is that, despite the second task graph in figure 7.2(b) containing what appears to be a substantially representative task graph, the communication of objects *A*, *B* and *S* is delayed. In fact these objects are communicated from the client to remote servers just before the parallel execution of tasks T1 (code line 11) instead of being communicated on the respective servers during the first *T1* task call (code line 2). Considering that the time spent to communicate the main objects from the client could be higher than the time spent to compute the whole second group of tasks (since the sub-vectors are possibly much smaller than the main objects and thus the respective task computation and data communication times), the performance loss caused by the use of two separate task graphs could be great.

## 7.3 Summary

In this chapter, we have shown how the automatic task graph generation method included in the SmartGridRPC model may not work in the case of irregular algorithms; i.e. algorithms where the flow of execution changes depending on values previously calculated by remote tasks. We have shown that this problem can be




(a) First *grpc_map* (b) Second *grpc_map*

**Figure 7.2:** *Two task graphs generated from the SmartGridRPC implementation of the adaptive algorithm example*

partially solved with some modification of the code of the application (table 7.2) or by mapping smaller code blocks (tables 7.4 and 7.6) that enclose only the static parts of the algorithms and thus avoid the dynamic parts.

However, the various fragmented task graphs generated by the partial mapping of code does not represent the underlying algorithm completely, therefore information needed to improve the the data communication and tasks computation of the application may be missing (for example, information such as possible data dependencies between the fragmented task graphs and possible parallel executions of tasks). The lack of this information could force the SmartGridRPC middleware to communicate data objects through the client machine instead of directly between servers. Additionally, the middleware could lack the knowledge to anticipate which communication could waste time. Finally, if the middleware scheduler is missing information about the execution of parallel tasks it could generate a computational unbalanced mapping solution. Therefore, a SmartGridRPC

middleware that use fragmented task graphs instead of a complete representative task graph may not obtain the best performance possible for the underlying algorithm and thus the application.

A possible solution would be to enhance the automatic task graph generation technique using additional specific functions or preprocessing tools to retrieve information about loops and conditions that change the flow of task calls. These supplementary techniques would be added directly in the code by the application programmer. However, these two approaches have some drawbacks, such as the difficulty of implementation and, in the case of additional functions, a greater performance penalty at run-time. Furthermore, a second problem is that the programmer would need to significantly change and clutter the client code in order to highlight all the possible changes of flow and execution.

A better solution to this problem, which we propose, is to use a specific high level language that permits the application programmer to explicitly specify a task graph that best represents the run-time execution of the irregular algorithm. Since the application programmer usually has a in depth knowledge of the algorithm used inside his application, he or she can generate the most representative task graph in the situation where the output of a remote task call can change the flow of execution. Therefore, a SmartGridRPC middleware that would use an explicitly generated representative task graph could achieve a faster execution of the application than a SmartGridRPC middleware that uses partial task graphs generated from smaller code blocks or a GridRPC middleware that uses the individual task mapping method.

A typical code of this specific language should contain all the necessary information needed to generate a complete task graph. Furthermore, the language should be easy enough to permit a user to easily define a simple algorithm but be expressive enough to permit the definition of a complicated algorithm. The language and its compiler should have these objectives: (i) to be easy to learn and to understand; (ii) to provide a way to change the sizes of tasks' objects from the client code and to calculate the relative execution time of the tasks; (iii) to specify possible changes in the flow of task calls and permit a user to select the flow dynamically from the client code. (iv) to allow highlighting of the tasks that can be executed asynchronously; (v) to permit a user to easily identify input and

output objects of a task; and (vi) to provide a technique to set an eventual client computation and the data objects used on it.

# Chapter 8

# Algorithm Definition Language: Generation of Explicit Task Graphs for Irregular Algorithms

As discussed in the previous chapter, the method that we propose to solve the problem of representative task graph generation for irregular algorithms is to utilise a high level language that permits a user to explicitly define task graphs. The idea of using a specific language is similar to and based on the concept presented in the mpC [61] and HeteroMPI [62] projects for heterogeneous parallel systems. These high level languages are used not only to generate the application code but also to generate the application performance model.

The existing languages in Grid computing that are used to explicitly generate task graphs or workflows are not expressive enough or they lack some essential features to solve the problem of irregular algorithms (see section 8.4). In this work, we have designed and implemented a language that permits a user to easily define a representative task graph for any kind of algorithm: Algorithm Definition Language (ADL) [54]. The main goal of ADL is to give a powerful tool to the application programmer that can help him or her in implementing a SmartGridRPC application with the best mapping and execution possible.

The ADL language is powerful enough to be able to fully describe a GridRPC algorithm and to provide a way to calculate the computational and communication

time of the application, yet flexible enough to generate the task graph for any type of algorithm. One of the objectives of ADL is to be user-friendly and easy to write. The ADL syntax is similar to the C language since this is a widely used language for computational science. The principal programming unit of this language is a module that is used to specify the group of tasks of an individual algorithm. Furthermore, the language is divided into well-defined sections that specify distinct parts of the algorithm and thus simplify the understanding of the algorithm and of the code. The ADL compiler does not directly produce the task graph from the module but generates C code that is used at run-time by the *grpc_map* method in the client application to build the task graph. In the next sections, we show how the application programmer can use ADL to originate a representative task graph for the conditional and adaptive example algorithms. Furthermore, we present a brief description of the language syntax and how ADL is integrated into the SmartGridRPC model and thus how it is possible to generate a task graph in the client code. In the last section, we presents the results of experiments with the example applications showing that the use of ADL significantly improves their performance.

## 8.1 Conditional Algorithm Using ADL

Table 8.1 on page 94 shows the ADL module that describes the algorithm of our example conditional application (see table 7.3 on page 84 to compare with the GridRPC implementation code). An ADL module is composed of the keyword *module* followed by the name, the list of parameters and the module body. The body is divided into the following sections. The *component* section includes a declaration of the tasks used in the algorithm, such as *T1*, *T2* and *T3* for the example application. The *IFO* section contains a declaration of data objects. In ADL, the data objects, that are used in a task and can be moved anywhere on the Grid, are called Identified Flying Objects (IFOs). Their declaration is composed of the type (in upper-case letters to differ from a variable type), the number of dimensions and the list of IFO names. The list of types to choose for an IFO is correlated to types used in the GridRPC API. The number of round bracket pairs, located after the type, represent the number of dimensions of an IFO. In the ex-

**Table 8.1:** *Example of an ADL module of the conditional algorithm*

```
1  module cndalg(int size, int cndtrue, int cndfalse)
2    {
3    component:
4      task "cond.idl"          T1,T2,T3;
5
6    IFO:
7      DOUBLE(size)           A[2],B[2],C[2],D;
8
9    algorithm:
10     parfor(int i=0;i<2;i++){
11       T1:(A[i],B[i])->(C[i]);
12     }
13
14     T2:(C[0],C[1])->(D);
15
16     client:(D)->();
17
18     parallel{
19       if(cndtrue)
20         parfor(int i=0;i<2;i++)
21           T3:(C[i],A[i])->(A[i]);
22
23       if(cndfalse)
24         parfor(int i=0;i<2;i++)
25           T3:(C[i],B[i])->(B[i]);
26     }
27
28   inout:
29   }
```

ample application, the IFO arrays (*A*, *B* and *C*) and the single IFO (*D*) are vectors of double precision numbers. The size of these vectors depends on the value of the parameter *size*. Finally, the *algorithm* section describes the flow of execution of the application and the *inout* section defines the input and output IFOs of the module, in this particular case there are none.

Table 8.1 shows how a task call is described in ADL. A remote call is composed of two parts, divided by a semicolon. The first part is the name of the task called (e.g. *T1*). The second part is the list of IFOs used as task inputs (e.g. *A* and *B* for task *T1*) followed by an arrow symbol and the list of output IFOs (e.g. *C*). This task call syntax is made in a way that easily highlights the parameters passed and the IFOs used as inputs and outputs of the task. The *parfor* construct specifies that the task calls between the iterations of the loop are asynchronous while the task calls inside the same iteration are sequential. The *parallel* construct indicates that all the included statements are considered asynchronous. One of the main differences between the ADL code and the application code is the use of the special keyword *client*, as a task name, to specify any local execution. For the purpose of task graph generation, ADL does not need to know which local computations will be done and which local data will be used in these computations. The only information needed is which IFOs are used in these computations (but not their values). Thus, in the case of a local computation, ADL requires only the information about the IFOs used as inputs and outputs of this computation. Therefore, in table 8.1 the client task has only *D* as an input and no output, and the name of the client task, *F1*, is not included in the specification.

The straightforward description of the conditional algorithm would be with an if-then-else statement, as it is in the original SmartGridRPC code in table 7.4. Instead, in the ADL example in table 8.1, the module contains two conditional statements, in lines 19 and 23, that check the value of the parameters *cndtrue* and *cndfalse*. The application programmer, by setting only one of these two values to true, can choose the flow of execution that is most likely to happen. Furthermore, the programmer can choose to generate and use a task graph that contains both branches of the execution by setting the two parameters to be true simultaneously. The parallel construct in line 18 is used for this option. Without it, the compiler will consider the two branches as being executed sequentially, one after the other.

The setting of different parameter values allows the application programmer to easily choose the most representative task graph from a set of different possible task graphs. Furthermore, the parameters in ADL are not only used to determine the control flow of the algorithm but also to specify the size and the number of IFOs utilised in the module. An IFO cannot change its size after the declaration, consequently all the parameters are considered constant in the ADL language.

**Table 8.2:** *Example of ADL use in the conditional algorithm application through the SmartGridRPC method*

```
1  grpc_map("ex_map",ADL,cndalg,"%d,%d,%d",size,1,1){
2    grpc_call_async(T1_hnd,&id1,A0,B0,C0);
3    grpc_call_async(T1_hnd,&id2,A1,B1,C1);
4    ...
5    if(F1(D)>tE){
6      ...
7    }
8    else{
9      ...
10   }
11 }
```

Table 8.2 shows how to use the *grpc_map* method with ADL to build the task graph. The first argument of the function is the string that specifies the mapping heuristic and it is the same as in any other SmartGridRPC example, like the one of the conditional algorithm in code of table 7.4 on page 85. The second argument is the keyword *ADL*. This specifies that the task graph will be built by using the code generated from the ADL module named in the following argument (e.g. cndalg). The next argument is a string that contains the quantity and the type of parameters passed to the module. The format is similar to the *printf* function call of the C language. The final arguments in the *grpc_map* method are the values or variables assigned to the parameters of the given ADL module. In table 8.2, the application programmer has decided that both conditional statements are true. The run-time execution of the *grpc_map* function is different from the case of the automatic method. The task graph is built and the mapping solution is generated directly

when the method is called. Therefore, the code inside the parenthesis block will be iterated only once and the task calls are executed normally on the servers specified in the mapping solution. Thus, the use of the SmartGridRPC function *grpc_local* is not needed when using ADL.

The task graph generated from the ADL code of table 8.1 together with the SmartGridRPC code of table 8.2 is illustrated in figure 8.1 on page 98. The values inside the circles and rectangles are respectively the size of an IFO and the computational complexity of a task. These are correlated to the value of the module parameter *size* passed through the third to last argument of the *grpc_map* method. One can see that the generated task graph contains the task calls for both possible flows of execution of the conditional statement. This information permits the SmartGridRPC mapper to map efficiently the different versions of tasks *T3* in order to minimise the total data communication, such as tasks *T3_6* and *T3_8* on the same server of task *T1_2* while tasks *T3_9* and *T3_9* are on the same server of task *T1_3*. If we compare the task graph generated from the ADL code with the three task graphs of figure 7.1 on page 86, generated by using the *grpc_map* method for smaller blocks of code (table 7.4), it is possible to see that the input arrays *A* and *B* can be sent directly to the servers where the various versions of tasks *T3* are going to be executed. Furthermore, the output array *C* can be broadcasted directly as well to the same servers. Due to better mapping and various direct and broadcasted communications of data, the SmartGridRPC implementation of the conditional example that use the ADL code can obtain better performance than the SmartGridRPC implementation without the use of ADL.

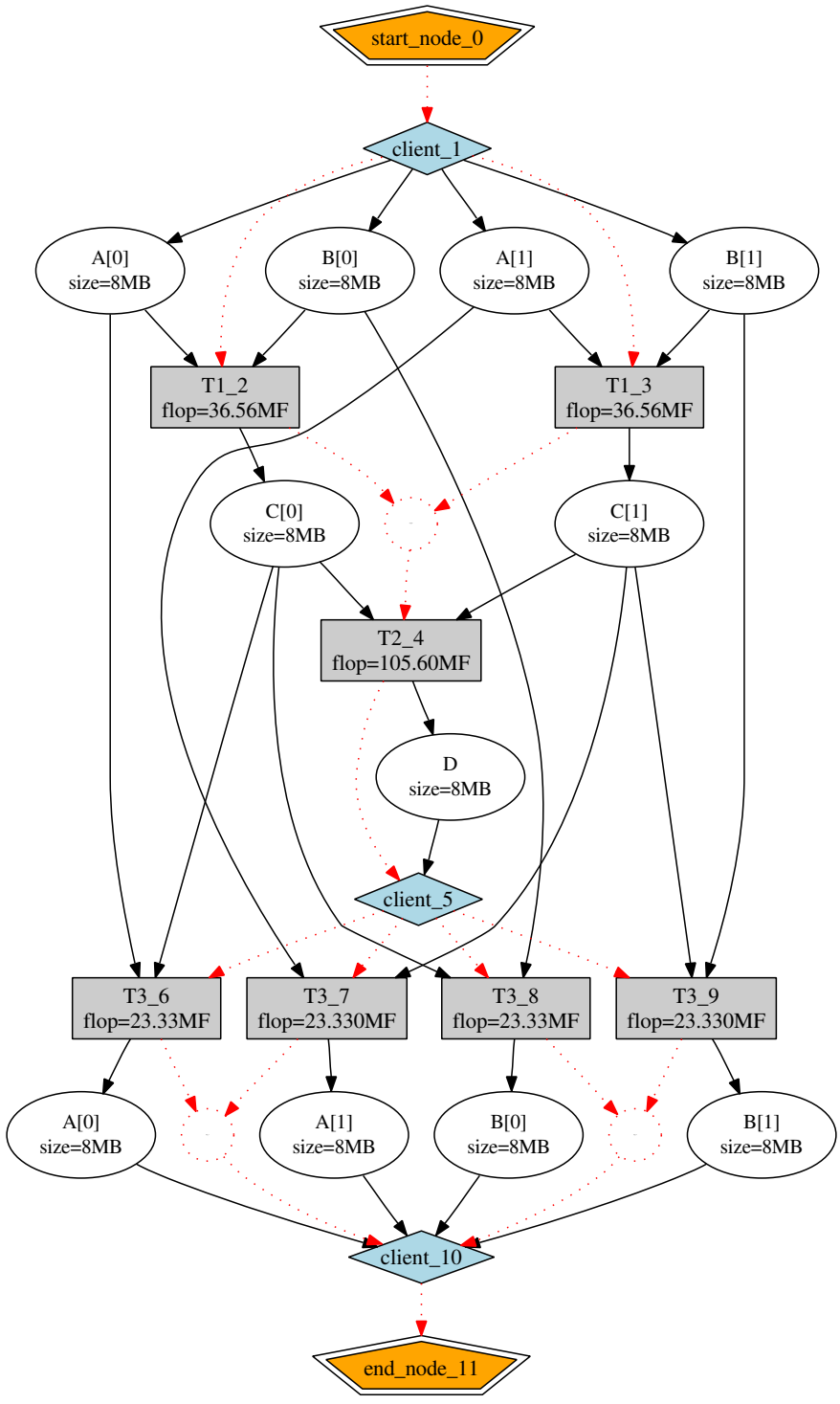



**Figure 8.1:** *The task graph generated from the ADL module of the conditional algorithm example*

## 8.2 Adaptive Algorithm Using ADL

Table 8.3 shows the ADL module that describes the algorithm of our example adaptive application. One can see that the sizes of data objects *A*, *B*, *C*, depend on the parameter *size*. The parameter *n* is used to determine the number of areas in vector *S* that could be generated by the task *T4*. Furthermore, the value of *n* is used to set the number of times that task *T5* is executed and hence the number of sub-vectors (*AS*, *BS*, *CS*) that are generated. The parameter *subsize* is an array of integers that contains the sizes of each sub-vector object. The dimension of this array depends on the value of *n*.

