Porting of Scientific Applications to Grid Computing on Grid*Way*.*

J. Herrera [1,†] E. Huedo [2] R. S. Montero [1]

I. M. Llorente [1,2]

[1] Departamento de Arquitectura de Computadores
y Automática
Facultad de Informática, Universidad Complutense de Madrid
28040 Madrid, Spain

[2] Laboratorio de Computación Avanzada, Simulación
y Aplicaciones Telemáticas
Centro de Astrobiología (CSIC-INTA)
28850 Torrejón de Ardoz, Spain

January 11, 2005

*This research was supported by Ministerio de Ciencia y Tecnología through the research grant TIC 2003-01321 and Instituto Nacional de Técnica Aeroespacial "Esteban Terradas" (INTA) – Centro de Astrobiología.

†Corresponding author. Tel: +34 91 394 75 41. Fax: +34 91 394 76 42. E-mail: jherrera@fdi.ucm.es.

Abstract

The expansion and adoption of Grid technologies is prevented by the lack of a standard programming paradigm to port existing applications among different environments. The Distributed Resource Management Application API has been proposed to aid the rapid development and distribution of these applications across different Distributed Resource Management Systems. In this paper we describe an implementation of the DRMAA standard on a Globus-based testbed, and show its suitability to express typical scientific applications, like High-Throughput and Master-Worker applications. The DRMAA routines are supported by the functionality offered by the GridWay[1] framework, which provides the runtime mechanisms needed for transparently executing jobs on a dynamic Grid environment based on Globus. As cases of study, we consider the implementation with DRMAA of a bioinformatics application, a genetic algorithm and the NAS Grid Benchmarks.

1 Introduction

The deployment of existing applications across the Grid continues requiring a high level of expertise and a significant amount of effort, mainly due to the characteristics of the Grid: complexity, heterogeneity, dynamism, high fault rate, etc. To deal with such characteristics, we have developed Grid*W*ay [9]: a Globus submission framework that allows an easier and more efficient execution of jobs on dynamic Grid environments. Grid*W*ay automatically performs all the job scheduling steps [18], provides fault recovery mechanisms, and adapts job scheduling and execution to the changing Grid conditions.

On the other hand, the lack of a standard programming paradigm for the Grid has prevented the portability of existing applications among different en-

[1] http://www.gridway.org

vironments. The *Distributed Resource Management Application API Working Group* (DRMAA-WG)[2], within the *Global Grid Forum* (GGF)[3], has recently developed an API specification for job submission, monitoring and control that provides a high level interface with *Distributed Resource Management Systems* (DRMS). In this way, DRMAA, or higher level tools that use DRMAA, could aid scientists and engineers to express their computational problems by providing a portable direct interface to a DRMS. It is foreseeable, as it happened with other standards like MPI or OpenMP, that DRMAA will be progressively adopted by most DRMS, making them easier and worthier to learn, thus lowering its barrier to acceptance, and making Grid applications portable across DRMS adhered to the standard.

In this work, we discuss several aspects of the implementation of DRMAA within the Grid*W*ay framework, and investigate the suitability of the DRMAA specification to distribute typical scientific workloads across the Grid. We demonstrate the ability of the Grid*W*ay framework when executing different computational workloads distributed using DRMAA. The examples shown resemble typical scientific problems whose structure is well suited to the Grid architecture. The experiments were conducted in the UCM-CAB research testbed, based on the Globus Toolkit [7], briefly described in Tables 1 and 2.

In Section 2, we first analyze several aspects involved in the efficient execution of distributed applications related to the barriers to use the Grid, and how they are overcome by the Grid*W*ay framework. Section 3 briefly describes the DRMAA standard, and the development and execution process adopted in this work. Then, in Sections 4 and 5, we illustrate how DRMAA can be used to implement several scientific application paradigms, like High-Throughput and Master-Worker, and provide results of real-life applications. Finally, in Sec-

[2] http://www.drmaa.org (2004)

[3] http://www.gridforum.org (2004)

tion 6, we evaluate the suitability of the DRMAA for implementing the *NAS Grid Benchmarks* (NGB) [20]. The NGB suite constitutes an excellent case-of-study, since it models distributed *communicating* applications typically executed on the Grid. The paper ends in Section 7 with some conclusions.

2 The Grid *W*ay Framework

The Grid *W*ay framework [9] provides the following techniques to allow a robust an efficient execution of jobs in heterogeneous and dynamic Grids:

- Given the dynamic characteristics of the Grid, the Grid *W*ay framework periodically adapts the schedule to the available resources and their characteristics [11]. Grid *W*ay incorporates a *resource selector* that reflects the applications demands, in terms of requirements and preferences, and the dynamic characteristics of Grid resources, in terms of load, availability and proximity (bandwidth and latency) [15].

- The Grid *W*ay framework also provides adaptive job execution to migrate running applications to more suitable resources. So improving application performance by adapting it to the dynamic availability, capacity and cost of Grid resources. Moreover, an application can migrate to a new resource to satisfy its new requirements or preferences [9].

- Grid *W*ay also provides the application with fault tolerance capabilities by capturing GRAM callbacks, by periodically probing the GRAM job-manager, and by inspecting the output of each job.

The aim of the Grid *W*ay project is similar to that of other projects [4, 5, 14]: simplify distributed heterogeneous computing. However, it has some remarkable differences. Our framework provides a submission agent that incorporates the runtime mechanisms needed for transparently executing jobs in a Grid by

combining both adaptive scheduling and execution. Our modular architecture for job adaptation to a dynamic environment presents the following advantages:

- It is not bounded to a specific class of application generated by a given programming environment, which extends its application range.
- It does not require new services, apart from Globus basic services, which considerably simplify its deployment.
- It does not necessarily require code changes, which allows reusing of existing software.
- It is extensible, which allows its communication with the Grid services available in a given testbed.

We would like to mention that the experimental framework does not require new system software to be installed in the Grid resources. The framework is currently functional on any Grid testbed based on Globus. We believe that this is an important advantage because of socio-political issues; cooperation between different research centers, administrators and users is always difficult.

3 Distributed Resource Management Application API

One of the most important aspects of Grid Computing is its potential ability to execute distributed communicating jobs. The DRMAA specification constitutes a homogenous interface to different DRMS to handle job submission, monitoring and control, and retrieval of finished job status. In this sense the DRMAA standard represents a suitable and portable framework to express this kind of distributed computations.

