Interpreted Applications within BOINC Infrastructure

Daniel Lombraña González[1], Francisco Fernández de Vega[1], L. Trujillo[2], G. Olague[2], M. Cárdenas[3], L. Araujo[4], P. Castillo[5], K. Sharman[6], A. Silva[7]

[1] University of Extremadura daniellg@unex.es, fcofdez@unex.es
[2] CICESE trujillo@cicese.mx, olague@cicese.mx
[3] Ceta-Ciemat miguel.cardenas@ciemat.es
[4] UNED lurdes@lsi.uned.es
[5] University of Granada pedro@atc.ugr.es
[6] Polytechnic University of Valencia ken@iti.upv.es
[7] CICA asilva@cica.es

Abstract. BOINC is one of the most employed middlewares in the scientific community. However, the development of BOINC applications could be difficult if the target application is an Interpreted Application such as Matlab, R or Java. The BOINC team provides an intermediate solution, the wrapper, which can run statically linked programs. Nevertheless when the application has lots of dependencies, BOINC will not be able to deploy it. In this paper, we propose to exploit the BOINC infrastructure with Interpreted Applications by complementing the wrapper program with a new application and extending the whole BOINC infrastructure by adding a new virtualization layer, and best of all without modifying the source code of the interpreted application. Three experiments using well-known interpreted applications -Java, R and Matlab- are performed to demonstrate the viability of running unmodified interpreted applications inside a BOINC infrastructure.
Key words: BOINC, Interpreted Applications, Virtualization.

1 Introduction

Nowadays it is common to have powerful desktop computers at universities, institutions, etc. A basic desktop has a powerful multicore microprocessor, 1 GB or more of RAM memory, large and fast hard disks, etc. However, those desktop features are not completely harnessed, because most of those PCs are used for non intensive CPU tasks such as browsing the web. Moreover, those PCs have large idle periods, so it is reasonable to say that the computing power of those PCs is wasted.

In order to exploit those PCs resources there are a technology which tries to harness all the PC resources by installing a software. This technology is known as Volunteer GRID Computing (VGC), and its main goal is to harness all the commodity computer resources and provide them to scientists. VGC employs a software named *middleware* which is in charge of harnessing the computing resources such as CPU power or hard disk space by being installed on the PCs.

There are different approaches: (i) *Condor* [10], (ii) *Xtremweb* [6]), (iii) *BOINC* [2]. From all the above middlewares, BOINC is the most used and widespread one. BOINC has a large pool of projects such as climate prediction simulations [1], physics research [12], etc. Additionally, BOINC has a great community support.

BOINC is a framework that was born from the project SETI@HOME [4]. Thanks to the VGC technology, SETI has achieved a virtual super computer of 414.040 TeraFLOPS[8] by employing cheap computers: PCs. Thus, BOINC is a good framework to run scientific computations. However, in order to use BOINC it is necessary to port the source code to BOINC or use a small BOINC application called *wrapper* to run statically linked legacy applications (applications that does not use the BOINC API) without the necessity of the porting step. However, there are interpreted applications (IAP) such as Matlab or R which are not statically linked and more important widely used by scientists. Hence, there is a set of widely used scientific IAP that cannot benefit from the BOINC technology.

In conclusion, BOINC is a great middleware to harness desktop computer resources. Nevertheless, BOINC has to deal with the problems of running scientific applications inside BOINC, having some times complex applications which cannot be run inside BOINC. Therefore, what we propose is to run IAP by complementing the wrapper functionality with other program. This new program called *the starter* lets aware the user of the lacking IAP and/or set up everything to receive and process the IAP jobs from its BOINC client. Finally, we extend the BOINC structure by adding a new layer which simplifies the deployment of IAP within BOINC.

The remainder of the paper includes an explanation about the BOINC middleware architecture in Section 2; the different ways of employing BOINC in a scientific project in Section 3; the experiments and results in Section 4. We conclude and briefly present the future plans for the proposal in Section 5.

