How Algorithm Definition Language (ADL) Improves the Performance of SmartGridSolve Applications

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

Technical Report UCD-CSI-2009-06
July, 2009

Abstract

In this paper, we study the importance of languages for the specification of algorithms in high performance Grid computing. We present one such language, the Algorithm Definition Language (ADL), designed and implemented for the use in conjunction with SmartGridSolve. We demonstrate that the use of this type of language can significantly improve the performance of Grid applications. We discuss how ADL can be used to improve the execution of some typical algorithms that use conditional statements, iterative computations and adaptive methods. We present experimental results demonstrating significant performance gains due to the use of ADL.

1. Introduction

Scientific numerical simulations typically need large amount of computational resources during their execution. Many scientific applications use Grid computing because it provides an easy way to gather computational resources, whether local or geographically distributed, that can be pooled together to solve large problems. An application to be executed in a Grid environment needs to logically divide its core algorithm into a group of tasks that can be executed remotely. GridRPC [6] is a standard promoted by the Open Grid Forum that provides a simple remote procedure call (RPC) mechanism to easily execute tasks in a Grid environment. Using the GridRPC API an application programmer can easily specify different tasks to be executed remotely. GridRPC works by individually mapping a task to a single server in the Grid and communicating the data between the client machine and the remote server. 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 [8], Ninf-G [7] and DIET [3].

In order to optimally minimise the total execution time of an application, a Grid middleware would require the full knowledge of all the tasks executed in the application's algorithm. A task graph, a direct acyclic graph (DAG) structure, can be used to fully represent the algorithm. 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 computational volume. This information is essential to choose the best server to execute a task and to minimise the amount of data movement in a Grid.

SmartGridSolve [2] is a Grid middleware that uses task graphs of algorithms to achieve higher performance in scientific applications. SmartGridSolve implements the new SmartGridRPC [1] model that expands the individual task mapping and client-server communication model of GridRPC 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, permits SmartGridSolve to calculate an optimal mapping solution of an algorithm that can fully exploit the Grid environment.

SmartGridSolve uses the SmartGridRPC API to automatically generate the task graph from the application code. 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 SmartGridSolve to obtain an optimal mapping for this kind of algorithm and therefore to obtain better results than GridSolve, as it is in the case of the real-life astrophysical application discussed in [5].

Unfortunately, this approach has the restriction that a representative task graph may not be always 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.

A solution to this problem, which we propose, is to use algorithm specification languages to provide the application programmers with a means that would allow them to use their knowledge of the application to explicitly specify a task graph that best represents the run-time execution of the irregular algorithm. The mapping generated for this task graph can achieve a faster execution time than the single task mapping of GridRPC or the mapping of smaller groups of tasks. In this paper we present one such language: the Algorithm Definition Language (ADL). We discuss how the use of ADL can improve the performance of SmartGridSolve applications. ADL [4] is a new language designed to help an application programmer to easily specify a task graph for all kinds of algorithms.

This paper is outlined as follows. Section 2 presents three example algorithms that model real-life irregular algorithms that are common in many scientific applications and outlines the GridRPC and SmartGridRPC implementations of these example algorithms. In addition, it shows how the automatic task graph generator works and demonstrates its restrictions. Section 3 shows how the application programmer can use ADL to generate representative task graphs for the example algorithms. It presents a brief description of the language syntax and how ADL can be used in conjunction with SmartGridSolve. Section 4 presents results of experiments with the example applications showing that the use of ADL significantly improves their performance.

2. Irregular Algorithms and their GridRPC & SmartGridRPC implementations

Irregular algorithms, where the flow of task calls change dynamically, are important because they are common in many scientific simulations. In this section, we first introduce three trivial examples of irregular algorithms. These examples are models of typical algorithms used to solve real-life problems. Then, for each algorithm, we present its GridRPC and SmartGridRPC implementations. The SmartGridRPC implementations are then used to show the restrictions of the automatic task graph generation method. We also present some programming techniques that help somehow mitigate these restrictions.

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 1 shows a pseudocode of the example iterative algorithm.

