Gridification of an Application: From Sequential to EGEE-Grid

Ricardo Marques
pg10898@alunos.di.uminho.pt

Universidade do Minho, Braga, Portugal

Abstract. This article reports the porting of FireStation (FS), a fire growth simulation application, from a Desktop to a Grid environment. The objective of this work is to exploit Grid capabilities in order to have a faster execution, to manage large data input/output files, to create a database of results which can avoid repeated simulations, and to allow an interactive monitoring of the fire growth. The development of a parallel version of FireStation with the Message Passing Interface (MPI), which can take profits of the Grid as an inherent parallel environment, was the first stage the work. The next step was to integrate this version with the the available Grid tools and services which suited our objectives: the LFC file catalog to manage large data files; the AMGA metadata catalog to create a database of previous executions and respective results; and finally WatchDog, used to monitor the simulation.
Both the parallel and the Grid version were implemented successfully, showing that the Grid can give a good response to improve the execution of a parallel application with large input output demands in real time, keep track of the simulations already computed by maintaining a metadata database, and allowing for soft real time monitoring.

1 Introduction

To take a decision in an emergency situation, Civil Protection (CP) authorities need to follow operational procedures involving several institutions (civil protection agencies, public administrators, research centers, etc.) each one with a different role (decision-makers, data and services providers, emergency squads, etc.). In these situations, a faster program and real time georeferenced data can help making the difference between an efficient rescue plan and an uncontrolled situation. That is the case of applications used for floods and fire predictions. The sooner the authorities have the information about how will the situation evolve in the terrain, the sooner they will be able to react. That is why this kind of applications need to be able to make the best use of the computational power available, in order to model hazard situations efficiently and based in real-time data.

In this report we describe the work of taking FS, an integrated system aimed at the simulation of fire growth over a complex topography, from a Desktop

environment to a Cluster environment, and finally from Cluster to a Grid environment. With this work, we intend to make FS take profits of the computational power available, as well as several Grid tools which can improve its execution. In the next section, the original version of FS is described. In the third and fourth sections, the parallel model of FS (P-FS) and its implementation are presented. In the fifth section we present the gridification of P-FS. Then, some results are presented, as well as the conclusions and the future work.

2 Application: FireStation

FireStation is an integrated system aimed at the simulation of fire growth over a complex topography [2, 3]. It integrates a module for wind field generation, as well as a module for the computation of the Fire Weather Index, from the Canadian System. The FS software was developed under the environment of the CAD application Microstation, from Bentley Company, in University of Coimbra.

The underlying software was written in MDL, a specific C language of Microstation that has built-in subroutines for the design of window-based interfaces, generation of visualization elements in the 3D space, on top of the usual mathematical capabilities of the C language. The FireStation core is based on mathematical models of fire propagation and wind fields generation.

2.1 Graphical Interface

The graphical environment of Microstation is three-dimensional. Thus the visualization process is allowed to employ, not only the normal top-view, but any other view perspectives as well, for map display and other visualization procedures.

2.2 Wind Field Generation Module

The wind model is a self-contained Fortran code, which runs as an external program. Wind is probably the single most important input parameter for fire spread computation. For the case of forest fires, one is primarily interested in the knowledge of the wind speed and direction at a distance to the ground equivalent to approximately midflame height. At this vertical level, wind is much influenced both by topography and by vegetation. Wind field behavior, as a fluid dynamics problem, is governed both by the continuity equation (mass conservation) and the Navier-Stokes equations [3], which describe the momentum conservation.

2.3 Fire Weather Index Module

The fire behavior predictions given by FireStation are aimed at support decision-making on forest and fire management activities at a local scale. Nevertheless, the system also incorporates a fire danger rating system applicable at a broader scale, namely at regional and national level. The fire danger rating system incorporates the Canadian Fire Weather Index (FWI), which integrates weather

and fuel parameters affecting fire potential. The system allows to have a broad assessment of large-scale fire potential through the evaluation of the daily and spatial variation of the fire danger index and estimate the moisture content of dead and live fine fuels through empirical relationships.

2.4 Fire Growth Simulation Module

The fire growth simulation module is in charge of computing the fire spread over a complex topography. For the present work, fire growth is the most important feature of FireStation, as this module is the one that will be parallelized, and ported to a Grid execution environment.