In the following list we describe the DRMAA interface routines implemented within the Grid*W*ay framework:

- Initialization and finalization routines: `drmaa_init` and `drmaa_exit`.
- Job template routines: `drmaa_get_attribute` and `drmaa_set_attribute`, `drmaa_allocate_job_template` and `drmaa_delete_job_template`. This routines enable the manipulation of job definition entities (job templates) to set parameters such as the executable, its arguments or the standard output streams.
- Job submission routines: `drmaa_run_job` and `drmaa_run_bulk_jobs`. *Bulk* jobs are defined as a group of n similar jobs, sharing the same job template, with a separate job id.
- Job control and monitoring routines: `drmaa_control`, `drmaa_synchronize`, `drmaa_wait` and `drmaa_job_ps`. This routines are used to control (killing, resuming, suspending, etc..) and synchronize jobs, and monitor their status.

The DRMAA interface (see [16] for a detailed description of the C API) includes more routines in some of the above categories as well as auxiliary routines that provides textual representation of errors, not implemented in the current version. All the functions implemented in the Grid*W*ay framework are thread-safe.

Although DRMAA could interface with DRMS at different levels, for example at the intranet level with SGE or Condor, in the present context we will only consider its application at Grid level. In this way, the DRMS (Grid*W*ay in our case) will interact with the local job managers (Condor, PBS, SGE...) through the Grid middleware (Globus). This development and execution scheme with DRMAA, Grid*W*ay and Globus is depicted in figure 1. There are several

projects underway to implement the DRMAA specification on different DRMS, like Sun Grid Engine (SGE) or Condor. However, to the best of the authors' knowledge, DRMAA has never been implemented in a Globus-based DRMS.

The DRMAA standard can help in exploiting the intrinsic parallelism found in some application domains, as long as the underlying DRMS is responsible for the efficient and robust execution of each job. We expect that DRMAA will allow to explore several common execution techniques when distributing applications across the Grid [1]. For example fault tolerance could be improved by replicating job executions (redundant execution) [19], the intrinsic parallelism presented in the workflow of several applications could be exploited, or several alternative task flow paths could be concurrently executed (speculative execution).

4 High-Throughput Applications

This example represents the important class of Grid applications called *Parameter Sweep Applications* (PSA), which constitute multiple independent runs of the same program, but with different input parameters. This kind of computations appears in many scientific fields like Biology, Pharmacy, or Computational Fluid Dynamics. In spite of the relatively simple structure of this applications, its efficient execution on computational Grids involves challenging issues [11].

The general structure of a PSA and its implementation with DRMAA are shown in figure 2. An initial job is submitted to perform some pre-processing tasks, and then several independent jobs are executed with different input parameters. Finally a post-processing job is executed.

4.1 A Test Case

In this case we consider an application that comprises the execution of 50

independent jobs. Each job calculates the determinant of an square matrix read from an input file (0.5MB). The experiment was conducted in the second configuration of the UCM-CAB Grid, described in Table 2. The overall execution time for the parameter sweep application is 40 minutes, with an average job turnaround time of 125 seconds. Figure 3 presents the dynamic productivity (jobs per minute) of the testbed during the execution of the PSA. Compared to the single host execution on the fastest machine in the testbed, these results represents a 35% reduction in the overall execution time.

4.2 A Real-Life Application: Computational Proteomics

Bioinformatics, which has to do with the management and analysis of huge amounts of biological data, could enormously benefit from the suitability of the Grid to execute High-Throughput applications. In the context of this paper, we consider a bioinformatics application aimed at predicting the structure and thermodynamic properties of a target protein from its amino acid sequences. The algorithm has been tested in the 5th round of Critical Assessment of techniques for protein Structure Prediction (CASP5) [3]. We have applied the algorithm to the prediction of thermodynamic properties of families of orthologous proteins, i.e. proteins performing the same function in different organisms. If a representative structure of this set is known, the algorithm predicts it as the correct structure.

Let us consider an experiment consisting in 88 tasks, each of them applies the structure prediction algorithm to a different sequence of the *Triosephosfate Isomerase* enzyme which is present in different organisms. The experiment was conducted in the first configuration of the UCM-CAB Grid, described in Table 1. The overall execution time for the bioinformatics application, when

all the machines in the testbed are available, is 7.15 hours with an average throughput of 12 jobs per hour.

This experiment was reproduced in two new situations. In the first case, babieca is shut down for maintenance in the middle of the experiment during one hour. As a consequence, the framework stops scheduling jobs in this host and the average job turnaround is reduced to 10 jobs per hour. Once babieca is restarted, Grid*W*ay schedules jobs on it again and the throughput increases to nearly 12 jobs per hour. The second case starts with pegasus unavailable, and it is *plugged* in to the Grid 3.5 hours after the experiment started. As could be expected, the absence of pegasus decreases the average throughput (9 jobs per hour), and increases the overall execution time to 9.8 hours. Figure 4 shows the dynamic job turnaround time during the execution of the application in the above situations.

5 Master-Worker Applications

We now consider a generalized Master-Worker paradigm, which is adopted by many scientific applications like genetic algorithms, N-body simulations or Monte Carlo simulations among others. A Master process assigns a description (input files) of the task to be performed by each Worker. Once all the Workers are completed, the Master process performs some computations in order to evaluate a stop criterion or to assign new tasks to more workers (see figure 5).

As an example of this paradigm, we will consider Genetic Algorithms (GA), which are search algorithms inspired in natural selection and genetic mechanisms. GAs use historic information to find new search points and reach an optimal problem solution. In order to increase the speed and the efficiency of sequential GAs, several Parallel Genetic Algorithm (PGA) alternatives have been developed. PGAs have been successfully applied in previous works, (see

for example [13]), and in most cases, they succeed to reduce the time required to find acceptable solutions.

In order to develop efficient genetic algorithms [12] for the Grid, the dynamism and heterogeneity of the environment must be considered. In this way, traditional load-balancing techniques could lead to a performance slow-down, since, in general the performance of each computing element can not be guaranteed during the execution. Moreover, some failure recovery mechanisms should be included in such a faulty environment. Taking into account the above considerations we will use a fully connected multi-deme genetic algorithm. In spite of this approach represents the most intense communication pattern (all demes exchange individuals every generation), it does not imply any overhead since the population of each deme is used as checkpoint files, and therefore transferred to the client in each iteration.

