A framework for performance evaluation of parallel applications on the Grid

Carlos Figueira Emilio Hernández

Eduardo Blanco *

Universidad Simón Bolívar,

Departamento de Computación y Tecnología de la Información,

Caracas 1080-A, Venezuela

{figueira,emilio,eduardo}@ldc.usb.ve

Abstract

Performance evaluation of applications running on a Grid is a challenging task. Grid's resources are heterogeneous in nature, often shared, and dynamic, all of which have important implications on the performance of an application executing on the Grid. For instance, applications performance will suffer from perturbation induced by external load on the network or computational nodes. Also, resources allocated to applications may vary between different executions. In this paper, we propose a simple framework that takes into account these factors to allow users to gain knowledge of fundamental performance characteristics of their parallel applications. This framework was incorporated in SUMA, a Grid-enabled platform for the execution of scientific applications in Java. We show some results of the utilization of this framework, which was tested by analyzing and tuning a parallel application.

1 Introduction

The *Grid* or *Computational Grid* [FK99] has the potential to become the platform of choice for high performance applications. Many of these applications will eventually be either ported to the Grid or developed directly on the Grid. Hence, tools as those used on single system platforms will be adapted to the Grid, and certainly new tools will be developed as well [Glo03, VGr].

The Grid gathers distributed resources under a common, simple view. Resources are allocated to applications according to their needs and availability. Resources may include computational nodes, repositories, instruments, etc. They are heterogeneous in nature, often shared, and dynamic, all of which have important implications on the performance of an application executing on the Grid. For instance, they will suffer from perturbation induced by external load on the network or computational nodes. Also, resources allocated to an application may vary between different executions, or even during the same execution.

Performance evaluation on Grids is recognized as a complex problem [GBN04]. Some tools, like Askalon [FJP⁺05], provide a coherent set of tools to help users in the tasks of performance analysis of parallel and distributed applications on the Grid. It follows the architecture proposed by the Global Grid Forum [TAG⁺] in order to ensure flexibility and interoperability. The tool proposed in [BBF⁺04] gives the user performance information of interactive applications on the Grid.

*This work was partially supported by grants from Fonacit (*Proyecto Agenda Petróleo* 97003588) and from Universidad Simón Bolívar (*Programa de Apoyo a Grupos de Investigación, Decanato de Investigación y Desarrollo*).

However, these tools fail to address important questions arisen in Grid environments, namely the dynamic overhead of the middleware and its impact on the performance, and the performance data scalability. A framework was introduced in [FH01] to cope with the problem of performance evaluation of applications running on Grids. It proposed the use of *condensed profiles*, which express basic metrics of an application in order to understand its performance and to achieve platform independent optimizations. The condensed profiles included a report of the overhead induced by the Grid middleware. It suggested to normalize computational nodes' power by using benchmarks to characterize platform performance.

In this paper, we extend that work to cope for parallel applications, limiting our scope to Grids whose parallel nodes are accessed through the middleware and that do not vary during an execution. We also explore the use of benchmarks and metrics r_∞ , $n_{1/2}$ [HJ81], as alternative means to normalize parallel computational nodes' power.

The rest of the document is structured as follows. In section 2 the extended framework of performance metrics for parallel applications on Grids is briefly described. Section 3 describes, through examples, the performance information handled by the framework, as well as the use of this information to tune a parallel application on the grid environment SUMA [HCFT00]. Section 4 introduces two different methods to obtain normalized power and shows how to infer whether platform independent improvements have been achieved. Finally, some conclusions are addressed in section 5.

2 Extended framework

Figure 1: Execution of an application on the Grid

We consider a simplified scenario for the execution of an application on the Grid, as shown in figure 1:

1. A user submits an application to the Grid, specifying a number of execution requirements like, for instance, a parallel platform with MPI.
2. After authentication and access control verification, the Grid's scheduler and resource locator component assigns execution to a given platform.
3. Execution starts on the chosen platform, probably requiring input data files transfers from the user's machine or from a repository (e.g., a remote file server).
4. After execution, a response is produced and returned to the user, which may include application's output files and data, as well as performance information (if requested).

