nuBOINC: BOINC Extensions for Community Cycle Sharing

João Nuno Silva, Luís Veiga, Paulo Ferreira
INESC-ID / Technical University of Lisbon
Distributed Systems Group
Rua Alves Redol, 9
1000-029 Lisboa, Portugal
{joao.n.silva, luis.veiga, paulo.ferreira}@inesc-id.pt

Abstract

Currently, cycle sharing over the Internet is a one-way deal. Computer owners only have one role in the process: to donate their computers' idle time. This is due to the fact that it is difficult for an ordinary user to install the required infrastructure, develop the processing applications and gather enough computer cycle donors.

In this paper we describe a set of BOINC extensions that allow any user to create and submit jobs that can take advantage of remote idle cycles. These jobs are processed by commonly available software (e.g. programming language interpreters or virtual machines, statistical software) that is installed in the remote donating computers.

In order to submit their jobs, users only have to provide the input files, select the processing application and define the command line to provide to that application. Later, users of the same software packages will contact the server, receive a set of jobs, and process them using the already installed commodity application. These users can later take advantage of other people's computer cycles.

This system allows an expressive definition of jobs providing considerable speed gains, while leveraging a cycle-sharing platform and widely available commodity applications. Furthermore, it is a good stating point for the development of a truly global communal computer cycle market.

1. Introduction

BOINC [3] is a successful platform for distribution of parallel jobs to be executed on remote computers. Researchers create and publicize projects that require solving a complex problem, by installing a BOINC server, developing the data processing applications and creating the data sets to be processed. Later, users willing to donate their personal computers' idle cycles register themselves and start processing data with the applications downloaded from the server and developed explicitly for such projects.

This kind of operation greatly limits the scope of users that can create projects to be remotely executed. Projects must have a large visibility in order to attract enough cycle donors and be composed of hundreds of individual tasks or workunits. Furthermore, project creators must have a large knowledge on C++ or Fortran programming.

Projects from users that do not satisfy the previous characteristics can not take advantage of available remote cycles. Even if the user has enough programming knowledge to create a project, if the project is short lengthened or not capable of attracting enough donors, the gains will be low.

There are some computer user communities that can take advantage of remote idle cycles to speed their jobs, but do not have the skills to efficiently use BOINC. These users range from hobbyists or designers, that use ray tracing software to render movies or complex images, to researchers that user statistical software packages to process very large data sets. Researches who develop data processing applications on Java or Python can also take advantage of remote cycles to speed their jobs.

To include these new users as job creators in a cycle-sharing system, two key requirements are still to be met: i) the users should be allowed to use the applications or programming languages they are literate on, and ii) there should be enough cycle donors to speed even small jobs.

The use of commodity software as job execution environment reduces the data processing code development cost, as users need not learn a new programming language (C, or Java), being allowed to use the most efficient tool for the task. It would be infeasible to install these commodity applications remotely, but allowing the use of widely available software would increase the potential number of users that already had them installed. Furthermore, the use of previously installed software, creates a sense of community among users, as each user is willing to donate his computer's idle cycles to solve a job similar to the ones he will

later submit.

This sense of usefulness, that comes from the use of the same software as other users may increase the participation on such projects. After donating cycles to other users, a computer owner will also take advantage of remote cycles to speed his jobs. This way, users are compelled to provide more and more cycles to others. This usage pattern can be promoted if, instead of just sharing (i.e. donate) them, users can actually lend them and expect to employ them later in return. When needed, the credits received by letting the execution of tasks from others, will be exchanged by processing time on remote computers. This new relationship within the system will increase the number of users and the amount of time each user is able to share with remote users.

In this paper we present extensions to BOINC that allow efficient execution of user submitted jobs, while allowing any user to have two complementary roles: owner of the jobs that are executed on remote computes and owner of the computers where jobs will be executed. Any user that lends his idle cycles to the execution of other users' jobs will also be able to take advantage of their remote computing resources. In order to accomplish this, we modified both the BOINC client and server software, and developed a custom BOINC application. The data processing code used by these jobs comprises commodity applications that are installed in the remote computers, only after their owners allow their use.

