

DataGrid

Job partitioning and checkpointing



WP1: Grid Work Scheduling

Document identifier: **DataGrid-01-TED-0119-0.3**

Date: **June 3, 2002**

Work package: **WP1: Grid Work Scheduling**

Partner(s): **INFN**

Document status: **DRAFT**

Deliverable identifier:

Abstract: This is a working document on the problem of job partitioning and job checkpointing; we analyze the application's requirements and describe how these could be addressed in the framework of the WP1 Workload Management System.

Delivery Slip

	Name	Partner	Date	Signature
From	Alessio Gianelle, Rosario Peluso, Massimo Sgaravatto	INFN Padova	22/5/2002	
Verified by				
Approved by				

Document Log

Issue	Date	Comment	Author
0.1	6/11/2001	First draft	Alessio Gianelle
0.2	14/11/2001	Added some comments after discussions with Lacaprara.	Alessio Gianelle
0.3	21/11/2001	General re-styling. Input from WP2's documents.	Alessio Gianelle
0.4	09/01/2002	Added Section on Checkpointable jobs.	Alessio Gianelle
0.5	14/01/2002	Added a first analysis of PROOF and IMPALA tools.	Alessio Gianelle
0.6	21/01/2002	General re-styling.	Massimo Sgaravatto
0.7	22/01/2002	Added some examples to illustrate the API.	Alessio Gianelle
0.8	28/01/2002	Added the section about Multiple Jobs.	Alessio Gianelle
0.9	04/02/2002	Input from the discussion with Prelz.	Alessio Gianelle
0.10	07/02/2002	General re-styling	Massimo Sgaravatto
0.11	14/02/2002	Removed section on multiple jobs and general re-styling	Massimo Sgaravatto
0.20	21/03/2002	Checkpoint section rewritten	Alessio Gianelle
0.21	26/03/2002	Partitioning section rewritten	Alessio Gianelle
0.22	05/04/2002	General re-styling	Massimo Sgaravatto
0.23	9/05/2002	Changes after discussions at WP1 meeting	Massimo Sgaravatto
0.24	22/05/2002	Changes addressing Ludek comments at WP1 meeting	Massimo Sgaravatto

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files
L ^A T _E X	DataGrid-01-TED-0119-0.3.tex

Contents

1	INTRODUCTION	6
1.1	OBJECTIVES OF THIS DOCUMENT	6
1.2	APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS	6
1.3	DOCUMENT AMENDMENT PROCEDURE	7
1.4	TERMINOLOGY	7
2	User requirements.	8
2.1	WP8: HEP requirements.	8
2.2	WP9: EO-application requirements.	8
2.3	WP10: Biology application requirements.	8
3	Analysis of user requirements.	8
4	Job checkpointing.	10
4.1	Job state.	11
4.2	Job checkpointing scenario.	11
4.3	Job checkpointing API.	13
4.4	Examples on job checkpointing.	15
5	Job partitioning.	17
5.1	Job partitioning scenario.	17
5.2	Example of job aggregator.	19
6	Architecture and technical aspects.	20
6.1	Job checkpointing.	21
6.2	Job partitioning.	22
7	Issues.	23
7.1	Specification of <i>JobSteps</i> for the job aggregator.	23
7.2	How to be sure that the sub-jobs have saved their final state.	23
7.3	Job aggregator not specified.	24
7.4	How to avoid that all sub-jobs are submitted to the same CE.	24
7.5	Problems saving large size states.	24

1 INTRODUCTION

1.1 OBJECTIVES OF THIS DOCUMENT

The purpose of this document is to provide an analysis of the problem of job partitioning, considering as input the requirements presented in the user requirement documents written by the application workpackages (WP8, WP9 and WP10). After analyzing the user requirements, we discuss how these could be addressed in the framework of the WP1 workload management system.

1.2 APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS

Applicable documents

References

- [R1] WP8 e WP10 Document, *DataGrid User Requirements and Specifications for the DataGrid Project*, Version 2.1 - September 2001. http://datagrid-wp8.web.cern.ch/DataGrid-WP8/Documents/Workspace/D8.1/D8.1.a_v2.1.pdf
- [R2] WP8 Document, *Long Term Specifications of LHC Experiment. Part B*, Version 2.0 - September 2001. http://datagrid-wp8.web.cern.ch/DataGrid-WP8/Documents/Workspace/D8.1/D8.1.b_v2.0.pdf
- [R3] WP9 Document, *Requirements Specification: EO Application Requirements for Grid*, Version 1.2 - October 2001. http://tempest.esrin.esa.it/datagrid/docs/docs/DataGrid-D09.1_2001_10_12.zip
- [R4] WP9 Document, *KNMI Detailed Use Cases*, Version 1.1 - October 2001. http://tempest.esrin.esa.it/datagrid/docs/docs/DataGrid-D09.1_2001_10_12.zip
- [R5] WP10 Document, *Requirements for grid-aware biology application*, Version 3.8 - September 2001. <http://marianne.in2p3.fr/datagrid/wp10/documents/DataGrid-10-D10.1-0102-3-8.doc>
- [R6] WP2 Document, *Query Optimization - Use case working document*, Version 3 - April, 2001. http://cern.ch/grid-data-management/docs/qo_use_case_v3.doc
- [R7] WP1 Document, *Job description language. Howto.*, Version 0.1 - September 2001.
- [R8] René Brun & Fons Rademakers, *Distributed Parallel Interactive Data Analysis Using the PROOF System*, Proceedings of Chep 2001 - September 2001.
- [R9] The Condor project. <http://www.cs.wisc.edu/condor>
- [R10] Condor Team, *Condor Version 6.2.2 Manual*, University of Wisconsin-Madison. <http://www.cs.wisc.edu/condor/manual/v6.2/>