**Table 8.3:** *ADL module of the adaptive algorithm example*

```
1  module adaptalg(int n, int size, int subsize[n])
2    {
3    component:
4      task "adapt.idl"       T1,T4,T5;
5
6    IFO:
7      DOUBLE(size)          A,B,C;
8      DOUBLE(subsize)       AS[n],BS[n],CS[n];
9      INTEGER(n)             S;
10
11   algorithm:
12     T1:(A,B)->(C);
13
14     T4:(C)->(S);
15
16     parfor(int i=0;i<n;i++){
17       T5:(S,A,B)->(AS[i],BS[i]);
18     }
19
20     parfor(int i=0;i<n;i++){
21       T1:(AS[i],BS[i])->(CS[i]);
22     }
23 }
```

**Table 8.4:** *Example of ADL use in the adaptive algorithm application through the SmartGridRPC method*

```
grpc_map("ex_map",ADL,adaptalg,"3,1000,{200,100,300}"){
 grpc_call(T1_hnd,&id1,A,B,C);
 grpc_call(T4_hnd,&id2,C,tE,S,n);
  for(int i=0;i<n;i++){
   grpc_call_async(T5_hnd,&id3,i,S,A,B,AS[i],BS[i]);
   grpc_call_async(T1_hnd,&id4,AS[i],BS[i],CS[i]);
   grpc_wait_all();
}
```

The application programmer, by setting the values of the three parameters in *grpc_map*, can directly specify the number of task calls and the sizes of objects in the task graph generated. Table 8.4 shows the use of the ADL module *adaptalg* in the modified adaptive SmartGridSolve application. In this case, the application programmer presumes that the remote computation of task *T4* produces $n = 3$ and thus the programmer chooses to generate a task graph with three *T5* calls. The size of data objects will be 1000 and the sub-vectors' sizes will be 200, 100 and 300. The method used to set the parameter values in this example, instead of being a string formatted like the *printf* method of the code in table 8.2, is a string directly containing the values for the parameters. Furthermore, the use of curly brackets permits a user to group different values for a parameter array, in this case the parameter *subsize*. Figure 8.2 shows the generated task graph from the SmartGridRPC code of table 8.4 in conjunction with the ADL code of table 8.3. The comparison between this task graph with the two partial task graphs of figure 7.2 on page 89 shows that the objects *A* and *B* can be communicated directly to the servers that execute the different versions of task *T5*.

In this case, it is difficult to generate a task graph that corresponds exactly to the flow of task calls since the value of *n* probably changes for each execution. There are two possible outcome for this and similar situations; the user generates a task graph that is a super-set of the task executions or a task graph that is a sub-set. In the later case SmartGridSolve handles this situation, i.e. task being executed but not existing in the mapping solution, by reverting the execution method of

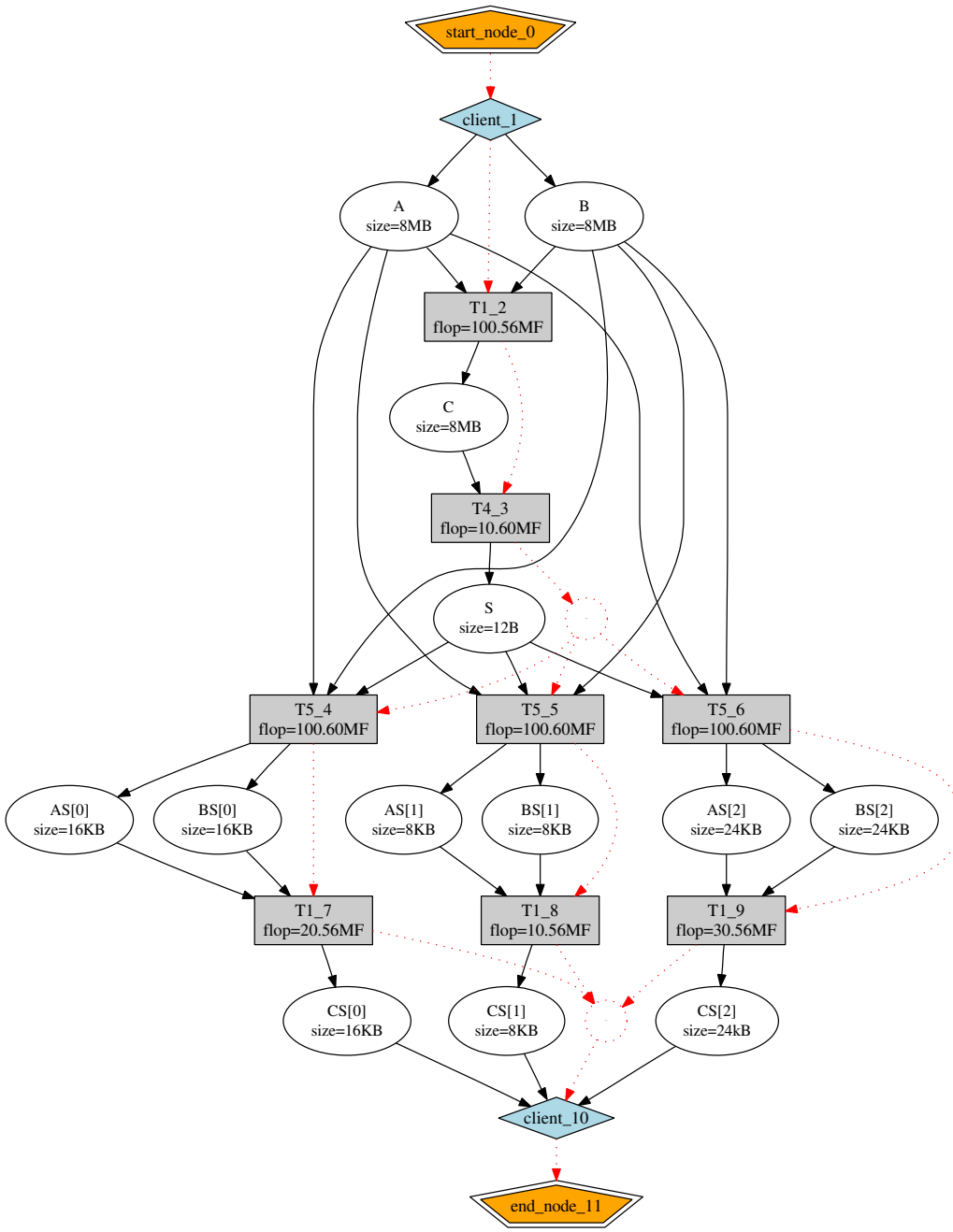



**Figure 8.2:** *The task graph generated from the ADL module of the adaptive algorithm example*

the missing remote tasks to the original GridSolve one. This involves more data communication through the client machine for tasks using the GridRPC model and probably a minor load computation unbalance. In the case of a super-set, i.e the task graph contains all the possible task calls, SmartGridSolve moves the objects needed by a task also if this task is not executed. These unused objects are stored in the server until the method *grpc_map* completes the execution phase. This involves a small waste of resources in the Grid environment, such as network and memory. The user can choose one of the two options that is similar to the real execution, thus reducing these small disadvantages in a way that they become insignificant in comparison to the the gains obtained by having a bigger group of tasks to map.

## 8.3 Experimental Results

In this section, we compare the execution times, for both conditional and adaptive example algorithms, of the three different implementations; the GridRPC version (tables 7.3 and 7.5), the SmartGridRPC with smaller mapping blocks version (tables 7.4 and 7.6) and the SmartGridRPC with ADL version (tables 8.2 and 8.4). The first implementation is executed through the GridSolve middleware while the last two are executed through the SmartGridSolve middleware. The hardware configuration used in the experiments consists of fives machines; a client and four remote servers. The four servers are heterogeneous however they have similar performance, from 422 to 531 MFlops, and the same size of main memory, 1GB each. The bandwidth of the communication links between servers is 1Gb/s. The client machine has a 100Mb/s connection to the servers. This represents a common situation where a user wants to use a powerful Grid environment through a relatively slow network connection. In the experiments, we vary the total input data size, $N$, from 24 to 576 megabytes. Each remote task executed has a log-linear complexity, $O(N \times \ln N)$. This complexity, with the slow client-to-server connection, permits the computation load and the communication load to have an equal impact on the total execution time of the experiments. In the following tables, the symbol $S_p$ stands for speed-up; while GS is the abbreviation for GridSolve and SGS is the one for SmartGridSolve with smaller mapping blocks.

**Table 8.5:** *Experimental results for the conditional algorithm applications*

| | GridSolve | SmartGridSolve smaller blocks | SmartGridSolve with ADL | |
|---|---|---|---|---|
| Data Size | Avg Time | Avg Time | Avg Time | |
| 24MB | 24.08s | 19.31s | 14.08s | |
| 48MB | 48.30s | 40.35s | 26.94s | |
| 96MB | 97.51s | 81.00s | 57.63s | |
| 192MB | 195.80s | 156.67s | 113.67s | |
| 384MB | 404.31s | 317.36s | 230.03s | |
| 576MB | 648.02s | 497.31s | 364.62s | |
| Data Size | | $S_p$ v GS | $S_p$ v GS | $S_p$ v SGS |
| 24MB | | 1.25 | 1.71 | 1.37 |
| 48MB | | 1.20 | 1.79 | 1.50 |
| 96MB | | 1.20 | 1.61 | 1.41 |
| 192MB | | 1.25 | 1.72 | 1.38 |
| 384MB | | 1.27 | 1.76 | 1.38 |
| 576MB | | 1.30 | 1.78 | 1.36 |

**Conditional Algorithm** Table 8.5 shows the results obtained by the GridSolve and SmartGridSolve executions of the three different implementations of the conditional algorithm. For each individual experiment, the average time is calculated from ten separate executions, where the condition of the conditional statement is set to return true in half of them. In the SmartGridSolve with ADL experiments, the task graph generated by ADL contains both branches of the execution. One can see that the SmartGridSolve implementation, with smaller blocks to map, is faster than the simple GridSolve implementation, showing the speed-up of approximately 1.2. Furthermore, the SmartGridSolve implementation, that uses a representative task graph generated from ADL, outperforms the other two implementations, displaying the speed-up of approximately 1.7 and 1.4 respectively.

**Adaptive Algorithm** Table 8.6 shows the results of experiments with three different implementations of the adaptive algorithm. As in the previous experiments, the average time is calculated from ten separate executions. The number of subvectors, $n$, and their sizes, $S$, generated by task *T4*, are constant between experiments of the same initial data size but change randomly between experiments with

different data sizes. The maximum number of sub-vectors is set to four and the sum of their sizes is set to be less than the size of the original input vectors (*A*, *B*, *C*). In the SmartGridSolve with ADL experiments, the task graph generated by ADL is set to contain four sub-vectors which sizes are one quarter of the original vector's size. Therefore, this task graph is a super-set of the possible executions since it contains the maximum number of remote tasks possible.

**Table 8.6:** *Experimental results for the adaptive algorithm applications*

| | GridSolve | SmartGridSolve smaller blocks | SmartGridSolve with ADL | |
|---|---|---|---|---|
| Data Size | Avg Time | Avg Time | Avg Time | |
| 24MB | 28.03s | 25.02s | 17.41s | |
| 48MB | 51.26s | 46.70s | 30.40s | |
| 96MB | 112.53s | 86.76s | 58.40s | |
| 192MB | 216.99s | 182.15s | 118.70s | |
| 384MB | 435.56s | 369.25s | 269.65s | |
| 576MB | 713.78s | 604.23s | 445.75s | |
| Data Size | | $S_p$ v GS | $S_p$ v GS | $S_p$ v SGS |
| 24MB | | 1.12 | 1.61 | 1.44 |
| 48MB | | 1.10 | 1.69 | 1.54 |
| 96MB | | 1.30 | 1.93 | 1.49 |
| 192MB | | 1.19 | 1.83 | 1.53 |
| 384MB | | 1.18 | 1.62 | 1.37 |
| 576MB | | 1.18 | 1.60 | 1.36 |

The speed-up demonstrated by the SmartGridSolve implementation (that uses smaller blocks to map) over the GridSolve execution is less than in the previous experiments, with the implementations of the conditional algorithm. The reason is that the tasks in the second mapping block of this application (see table 7.6) are less computationally intensive than the tasks in the first mapping block. The sub-vectors are smaller than the original vectors. Therefore, the improved SmartGrid-Solve mapping of the parallel tasks in the second block is limited by the execution time of the tasks in the first block. At the same time, the speed-up obtained by the SmartGridSolve implementation using ADL over the other two implementations is similar to the conditional algorithm experiments. The reason is that the use of

ADL permits SmartGridSolve to map a larger group of tasks and to minimise the amount of data moved in the network.

## 8.4 Related Work

The need to directly define, for performance or direct execution purposes, a construct that precisely represents the underlying algorithm of an application, or the application itself, is common in many different fields of computational science. Therefore, various software, tools and languages exist that permit a user to generate a representative construct of an application's algorithm. However, in this chapter we focus on the different languages already existing that are used to generate task graphs or similar structures for systems in the Grid computing field. Furthermore, we analyse the languages specifically designed for GridRPC middlewares.

### 8.4.1 Languages Used in Workflow Management Systems for Grid Computing

In the area of workflow management systems for Grid computing there has been a huge amount of research and implementation of languages designed to explicitly generate task graph like structures. A workflow management system provides various tools, components, middlewares and applications to define, manage and execute scientific workflows on a Grid environment [85]. A workflow is a description of a sequence of operations and procedures with data and files synchronisations that define the execution of an application.

The concept of workflow is similar to that of a task graph. However it has a broader meaning since, instead of representing only a group of tasks, the workflow can represent a group of programs, jobs, services, tasks and activities. Additionally, in the workflow concept the notion of client computation is missing, since a workflow is usually a complete distributed application itself and workflow management systems do not have a client application. However, the structures used to represent workflows are usually the same structures used to represent task graphs.

Therefore, it is important to analyse and compare the various languages used to define workflows in these systems.

Considering that many workflow management systems are evolutions of batch management systems for Grid computing and it was common in these systems to use script languages to execute remote computations; the simplest method used by workflow management systems to generate a workflow is a script-based language. However, the expressiveness of these languages is limited since they are originally designed for the execution of distributed jobs.

More recent workflow management systems, designed specifically for Grid computing, use XML-based languages to take advantage of the extensibility and portability of XML. Furthermore, these systems often use XML also to define the jobs and services used. Some examples of workflow systems that utilise a XML-based language are GridFlow [21], Taverna [70], Triana [30], GridAnt [8], etc. Usually tags, elements and attributes of these XML-based languages vary between systems. However, all these languages suffer from the difficulty to express large and complicated workflows in XML, since all the information needs to be explicitly written when using a markup language. In some cases, such as GridAnt, in order to avoid the expressiveness limitation of a XML-based language, specific conditional or loop statements are added to the language. In some other cases, such as Taverna and Triana, a graphical user interface is implemented to help the user to visually generate the XML file and therefore the workflow. However, this visual help may not always be useful when the execution contains many jobs with a high amount of data dependency between them, since this situation could produce an extremely complicated workflow of which the visualisation and editing is difficult.