Inputs In order to simulate the fire growth, FireStation three kinds of data: a description of the characteristics of the terrain, the wind conditions affecting that terrain, and some control parameters.

The terrain information is described using the BEHAVE model [2]. This model divides the terrain into cells, over which its properties are assumed to be constant. The description of the topography and fuel characteristics of the terrain is registered in two files which store the altitude and the fuel model for each cell.

The information about the wind conditions at midflame high affecting the terrain, is produced previously by the Wind Field Generation Module. Finally, some control data such as the ignition points and the stopping criteria have to be passed to the application. The stopping criteria can be maximum simulation time, maximum area burned or maximum number of cells burned.

The Propagation Model The fire growth simulation is a process of contagion between burning and non-burning cells: at a generic time instant, the time the fire takes to propagate from each burning cell to its neighbors is computed. This calculus is based on the cells' wind information, slope and fuel characteristics. Each non-burning neighbor cell is thus assigned a number which represents the shortest time instant at which ignition will take place. In the next iteration, the non-burning cell with the shortest time assigned becomes a burning cell, and the propagation process is repeated. This model is based on a Dijkstra's dynamic programming algorithm, leads to a time progression which may not be constant.

The choice of the cells defined as neighbors plays an important role in terms of discretization errors. Figure 1 exemplifies the case where 16 neighbors are defined: Each burning cell (in black) propagates the fire, at most, to the 16 neighbor cells (in green).

For the present work, it is interesting to look inside the core of the application: the fire propagation algorithm, shown in algorithm 1.

The fire propagation algorithm consists on a cycle which executes while there are still cells marked to burn and none of the stopping criteria is verified. If there are no cells marked to burn, then it means that there is no fire in the area of calculus, so the simulation can terminate. If any of the stopping criteria

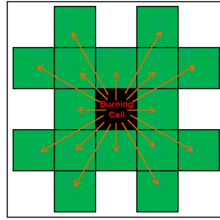

Fig. 1. Fire propagation from a burning cell to its neighbor cells.

Algorithm 1 Fire propagation algorithm.

```
While(ExistPredictions && ¬StoppingCriteriaMatched) {
  1. Fetches the cell set to burn in the shortest period of time
  C = GetNextBurningCell();
  2. The present time value is set to the ignition time of cell C
  PresentTime = GetBurningPrediction(C);
  3. Cell C is burned
  BurnCell(C);
  4. Refreshes the number of cells burned and the total burned area:
  TotalBurnedCells = TotalBurnedCells + 1;
  BurnedArea = TotalBurnedCells * CellArea;
  5. Computes the fire propagation from C to its neighbour cells in the form of predictions,
and incorporates the cells and the predictions in the list of predictions:
  ComputePropagation(C);
  6. Makes a set of tests to decide whether the simulation will be stopped or not
  If(Any Stopping Criteria is Verified) {
    StoppingCriteriaMatched = True;
  } (End If)
} (End While)
```

is reached, then the simulation also stops. This stopping criteria is one out of those which can be passed as input (see section 2.4), and also if the fire reached the limits of the map. In each iteration, the program selects the cell C predicted to burn in the shortest time interval (`1.` in algorithm 1), and sets the current time to C's ignition time (`2.`). Then cell C is burned (`3.`), and the area burned and total of burned cells values are refreshed (`4.`). The next step is to calculate the fire propagation from C to its neighbor cells. This is made computing the ignition time for each C neighbor cell, and incorporate the results in the list of predictions. Finally, a set of tests is made in order to decide if the simulation should continue.

Outputs The output of the fire growth simulation is a list of the burned cells, and its respective ignition time. However, this output can be post-processed in order to get some more information. Examples of this information is the fire spread rate and direction, the area burned and its perimeter, and the linear intensity of the fire front.

3 P-FireStation Model

The goal of the parallelization phase was to produce a prototype of FireStation which can take profits of the computational power available in Cluster environments. The main idea is to use multiple nodes to simulate the fire growth in parallel. High performance Input/Output stages is also an objective, due to the potentially large data files involved in the simulation. The starting point to produce such a prototype was a C version of FireStation. This version only included the Fire Propagation Module.