Algorithm 1 Iterative

```
while Error ε is bigger than threshold t_ε do
  Compute Solution
  Compute Error ε
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 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.

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 3 shows a trivial example. In this example, the computational domain is divided in many sub-domains by finding the location where an error is greater than a

Algorithm 2 Conditional

```
Compute Solution
Compute Error ε
if Error ε is bigger than threshold t_ε then
   Correct Solution by ε
else
   Correct Solution by t_ε
end if
```

threshold. In the next step, each new sub-domain is used for further computation. A real-life example of the adaptive algorithm is the AMR (Adaptive Mesh Refinement) method.

Algorithm 3 Adaptive

```
Compute Solution in Domain D
Compute Error ε
Find Sub-domains S of D where ε > t_ε
for all S^i in S do
   Compute Solution in Sub-domain S^i
end for
```

2.1. GridRPC and SmartGridRPC implementations of the irregular algorithms

The following applications use the GridRPC methods *grpc_call* and *grpc_call_async* to execute blocking and asynchronous remote calls respectively. The first argument of both methods 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. The code uses the method *grpc_wait_all* to block the execution until any previously issued asynchronous request has completed. The SmartGridRPC examples utilise a new method, *grpc_map*. This method is used to define a specific area of code. All the task calls contained in this area are mapped as a group of tasks on to a fully connected network. The first parameter of the *grpc_map* method allows the application programmer to choose which mapping heuristic 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. SmartGridRPC defines another new function, *grpc_local*, which is used in the following examples as well. This API function is used to identify the area of code that contains local computations. The data objects used in the applications are all vectors of double precision numbers except when indicated different.

Iterative Algorithm Table 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 of the latter task, *D*, is then used as 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.

When the application is executed, the GridSolve middleware maps each *grpc_call_async* and *grpc_call* 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 executed remotely. At the end of the task execution, the data is communicated back to the client.

Table 1. Example of GridRPC implementation of an iterative algorithm

```
while(E>tE){
  grpc_call_async(T1_hnd,&id1,A0,B0,A0);
  grpc_call_async(T1_hnd,&id2,A1,B1,A1);
  grpc_wait_all();
  grpc_call(T2_hnd,&id3,A0,A1,D);
  F1(D,E);
}
```

Table 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 SmartGridSolve 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 done on the first iteration, but only on the execution phase, which is done on the second iteration.

Table 2. Example of SmartGridRPC implementation of an iterative algorithm

```
while(E>tE){
  grpc_map("ex_map",auto){
    for(i=0;i<nloops;i++){
      grpc_call_async(T1_hnd,&id1,A0,B0,A0);
      grpc_call_async(T1_hnd,&id2,A1,B1,A1);
      grpc_wait_all();
      grpc_call(T2_hnd,&id3,A0,A1,D);
      grpc_local(){
        F1(D,E);
      }
    }
  }
}
```

In table 2 it is possible to see that for this example the *grpc_map* method is used inside a while loop. A straightforward SmartGridRPC implementation would be to apply the *grpc_map* to the

whole loop, letting the middleware find the optimal mapping for the group of all 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 2 executes more iterations than the code of table 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.

Conditional Algorithm Table 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, the object *A* will be processed, otherwise it will be object *B*.

Table 3. Example of GridRPC implementation of a conditional algorithm

```
grpc_call_async(T1_hnd,&id1,A0,B0,C0);
grpc_call_async(T1_hnd,&id2,A1,B1,C1);
grpc_wait_all();
grpc_call(T2_hnd,&id3,C0,C1,D);
if(F1(D)>tE) {
  grpc_call_async(T3_hnd,&id4,C0,A0,A0);
  grpc_call_async(T3_hnd,&id5,C1,A1,A1);
  grpc_wait_all();
}
else {
  grpc_call_async(T3_hnd,&id4,C0,B0,B0);
  grpc_call_async(T3_hnd,&id5,C1,B1,B1);
  grpc_wait_all();
}
```

The ideal task graph, to be used to map this application, would represent the exact run-time execution of the remote tasks. As it has been 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. A technique, that allows the application programmers to still avail themselves of the group mapping in SmartGridSolve, is to break the whole code into smaller blocks suitable for the automatic task graph generation as show in table 4. This solution however produces many small groups of tasks. Therefore SmartGridSolve will minimise the execution time of these small groups rather than the whole algorithm, thus producing a less optimal mapping. Additionally, the data objects used between groups instead to be communicated directly between servers, they

will be communicated through the client machine. Figure 1 shows a possible ideal task graph for this example. This graph illustrates the data dependencies between the tasks executed before and after the conditional statement.

Table 4. Example of SmartGridRPC implementation of a conditional algorithm