The initial population is uniformity distributed among the available number of nodes, and then a sequential GA is locally executed over each subpopulation. The resultant subpopulations are transferred back to the client, and worst individuals of each subpopulation are exchanged with the best ones of the rest. Finally, a new population is generated to perform the next iteration [6]. The experiments shown in the following subsections were performed in the second configuration of the UCM-CAB Grid, described in Table 2

5.1 A Test Case

We consider a simple distribution scheme for a genetic algorithm. The master acts as the control process by creating worker jobs. Each worker task is initiated with an identical-sized sets of individuals, and evolves the population a fixed number of iterations. The master receives the results, evaluates the fitness function, and if convergence is not achieved it exchanges some individuals and

repeats the process.

Figure 6 shows the execution profile of three generations of the above Master-Worker application. The average execution time per iteration is 120 seconds, with an average computational and transfer times per worker of 15.7, and 23.3 seconds respectively. In this case the total turnaround time is 360 seconds with an average CPU utilization of 22%.

5.2 A Grid-Oriented Genetic Algorithm

The previous algorithm may incur in performance losses when the relative computing power of the nodes involved in the solution process greatly difers, since the iteration time is determined by the slowest machine. In order to prevent these situations we allow an *asynchronous* communication pattern between demes. In this way, information exchange only occurs between a fixed number of demes, instead of synchronizing the execution of all subpopulations. The minimum number of demes that should communicate in each iteration depends strongly on the numerical characteristics of the problem. We refer to this characteristic as *dynamic connectivity*, since the demes that exchange individuals differs each iteration. The scheme and implementation of this algorithm is depicted in figure 7.

We evaluate the functionality and efficiency of the Grid-Oriented Genetic Algorithm (GOGA) described above in the solution of the One-Max problem [17]. The One-Max is a classical benchmark problem for genetic algorithm computations, and it tries to evolve an initial matrix of zeros in a matrix of ones. In our case we consider an initial population of 1000 individuals, each one a 20x100 zero matrix. The sequential GA executed on each node performs a fixed number of iterations (50), with a mutation and crossover probabilities of 0,1% and

60%, respectively. The exchange probability of best individuals between demes is 10%.

Figure 8 shows the execution profile of 4 generations of the GOGA, with a 5-way *dynamic connectivity*. Each subpopulation has been traced, and labelled with a different number (P_{deme}). As can be shown, individuals are exchanged between subpopulations $P1, P2, P3, P4, P5$ in the first generation; while in the third one the subpopulations used are $P1, P2, P4, P7, P8$. In this way the *dynamic connectivity*, introduces another degree of randomness since the demes that communicate differ each iteration and depend on the dynamism of the Grid.

6 The NAS Grid Benchmarks

The *NAS Grid Benchmarks* [8] have been presented as a data flow graph (DFG) encapsulating an instance of a *NAS Parallel Benchmarks* (NPB) [2] code in each graph node, which *communicates* with other nodes by sending/receiving initialization data. The NGB suite models applications typically executed on the Grid and therefore constitutes an excellent case-of-study for testing the functionality of the DRMAA and the environment itself.

NGB is focused on computational Grids, which are used mainly for running compute-intensive jobs that potentially process large data sets. Each benchmark comprises the execution of several NPB codes that symbolize scientific computation (flow solvers SP, BT and LU), post-processing (data smoother MG) and visualization (spectral analyzer FT). Like NPB, NGB specifies several different classes or problem sizes, in terms of mesh size and number of iterations. The four families defined in the NGB are:

- *Embarrassingly Distributed* (ED) models High-Throughput applications, whose structure and implementation with DRMAA has been discussed in

section 4.

- *Helical Chain* (HC) represents long chains of repeating processes, such as a set of flow computations that are executed one after the other, as is customary when breaking up long running simulations into series of tasks, or in computational pipelines.

- *Visualization Pipe* (VP) represents chains of compound processes, like those encountered when visualizing flow solutions as the simulation progresses.

- *Mixed Bag* (MB) again involves the sequence of flow computation, post-processing, and visualization, but now the emphasis is on introducing asymmetry.

Grid benchmarks should provide a methodology to assess the functionality, performance and quality of service provided by a Grid environment. In this work we will concentrate in testing the functionality of our testbed made up of: local schedulers (fork and PBS), middleware (Globus toolkit), and high level tools (Grid*W*ay and DRMAA). In the NGB reports presented below, for the shake of completeness, we also include some performance metrics like job turnaround time, resource usage, and data transfers and execution times. Moreover the Globus overhead, as well as the Grid*W*ay overhead (scheduling time), are included in all measurements. The experiments shown below were all conducted in the second configuration of the UCM-CAB Grid, described in Table 2.

6.1 Helical Chain

The HC benchmark consists in a sequence of jobs that model long running simulations that can be divided in different tasks. Each job in the sequence uses the computed solution of its predecessor to initialize. Considering this

dependencies each job in the chain can be scheduled by Grid*W*ay once the previous job has finished (see figure 9).

Results of the HC benchmark (class A) for this scheduling strategy are shown in figure 10. The turnaround time is 17.56 minutes, with an average resource usage of 20.21%. The MDS delay in publishing resource information results in an oscillating scheduling of the jobs. This schedule clearly reduces the performance obtained compared to the optimal turnaround time[4] of 6.18 minutes.

Nevertheless, this kind of applications can be submitted through the Grid*W*ay framework as a whole. The output files of each task in the chain are handled by the framework as checkpoint files. In this way the application can take advantage of the self-adapting capabilities provided by Grid*W*ay:

- The application can progressively change its resource requirements depending on the task of the chain to be executed. So, the application does not have to impose the most restricted set of requirements at the beginning, since it limits the chance for the application to begin execution [10].

- The application can generate a performance profile to provide monitoring information in terms of application metrics (for example time to perform each task of the chain). This performance profile can be used to guide the job scheduling. Thus, the application could migrate to other host when some resources (disk space, free CPU...) are exhausted [10].

- The application can be migrated when a *better* resource is found in the Grid. In this case the time to finalize, and file transfer costs must be considered to evaluate if the migration is worthwhile [15].

When the HC benchmark is submitted as a whole job, the average resource usage increases to 91%, since the nine tasks of the same chain are scheduled to

[4]Belonging to a serial execution on the fastest machine.

the same host (cygnus). In this case, the turnaround time is 7 minutes and the average execution time is reduced to 6.4 minutes. This supposes a decrement in the job turnaround time of 60% compared to the first scheduling strategy and an increment of only 11% compared to the optimal case.

6.2 Visualization Pipe and Mixed Bag

Although this kind of benchmarks could be serialized and adaptively executed like the previous one, they are more suitable to be implemented as a workflow application to exploit the parallelism they exhibit.