Finally, another common method in workflow management systems is the use of specifically designed high level languages. These languages can be based on common imperative languages (C, Fortran, etc.), or be data driven languages, i.e. the generation of the workflow depends on data dependencies. In the following part of this section, we present some significant languages used in workflow management systems to explicitly generate the task graph, where these languages represent indicative cases of script-based languages (DAGMan), XML-based languages (AGWL), imperative high level languages (YvetteML) and data driven

high level languages (Chimera VDL).

### DAGMan

In the Condor batch management system exists a specific meta-scheduler component called DAGMan [82] that permits a user to submit various jobs as a workflow (DAG structure) to the Condor scheduler. The user specifies the workflow using a specific language. This language code is parsed by DAGMan to generate the workflow. In the code the various remote jobs and the direct dependencies between these jobs are described. These dependencies have to be directly indicated by a special keyword by the programmer and they do not indicate any data communication but only the order of execution. Consequently the respective data communications have to be defined through the use of pre-processing and post-processing commands associated with each job. The language used is similar to a script language, however it does not contain any control flow constructs, such as conditional and loop statements. Furthermore, the ability to define and use variables in the language is minimal and the flow of execution is fixed. It is not easy to generate large workflows using the DAGMan language since its expressiveness is very limited.

### Abstract Grid Workflow Language (AGWL)

AGWL (Abstract Grid Workflow Language) [44] is an XML-based language used to generate workflow for scientific Grid applications and it is part of the Grid application development and computing environment Askalon [43]. The main aim of AGWL is to hide the details and complexities of the underlying Grid infrastructure to the programmer and to allow the easy generation of scientific workflows. An important characteristic of AGWL is the ability to define complex workflows, since the language contains basic control flow constructs, such as conditional and loop statements, and advance control flow constructs, such as parallel sections and parallel loops. Furthermore, AGWL permits a user to specify the data computed in the various services and thus to control the data flow. The data flow constructs and control constructs are defined in the language by using specific starting and ending tag pairs. For example, `<while> </while>`, `<if> </if>`, `<dataIn> </dataIn>`,

<value> </value>. Furthermore, each attribute or parameter of these constructs has to be preceded by its name, such as *name="name"*, *source="source"*. While the expressiveness of this language is far superior to a typical XML-based language, in fact it is similar to that of a high level language, the need to directly specify each tag pair and attribute in the workflow make the code writing tedious, long and error prone. Furthermore, the readability of the code suffers. Another limitation of AGWL is that the expressions in the code are specified using XSL transformations [3]. XSLT is used to transform a XML document into another XML one. In fact the AGWL code has to be pre-processed into a concrete XML code, which contains all the data of the executed workflow. The use of a concrete XML to specify the workflow limits the ability to dynamically change the execution flow during run-time. Even if the AGWL code could be used directly at run-time to have the ability to change the flow dynamically, the use of XSL transformations could have a great performance impact since this technique is specifically designed to process XML documents and not workflow structures.

**YvetteML**

YML [36] is a workflow management system designed to provide scientific users with a simple way to develop and execute applications on peer-to-peer and Grid middlewares over large scale environments. It is mainly aimed at parallel and distributed numerical applications, specifically linear algebra, and solving problems that need a large amount of data. However, YML can be used for any type of application. To hide the complexity of the underlying infrastructure from the user, YML is logically divided into two layers [28]; a front end, which is composed of user tools for developing and managing services and workflows, and a back end, which schedules and executes the workflow on the middleware. This logical structure permits YML to execute the workflow on multiple middlewares such as XtremWeb [45] and OmniRPC [72]. A workflow in YML is represented by a directed graph that can contain loops, iterations and branching. The workflow is defined using a graph description language called YvetteML [37]. This is a high level language with a syntax similar to Pascal and C. In order to describe complex workflows, YvetteML contains several constructs such as service

calls, sequential and parallel iterations, parallel sections, conditional branches and events management. Another characteristic of this language is the possibility to modularise the code [35], i.e. the ability to create a hierarchy of workflows and thus reuse code. Furthermore, YvetteML contains a specific data type that permits a user to define a collection of elements. This data type permits the workflow to have out-of-core executions, global communications and services with unknown number of parameters. These characteristics make the language highly expressive and therefore it is possible to define complex workflows. However, the few limitations of this language are the difficulty of identifying input and output data on a remote computation.

**Chimera Virtual Data Language**

Pegasus [33] is a framework that permits a user to define and execute complex workflows on Grid environments and it is aimed at large-scale data problems. Pegasus works by taking an abstract workflow as an input, where the various jobs are defined independently of the Grid resources available, and generate a concrete workflow, where the jobs are linked to specific resources. Then, this concrete workflow is executed and mapped using DAGMan. The abstract workflow is defined by a programmer using the Chimera system [48] and its virtual data language (VDL). VDL is a data driven language, since the generation of the abstract workflow depends on existing files and selected output file name. In the VDL language, a user defines a job template by using the keyword TR (transformation). A transformation contains the information needed to execute the job, such as executable name, location, command line arguments and input/output files used. The execution of a transformation, thus a job, is defined in the language using the keyword VD (derivation). Associated with a derivation is the name of the template that defines the job and the data files computed. It is possible to use variables instead of strings for the arguments and files in a transformation and a derivation. Given a desired output filename, Chimera analyses the dependencies between the various derivations and transformations to find all the files needed to generate the desired one. Then, Chimera will build the abstract workflow with only the jobs that are needed to create the missing files. Therefore, the workflow generated changes de-

pending on the output filename selected and the existing files. The expressiveness of the VDL language is superior to a XML-based language or a script language, where the various jobs and data dependencies have to be specified directly, since the flow of execution changes depending on input parameters. However, the grammar and syntax of the language are not easy to understand and consequently its usage is not trivial.

### 8.4.2 Languages Used in GridRPC Middlewares

Amar *et al.* [6] introduce a new special agent called MADAG in the DIET middleware. This new component accepts a direct acyclic graph (DAG) structure that represents the flow of executions of the application and permits the middleware to execute directly the various tasks described on the Grid. In their work the authors use the word workflow to define this structure; in this particular case the meaning of the word workflow can be considered to be similar to the word task graph that we use in this thesis.

The user defines the task graph by creating an XML file that contains the necessary information. Then, this XML file is used in the client as an argument of a specific new DIET function. This function executes the task graph without the need to create the respective task calls in the client code. This implementation does not follow the RPC style of calling each task in the application since the only function call used in the code is the one that submits the task graph as a single entity and executes it in the Grid environment. The user, for each task call in the XML file, has to explicitly write the arguments of the task, if they are an input or output, their name, their type and eventually their possible value. This information has to be included also if two tasks have the same arguments. The limitations to this approach are:

- Since the task graph is executed directly by a single function call, this approach does not follow the GridRPC model and the intermediate results between task calls cannot be sent back to the client.

- Furthermore, since the client code is missing the respective task calls, it may be difficult to understand the algorithm described in the task graph

because the XML format is not very expressive and it is not similar to human languages.

- The flow of execution is fixed in the XML file. Therefore, the number of tasks executed, their arguments and the flow itself are not easy to modify.
- The task graph cannot change at run-time depending on some initial values since the XML file does not contain any technique to change the generation flow such as conditional or loop statement.
- It is not user friendly as it can be difficult and time consuming to write the XML description for each task in the graph.

Caron *et al.* [24] introduce in the DIET project a graphical user interface (GUI) that permits the user to graphically design, by drag and drop, the task graph of the application and save the task graph in the XML file to be executed. This approach has the advantage of an easier and less error prone development of the task graph with respect to the XML file. Furthermore, it is easier to understand the algorithm implemented since the graphical representation of the task graph is more expressive than the XML file. However, many previously mentioned limitations still apply for this approach since the execution of the task graph is still done by a single function call and the graphical representation of the flow of execution is still fixed, i.e. it flow does not change depending on initial input values.

Before DIET introduced the use of a task graph and the XML file to define it, the SmartNetSolve middleware, the predecessor to SmartGridSolve, was already using a XML file to generate a task graph of the the GridRPC application's algorithm. SmartNetSolve presented by Brady *et al.* [19] already implemented the collective mapping of a group of tasks and the fully connected network features. However, the automatic task graph method was not conceived at that time. Therefore, the method chosen to generate the task graph was to use a XML file to describe the algorithm and the group of task calls to map collectively. This approach is similar to the one later implemented in DIET and thus they share the same limits. However, the main difference between the two is that SmartNetSolve did not execute directly the task graph generated by the XML file but the task graph was only used to generate the performance model and the mapping. The

real execution of the remote task calls was still happening on the client application through the use of the GridRPC specific methods where the XML file was parsed by a method similar to *grpc_map* to generate the task graph. Therefore, SmartNetSolve was permitting client computation inside the group of tasks.

## 8.5 Summary

In this chapter, we have studied how the problem that the automatic mapper encounters with irregular algorithms can be overcome by using Algorithm Definition Language (ADL) to explicitly specify the task graph that best represents the implemented algorithm of a SmartGridRPC application.

The ADL language is designed to be easy to understand and use, since its syntax is similar to C. It is modular and a module is divided in well defined sections that permits the code to be read easily. Furthermore, The language syntax easily highlights to a user: the data objects used in the algorithm that are communicated on the Grid network, the remote task call executed, the input output objects of a task and the eventual parallel executions. More details about the ADL syntax, semantic and compiler functionalities will be presented in the next chapter.

We have compared the task graphs generated by using ADL with the partial task graphs generated by using *grpc_map* in smaller block of code. We have emphasised how the former contains more information about the group of tasks to map collectively. This information can be important to obtain the benefits previously discussed of SmartGridRPC, such as better mapping, improve computation and communication load. We have conducted experiments that demonstrate significant performance gains in the conditional and adaptive examples due to the use of ADL in conjunction with SmartGridSolve. Therefore, we have shown that ADL in conjuction with SmartGridRPC is a further powerful tool in the hand of scientific users to develop distributed scientifc applications. The use of ADL has the potential to increase the types of distributed applications that can obtain high performance with a SmartGridRPC middleware.

# Chapter 9

# ADL: Language and Compiler

In the first part of this chapter, we present an in depth analysis of the syntax and semantic of the language. The following sections explain how the compiler works and how it is implemented. Finally in section 9.4, the special multi-size multi-dimensional array feature of ADL is introduced.

## 9.1 Language

The ADL language grammar is based on the C language syntax, the reason for this is that C language is extensively used for scientific applications. Furthermore, ADL uses the same approach of C for array indexing by starting the indexing from the value zero. A full description of the grammar of the ADL language is available in the appendix A. One of the main features of the ADL language is the logical division into two groups of the variables used in a module. The first group is composed of local variables and parameters. They are used in the same way as a typical programming language, i.e. to change the flow of execution and data. The second group is composed of Identify Flying Objects (IFOs). They represent the data that are used by remote tasks and can be located anywhere in the Grid environment. The ADL language differentiates between these two types of data in the grammar. Therefore, each type of data can be only used in specific statements, expressions and list of arguments and they cannot be interchanged.

Another important characteristic of ADL is the use of two types of components: tasks and modules. A task is an atomic computation that is executed remotely by a SmartGridRPC call; while a module represents an algorithm which is a changeable flow of execution composed of different tasks and/or sub-modules. A modular approach is a simple way of describing an algorithm and keeping the code easy to write and to understand. In ADL, a module is divided into well defined sections in order to further simplify the reading of code; each section specifies a characteristic of the algorithm and the module.

### 9.1.1 Module Definition

The first step in using the ADL language is to define a module. This is done by using a syntax that is composed of the keyword *module*, the name of the module, the list of parameters and the module body (see table 9.1). This last one is divided into different sections that are: *a*) *component* section, which contains the declaration of tasks and sub-modules used in the module; *b*) *IFO* section, which includes the data objects that are used by the remote task calls and sub-modules; *c*) *algorithm* section, where the flow of execution is described using the previously declared tasks, sub-modules and IFOs; and *d*) *inout* section, which contains the input and output IFOs of the module. The order of these sections in the body is fixed.

**Table 9.1:** *Example of an ADL module definition*

```
1  module name(parameters)
2  {
3    component:
4
5    IFO:
6
7    algorithm:
8
9    inout:
10
11 }
```

The parameters in a module are declared in the same way as the parameters in a function of C language. Therefore, ADL uses the same set of types, qualifiers and rules as this language. The types used are *char*, *int*, *float* and *double*; where the qualifiers are *short*, *long*, *signed* and *unsigned* (see table 9.2 for various examples of declarations). Given that ADL does not use pointers, the *void* type and the use of the asterisk symbol for pointer declaration are therefore not recognised. Furthermore, the use of the qualifier *const* in ADL is redundant, since all the parameters in ADL are always constant. One design choice, made in the ADL project, is that the number of IFOs and their sizes cannot be changed after their declaration in the IFO section. This simplifies the writing and understanding of the ADL code and it eases the generation of the task graph. Therefore, given that parameters are used to set these IFO characteristics, they have to be constant.

**Table 9.2:** *Example of the declaration of various parameters*

```
1 module modex1(char a, short b, unsigned int c, long d){...}
2
3 module modex2(float f, double g){...}
4
5 module modex3(int vector[3], int matrix[3][4]){...}
6
7 module modex4(int numdim, int dimsize[numdim]){...}
```

In ADL, it is possible to declare a parameter as an array of elements or a multi-dimensional array using the same syntax as C. Therefore, the square brackets are used to indicate the size of each dimension of the declared array (see line 5 in table 9.2). The only difference is that the size of an array cannot be empty. Additionally, in ADL is possible to set an array size using a parameter defined in the same parameter list (see line 7 in table 9.2). This leads to a special feature of ADL called a multi-size multi-dimensional array, which will be discussed in section 9.4.

### 9.1.2 Component

The component section in ADL includes the declaration of tasks and modules used in the algorithm; the syntax of this is shown in table 9.3. Task declaration is composed of the keyword *task*, a string and a list of task names. The ADL language needs to provide the compiler with specific information about the remote tasks used in the module. The ADL compiler, for each task, requires the number and type of input/output arguments and the eventual computational complexity of the task. These data are retrieved from a GridSolve Interface Definition Language (gsIDL) [41] file. The gsIDL is the mechanism through which GridSolve and SmartGridSolve enables computational methods to be invoked remotely as a task on a Grid environment (see the following subsection). Therefore in the task declaration, the string after the keyword *task* is used to specify the name of the gsIDL file that contains the data of all the tasks introduced in the following list. If the gsIDL file does not include all of the tasks declared, the ADL compiler generates an output error.

**Table 9.3:** *Example of a component section with tasks and modules declaration*

```
1 component:
2   task     "blas.idl"       ddot,dgemm;
3   module   "example.adl"    example;
4   module                    example2;
```

An idea for future work would be to give the application programmer the ability to provide, “ad hoc”, the task information to the compiler. This would allow task definition in ADL, in a similar way that a module is defined, by using the keyword *task* instead of the keyword *module* and a specific structure of the body. Being able to use a gsIDL file for retrieving the task information would be convenient for already existing GridSolve applications; while in the case of a new application, it would be more appropriate to directly specify the information of a task needed by using the “ad hoc” task definition in ADL.