```
grpc_map("ex_map",auto){
  grpc_call_async(T1_hnd,&id1,A0,B0,C0);
  grpc_call_async(T1_hnd,&id2,A1,B1,C1);
  grpc_wait_all();
  grpc_call(T2_hnd,&id3,C0,C1,D);
}
if(F1(D)>tE) {
  grpc_map("ex_map",auto){
    grpc_call_async(T3_hnd,&id4,C0,A0,A0);
    grpc_call_async(T3_hnd,&id5,C1,A1,A1);
    grpc_wait_all();
}
else {
  grpc_map("ex_map",auto){
    grpc_call_async(T3_hnd,&id4,C0,B0,B0);
    grpc_call_async(T3_hnd,&id5,C1,B1,B1);
    grpc_wait_all();
}
```

Adaptive Algorithm Table 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 a threshold value tE are passed to the task *T2*. Task *T2* checks C and outputs a vector of areas, S, where the solution error is greater than the threshold, and the number of such areas, n. Task *T3* 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 5. Example of GridRPC implementation of an adaptive algorithm

```
grpc_call(T1_hnd,&id1,A,B,C);
grpc_call(T2_hnd,&id2,C,tE,S,n);
for(int i=0;i<n;i++){
  grpc_call_async(T3_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();
}
```

In this example, the outputs of task *T2* 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, unpredictable will be not only the flow of execution 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 6.

Table 6. Example of SmartGridRPC implementation of an adaptive algorithm

```
grpc_map("ex_map",auto){
  grpc_call(T1_hnd,&id1,A,B,C);
  grpc_call(T2_hnd,&id2,C,tE,S,n);
}
for(int i=0;i<n;i++)
  grpc_map("ex_map",auto){
    grpc_call_async(T3_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();
  }
```

3. ADL and Task Graph

As shown in section 2, the automatic task graph construction method cannot generate representative task graphs for all the irregular algorithms. Therefore, SmartGridSolve cannot generate the best mapping solution possible for this type of algorithm. We have shown that this problem can be partially solved by mapping smaller code blocks, with some modification of the code of the application (table 2) or without it (tables 4 and 6).

In this paper, we present a comprehensive solution of this problem, which is the use of ADL, the Algorithm Definition Language. This new language and its compiler provide a means to the application programmer to explicitly specify a task graph for all kinds of algorithms. The programmers, with their knowledge of the application's algorithm, possess the right information to produce a representative task graph for any given group of tasks in the application. The ADL syntax is similar to the *C* language. The principal programming unit of this language is a module that is used to specify the group of tasks of an individual algorithm. The ADL compiler does not directly produce the task graph from the module but generates the code that is used at run-time to build the task graph. In this section, we show how the application programmer can use ADL to originate a representative task graph for the conditional and adaptive example algorithms.

Conditional Algorithm Table 7 shows the ADL module that describes the algorithm of our example conditional application. An ADL module is composed of a name, a list of parameters and a body. The body is divided in 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.

Table 7. ADL module of the conditional algorithm example

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

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. In the example application, the IFOs A, B, C, D, are vectors of double precision numbers. The sizes of these vectors depend on the value of the parameter *size*. Finally, the *algorithm* section describes the flow of execution of the application.

Table 7 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*), followed by a list of parameters needed. 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 7 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 4. Instead, in the ADL example in table 7, the module contains two conditional statements, in lines 15 and 18, 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 14 is used to avail this option. Without it, the compiler will consider the two branches as being executed sequentially, one after the other. The setting of different parameters 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. Example of ADL use in the conditional algorithm application through SmartGridRPC API