Since GridWay does not directly support workflow execution, we have developed a *workflow engine* taking advantage of the DRMAA programming interface, see figure 11. This algorithm follows a greedy approach, although different policies could be used to prioritize the jobs submitted to at each moment, for example, submit first the job with a more restricted set of requirements, with more computational work or with more jobs depending on it.

These benchmarks are combinations of the ED (fully parallel) and HC (fully sequential) benchmarks described above. They exhibit some parallelism that should be exploited, but it is limited by the dependencies between jobs. In the case of VP, the parallelism is even more limited due to the low pipe width (only 3, for all classes) and the long times to fill and drain the pipe (with class A, it only executes once with full parallelism).

Figure 14 shows the results for the VP.A benchmark. Dashed lines represent dependencies between jobs and thicker lines represent the critical path, which determines the wall time. In this case, the turnaround time is 21.68 minutes, with an average resource usage of 35.25%. Execution and transfer times are 22.93 and 8.1 minutes, respectively.

Figure 15 shows the results for the MB.A benchmark. Again, dashed lines

represent dependencies between jobs and thicker lines represent the critical path. In this case, the turnaround time is 16.8 minutes, with an average resource usage of 45.7%. Execution and transfer times are 23.03 and 9.7 minutes, respectively.

Figures 14 and 15 show the differences between the VP and MB benchmarks. Both exhibit some degree of parallelism, since the sum of execution time is greater than the wall time, that could be increased by widening the pipe (limited to three jobs) and reducing the Grid overload (scheduling, file staging, job submission...). The parallelism obtained by the VP benchmark is very poor, due to the stages of filling and draining the pipe, being the sum of the execution times only a 4.57% greater than the wall time. On the other hand, the MB benchmark reaches a considerable parallelism degree, having a sum of the execution times a 27.06% greater than the wall time. In fact, the sum of the execution time in both benchmarks is very similar (22.93 for VP and 23.03 for MB), however, the wall time is a 23.21% lower in the case of the MB benchmark, due to its greater parallelism degree from the beginning, which enables a better use of the resources (34.93% for VP, while 45.7% for MB).

7 Conclusions

DRMAA can clearly aid the rapid development and distribution across the Grid of typical scientific applications. In fact, we believe that DRMAA will become a standard for Grid application development. This would help users, making Grid applications portable across DRMS adhered to the standard, and DRMS vendors, making DRMS easier and worthier to learn.

We have presented an implementation of DRMAA on top of the Grid*Way* framework and Globus. The functionality, robustness and efficiency of this environment have been demonstrated through the execution of typical computational models, namely: High-Throughput and Master-Worker. This preliminary

study has been completed with the analysis of three real-life applications: a protein structure prediction model, a Grid-oriented genetic algorithm and the NGB suite. In these cases, DRMAA also represents a suitable and portable framework to develop scientific codes.

References

[1] R. M. Badia, J. Labarta, R. Sirvent, J. M. Cela, and R. Grima. GridSuperscalar: A Programming Paradigm for Grid Applications. In *Workshop on Grid Applications and Programming Tools (GGF8)*, pages 26–37, 2003. to be published in J. of Grid Computing.

[2] D. H. Bailey, E. Barszcz, and J. T. Barton. The NAS Parallel Benchmarks. *Intl. J. of Supercomputer Applications*, 5(3):63–73, 1991.

[3] U. Bastolla. Sequence-Structure Alignments with the Protfinder Algorithm. In *Abstracts of the 5th Community Wide Experiment on the Critical Assessment of Techniques for Protein Structure Prediction*, December 2002. Available at http://predictioncenter.llnl.gov/casp5/doc/Abstr.doc.

[4] F. Berman et al. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):369–382, 2003.

[5] R. Buyya, D.Abramson, and J. Giddy. A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker. *Future Generation Computer Systems*, 2002.

[6] Erick Cant-Paz. A Survey of Parallel Genetic Algorthms, July 1999.

[7] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. J. of Supercomputer Applications*, 11(2):115–128, 1997.

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

[9] E. Huedo, R. S. Montero, and I. M. Llorente. A Framework for Adaptive Execution on Grids. *Intl. J. of Software – Practice and Experience (SPE)*, 34:634–651, 2004.

[10] E. Huedo, R. S. Montero, and I. M. Llorente. Adaptive Scheduling and Execution on Computational Grids. *J. of Supercomputing*, 2004. (in press).

[11] E. Huedo, R. S. Montero, and I. M. Llorente. Experiences on Adaptive Grid Scheduling of Parameter Sweep Applications. In *Proc. of the 12th Intl. Conf. on Parallel, Distributed and Network based Processing (PDP 2004)*. IEEE Computer Society, February 2004.

[12] H. Imade, R. Morishita, I. Ono, N. Ono, and M. Okamoto. A Grid-oriented Genetic Algorithm Framework for Bioinformatics. *New Generation Computing*, 22(2):177–186, 2004.

[13] Lishan Kang and Yuping Chen. Parallel Evolutionary Algorithms and Applications. 1999.

[14] G. Lanfermann et al. Nomadic Migration: A New Tool for Dynamic Grid Computing. In *Proc. of the 10th Symp. on High Performance Distributed Computing (HPDC10)*, August 2001.

[15] R. S. Montero, E. Huedo, and I. M. Llorente. Grid Resource Selection for Opportunistic Job Migration. In *Proc. of the 9th Intl. Conf. on Parallel and Distributed Computing (Euro-Par 2003)*, volume 2790 of *Lecture Notes in Computer Science*, pages 366–373. Springer–Verlag, August 2003.

[16] H. Rajic et al. Distributed Resource Management Application API Specification 1.0. Technical report, DRMAA Working Group – The Global Grid Forum, 2003.

[17] J.D. Schaffer and L.J. Eshelman. On Crossover as an Evolutionary Viable Strategy. In R.K. Belew and L.B. Booker, editors, *Proc. 4th Intl. Conf. Genetic Algorithms*, pages 61–68. Morgan Kaufmann, 1991.

[18] J. M. Schopf. Ten Actions when Superscheduling. Technical Report GFD-I.4, Scheduling Working Group – The Global Grid Forum, 2001.

[19] Paul Townend and Jie Xu. Fault Tolerance within a Grid Environment. In Simon Cox, editor, *Proc. of UK e-Science All Hands Meeting, Nottingham (UK)*, September 2003.