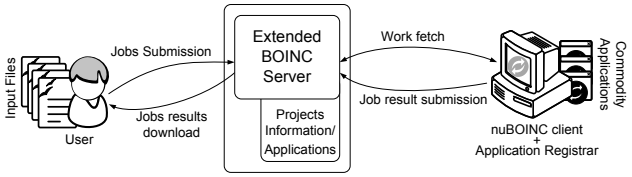

Figure 1. Extended BOINC usage

As shown in Figure 1 there are two different roles while interacting with our extended BOINC server: BOINC clients that execute the jobs and users that submit them. To submit and create new jobs, users must: i) select the commodity application that should be used to process the data, ii) provide the input files (data or code), and iii) define the number of jobs to create, the name of the output files and the arguments that should be used to invoke the commodity application. After creating and storing the information for each job, the server waits for BOINC clients work requests to distribute each job's information.

When contacting the server, the extended BOINC client sends the identifiers of the commodity applications the client owner has already installed. The server then selects the jobs to send according to this information. After receiving each job information (input files and arguments), the BOINC client invokes the correct commodity application to process the input files. After each job completion, the BOINC client submits the putput to the BOINC server. All data storage and communication is handled by the BOINC infrastructure.

From preliminary experiments we can conclude that the extensions to BOINC allow the definition and execution of a myriad of jobs that can take advantage of remote idle cycles. We managed to execute a batch of image rendering, necessary to create an animation video, as well as to process a batch of data input files within the R environment. Furthermore, several instances of a Java application were executed on remote computers. The speedups accomplished are satisfactory but not close to optimal. This is due to the fact that BOINC scheduling policies were designed for projects with thousands of jobs and when there is a continuous source of jobs. Furthermore current BOINC scheduling policies do not take into account the owners of the BOINC client and the submitted jobs, not guaranteeing fairness on the order of the execution of user submitted jobs .

In the next section we present existing cycle-sharing platforms and how they relate to our proposed solutions. In the following sections, we present the extensions made to BOINC and the evaluation done. Finally, we present the conclusions and future enhancements necessary to optimize the execution of user defined jobs.

2. Related work

BOINC is the best known platform for the creation and execution of distributed computing projects. BOINC provides all the data storage, communication and client management infrastructure. The project manager has to program a C++ or Fortran application that will be executed on the client computers to processes data. Even though it is easy to install project hosting infrastructures, two issues arise: it is necessary knowledge on C++ or Fortran to program the applications and, in order to have some speedup gains, it is necessary to publicize the project in order to attract clients.

BOINC wrappers [10] allows the use of legacy applications as processing code in a BOINC project. Project developers develop a simple wrapper and define the configuration file where it is stated how the legacy aplication will be executed. Before execution of jobs, both the wrapper and the legacy application are downloaded by the client. The wrapper is executed and, instead of processing the data, it invokes the legacy application. Even with this solution, short length projects or without capacity to raise cycle donors can not take advantage of BOINC.

XtremWeb [7] and Leiden Classical [11] are distributed computing projects that allow registered users to submit their jobs, as opposed to plain BOINC installations where only the system administrator creates jobs. In Leiden Classical there is only one data processing application and users

only submit input files to be processed by that application. XtremWeb is more versatile as it hosts several installed applications. In XtremWeb users provide the input files and define the command line arguments used to invoke the application. XtremWeb allows the use of a broader set of applications, but still requires the system administrator to install them. A user is not allowed to install a new data processing application to solve his problems.

P2P cycle sharing infrastructures may seem to allow job submission by users, but at the moment this is not the case. JXTA-JNGI [13] provides an API for the development of distributed cycle sharing systems. The development of a proprietary cycle sharing system, where the owner defines all the code and data communication, is possible but to make it efficient over the Internet it is necessary to publicize and gather cycle donors. Marcin Cieslak [5] describes how JXTA-JNGI can be used to cope with the scalability problems that rise from the existence of one server. A P2P architecture is proposed but nothing is done in order to facilitate jobs creation.

Other generic infrastructures (such as POPCORN [9] or P2P-G2 [8]) exist but still require the explicit programming of the code to be remotely used. These solutions free the user from programming the distribution and communication protocols, but still require the coding (in Java or C#) of the remotely executed code. A project developed with any of these solutions would still require a lot of publicity to get enough cycle donors.

The use of commodity data processing applications as a remote processing environment has also been proposed in NetSolve [12]. The major difference lies in the fact that in order to use NetSolve the commodity application (e.g. Mathematica, MATLAB, Octave) should be extended with an API to allow the distribution of the work. Furthermore, users should also adapt their scripts or applications to distribute lengthy functions.

