SYNTHETIC GRID WORKLOADS WITH IBIS, KOALA, AND GRENCHMARK

Alexandru Iosup and Dick H.J. Epema
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology,
Mekelweg 4, 2628 CD, Delft, The Netherlands
A.Iosup@tudelft.nl
D.H.J.Epema@tudelft.nl

Jason Maassen and Rob van Nieuwpoort
Department of Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands
Jason@cs.vu.nl
Rob@cs.vu.nl

Abstract Grid computing is becoming the natural way to aggregate and share large sets of heterogeneous resources. However, grid development and acceptance hinge on proving that grids reliably support real applications. A step in this direction is to combine several grid components into a demonstration and testing framework. This paper presents such an integration effort, in which three research prototypes, namely a grid application development toolkit (Ibis), a grid scheduler capable of co-allocating resources (Koala), and a synthetic grid workload generator (GrenchMark), are used to generate and run workloads comprising well-established and new grid applications on our DAS multi-cluster testbed.

Keywords: Grid, performance evaluation, synthetic workloads.

1. Introduction

Grid computing's long term promise is a seamlessly shared infrastructure comprising heterogeneous resources, to be used by multiple organizations and independent users alike [12]. With the infrastructure starting to fulfill the requirements of such an ambitious promise [4], it is crucial to prove that grids can run real applications, from traditional sequential and parallel applications to new, grid-specific, applications. As a consequence, there is a clear need for generating workloads comprising of real applications, and for running them in grid environments, for demonstration and testing purposes.

A significant number of projects have tried to tackle this problem from different angles: attempting to produce a representative set of grid applications like the NAS Grid Benchmarks [13], creating synthetic applications that can assess the status of grid services like the GRASP project [7], and creating tools for launching benchmarks and reporting results like the GridBench project [21].

This work addresses the problem of generating and running synthetic grid workloads, by integrating the results of three research projects coming from CoreGRID partners, namely the grid application development toolkit Ibis [22], the grid scheduler KOALA [17], and the synthetic grid workload generator and submitter GRENCHMARK. Ibis is being developed at VU Amsterdam[1] and provides a set of generic Java-based grid applications. KOALA is being developed at TU Delft[2] and allows running generic grid applications. Finally, GRENCHMARK is being developed at TU Delft[3] and is able to generate workloads comprising typical grid applications, and to submit them to arbitrary grid environments.

2. A Case for Synthetic Grid Workloads

There are three ways of evaluating the performance of a grid system: analytical modeling, simulation, and experimental testing. This section presents the benefits and drawbacks of each of the three, and argues for evaluating the performance of grid systems using synthetic workloads, one of the two possible approaches for experimental testing.

2.1 Analytical Modeling and Simulations

Analytical modeling is a traditional method for gaining insights into the performance of computing systems. Analytical modeling may sim-

[1]Ibis is available from `http://www.cs.vu.nl/ibis/`.
[2]KOALA is available from `http://www.st.ewi.tudelft.nl/koala/`.
[3]GRENCHMARK is available from `http://grenchmark.st.ewi.tudelft.nl/`.

plify *what-if* analysis, for changes in the system, in the middleware, or in the applications. However, the sheer size of grids and their heterogeneity make realistic analytical modeling hardly tractable.

Simulations may handle complex situations, sometimes very close to the real system. Furthermore, simulations allow the *replay* of real situations, greatly facilitating the discovery of appropriate solutions. However, simulated system size and diversity raises questions on the representativeness of simulating grids. Moreover, nondeterminism and other forms of hidden dynamic behavior of grids make the simulation approach even less suitable. Even if these problems are overlooked, the simulation outcome is greatly dependent on the used (synthetic) workloads [9, 11].

2.2 Experimental Testing

There are three ways to experimentally assess the performance of grid systems: *using real grid workloads*, *using synthetic grid workloads*, and *benchmarking*.

We argue that traces of real grid workloads (short, *traces*) are difficult to replay in currently existing grids: the infrastructure changes too fast, leading to incompatible resource requests when re-running old traces. This renders the potential *use of real traces* unsuitable for the moment. Synthetic grid workloads derived from one or several traces, may be used instead.

Benchmarking is typically used to understand the quantitative aspects of running grid applications and to make results readily available for comparison. A benchmarks comprises a set applications representative for a class of systems, and a set of rules for running the applications as a synthetic system workload. Therefore, a benchmark is a single instance of a synthetic workload.