[20] R. F. Van der Wijngaart and M. A. Frumkin. NAS Grid Benchmarks Version 1.0. Technical Report NAS-02-005, NASA Advanced Supercomputing (NAS), NASA Ames Research Center, Moffett Field, CA, 2002.

List of Tables

Table 1: Characteristics of the machines in the first configuration of the UCM-CAB research testbed.

Name	VO	Model	Speed	OS	Memory	DRMS
babieca	CAB	5×Alpha EV67	466MHz	Linux 2.2	256MB	PBS
pegasus	UCM	Intel P4	2.4GHz	Linux 2.4	1GB	fork
solea	UCM	2×Sun US-II	296MHz	Solaris 7	256MB	fork
ursa	UCM	Sun US-IIe	500MHz	Solaris 8	128MB	fork
draco	UCM	Sun US-I	167MHz	Solaris 8	128MB	fork

Table 2: Characteristics of the machines in the second configuration of the UCM-CAB research testbed.

Name	VO	Model	Speed	OS	Memory	DRMS
babieca	CAB	5×Alpha EV67	466MHz	Linux 2.2	256MB	PBS
hydrus	UCM	Intel P4	2.5GHz	Linux 2.4	512MB	fork
cygnus	UCM	Intel P4	2.5GHz	Linux 2.4	512MB	fork
cepheus	UCM	Intel PIII	600MHz	Linux 2.4	256MB	fork
aquila	UCM	Intel PIII	666MHz	Linux 2.4	128MB	fork

List of Figures

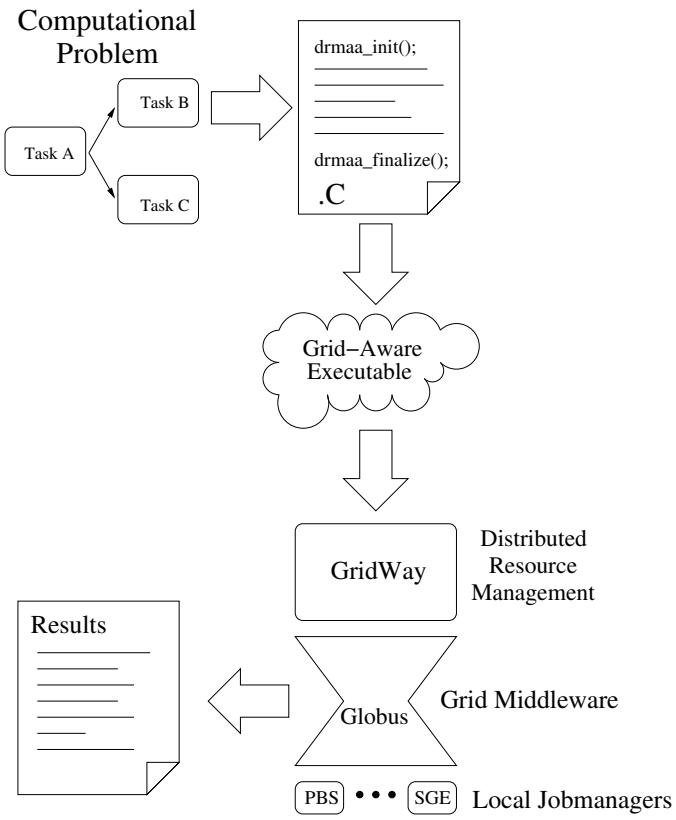
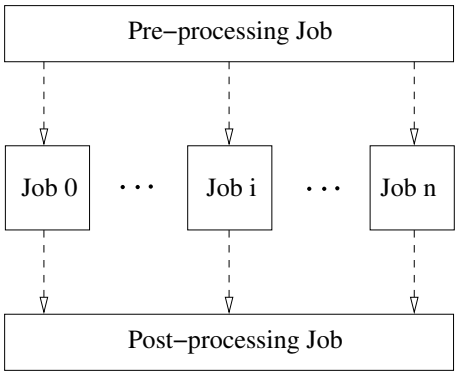


Figure 1: Development and execution cycle using the DRMAA interface




```
rc = drmaa_init(contact, err);

// Execute initial job and wait for it
rc = drmaa_run_job(job_id, jt, err);
rc = drmaa_wait(job_id, &stat, timeout,
    rusage, err);

// Execute n jobs simultaneously and wait
rc = drmaa_run_bulk_jobs(job_ids,jt,1,
    JOB_NUM,1,err);
rc = drmaa_synchronize(job_ids, timeout, 1, err);

// Execute final job and wait for it
rc = drmaa_run_job(job_id, jt, err);
rc = drmaa_wait(job_id, &stat, timeout,
    rusage, err);

rc = drmaa_exit(err_diag);
```

Figure 2: High-Throughput scheme and its codification using the DRMAA standard.

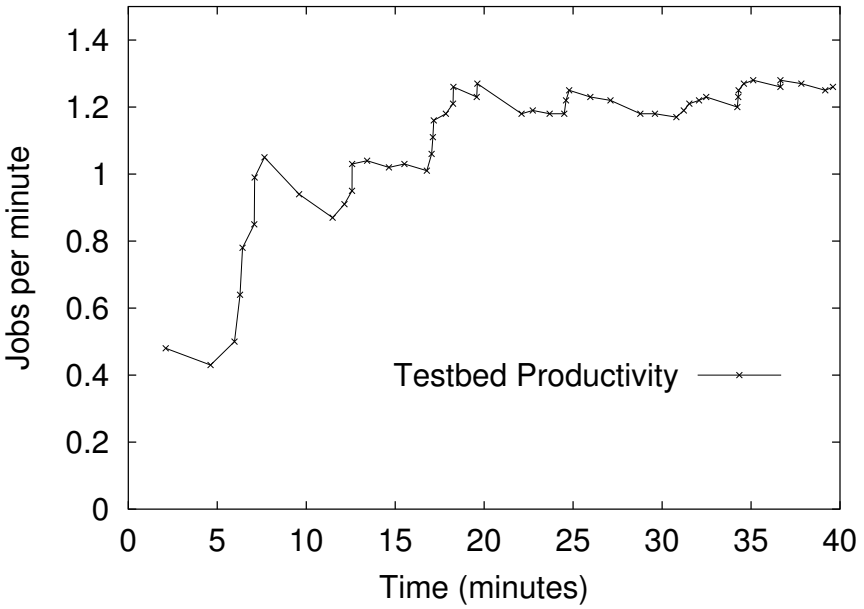


Figure 3: Testbed productivity in the execution of the High-Throughput application.