1.3 DOCUMENT AMENDMENT PROCEDURE

This note must be considered as a working document. There are still various open issues that must be clarified, and many other items have not been addressed yet. Therefore the content of this document will change very often, in particular as result of:

- Feedbacks received
- Changes/evolutions/refinements of user requirements
- Evolution in the WP1 workload management system architecture
- Evolution in the DataGrid architecture

1.4 TERMINOLOGY

Definitions

Glossary

CE	Computing Element
DAG	Directed Acyclic Graph
DAGMan	Directed Acyclic Graph Manager
HEP	High Energy Physics
JDL	Job Description Language
LB	Logging & Bookkeeping
MPI	Message Passing Interface
MRI	Magnetic Resonance Image
RB	Resource Broker
SE	Storage Element
UI	User Interface

2 User requirements.

In this section we provide a high level description of the user requirements, concerning the problem of job partitioning, as they are described in the user requirement documents ([R1], [R2], [R3], [R4], [R5]) written by the application workpackages (WP8, WP9, WP10).

2.1 WP8: HEP requirements.

In *DataGrid User Requirements and Specifications for the DataGrid Project* ([R1]) and in *Long Term Specifications of LHC Experiments. Part B* ([R2]), HEP applications underline that HEP data are characterized by independent events, and therefore this peculiarity can be exploited by parallelizing the analysis of these data: analysis jobs can be decomposed in many sub-jobs, which can be executed in parallel over many computing resources. A job in general can be partitioned in different ways: it is necessary to query different catalogues and it is necessary to interact with other Grid services to determine how a job can be “decomposed” in sub-jobs. An other distinguish characteristic is that the various sub-jobs don’t need to communicate with each other directly, using for example an interprocess communication layer like MPI. For what concerns the output, in general it may be necessary to transparently recombining the distributed outputs of the various sub-jobs into a single stream.

2.2 WP9: EO-application requirements.

WP9 does not express particular requirements on job partitioning in its User Requirement Document [R4]. They only report that the JDL specification shall be able to specify that an application shall run multiple times processing a collection of input files.

2.3 WP10: Biology application requirements.

Biologists of WP10, in their *Requirements for Grid-aware biology applications* ([R5]) document, express requirements in some ways similar to the ones reported by WP8, for what concerns job partitioning. For example in *parallel isochromat simulation* a virtual object is composed of many isochromats, and since the resulting k-space is the summation of the k-spaces obtained for each isochromat separately, the various isochromats can be distributed on separate computing nodes. No communication is needed between the nodes during the simulation kernel execution. Then the results of the various simulations (the different k-spaces) are retrieved back for summation and final reconstruction.

An other interesting use case is *parallel sequence implementation*, where for specific MRI sequences like multi-slice imaging, each node could be in charge of computing the image of one slice (a 2D image). Therefore it is necessary to distribute only a part of the object, and each node can work independently from the others, with no communication at all between them.

3 Analysis of user requirements.

From the analysis of the User Requirement Documents, we can say that job partitioning takes place when a job has to process a large set of “independent elements”. In these cases it may be therefore worth to “decompose” the job into “smaller” sub-jobs (each one responsible to process just a sub-set of the original large set of “elements”), in order to reduce the overall time needed

to process all these “elements”, and to optimize the usage of all available Grid resources. These independent “elements” can be events (in the case of HEP applications), isochromats (in the case of parallel isochromat simulation), images of slices that must be processed (for some MRI applications), etc. The distinguish and important peculiarity of the problem is that the various “elements” are independent, and therefore the sub-jobs responsible to process them don’t need any communication between them during their execution. The various sub-jobs, which can run simultaneously on different computing resources, all run the same application. Users may also require that the output data produced by the various sub-jobs are “recombined” together, through an other job, to a single stream, which is the actual output that must be returned to the user. Using a notation introduced in [R6], if the initial job is described in the form:

$$Job = c(F)$$

where F is a set of logical files and c is a computation that must be applied to them, the job partitioning can be described as:

$$c(F) = c_{ag}(c(a_1, f_1), c(a_2, f_2), \dots c(a_N, f_N))$$

where $c(a_i, f_i)$ represents the i^{th} sub-job executed on the files f_i , and $F = \{f_1, f_2, \dots f_N\}$. f_i can be a single logical file, or a set of logical files.

a_i is an “hint” given to the application, which can be necessary to specify which “elements” stored in f_i have to be considered (for example f_j could “store” a certain number of HEP events, and a_j could express which of these events must be processed by job j).

The job c_{ag} is optional and represents the “aggregation” job which in some cases must be executed at the end, to recombine the output files produced by the various sub-jobs.

More in general, and not strictly thinking about partitioning of the input data set, we can say that a job can be partitioned in sub-jobs if the job can be described as “composed” by many **independent** steps. For example a step can be the analysis of a specific sub-set of events, the processing of a isochromat, the Monte Carlo generation of a specific set of HEP events (just a sub-set of the whole set of events that must be produced in that Monte Carlo simulation), etc. This is a more general definition of the problem, which can be applied to a wider set of problems.

Considering this general definition of job partitioning, the problem is therefore how to partition the job into the different independent steps, which can then be executed in parallel by different sub-jobs. This means, for example, partitioning the input set of events that must be processed into subsets (which are then processed in parallel by independent sub-jobs), partitioning the whole set of seeds and other parameters needed for a Monte Carlo production into subsets (then used to generate the various simulated HEP events by different sub-jobs), etc.