2 BOINC

BOINC is a middleware which harness the commodity computer resources for a given project. BOINC has two main key features: (i) it is *multiplatform* (MacOSX, MS. Windows and GNU/Linux) and (ii) *open source*. BOINC employs a Master-Slave architecture where the slaves are the desktop PC clients and the server is in charge of distributing the work between the clients. The next subsections explain more deeply the BOINC architecture.

2.1 The Client

The client is a multiplatform application which runs on Windows, GNU/Linux and MacOSX. The client is in charge of polling the BOINC project server and request jobs to process them. Once, the server has jobs for the client, the client downloads all the necessary files, stores them in a client folder, and starts to process them. When the client has finished the computation, uploads the results (the output files) to the server and request again new jobs.

[8] Data obtained from the web http://boincstats.com

2.2 The Server

The server is the main place where the scientist creates a research project. The server is in charge of:

- *Hosting the scientific project experiments.* A project is composed by a binary (the algorithm) and some input files. BOINC builds different binaries for each different architecture, hardware and software OS.
- *Creation and distribution of jobs.* The server uses the binary and input files to create a *Work Unit* (WU). A WU describes how the experiment must be run by the clients, specifying which is the binary, its input files, the command line arguments for the binary and the output file names that will be generated by the binary.
- *Assimilation and validation of results.* When the clients finish the computations, the output files are uploaded to the server. Then, the server launches two processes. First, the *validator* program, which is in charge of assuring that all the results are correct and the clients have not cheated. Once the *validator* validates each result, the server launches the second program: the *assimilator*. The *assimilator* is in charge of parsing the output files of the project to perform tasks like: compute some statistics, export the results to a data base, etc.

In summary, the server is in charge of building and distributing a WU which is composed by the input files that a binary needs in order to compute a given problem. Once the clients have performed all the computations and the results (output files) are uploaded to the server; the server launches the last two main processes to check the correctness of the results and do some data analysis (for example some statistics).

3 Using the BOINC Framework

The previous section has explained the main structure of a standard BOINC project (server and client). The structure is simple, but the process to build a BOINC project could be difficult due to the necessity of building BOINC binaries using its libraries.

A BOINC project can be built from four different points:

1. *From scratch.* In this case, the scientist or developer starts a BOINC project from scratch. There are not source code at all, so the developer has only to take into account all the BOINC requirements about libraries, and programming language structures. BOINC is coded in C++, so the developer will create its BOINC scientific project coding it in C++. This case is the ideal one.
2. *Adapting a C++ program.* In this case, the developer or scientist has to change all the Input/Output (I/O) methods that his application uses by the BOINC I/O routines. This change is necessary because the application is going to be deployed under a parallel environment, so BOINC uses special I/O routines which circumvents and solves all the derived problems from a parallel environment.

3. *Porting a program to C++.* In this case, the researcher has a program which is coded in a different language to C++. In this scenario the scientist has to port all the code to C++. Thus, the modifications are important, and in some cases the researcher cannot afford the porting step due to time constraints or complexity reasons.
4. *Using the* wrapper. The wrapper allows to run applications which don not use the BOINC API, for example when the source code of the application is not available. This last method allows to run applications from the previous point, where the scientist has the source code but the porting step to BOINC is very difficult. In summary, with this new method the researcher has not to port his code to C++, but only if his application is statically linked.

However there are some scientists who employ programs like Matlab, R, etc. to solve their problems, so all the source code of their applications are coded in Matlab or R programming language. Additionally, those IAP usually have lots of modules or libraries which expand its usability to different fields such as physics, simulations, GRID, etc., increasing its complexity. The GRID modules are not suitable for VGC, so the *wrapper* solution is still needed. Nevertheless, and due to the complexity of the IAP, the *wrapper* in some cases will not be sufficient to tackle the IAP problem.

Hence, what we propose is to run any IAP with BOINC using the *wrapper* and the *starter*. Finally, we extend the possibilities of using BOINC by employing the *virtualization* technology. The next two subsections explained more deeply the *wrapper*, the *starter* and the *virtualization* extension.