The parallelization strategy is to assign a subset of cells obtained by partitioning the whole map to each process participating in the simulation. Each process should then compute the fire growth in its own subset. The eventual fire propagation to a neighbor subset has to be communicated to the process in charge of it, so that the neighbor process is aware that the fire is burning its terrain, and can simulate the fire spread from that point on. When all the processes terminate the simulation, the results should be joint.

3.1 Domain Decomposition

The domain decomposition phase consists on deciding how will the terrain be distributed among the processes involved in the simulation. A natural solution, regarding the fire propagation algorithm, is to assign each processor one or more contiguous subsets of cells. In figure 2 we can see an example of a domain decomposition for 9 processes, where each process is assigned a single sub-terrain.

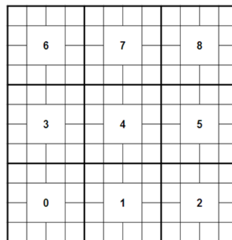

Fig. 2. Domain decomposition example for 9 processors.

The number in each of the 9 squares in bold, corresponds to the rank of the process in charge of it.

In order to compute the fire spread in its own sub-terrain, each processor needs more information than just the one corresponding to its sub-terrain. Let us focus of process 4 and figure 3 (left).

When this process is computing the fire spread from a burning cell in the border with another process (marked in red), it has to make predictions for the 16 cell surrounding the cell in red. The surrounding cells are marked in green

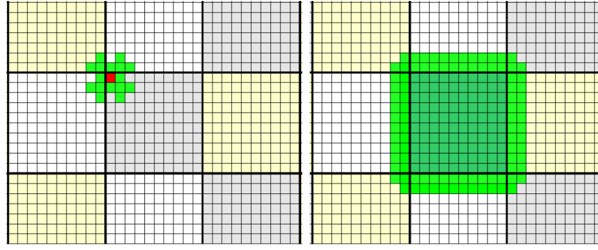

Fig. 3. Fire propagation to neighbor processes (left) & information needed to compute the fire growth in a sub-terrain (right).

in the figure. The cells to where the fire can be spread are marked in green. However, some of these cells do not belong to its sub-terrain. As a consequence, to calculate the fire growth in its sub-terrain, the process needs to have not only information about its local cells, but also information about all the cells from other processes for which it can possibly make predictions. Figure 3 (right) shows the area needed by the process 4 in order to compute the simulation.

In dark green we have the information corresponding to local cells, and in light green, the information about cells belonging to other processes for which process 4 might make predictions.

3.2 Parallel Fire Growth Computation

The fire growth sequential algorithm presented in section 2.4 had to be rearranged for a parallel execution. The idea was to keep the most possible of the sequential version, and inject parallel instructions in order to produce a parallel prototype. The result is shown in Algorithm 2.

This algorithm, based in a message passing parallel paradigm, can be divided in three main conceptual parts: the sleeping part, the burning and predictions parts, and the termination part. The instructions injected to allow a parallel execution are marked with a `P`. Those which were kept from the sequential algorithm are marked with an `S`.

The Messages In a parallel fire growth simulation, we identified three types of messages which can be exchanged between the processes:

- work messages, which carry information about burning time predictions for a set of cells;
- termination messages. When a termination message is received by a process, the process ends the fire propagation calculus, registers the results obtained so far in a file, end terminates its execution;
- messages which demand a test to the current simulation time of each process. These messages, are sent from a process which is interested in the test, to all the other processes.

Algorithm 2 Parallel Fire Propagation Algorithm.

```
While(¬StoppingCriteriaMatched) {

  P1. Cheking If Should Sleep Or Not
  While (¬ExistPredictions or OverMaximumTime) {

    P1.1. Over The Maximum Simulation Time!
    While(OverMaximumTime) {
      CheckOthersTime();
      If(AllOverMaximumTime) Then Terminate;
      Else {
        Sleep();
        ReceiveMessages();
        If(EndSimulation) Terminate;
      } (End Else)
    } (End While P1.1)
    P1.2. There Are No Cells Predicted To Burn!
    while(¬ExistPredictions) {
      Sleep();
      ReceiveMessages();
      If(EndSimulation) Terminate;
    }(End While P1.2)
  } (End While P1.)

  P2. Receive Eventual Messages
  ReceiveMessages();
  If(EndSimulation) Terminate;
  Sequential 1, 2 and 3
  P3. Calculate the predictions for the burning of the neighbour cells
  CalculatePredictions(ii, jj);
  Sequential 5.

} (End While)
```