Unfortunately at the moment we think that it is not possible to find a general solution to this problem for many different reasons, such as:

- The rules and the criteria that must be used when partitioning a job into independent steps are not always the same and can not be generalized, since they are very application specific.
- To find the best (in term of performance) partitioning of a job, some application specific hints (not available to a possible generic solution to the problem) are needed.
- To determine in which way a job can be decomposed, some application specific catalogues/databases should be queried, and, since these aren’t “standard” Grid services, we can’t make any assumptions on them (their availability, their specific purposes, the mechanisms to interact with them, etc.).

-
- Reading the user requirement documents, it comes out that applications assume that job partitioning is very application specific, and therefore they think that this problem must be addressed by their own (see for example the PROOF environment [R8] and the proposed "CMS grid job decomposition service" [R2]).

As previously discussed, for most applications partitioning a job means partitioning the input set of "independent elements" (events, isochromats, etc.) which must be processed, into subsets, which can then be processed by parallel independent jobs. In the DataGrid architecture these input "independent elements" are stored in the input data files that must be processed, and therefore in this scenario the problem is partitioning the input data set. In this case also the following issues must be underline:

- The existing DataGrid architecture is file based, and it is not based on such "independent elements", which are not "described" in any existing DataGrid services.
- There is not a general rule defining how the "independent elements" are stored in the Grid files. Therefore, for example, a single Grid logical file can store multiple "independent elements", and/or a single "independent element" can be stored in multiple logical files.
- A generic Grid service which maps the "independent elements" to the Grid logical files (i.e. a meta-data catalog) does not exist in the DataGrid architecture.

Therefore we do not think that a general solution to the problem of job partitioning can be found, since it is too strictly tied with applications.

In section 5 we propose a different approach to the problem of job partitioning, in the context of checkpointable jobs, discussed in section 4.

4 Job checkpointing.

Checkpointing a job during its execution means "saving" in some way its state, so that the job execution can be suspended, and resumed later, starting from the same point where it was first stopped, without the need to restart the job from its beginning, and therefore without losing the computation done until that moment.

Checkpointing a job from time to time can prevent data loss, due to unexpected problems (e.g. failures such as machine crashes, etc.). This is useful to increase the reliability, in particular for long-running jobs.

Checkpointing can also be a useful mechanism if the Resource Broker decides, for some reasons, that it is not worth to continue the execution of a job on a particular resource (for example because there are other jobs with higher priority which must be executed first, because that resource is not too suitable for that job, and therefore it is better to find an other resource, etc.). With a job checkpointing facility in place, instead of killing the job and therefore losing the processing done until that moment, it is possible to restart the job, on an other resource chosen by the RB, from the last saved state. These capabilities (i.e. the RB able to decide to stop the execution of a running job) haven't been implemented yet in the Resource Broker, but we feel that a mechanism to support such scenarios can be quite useful.

We don't want to address here the classic checkpointing problem, that is saving somewhere all the information related to a process (process's data and stack segments, information about

open files, pending signals, CPU state, etc.) as it is addressed in other project (e.g. Condor [R9]). Instead, the idea is providing users with a “trivial” checkpointing service: through a proper API, a user can save, at any moment during the execution of a job of his, the state of this job. The hypothesis is, of course, that the job can be restarted from an “intermediate” (i.e. previously saved) state.

4.1 Job state.

As described in section 4, the idea is that users can insert in the code of their applications some specific function calls (see section 4.3) to save, from time to time, the state of their jobs. Many applications, and in particular most DataGrid applications, can be seen as “composed” by a set of sequential steps/iterations. For example a step can represent the processing of a MRI image, the analysis of an HEP event, the processing of a file, etc. In these cases it may be worth to save the state of the job after each of these steps. A checkpointable application must be able, of course, to restart itself from a previously saved state: using a specific function call (see section 4.3) a previously saved state can be “loaded”, and the application must be able to restart from this “intermediate” state.

In the “trivial” checkpointing service that we are going to provide, a state is defined by the user, and it is represented by a list of `<var, value>` pairs.

It is up to the user to define which `<var, value>` pairs must be saved as state of the job: he is the only one who knows which are the right ones. They must allow to represent exactly what that job has done until that moment, and they must be chosen by the user in such a way that, relying on them, the job can restart later its processing from this intermediate state.

The hypothesis is also that a job is represented by a single state.

4.2 Job checkpointing scenario.

In this section we describe a typical scenario for job checkpointing, considering a user perspective (some discussions on architecture are presented in section 6).

When a user submits a checkpointable job to the Grid, using the `dg-job-submit` command, first of all he must “declare” the job as checkpointable. This must be specified in the JDL expression for that job, with a specific attribute:

$$JobType = Checkpointable$$

The user code must include calls to the API functions that we are going to provide, and therefore it must be linked with a specific library.

In his code, the user has to define what is a state, that is he has to specify which `<var, value>` pairs are needed to represent exactly what that job has done until that moment and are “enough” to restart later the processing from an intermediate state: this is done using the `save_value` function of the API (see section 4.3).

Then the code must include calls to persistently save, from time to time, the state of the job: this is done using the `save_state` function (see section 4.3).

As mentioned in section 4.1 many applications can be seen as “composed” by a set of sequential steps, where for example a step can represent the processing of a file, the analysis of an HEP event, etc. The various steps can be represented by a `main_stepper` set of iterations, and the iterations through the various steps can be done using a `get_next_step` function (described in section 4.3). It is usually worth to save the state of the job after each step.