3.1 The Wrapper

The *wrapper* is a small BOINC program. This program is written in C++ and basically wraps the "legacy application" (applications which does not use the BOINC API) hidden in it from the BOINC structure. Thanks to the wrapper, the BOINC client only sees one binary: the wrapper. The real binary is treated as a standard input file.

The wrapper uses an input file called *job.xml* which represents a job description file. This job file specifies a sequence of tasks where a descriptor for each task is defined. The following variables are configured:

- *Binary.* The name of the "legacy binary".
- *Input/Output files.* The names of the input/output files which are necessary for the legacy application.
- *Command line arguments.* The command line arguments that should be passed to the binary.

Thanks to the tasks section, it is possible to divide a big job into small pieces and run preprocessing and/or postprocessing tasks with different legacy binaries.

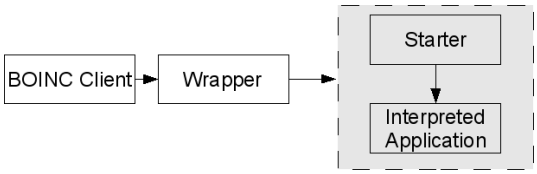
Up to now we have presented how it is possible to run "legacy applications" inside BOINC. However, if we want to run a Matlab or R script it is a bit more complex, due to BOINC cannot package a whole Matlab environment within BOINC.

Moreover, BOINC has not the feature of detecting the PC client installed software. Due to this problem we need a tool to check if the client has the possibility of running IAP jobs, so what we have created is a new program called *starter* which will deal with this issue.

3.2 The Starter

The *starter* is a small program which sets up the environment to run the IAP, for example a R script. The *starter* varies depending on the OS platform, so basically you have to create a starter for each target platform. In the case of GNU/Linux and thanks to its internal treatment of executables, the starter could be a script. The *starter* can check if the client has all the necessary infrastructure to run IAP,

Fig. 1. The Wrapper and the Starter



like for example Matlab. This checking step can be done by consulting system variables, or by searching on the installed applications. Moreover, if the needed infrastructure does not need administrative privileges to install it, the *starter* program can download all the necessary files, install them on a temporary folder, and delete everything after the computation has finished, leaving the computer without any trace. However, in most of the cases the client will need administrative privileges (take into account institutional computers from Universities, Administrations, Companies, etc.) so if a BOINC project needs to run a Matlab script the client has to install it before attaching his computer to the BOINC project. It is also important to point out that the starter can be used to warn the client advising that an installation step is necessary if the client wants to collaborate with the project.

In summary, the *starter* is launched via de *wrapper*, and the *starter* finally starts and deals with all the IAP (see Fig. 1) by warning the user or setting up the client to accept jobs from an IAP.

4 Experiments and Results

We have set up three different BOINC projects in order to check if our approach of running IAP such as R or Java within BOINC is possible. We used three different applications, one using the R statistical software, other using a Java-based Evolutionary Computation Research system and a Computer Vision problem which

employs Matlab. Moreover, the goal of the experiments is not to solve a given problem (we are not interested in checking the quality of the obtained results) but to check if it is feasible to run IAP within a BOINC infrastructure.

We have used for all the experiments GNU/Linux clients (except for the Matlab), due to GNU/Linux provides more flexibility with the scripting languages. Additionally, we have measure the performance improvement that it is possible to achieve when a BOINC model is used comparing it with the traditional and sequential mode of running only one machine. The standard equation to measure the speed up is:

$$A = \frac{T_{seq}}{T_B} \tag{1}$$

where A is the acceleration, T_{seq} is the consumed time by the sequential mode, and T_B is the consumed time by the BOINC mode. Additionally, we also measure the available computing power (CP) by using the method described by Anderson and Fedack in [3]:

$$CP = X_{arrival} * X_{life} * X_{ncpus} * X_{flops} * X_{eff} * X_{onfrac} * X_{active} * X_{redundancy} * X_{share} \tag{2}$$

For all the experiments that we have performed, $X_{redundancy}$ is equal to 1 because we didn't use the redundancy feature provided by BOINC. X_{share} is also equal to 1 because none of the clients shared its resources with other BOINC projects. Finally, $X_{arrival}$ and X_{life} are two important variables due to they measure the host churn (the volunteer computing project's pool of hosts is dynamic). The X_{life} variable is adapted in the three experiments, because we do not employ the same amount of time as Anderson in [3] (hosts that have not communicated in at least 1 month), to hosts that haven't communicated in at least 1 day. The rest of the variables measure the hardware specifications, see [3] for more details.