The declaration of a module, in the component section, is similar to the task declaration with the difference that the keyword used is *module* and it is not nec-

essary to include a string after the keyword. The eventual string is the name of the ADL file that contains the code of all the module names that are into the subsequent list of names. If the string is missing the compiler retrieves, for each module in the list, a file object that contains the information needed. The name of the file object is the same name as the module represented and the file extension is ".mod". The directories, where a module file objects is searched, can be passed to the compiler using a specific argument or environment variable.

### GridSolve IDL

Table 9.4 shows an example of the GridSolve IDL syntax. This example (from "lapack.idl" file) contains the definition of the *dgesv* task, which is a LAPACK routine that computes the solution to a real system of linear equations $A * X = B$. This language permits a programmer to describe the data type of each argument (integer, float, double etc.), the object type of each argument (scalar, vector, or matrix) and whether each argument is an input, an output or an input-output.

**Table 9.4:** *Example of dgesv task definition in GridSolve IDL file "lapack.idl"*

```
1  SUBROUTINE dgesv(
2    IN int N, IN int NRHS, INOUT double A[LDA][N],
3    IN int LDA, OUT int IPIV[N], INOUT double B[LDB][NRHS],
4    IN int LDB, OUT int INFO
5    )
6    "This solves Ax=b using LAPACK"
7    LANGUAGE = "FORTRAN"
8    LIBS = "$(LAPACK_LIBS) $(BLAS_LIBS)"
9    COMPLEXITY = "2.0*pow(N,3.0)*(double)NRHS"
10   MAJOR="COLUMN"
```

This gsIDL example specifies that the first two arguments of the calling sequence are input scalar integers. The third argument of the calling sequence is an input-output, which is a matrix of elements of type double. The fourth argument is an output scalar argument. The fifth argument is an output vector of integers and the sixth is an input-output matrix of doubles. The seventh is a scalar integer

input and the eighth is a scalar output integer. Included in the task definition is a formula that is used in conjunction with the calling sequence to generate a function for calculating the computation load of the task. This formula is denoted in the gsIDL file by the *COMPLEXITY* keyword.

### 9.1.3 IFO: Identified Flying Object

An IFO represents a data object that is used in a remote call, it differs from parameters and variables since it can be anywhere on the Grid environment. The IFOs are important for generating the task graph; since depending on their size and type, they can influence the computational time of the different tasks and the total communication time of the application. Furthermore, the knowledge of which task uses which IFO can help the compiler to determine the dependencies between tasks. In the ADL language, an IFO cannot be used as an element of an expression statement or as an argument of a module parameter list. A variable and a parameter cannot be used as an argument of the input and output list of the module and the task calls. The syntax of an IFO declaration is composed of an IFO type (in upper-case letters to differ from a variable type), the number of dimensions with their sizes and the list of IFO names. For an IFO, the list of types to choose from is linked to the types available in the gsIDL and the GridRPC API. These types are similar to those found in C and Fortran and they are:

- **CHAR**, one byte character.
- **INTEGER**, four bytes integer number.
- **FLOAT**, four bytes single precision floating point number.
- **DOUBLE**, eight bytes double precision floating point number.
- **SCOMPLEX**, eight bytes complex number compose of two single precision numbers.
- **DCOMPLEX**, sixteen bytes complex number compose of two double precision numbers.

The use of upper-case letters for the name of an IFO and lower-case letters for parameters and variables names is not compulsory. However, this technique is suggested since it allows a user to easily discern the IFOs from the other variables and to simply highlight them in the code.

**Table 9.5:** *Example of IFO declaration*

```
1 IFO:
2   CHAR             A,B[10];  // Scalar
3   INTEGER(3)       C;        // Vector
4   DOUBLE(nx)(ny)   D;        // Matrix
5
```

Table 9.5 shows an example of the IFO declaration syntax. The number of round bracket pairs, written after the type, represents the number of dimensions of an IFO. It specifies if the IFO is a scalar, vector, matrix or multi-dimensional object, i.e respectively a no bracket pair, one pair, two pairs and more than two round bracket pairs. The value inside the round bracket pairs correspond to the size of the specific dimension and it cannot be changed in the algorithm section. Consequently the application programmer can use only a literal number or a parameter as the size inside the brackets. In table 9.5, the IFO named *C* (line 3) is a vector composed of 3 integer elements; while the IFO named *D* (line 4) is a matrix with *nx* rows and *ny* columns of double elements. It is possible to declare a multi-dimensional array of IFOs, with all the IFOs having the same number of dimensions and the same size for dimension, by utilising the same syntax used for the declaration of a multi-dimensional array of parameters, i.e. square brackets. Furthermore, it is also possible to declare a multi-dimensional array of IFOs where each IFO has a different size; this is explained in more detail in section 9.4.

### 9.1.4 Algorithm

The algorithm section describes how tasks and/or modules are executed, which IFOs are used and how the parameters change the flow of execution. It is similar to the body of a C function. It can contain loop statements (using the keywords

*for* or *while*) and/or selection statements (using the keywords *if* or *switch*). The syntax used in these statements is the same as the C language. One difference is the use of two specific constructs for parallel execution of tasks, the keywords *parallel* and *parfor*.

The rules used to declare variables in ADL are the same as those used in the C language. However, there are two differences in the actual implementation of ADL language when compared to the C language. At the moment of writing, variables can be declared only at the beginning of the algorithm section, before any other statements. Furthermore, in ADL setting an initial value to a variable has not yet been implemented. The data types and qualifiers available in variable declarations are the same ones used in the parameters declaration.

The syntax of the task and module calls is made in a way to easily highlight the parameters passed and the IFOs used as inputs and outputs. There is a small difference between a task call and a module call that permits a user to simply discern the two various calls. A remote task call is composed of two parts divided by a colon. In the first part there is the name of the task called. In the second part there is the list of input IFOs (with the IFOs separated by commas and contained between two round brackets), followed by an arrow symbol ($->$) and then the list of output IFOs (with the same syntax of the input list). A call statement is completed by the semicolon symbol. A module call is composed in the same way as a task call. The only difference is in the first part where the module call contains, before the colon, a list of parameters to pass to the module (with the same syntax of the input and output lists).

In table 9.6 is possible to see various examples of module and task calls. Thanks to the use of the colon and the arrow symbols, it is easy to recognise the parameters and the input/output IFOs for the different remote calls. The modules used in this table are the same ones declared in table 9.2 and it is possible to see how the parameter lists match the module parameter declarations. Each module in table 9.6 has a different number of input and output IFOs and thus this table shows different methods to indicate input and output IFOs lists. The first module (line 3) has both input and output IFOs. The second module (line 5) does not have any input or output IFOs. The third example (line 7) has only output IFOs while the last module has only input IFOs. In the case of a module call without output

**Table 9.6:** *Example of module calls in ADL*

```
1  algorithm:
2
3    modex1(a,b,c,d):(A,B,C)->(C);
4
5    modex2(f,g):()->();
6
7    modex3(vec,mat):()->(A,B);
8
9    modex4(ndim,dsize):(A);
10
11   taskex1:(A)->(B,C);
12
13   taskex2:(A);
```

IFOs, it is possible to write the arrow symbol followed by an empty list, as in the second example, or to simply indicate only the input list, as in the last module example of line 9. Table 9.6 shows also two examples of task calls (line 11 and 13). It is possible to see that a task call does not have the list of parameters before the colon. While after the colon, the same rules of a module call, for input and output lists of IFOs, are applied for a task call.

A remote task executes a specific function in the remote machine. Therefore, the IFOs of a task call represent the arguments used in the calling sequence of this underlying function. Usually a remote task is composed of many IFOs that are not always significant to generate a task graph. An IFO is significant when contains important data for the algorithm and when is not a scalar, i.e the IFO may be a large object and thus have an impact on data communication. The non-significant IFOs are, for example, size of arrays, flags or return value arguments of the underlying function (see arguments *N*, *NRHS*, *LDA*, *LDB* and *INFO* of the task *dgesv* in table 9.4). These arguments are important for the normal execution of the underlying function, and thus of the remote task call, since they can change the final data. However, the ADL compiler does not needed to know any information about these non-significant arguments to generate a task graph. The task call examples previously introduced in table 9.6, and also all the ADL code examples

of chapter 8, contain only significant IFOs in the lists of input and output. This choice was made to simplify the reading and understanding of the language.

**Table 9.7:** *Example of dgesv task call in a ADL module*

```
1  module ex5(int n, int nrhs, int lda, int ldb) {
2    component:
3      task "blas.idl"         dgesv;
4
5    IFO:
6      DOUBLE(lda)(n)           A;
7      DOUBLE(ldb)(nrhs)        B;
8      DOUBLE(n)                IPIV;
9
10   algorithm:
11     dgesv:(@,@,A,@,B,@)->(A,IPIV,B,@);
12     dgesv:(@n,@nrhs,A,@,B,@)->(A,IPIV,B,@);
13
14   inout:
15   }
```

Table 9.7 shows an example of two more realistic task calls, since the task *dgesv* contains non-significant IFOs in the input and output lists. There is a special symbol called ignore (@) in the task calls. This symbol indicates that the IFO at that particular position in the task call is non-significant to generate a task call and thus can be ignored. In the task call of line 11, the application programmer indicates that ADL compiler can ignore the first, second, fourth and sixth input elements (*N*, *NRHS*, *LDA* and *LDB* in table 9.4) and the last output element (*INFO*). In some cases, a task call may have a high number of consecutive non-significant IFOs. It can be tedious and error prone to write the symbol ignore for each of these IFOs. An alternative solution is to utilise the special symbol ellipsis (...) to indicate that all the following IFOs in the list can be ignored.

In the declaration of *A*, *B* and *IPIV* of table 9.7 (respectively line 6, 7 and 8), the parameters used in the round bracket to indicate the sizes of IFOs match the arguments used to declare the sizes of vector and matrix arguments in the *dgesv* definition of table 9.4. For example the IFO declaration "DOUBLE(lda)(n) A" in

the ADL code corresponds to the "A[LDA][N]" matrix argument declaration in the GridSolve IDL code.

The ADL compiler uses the formula specified by the *COMPLEXITY* keyword in the GridSolve IDL file to calculate the computation load of a task. In the *dgesv* example of table 9.4, this formula includes the *N* and *NRHS* arguments of the function calling sequence. Therefore, the ADL compiler needs the values of these two arguments to calculate the computational load of the *dgesv* task. In the ADL example of table 9.7, the ADL compiler cannot retrieve the values of these two arguments from the task call of line 11, since the symbol ignore is used in the input list. In this particular case, the values of the function arguments *N* and *NRHS* correspond to the values of the ADL parameters *n* and *nrhs*. However, it is not possible to use directly these two parameters in the task call input list, since only IFOs can be passed in the input and output lists of a task or module call. The solution implemented to solve this problem is to use the symbol ignore (@) to specify an expression (called an "at expression") which transforms a parameter or a variable into an IFO. This expression allows the compiler to link the value of the indicated parameter to the relative function argument. Table 9.7 shows an example of this specific expression on the task call of line 12. It is possible to see that the parameters *n* and *nrhs* are included after the symbol ignore in the two first element of the input list. This action links the values of these two ADL parameters to the function arguments *N* and *NRHS*. Therefore in this example, the ADL compiler calculates the computational load of the task using the values of *n* and *nrhs* in the complexity formula.

### 9.1.5 Inout

The input and output IFOs of a module, which are passed during a module call, can be declared in the *inout* section of the module definition. The syntax of the input IFOs declaration is composed of the keyword *input* followed by a comma-separated list of IFO names. This list is then terminated by a semicolon symbol. In the case of output IFOs, the syntax is the same one, but in this case the keyword *output* is used instead of the keyword *input*. If a module does not have any input and output IFOs, the *inout* section can be left empty. In the case that there are no

IFOs in one of the two lists, the solution is to use the specific keyword followed directly by the semicolon symbol, therefore without any IFO name in the list of names. The two lists of IFOs can be declared in any order in the *inout* section. IFOs are passed by reference between

In a module call it is possible to pass, as an argument of the input or output list, a individual IFO or a multi-dimensional array of IFOs. In the case of a single IFO, the ADL compiler checks that the type, the number of IFO dimensions and their sizes are the same between the IFOs of the module call and the corresponding IFOs declared in the input or output list of the module called. In the case of an array of IFOs, the compiler also has to check the number of dimensions of the array and the size of these dimensions.

**Table 9.8:** *An example declaration of input and output IFOs lists in an ADL module*

```
1  module ex6(int n, int size, int i){
2    component:
3      task "example.idl"       sum;
4
5    IFO:
6      DOUBLE(size)          A[n],B[n],RES;
7
8    algorithm:
9      sum:(RES,A[i],B[i])->(RES);
10     if(i<n)
11        ex6(n,size,i+1):(RES,A,B)->(RES);
12
13   inout:
14     input  RES,A,B;
15     output RES;
16   }
```

Table 9.8 shows an example of the syntax used for the declaration of the input and output lists of IFOs. In this example, the module is recursive and it takes as inputs three elements and as an output only one element. All these elements are vectors of type double (see IFOs declaration in line 6). The first input (*RES*) is a single IFO while the other two inputs (*A* and *B*) are arrays of IFOs. The only

output element of the module is the IFO *RES* that is also the first input element. The arrays of IFOs are indicated using only the name of the arrays in the input list declaration, without the need of any other symbol (line 14); this is as the input list of the module call (line 11). As previously mentioned, the ADL compiler knows that *A* and *B* in the input list declaration are arrays of IFOs and therefore it checks that the elements in the module call input list are arrays of IFOs with the same characteristics.

## 9.2 Compiler

The main objective of the ADL compiler is to allow an external code (such as a SmartGridRPC middleware) to easily obtain at run-time a task graph of a module; where the task graph can be changed depending on the values of the parameters passed.

During design time, there were three possible techniques that could have been deployed in order to achieve this result. The first technique is to make the compiler directly output the task graph, given the ADL code and the value of the parameters as inputs. The task graph would be saved in a file using a special data structure and then it would be loaded into the external code at run-time from this file. The second technique is to directly insert parts of the compiler functionality into the external code at run time. The compiler would parse the ADL code and then yield a special tree structure that corresponds to the ADL code of the module, such as a syntax tree. This tree structure would be saved in a file that would be loaded by the external code at run-time. Then, the external code would traverse this tree structure depending on the values of the parameters and during this traversal of the tree the task graph would be generated. The third technique that was considered is to make the ADL compiler yield a C language code that contains special methods and structures which can be used at run-time to generate the task graph. The code yielded would be specific to the module that was used as the input of the ADL compiler. At run-time, the external code has only to call these special methods by passing the desired values of the parameters.

The main drawback of the first technique is that the task graph is fixed at run time. This is against the main objective of ADL, which is to be able to change

the task graph at run-time depending on the value of the parameters, therefore this technique was not considered. The second technique has the small issue that, in order to execute some of the compiler functionalities, the external code needs to load part of the compiler code and thus to have a deep integration with the compiler internal implementation. Furthermore, this technique has the drawback that the action of loading the tree structure from a file and of traversing the tree are very time consuming. The third technique has the small issue that the external code needs to be linked together with the code generated by the ADL compiler. However once this is done, for example by creating a library with the code generated, the final application does not need to keep any link to the compiler. Given that in the case of the second technique the loading and traversing of the tree is happening for every execution, while for the third technique the code generated needs to be compiled only once and then the execution is really fast, the third technique is more favourable. Therefore, we chose to implement the third technique for the ADL compiler.