If the user wants to exploit the *get_next_step* function, to iterate between the various steps, he must define the content of *main_stepper*. This is done in the JDL expression with a proper attribute:

$$JobSteps = \{label_1, label_2, label_3, \dots\}$$

The checkpointing framework is useful when a job ends in an “abnormal” way, that is when it exits before it completes, because of an “external” problem (e.g. a machine crash). If (and only if) it is clear that the problem was “external” to the job, then the job is automatically rescheduled by the RB (possibly on a CE different than the one where the problem happened) and resubmitted. If a state for that job was saved in its previous execution, the job doesn’t need to start from the beginning, but it can start from the “point” corresponding to the last saved state. This means that, first of all, the last saved state is “loaded”. Then the user’s code, using this “retrieved” state, must be able to start from the point corresponding to this intermediate state (if the *main_stepper* mechanism is used, it is just necessary to “jump” to the next iteration, using the *get_next_step* function: see section 4.4 for an example).

Unfortunately it is not so straightforward to “automatically” (by the Grid middleware) understand when a job ends in an “abnormal” way. As explained in section 6, only few failure scenarios can be unambiguously identified as problems “external” to the job. In the other cases the Grid middleware can only report that the job has been completed in a normal way, and this couldn’t be true. If this is the case (i.e. some problems occurred), the user can retrieve an intermediate state for this job (usually the last saved one), and resubmit the job, pointing out that it must start using this “intermediate” state.

The “retrieving” of a state of a job (saved during job execution) can be done using a *dg-get-job-chkpt* command:

```
dg-get-job-chkpt < dg_jobID > [-ns state_number] [-ckpt file_path]
```

Using this command, the user retrieves the last state, or a previous one according to the value of *state_number* (-1 for the last but one saved state, -2 for the last but two saved state, etc.) of the job *dg_jobID*. This state is “stored” in the file system of the machine from where the command was issued, in the file specified by *file_path* (in this file the values of the *State* object data members are “dumped”).

This file should then be specified when the job is resubmitted:

```
dg-job-submit < job_description_file > [-ckpt file_path] [-resource res_id]
[-notify e_mail_address] [-verbose]
```

The option *-ckpt* is therefore used to specify the file where the initial state that must be considered is “stored” (the one specified when the state was retrieved using the command *dg-get-job-chkpt*).

It is possible, using the command *dg-get-job-chkpt*, to retrieve a state different that the last saved one, since the user could find that the problem in his job was due to some errors in the code, and the last saved state for the job can not be used, since it is “affected” by the buggy code, while a previous saved state is still correct, and therefore can be used.

4.3 Job checkpointing API.

In the checkpointing framework that we are proposing, we can define the *state* of a job through the use of an abstract object:

Object State:

```
{
  // Data Members
  Label_t state_id = ``label``;
  VarValueSet var_value_pairs[] =
    { ``var1``=``value1``, ``var2``=``value2``, ... };
  StepsSet main_stepper =
    { ``element1``, ``element2``, ``element3``, ... };
  Label_t current_step;

  // Methods
  int save_value(Pair);
  int save_state();
  string get_string_value(string);
  int get_int_value(string);
  double get_double_value(string);
  State load_state(Label_t);
  Label_t get_next_step();
  int set_final_state();
  bool is_final_state(Label_t);
  float get_state_size();
  float get_max_state_size();
}
```

Each state has a name that identifies it unambiguously: the first data member (*state_id*) represents such identifier, and it is set automatically when the object is created. This label is then used by the *load_state* method (see below).

The *var_value_pairs* data member represents the set of <var, value> pairs, that is the variables with their values, used to describe the status of the job. We don't put any restrictions on the types and also on the names of the variables. E.g.:

```
VarValueSet var_value_pairs[] =
  { ``calibration file``=``LF:<LFN>``,
    ``data_card``=``XYX``,
    ``sum``=``3212.32``, ... }
```

main_stepper defines the list of steps/iterations that must be considered to process this job (to be used for those jobs which can be represented by a set of steps). A step could represent,

for example, an HEP event/a set of events that must be analyzed, an isochromat that must be processed, an image's name etc. *main_stepper* can be defined as a list of elements, or as a numeric range. *main_stepper* could also be seen as a list of labels defining some points inside the code of the job. So when a job starts, from a given state, it can “jump” to the right label, and starts its computation from that point. E.g.:

```
StepsSet main_stepper={'file1', 'file2', 'file3'}
StepsSet main_stepper={'event1', 'event2'}
StepsSet main_stepper={'label1', 'label2'}
StepsSet main_stepper={1..1000000}
```

The last data member (*current_step*) of the object represents an element of the *main_stepper* set; it represents the current step (i.e. the step that is considered by the job at that moment).

The methods for this *State* object are:

int save_value(Pair pair)

This function saves the pair *pair* in the *var_value_pairs* data member. The argument *pair* is a pair of strings which define respectively the name and the value of the variable that must be saved in the job state. If the variable already exists, its value is updated; otherwise the variable is added to the *var_value_pairs* set.

int save_state()

This function saves persistently the current *State* object.

string get_string_value(string var_name)

int get_int_value(string var_name)

double get_double_value(string var_name)

These functions return the value of the variable *var_name* stored in the *var_value_pair* set with name *var_name*; they return respectively a string value, an int value or a double value.

State load_state(Label_T stateid)

This function “retrieves” a state for a job, whose identifier is the one specified as argument.