Benchmarks present severe limitations, when compared to synthetic grid workloads generation. They have to be developed under the auspices of an important number of (typically competing) entities, and can only include well-studied applications. Putting aside the considerable amounts of time and resources needed for these tasks, the main problem is that grid applications are starting to develop just now, typically at the same time with the infrastructure [19], thus limiting the availability of truly representative applications for inclusion in standard benchmarks. Other limitations in using benchmarks for more than raw performance evaluation are:

- Benchmarking results are valid only for workloads truly represented by the benchmark's set of applications; moreover, the number of applications typically included in benchmarks [13, 21]is typically small, limiting even more the scope of benchmarks;

- Benchmarks include mixes of applications representative at a certain moment of time, and are notoriously resistant to include new applications; thus, benchmarks cannot respond to the changing requirements of developing infrastructures, such as grids;
- Benchmarks measure only one particular system characteristic (low-level benchmarks), or a mix of characteristics (high-level benchmarks), but not both.

An extensible framework for *generating and submitting synthetic grid workloads* uses applications representative for today's grids, and fosters the addition of future grid applications. This approach can help overcome the aforementioned limitations of benchmarks. First, it offers better flexibility in choosing the starting applications set, when compared to benchmarks. Second, applications can be included in generated workloads, even when they are in a debug or test phase. Third, the workload generation can be easily parameterized, to allow for the evaluation of one or a mix of system characteristics.

2.3 Grid Applications Types

From the point of view of a grid scheduler, we identify two types of applications that can run in grids, and may be therefore included in synthetic grid workloads.

- *Unitary applications* This category includes single, unitary, applications. At most the job programming model must be taken into account when running in grids (e.g., launching a name server before launching an Ibis job). Typical examples include sequential and parallel (e.g., MPI, Java RMI, Ibis) applications. The tasks composing a unitary application, for instance in a parallel application, can interact with each other.
- *Composite applications* This category includes applications composed of several unitary or composite applications. The grid scheduler needs to take into account issues like task inter-dependencies, advanced reservation and extended fault-tolerance, besides the components' job programming model. Typical examples include parameter sweeps, chains of tasks, DAG-based applications, and even generic graphs.

2.4 Purposes of Synthetic Grid Workloads

We further present five reasons for using synthetic grid workloads.

- *System design and procurement* Grid architectures offer many alternatives to their designers, in the form of hardware, of operating software, of middleware (e.g., a large variety of schedulers), and of software libraries. When a new system is replacing an old one, running a synthetic workload can show whether the new configuration performs according to the expectations, before the system becomes available to users. The same procedure may be used for assessing the performance of various systems, in the selection phase of the procurement process.

- *Functionality testing and system tuning* Due to the inherent heterogeneity of grids, complicated tasks may fail in various ways, for example due to misconfiguration or unavailability of required grid middleware. Running synthetic workloads, which use the middleware in ways similar to the real application, helps testing the functionality of the grids and detecting many of the existing problems.

- *Performance testing of grid applications* With grid applications being more and more oriented toward services [15]or components [14], early performance testing is not only possible, but also required. The production cycle of traditional parallel and distributed applications must include early testing and profiling. These requirements can be satisfied with a synthetic workload generator and submitter.

- *Comparing grid components* Grid middleware comprises various components, e.g., resource schedulers, information systems, and security managers. Synthetic workloads can be used for solving the requirements of component-specific use cases, or for testing the Grid-component integration.

- *Building runtime databases* In many cases, getting accurate information about an application's runtime is critical for further optimizing its execution. For many scheduling algorithms, like backfilling, this information is useful or even critical. In addition, some applications need (dynamic) *on-site* tuning of their parameters in order to run faster. The use of historical runtime information databases can help alleviate this problem [18]. An automated workload generator and submitter would be of great help in filling the databases.

In this paper we show how GrenchMark can be used to generate synthetic workloads suitable for one of these goals (functionality testing and system tuning), and lay out a research roadmap that may lead to fulfilling the requirements of all five goals (see Section 6).

3. An Extensible Framework for Grid Synthetic Workloads

This section presents an extensible framework for generating and submitting synthetic grid workloads. The first implementation of the framework integrates two research prototypes, namely a grid application development toolkit (Ibis), and a synthetic grid workload generator (GrenchMark).

3.1 Ibis: Grid Applications

Ibis is a grid programming environment offering the user efficient execution and communication [8], and the flexibility to run on dynamically changing sets of heterogeneous processors and networks.