Termination When should the simulation terminate is a question which is harder to answer when we are talking about the parallel algorithm, compared to the sequential one. In section 3.2, we stated that the simulation could stop due to the reaching of one of these values: maximum simulation time, maximum area burned, maximum number of cells burned, or because the fire got out of the map limits. For simplicity reasons, we decided that the parallel prototype stopping criteria should be limited to maximum simulation time and whether the fire is out of the map or not. This maximum simulation time is a measure of the time when the cells are burned, and not the time of the simulation itself. For example, if we say that the time is 1000 seconds, the simulation will proceed till a cell is burned at the time 1000 or later, and not till the program is executing for 1000 seconds.

In the parallel algorithm, the simulation can only end if all the processors are over the maximum simulation time. The counter prove if this statement is easy to do. Let's imagine a case where the simulation is made by two processes. If the simulation ended when only one of them had reached the maximum simulation time (let us say 20 minutes), then both would register the results they had till that point. But the second processor was in a time minor than 20 minutes. He could only have had the time of simulating the fire growth till, for example, 15 minutes (due to work balancing reasons, or because it executed in an inferior machine). This would lead to a corruption of the final results. The solution for

this problem is to synchronize all the process. If a process reaches the maximum simulation time, then it should wait for the other to reach it also. Only when all the processes involved in the simulation reach that limit, the simulation can be terminated.

We can see this behavior in algorithm 2 if we focus on the *first while* of `P1.1`: the process checks if all the processes are over the maximum simulation time, and if they are, then the simulation ends, otherwise, it goes to sleep, till a new message is received.

To Sleep or Not to Sleep In a parallel execution, a node might find itself in a situation where there is no work to do because there are no burning cells in its sub-terrain, or because the maximum simulation time was already reached, and the process has to wait for the others (`P1.` in algorithm 2). In both situations, the program should take the decision "go to sleep".

The sleeping state is a state where the program is just waiting for a message to be received. When the program detects an incoming message, it wakes up and receives it. The action which follows the receiving of the message, depends on the message received, and on the reason for which it was sleeping.

But before analyzing this change of state, it is important to refer a feature of this parallel algorithm: the capacity to go back in the simulation time. If a process receives a prediction to a local cell C which has already burned at a given time t1, and this new prediction says that cell C should burn at a time t0 $<$ t1, then the algorithm will go back in time till t0. Cell C will be (re)burned at time t0, and the simulation will proceed treating every cell burned at a superior time than the current simulation time as non-burned cells.

After this parenthesis, we can go back to the analysis of the state changes. Imagine that a process is sleeping because it reached the maximum simulation time (`P1.1.`). After waking up due to an incoming message, it can take one of the following actions depending on the type of message received:

- **Terminate the simulation:** the process will take this action if a termination message is received (from a process where the fire propagation reached the limits of the global map), or if it receives a message for a collective check of the simulation time and the result is that all the processes are either above the maximum simulation time or have no work;
- **Continue the simulation:** the computation of the fire growth will be continued if a work message making the process go back to a simulation time inferior to the maximum limit;
- **Go back to sleep:** the process will go back to sleep if it receives a work message which does not cause the simulation time to change, or if it receives a message for a collective check of the simulation time and the result is that there is one or more processes computing the fire growth, with a simulation time inferior to the maximum limit;

If, on the other hand, the process is sleeping because it had no work to do (`P1.2.`).

- **Terminate the simulation:** the program will take this action in the same situations of the previous case study;
- **Continue the simulation:** the computation of the fire growth will be continued if a work message with a prediction for a non-burned cell is received, or if the work message received contains a prediction for a burned cell, with a time inferior to its burning time;
- **Go back to sleep:** the process will go back to sleep if it receives a message for a collective check of the simulation time and the result is that there is one or more processes computing the fire growth, with a simulation time inferior to the maximum limit;

Burning and Predictions The parallel burning and prediction phase is just slightly different from the sequential one. In `P3.`, the propagation of the fire from the actual cell to its neighbors is calculated. The behavior of the *CalculatePredictions* function should now be the following: if the cell predicted to burn belongs to the local process, then it will be appended to the local list of cells predicted to burn, just as in the sequential algorithm; if, on the other hand, the cell predicted to burn belongs to a neighbor process, then a message with the cell location and burning time should be sent to it.