```
grpc_map("ex_map",ADL,cndalg,"%d,%d,%d",size,1,1){
  grpc_call_async(T1_hnd,&id1,A0,B0,C0);
  grpc_call_async(T1_hnd,&id2,A1,B1,C1);
  ...
  if(F1(D)>tE){
    ...
  }
  else{
    ...
  }
}
```

Table 8 shows how to use the *grpc_map* method with ADL to build the task graph. The first argument of the function is the same as in the example of table 4. 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 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 match the parameters of the given ADL module. In table 8, 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 while 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 in this situation.

The task graph generated from the code of table 8 is illustrated in figure 1. 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 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.

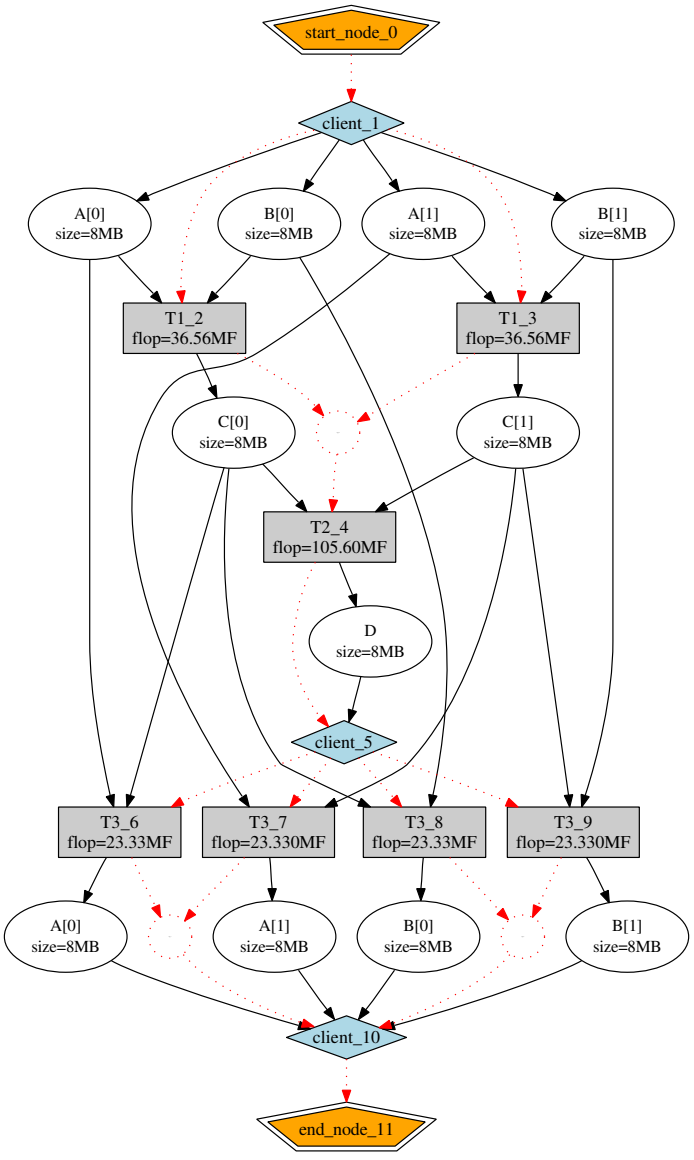

Figure 1. The task graph generated from the ADL module of the conditional algorithm example

Adaptive Algorithm Table 9 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 *T2*. Furthermore, the value of n is used to set the number of times that task *T3* is executed and hence the number of sub-vectors (AS, BS, CS) that are generated. The parameter *subsize* is a vector of integers that contains the sizes of each sub-vector object. Its dimension depends on the value of n.

Table 9. ADL module of the adaptive algorithm example