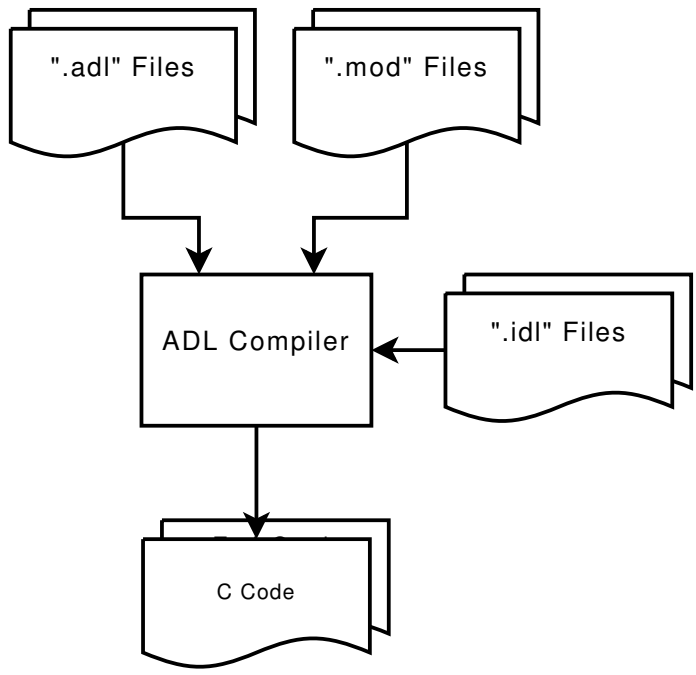



**Figure 9.1:** *Example of the use of the ADL compiler*

Figure 9.1 shows the various steps of a typical utilisation of the compiler. The ADL compiler takes as an input a single or multiple files (".adl" extension) which contain the module and sub-modules that describe the application's algorithm. While the compiler is compiling the ADL code, it fetches the files which contain the definitions of the sub-modules and tasks that are declared in the component section. These files are module objects (".mod" extension) and gsIDL task definitions (".idl" extension). At the end of the execution, the ADL compiler yields a C code file for each module used in the algorithm. In order to generate the task graph, the external code needs to be compiled together with the yielded code.

### 9.2.1 Internal Structure

The internal structure of the ADL compiler follows the typical structure of a compiler, as shown in figure 9.2 on page 128. It is composed of a scanner (i.e. lexical analyser), a parser (i.e. syntax analyser), an attribute syntax tree (i.e. intermediate representation), tree manipulation procedures and a code generator.

The ADL input file is read by the scanner which separates the code into tokens. These tokens are recognised in the code using specific rules and regular expressions. The parser uses these tokens as inputs and then, by following the grammar that specifies the ADL language, it generates an attribute syntax tree. During this execution, the parser also reads the information from the gsIDL files. The parser is a look-ahead LR parser (LALR) and the grammar is written in the BNF notation (see appendix A). The attribute syntax tree represents, in tree form, the code parsed in almost a one-to-one correspondence. The attributes of the tree are used to do the semantic check on the ADL code parsed, such as the correctness of the type and number of the parameters and IFOs in a task or module call. Furthermore, the attribute syntax tree is the internal structure of the module file object. Therefore the various module objects retrieved are merged in a singular tree during the compilation. The attribute syntax tree is used by the code generator to produce the corresponding C code of the modules.

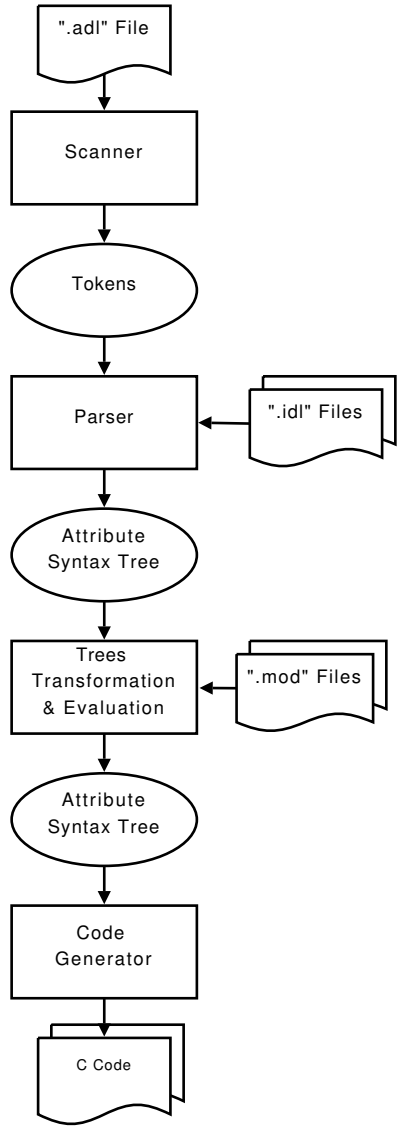



**Figure 9.2:** *Internal structure of the ADL compiler*

### 9.2.2 Output Code

The choice to use the C language as a target language for the code generated by the ADL compiler was made for three main reasons. The first reason is that the C language is well-known by the scientific community and therefore it is used in many scientific applications. Second, the GridSolve middleware and consequently the SmartGridSolve middleware are written in C, and this simplifies the integration of the generated code into the SmartGridSolve middleware. Furthermore, C code can be easily integrated with code written in many other different languages. Third, the tools used to create the ADL compiler (see section 9.3) are well-suited to produce a compiler that generates procedural programming languages code, as it is the C language.

The ADL compiler generates for each module two C functions, called internal and external. The internal function is used when a module calls another module inside its algorithm section. Therefore the function of the caller module executes the internal function of the called module. It is the internal function that contains the algorithm section of the module. The external function, which has the same name of the module, is used to interface between the internal function of a module and the external code. The majority of the structures used to represent the parameters, variables and IFOs are initialised in the external function. Furthermore, the external function needs a common interface between modules in order to permit the external code to use different modules interchangeably. To achieve this, the external function utilises the variable arguments list technique, in a similar way to the *printf* function call of the C language. In chapter 8, two examples are presented on how this technique can be used to generate the task graph of a module; specifically in table 8.2 on page 96 for the conditional algorithm example and in table 8.4 on page 100 for the adaptive algorithm example.

In order to generate the task graph of a module, an external code (such as the *grpc_map* method of SmartGridSolve) needs to call the external function of the module using a special wrapper method. Table 9.9 shows how this wrapper method is used (in the upper part of the table) and its code (in the lower part of the table). The wrapper method takes as arguments the pointer to the external function (i.e. the same name of the module), the pointer to the task graph structure,

**Table 9.9:** *Example of the task graph generation in the external code through the use of the external function and the wrapper method*

```
1 // External code
2 adltg(cndalg,&tg,str,size,cndtrue,cndfalse);
```

```
1 int adltg(adlmain_pt ext_func, adltg_pt tg, char *str, ...){
2   adlva_t  args; // Variable arguments list
3   va_start(args.va, str);
4   int ret = (*ext_func)(tg,&args); // Calls external-function
5   va_end(adlva.va);
6   return ret;
7 }
```

and the string and the arguments of the variable arguments list. Inside the wrapper method: the variable argument list is initialised (line 3); then the external function of the module is called (line 4) using as arguments the task graph structure and the variable arguments list; finally the variable arguments list is finalised (line 5). It is the external function that produces the task graph, which is returned to the external code through the pointer to the task graph structure.

### Pseudo-code of the Output Code

In the following part of this subsection, we introduce the pseudo-code of the code generated by the ADL compiler for the most important part of an ADL module. The module used as the base example is in table 9.8 on page 124. This is a simple module that contains the base functionalities of the ADL language, such as the parameters' initialisation, the array of IFOs, the task call, the module call and the input and output of IFOs.

Table 9.10 shows how the parameters of the module *ex6* are initialised in the generated C code. The upper part of the table shows the portion of the ADL code considered while the lower part of the table shows the pseudo-code. This pseudo-code represents the code at the beginning of the external function of the module. In this particular case, the initialisation of the parameter is straightforward since all the parameters are single variables (see section 9.4 for multi-dimensional array initialisation). The variable arguments list that is passed as an argument of the

**Table 9.10:** *Example of parameters' initialisation in the generated C code*

```
1  module ex6(int n, int size, int i)
```

```
1: {args is the variable arguments list}
2: init n ← first argument of args
3: init size ← second argument of args
4: init i ← third argument of args
```

external function is used to retrieve the values of the parameters. These values are in the same order as the initialisation of the parameters.

**Table 9.11:** *Example of IFOs initialisation in the generated C code*

```
1  DOUBLE(size)          A[n],B[n],RES;
```

```
1: init array A ← n elements
2: for i = 1 to n do
3:     init element i of A ← size size and type DOUBLE
4: end for
5: ... {Similar initialisation for B}
6: init RES ←size size and type DOUBLE
```

Table 9.11 shows how the IFOs of the module *ex6* are initialised in the generated C code. Also this pseudo-code represents the code at the beginning of the external function of the module. The array of IFOs $A$ and the array of IFOs $B$ (that is not shown) are initialised to have the value of the parameter $n$ as the number of elements of the array. Then for each element in the array, the size of the IFO is set to be the value of parameter *size* and of type double. The same size and type are set for the single IFO *RES*. In the C code generated, the array of IFOs and a single IFO are defined using a specific structure that can be managed as a single element or as an array of elements. This structure allows the code to have an array of arrays. A similar structure is used for the parameters. When an IFO, an array of IFOs, a parameter and an array of parameters are passed between internal functions in the C code, they are passed as pointers to their structures. Therefore in the ADL language, parameters and IFOs are passed by reference between modules.

Table 9.12 shows how the algorithm section of the module *ex6* is executed in

**Table 9.12:** *Example of the algorithm section in the generated C code*

```
1  sum:(RES,A[i],B[i])->(RES);
2  if(i<n)
3    ex6(n,size,i+1):(RES,A,B)->(RES);
```

```
1: input ← RES, element i of A, element i of B
2: output ← RES
3: call sum_task ← input, output
4: if i < n then
5:    input ← RES, A, B
6:    output ← RES
7:    parameters ← n, size, i + 1
8:    call ex6_internal ← parameters, input, output
9: end if
```

the generated C code. This pseudo-code represents the code inside the internal function of the module. It is the internal function of a module that is called when there is a module call in the ADL code (see line 8 of the pseudo-code). The interface of the internal function takes as input three lists: the list of input IFOs, the list of output IFOs and the list of parameters. The order of the elements in these lists is set by the inout section and by the parameter list of the ADL code of the module. A task call in the ADL code is represented in the generated code by a call to a special task function (see line 3 of the pseudo-code). Since task calls take as arguments only input and output IFOs, the task functions take only two arguments, i.e. not the list of parameters.

When a task function is executed, a special identifier of the IFO is memorised for each input and output IFO in a global structure together with a unique identifier of the task call. At the end of the execution of the external function, this global structure is populated with all identifiers of the task calls and respective IFOs of the module and sub-modules. Then, these identifiers are analysed while traversing the global structure in order to build a list of data and task dependencies. At the end, the global structure with the list of dependencies is used to build the task graph of the module. This task graph is then returned to the external code that called the external function of the module.

## 9.3 Compiler Implementation

The ADL compiler is made using the Cocktail Toolbox [52], which is a set of tools that allows a user to generate nearly all functionalities of a compiler, and the C language, which is used to manage and to connect these different functionalities. The Cocktail tools [53] used in ADL are: (i) *rex* that builds the scanner from the scanner specification file; (ii) *lark* that generates the LALR parser from the parser specification file; (iii) *ast* that provides the tree manipulating procedures and is used to define the structure of the attribute syntax tree; (iv) *ag* which generates the procedures that transform and evaluate the attributes of the attribute syntax tree; and (v) the *puma* tool which produces the methods that are used to build the final C code from the attribute syntax tree. These tools take as inputs different files, which are coded using different languages, and output C code files, which need to be compiled together to obtain the ADL compiler. Another important instrument of the Cocktail toolbox is the *Reuse* library which contains the data types and procedures that are useful for the development of the compiler, such as hash map, string manipulation and dynamic array functionalities. In the following subsections we introduce, for each function of the ADL compiler, the code that we specified and tools that we used in order to produce that functionality.

### 9.3.1 Scanner

The scanner of the ADL compiler is made using the *rex* tool. This tool takes a specification file as the input, which contains the rules that define the various tokens to be recognised, and it outputs C code, which contains the functions that read the input file and return the tokens matched. The rules in the specification file are composed of various different regular expressions. In order to match all the possible tokens, we defined five families of rules in the specification file, which allow the scanner to identify: (1) the keywords, such as *if*, *double* and *IFO*; (2) the symbols, such as open or close parenthesis, semicolon, colon and full arrow; (3) the identifiers, such as the names of variables, parameters, task and modules; (4) the mathematical constants; (5) and the strings. Table 9.13 shows the regular expressions and the rule that recognise the identifiers in the ADL code.

**Table 9.13:** *Example of the regular expressions and rules used to generate the scanner*

```
1 ...
2 D = {0-9}          .
3 L = {a-zA-Z_}   .
4
5 RULES
6 L(L|D)* : { return IDENTIFIER; }
7 ...
```

### 9.3.2 Parser

The parser of the ADL compiler is made using the *lark* tool. This tool accepts a specification file as the input, which contains the attribute grammar, and outputs a C code. The attribute grammar defines the syntax of the ADL language. It is composed of different rules, where each rule matches a specific sequence of tokens and executes a specific action. In the case of the ADL compiler, we defined that for each action a node of the attribute syntax tree is produced, where the node represents the rule recognised. The C code yielded by the *lark* tool contains the method that reads the tokens generated by the scanner and outputs the final attribute syntax tree.

**Table 9.14:** *Example of the grammar used to generate the parser*

```
1 iteration_stmt = <
2   ...
3   = fs:for_specifier '(' e1:expr';'e2:expr';'e3:expr')' bs:block_stmt
4   { Tree:=mIRFor(fs:ForSpec,e1:Tree,e2:Tree,e3:Tree,bs:Tree); } .
5   ...
6 > .
7 for_specifier = <
8   = FOR    { ForSpec:=For; } .
9   = PARFOR { ForSpec:=ParFor; } .
10 > .
```

Table 9.14 shows the two rules that match the possible variations of a *for* statement. The bottom rule (*for_specifier*) recognises the token (*FOR* or *PAR-*

*FOR*) used as a keyword of the statement. The upper rule recognises the different expressions of the *for* statement and the following block of code. The code inside the curly brackets is the action taken when the rule is recognised. In this example, the parser generates the node *IRFor* using the method *mIRFor*, which is produced by the *ast* tool (see the following sub-section), where the leaves of the node are the expressions of the statement and the attribute is the type of the *for* statement.

### 9.3.3 Attribute Syntax Tree

The methods that are used in the parser to produce the attribute syntax tree are made using the *ast* tool. This tool takes a file as the input, which describes the structure of the tree, and outputs a C code, which contains the functions that generate the tree. The attribute syntax tree is composed of various nodes where each node represents a syntax rule and contains various sub-trees (leaves) and attributes. Table 9.15 shows the structure of the node *IRFor* that represents the *for*

**Table 9.15:** *Example of the code used to generate the attribute syntax tree*

```
1 StmtList =<
2   NoStmt = .
3   Statement = Next: StmtList REV <
4     ...
5     IRFor = [ForSpec: int] IRInit:Expr IRCond:Expr
6             IRIncr:Expr  IRBody:StmtList.
7     ...
8   > .
9 > .
```

statement. The arguments of the method used to build the node, *mIRFor* (presented in the previous table), are the elements defined in the structure of the node. In order to simplify the generation of the attribute syntax tree, we defined its structure as similar as possible to the grammar used by the parser; thus similar to the syntax of the ADL language. The attributes of the attribute syntax tree are evaluated by specific methods that are generated using the *ag* tool. We specify for each node of the tree and for each attribute of the node a distinct computation. These computations are used to analyse the semantic of the ADL code.