Before burn the cell and calculate the predictions, the process should check from eventual incoming messages (`P2.`).

3.3 Monitoring the Simulation

One of the improvements we want to do to the sequential version of FireStation, is the ability to watch the fire growth in real time, that is, to see the burned cells of the map as they are marked as burned, and not only see the results in the end.

To allow the monitoring, each process should write to a local file the cells which are marked as burned, right after their burning. That is, in each iteration, the burned cell should be written in this local file. This will permit that, at the same time, an external application is constantly reading the information placed in each local file (remember that there is one local file per process). This information can then be shown in the screen, allowing the user to have an idea of the fire growth, before the end of the simulation.

In a long simulation, this feature can have an important role in the decision of the civil protection, as it provides the decider with faster information.

4 P-FS Implementation

The implementation strategy was to use the MPI protocol to define a framework containing functions which should be injected in the sequential code, allowing FireStation to execute in a cluster environment.

4.1 MPI

MPI [4, 5, 6] is a specification defining the rules for the implementation of an API that allows processes executing in different processors to communicate with each others. This communication is based on a message exchanging mechanism. Its objectives are performance, scalability and portability [11]. It offers a set of primitives which allow a collective or point-to-point communication. This communication can be made in several ways: synchronous, blocking, immediate, etc.. The concept of message passing makes this mechanism specially suitable for distributed memory cluster environments, where there are no common memory blocks to be used for inter-process communication executing in different machines.

4.2 The Parallel Framework

In order implement the FireStation parallel version, a framework containing a set of functions, tags and MPI types was developed.

Following is a list of the main functions contained in this framework, and their description:

- *goToSleep_MPI()*: puts a process in sleeping mode until a message from another process is received.
- *receiveMsg_MPI()*: receives eventual messages from other processes;
- *sendMsg_MPI(int tag, int proc, prevision p)*: this function has three usage cases. If a *work* tag is used, a message to process proc containing the prevision p. If the tag value is *termination*, a termination message is sent to all the other processes, causing the end of the simulation. In case the tag is a *AllReduce* tag, this function tests if all the processes are over the maximum simulation time, and causes the termination of the simulation if so.

4.3 Domain Decomposition

The input stage were implemented with MPI-IO, according to figure 3 right, using local arrays with ghost area [6]. The local arrays, where the terrain and wind information will be placed, have to be allocated with a few extra rows and columns in each dimension. This extra area, which is not really part of the local array, is often referred as ghost area (in light green in the figure). The ghost area is used to store the information about cells belonging to neighbor processes needed for the calculus of local cells fire propagation. As the local array (storing the information marked in dark green in the same figure) has ghost area around it, the data read from file is not stored contiguously in memory, that is, the rows of the local array in memory are separated by a few elements of ghost area. This non-contiguous memory layout is described in terms of an MPI derived datatype, and specified as the datatype argument to a single MPI_File_read_all function.

After each process reads its data to the local array, it identifies its neighbors and makes an *MPI_Send_recv* with each one of them in order to exchange the ghost areas.

The use of MPI-IO as a high performance library to read the data from files, along with the distribution of the total workload among the processes (remember that each process only reads a part of the input file) is extremely important to allow simulations involving very large datasets. It can make the input stage much faster, as well as solve limitations in the local memory of each working node (for example, if the file is larger than the memory available in the node).

4.4 Parallel Fire Growth Computation

Accordingly to the parallelization strategy stated in this document, the functions defined in the parallel framework were injected in the sequential version of FireStation, in order to implement algorithm 2.

5 G-FireStation

The objective of porting FS to the Grid (creating the G-FS) is to exploit Grid capabilities in order to manage large data input/output files, to create a large data base of simulation results and to allow a soft real time monitoring of the fire growth.

To achieve these goals, several Grid tools available were used: gLite for job description, job submission, output retrieval and data management; AMGA for the creation and management of an executions' database; and WatchDog as a non-intrusive application monitoring tool.