Our proposed job definition user interface is closer to Nimrod [1] in the way data distribution is defined: the user defines the input files, the type of parameters and how they vary. Nimrod then generates all parameter combinations and assigns each parameter combination to task. Even though Nimrod helps on the combination of all parameters, the user must still have some programming knowledge, because the processing application must be coded and the data type of each parameter must be defined.

3. Usage

The extensions to BOINC we present in this paper allow user submitted jobs to be executed on remote hosts, using commodity applications.

In order to share his computer idle processing cycles, a user only has to download a modified BOINC client (the *nuBOINC client*) and the commodity *application registrar*. In this initial version these are different applications, but the *application registrar* can be easily integrated to the graphical version of the BOINC client.

The first steps are similar to the donation of cycles to any regular BOINC project: the user creates an account on our modified BOINC server and registers his *nuBOINC client* on our project (*ComBOINC project*).

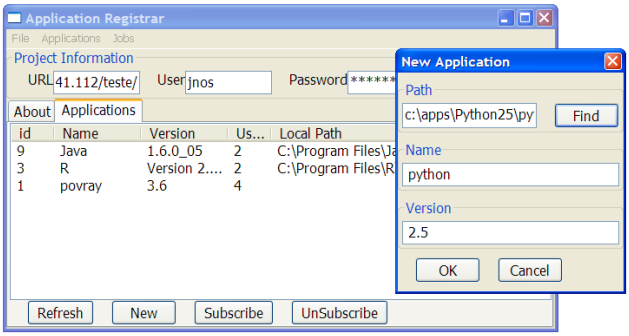

Figure 2. Application Registrar user interface

Before executing the modified BOINC client it is necessary to define which commodity applications can be used. This registrations is made in the *application registrar*; its user interface is shown in Figure 2. In this application the user locates the executables he allows to be used as processing environment for jobs. This application fetches from the modified BOINC server the list of applications already registered by other users. Now, the user can either assign any local executable to an already registered application (such as POVray in Figure 2), or he can register a new application using the *New Application* form.

After registering the commodity applications, when executing the BOINC client, some of the jobs will use those applications. The BOINC client will handle all data communication and invocation of the required applications.

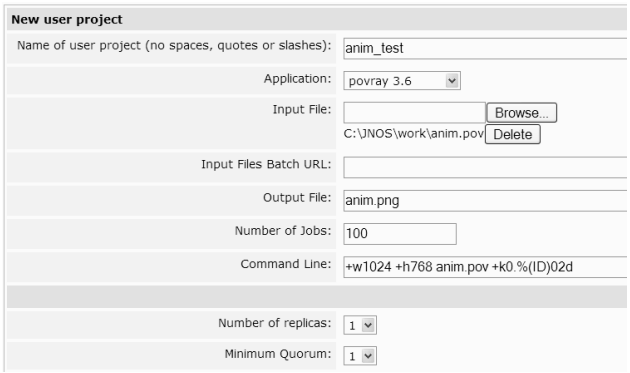

Figure 3. User project submission interface

A user that has a batch of jobs to be executed on re-

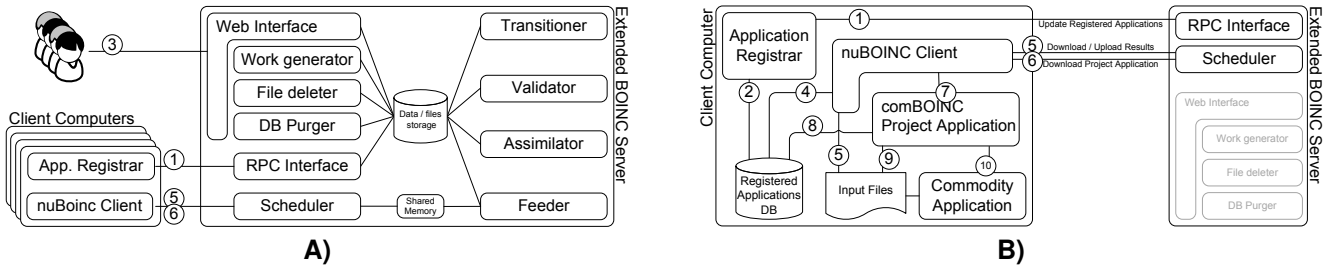

Figure 4. Client and server architecture: A) Detailed server view, B) Detailed client view

mote computers may use the provided web interface (Figure 3) to submit them. In this page the user uploads the input files and selects the application that should be used to process them. The user should also define the command line arguments that should be used when invoking the commodity application. In the example, the user wants to process a file (`anim.pov`) with the POVray ray tracer and generate a movie with 100 frames. For each frame (corresponding to one time instant), one different image will be generated. One hundred jobs will be created and, on each one, the POVray executable will be invoked with the `+w1024 +h768 anim.pov +k0.%(ID)02d` command line; `anim.pov` is the input file and `%(ID)02d` will be replaced with the job identifier (00, 01, . . . , 98, 99).

