ADL: An Algorithm Definition Language for SmartGridSolve

Michele Guidolin, Alexey Lastovetsky
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
{ michele.guidolin, alexey.lastovetsky }@ucd.ie

Abstract

SmartGridSolve is an extension of GridSolve that expands the single task map and client-server model of GridRPC by implementing server to server communication and the mapping of a group of tasks.

In order to accomplish this functionality SmartGridSolve needs a task graph that highlights tasks' execution order, communication volume and computation volume for a given group of tasks.

This work presents the Algorithm Description Language (ADL), a language that helps the application programmer to easily specify a task graph for any given algorithm. The language is modular, it has a well-defined structure and its syntax is similar to "C" language.

This poster paper introduces a trivial example of SmartGridSolve application and the use of ADL to build the relative task graph with an overview of the language syntax.

1. Introduction

GridRPC [5] is a standard promoted by the Open Grid Forum that provides a simple remote procedure call (RPC) mechanism for a Grid environment. Using the GridRPC API an application programmer can easily specify different tasks to be executed remotely.

The high level programming model of GridRPC offers the possibility to smoothly design grid applications, with asynchronous parallel tasks, by hiding the complexity of the interface to the Grid from the application programmer. The middleware system is responsible for individually mapping a task to a single server in the Grid and communicating the data between the server and the client computer. 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. A number of grid middleware systems are GridRPC compliant including GridSolve [7], Ninf-G [6] and DIET [3].

SmartGridSolve [1], previously implemented as SmartNetSolve [2], is an extension of GridSolve that expands the single task map and client-server model of GridRPC. This is done by implementing server to server communication and the mapping of groups of tasks. The collective mapping of tasks, with the possibility to use a fully connected network, helps SmartGridSolve find an optimal mapping solution that can exploit fully a Grid environment.

SmartGridSolve, in order to map a group of tasks, needs to build a task graph of the undergoing task calls present in the grid application. 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 [1].

SmartGridSolve introduces a new API that automatically generates the task graph. This API works by iterating twice through the application code that contains the task calls to be mapped collectively. On the first iteration of 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 of the code, after producing the mapping by using the new task graph, the code is normally executed and the task calls are performed.

One advantage of this method is that the application programmer only has to make minimal modifications to the original GridRPC code. Unfortunately this approach has the restriction that a task graph is not always generated for every kind of algorithm. 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. The application programmer can choose to create the task graph from a smaller block of code to avoid this problem but the resulting group of tasks to be mapped will generate a less optimal execution.

To solve this problem we designed the Algorithm Definition Language (ADL) and the respective compiler that allows the application programmer to easily describe all kinds

of algorithms and generate the corresponding task graph. In the situation where the output of a remote task call can change the flow of execution, the application programmer can know the best way to generate the task graph. The main goal of ADL is to give a powerful tool to the application programmer that can help implement a SmartGridSolve application with the best mapping and execution possible.

This work is outlined as follows, section 2 presents a trivial SmartGridSolve application that shows how the automatic task graph generator works and the restriction previously detailed. Section 3 shows how to use ADL to describe the example application and gives an overview of the language syntax. Section 4 details how the ADL compiler integrates with the example application using SmartGridSolve. Finally in section 4 we present the task graph and how ADL can help the application programmer to obtain an optimal mapping solution for the given example.

2. SmartGridSolve Example

In this section we introduce a pseudo GridRPC application and its implementation in SmartGridSolve. This application is specifically designed to show the restrictions of the automatic task graph generation method.

The example application uses the GridRPC APIs *grid_call* and *grid_call_async* to execute respectively a blocking and an asynchronous remote call. The first argument of both APIs is the handler of the task executed, the second is the session ID of the remote call while the following arguments are the parameters of the task. Furthermore the code uses the method *grid_wait_all* to block the execution until any previously issued asynchronous request has completed.