The G-FireStation should perform the following tasks in its Grid execution:

- Download the large data files from a Grid Storage Element;
- Start the job monitoring tool (WatchDog);
- Execute the P-FS, using WatchDog to report the fire growth evolution;
- Register the results in a Grid Storage Elements after termination;
- Add an entry with the information about this execution in the data base server, AMGA;

5.1 Data Management

In a fire growth simulation, the are three potentially large data files: the wind file, the terrain altitudes, and the fuel distribution. For this reason, these files were previously stored in a SE. Before the execution of the P-FS application, the G-FS Job executes a script which downloads these input files from the SE to the Working Nodes (WNs) using simple gLite Data Management commands.

As the output of a simulation is also a potentially large file, it is uploaded to a SE after the termination of the simulation. To accomplish this task, the G-FS executes another script, once the simulation ends.

5.2 Executions' Database

To avoid repeated G-FS executions, and to keep track of the results and how they were obtained, we decided to create an executions' database, using AMGA [8].

AMGA (Arda Metadata Catalog Project) is a Grid service that allows users to store information used to characterize the contents of the files stored in the Grid SEs. Each file can be characterized by a set of attributes. AMGA can be accessed using both the mdclient client application or a set of API calls.

Table 1 is a simple example of G-FS executions' description.

Entry ID	Map	FuelDistr	Wind	Ignitions
/Grid/firePT1.out	/Grid/portugal.asc	/Grid/fueldistrPT.asc	/Grid/windsPT.out	/Grid/ignitionPT.dat
/Grid/fireES2.out	/Grid/spain.asc	/Grid/fueldistrES.asc	/Grid/winds.out	/Grid/ignitionES.dat

Table 1. G-FS executions' table.

Each entry is characterized by an entry identifier, which in our case is the name of the results file (as it is unique). Each entry is defined by the map used in the simulation (Map), the fuel distribution (FuelDistr), the wind affecting the terrain (Wind) and the ignition points (Ignitions). More fields can be added to this table, in order to increase the degree of information per execution.

Using this database, the user can avoid repeated executions, by checking in the executions' table if any execution with the same inputs took place before. If so, the user can use the information in the table to get the output directly from the SE, without having to submit a job to the Grid. Each time a new execution takes place, a new entry is added to the table, by executing a simple script.

5.3 Job Monitoring

WatchDog is a tool which allows Grid users to monitor the state of their applications during the execution [9]. It consists on three bash scripts which should be taken to the CE along with the application to be monitored: the configuration script, the control script, and the WatchDog core script. To actually monitor the changes in the files, the user has to start the core script before the application starts executing, and stop it after the application is terminated.

Watchdog runs beside the main job and consists of a simple infinite loop. In each iteration any file change will be reported to the user via LFC or AMGA. At the end of each iteration the Watchdog sleeps for a given and configurable number of seconds and then a new iteration will start.

FireStation was rearranged such that each process registers into a local file its burned cells, in real time. We called this file *log file*. Whenever a cell burns, the process appends a line with the cell's location and its ignition time to the *log file*. There is one *log file* per process.

Algorithm 3 Simple MPI Job.

```
JobType = MPICH;
Executable = pearson_blocking;
CpuNumber = 3;
StdOutput = pearson.out;
StdError = pearson.err;
InputSandbox = {pearson_blocking};
OutputSandbox = {pearson.err, pearson.out};
Requirements =
 (!RegExp(ce.cp.di.uminho.pt,other.GlueCEUniqueID));
RetryCount = 0;
```

Using this feature, the task of monitoring the simulation became quite simple: WatchDog was configured in order to keep track of the changes in the *log files* produced by each processor. During the simulation, WatchDog makes snapshots of the changes in the log files, publishes them in the LFC Catalog. The user can access and process that information: for example, a graphical application can be constantly reading the snapshots and show a map with the evolution of the simulation in real time.

5.4 Job Description

To submit a job to the Grid, one must specify its characteristics and constraints, which are taken into account to select the best resource to execute the job. This is done with a high-level language called Job Description Language (JDL) [10].

In algorithm 3 we have an example of a JDL describing a simple MPI Job.