The Ibis distribution package comes with over 30 working applications, in the areas of physical simulations, parallel rendering, computational mathematics, state space search, bioinformatics, prime numbers factorization, data compression, cellular automata, grid methods, optimization, and generic problem solving. The Ibis applications closely resemble real-life parallel applications, as they cover a wide-range of computation/communication ratios, have different communication patterns and memory requirements, and are parameterized. Many of the Ibis applications report detailed performance results. Last but not least, all the Ibis applications have been thoroughly described and tested in various grids [8, 22]. They work on various numbers of machines, and have automatic fault tolerance and migration features, thus responding to the requirements of dynamic environments such as grids. For a complete list of publications, please visit `http://www.cs.vu.nl/ibis`. Therefore, the Ibis applications are representative for grid applications written in Java, and can be easily included in synthetic grid workloads.

3.2 GrenchMark: Synthetic Grid Workloads

GrenchMark is a synthetic grid workload generator and submitter. It is *extensible*, in that it allows new types of grid applications to be included in the workload generation, *parameterizable*, as it allows the user to parameterize the workloads generation and submission, and *portable*, as its reference implementation is written in `Python`.

The workload generator is based on the concepts of *unit generators* and of job description files (JDF) *printers*. The *unit generators* produce detailed descriptions on running a set of applications (*workload unit*), according to the workload description provided by the user. There is one unit for each supported application type. The *printers* take the

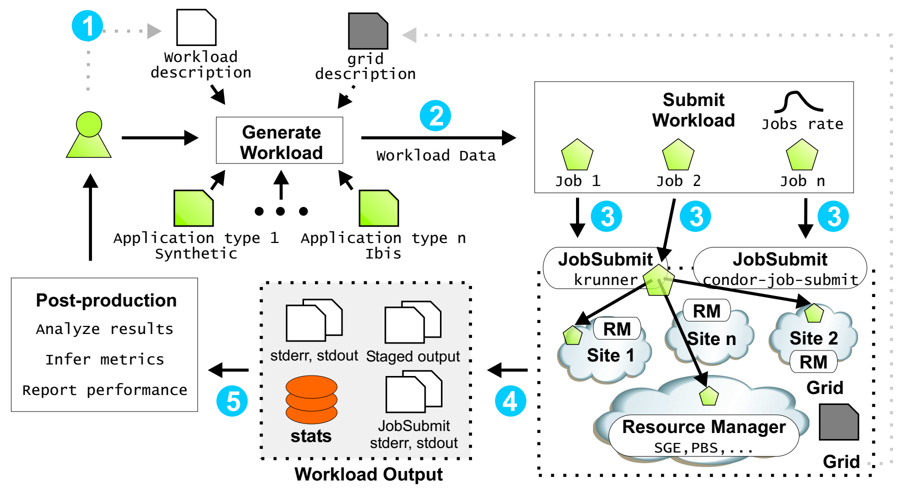

Figure 1. The GrenchMark process.

generated workload units and create job description files suitable for grid submission. In this way, multiple unit generators can be coupled to produce a workload that can be submitted to any grid resource manager, as long as the resource manager supports that type of applications.

The grid applications currently supported by GrenchMark are sequential jobs, jobs which use MPI, and Ibis jobs. We use the Ibis applications included in the default Ibis distribution (see Section 3.1). We have also implemented three *synthetic applications*: `sser`, a sequential application with parameterizable computation and memory requirements, `sserio`, a sequential application with parameterizable computation and I/O requirements, and `smpi1`, an MPI application with parameterizable computation, communication, memory, and I/O requirements. Currently, GrenchMark can submit jobs to Koala and Globus GRAM.

The workload generation is also dependent on the applications inter-arrival time [6]. Peak job arrival rates for a grid system can also be modeled using well-known statistical distributions [6, 16]. Besides the Poisson distribution, used traditionally in queue-based systems simulation, modeling could rely on uniform, normal, exponential and hyper-exponential, Weibull, log normal, and gamma distributions. All these distributions are supported by the GrenchMark generator.

The workload submitter generates detailed reports of the submission process. The reports include all job submission commands, the turnaround time of each job, including the grid overhead, the total turnaround time of the workload, and various statistical information.