The execution activity will generally consist of more than a single execution. For every execution, the scheduler might choose a different platform, raising the problem of relating performance data from different platforms. Also, environment conditions (e.g., network traffic) may vary between executions. The framework components described below take into account these particular factors.

2.1 Middleware overhead

The middleware, which provides the single system illusion, introduces perturbation of different types:

- *Latency*, regarded as the time elapsed since the application is submitted to the Grid until execution starts at the selected platform. This time encompasses authentication, resource finding, and actual transfer of executables and initial input data.
- *Overhead* caused by remote accesses during execution at the platform, when necessary, including objects, classes and files from the user machine or remote repository.
- *Delay* caused by results transfer to the user.

Information about time and resource usage incurred by perturbations of these three kinds is delivered to the client. It can be used, for instance, to gain accuracy in estimating the actual execution time at the platform. Additionally, Grid administrators can process this information to assess middleware performance.

2.2 Condensed profile

The condensed profile involves performance metrics related to the overall execution and performance metrics of selected nodes of the parallel architecture. In order to achieve scalability, we chose a fixed size performance data format, based on statistical summaries. This fixed size format does not depend on the number of nodes or any other platform characteristic, hence the performance data transferred through the network remain small regardless of the size of the platform. The overall performance metrics include: total execution time measured at the front-end, and mean and variance of CPU time and communication/synchronization times. These metrics were selected to help in detecting problems related to load balancing, bad granularity or unsuitable data distribution. This information is supported by metrics collected from those nodes that exhibits extreme behavior like, for instance, nodes with highest and lowest execution or communication/synchronization time.

3 Analysis of a parallel execution

We conducted some experiments in order to assess the power of the condensed profiles. These experiments were performed using SUMA, a java-enabled Grid system.¹

SUMA aims at executing high performance Java byte-code applications [Jav, MLVB05]. It offers interactive and batch execution on sequential and parallel nodes, transparently allocated by the middleware, as well as added services like profiling [FH01] and checkpointing [CH01]. Applications are sent to Execution Nodes, where execution starts, redirecting I/O to client machines, and loading classes and data from client machines. SUMA middleware was originally built on top of commodity software and communication technologies, including Java and CORBA. A recent reimplementaion called SUMA/G [CH05] is partially based on Globus services [The06]. In particular, we say that SUMA/G is a Grid-enabled execution platform

¹<http://www.suma.ldc.usb.ve>

because it includes mechanisms for utilizing the Globus Security Infrastructure. By reimplementing some of the SUMA components on top of Globus services we can connect to deployed well-known grids, while keeping SUMA execution model unchanged. Additional functionalities are inherited from Globus, such as the I/O services.

Figure 2: Location of user, SUMA core components and execution resource machines

For our experiments, SUMA was deployed on interconnected LAN's located on two buildings of the Universidad Simón Bolívar campus, as shown in figure 2. In such a local deployment, the middleware overhead is low. We focus here on two fundamental questions:

- Does the condensed profile offer a useful picture of the performance of a parallel application?
- Is the condensed profile useful for achieving high level optimizations of a parallel application?

3.1 Experiments environment

Execution Node is taken from a cluster (called CAR), composed by 24 dual 600-800 MHz Pentium III PC's with 512 MBytes RAM, running Linux, interconnected by a 1.9 GHz full-duplex (per link) Myrinet network. The message passing library is LAM-MPI; the JVM is JDK 1.2.2; mpiJava [BCFK99] version is 1.1. The back-end profiler is based on MPE [GLon] and *HProf* [Sun05]. Note the instrumentation overhead is kept low by using statistical sampling (*HProf*) and instrumenting only the actual communication methods (MPE). Condensed profiles are locally stored in each node, then collected and summarized at the front-end processor.