### 9.3.4 Code Generator

The code generator functionality takes as the input the attribute syntax tree and produces the target C code. This target code contains the internal and external functions that are used at run-time to build the task graph of the module.

The code of this functionality, which is part of the ADL compiler, is made using the *puma* tool and a specific input file. The code generated by *puma* works by traversing, multiple times, each node of the attribute syntax tree. During the traversing, the code matches the node traversed, and the specific values of the attributes and the leaves of the node, with an action to take. The nodes to recognise and the respective actions to perform during the traversing of the tree are defined in the input file passed to the *puma* tool. For the ADL compiler, we specified that each action performed writes in a file the target C code that represents the matched node.

**Table 9.16:** *Example of the code used to generate the target C code*

```
1 IRFor({For},IRInit,IRCond,IRIncr,IRBody),
2 TGCode(TGFile:=TGFile(_,_,FD,NTab),Ret:=Ret):-
3   printTGFile(FD,NTab,0,0,"");
4   Ret:=Ret & TGCodeExpr(IRInit,TG);
5   printTGFile(FD,0,0,1,";");
6   printTGFile(FD,NTab,0,0,"for(; ");
7   Ret:=Ret & TGCodeExpr(IRCond,TG);
8   printTGFile(FD,0,0,0,"; ");
9   Ret:=Ret & TGCodeExpr(IRIncr,TG);
10  printTGFile(FD,0,0,1,")");
11  Ret:=Ret & TGCodeAlgo(IRBody,TG);
12  RETURN Ret;
13  .
```

Table 9.16 shows a part of the input file passed to the *puma* tool that is used to produce the code generator functionality. This example contains the action taken when the node traversed is the *IRFor* node that represent the *for* statement. The first line of the example indicates the node to match, the value of the attribute *ForSpec* and the value of the leaves. While the lines after the *:-* symbol indicate the action to take. In this case, the action prints to file the target C code, with

the method *printTGFile*, and traverses recursively the leaves of the node, with the method *TGCodeEpxr*.

## 9.4 Multi-size Multi-dimensional Array

As previously mentioned, one of the main characteristics of the ADL language is the possibility of declaring a parameter array where the size of the array is set by a parameter declared in the same parameters list. This allows a programmer to create multi-dimensional arrays with multiple sizes for each dimension after the first one. This technique can be useful to easily declare arrays of IFOs, or tree like structures of IFOs, where each IFO has a different size.

**Table 9.17:** *Example of the declaration of a multi-size multi-dimensional arrays with one dimension*

```
1 module msmda1(int ndim1, int sizeA[ndim1]) {
2   ...
3   IFO:
4     DOUBLE(sizeA)          A[ndim1];
5   ...
6 }
```

The table 9.17 shows an example of the declaration of a multi-size multi-dimensional array and how this array is used to declare an array of IFOs with different sizes for each IFO. The array in this example has only one dimension and it is similar to the example shown in table 8.3 for the adaptive algorithm. The programmer, by setting the value of *ndim1* parameter, can decide the number of parameters in the array *sizeA* and the number of IFOs in the array *A*. Then, he or she can choose the size of each IFO in the array by setting the respective value of the parameter of the *sizeA* array.

Table 9.18 shows the pseudo-code of the code generated from the ADL module previously introduced. This pseudo-code initialises the multi-dimensional multi-size array of parameters in the first five lines, and initialises the respective array of IFOs in the following lines of code. The number of elements in the *sizeA* array

**Table 9.18:** *Example of the initialisation of a multi-size multi-dimensional array with one dimension in the generated C code*

```
1: init ndim1 ← first argument of args
2: init array sizeA ← ndim1 elements
3: for i = 1 to ndim1 do
4:     init element i of sizeA ← next argument of args
5: end for
6: init array A ← ndim1 elements
7: for i = 1 to ndim1 do
8:     init element i of A ← size (value i of sizeA) and type DOUBLE
9: end for
```

depends on the value of *ndim1*. Furthermore, the value of each element in the array is retrieved from the variable arguments list *args*. A similar code is executed for the array of IFOs. The number of elements for this array depends on the value of *ndim1*. Then, for each element in the array *A*, the size of the IFO is set by the value of the equivalent element (*i*) of the array of parameters *sizeA*.

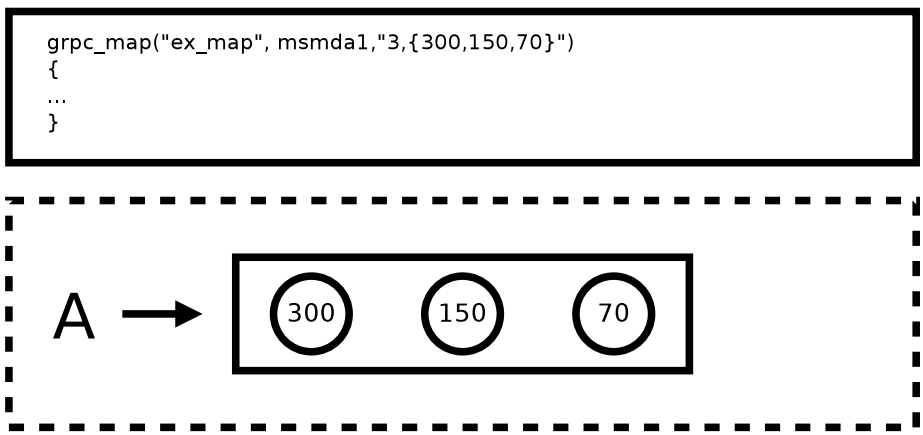



**Figure 9.3:** *Example of a multi-size multi-dimensional array of IFOs with one dimension*

Figure 9.3 shows an example utilisation of the ADL code in table 9.17. The upper part of the figure presents how the module is called, while the lower part shows the resulting IFO. The values of the parameters are set through the use of the string in the *grpc_map* method. The value of *ndim* is the first number of the string (3), while the values of the array *sizeA* are the following numbers inside the

pair of curly brackets (300, 15, 70). the number of IFOs in the array *A* is equal to the value of parameter *ndim1*, while the size of the IFOs (the circles in the picture) are equal to the values of the parameters of *sizeA*.

**Table 9.19:** *Example of the declaration of a multi-size multi-dimensional array with two dimensions*

```
1 module msmda2(int ndim1, int ndim2[ndim1],
2               int sizeB[ndim1][ndim2]) {
3   ...
4   IFO:
5     DOUBLE(sizeB)          B[ndim1][ndim2];
6   ...
7 }
```

In table 9.19 there is a more advanced example of the utilisation of the multi-size multi-dimensional array technique. The array *sizeB* has two dimensions: the first dimension is declared using the simple parameter *ndim1*, while the second dimension is declared using the multi-size multi-dimensional array *ndim2*. The array *sizeB* is used to set the size of each IFO in the array of IFOs *B*. It is important to notice that the parameters used to set the dimension of the array of IFOs *B* are the same parameters used to set the dimension of the array of parameters *sizeB*.

Table 9.20 shows the pseudo-code of the C code generated that initialises the multi-dimensional multi-size array of parameters with two dimensions, in the first eight lines. That, in turn, initialises the respective IFO, in the following lines of code. The number of elements in the first dimension of the array *sizeB* depends on the value of *ndim1*. Then, each element in the first dimension of *sizeB* is an array as well, where the number of elements of this array depends on the value of the equivalent element (*i*) of the array *ndim2* (see line 3). Thus, all the elements in the first dimension are arrays of different sizes. The value of the elements in the second dimension of the array *sizeB* are retrieved from the variable arguments list (see line 5). The initialisation of the array of IFOs *B* is similar to one of the array of parameters *sizeB*. Thus, *B* is an array of arrays where all the arrays in the second dimension have different sizes. Furthermore, all the IFOs in *B* have

**Table 9.20:** *Example of the initialisation of a multi-size multi-dimensional array with two dimensions in the generated C Code*

```
 1: init array sizeB ← ndim1 elements
 2: for i = 1 to ndim1 do
 3:     init element i of sizeB ← (value i of ndim2) elements
 4:     for j = 1 to (value i of ndim2) do
 5:         init element i and j of sizeB ← next argument of args
 6:     end for
 7: end for
 8: init B ← ndim1 elements
 9: for i = 1 to ndim1 do
10:     init element i of B ← (value i of ndim2) elements
11:     for j = 1 to (value i of ndim2) do
12:         init element i and j of B ← size (value i and j of sizeB) ...
13:         ... and type DOUBLE
14:     end for
15: end for
```

different sizes.

Figure 9.4 shows an example of utilisation of the ADL code in table 9.19 The values of the parameter are set through the use of the string in the *grpc_map* method. The value of *ndim1* is equal to three and therefore the first dimension of the array of IFOs *B* contains three elements. The first pair of curly brackets in the string have the values (2, 1, 3) of the array *ndim2*. These values are the numbers of elements of the arrays in the second dimension. The figure shows that the array in the first position has two elements, the one in the second position has only one element, while the last array has three elements. Finally, the last pair of curly brackets in the string contains the values of the array *sizeB* and thus the sizes of the IFOs. This last pair is divided in three sub-pairs of curly brackets. These represent the three elements of the first dimension of the array *sizeB*. The number of elements of each one of these sub-pairs is equal to the equivalent value of the array *ndim2*. The array of IFOs *B* is composed of three elements where: the first element is an array of two IFOs of sizes 100 and 200; the second element is an array of one element of size 50; and the third element is an array of three elements of sizes 300, 150 and 70.

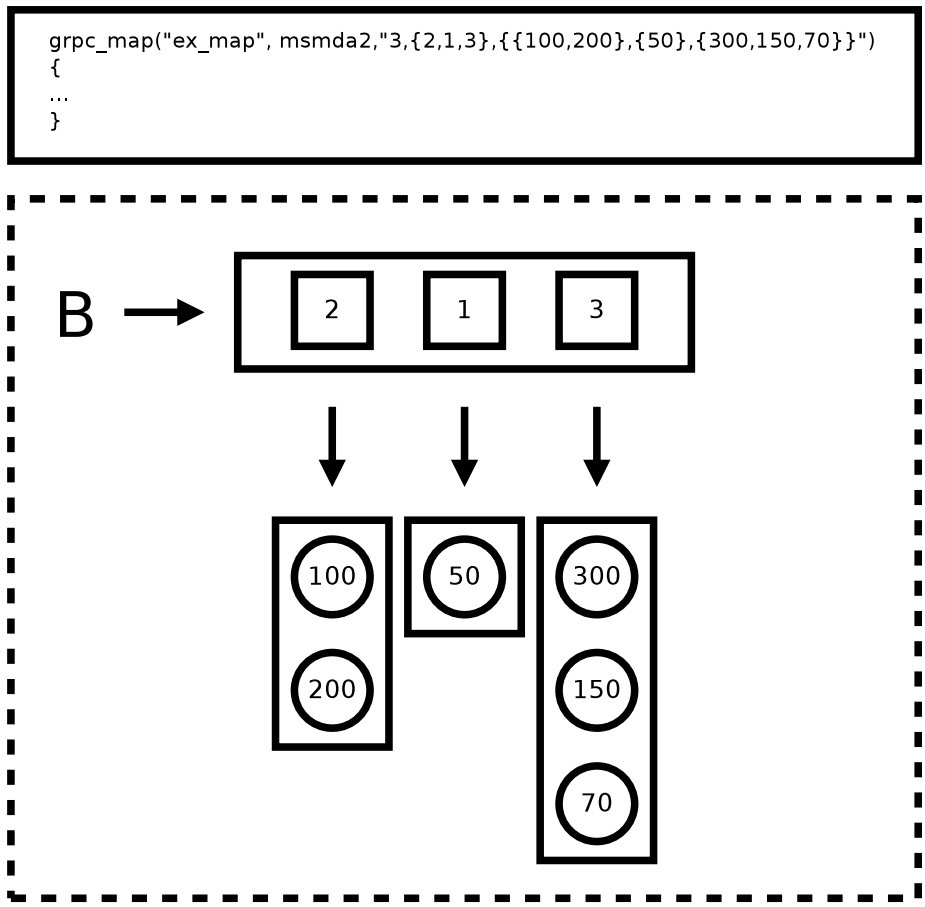



**Figure 9.4:** *Example of a multi-size multi-dimensional array of IFOs with two dimensions*

## 9.5 Summary

In this chapter, we have presented the grammar, syntax and semantic of the language by doing an in depth analysis of each section of the ADL module. We have introduced the characteristics and motivations of the language such as the division between local variables and parameters and the Identify Flying Objects (IFOs), and the division of the language in modules. Furthermore, we have shown how a task can be added in the language by using the GridSolve IDL. In the second part of this chapter, we have introduced the ADL compiler, its internal structure, and how the code is produced and used to generate the task graph by the Smart-GridSolve middleware. Furthermore, we have presented how the various part of

the compiler (the scanner, parser, attribute syntax tree and code generator) are implemented using the cocktail toolbox. Finally, we have introduced the multi-dimensional arrays technique that can be useful to easily declare arrays of IFOs, or tree like structures of IFOs, where each IFO has a different size.

# Chapter 10

# Conclusion and Future Work

The idea behind the design and development of GridRPC was to create a standard that provides an appropriate programming model for Grid computing while meeting the needs of computational scientists. Thus permitting a scientific user to easily create Grid applications that take advantage of the vast resources available on the Grid. Since the remote procedure call programming method is a widely used approach for distributed computing and it is easy to understand and easy to use, a RPC model adjusted for Grid computing was chosen. The GridRPC programming model has several non performance related benefits and thanks to the experience of developing a Grid-enabled version of Hydropad we have found the beneficial characteristics of GridRPC to be: (a) an easy and powerful development paradigm; (b) the ability to simply reuse code and algorithms; (c) the possibility of exploiting the natural task parallelism of scientific problems; (d) its portability and (e) the increased control over the application.

Naturally, since the targets of the GridRPC standard are computational scientists, the performance potential of the model and relative middlewares are important and this potential has to be applicable to a wide range of distributed scientific applications. However, we have determined that a comprehensive performance evaluation of GridRPC middlewares, which shows their limits and benefits for any types of distributed applications, has never been done. Generally, the only applications used as performance analysis tools are of classes I and II, which are ideal to get top performance in a Grid environment. This situation is problematic

because the objective of reaching a wide audience with the GridRPC standard may not get achieved; given that many systems exist that are highly specialised in executing class I and II applications in Grid environments, such as batch management systems and stream processing systems.

The need to analyse the performance of GridRPC programming systems also for applications of class III has motivated us in designing and developing the GridRPC implementation of Hydropad. This real life tightly synchronised task parallel distributed application has characteristics that are missing on all the other applications used to analyse GridRPC middlewares, such as tasks with a balanced ratio between computation and communication, a high level of data synchronisation between tasks and a minimal task parallelism. This kind of application represents border line applications that can achieve high performance using Grid computing. Therefore, Grid-enabled Hydropad is an effective test for GridRPC programming systems.

In the experimental results using the GridSolve middleware, we have identified that the performance benefits of the GridRPC implementation of Hydropad versus the original sequential execution can be grouped into two categories: (1) the faster solution of a given problem, thanks to the parallel execution of tasks and the computational powerful servers on the Grid, and (2) the decrease in the amount of memory used on the client machine, and thus paging for large problems, since the temporary memory of the remote tasks is allocated on the servers. However, our experimental results show that the GridRPC version of Hydropad has a faster execution than the sequential version only when there are fast client-to-server connection links, i.e. 1Gb/s in these experiments. In fact, in the experiments with slow client-to-server links the GridRPC version of Hydropad is slower with small problems than the sequential problems. Despite that, in the same experiments the GridRPC version of Hydropad achieves faster execution times with problems larger than the client main memory thanks to less paging on the client machine. However, considering the limitations in the communication model of GridRPC, we believe that GridRPC as it stands is not the ideal paradigm for executing tightly synchronised task parallel distributed application on a Grid environment.