Label_t get_next_step()

This function returns the next element of the *main_stepper* set (i.e. the next step that must be considered by the job), or “*NULL*” if the job is in its last step.

int set_final_state()

This function is used to specify that the state is the last one for that job. This method and the next one (*is_final_state*) are used in particular in the context of job partitioning (see section 7.2).

bool is_final_state(Label_t stateid)

This function is used to check if the state specified as argument is the last one for that job (i.e. it has been “marked” using the *set_final_state* method).

float get_state_size()

This method returns the size of the current *State* object. This is used (with the *get_max_state_size* method) to be sure that the state can be persistently saved (see section 7.5).

float get_max_state_size()

This function returns the maximum size of a state that can be persistently saved.

4.4 Examples on job checkpointing.

We present here a very simple pseudo-code example to describe how the APIs introduced in section 4.3 can be used.

```
#include <api.h>

/* This job analyze some events calculating the total
   number of collisions which occur in these events */

Label_t event;
State jobstate;

if (jobstate.get_int_value(``collision``))
    then sum = jobstate.get_int_value(``collision``)
    else sum = 0;

while ((event = jobstate.get_next_step()) != NULL) {
    col = calculate_collision(event);
    sum = sum + col;
    <other processing for this event>
    print("The last event has generated " + col + " collisions");
    print("The total number of collisions is " + sum);
    /* save an intermediate state */
    jobstate.save_value(``collision``, sum);
    jobstate.save_state();
}

jobstate.set_final_state();
jobstate.save_state();
```

In this example, first of all a State object (*jobstate*) is instantiated. This means that the last previously saved state is retrieved (the data members are initialized). Note that a saved state is always available, since it is assumed that, when a job is created, a state for this job is automatically created, just initializing the *main_stepper* data member, using the value specified in the *JobSteps* attribute of the JDL expression (see section 6 for more details). Supposing that, for this specific example, the job has to analyze 1000000 events, in the JDL for this job is would have been specified:

```
JobSteps={1..1000000};
```

and therefore *main_stepper* was initialized as:

```
StepsSet main_stepper={1..1000000}
```

For this particular example we assume that the state is represented by the number of collisions found so far.

Then the number of collisions found so far is updated with the value read from the “retrieved” state, or initialized as 0, if this value is not available.

Then for each iteration, defined by the while loop (i.e. for each event that must be processed) the number of collisions is found, and added to the number of collisions found so far. Some other processing regarding that event are performed as well. At the end of every iteration the state of the job (in this example represented by the number of collisions found) is persistently saved using the *save_state* function. The function *get_next_iterator* is used “to go” to the next iteration that must be considered (i.e. to the next event that must be processed, in this particular example).

Therefore if the job doesn’t end in a “normal” way, it can be resumed later. In fact, when the job restarts its execution, the last saved state is automatically retrieved, and then using the function *get_next_iterator* it is possible “to jump” to the next event to analyze.

After the last iteration the *set_final_state* method is used, to specify that this is the final state for the job.

In the following example, on the contrary, we don’t rely on *main_stepper*. The application is “composed” by a set of *n* sequential instruction sets (<instruction set 1>, <instruction set 2>, ..., <instruction set n>). After each step the “label” (represented by the variable *step*) of the next step is stored in the *jobstate* object, which is then persistently saved by calling the *save_state* method. As first instruction, the value of the *step* variable is read from the last saved state automatically retrieved (is available), to see where the computation must start. Note the *set_final_state* call just before saving the final state of the job:

```
State jobstate;
int step = jobstate.get_int_value( "step" );

switch( step )
{
  case 1:
    /*
      <Instruction set 1>;
      jobstate.save_value( "step", 2);
      jobstate.save_state();
    */
  case 2:
    /*
      <Instruction set 2>;
      jobstate.save_value( "step", 3);
      jobstate.save_state();
    */
  [...]
}
```

```

case n:
  /*
    <Instruction set n>;
    jobstate.set_final_state();
    jobstate.save_state();
  */
}

```

5 Job partitioning.

As reported in section 3, we don't think that a generic service, suitable for all applications and for all environments able to "decompose" a job in sub-jobs, can be implemented. Instead, we think that the APIs introduced in section 4.3 for the checkpointing problem, can be exploited also in the context of job partitioning.

As we have already mentioned, the idea is that the processing of a job should be described as a set of **independent** steps/iterations. The goal is to exploit this characteristic, considering different, simultaneous, independent sub-jobs, each one taking care of a step or of a sub-set of steps, and which can be executed in parallel. The "partial" results (that is the results of the various sub-jobs) can be represented by job states (the "final" job states of the various sub-jobs), as defined in section 4.1, and which can then be "merged" together by a "job aggregator".

We want to stress again that the hypothesis is, of course, that the various steps are independent, and therefore can be "processed" in parallel by different independent jobs.

5.1 Job partitioning scenario.

When a user submits a partitionable job to the Grid, first of all he must "declare" the job as partitionable. This is done using a specific JDL attribute:

$$JobType = Partitionable$$

Then in the JDL file the characteristics and the requirements of the partitionable job must be specified, as usual. User can also specify, in the JDL file, the "job aggregator", that is the job which is responsible to collect and "merge" together the results for the various sub-jobs in which the original job is being partitioned.

In the same way, a "pre-job", that is a job which must be executed before the various sub-jobs, can be specified as well.