Table 1. GridRPC application

```
...
 grpc_call_async(T1_hnd,&id1,VA0,VB0,VC0);
 grpc_call_async(T1_hnd,&id2,VA1,VB1,VC1);
 grpc_wait_all();
 grpc_call(T2_hnd,&id3,VC0,VC1,VD);
 if(F1(VD)<0){
   grpc_call_async(T3_hnd,&id1,VC0,VD,VC0);
   grpc_call_async(T3_hnd,&id2,VC1,VD,VC1);
   grpc_wait_all();
}
...
```

The pseudo application consists of three tasks, *T1*, *T2*, *T3*, which have all the parameters as inputs except the last one that is an output. The objects used in the application are all vectors of double precision numbers. In table 1 it is possible to see the simple GridRPC version of the code. At the beginning two parallel remote *T1* calls are executed. The respective output objects, *VC0* and *VC1*, are then used as inputs of the remote task *T2*. The output of the last task, *VD*, is then computed in a local function *F1*. If the function returns a value less then zero the code executes two parallel remote calls of the task *T3*.

SmartGridSolve introduces a new API, *gs_smart_map*, that defines the area of code in which all the task calls contained within it are mapped as a group of tasks in a fully connected network. Table 2 shows how the new API can be used in the example application. The task calls inside the curly brackets block will be mapped as a group. The first parameter of the *gs_smart_map* method gives the application programmer the possibility to choose which mapping function to use. The second one indicates the tool that will be used to generate the task graph while the following parameters depend on the previous choice.

Table 2. Example of the optimal SmartGridSolve mapping method in the application

```
...
gs_smart_map("ex_map",auto){
   grpc_call_async(T1_hnd,&id1,VA0,VB0,VC0);
   grpc_call_async(T1_hnd,&id2,VA1,VB1,VC1);
   grpc_wait_all();
   grpc_call(T2_hnd,&id3,VC0,VC1,VD);
   if(F1(VD)<0){
     grpc_call_async(T3_hnd,&id1,VC0,VD,VC0);
     grpc_call_async(T3_hnd,&id2,VC1,VD,VC1);
     grpc_wait_all();
  }
}
...
```

The code example in table 2 uses the automatic task generator to build the task graph. At run-time when the *gs_smart_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 normally on the respective server specified in the mapping solution [1].

The choice of which task calls to include in the group of tasks to be mapped can influence the total execution time of the application. If the task graph contains more elements, SmartGridSolve has more opportunity to fine tune the mapping solution and consequently to minimise the data communication volume and the total computation time of the group of tasks.

The *gs_smart_map* block chosen in table 2, that includes all the task calls to be mapped, is the one that gives more

options to the SmartGridSolve mapper to minimise the example application's execution time. Unfortunately the automatic task generator may not be able to create the task graph for this particular situation. On the first iteration the task calls are discovered but not executed, consequently the output objects don't contain any valid data. When in the first iteration the local function *F1* computes the output object VD, as a result the behaviour of the function could be unpredictable. This could cause the application to fail and stop the execution.

However the function *F1* may work in some situations which will result in an arbitrary return value. The conditional check of the value could produce a different flow of execution on the first iteration compared to the second iteration. Consequently the task graph generated may not represent the real tasks execution. SmartGridSolve can handle this situation, of a task being executed but not existing in the mapping solution, by reverting the executing method of a remote task to the original GridSolve one.

Table 3. Example of an alternative mapping method for the application

```
...
gs_smart_map("ex_map",auto){
    grpc_call_async(T1_hnd,&id1,VA0,VB0,VC0);
    grpc_call_async(T1_hnd,&id2,VA1,VB1,VC1);
    grpc_wait_all();
    grpc_call(T2_hnd,&id3,VC0,VC1,VD);
}
if(F1(VD)<0){
  gs_smart_map("ex_map",auto){
     grpc_call_async(T3_hnd,&id1,VC0,VD,VC0);
     grpc_call_async(T3_hnd,&id2,VC1,VD,VC1);
     grpc_wait_all();
  }
}
...
```

This arbitrary execution of the application is not acceptable for a stable system. A method to avoid this situation is to divide the area to map in smaller blocks as shown in table 3 for the example application. This approach reduces the size of the task graph and consequently means SmartGridSolve will produce a less optimal mapping and execution time.