In order to overcome the communication model limitations of GridRPC and the restrictions caused by GridRPC individual task mapping, Brady *et al.* [17] designed an extension of the GridRPC model, SmartGridRPC, and developed the relative middleware, SmartGridSolve, that permit a Grid application to avail of collective mapping of a group of tasks, direct server-to-server communication, broadcast communication and data caching on a server. This is done in SmartGridRPC thanks to the detailed information about the underlying algorithm of the group of tasks stored in the task graph DAG structure. SmartGridRPC, in order to retrieve this information, define two new methods, *grpc_map* and *grpc_local*, in the GridRPC API that automatically generate the task graph. These two new methods are introduced in SmartGridRPC without changing the RPC programming model concept and thus SmartGridRPC maintains the previously discussed non performance related benefits of the GridRPC model.

In the experimental analysis of SmartGridRPC we have used Hydropad as the test case by comparing its SmartGridSolve version against the GridSolve and sequential ones. Our analysis shows the performance benefits that SmartGridRPC delivers to class III applications are: (i) improved computation load; (ii) improved communication load; (iii) further reduced memory usages and paging on the client machine; and (iv) minimal performance influence by the client-side hardware. We have shown that the performance advantages generated by the new features of SmartGridRPC allow Hydropad to obtain quite significant performance gains in comparison to the GridRPC implementation and to the sequential one. An important point in the experiments is that these performance gains are not influenced negatively by the bottleneck of a slow client-servers connection and/or low amount of memory in the client machine as with the GridRPC model. Therefore, we have shown that SmartGridRPC is an effective alternative tool for computational scientists to deploy class III applications and thus a wide range of distributed scientific applications on a Grid environment.

Additional experiments worth investigating would be analysing the performance of SmartGridRPC model when the Grid environment is composed of geographically separated cluster of clusters; since with this set-up new dynamics such as network delay and server stability would need to be taken into account. Therefore, a possible future work would be to implement the server side code

of the various Hydropad tasks using MPI or openMP to utilise the natural data parallelism of the N-Body, FFT and PPM algorithms. Unfortunately, at the time of writing the integration of MPI or openMP tasks on SmartGridRPC is not yet ready.

While the SmartGridRPC programming model achieves high performance on different types of applications, the performances are dependant on the use of a task graph that is representative of the underlying application's algorithm. In this thesis, we have analysed the automatic task graph generator method of SmartGridRPC in the case of irregular algorithm, i.e. algorithm that change the flow of execution depending on values computed by remote tasks. The problem is that the automatic task graph generation may not work for this kind of algorithm or may not generate a representative task graph. We have presented three different examples of irregular algorithms: (1) iterative algorithm, where the algorithm iterates the computation of a problem until its solution reaches a desired accuracy; (2) conditional algorithm, where the flow of execution depends on a conditional statement; and (3) adaptive algorithm, where the internal data structure, and consequently the flow of execution, depends on the data processed. In this thesis, we have introduced three trivial example applications, implemented in GridRPC and SmartGridRPC, of these irregular algorithms. We have shown in the SmartGridRPC examples two techniques that can partially solve these example problems. These techniques are; modifying the code to map multiple iterations (in the case of iterative algorithm) and generating partial task graphs by mapping smaller areas of code where the algorithm is static (in the other two algorithms). While the first technique generates a task graph that is representative of the algorithm and permits the application to achieve high performance (the iterative algorithm is the main algorithm used in Hydropad), the other technique generates small task graphs that are missing some important information needed to achieve high performance. Therefore, in the case of irregular algorithms, SmartGridRPC cannot always achieve the highest performance possible.

In order to overcome the problem of the automatic task graph generation with irregular algorithms, we designed and developed a specific high level language called Algorithm Definition Language (ADL). This language permits the application programmer to specify explicitly the task graph for any type of application's

algorithms. The language has the following characteristics: (•) a syntax similar to C language; (•) it is a modular language and each module is divided into well-defined sections that specifies distinct parts of the algorithm; (•) it provides a way of changing the sizes of tasks' objects from the client code and to calculate the relative execution time of the tasks; (•) it specifies possible changes in the flow of task calls and permits a user to select the flow dynamically from the client code; (•) it highlights the tasks that can be executed asynchronously; (•) it permits an easy identification of input and output objects of a task; and (•) it provides a technique to set an eventual client computation and the data objects used on it. The design objectives of ADL are ease of use and ease of code understanding. The ADL compiler works by generating C code that can be easily integrated in the SmartGridRPC client code. Then, the task graph is built at run time in the client code using functions and structures introduced in the compiler output code.

In this thesis, we have shown how to use ADL to generate representative task graphs for the conditional and adaptive trivial examples and presented how the application user can set the task graph flow and object data sizes from the client code. In the experimental results, we have compared the execution time of the SmartGridSolve with ADL version of the examples versus the SmartGridSolve with smaller code maps and GridSolve version of the examples. These experiments show that the use of ADL to explicitly specify the task graph permits SmartGridRPC to obtain high performance in comparison to the other two versions, thanks to more information being contained in the explicit task graph. However, while these experimental results show that ADL permits GridRPC applications whit irregular algorithms to obtain high performance in conjunction with SmartGridRPC model; since the examples used are artificial, we think that more investigation is needed with additional experimental analysis using a real life application where the main algorithm is irregular.

The work presented in this thesis shows that is possible to implement iterative, conditional and adaptive applications in SmartGridRPC with good results by implementing two techniques. The first one is by slightly changing the code to make the algorithm static (as shown in the example of the iterative algorithm), the second one is by describing the algorithm and thus the task graph in ADL (as shown in the conditional and adaptive algorithms examples).

Future work for the ADL project could include modifying the language and the compiler output to easily interface with other GridRPC middlewares. This should be possible by implementing a method that allows the integration between language and middleware, similar to *grpc_map*, in the other projects. While the language, other than reading the task information from the gsIDL file, could also read this information from a Ninf-G IDL. Another option is to let the application programmer include the information about the task in a special construct similar to a module. Furthermore, as we have previously mentioned, since the information needed to generate a task graph is similar to the one needed to generate a workflow, ADL could be used as workflow generator for workflow management systems.

We think that another important future step for the ADL project that is worth investigating is to allow the language and the compiler to directly generate the GridRPC application code. The idea is to execute the task graph generated by ADL using only the *grpc_map* function in the client code without the need to write the subsequent GridRPC methods, in a similar but more dynamic way than is done in DIET with the XML workflow. The language will need further keywords and special statements in order to do this. These new features would highlight the changes in the flow of execution of the task graph but would also generate the normal flow of execution in the client code. This new functionality could permit a scientific user to easily generate distributed scientific applications since the expressiveness of the ADL syntax, which easily highlights parallel remote tasks and the input output data dependencies between tasks, promotes the development of task parallel applications.

# Bibliography


[1] EGEE Project. gLite. `http://glite.web.cern.ch/glite/`, June 2009.

[2] Open Grid Forum. `http://www.ogf.org/`, June 2009.

[3] XSL Transformations (XSLT) version 1.0. `www.w3.org/TR/xslt`, August 2009.

[4] K. Aida, Y. Futakata, and S. Hara. High-Performance Parallel and Distributed Computing for the BMI Eigenvalue Problem. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 177, Washington, DC, USA, 2002. IEEE Computer Society.

[5] K. Aida and T. Osumi. A Case Study in Running a Parallel Branch and Bound Application on the Grid. In *SAINT '05: Proceedings of the The 2005 Symposium on Applications and the Internet*, pages 164–173, Washington, DC, USA, 2005. IEEE Computer Society.

[6] A. Amar, R. Bolze, A. Bouteiller, P. K. Chouhan, A. Chis, Y. Caniou, E. Caron, H. Dail, B. Depardon, F. Desprez, J.-S. Gay, G. Le Mahec, and A. Su. DIET: New Developments and Recent Results. In L. et al. (Eds.), editor, *CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar2006)*, number 4375 in LNCS, pages 150–170, Dresden, Germany, August 28-29 2006. Springer.

[7] A. Amar, R. Bolze, Y. Caniou, E. Caron, A. Chis, F. Desprez, B. Depardon, J.-S. Gay, G. Le Mahec, and D. Loureiro. Tunable scheduling in a GridRPC framework. *Concurr. Comput. : Pract. Exper.*, 20(9):1051–1069, 2008.

[8] K. Amin, G. von Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Work.ow System. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 7*, page 70210.3, Washington, DC, USA, 2004. IEEE Computer Society.

[9] G. Antoniu, E. Caron, F. Desprez, A. Fèvre, and M. Jan. Towards a Transparent Data Access Model for the GridRPC Paradigm. In S. A. et al. (Eds), editor, *HiPC'2007. 14th International Conference on High Performance Computing.*, number 4873 in

LNCS, pages 269–284, Goa. India, December 17-20 2007. Springer Verlag Berlin Heidelberg.

[10] D. Arnold, H. Casanova, and J. Dongarra. Innovations of the NetSolve Grid Computing System. *Concurr. Comput. : Pract. Exper.*, 14(13-15):1457–1479, 2002.

[11] D. C. Arnold, D. Bachmann, and J. Dongarra. Request Sequencing: Optimizing Communication for the Grid. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1213–1222, London, UK, 2000. Springer-Verlag.

[12] D. C. Arnold and J. Dongarra. The NetSolve Environment: Progressing Towards the Seamless Grid. In *ICPP '00: Proceedings of the 2000 International Workshop on Parallel Processing*, page 199, Washington, DC, USA, 2000. IEEE Computer Society.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[14] M. Beck, D. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore, G. Obertelli, J. Plank, M. Swany, S. Vadhiyar, and R. Wolski. Middleware for the use of storage in communication. *Parallel Comput.*, 28(12):1773–1787, 2002.

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

[16] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[17] T. Brady, J. Dongarra, M. Guidolin, A. Lastovetsky, and K. Seymour. SmartGridRPC: The New RPC Model for High Performance Grid Computing. Technical Report UCD-CSI-2009-10, School of Computer Science and Informatics, University College Dublin, October 2009.

[18] T. Brady, M. Guidolin, and A. Lastovetsky. Experiments with SmartGridSolve: Achieving Higher Performance by Improving the GridRPC Model. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid 2008)*, Tsukuba, Japan, 29 September - 01 October 2008. IEEE Computer Society.

[19] T. Brady, E. Konstantinov, and A. Lastovetsky. SmartNetSolve: High Level Programming System for High Performance Grid Computing. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, 25-29 April 2006.

[20] Y. Caniou, E. Caron, H. Courtois, B. Depardon, and R. Teyssier. Cosmological Simulations using Grid Middleware. In *Fourth High-Performance Grid Computing Workshop (HPGC'07)*, Long Beach, California, USA, March 26 2007. IEEE.

[21] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 198, Washington, DC, USA, 2003. IEEE Computer Society.

[22] E. Caron, B. DelFabbro, F. Desprez, E. Jeannot, and J.-M. Nicod. Managing data persistence in network enabled servers. *Sci. Program.*, 13(4):333–354, 2005.

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

[24] E. Caron, F. Desprez, and D. Loureiro. All-in-one Graphical Tool for the management of DIET a GridRPC Middleware. In *CoreGRID Workshop on Grid Middleware (in conjunction with OGF'23)*, Barcelona, Spain, June 2-6 2008. To appear.

[25] H. Casanova and J. Dongarra. NetSolve: a network server for solving computational science problems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 40, Washington, DC, USA, 1996. IEEE Computer Society.

[26] H. Casanova, M. Kim, J. S. Plank, and J. J. Dongarra. Adaptive Scheduling for Task Farming with Grid Middleware. *Int. J. High Perform. Comput. Appl.*, 13(3):231–240, 1999.

[27] P. K. Chouhan, H. Dail, E. Caron, and F. Vivien. Automatic Middleware Deployment Planning On Clusters. *Int. J. High Perform. Comput. Appl.*, 20(4):517–530, 2006.

[28] L. Choy, O. Delannoy, N. Emad, and S. G. Petiton. Federation and Abstraction of Heterogeneous Global Computing Platforms with the YML Framework. *Complex, Intelligent and Software Intensive Systems, International Conference*, 0:451–456, 2009.

[29] G. Chun, H. Dail, H. Casanova, and A. Snavely. Benchmark probes for grid assessment. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 276, 2004.

[30] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1021–1037, 2006.

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

[32] H. Dail and F. Desprez. Experiences with Hierarchical Request Flow Management for Network-Enabled Server Environments. *International Journal of High Performance Computing Applications*, 20(1):143–157, 2006. Sage Science Press.

[33] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *Proceedings of 2nd EUROPEAN ACROSS GRIDS CONFERENCE*, Nicosia, Cyprus, 2004.

[34] B. Del-Fabbro, D. Laiymani, J.-M. Nicod, and L. Philippe. DTM: a service for managing data persistency and data replication in network-enabled server environments: Research Articles. *Concurr. Comput. : Pract. Exper.*, 19(16):2125–2140, 2007.

[35] O. Delannoy. *YML: A scientific Workflow for High Performance Computing*. PhD thesis, University of Versailles Saint-Quentin, September 2008.

[36] O. Delannoy, F. Emad, and S. Petiton. Workflow Global Computing with YML. In *GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 25–32, Washington, DC, USA, 2006. IEEE Computer Society.

[37] O. Delannoy and S. Petiton. A Peer to Peer Computing Framework: Design and Performance Evaluation of YML. In *ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 362–369, Washington, DC, USA, 2004. IEEE Computer Society.

[38] F. Desprez et al. *DIET User's Mannual, Version 2.3*. INRIA, ENS-Lyon, UCBL, July 2008.

[39] F. Desprez and E. Jeannot. Adding Data Persistence and Redistribution to NetSolve. Research Report 2001-39, Ecole Normale Superieure de Lyon, 46 Allee d'Italie, 69364, Lyon, France, 2001.

[40] F. Desprez and E. Jeannot. Improving the GridRPC Model with Data Persistence and Redistribution. In *ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 193–200, Washington, DC, USA, 2004. IEEE Computer Society.

[41] J. Dongarra, K. Seymour, and A. YarKhan. *Users' Guide to GridSolve, Version 0.15*. University of Tennessee, Knoxville, TN, USA, 2006.

[42] J. Dongarra, K. Seymour, and A. YarKhan. GridSolve: The Evolution of A Network Enabled Solver. In *IFIP International Federation for Information Processing*, volume 239, pages 215–224. Springer Boston, November 2007.

[43] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, Jr., and H.-L. Truong. ASKALON: a tool set for cluster and Grid computing: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):143–169, 2005.

[44] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an Abstract Grid Workflow Language. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 676–685, Washington, DC, USA, 2005. IEEE Computer Society.

[45] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A Generic Global Computing System. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 582, Washington, DC, USA, 2001. IEEE Computer Society.

[46] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[47] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[48] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, Washington, DC, USA, 2002. IEEE Computer Society.

[49] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[50] M. Frumkin and R. F. Van der Wijngaart. NAS Grid Benchmarks: A Tool for Grid Space Exploration. *Cluster Computing*, 5(3):247–255, 2002.

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

[52] J. Grosch. Cocktail Toolbox. `http://www.cocolab.com/en/cocktail.html`, September 2009.