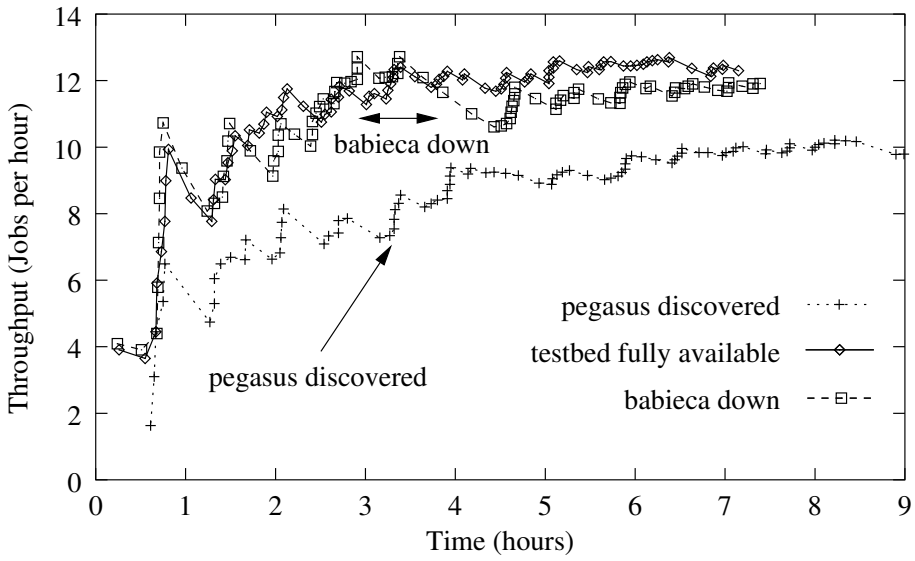
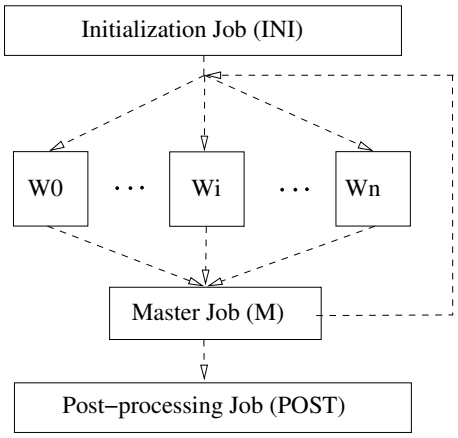


Figure 4: Dynamic throughput in the execution of the application when the testbed is fully available, when pegasus is discovered and when babieca is down.




```
rc = drmaa_init(contact, err_diag);

// Execute initial job and wait for it
rc = drmaa_run_job(job_id, jt, err_diag);
rc = drmaa_wait(job_id, &stat, timeout,
    rusage, err_diag);

while (exitstatus != 0) {
  // Execute n Workers concurrently and wait
  rc = drmaa_run_bulk_jobs(job_ids, jt, 1,
      JOB_NUM, 1, err_diag);
  rc = drmaa_synchronize(job_ids, timeout,
      1, err_diag);

  // Execute the Master, wait and get exit code
  rc = drmaa_run_job(job_id, jt, err_diag);
  rc = drmaa_wait(job_id, &stat, timeout,
      rusage, err_diag);
  rc = drmaa_wexitstatus(&exitstatus, stat,
      err_diag);
}

rc = drmaa_exit(err_diag);
```

Figure 5: Master-Worker application and its codification using the DRMAA standard.

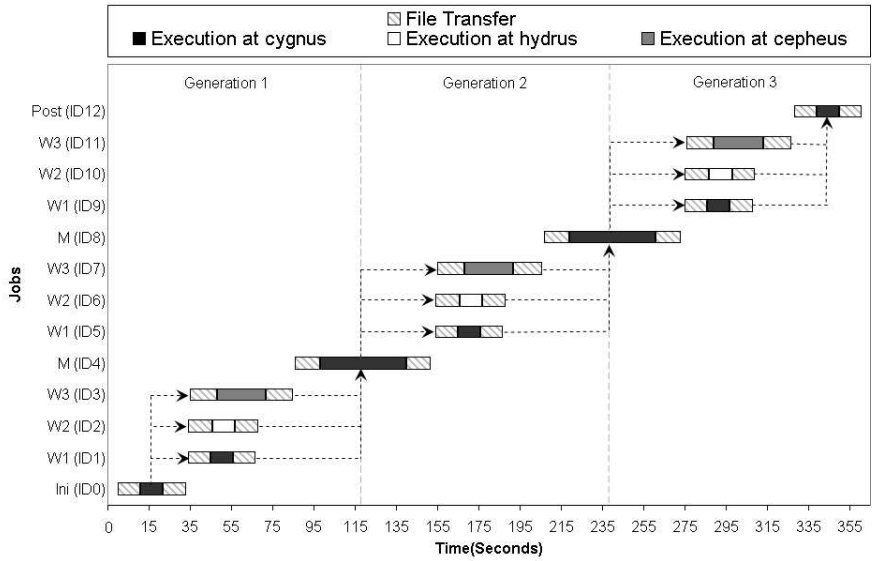
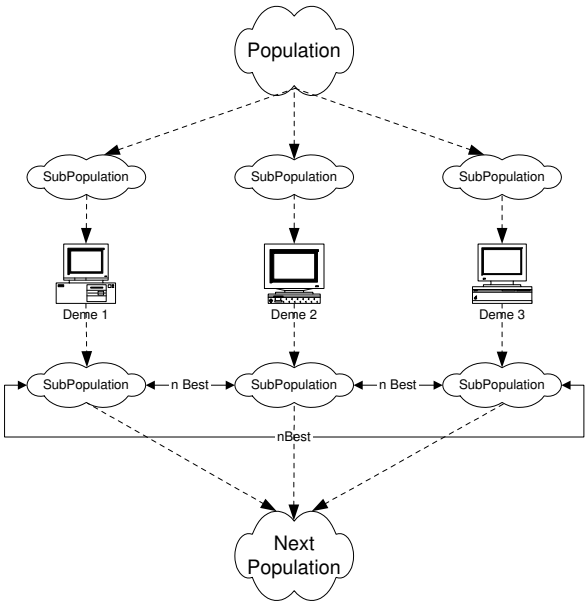