The values in the attributes *JobType*, *Executable* and *CpuNumber* mean that this job executes in 3 nodes an MPI program named *pearson_blocking*, compiled with the MPI flavor MPICH. The input for this job is just the executable binary file itself as we can see from the values of *Executable* and *InputSandbox*. The standard output and standard error are registered in the *pearson.out* and *pearson.err* respectively. The user can fetch these two files in the end of the execution, because they are declared in the *OutputSandbox* attribute. This job has the special requirement to run in *ce.cp.di.uminho.pt*, a civil protection University of Minho's Grid Site.

However, this example is just to introduce the JDL and the description of MPI Jobs, and it is not the best option to submit MPI Jobs, specially if we need to do pre/post processing operations (compiling, downloading input files, etc.). The best solution is achieved by using the mpi-start scripts, which were developed by the int.eu.grid project and based on the work of the MPI working group that contains members from both int.eu.grid and EGEE.

Using the mpi-start system requires the user to define a wrapper script and a set of hooks. The mpi-start system then handles most of the low-level details of running the MPI job on a particular site.

Wrapper Script for Mpi-Start Users typically use a script that sets up paths and other internal settings to initiate the mpi-start processing. The script first

Algorithm 4 FireStation's job description file.

```
Type = ''Job'';
JobType = Normal;
CPUNumber = 12;
Executable = fireStation-start-wrapper.sh;
Arguments = fireStation MPICH;
StdOutput = std.out;
StdError = std.err;
InputSandbox = { fireStation-start-wrapper.sh, fireStation, ignition.dat, Control.dat,
  fuelmodels.fls, fireStation-hooks.sh, fireStation-download-input.sh,
  fireStation-upload-output.sh, synchronize-amga-se.sh };
OutputSandbox = { std.err, std.out };
Requirements = Member(MPI-START, other.GlueHostApplicationSoftwareRunTimeEnvironment)
  && Member(MPICH, other.GlueHostApplicationSoftwareRunTimeEnvironment) ;
```

sets up the environment for the chosen flavor of MPI using environment variables supplied by the system administrator. It then defines the executable, arguments, MPI flavor, and location of the hook scripts for mpi-start. Lastly, the wrapper invokes mpi-start itself.

Hooks Script for Mpi-Start The user may write a script that is called before and after the MPI executable is run. The pre-hook can be used, for example, to compile the executable itself or download data. The post-hook can be used to analyze results or to save the results on the grid.

Job Description The user must define a JDL file describing the requirements for the job. The JobType must be MPICH and the attribute NodeNumber must be defined. The JobType attribute must be MPICH in all cases.

Running the MPI Job Running the MPI job is no different from any other grid job. Use the commands glite-wms-job-submit, glite-wms-job-status, and glite-wms-job-output to submit, check the status, and recover the output of a job.

The G-FireStation Job Description The G-FS JDL file, describing the requirements of the G-FS job, is shown in algorithm 4.

As we can see by the inclusion of a wrapper and hooks files in the Input Sandbox, it makes use of the mpi-start mechanism described previously. This is the reason why the Executable is the wrapper script, and not the P-FS binary file itself. In the Input Sandbox are also carried the parametric data for the simulation (ignition.dat, control.dat and fuelmodels.dat); scripts for downloading/uploading data from/to the SE respectively (fireStation-download-input.sh and fireStation-upload-output.sh); a script to add an entry of the execution in the executions' database (synchronize-amga-se.sh); and finally, the WatchDog scripts in order to allow the monitoring of the simulation. This job will run in 12 nodes of a Grid Site with the MPI-START mechanism enabled (CPUNumber and Requirements respectively).

Algorithm 5 FireStation's pre and post processing phases.

```
# This function will be called before the MPI executable is started.
pre_run_hook () {
# Get the Input Files from the Storage Element
. fireStation-download-input.sh
# Start WatchDog
./watchdog.ctrl start
return 0
}
# This function will be called before the MPI executable is finished.
post_run_hook () {
# Stop WatchDog
./watchdog.ctrl stop
# Uploading the output form the CE to the SE
. fireStation-upload-output.sh
# Synchronising Amga database with the SE
. synchronize-amga-se.sh jobid.amga
return 0
}
```

Algorithm 5 contains the hooks file, describing the pre and post run phases of the application.