Core components of the Grid (e.g., Scheduler, CORBA Name Server, etc.) are installed on a dual Pentium III PC with 256 MBytes RAM. The user interface (client) runs on a Pentium IV PC 1.5 GHz

with 256 MBytes RAM. All Execution Node, Grid core components and client run on two networks scattered over the campus, as shown on figure 2.

The application is a model of wave propagation in 2D homogeneous surface, whose core computation is a triple loop (2D + time) of medium size (400x400 surface points, computed over 600 time steps). It is a SPMD mpiJava application; the coordinator process initializes and distributes data to the rest of the processes, then computes and receives final results from them. All processes are arranged on a logical linear array; each one of them computes a band of the result matrix, and exchange frontier rows with its neighbors on each iteration of the outer loop. For this experiment, We suppressed saving the results to file.

3.2 Analysis of condensed profile

Total time	88 sec.
Profiling overhead	22 sec.
Middleware overhead	1 sec.

Table 1: Global performance information on 8 nodes, as seen by the user

Time at front-end	64 sec.	
Execution time	Mean: 62 sec.	Variance: 0
Com/Sync time	Mean: 3 sec.	Variance: 48

Table 2: General performance information at the Execution Node

The parallel application is run on 8 nodes of the cluster. Information taken from the condensed profile are shown in tables 1, 2, 3, 4 and 5, where only relevant information is displayed. The inclusive time used in this document is computed as the percentage, related to the node's total execution time, of the time spent in a particular method and, recursively, the methods it invoked.

Table 1 shows performance information as seen by the user. As expected, the middleware overhead (1 sec. vs 88 sec. of total time) is negligible. Note however that there is considerable overhead (22 sec.) due to profiling (e.g. collecting and processing performance data).

Table 2 shows information recorded at the front-end of the Execution Node. Note the mean communication/synchronization time (3 sec.) is low with respect to mean execution time (62 sec.); it suggests that we might benefit from increasing the number of nodes, reducing the mean execution time at the expense of the communication time. Additionally, the com/sync. time variance (48) suggests that there could be some unbalance among the nodes.

Tables 3, 4 and 5 illustrate an homogeneous behavior across all nodes.

3.3 Execution on several platforms

Additional experiments show how the detailed performance information of most relevant nodes provide useful data about application characteristics when dealing with multiple platforms. Tables 6 and 7 include excerpts from condensed profiles of, respectively, fastest, and maximum communication/synchronization

Host name	<i>car07</i>
Total Time	62 sec.
CPU Time	59 sec.
Com/Sync Time	3 sec.

Method	% Time incl.
AcousticMPI1.dim2acust	24
AcousticMPI1.itera	24
AcousticMPI1.main2	24
AcousticMPI1.main	23

Table 3: Node with lowest CPU time

Host name	<i>car0</i>
Total Time	63 sec.
CPU Time	53 sec.
Com/Sync Time	10 sec.

Method	% Time incl.
AcousticMPI1.main2	25
AcousticMPI1.dim2acust	23
AcousticMPI1.itera	23
AcousticMPI1.main	21

Table 4: Node with highest communication/synchronization time

Host name	<i>car01</i>
Total Time	63 sec.
CPU Time	62 sec.
Com/Sync Time	1 sec.

Method	% Time incl.
AcousticMPI1.main2	25
AcousticMPI1.main	24
AcousticMPI1.dim2acust	23
AcousticMPI1.itera	23

Table 5: Node with lowest communication/synchronization time

time from executions on clusters with different number of nodes. Tables 8 and 9 complement former table with general performance information about those nodes.

Nodes	Fastest	
	%	Method
4	24	AcousticMPI1.main2
	24	AcousticMPI1.main
	24	AcousticMPI1.dim2acust
	24	AcousticMPI1.itera
8	24	AcousticMPI1.main2
	23	AcousticMPI1.main
	22	AcousticMPI1.dim2acust
	22	AcousticMPI1.itera
16	24	AcousticMPI1.main2
	23	AcousticMPI1.main
	22	AcousticMPI1.dim2acust
	22	AcousticMPI1.itera
	1	mpi/MPI.InitNative
24	24	AcousticMPI1.main2
	23	AcousticMPI1.main
	18	AcousticMPI1.dim2acust
	17	AcousticMPI1.itera
	6	mpi/MPI.InitNative

Table 6: Summary of detailed information of fastest nodes on several platforms (% of inclusive time)

Tables 6 and 7 suggest that the application behaves similarly in all platforms. Note that communication/synchronization time increases as the number of nodes rise from 8 to 16 and 24 nodes; it corresponds with a higher proportion of execution time attributed to `mpi` methods in tables 6 and 7.