[53] J. Grosch and H. Emmelmann. *A Tool Box for Compiler Construction*. CoCoLab, Achern, Germany, 1990.

[54] M. Guidolin and A. Lastovetsky. ADL: An Algorithm Definition Language for SmartGridSolve. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid 2008)*, Tsukuba, Japan, 29 September - 01 October 2008. IEEE Computer Society.

[55] M. Guidolin and A. Lastovetsky. Grid-Enabled Hydropad: a Scientific Application for Benchmarking GridRPC-Based Programming Systems. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium (IPDPS 2009)*, Rome, Italy, 25-29 May 2009. IEEE Computer Society.

[56] M. Hardt, K. Seymour, J. Dongarra, M. Zapf, and N. V. Ruiter. Interactive Grid-Access Using Gridsolve and Giggle. *Computing and Informatics*, 27(2):233–248, 2008.

[57] R. L. Henderson. Job Scheduling Under the Portable Batch System. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

[58] R. Higgins and A. Lastovetsky. Managing the Construction and Use of Functional Performance Models in a Grid Environment. Rome, Italy, 25/05/2009 2009.

[59] R. Hockney and J. Eastwood. *Computer Simulation Using Particles*. McGraw Hill, New York, 1981.

[60] E. Jeannot and G. Monard. Computing Molecular Potential Energy Surface with DIET. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, pages 286–291, Washington, DC, USA, 2005. IEEE Computer Society.

[61] A. Lastovetsky. Adaptive parallel computing on heterogeneous networks with mpC. *Parallel Comput.*, 28(10):1369–1407, 2002.

[62] A. Lastovetsky and R. Reddy. HeteroMPI: towards a message-passing library for heterogeneous networks of computers. *J. Parallel Distrib. Comput.*, 66(2):197–220, 2006.

[63] A. Lastovetsky, R. Reddy, and R. Higgins. Building the Functional Performance Model of a Processor. Dijon, France, April 23-27 2006 2006. ACM, ACM.

[64] C. Lee and D. Talia. Grid programming models: Current tools, issues and directions. In *In Grid Computing: Making The Global Infrastructure a Reality*, pages 555–578. Wiley, 2003.

[65] Y. Li, J. Dongarra, K. Seymour, and A. YarKhan. Request Sequencing: Enabling Workflow for Efficient Problem Solving in GridSolve. In *GCC '08: Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing*, pages 449–458, Washington, DC, USA, 2008. IEEE Computer Society.

[66] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, June 1988.

[67] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Technical Report GFD.52, Open Grid Forum, 2005.

[68] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of Ninf: towards a global computing infrastructure. *Future Gener. Comput. Syst.*, 15(5-6):649–658, 1999.

[69] H. Nakada, H. Takagi, S. Matsuoka, U. Nagashima, M. Sato, and S. Sekiguchi. Utilizing the Metaserver Architecture in the Ninf Global Computing System. In *HPCN Europe 1998: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 607–616, London, UK, 1998. Springer-Verlag.

[70] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[71] K. Osawa and K. Aida. Speed-up Techniques for Computation of Markov Chain Model to Find an Optimal Batting Order. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 315, Washington, DC, USA, 2005. IEEE Computer Society.

[72] M. Sato, T. Boku, and D. Takahashi. OmniRPC: a Grid RPC ystem for Parallel Programming in Cluster and Grid Environment. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 206, Washington, DC, USA, 2003. IEEE Computer Society.

[73] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure. In *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 491–502, London, UK, 1997. Springer-Verlag.

[74] K. Seymour, C. Lee, F. Desprez, H. Nakada, and Y. Tanaka. The End-User and Middleware APIs for GridRPC. In *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12*, Brussels, Belgium, September 2004.

[75] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In

*Proceedings of the Third International Workshop on Grid Computing (Grid 2002)*, pages 274–278, London, UK, 2002. Springer-Verlag.

[76] A. Snavely, G. Chun, H. Casanova, R. F. Van der Wijngaart, and M. A. Frumkin. Benchmarks for grid computing: a review of ongoing efforts and future directions. *SIGMETRICS Perform. Eval. Rev.*, 30(4):27–32, 2003.

[77] A. Takefusa, S. Matsuoka, H. Ogawa, H. Nakada, H. Takagi, M. Sato, S. Sekiguchi, and U. Nagashima. Multi-client LAN/WAN performance analysis of Ninf: a high-performance global computing system. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–23, New York, NY, USA, 1997. ACM.

[78] H. Takemiya, Y. Tanaka, S. Sekiguchi, S. Ogata, R. K. Kalia, A. Nakano, and P. Vashishta. Sustainable adaptive grid supercomputing: multiscale simulation of semiconductor processing across the pacific. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 106, New York, NY, USA, 2006. ACM.

[79] 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*, 1(1):41–51, 2003. Springer.

[80] Y. Tanaka, H. Takemiya, H. Nakada, and S. Sekiguchi. Design, Implementation and Performance Evaluation of GridRPC Programming Middleware for a Large-Scale Computational Grid. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 298–305, Washington, DC, USA, 2004. IEEE Computer Society.

[81] Y. Tanimura, H. Nakada, Y. Tanaka, and S. Sekiguchi. Design and Implementation of Distributed Task Sequencing on GridRPC. In *CIT '06: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*, page 67, Washington, DC, USA, 2006. IEEE Computer Society.

[82] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor: a distributed job scheduler. pages 307–350, 2002.

[83] Y. Yamamoto, H. Nakada, H. Shimodaira, and S. Matsuoka. Parallelization of Phylogenetic Tree Inference Using Grid Technologies. In *LSGRID*, pages 103–116, 2004.

[84] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra. Recent Developments in GridSolve. *International Journal of High Performance Computing Applications*, 20(1):131–142, 2006. Sage Science Press.

[85] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3-4):171–200, September 2005.

# Appendix A

# ADL - Grammar

This appendix presents the context free grammar of the ADL language. The chosen grammar consists of terminals, nonterminals, productions and a start symbol. They are presented using the following rules:

1. A production consists of a nonterminal (left side), an assignment symbol ($\rightarrow$) and a sequence of terminals and/or nonterminals (right side).
2. Different lines in a production means different choices in the production construction.
3. Terminals are represented by using monospace `strings`.
4. Nonterminals are represented by using italic *strings*.
5. In this appendix some abbreviations are used for the following common words: (i) expr for expression; (ii) stmt for statement; (iii) arg for argument; (iv) spec for specifier; and (v) oper for operator.
6. When a nonterminal is introduced and the relative production is not available in the current section, a reference number of the section containing the production is added. For example the production of the nonterminal symbol *start*$^{(A.1)}$ is in section A.1.

## A.1 Programs definition

*start* → *translation-unit*
*translation-unit* → *definition-seq*
*definition-seq* → *root-definition* (A.2)
| *root-definition* *definition-seq*

## A.2 Components definition

*root-definition* → *component-spec* (A.4.2) *root-declarator* *root-body*
*root-declarator* → *identifier* (A.6) ( )
| *identifier* ( *parameter-list* )
*parameter-list* → *parameter-declaration*
| *parameter-declaration* , *parameter-list*
*parameter-declaration* → *c-type-spec-list* (A.4.1) *declarator* (A.4.4)
*root-body* → { *section-stmt-seq* (A.3.4) }
| { *statement-list* (A.3) }
| { *statement-list* *section-stmt-seq* }
| { }

## A.3 Statements

*statement-list* → *statement*
| *statement-list* *statement*

*statement* → *compound-stmt* (A.3.1)
| *labeled-stmt* (A.3.2)
| *jump-stmt* (A.3.3)
| *declaration-stmt* (A.3.1)
| *parallel-stmt* (A.3.1)
| *expression-stmt* (A.3.1)
| *selection-stmt* (A.3.2)
| *iteration-stmt* (A.3.3)
| *call-stmt* (A.3.5)
| *inout-stmt* (A.3.6)

*block-stmt* → *compound-stmt*
| *jump-stmt*
| *parallel-stmt*
| *expression-stmt*
| *selection-stmt*
| *iteration-stmt*
| *call-stmt*

### A.3.1 Base Statements

*compound-stmt* → { *statement-list* (A.3) }
| { }

*expression-stmt* → *expression* (A.5) ;
| ;

*declaration-stmt* → *block-declaration* (A.4)

*parallel-stmt* → `parallel` *compound-stmt*

### A.3.2 Conditional Statements

*selection-stmt* → `if (` *expression*$^{(A.5)}$ `)` *block-stmt*$^{(A.3)}$
| `if (` *expression* `)` *block-stmt* `else` *block-stmt*
| `switch (` *expression* `)` *compound-stmt*$^{(A.3.1)}$

*labeled-stmt* → `case` *constant-expr*$^{(A.5)}$ `:`
| `default :`

### A.3.3 Iteration Statements

*iteration-stmt* → `while (` *expression*$^{(A.5)}$ `)` *block-stmt*$^{(A.3)}$
| `do` *block-stmt* `while (` *expression* `) ;`
| *f-spec* `(` *f-expr* `;` *f-expr* `;` *f-expr* `)` *block-stmt*

*f-spec* → `for`
| `parfor`

*f-expr* → *expression*
| $\varepsilon$

*jump-stmt* → `continue ;`
| `break ;`

### A.3.4 Section Statement

*section-stmt-seq* → *section-stmt*
| *section-stmt* *section-stmt-seq*

*section-stmt* → *identifier*$^{(A.6)}$ `:` *statement-list*$^{(A.3)}$
| *section-spec* `:` *statement-list*
| *section-spec* `:`

*section-spec* → `IFO`
| `algorithm`
| `component`
| `inout`

### A.3.5 Call Statement

*call-stmt* → *call-identifier call-arg* `->` *call-arg* `;`
| *call-identifier call-arg* `;`
*call-identifier* → *identifier*(A.6) `:`
| *function-expr*(A.5) `:`
*call-arg* → `( )`
| `(` *arg-expr-list-ellipsis*(A.5) `)`

### A.3.6 In & Out Statement

*inout-stmt* → *inout-definition* `;`
*inout-definition* → *inout-spec arg-expr-list*(A.5)
| *inout-spec*
*inout-spec* → `input`
| `output`

## A.4 Declarations

*block-declaration* → *c-declaration*(A.4.1)
| *component-declaration*(A.4.2)
| *ifo-declaration*(A.4.3)

### A.4.1 C declaration

*c-declaration* → *c-type-spec-list c-init-declarator-list* ;
*c-init-declarator-list* → *c-init-declarator*
| *c-init-declarator* , *c-init-declarator-list*
*c-init-declarator* → *declarator* (A.4.4)
| *declarator* = *c-initialiser*
*c-type-spec-list* → *c-type-spec*
| *c-type-spec c-type-spec-list*
*c-type-spec* → `char`
| `short`
| `int`
| `long`
| `float`
| `double`
| `signed`
| `unsigned`
| `const`
*c-initialiser-list* → *c-initialiser*
| *c-initialiser-list* , *c − initialiser*
*c-initialiser* → *constant-expr* (A.5)

### A.4.2 Component Declaration

*component-declaration* → *component-spec declarator-list* (A.4.4) ;
| *component-spec string* (A.6) *declarator-list* ;
*component-spec* → `function`(Future work)
| `task`(Future work)
| `module`

### A.4.3 IFO Declaration

*ifo-declaration* → *ifo-spec declarator-list*$^{(A.4.4)}$ ;
*ifo-spec* → *ifo-type-spec*
| *ifo-type-spec size-spec-list*
*ifo-type-spec* → `CHAR`
| `INTEGER`
| `FLOAT`
| `DOUBLE`
| `SCOMPLEX`
| `DCOMPLEX`
| `BYTE`
*size-spec-list* → *size-spec*
| *size-spec size-spec-list*
*size-spec* → `(` *constant-expr-list*$^{(A.5)}$ `)`
| `(` *constant-expr*$^{(A.5)}$ `)`

### A.4.4 Declarator

*declarator-list* → *declarator*
| *declarator* , *declarator-list*
*declarator* → *identifier*
| *declarator* `[` *constant-expr*$^{(A.5)}$ `]`
| *declarator* `[` *constant-expr-list*$^{(A.5)}$ `]`

## A.5 Expressions

*expression* → *assignment-expr*(A.5.6)
*constant-expr* → *conditional-expr*(A.5.6)
*constant-expr-list* → *constant-expr* : *constant-expr*
| *constant-expr* : *constant-expr-list*
*at-expr* → @
| @ *constant-expr*
| *unary-expr*(A.5.2) @ *constant-expr*
*arg-expr* → *constant-expr*
| *at-expr*
*arg-expr-list* → *arg-expr*
| *arg-expr-list* , *arg-expr*
*arg-expr-list-ellipsis* → *arg-expr-list*
| *arg-expr-list* , ...
| ...
*function-expr* → *identifier*(A.6) ( )
| *identifier* ( *arg-expr-list* )

### A.5.1 Base Expressions

*primary-expr* → *identifier*(A.6)
| *constant*(A.6)
| ( *expression*(A.5) )
*postfix-expr* → *primary-expr*
| *function-expr*(A.5)
| *postfix-expr* [ *constant-expr-list*(A.5) ]
| *postfix-expr* [ *constant-expr*(A.5) ]
| *postfix-expr* ++
| *postfix-expr* --

### A.5.2 Unary Expressions

*unary-expr* → *postfix-expr* (A.5.1)
| *unary-oper unary-expr*

*unary-oper* → `+`
| `-`
| `~`
| `!`
| `++`
| `--`

### A.5.3 Binary Expressions

*mult-expr* → *unary-expr* (A.5.2)
| *mult-expr* `*` *unary-expr*
| *mult-expr* `/` *unary-expr*
| *mult-expr* `%` *unary-expr*

*add-expr* → *mult-expr*
| *add-expr* `-` *mult-expr*
| *add-expr* `+` *mult-expr*

*shift-expr* → *add-expr*
| *shift-expr* `<<` *add-expr*
| *shift-expr* `>>` *add-expr*

### A.5.4 Selection Expressions

*relational-expr* → *shift-expr* (A.5.3)
| *relational-expr* `<` *shift-expr*
| *relational-expr* `>` *shift-expr*
| *relational-expr* `<=` *shift-expr*
| *relational-expr* `>=` *shift-expr*

*equality-expr* → *relational-expr*
| *equality-expr* `==` *relational-expr*
| *equality-expr* `!=` *relational-expr*

### A.5.5 Logical Expressions

*and-expr* → *equality-expr*(A.5.4)
| *and-expr* `&` *equality-expr*

*exclusive-or-expr* → *and-expr*
| *exclusive-or-expr* `^` *and-expr*

*inclusive-or-expr* → *exclusive-or-expr*
| *inclusive-or-expr* `|` *exclusive-or-expr*

*logical-and-expr* → *inclusive-or-expr*
| *logical-and-expr* `&&` *inclusive-or-expr*

*logical-or-expr* → *logical-and-expr*
| *logical-or-expr* `||` *logical-and-expr*

### A.5.6 Top Expressions

*conditional-expr* → *logical-or-expr*(A.5.5)
| *logical-or-expr* `?` *conditional-expr* `:` *conditional-expr*

*assignment-expr* → *conditional-expr*
| *unary-expr*(A.5.2) *assignment-oper* *assignment-expr*

*assignment-oper* → `=`
| `*=`
| `/=`
| `%=`
| `-=`
| `<<=`
| `>>=`
| `&=`
| `~=`
| `|=`

## A.6 Base Nonterminals

*identifier* → `IDENTIFIER-TOKEN`

*constant* → `CONSTANT-TOKEN`

*string* → `STRING-TOKEN`

... so long, and thanks for all the Guinness!