Figure 6: Execution profile for three iterations of the Master-Worker application.




```
rc = drmaa_init(contact, err_diag);

// Execute all jobs simultaneously
for (i=0; i < ALL_JOBS; i++)
  rc = drmaa_run_job(job_id, jt, err_diag);

// Execute GOGA if it doesn't rise objective function
while (!objective_function()) {
  // Wait for (dynamic connectivity degree) jobs
  //and store results
  for (i=0; i<NUM_JOBS; i++)
    rc = drmaa_wait(job_id, &stat, timeout, rusage,
        err_diag);
  store_results();

  // Execute (dynamic connectivity degree) jobs simultaneously
  for (i=0; i<NUM_JOBS; i++)
    rc = drmaa_run_job(job_id, jt, err_diag);
}

rc = drmaa_exit(err_diag);
```

Figure 7: Scheme and implementation of fully-connected multi-deme genetic algorithm

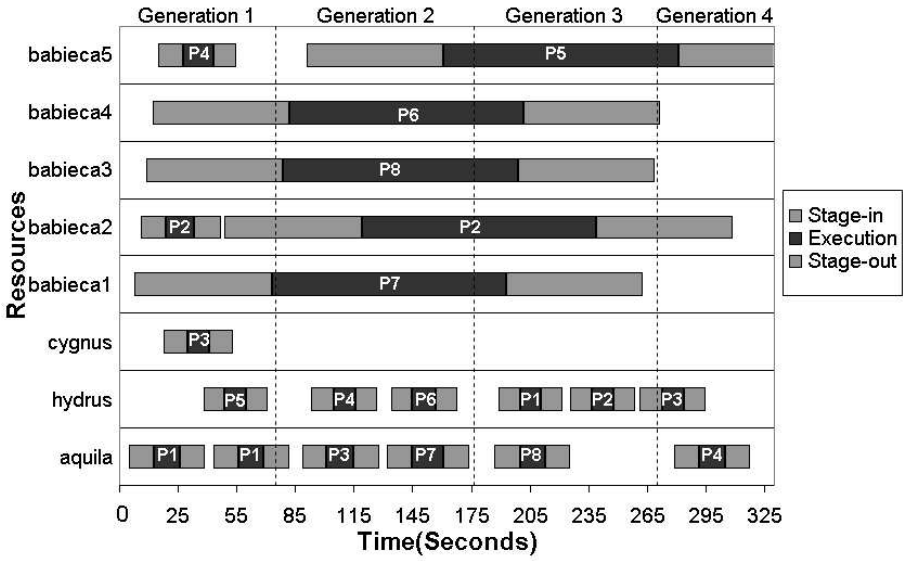
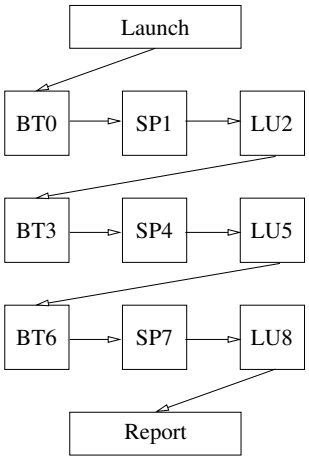


Figure 8: Execution profile of four generations of the One-Max problem, each subpopulation has been labelled with P_{deme}




```
// Initialization
jobs[0].jt = BT;
jobs[1].jt = SP;
jobs[2].jt = LU;
jobs[3].jt = BT;
jobs[4].jt = SP;
jobs[5].jt = LU;
jobs[6].jt = BT;
jobs[7].jt = SP;
jobs[8].jt = LU;

drmaa_init(contact, err);

// Submit all jobs consecutively
for (i = 0; i<9; i++) {
  drmaa_run_job(job_id, jobs[i].jt,
      err);
  drmaa_wait(job_id, &stat, timeout,
      rusage, err);
}

drmaa_exit(err_diag);
```

Figure 9: Structure and implementation of the HC benchmark using DRMAA.

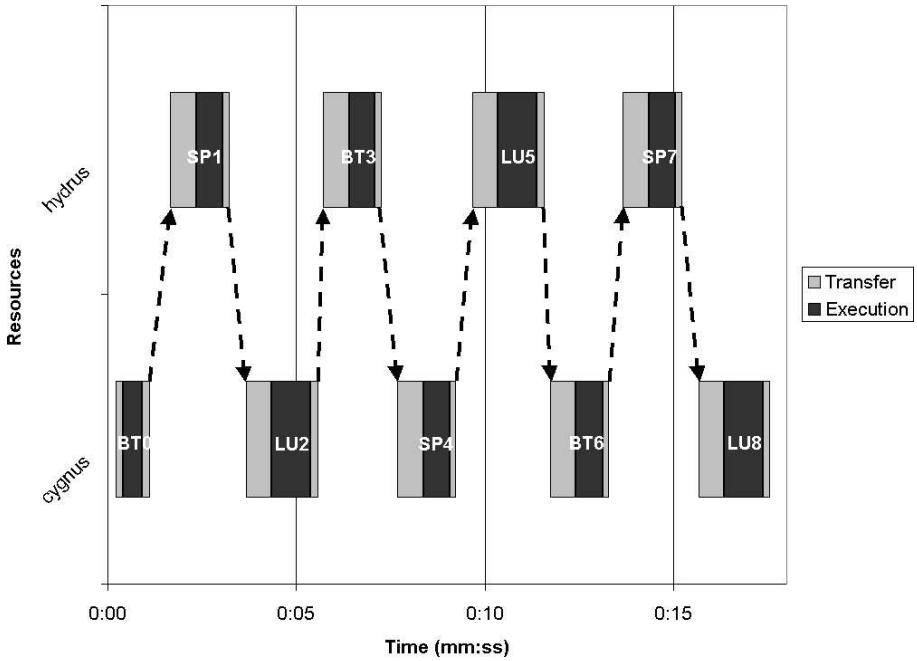
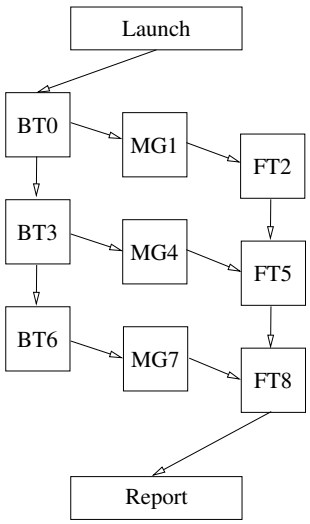


Figure 10: Results with the HC.A benchmark.