3.4 High level tuning

Tables 3, 4 and 5 hint at a set of four methods: `main`, `main2`, `dim2acust`, and `itera`, all belonging to class `AcousticMPI1`, which altogether account for more than 20% of the node execution time and represent by far the most important methods of the application.² It turns out that those four methods are part of a calling sequence `main` $\rightarrow$ `main2` $\rightarrow$ `dim2acust` $\rightarrow$ `itera`. We decided to concentrate our optimization effort on `itera`, which contains the core computation loop. Note that while there is communication (i.e., row exchange) inside the outer loop, it is not significant from the point of view of performance, as indicate the communication/synchronization metrics on table 2 (3 sec. vs 64 sec. of total time measured at the front-end).

²Run-time and communication methods account for an important share of the execution time; they are not taken into account for our optimization purposes.

Nodes	Max. Com/Syn	
	%	Method
4	24	AcousticMPI1.main2
	24	AcousticMPI1.dim2acust
	24	AcousticMPI1.itera
	21	AcousticMPI1.main
	3	AcousticMPI1.exchangeRows
8	23	AcousticMPI1.main2
	22	AcousticMPI1.dim2acust
	22	AcousticMPI1.itera
	20	AcousticMPI1.main
	3	AcousticMPI1.exchangeRows
16	23	AcousticMPI1.main2
	21	AcousticMPI1.dim2acust
	21	AcousticMPI1.itera
	16	AcousticMPI1.main
	7	mpi/Comm.Recv
24	24	AcousticMPI1.main2
	17	AcousticMPI1.main
	16	AcousticMPI1.dim2acust
	16	AcousticMPI1.itera
	6	mpi/Comm.Recv

Table 7: Summary of detailed information of nodes with maximum communication/synchronization time on several platforms (% of inclusive time)

A tuned version, called `AcousticMPI2`, was generated by modifying the method `itera` to take some independent computations out of the loop. The new general performance is shown on tables 10 and 11. An improvement of 22 % (68 vs 88 seconds) overall and 31 % (44 vs 64 seconds) at the node was achieved.

Playing the role of Grid infrastructure developers and administrators, we decided to investigate why profiling overhead was so high (22 sec.) compared to total time in both original (66 sec.) and tuned (88 sec.) application versions on the 8 nodes cluster. We discovered that one internal performance trace processing was unnecessarily complex. After optimizing the trace processing algorithm, the profiling overhead was reduced from 22 seconds to less than 4 seconds.

4 Normalized power

Performance characterization of a platform aims to express key parameters from which, ideally, it is possible to infer the performance of applications that will run on that platform. There exist different ways of characterizing performance of a platform [Her96]. We consider here two widespread and simple yet powerful methods: benchmarking, and metrics r_∞ and $n_{1/2}$.

Nodes	fastest	
	T. CPU	T. Com/Syn
4	118	2
8	63	2
16	42	2
24	39	3

Table 8: Summary of general performance information of fastest nodes (time in seconds)

Nodes	Max. Com/Syn	
	T. CPU	T. Com/Syn
4	105	16
8	58	8
16	32	14
24	33	13

Table 9: Summary of general performance information of nodes with maximum communication/synchronization time (in seconds)

Total time	68 sec.
Profiling overhead	22 sec.
Middleware overhead	1 sec.