4. BOINC Extensions

The architecture of the developed infrastructure (presented in Figure 4) closely match the one of a regular BOINC installation [4]. THis figure also shows the ordered interactions between the different components (circled numbers). The *Application registrar* and *RPC Interface* components do not exist on regular BOINC installations, while all others maintain the same functionality.

In order to allow the execution of user submitted jobs a few changes to BOINC (server and client) were coded. To register the application allowed to be executed remotely , an auxiliary application (*Application Registrar*) was developed. The interaction between the *Application Registrar* and the BOINC server is made by XML-RPC calls.

The original BOINC *web interface* was modified in order to allow the job submissions and the results retrieval. The modules responsible for job creation and deletion (*Work generator*, *File Deleter* and *DB Purger*) are now invoked by the users by means of the *web interface*.

The modules that verify the validity and correction of a job execution (*Validator*) and that replicate or change a job state in case of a erroneous or successful execution (*Transitioner* and *validator*) were not modified.

The *feeder* and *scheduler* modules (responsible for delivering work to clients) were modified, so that the required *commodity applications* and the installed commodity application on the remote computer match.

With respect to job information organization within the BOINC server, only one modification was made. In regular BOINC installations, work is grouped in projects and all jobs from the same project are executed by the same application. With our extension, all user submitted jobs are processed within the same BOINC project (*ComBOINC project*) but belong to different *user projects*: sets of jobs, submitted by one user, that are to be processed by one commodity application. All other internal data organization remains. For each job there is one workunit (input files and execution parameters) and several replicas of each workunit, called results. As in any BOINC installation, results are sent to remote clients to be processed; after the processing of all results associated to a workunit, the valid or erroneous execution outcome and output is stored in the corresponding workunit.

On the client side some modifications were also made as shown in Figure 4. Besides the inclusion of the *Application Registrar*, the processing of the data received from the server is not performed by the *Project Application*, but by a previously installed *Commodity Application*.

Some of the interactions between the client computers and the BOINC server are also different. The server must be informed by all *Application registrars* about all registered applications, each job information is submitted by the client and, when requesting work, every *BOINC client* has to inform the server about the available *commodity applications* it provides.

4.1. Application registrar

After signing in at the BOINC server, the user must define what applications he allows to be used by remote users. The user must execute the supplied *application registrar* tool (interface presented in Figure 2). Only after registering a commodity application through the *application registrar*, such application is made available to other users.

After inserting the *ComBOINC project* and user information, this tool presents a list of applications (name and version), that other users made available, fetched from the BOINC server. If any of the presented applications is the

one the user wants to register, he only has to insert the corresponding executable disk location (screen shot not shown). If the user wants to register a new application, he must fill the *New Application* form. The user will supply the path of the executable and the name and version of the application. This information will later be presented to other users wishing to register their applications.

In order to interact with the server database (step 1 in figure 4), a XML-RPC interface was developed. The exported methods allow the query of the registered applications (name, version and number of instances) and the management of each user applications.

The information about the path of the executable is stored locally on each client (step 2 in Figure 4) on the local *Registered Application DB*. This information will later be necessary when fetching and executing the jobs.

4.2. User interface for job submission

The submission of the jobs to be executed on remote computers is also made in a straightforward way. A web browser is used to supply the input files and to define each job parameters. After logging in (at the *ComBOINC project*), a web page is supplied with a user interface similar to the one presented in Figure 3.

Here the user must define the total number of jobs, the name of the output file and the command line to supply to each job. There are also means to supply the input files. The user can upload files that will be accessed by every job or provide an URL pointing to a directory containing several files. Each one of the files present in that URL will be fed to a different job.

The information submitted by the browser (step 3 in Figure 4) is handled by a PHP script: for each defined job a workunit is created, containing the information about input files, output files, commodity application to be used and command line parameters. After the input files have been uploaded to the server and the workunit templates created, the *Work generator* is invoked in order to store relevant information in the database.