For what concerns the partitionable job, this is specified using the usual attributes (*executable*, *stdin*, *stdout*, *stderr*, *Requirements*, *Rank*, *InputSandbox*. etc.), apart from the *OutputSandbox* attribute, which is not "applicable" here. If some files have to be "transferred" from the sub-jobs to the job aggregator, they have to be saved in a storage element, and the identifiers of these files can be provided to the job aggregator via specific <var, value> pairs in the sub-jobs' states.

Then the user must specify which are the independent steps/iterations which "compose" the partitionable job. This is done via a specific JDL attribute:

$$JobSteps = \{element_1, element_2, \dots, element_n\}$$

As described in section 6.2, the “weights” for the elements of the *JobSteps* list can be specified as well:

$$StepWeight = \{weight_1, weight_2, \dots, weight_n\}$$

as an “hint” to “help” the partitioning process, responsible to partition the list *JobSteps* composed by n elements in to m sub-lists. If the user doesn’t specify this attribute, then all the elements are considered with the same weight.

For what concerns the job aggregator and the pre-job, they are described in the JDL expression specifying their attributes within respectively the *PreJob* attribute and the *Aggregator* attribute, e.g.:

```
JobType = Partitionable;
Executable = ...;
JobSteps = ...;
StepWeight = ...;
Requirements = ...;
...
...
Prejob =
[
  Executable = ...
  Requirements = ...;
  ...
  ...
Aggregator =
[
  Executable = ...
  Requirements = ...;
  ...
  ...
];
```

Once the user has submitted the job, using the usual *dg-job-submit* command, a specific service is then responsible to partition the list specified as *JobSteps* in the JDL expression into sub-lists, each one then corresponding to a sub-job (in section 6.2 it is discussed how this could be done).

At this point a DAG is created (i.e. the corresponding JDL expression for the DAG is created) and submitted to the Grid. The resulting DAG will look like as the one represented in figure 5.1.

PJ represents the pre-job. *SJ₁*, *SJ₂*, ..., *SJ_m* represent the sub-jobs in which the “original” partitionable job has been “decomposed”, while *AJ* represents the job aggregator.

The code for the partitionable job should rely on the checkpointing API described in section 4.3. In particular at the end of its computation, each sub-job **must save** its state, by calling the *save_state* method.

When all sub-jobs have completed their computation (and each of them has saved its final state) using the WP1 DAGMan mechanisms, the scheduling and the submission of the job aggregator is automatically triggered.

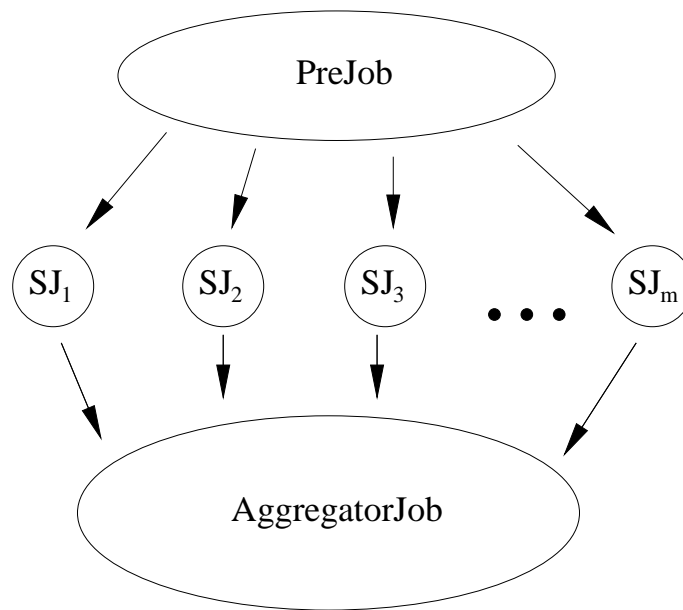


Figure 1: DAG created for a partitionable jobs

The job aggregator will “load” (using the *load_state* function) the final states of the various sub-jobs (specified in the *JobSteps* attribute of its JDL expression, and therefore stored in the *main_stepper* of its initial state) and will “collect” and/or “merge” together the results of the various sub-jobs (see section 5.2).

5.2 Example of job aggregator.

In this section we show the pseudo-code of a possible job aggregator:

```

#include <api.h>

State AggState;
Label_t SubJobId;

while (SubJobId = AggState.get_next_step()) {
    State tmpstate ( AggState.load_state((SubJobId)));

    if (tmpstate.is_final_state())
        then {
            // the next instruction uses
            // tmpstate.get_{int, double, string}_value() functions
            processing_state(tmpstate);
        }
    else {
        print(`Error: at least one sub-job didn't save its final
            state`);
    }
}

```

```
        exit(1);
    }
}

print_overall_result();
```

First of all an object *State* (*AggState*) for the aggregator job is instantiated. Therefore its initial state is retrieved, and the *main_stepper* data member is initialized with the identifiers of the final states of the various sub-jobs, which were specified as *JobSteps* in the JDL expression.

Then these sub-jobs' states are retrieved (using the *load_state* function). If the retrieved state is the final one (this is checked using the *is_final_state* method) then the `<var, value>` pairs associated to it are in some way processed, to “produce” the overall result of the partitionable initial job. Otherwise the job aggregator is aborted.

6 Architecture and technical aspects.

In this section we describe how the proposed checkpointing and partitioning frameworks could be implemented, in the context of the existing architecture of the WP1 Workload Management System. We first discuss about common aspects, while in subsections 6.1 and 6.2 items concerning only respectively job checkpointing and job partitioning are presented.

From the description given in sections 4 and 5, it is clear that the main functionality that must be provided is the possibility to persistently save the state of a job, and to retrieve a previously saved state.