3.3 Using the Framework

Figure 1 depicts the typical usage of our framework. First, the user describes the workload to be generated, as a formatted text file (1). Based on the user description, on the known application types, and on information about the grid sites, a workload is then generated by GrenchMark (2). A generated workload is then submitted or resubmitted to the grid (3). The grid environment is responsible for executing the jobs and returning their results (4). The results include the outcome of the jobs, and detailed submission reports. Finally, the user processes all results in a post-production step (5).

4. A Concrete Case: Synthetic Workloads for the DAS

This section presents a concrete case for our framework: generating and running synthetic workloads on the DAS [3], a 400 processors multi-cluster environment. The Ibis applications were combined with the synthetic applications, to create a pool of over 35 grid applications. The GrenchMark tools were used to generate and launch the synthetic workloads.

4.1 Koala: Scheduling Grid Applications

A key part of the experimental infrastructure is the Koala [17] grid scheduler. To the author's knowledge, Koala is the only fault-tolerant, well-tested, and deployed grid scheduler that provides support for *co-allocated* jobs, that is, it can simultaneously allocate resources in multiple grid sites to single applications which consist of multiple components. Koala was used to submit the generated workloads to the DAS multi-cluster. Its excellent reporting capabilities were also used for evaluating the jobs execution results.

For co-allocated jobs, Koala gives the user the option to specify the actual execution sites, i.e., the clusters where job components should run. Koala supports *fixed* jobs, for which users fully specify the execution sites, *non-fixed* jobs, for which the user does not specify the execution sites, leaving instead Koala to select the best sites, and *semi-fixed* jobs, which are a mix of the previous two. Koala may schedule different components of a non-fixed or of a semi-fixed job onto the same site. We used this feature heavily for the Ibis and the synthetic MPI applications. The structure of all used applications requires interaction between their co-allocated components.

Table 1. The experimental workloads. As the DAS has only 5 sites; jobs with more than 5 components will have several components running at the same site.

		# of	*# of*	*Component*		*Success*
Workload	*Applications types*	*Jobs*	*CPUs*	*No.*	*Size*	*Rate*
`gmark1`	synthetic, sequential	100	1	1	1	97%
`gmark+`	synthetic, seq. & MPI	100	1-128	1-15	1-32	81%
`ibis1`	N Queens, Ibis	100	2-16	1-8	2-16	56%
`ibis+`	various, Ibis	100	2-32	1-8	2-16	53%
`wl+all`	all types	100	1-32	1-8	1-32	90%

```
# File-type: text/wl-spec
#ID Jobs Type   SiteType Total SiteInfo ArrivalTimeDistr   OtherInfo
?   25   sser   single    1       *:?   Poisson(120s)      StartAt=0s
?   25   sserio single    1       *:?   Poisson(120s)      StartAt=60s
?   25   smpi1  single    1       *:?   Poisson(120s)      StartAt=30s,ExternalFile=smpi1.xin
?   25   smpi1  single    1       *:?   Poisson(120s)      StartAt=90s,ExternalFile=smpi2.xin
```

Figure 2. A GrenchMark workload description example.

4.2 The Workload Generation

Table 1 shows the structure of the five generated workloads, each comprising 100 jobs. To satisfy typical grid situations, jobs request resources from 1 to 15 sites. For parallel jobs, there is a preference for 2 and 4 sites. Site requests are either precise (specifying the full name of a grid site) or non-specified (leaving the scheduler to decide). For multi-site jobs, components occupy between 2 and 32 processors, with a preference for 2, 4, and 16 processors. We used combinations of parameters that would keep the run-time of the applications under 30 minutes, under optimal conditions. Each job requests resources for a time below 15 minutes. Various inter-arrival time distributions are used, but the submission time of the last job of any workload is kept under two hours.

Figure 2 shows the workload description for generating the `gmark+` test, comprising 100 jobs of four different types. The first two lines are comments. The next two lines are used to generate sequential jobs of types `sser` and `sserio`, with default parameters. The final two lines are used to generate MPI jobs of type `smpi1`, with parameters specified in external files `smpi1.xin` and `smpi2.xin`. All four job types assume an arrival process with Poisson distribution, with a average rate of 1 job every 120 seconds. The first job of each type starts at a time specified in the workload description with the help of the `StartAt` tag.

4.3 The Workload Submission

GrenchMark was used to submit the workloads. Each workload was submitted in the normal DAS working environment, thus being in-

Table 2. A summary of time and run/success percentages for different job types.