This version of the job submission interface, limits the type of jobs possible to batches of files (all processed with the same parameters) or to jobs with one variable parameter. The development of a more complex user interface (currently under development) will allow the definition of jobs, where the variable parameters are more complex, for instance multidimensional partitions, parameter combinations or based on more complex criteria (predicates, rules or regular expressions).

4.3. Database Tables

In order to accommodate the new information related to the registered applications and user submitted jobs it was necessary to add some tables to the original BOINC database. The relational model of the new information to be stored is presented in Figure 5. Shaded entities sets were already present in the original data model.

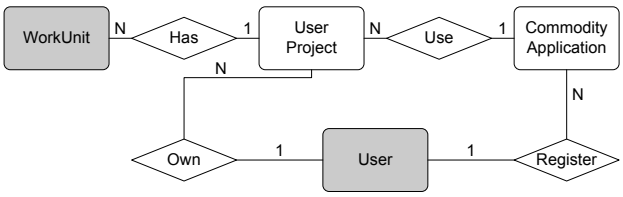

Figure 5. BOINC database new information

The *Commodity Application* entity set and the *Register* relationship were added to accommodate the names and versions of the commodity applications available on remote hosts. In order to restrict user access to workunits, results and their execution information, it was necessary to store the ownership information on the database (*User Project* entity set and *Has* and *Own* relationships). It was also necessary to store which commodity applications should be used to process the user generated jobs (*Use* relationship).

4.4. nuBOINC Client

Besides all the information regular BOINC clients send to the server, our modified client (nuBOINC client) also sends the identification of the commodity applications allowed to be used. This information is required for the selection of the suitable jobs to be executed on that client.

Before sending any request to any BOINC server the *nuBOINC client* tries to find information about the previously registered commodity applications (step 4 in Figure 4), stored by the *application registrar* on the local *Registered Applications DB*. If the information regarding registered *commodity applications* is found, a list of those applications is sent to the server along with a regular work request. The answer to this request contains the input files to be processed (step 5).

Besides the *commodity applications* list attachment to work requests, the interaction between our client and the BOINC servers is unchanged. After receiving a workunit to be processed (step 5) the client verifies if the required *BOINC application* exists. If this *BOINC application* is not present on the client computer, it is downloaded from the server (step 6). The *BOINC application* is executed (step 7) and processes its input files (step 9). After completion of the workunit processing, the output is sent to the server.

With the exception of the work request (that include a list of commodity applications) all other steps are common to any other BOINC client. The similarities between our modified client and a regular one make it compatible with any regular server. This way our nuBOINC client can process work from the *comBOINC project* or from regular BOINC projects. In this version of the client, the time slicing between regular jobs and *user jobs* follows the original BOINC scheduling rules.

4.5. Scheduler and Feeder

When contacted by an extended nuBOINC client, the comBOINC server should be able to return a workunit suitable to any of the commodity applications installed on the client. In order to accomplish this the scheduler and feeder modules had to be modified.

When requesting more jobs, the client can include the identification of the installed commodity applications. In this case, the scheduler has selects results that can be handled by any one of those applications. This way work is sent only to hosts that can handle it.

As shown in Figure 4, the *scheduler* does not interact with the database, it only accesses a shared memory segment. This shared memory segment is populated with the not yet processed results. The feeder module had to be adapted to handle the changes in the database and to communicate to the *scheduler* the possible *commodity application* associated with the results.

If the extended BOINC server hosts several projects, all other ordinary requests are handled in the same manner as in a regular server, guaranteeing the compatibility with all clients.

4.6. comBOINC Project application

If the downloaded workunit belongs to the *comBOINC project*, and consequently requiring our *Project Application*, the processing of the input files is different from the regular cases. In a regular BOINC project, its *project application* has all the code to process the data, while in a *comBOINC project*, its *project application* only handles the invocation of the correct *commodity application* that is needed to process the input files.

The parameters of our *project application* invocation (step 7) include the identifier of the *commodity application*, the names of the input files and the parameters to be used when invoking the *commodity application*.

Our *project application* first starts by finding the location of the required *commodity application* (step 8). Then creates a temporary directory, copies the input file there (step 9) and invokes the *commodity application*.

Upon *commodity application* completion, the output files are copied from the temporary directory. Then, the client returns them to the server in the normal way.

4.7. Commodity applications

Currently, the kind of projects we allow to be solved fit either in the parameter sweep category or in batch file processing, whose execution applications are already installed in the remote personal computers. These applications should either be parameterized through the command line or receive a script or configuration files as input. They should also easily generate output files and print error to the standard output.