Table 10: Global performance information of tuned version on 8 nodes

Time at front-end	44 sec.		
Execution time	Mean: 42 sec.	Variance: 0	
Com/Sync time	Mean: 2 sec.	Variance: 14	

Table 11: General performance information of tuned version at the Execution Node

4.1 Benchmarking

Benchmarking consists on running one or more programs on the platform, then summarizing results (e.g., execution time, MFLOPS, etc.) in a single number by using, for instance, arithmetic or geometric means [PH96]. The popularity of this method is due in part to the fact that it naturally takes into account the interaction between the application and the system components (hardware, operating system, libraries, compiler, interpreter, etc.). Normalization is achieved by relating the benchmark execution performance results to a reference platform.

Benchmark accuracy relies on the affinity between the programs used in the benchmark and the applications that will actually run on the platform. Several benchmarks for parallel platforms, in C and FORTRAN, are well established and are commonly used by the high performance computing community [BBB⁺94, DH95]. Only recently, some benchmarks have been ported or developed in sequential and parallel Java versions [FSJY02, EPC06].

4.2 Performance characterization using r_∞ and $n_{1/2}$

A model based on metrics r_∞ and $n_{1/2}$ has been used to characterize performance of parallel architectures [HJ81]. While conceived for vector architectures, the model has been extended to other types of architectures, including distributed memory parallel machines [Hoc95]. It describes execution time using a first-order approximation shown in equation (1), where n is number of operations (typically, floating point operations); r_∞ is the peak performance, often expressed in MFLOPS; $n_{1/2}$ is the number of operations to reach half the peak performance.

$$t = \frac{n + n_{1/2}}{r_\infty} \quad (1)$$

4.3 Quality of normalized power

We conducted experiments to assess two alternative methods to obtain the normalized power:

1. Benchmarking. The normalized power is the ratio of execution time of the benchmark on a given platform to the execution time on a reference platform.
2. Ratio of r_∞ . The normalized power is the ratio of platform's r_∞ to the reference platform's r_∞ . It was possible to simplify and discard $n_{1/2}$ because experiments showed that accuracy did not vary significantly [Fig02].

The experiments consist on running an application on the reference platform, then on a test platform. The execution time measured on the test platform is compared with a *predicted time*, obtained by applying equation (2)

$$\text{Predicted time} = \frac{\text{Time on reference platform}}{\text{Test platform's normalized power}} \quad (2)$$

In order to be able to compare measured times, middleware overhead must be subtracted from the time measured at the node.

The application to be used as benchmark is a reduced version of `AcousticMPI1`. The reference platform is a two node cluster with 10 Mbps Ethernet, each node consisting of a dual processor 600 MHz Pentium III, with 256 MBytes SDRAM PC133 (512 cache *off-chip*), and Linux RedHat 6.2.

Nodes	Normal. Power	Time	Predicted	Error
2	1.5	236	231	1.8
4	2.9	123	117	4.7
8	5.5	68	61	9.5
12	6.1	58	55	4.0
16	8.3	48	41	14.5
24	10.6	47	32	31.4

Table 12: Predicted time (seconds) and error (%) using benchmarks for normalized power

Nodes	Normal. Power	Time	Predicted	Error
2	1.4	236	236	0.1
4	2.7	123	120	2.1
8	5.3	68	62	8.7
12	5.5	58	55	2.3
16	7.7	48	43	10.7
24	10.2	47	32	31.4

Table 13: Predicted time (seconds) and error (%) using r_∞ for normalized power

Table 12 shows prediction results for different cluster sizes using normalized power based on benchmarks. Table 13 shows the prediction results using r_∞ for different cluster sizes.

As indicated in tables 12 and 13, both methods have high (except for 24 nodes) and similar prediction quality. This is related to a good match between the benchmark used for computing the normalized power, and the actual application used in our experiments. In the general case, as Grids will accommodate a possibly high diversity of applications, the quality will be lower than in our experiments.

High error for 24 nodes (31.4 %) exposes a deficiency, related to sensibility to the characteristics of the application used to estimate platforms' performance, both for benchmark and r_∞ , $n_{1/2}$ based methods. In this particular case, the application behaved well for smaller clusters but, when used to characterize a 24 nodes cluster, it turned out to be too small to exploit (and hence capture) the platform power.