3. ADL Example

A task call executed that is not in the mapping solution has a negative impact on the total computational time of the application. SmartGridSolve has to revert to using the original GridSolve client-server single task map method to execute this task. This worst case scenario can be statistically reduced in a way that the negative impact becomes insignificant compared to the the gain obtained by having a bigger group of tasks to map.

This situation suggests that code like the example in table 2, where an application programmer chooses a large area to map with *gs_smart_map* despite the risk that the mapping solution doesn't match the task calls, can become favourable. An application programmer, that has good knowledge of the application, can estimate in advance the return value of the *F1* function and consequently guess the more likely flow of execution. As previously indicated the automatic task builder has some restrictions for this particular type of application where the flow of execution depends on task call outputs. Algorithm Definition Language (ADL), a language to easily specify task graphs, was designed to give more options to the application programmer for solving this kind of problem.

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, however flexible enough to generate the task graph for any kind of algorithm. Another ADL feature is that it is user-friendly and easy to write. The language syntax is similar to *C* language. It uses different modules to define an algorithm and each module, to simplify further the reading, is divided in well defined zones. A zone specifies a characteristic of the algorithm. Table 4 shows the ADL module that describes the example application from the previous section.

Table 4. ADL example

```
module  example (int size, int cond) {
  component:
    task "file.idl"      T1,T2,T3;
  IFO:
    DOUBLE(size)         VA[2],VB[2],VC[2],VD;
  algorithm:
    parallel{
        T1:(VA[0],VB[0]) ->(VC[0]);
        T1:(VA[1],VB[1]) ->(VC[1]);
    }
     T2:(VC[0],VC[1]) ->(VD);
    client:(VD) ->();
    if (cond){
       parfor(int i=0;i<2;i++){
          T3:(VC[i],VD) ->(VC[i]);
       }
    }
}
```

The data objects *VA*,*VB*, *VC*, *VD* in the application are important for generating the task graph. The communication time and computational time of the application will change depending on the size and type of these objects. In

ADL we reference an object that is used by a remote task and can be moved anywhere on the Grid as an Identify Flying Object (IFO). The data objects declaration is made in the *IFO* zone and it 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 correlates 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. The value inside the pair specifies the number of elements for that dimension. As in the example application the IFOs defined in table 4 are vectors of double precision numbers. The sizes of the vectors depend on the value of the parameter *size*.

The *component* zone includes the declaration of the tasks used in the algorithm like *T1*, *T2*, *T3* for the example application. The ADL language needs to provide the compiler with some specific information about a task used. The ADL compiler, for each task, requires the number and type of input/output arguments and the eventual computational complexity of the task. All this information can be provided "ad hoc" by the application programmer or retrieved from a GridSolve Interface Definition Language (gsIDL) file. The gsIDL is the mechanism through which GridSolve and SmartGridSolve enables normal function methods to be invoked remotely on a Grid environment [4]. The possibility to chose a gsIDL file may be convenient for already existing GridSolve applications, while in the case of a new application, the programmer can directly specify the information needed by using ADL.

In the *algorithm* zone of table 4 is possible to see how a task call is described in ADL. A remote call is composed of two parts, divided by a semicolon. In the first part there is the name of the task called (e.g. *T1*), followed by an eventual list of parameters needed. In the second part there is the list of IFOs used as task inputs, e.g. *VA* and *VB* for task *T1*, follow by an arrow symbol and the list of output IFOs (e.g. *VC*). This task call syntax is made in a way that easily highlights the parameters passed and the IFOs used as inputs and outputs of a task.

The *algorithm* zone in the example ADL module describes the flow of execution of the application. The use of the keywords *parallel* and *parfor* indicates that the remote calls inside the curly brackets are 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 a local execution. The ADL compiler, to generate the task graph, doesn't need to know the value of an IFO but instead where it is used. Consequently in the case of a local computation, ADL requires only the information from the IFOs used as inputs and outputs. In table 4 the client task, *F1*, has only *VD* as an input and no output.