```
module adaptalg(int n, int size, int subsize[n]){
  component:
    task "adapt.idl"     T1,T2,T3;
  IFO:
    DOUBLE(size)          A,B,C;
    DOUBLE(subsize)       AS[n],BS[n],CS[n];
    INTEGER(n)            S;
  algorithm:
    T1:(A,B)->(C);
    T2:(C)->(S);
    parfor(int i=0;i<n;i++){
      T3:(S,A,B)->(AS[i],BS[i]);
      T1:(AS[i],BS[i])->(CS[i]);
    }
 }
```

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 10 shows the use of the ADL module *adaptalg* in the modified adaptive SmartGridSolve application. In this case, the application programmer chooses to generate a task graph with three *T3* calls. The size of data objects will be 1000 and the sub-vectors' sizes will be 200, 100 and 300. Figure 2 shows the generated task graph.

Table 10. Example of ADL use in the adaptive algorithm application through SmartGridRPC API

```
grpc_map("ex_map",ADL,adaptalg,"3,1000,{200,100,300}"){
  grpc_call(T1_hnd,&id1,A,B,C);
  grpc_call(T2_hnd,&id2,C,tE,S,n);
  for(int i=0;i<n;i++){
   grpc_call_async(T3_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();
  }
}
```

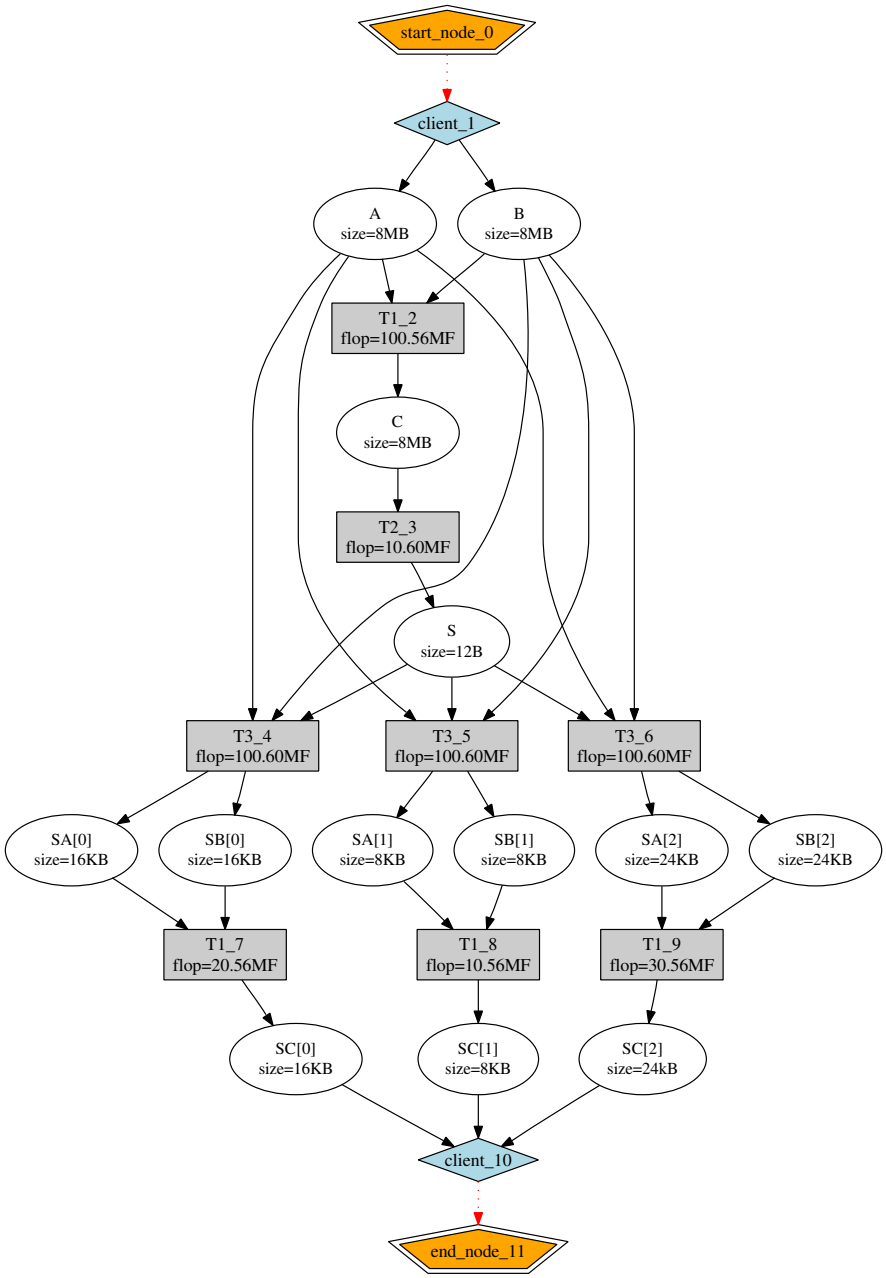

Figure 2. The task graph generated from the ADL module of the adaptive algorithm example

4. 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 3 and 5), the SmartGridRPC with smaller mapping blocks version (tables 4 and 6) and the SmartGridRPC with ADL version (tables 8 and 10). 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.

Table 11. Experimental results for the conditional algorithm applications

	GridSolve	SmartGridSolve with smaller blocks	
Data Size	Avg Time	Avg Time	S_p v GS
24MB	24.08s	19.31s	1.25
48MB	48.30s	40.35s	1.20
96MB	97.51s	81.00s	1.20
192MB	195.80s	156.67s	1.25
384MB	404.31s	317.36s	1.27
576MB	648.02s	497.31s	1.30

	SmartGridSolve with ADL		
Data Size	Avg Time	S_p v GS	S_p v SGS
24MB	14.08s	1.71	1.37
48MB	26.94s	1.79	1.50
96MB	57.63s	1.61	1.41
192MB	113.67s	1.72	1.38
384MB	230.03s	1.76	1.38
576MB	364.62s	1.78	1.36

Table 11 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 calculate 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.