The following subsections explains more deeply how the experiments were set up and presents the obtained results.

4.1 ECJ a Java-based Evolutionary Computation Research System

ECJ is a Java-based evolutionary computation research system where it is possible to run different kinds of evolutionary strategies[9]. We have used the Genetic Programming (GP) automated method (see [9]) which creates a working computer program from a high-level problem statement of a problem. Genetic programming starts from a high-level statement of "what needs to be done" and automatically creates a computer program to solve the problem.

The problem that we are going to try to solve with GP is the Multiplexer of 20 bits. The problem consists in evolving a computer program which performs the 20-multiplexer function (see [8]). In general, the input to the boolean multiplexer function consist of k address bits a_i and 2^k data bits d_i, which has the form $a_{k-1} \cdots a_1 a_0 d_2^{k-1} \cdots d_1 d_0$ with length equal to $k + 2^k$. The search space for this function is equal to 2^{k+2^k}, hence the search space for $k = 4$ (20-multiplexer) is equal to $2^{1048576}$.

[9] For more information see `http://cs.gmu.edu/~eclab/projects/ecj/`

ECJ employs JAVA to run the experiments, so the clients a priori need to have a Java environment installed, as we have explained previously (see Section 3.2). However, Java provides a statically linked version where it is not necessary to install it on all the system, but only uncompressing it on a given folder of the machine. Thus, the clients do not need to pre-install a Java environment to run ECJ with BOINC. Moreover, thanks to this feature, the BOINC client can install the Java environment in one of the BOINC's folders and run the experiments from there without disturbing the client by warning him about an extra needed step in order to collaborate with the BOINC project.

An important issue with the Java installation process is the step of accepting the license. The first time you install a Java environment you have to accept a license. As we are going to employ Java in our University we can accept the license only once and then use it for all our computers. In summary, the BOINC project is composed by the following files:

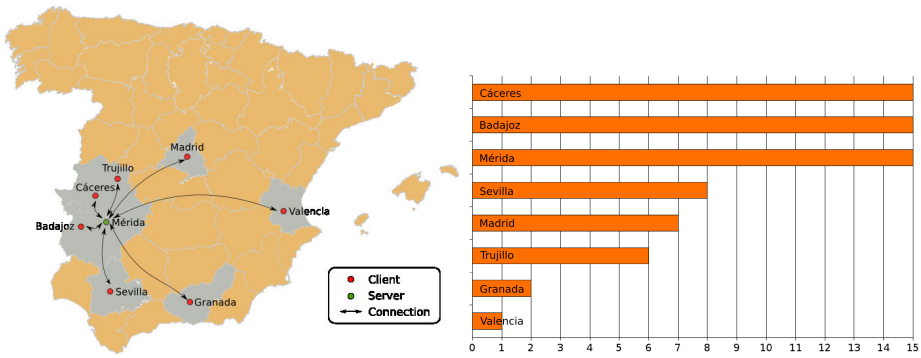
- *ECJ*. A compressed file which contains the ECJ environment.
- *Java*. A compressed file which contains the Java environment and the accepted license.
- *Starter*. A BASH script which uncompress Java and ECJ, and then runs the GP-ECJ experiments.
- *Wrapper*. The wrapper to run the starter script with its associated *job.xml* file (see Section 3.1).