		Turnaround [s]			*Runtime [s]*				*Run+*
Job name	*Job type*	*Avg.*	*Min*	*Max*	*Avg.*	*Min*	*Max*	*Run*	*Success*
`sser`	sequential	129	16	926	44	1	588	100%	97%
`smpi1`	MPI	332	21	1078	110	1	332	80%	85%
`N Queens`	Ibis	99	15	1835	31	1	201	66%	85%

fluenced by the background load generated by other DAS users. Some jobs could not finish in the time for which they requested resources, and were stopped automatically by the KOALA scheduler. This situation corresponds to users under-estimating applications' runtimes. Each workload ran between the submission start time and 20 minutes after the submission of the last job. Thus, some jobs did not run, as not enough free resources were available during the time between their submission and the end of the workload run. This situation is typical for real working environments, and being able to run and stop the workload according to the user specifications shows some of the capabilities of GRENCHMARK.

5. The Experimental Results

This section presents an overview of the experimental results, and shows that workloads generated with GRENCHMARK can cover in practice a wide-range of run characteristics.

5.1 The Performance Results

Table 1 shows the success rate for all five workloads (column *Success Rate*). A successful job is a job that gets its resources, runs, finishes, and returns all results within the time allowed for the workload. We have selected the success rate metric to show that GRENCHMARK can be used to evaluate the arguably biggest problem of nowadays grids, i.e., the high rate of failures. The lower performance of Ibis jobs (workload `ibis+`) when compared to all the others, is caused by the fact that the system was very busy at the time of testing, making the resource allocation particularly difficult. This situation cannot be prevented in large-scale environments, and cannot be addressed without special resource reservation rights.

The turnaround time of an application can vary greatly (see Table 2), due to different parameter settings, or to varying system load. The variations in the application runtimes are due to different parameter settings.

As expected, the percentage of the applications that are actually run (Table 2, column *Run*) depends heavily on the job size and system load. The success rate of jobs that *did* run shows little variation (Table 2, column *Run+Success*). The ability of GrenchMark to report percentages such as these enables future work on comparing of the success rate of co-allocated jobs, vs. single-site jobs.

5.2 Dealing With Errors

Using the combined GrenchMark and Koala reports, it was easy to identify errors at various levels in the submission and execution environment: the user, the scheduler, the local and the remote resource, and the application environment levels. For a better description of the error levels, and for a discussion about the difficulty of trapping and understanding errors, we refer the reader to the work of Thain and Livny [20].

We were able to identify bottlenecks in the grid infrastructure, and in particular in Koala, which was one of our goals. For example, we found that for large jobs in a busy system, the percentage of unsuccessful jobs increases dramatically. The reason is twofold. First, using a single machine to submit jobs (a typical grid usage scenario) incurs a high level of memory occupancy, as for each job a Koala job submitter is started, and one set of Java core and GRAM libraries are loaded into memory. A possible solution is to allow a single Koala job submitter to handle multiple jobs simultaneously. Second, there are cases when jobs attempt to claim the resources allocated by the scheduler, but fail to do so, for instance because a local request leads to resources being claimed by another user (scheduling-claiming atomicity problem). A potential solution is to use an exponential back-off mechanism when (re-)scheduling such jobs.

6. Proposed Research Roadmap

In this section we present a research roadmap for creating a framework for synthetic grid workload generation, submission, and analysis. We argue that such a complex endeavor cannot be completed in one step, and, most importantly, not by a single research group. We propose instead an iterative roadmap, in which results obtained in each of the steps are significant for theoretical and practical reasons.

Step 1. Identify key modeling features for synthetic grid workloads;

Step 2. Build or extend a framework for synthetic grid workloads generation, submission, and analysis;

Step 3. Analyze grid traces and create models of them;

Step 4. *Repeat from Step 1 until the framework includes enough features*;

Step 5. Devise grid benchmarks for specific goals (see Section 2.4);

Step 6. *Repeat from Step 1 until all the important domains in Grid are covered*;

Step 7. Create a comprehensive Grid benchmark, in the flavor of SPEC [1] and TPC [2].

The work included in this paper represents an initial Step 1-3 iteration. We have first identified a number of key modeling features for synthetic grid workloads, e.g., application types. We have then built an extensible framework for synthetic grid workloads generation, submission, and analysis. Finally, we have used the framework to test the functionality of and tune the KOALA scheduler.