In the ADL description of the algorithm, the conditional construct instead of checking the return value of the function *F1* (as in the example application), checks the value of the parameter *cond*. The value of *cond* will influence the flow of execution and consequently the task graph generated. Parameters in ADL are not only used to determine the flow of an 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.

4. ADL and Task Graph

In figure 1 it is possible to see how the ADL module is compiled and integrated with the GridRPC example application. The task graph used by SmartGridSolve is not generated directly by compiling the ADL code. For the given module the ADL compiler will yield a *C* file that has to be compiled within the client application. The produced *C* file contains the code to generate the task graph. This code will be executed at run-time by the *gs_smart_map* API. As previously indicated the ADL compiler needs some specific information for each task call used in the module compiled. This information can be provided by the application programmer or it can be retrieved from a gsIDL file as for example in table 4 and figure 1.

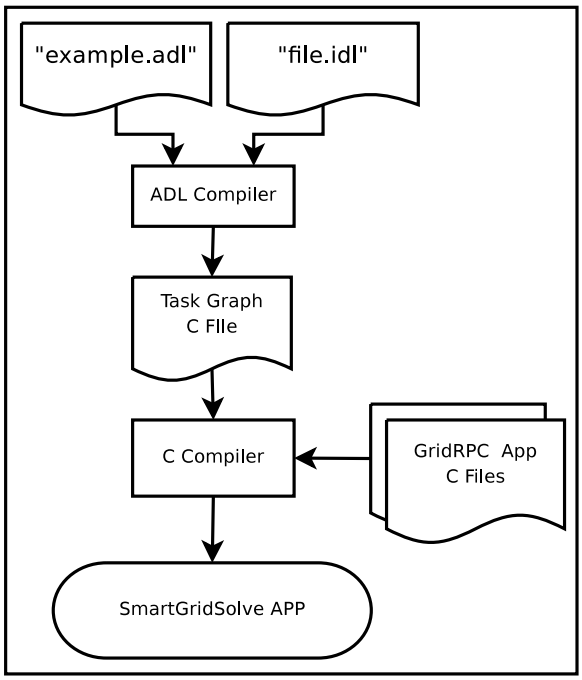

Figure 1. Use of ADL compiler

In table 5 it is possible to see how to use the *gs_smart_map* API with ADL to build the task graph. The first argument of the API is the same as in the example in table 2. The second argument, instead of the keyword *auto*,

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. The final arguments in the *gs_smart_map* API match the parameters of the given ADL module.

Table 5. Example of ADL use in the application through SmartGridSolve API

```
gs_smart_map("ex_map", ADL,"example",size, 1){
    grpc_call_async(T1_hnd,&id1,VA0,VB0,VC0);
    grpc_call_async(T1_hnd,&id2,VA1,VB1,VC1);
    grpc_wait_all();
    grpc_call(T2_hnd,&id3,VC0,VC1,VD);
    if(F1(VD)<0){
      grpc_call_async(T3_hnd,&id1,VC0,VD,VC0);
      grpc_call_async(T3_hnd,&id2,VC1,VD,VC1);
      grpc_wait_all();
   }
}
```

The run-time execution of the *gs_smart_map* function is different from the case of the automatic method. The task graph is built and the mapping solution generated directly when the API is called. Consequently the code inside the parenthesis block is iterated only once while the task calls are executed normally on the server specified in the mapping solution.

A programmer, who has a good understanding of the application, can influence the task graph generated to match, as closely as possible, the task calls executed. He can do so by changing the value of the parameter *cond* passed through the last argument of *gs_smart_map*. In table 5 the application programmer is guessing that the conditional check of the function *F1* return value is positive. If this assumption is true in the majority of cases, SmartGridSolve will normally generate an optimal mapping solution and consequently minimise the total execution time of the application.

The task graph generated from the code of table 5 is illustrated in figure 2. The rectangles in the graph represent remote tasks, the diamonds represent the client computation and the circles represent the IFOs. 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. 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 an IFO and the computational complexity of a task. These are correlated to the value of the module parameter *size* passed through the second to last argument of the *gs_smart_map* method. In a task graph all the input IFOs, that are not generated by a remote task, are retrieved from the client. All the output IFOs that are not used by a task call, are sent back to the client.