4.4 Predicting platform independent speedup

Normalized power can be used to compare performance on heterogeneous platforms. Given the improvement shown in section 3.4 by version `AcousticMPI2` measured on an 8 nodes cluster, we can estimate whether or not there will be improvement in other platforms using normalized power.

Table 14 presents the estimated execution time and speedup of tuned code `AcousticMPI2` (version 2) with respect to original code `AcousticMPI1` (version 1) on two different platforms:

- Two nodes from cluster CAR, described in section 3.1.
- The reference platform, a two nodes cluster described in section 4.3.

The estimated values (in *italics*) for tuned version (**Ver. 2**) on each platform (2 nodes from cluster CAR and 2 nodes from reference platform) are obtained by using the execution time of that version on the 8

nodes platform, scaled by the ratio of the normalized powers. From table 13, these ratios are 0.26 (1.4/5.3) and 0.18 (1/5.3) for 2 node platform, and 2 node reference platform, respectively. The values for the original version (**Ver. 1**) were actually measured. The speedup shown compares the estimated execution time of the tuned version with respect to the measured execution time of the original version, for each platform.

Measured execution times and speedup for 8 nodes platform are also included in table 14 for comparison purposes.

Platform	Time (sec.)		Speedup (%)
	Ver. 1	Ver. 2	
8 nodes platform	64	44	31
2 nodes platform	236	<i>167</i>	29
2 nodes reference platform	329	<i>233</i>	29

Table 14: Estimated execution time and speedup of tuned application on two platforms, from measured time on 8 nodes platform (*estimated values in italics*)

Results shown on table 14 indicate that we could expect the new version to run faster than the original version on relatively different platforms.

5 Conclusions

The performance evaluation of applications on the Grid must take into account distinctive features such as heterogeneity, dynamics and middleware overhead.

The framework proposed in this work captures simple yet essential metrics to understand performance from a high point of view. It separates the influence of the middleware from the application performance, and reports high level and scalable metrics of what actually happens in the execution platform. The middleware overhead is reported thanks to monitoring mechanisms built in the middleware components. Scalability is achieved through simple statistical reduction techniques locally processed, such as to avoid the communication of large sets of performance data.

Our experiments demonstrated that the framework can be effective in identifying coarse performance characteristics of an application, leading to high level, platform independent optimizations. Also, overhead metrics built in the system allowed us to detect a Grid infrastructure implementation problem which incurred in considerable overhead.

Execution on heterogeneous platforms needs a basis to compare performance data (see [VR99]). We showed how two different method, benchmarking (as in Netsolve [CD97]) and r_∞ , can be used to determine a normalized power, which can in turn effectively serve for that purpose. Incorporating the recently proposed benchmarks in parallel Java in our framework will help in improving the normalized power accuracy.

Our current approach exhibits some limitations that motivate future work. For instance, using dynamic [HMC94] instead of postmortem instrumentation, would enable the performance evaluation of long running applications. Hence, performance data of partial execution would be periodically transmitted during the application execution.

Further experiments, using different classes of applications running on widely deployed Grids, will be carried out to better assess the framework. We also plan to add facilities to process condensed profiles in order to automatically report potential performance problems to users.