One of the main drawbacks of the wrapper is that the wrapper is unable to do checkpointing. The BOINC team developers, recommends us to implement the checkpointing facility in the "legacy application". In this case, ECJ provides this feature, so the *starter* script runs and sets up the ECJ environment to run the GP 20-multiplexer problem. Moreover, thanks to the *starter* script it is possible to handle the stop and restart of BOINC client as a normal BOINC project, taking into account all the previous saved checkpointing states.

Once the server was set up, clients from different universities and institutions of Spain collaborated with this project: CICA in Sevilla, University of Extremadura (Cáceres, Badajoz and Mérida), University of Granada, Polytechnic University of Valencia, UNED in Madrid, and Ceta-Ciemat in Trujillo (see Fig. 2). This distributed infrastructure was possible to use because, thanks to the Java statically linked version and the *starter* script, the clients didn't have to be disturbed with the Java installation step.

As we have stated at the beginning of the section, the objective of this experiment was not to solve the problem but to check if the Java-based application ECJ was able to run within BOINC and produce some results. Using the above infrastructure, 42 runs of the experiment were performed during 7.75 days. A total of 41 machines were used to solve the problem. From the 41 PCs, 7 produced the 42 runs due to some machines were turned off for hours, others still computing, etc. (typical VGC behavior). The obtained speed up was 1.95, see Tab. 1. The acceleration was nice, although not impressive but it was obtained for free with a quite small number of volunteers. Finally, the best found Fitness was $Raw = 180224.0$ $Adjusted = 5.54862e-06$ $Hits = 868352.0$. It is important to remark that one

Fig. 2. Distributed Infrastructure



(a) Interconnected Cities (b) Clients per City

Table 1. Execution time for ECJ and ECJ-BOINC

	T_{seq}	T_B	Acc.	CP
20 bits, 42 runs, 50 Gen, 1000 Ind.	1305330s	669759s	1.95	23 GFLOPS

of the main benefits of this approach is that ECJ hasn't been modified at all in order to support the BOINC framework. Thus, the obtained benefit is big due to the researcher obtains a parallelized ECJ environment, without touching a line of ECJ source code.

4.2 R a Statistical tool

R is an open source statistical software[10]. R does not provide a statically linked version, so there are two options if we want to use it with BOINC: (i) compile it building a statically version or (ii) install it on all clients before the BOINC client. The faster and easiest way is to install it on all the clients, moreover when most of the GNU/Linux distributions have R packaged. Thus, we set up a laboratory with 20 GNU/Linux computers in the University of Extremadura, Mérida, Spain where we performed a proof of concept.

In all the clients we installed the R environment, and after that the BOINC client (see Fig.3). In this way, the BOINC client through the *starter* script will be able to run any R application or script. The R script is a proof of concept, so the results and measurements of the T_{seq} and T_B are negligible due to the time it tooks to run the script is not noticeable. The BOINC project is composed with the following items:

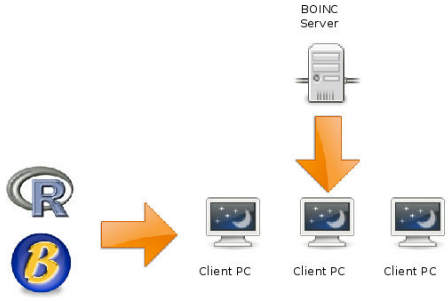
– *Wrapper*. The wrapper has its *job.xml* file that specifies which is the binary to be started.

[10] For more information see `http://www.r-project.org/`

– *Starter*. The starter basically sets up the R environment for running later the R script.
– *R script*. This is the "real" legacy application. It's the R problem to be solved by the script.