7. Conclusions and Ongoing Work

This work has addressed the problem of synthetic grid workload generation and submission. We have integrated three research prototypes, namely a grid application development toolkit, Ibis, a grid metascheduler, KOALA, and a synthetic grid workload generator, GRENCHMARK, and used them to generate and run workloads comprising well-established and new grid applications on a multi-cluster grid. We have submitted a large number of application instances, and presented overview results of their execution.

We are currently adding to GRENCHMARK composite applications generation capabilities and an automatic results analyzer. For the future, we plan to prove the applicability of GRENCHMARK for specific grid performance evaluation, such as an evaluation of the DAS support for High-Energy Physics applications [10], and a performance comparison of co-allocated and single site applications, to complement our previous simulation work [5].

Acknowledgments

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265). Part of this work was also carried out in the context of the Virtual Laboratory for e-Science project (`www.vl-e.nl`), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). We would also like to thank our reviewers for their helpful comments, Hashim Mohamed and Wouter Lammers for their work on KOALA, and Gosia Wrzesiñska, Niels Drost, and Mathijs den Burger for their work on Ibis.

References

[1] The Standard Performance Evaluation Corporation. SPEC High-Performance Computing benchmarks. [Accessed] March 2006. [Online] `http://www.spec.org/`.

[2] Transaction Processing Performance Council. TPC transaction processing and database benchmarks. [Accessed] March 2006. [Online] `http://www.tpc.org/`.

[3] Henri E. Bal et al. The distributed ASCI supercomputer project. *Operating Systems Review*, 34(4):76–96, October 2000.

[4] F. Berman, A. Hey, and G. Fox. *Grid Computing: Making The Global Infrastructure a Reality.* Wiley Publishing House, 2003. ISBN: 0-470-85319-0.

[5] Anca I. D. Bucur and Dick H. J. Epema. Trace-based simulations of processor co-allocation policies in multiclusters. In *HPDC*, pages 70–79. IEEE Computer Society, 2003.

[6] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP*, volume 1659 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 1999.

[7] G. Chun, H. Dail, H. Casanova, and A. Snavely. Benchmark probes for grid assessment. In *IPDPS*. IEEE Computer Society, 2004.

[8] Alexandre Denis, Olivier Aumage, Rutger F. H. Hofman, Kees Verstoep, Thilo Kielmann, and Henri E. Bal. Wide-area communication for grids: An integrated solution to connectivity, performance and security problems. In *HPDC*, pages 97–106. IEEE Computer Society, 2004.

[9] Carsten Ernemann, Baiyi Song, and Ramin Yahyapour. Scaling of workload traces. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2003.

[10] D. Barberis et al. Common use cases for a high-energy physics common application layer for analysis. Report LHC-SC2-20-2002, LHC Grid Computing Project, October 2003.

[11] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP*, volume 1459 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 1998.

[12] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 15(3), 2002.

[13] Michael Frumkin and Rob F. Van der Wijngaart. Nas grid benchmarks: A tool for grid space exploration. *Cluster Computing*, 5(3):247–255, 2002.

[14] Vladimir Getov and Thilo Kielmann, editors. *Component Models and Systems for Grid Applications*, volume 1 of *CoreGRID series*. Springer Verlag, June 2004. Proceedings of the Workshop on Component Models and Systems for Grid Applications held June 26, 2004 in Saint Malo, France.

[15] M. Humphrey et al. State and events for web services: A comparison of five WS-Resource Framework and WS-Notification implementations. In *Proc. of the 14th IEEE HPDC*, Research Triangle Park, NC, USA, July 2005.

[16] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel & Distributed Computing*, 63(11):1105–1122, Nov 2003.

[17] H.H. Mohamed and D.H.J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *Proc. of the 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2005)*, Cardiff, UK, May 2005.

[18] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting application run times with historical information. *J. Parallel Distrib. Comput.*, 64(9):1007–1016, 2004.

[19] Allan Snavely, Greg Chun, Henri Casanova, Rob F. Van der Wijngaart, and Michael A. Frumkin. Benchmarks for grid computing: a review of ongoing efforts and future directions. *SIGMETRICS Perform. Eval. Rev.*, 30(4):27–32, 2003.

[20] Douglas Thain and Miron Livny. Error scope on a computational grid: Theory and practice. In *HPDC*, pages 199–208. IEEE Computer Society, 2002.

[21] G. Tsouloupas and M. D. Dikaiakos. GridBench: A workbench for grid benchmarking. In P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *EGC*, volume 3470 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2005.

[22] Rob V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency & Computation: Practice & Experience.*, 17(7-8):1079–1107, June-July 2005.