References

- [BBB⁺94] D. H. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.
- [BBF⁺04] Bartosz Balis, Marian Bubak, Włodzimierz Funika, Roland Wismüller, Marcin Radecki, Tomasz Szepieniec, Tomasz Arodz, and Marcin Kurdziel. Performance Evaluation and Monitoring of Interactive Grid Applications. *Lecture Notes in Computer Science*, 3241:345–352, 2004. Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users Group Meeting.
- [BCFK99] M. Baker, B. Carpenter, G. Fox, and Sung Hoon Koo. mpiJava: An Object-Oriented Java Interface to MPI. *Lecture Notes in Computer Science*, 1586, 1999.
- [CD97] Henri Casanova and Jack Dongarra. Netsolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.
- [CH01] Yudith Cardinale and Emilio Hernández. Checkpointing Facility in a Metasystem. *Lecture Notes in Computer Science*, 2150:75–79, August 2001. 7th International Euro-Par Conference.
- [CH05] Y. Cardinale and E. Hernández. Parallel Checkpointing on a Grid-enabled Java Platform. *Lecture Notes in Computer Science*, 3470(EGC2005):741 – 750, February 2005.
- [DH95] J. Dongarra and T. Hey. The PARKBENCH Benchmark Collection. *Supercomputer*, 11(2-3):94–114, 1995.
- [EPC06] EPCC. The Java Grande Forum Benchmark Suite. <http://www.epcc.ed.ac.uk/javagrande>, 2006.
- [FH01] Carlos Figueira and Emilio Hernández. Profiling Facility on a Metasystem. *Lecture Notes in Computer Science*, 2110:42–51, 2001. Conference on High Performance Computer and Networking (HPCN’01).
- [Fig02] Carlos Figueira. *Evaluación de Desempeño de Aplicaciones en Metasistemas*. PhD thesis, Universidad Simón Bolívar, 2002.
- [FJP⁺05] Thomas Fahringer, Alexandre Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Jr, and Hong-Linh Truong. ASKALON: a tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):143–169, February - April 2005.
- [FK99] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Computational Grids, pages 15–51. Morgan Kaufmann Publishers, Inc., 1999.
- [FSJY02] Michael Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Implementation of the nas parallel benchmarks in java. Technical report, Nasa Advanced Supercomputing, 2002.
- [GBN04] G. Gombas, Z. Balaton, and Z. Nemeth. Performance evaluation on grids: Directions, issues and open problems. In *12th Euromicro Conference on Parallel, Distributed and Network based Processing*, 2004.

- [Glo03] Global Grid Forum. Grid Performance and Information Services, 2003. <http://www.gridforum.org/>.
- [GLon] William Gropp and Ewing Lusk. User's guide for mpe: Extensions for mpi programs, electronic publication. <http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeguide/paper.htm>.
- [HCFT00] Emilio Hernández, Yudith Cardinale, Carlos Figueira, and Alejandro Teruel. SUMA: A Scientific Metacomputer. In *Parallel Computing: Fundamentals and Applications. Proceedings of the International Conference ParCo99*, pages 566–573. Imperial College Press, 2000. ISBN 1-86094-235-0.
- [Her96] Emilio Hernández. *A Methodology for the Design of Parallel Benchmarks*. PhD thesis, University of Southampton, 1996.
- [HJ81] R. Hockney and C. R. Jesshope. *Parallel Computers: Architectures, Programming and Algorithms*. Adam Hilger Ltd., Bristol, UK, 1981.
- [HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High-Performance Computing Conference*, pages 841–850, 1994.
- [Hoc95] R. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 11(2-3):94–114, 1995.
- [Jav] Java Grande Forum. <http://www.javagrande.org>.
- [MLVB05] S. Markidis, G. Lapenta, W. B. VanderHeyden, and Z. Budimlic. Implementation and performance of a particle-in-cell code written in Java. *Concurrency and Computation: Practice and Experience*, 17:821–837, 2005.
- [PH96] David Patterson and John Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan/Kaufmann, second edition, 1996.
- [Sun05] Sun Microsystems. Javatm virtual machine profiler interface (jvmpi). <http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html>, Consulted April 2005.
- [TAG⁺] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A grid monitoring architecture. Global Grid Forum. <http://www-didc.lbl.gov/GGF-PERF/GMA-WG/>.
- [The06] The Globus Alliance. The Globus Toolkit, 2006. <http://www.globus.org/>.
- [VGr] VGrADS. The Virtual Grid Application Development Software Project. <http://vgrads.rice.edu>.
- [VR99] Jeff Vetter and Daniel Reed. Managing performance analysis with dynamic statistical projection pursuit. In *SC'99*, 1999.