For this purpose the LB service can be used, adding one or more new “fields” for each entry (i.e. for each job) to store its states.

The LB service, for each job, should also include as new attributes:

- when the last state for this job has been saved
- a counter, used to avoid infinite resubmissions (see section 6.1)

As described in section 4.2, multiple states for the same job can be kept in the LB server: for each job its last n states are persistently stored (where n is a configurable parameter of the LB server) in a cyclic way (when a new state must be saved, and there aren't free slots for that job, the oldest state for that job is overwritten).

When the function *save_state* is called from a user's job, the various `<var, value>` pairs corresponding to the *State* object data members, are saved in the LB server. In the LB server, the attribute which specifies when the last state was saved is updated as well. Moreover the counter (used to avoid infinite resubmissions) is reset to 0.

As introduced in section 4.4, a state is not created only when a *save_state* is issued: when a job is submitted to the Grid, and therefore when an entry for this job is created in the LB service, an “empty” state object for this job is “automatically” created as well. The *main_stepper* data member of this *State* object is initialized with the value of *JobSteps* specified in the JDL expression, while *var_value_pairs* is initialized as empty set.

When a *State* object is instantiated in a user's job, the last saved state for this job (which exists for sure, since, as explained above, an initial state is created when the job is submitted) must be automatically retrieved. For this purpose the LB server is contacted: if the counter used to avoid infinite resubmissions has reached a certain (configurable) threshold, then the operation fails; otherwise the counter is updated (adding 1).

A similar scenario happens when the *load_state* method is issued: the last saved state for the required job is retrieved.

When a job completes its execution, its last n states are kept in the LB server, in order to let the user retrieve them later. They are purged from the LB server just after a certain (configurable in the LB server) time (e.g. 1 month) after job completion.

6.1 Job checkpointing.

As introduced in section 4.2, when it is found that a job didn't complete its execution in a "normal" way because of an "external" problem to the job itself, the job must be automatically resubmitted. Unfortunately it is not straightforward to understand when such failures, "external" to the job, happen. For example, no assumptions can be done on the exit code of the job to understand if it completed its execution in the expected way. Only the user, checking the exit code of his job, checking the output and/or log files, etc. can understand if the job terminated as expected. For the time being, just two different failures can be unambiguously "detected" as due to problems external to the job:

- The remote resource where the execution was taking place can't be contacted anymore (i.e. a *Globus down* signal is detected). In this case the job is considered as failed, it is removed from the job queue, and therefore it should be automatically resubmitted, starting from its last saved state (if any).
- In the current implementation, a user job is "wrapped" in a script which is responsible to transfer the InputSandbox files, to run the user job, and then to copy back the output files of the OutputSandbox. If it is detected that not all these steps haven't been done, it means that something failed, and therefore the job should be automatically resubmitted.

When it is detected that a checkpointable job failed for a problem external to the job, then the RB performs again the matchmaking for this job, possibly choosing as target resource a CE different than the one where the failure happened. The job is then submitted to the chosen (by the RB) CE, using the same *dg-job-id*. So, when the job starts its execution, when a *State* object is instantiated, the last saved state for that job is automatically "loaded", and the job can restart its execution from the point corresponding to this state.

As mentioned in section 6, to avoid infinite resubmissions, a counter is used: in this way it is possible to check how many times the job restarts its execution from the same state. If this counter exceeds a certain threshold, the job is cancelled.

To be able to resubmit a job which failed for some reasons, but which isn't automatically resubmitted (because the problem which occurred to the job couldn't be detected as "external" problem to the job, or because the problem was "internal" to the job) the user can first of all retrieve the last (or a previous one) saved state for the job, and then resubmit the job, specifying that the "retrieved" state must be considered as initial state (see section 4.2).

The job state retrieval can be performed using the *dg-get-job-chkpt* command.

The retrieved state (represented by a file where the various `<var, value >` pairs have been “dumped”) can then be specified to resubmit the job (as `-chkpt` argument of the `dg-job-submit` command, as explained in section 4.2). In this case, in the LB server the specified state will be stored as initial state for this job (and therefore it will be “loaded” by the job when a *State* object will be instantiated).

6.2 Job partitioning.

When a partitionable job is submitted, as discussed in section 5.1, the first problem is to partition the *JobSteps* list in to sub-lists, each one corresponding to a sub-job.

The strategies, the policies, the parameters that should/could be taken into account in such partitioning can be various, and will be investigated to find which are the best approaches. As a starting point, just as a proof of concept for the whole framework, we propose to partition the *JobSteps* list in m sub-lists, where m represents the total number of available CPUs for all the CEs satisfying the requirements for this job. How to optimize the partition of the n elements of the *JobSteps* list into m sub-lists is not trivial, since the Grid doesn't know anything about the complexity, the time taken to process a single element (these times can be different for the different elements of the *JobSteps* list, etc.). A “hint” could be given by the user, to help the Grid middleware in performing a “good” partitioning: as reported in section 5.1, the user can specify in the JDL expression a list of “weights” for the various elements of the *JobSteps* list. This can be done via a new JDL attribute, which is a list of n numbers: the first element of this list represents the weight for the first element of the *JobSteps* list, etc. E.g.:

$$StepWeight = \{10, 134, 2, \dots\}$$

If the user doesn't specify this attribute, then all the elements are considered with the same weight, so a simple partitioning will be done: each sub-list will have $\frac{n}{m}$ elements.

Other possible approaches and algorithms will be investigated in the future.