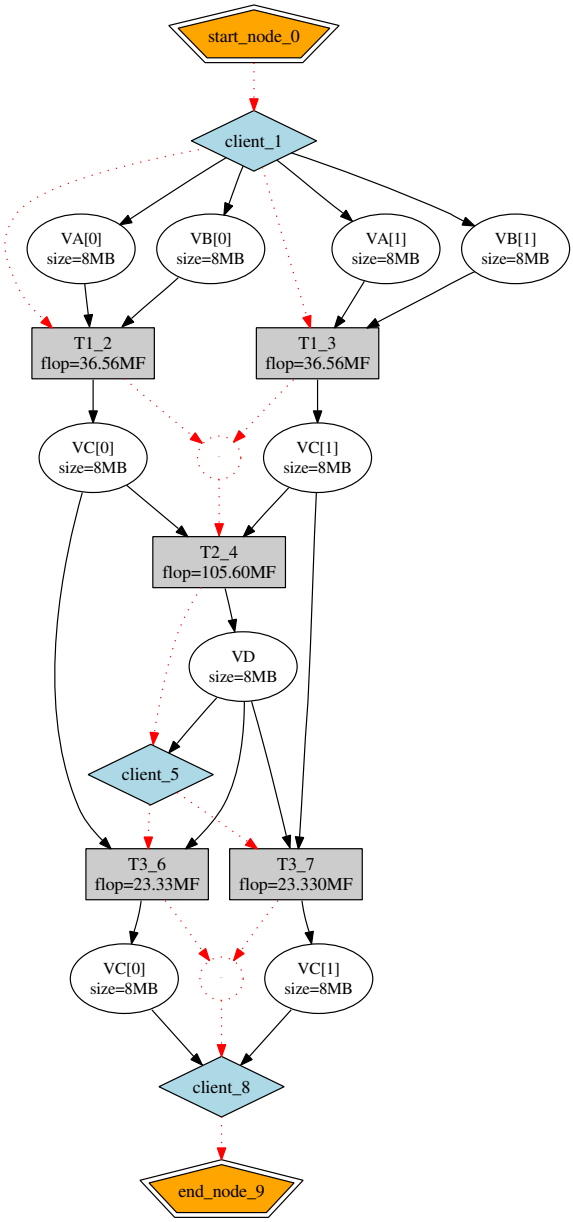

Figure 2. The task graph generated from the ADL module example.

The node *client_5* in the task graph represents the local computation of function *F1*. If, during the execution of the application, the conditional statement that checks the return value of the function is negative, the task graph underneath the node *client_5* will not be used. In this particular case, despite the fact that SmartGridSolve does not use all the task graph, the total execution time of the application will be similar to the one obtained if the task graph was gener-

ated by fine grained mapping (e.g. example in table 3). Furthermore, if the conditional statement is true, it is possible to see that coarse grained mapping is more favourable. In this case the outputs of tasks *T1* and *T2* can be sent directly to the servers that execute the parallel *T3* tasks without the need to pass trough the client as would be the case using fine grained mapping.

One can see from this example that the ADL Language permits the application programmer to map a large group of tasks where the automatic task graph generator would not work. This allows SmartGridSolve to find an optimal mapping solution that can minimise the total communication volume, as shown in the previous example, and the task computational time. Consequently minimising the total execution time of the application.

5. Conclusion

We have presented in this paper the specifications of the ADL language and its compiler. One of the goals of ADL is to overcome the restriction that the automatic task builder exhibits on applications where the flow of execution depends on task call outputs. We demonstrate that the ADL language overcomes this limitation and permits the application programmer to use SmartGridSolve, with an optimal mapping solution, for any kind of GridRPC application. This work was supported by the Science Foundation Ireland.

References

[1] 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.

[2] 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. IEEE Computer Society.

[3] 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.

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

[5] 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 *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 274–278, London, UK, 2002. Springer-Verlag.

[6] 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.

[7] 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.