The R script basically computes some float and integer operations and finally writes an output file with the results. We set up a WU for 30 results, and all of them were uploaded into the server, so the experiment was successful. In spite of the successful of the proof of concept, R does not have a checkpointing facility so running big jobs of R could never be finished if the client machine does not be turned on at least the necessary time to complete one job. Thus, BOINC is a good option to run R scripts under controlled environments like laboratories of computers, where the researchers can control all the PCs for a given period of time. However, if we are talking of pure VGC, the R scripts could not finish if the home user powers off the computer or interrupts BOINC repeatedly in sort time intervals. Due to the problem of checkpointing in R, it is not trivial to run

Fig. 3. R and BOINC installation



it inside BOINC, so we propose the use of Virtual Machines (VMs). The VMs allows us to create virtual hosts where any software can be loaded and run, and best of all where it is possible to restart the whole system from a previous saved state. The next subsection explains the benefits of employing VMs within BOINC infrastructure.

4.3 Virtual Machines within BOINC Infrastructure

The *Virtualization* is a technology [5, 11] which allows to create virtual hosts where an OS can be loaded and run, so any scientific application can be run within this technology. The benefits of using this technology are the following ones:

– *Resource Isolation*. Thanks to virtualization each VM is isolated inside the host machine. This feature is very useful from the standpoint of security, the problems that can arise will affect only to the VM and not to the host machine.

– *Guest OS instantiation*. This feature allows the creation of OS images which can be loaded into any machine that is compatible with the employed virtualization technology.
– *Snapshots* or state serialization (also known as checkpointing). This feature lets restart a complete system from a previous saved state.

Therefore, thanks to the *snapshots* feature, BOINC can have for any IAP, a checkpointing facility. Moreover, by using the VMs, the scientists do not have to take care of the hardware and OS target platform, because the virtualization creates a special layer where the real hardware and OS are hidden. Furthermore, the researchers do not have to modify any single source code line, because with the virtualization they can create copies of their own scientific environments and run them in any BOINC platform. In conclusion, BOINC with a virtualization technology lets create custom executable environments with VGC.

There are different products that provides a virtualization technology. We have chosen VMware Player [11] because it is multiplatform, free, and runs in the same platforms as BOINC does. This last point is important because one of the goals of BOINC is to harness all the available resources.

In order to check the new approach one experiment was performed. We chose a Computer Vision problem which uses Matlab[11] and several tool boxes. The computer vision problem is a real life and time consuming problem (18 hours in average to obtain a solution), that has already been solved in a sequential fashion (see [13] for more details).

As we have explained, the Matlab environment employs several toolboxes, therefore deploying this environment within BOINC is very difficult. Moreover if the *virtualization* technology is not used, the Matlab-BOINC project lacks a checkpointing facility, so most of the WU will never get finished. Thus, the scientists are only required to prepare an image of their system: in this case a Debian GNU/Linux OS and the Matlab software. Then the BOINC administrator creates the BOINC project using the image as other input file.

Two laboratories from the University of Extremadura, Spain at Centro Universitario de Mérida were used as the test bed. In total, 20 PCs were employed, 10 with MS. Windows OS and the rest with GNU/Linux. All the computers were set up with BOINC and VMware (for further details see [7]). In summary, the BOINC project was composed with the following items:

– *VMware Image*. VMware employs images to run VMs, so the researcher has to provide one of this images. The image is a virtual PC with its hardware and OS, in other words, the scientific environment that the researcher needs.
– *Starter*. This script is needed to start the VMware Player software. Additionally, it will be used to take snapshots of the running VM by sending a signal to the VMware player. If the client powers off the PC or stops the BOINC execution, the *starter* will take care of resuming the computation from the last saved snapshot.
– *Wrapper*. The wrapper and its *job.xml* file as in all the previously explained experiments.

[11] For more information see `http://www.mathworks.com/`

While the experiment was running, the GNU/Linux laboratory was stopped because some administrative tasks were needed to be performed. This behavior is the hoped behavior from a VGC technology. The other 10 Windows PCs produced 12 solutions during 48 hours. The consumed time by each solution was in average of 18 hours. The total time consumed to produce the 12 solutions by a sequential run was 215 hours, so the obtained speed up was of 4.48. The CP obtained was of 25.76 GFLOPS (see Tab. 2). In conclusion, the *virtualization* technology opens the

Table 2. Execution time for Sequential and VM-BOINC Matlab

	T_{seq}	T_B	Acc.	CP
75 Gen, 75 Ind.	215h	48h	4.48	25.67 GFLOPS

possibility of running complex IAP such as Matlab, R, Java, etc. within a BOINC infrastructure. Moreover, the researcher benefits from this approach because his applications does not need to be ported or modified in any way and harness the parallel BOINC infrastructure. Additionally, the VM allows to never lose the on going work because of the virtualization's snapshot feature.