The next step is the definition of a DAG, composed by $m + 2$ nodes: m nodes representing the m sub-jobs (which can be executed in parallel) and 2 other nodes, representing the pre-job (which is executed before the m sub-jobs), and the job aggregator (which can start its execution when the other m sub-jobs have terminated their run). Defining the DAG means creating its JDL expression, where the nodes and the dependencies between them are described, and where the JDL for each single node is specified.

Supposing that in the JDL expression of the “original” partitionable job it was specified:

$$JobSteps = \{el_1, el_2, el_3, el_4, el_5\}$$

and this lists is partitioned in 2 sub-lists:

$$\{el_1, el_2\}$$
$$\{el_3, el_4, el_5\}$$

a DAG with 2 sub-jobs and in case a pre-job and an aggregator job will be created.

In the JDL expression for the first sub-job it will be specified, besides the other parameters (*Executable*, *Requirements*, etc.):

$$JobType = Checkpointable$$

$$JobSteps = \{el_1, el_2\}$$

The JDL expression for the second sub-job will be the same that the one for the first sub-job, apart from the *JobSteps* attribute, which will be:

$$JobSteps = \{el_3, el_4, el_5\}$$

In the JDL expression of the job aggregator, as *JobSteps* the list of identifiers for the final states of the two sub-jobs will be specified:

$$JobSteps = \{stateid_1, stateid_2\}$$

Then the DAG is submitted to the RB. The hypothesis is, of course, that the matchmaking is performed for each node of the DAG, and therefore in general the various nodes can be submitted to different CEs (“lazy” scheduling). The various sub-jobs must save their final state, using the *save_state* call. When all the sub-jobs have completed their execution, and they all have saved their final state, the job aggregator can start its execution (this is triggered by the WP1 DAGMan mechanisms). This job will have to “retrieve” the final states of the various sub-jobs (specified in *main_stepper* of its initial state). This is done using the *load_state* method.

The job aggregator will then perform some processing to collect together the results (represented by the final states) of the various sub-jobs.

7 Issues.

In this section we discuss about some issues that should be investigated or better clarified. For some of them a possible solution is presented.

7.1 Specification of *JobSteps* for the job aggregator.

As reported in section 6.2, in the JDL for the job aggregator, as *JobSteps* attribute the final states of the various sub-jobs must be specified. The problem is how to define these final states of these sub-jobs. As represented in the pseudo-code of the example in section 5.2, the *get_next_step* method goes through these “values”, and are then used to load the states of the sub-jobs (using the *load_state* method). Therefore these values represent the identifier of the states that must be retrieved.

A possible approach is to use as state identifier the *dg-job-id* of a job. Therefore, with the hypothesis that, when the DAG is built, the *dg-job-id* of the various sub-jobs is known (could be *<DAGId>.1*, *<DAGId>.2*, ..., where *<DAGId>* is the *groupId* for the “initial” checkpointable job submitted by the user), then the *JobSteps* attribute for the job aggregator can be defined.

7.2 How to be sure that the sub-jobs have saved their final state.

The aggregator job must start when all the sub-jobs have completed their execution. Moreover all these sub-jobs must have saved their final state in the LB server.

The start of the job aggregator just after the completion of all sub-jobs is managed by the WP1 DAGMan mechanisms, and therefore this is not an issue.

The check to control if all the sub-jobs have saved their final states can be done by the job aggregator, by using the *is_final_state* method, assuming that in the code of the sub-jobs the final state has been “identified” using the *set_final_state* method.

Using this approach it is also possible to “manage” scenarios where the job aggregator must be run even if one job failed, or less than x % of sub-jobs failed, or there are other such conditions: it is up to the job aggregator to perform the check, and decide if it has to “collect” the intermediate results (the results of the sub-jobs who terminated their execution), or if it has to stop.

7.3 Job aggregator not specified.

If the job aggregator is not specified for a partitionable job, the user should in any case be allowed to retrieve the last state for the various sub-jobs, that is he must be allowed to issue a *dg-get-job-chkpt*, specifying as *dg-job-id* the identifier of the partitionable job returned when the job was submitted. This means that, given the *dg-job-id* for the “original” partitionable job, the system must be able to know the *dg-job-id* of the various sub-jobs¹.

7.4 How to avoid that all sub-jobs are submitted to the same CE.

As reported in section 6.2, the matchmaking is performed for each node of the DAG. The nodes associated to the m sub-jobs all have the same CE requirements, data requirements and rank. Since the matchmaking for these m sub-jobs is done practically at the same time, it is likely that the RB will choose the same CE for all these sub-jobs, since the CEs’ attribute *EstimatedTraversalTime* used as default rank is going to be the same in the matchmaking of all these sub-jobs (since this process is done more or less at the same time). Some mechanisms must therefore be implemented to evenly distribute the sub-jobs between all the available processors for all the CEs suitable for these sub-jobs (this problem is not specific to checkpointable jobs, but it is an issue also when a “bunch” of jobs with the same requirements are submitted at the same time).

7.5 Problems saving large size states.

As reported above, jobs’ states are saved in the LB server, when the *save_state* method is called. Problems could occur if a large size state has to be saved (e.g. the available space in the LB server can be limited, etc.): in case of failure, the *save_state* method will return a proper error.

User can check, before issuing the *save_state* function, what is the size of the current state using the *get_state_size* method, and the maximum state size that can be saved in the LB server using the *get_max_state_size* method (see section 4.3).

¹This means to be able to get the *dg-job-id* of the various nodes given the *dg-job-id* of the whole DAG