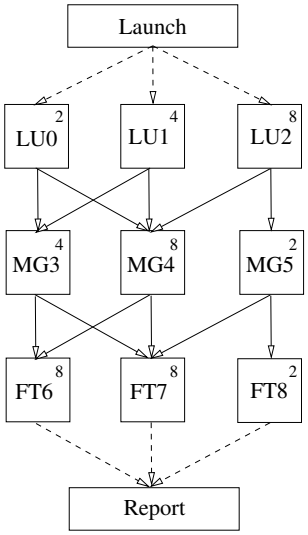
```
drmaa_init(contact, err);

// Loop until all jobs are finished
while (there_are_jobs_left(jobs)) {

  // Submit jobs with dependencies solved
  for (i = 0; i<num_jobs; i++)
    if (is_job_ready(jobs, i))
      drmaa_run_job(jobs[i].id,
          jobs[i].jt, err);

  // Wait any submitted job to finish
  job_id = "DRMAA_JOB_IDS_SESSION_ANY";
  drmaa_wait(job_id, &stat, timeout,
      rusage, err);
  set_job_done(jobs, job_id);
}

drmaa_exit(err_diag);
```

Figure 11: Implementation of the *workflow engine*.




```
// Initialization
jobs[0].jt = BT; jobs[0].dep = "";
jobs[1].jt = MG; jobs[1].dep = "0";
jobs[2].jt = FT; jobs[2].dep = "1";
jobs[3].jt = BT; jobs[3].dep = "0";
jobs[4].jt = MG; jobs[4].dep = "3";
jobs[5].jt = FT; jobs[5].dep = "2 4";
jobs[6].jt = BT; jobs[6].dep = "3";
jobs[7].jt = MG; jobs[7].dep = "6";
jobs[8].jt = FT; jobs[8].dep = "5 7";
```

Figure 12: Structure and *workflow engine* initialization of the VP benchmark.




```
// Initialization
jobs[0].jt = LU; jobs[0].dep = "";
jobs[1].jt = LU; jobs[1].dep = "";
jobs[2].jt = LU; jobs[2].dep = "";
jobs[3].jt = MG; jobs[3].dep = "0 1";
jobs[4].jt = MG; jobs[4].dep = "0 1 2";
jobs[5].jt = MG; jobs[5].dep = "2";
jobs[6].jt = FT; jobs[6].dep = "3 4";
jobs[7].jt = FT; jobs[7].dep = "3 4 5";
jobs[8].jt = FT; jobs[8].dep = "5";
```

Figure 13: Structure and *workflow engine* initialization of the MB benchmark.

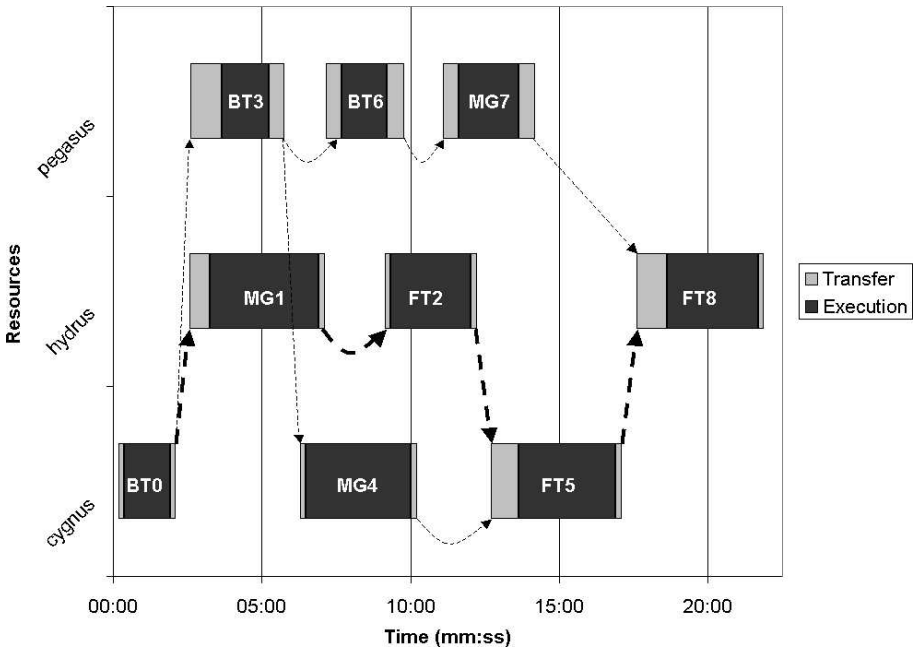


Figure 14: Results with the VP.A benchmark.

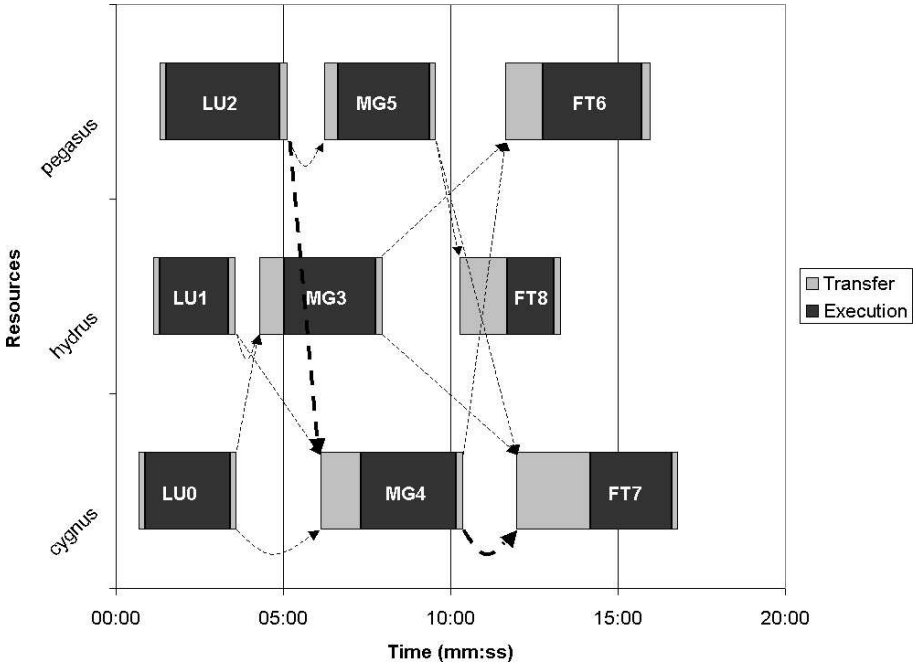


Figure 15: Results with the MB.A benchmark.