5 Conclusions

We have presented a new approach to run IAP such as Java, Matlab, R, etc. The new approach harness the program called *wrapper* which enables to run "legacy applications" (applications that do not use the BOINC API) by extending its functionality with a new program called *starter* that is in charge of setting up the environment to run the IAP. Finally we have proposed an extension by introducing the virtualization technology. We have proposed the use of VMs because the wrapper and the starter cannot assure the execution of IAP when a checkpointing facility is not embedded in the IAP. Thanks to the snapshots feature of the VMs any IAP can be run within BOINC. Moreover, without modifying the source code of the IAP.

We have tested our approach with three different experiments using different IAPs: ECJ, R and a Matlab. The ECJ experiment showed how is possible to run any Java software within BOINC and without the necessity of previously installing Java. The R experiment was a proof of concept that showed how it is possible to run R scripts in BOINC PCs by installing also the R environment in the client. Moreover, we explained how this last approach is not sufficient due to R lacks from a checkpointing facility. For this reason we extended BOINC with a virtualization layer. The Matlab experiment showed how it is possible to run any IAP within BOINC by installing a VM software in the client, demonstrating the benefits of using VMs within BOINC.

As a future work, it will be interesting to add to BOINC the possibility of installing IAP/VMs when they are necessary or at least warning the user about the lack of the desired IAP. Some discussion have been done in this way by the

BOINC community and also related to the virtualization layer in the Pangalactic forum of 2007.

6 Acknowledgments

This work was supported by Cátedra CETA-CIEMAT Universidad de Extremadura, Regional Gridex project PRI06A223 Junta de Extremadura and National Nohnes project TIN2007-68083-C02-01 Spanish Ministry of Science and Education.

References

1. M. Allen. "Do it yourself climate prediction". *Nature* (1999).
2. D. Anderson. "Boinc: a system for public-resource computing and storage". In "Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on", pp. 4–10 (2004).
3. D. Anderson, G. Fedak. "The Computational and Storage Potential of Volunteer Computing". *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)* (2006).
4. D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer. "Seti@home: an experiment in public-resource computing". *Commun. ACM*, **45**(11), 56–61 (2002). ISSN 0001-0782. doi:http://doi.acm.org/10.1145/581571.581573.
5. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. "Xen and the art of virtualization". *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 164–177 (2003).
6. G. Fedak, C. Germain, V. Neri, F. Cappello. "XtremWeb: A Generic Global Computing System". *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01)* (2001).
7. D. L. González, F. F. de Vega, L. Trujillo, G. Olague, B. Segal. "Customizable execution environments with virtual desktop grid computing". *Parallel and Distributed Computing and Systems, PDCS* (2007).
8. J. Koza. "A hierarchical approach to learning the Boolean multiplexer function". *Rawlins [1863]*, pp. 171–192.
9. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA (1992). ISBN 0-262-11170-5.
10. J. B. M. Litzkow, T. Tannenbaum, M. Livny. "Checkpoint and migration of unix processes in the condor distributed processing system". Technical report, University of Wisconsin (1997).
11. J. Nieh, O. C. Leonard. "Examining VMware". *j-DDJ*, **25**(8), 70, 72–74, 76 (2000). ISSN 1044-789X.
12. G. Robert-Démolaize. *Design and Performance Optimization of the LHC Collimation System*. Master's thesis, CERN (2006).
13. L. Trujillo, G. Olague. "Synthesis of interest point detectors through genetic programming." In M. Cattolico, editor, "Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2006, Seattle, Washington, USA, July 8-12, 2006", volume 1, pp. 887–894. ACM (2006).