Table 12 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 sub-vectors, n, and their sizes, S, generated by task *T2*, are constant between experiments of 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 vectors size. The speed-up demonstrated by the SmartGridSolve implementation, with 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 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 SmartGridSolve 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 use of ADL permits SmartGridSolve to map a larger group of tasks and to minimise the amount of data moved in the network.

Table 12. Experimental results for the adaptive algorithm applications

	GridSolve	SmartGridSolve with smaller blocks	
Data Size	Avg Time	Avg Time	S_p v GS
24MB	28.03s	25.02s	1.12
48MB	51.26s	46.70s	1.10
96MB	112.53s	86.76s	1.30
192MB	216.99s	182.15s	1.19
384MB	435.56s	369.25s	1.18
576MB	713.78s	604.23s	1.18

	SmartGridSolve with ADL		
Data Size	Avg Time	S_p v GS	S_p v SGS
24MB	17.41s	1.61	1.44
48MB	30.40s	1.69	1.54
96MB	58.40s	1.93	1.49
192MB	118.70s	1.83	1.53
384MB	269.65s	1.62	1.37
576MB	445.75s	1.60	1.36

5. Conclusion

In this paper, we have studied how the Algorithm Definition Language (ADL), in conjunction with SmartGridSolve, can be used to improve the execution of typical applications with irregular algorithms. We have conducted experiments demonstrating significant performance gains due to the use of ADL in applications that use conditional statements, iterative computations and adaptive methods. We have demonstrated that a programmer can improve the performance of their Grid application by using algorithm specification languages to descibe the underlying algorithms. This work was supported by the Science Foundation Ireland.

References

[1] T. Brady, J. Dongarra, M. Guidolin, A. Lastovetsky, and K. Seymour. SmartGridRPC: The New RPC Model for High Performance Grid Computing and its Implementation in SmartGridSolve. Manuscript submitted for publication, April 2009.

[2] 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)*. IEEE Computer Society, 2008.

[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] 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)*. IEEE Computer Society, 2008.

[5] 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)*. IEEE Computer Society, 2009.

[6] 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)*. Springer, 2002.

[7] 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.

[8] 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.