In the pre run phase, the inputs are downloaded from the SE to the WNs, WatchDog is started, and the P-FS binary file is given permissions to execute. After the execution terminates, WatchDog is turned off, the result is uploaded to the SE and a new entry is added to the AMGA database.

6 Results

We built a parallel prototype of FireStation (P-FS) implemented with MPI, which makes use of multiple nodes and high performance libraries for the Input/Output stages, which can speed up the execution time in large simulations. The fire propagation algorithm had to be redesigned, in order to be executed in parallel by multiple processes. This was accomplished injecting parallel primitives in the sequential code, and maintaining the core of the sequential algorithm. The prototype executes successfully in a cluster environment.

Furthermore, this prototype was taken to the a Grid environment. The set of tools available in the Grid allowed us to enhance the execution of the application: the LFC file catalog was used to manage large data files; the AMGA metadata catalog to create a database of previous executions and respective results; and finally WatchDog, to monitor the simulation.

7 Conclusions

This work reported can be divided in two main phases: the parallelization of FS, and the gridification of the parallel prototype.

In the first phase, the goal was to use create a P-FS which could replicate the behavior of the FireStation's Propagation Module, running in a cluster environment. The idea was to achieve it using a message passing programming paradigm,

high performance input/output libraries, and injecting parallel primitives in the sequential version. This would allow the parallelization of the module in a fast and clean way, without requiring technical knowledge about the fire propagation model. The successful implementation of the model proved that this approach is feasible, and can be used for the parallelization of other FS modules.

The second phase's goals were to execute P-FS in a Grid environment, allowing to get input data in real time, create a database which could avoid repeated executions and improve the efficiency of the prototype by returning immediately a result previously computed and to find a solution for the job monitoring. These goals were completely achieved, showing that the Grid can give a good response to improve the execution of an application with large input output demands in real time, keep track of the simulations already computed by maintaining a metadata database, and allowing for soft real time monitoring.

8 Future Work

As future work, the G-FS prototype can be integrated with a graphical interface. This graphical interface could allow the user to choose the ignition points, to select a part of the map where he/she wants the simulation to take place, etc.. It could also use the information published by WatchDog in order to allow the user to see the fire growth over the map.

G-FS could then be placed in a Grid portal. This would allow the application to be available over web, and could also be used to enable access control. Genius, a grid portal which allow the interaction with graphical interfaces could eventually be a good solution.

References

1. I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, International J. Supercomputer Applications, 15(3), 2001.
2. AMG Lopes, MG Cruz, and DX Viegas. FireStation - an integrated software system for the numerical simulation of re spread on complex topography. Environmental Modelling and Software, 17(3):269285, 2002.
3. António Lopes, FireStation User's Manual, Universidade de Coimbra 2000.
4. Ian Foster, Addison-Wesley, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.
5. N. MacDonald, E. Minty, J. Malard, T. Harding, S. Brown, M. Antonioletti, Writing Message Passing Parallel Programs with MPI, Edinburgh Parallel Computer Centre (EPCC)
6. Using MPI-2: Advanced Features of the Message Passing Interface W Gropp, R Thakur, E Lusk - 1999 - MIT Press Cambridge, MA, USA
7. N. MacDonald, E. Minty, J. Malard, T. Harding, S. Brown, M. Antonioletti, Writing Message Passing Parallel Programs with MPI, Edinburgh Parallel Computer Centre (EPCC).
8. Nuno Santos and Birger Koblitz, Metadata services on the Grid, Proceedings of the X International Workshop on Advanced Computing and Analysis Techniques in Physics Research - ACAT 05

9. Riccardo Bruno - INFN Catania, A watchdog utility to keep track of job execution, https://grid.ct.infn.it/twiki/bin/view/PI2S2/WatchdogUtility
10. Stephen Burke, Simone Campana, Patricia Méndez Lorenzo, Christopher Nater, Roberto Santinelli, Andrea Sciabà, gLite 3.1 User Guide, Manuals Series
11. Sayantan Sur, Matthew J. Koop, Dhabaleswar K. Panda, High-performance and scalable MPI over InniBand with reduced memory usage: an in-depth performance analysis, Proceedings of the 2006 ACM/IEEE conference on Supercomputing.