The usable applications include a large set of applications: ray tracing software (POVray, YafRay), image or video processing (convert package, ffmpeg, ...), computer algebra (Maxima, Sage, ...), statistical software and data analysis (S-PLUS or R) or more general numerical computation applications (Matcad or Octave). Any other interpreted language execution environment (Java, python, Lisp) can also be used.

All these packages are used by a large community and are available on the largest families of operating systems: Windows and Unix and its derivative (Linux, Mac OS X).

Some of these *commodity applications* (specifically the programming languages interpreters) may be used to attack and abuse the cycles donor remote computer. To reduce those risks, our modified BOINC client and the *commodity applications* should be executed on a restricted environment. A virtual appliance, with a minimal operating system and the necessary applications, running inside VMware or other virtualization platform can be used.

4.8. Execution output management

After the processing of a workunit, its output files should be made available to the user who created it. This is accomplished by means of a few PHP scripts, incorporated in the BOINC *Web Interface*. After logging in, these scripts allow each user to inspect every workunit state (waiting to be processed, valid, erroneous) and to download completed results. Furthermore, after a user has downloaded all his output files, he can delete them, the input files files and his workunits information. This maintenance is performed by the *File deleter* and *DB Purger* modules.

5. Evaluation

In order to evaluate the usability and performance gains, we deployed our BOINC modified server and allowed some clients to use it; all the presented experiments were made on our local network. At the moment we are working on the

deployment over a larger scale allowing any user to register.

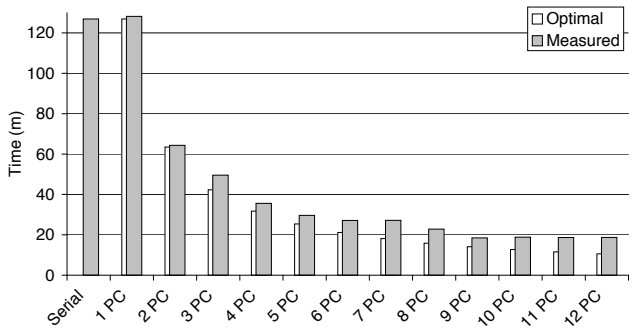

Figure 6. Movie rendering times

The first experiment performed consisted on using a ray tracer to generate an animation with 100 frames. On a Pentium 4 running at 3.2GHz with Linux, each frame took between 3 and 100 seconds, giving a total rendering time of about 127 minutes. The times for the execution of these jobs on several computers, shown in Figure 6, were measured with identical computers connected by a 100 Mbit/s local network. We present the time to execute the jobs serially on one computer (both locally and by means of the BOINC infrastructure) and on several computers.

As expected, the speedups are in line with the number of cycle donor hosts. The overhead incurred by using our job distribution platform is minimal, only 2 minutes. This is caused by the job submission and *nuBOINC client* startup. With the participation of another host, even during a small period, this overhead is not noticeable. On an wide area network, or with larger input files, this overhead is larger, but is easily surpassed with the contribution of another user.

With high number of participant computers, the difference between the optimal and the measured times varies. This can be explained with the help of Figure 7.

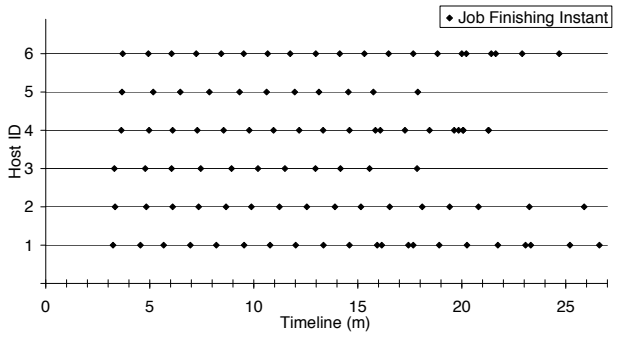

Figure 7. Jobs finishing instants

In this example we present every job finishing instant, for the rendering of 100 images on 6 computers. We can observer that from minute 22 on, only 3 hosts continue processing results, this being due to the fact that results are distributed in batches. As these batches maintain the same size during all execution, close to the finish one computer receives a batch whose jobs could be distributed equally to other clients, leading to a unbalanced job distribution. This inefficiency is added to the overhead incurred from using BOINC. With some numbers of participant computers (2 and 9 PCs in our example) the distribution of tasks between clients is closer to optimal.

In a system such as ours, gains from the use of remote computers are also dependent on the amount of time a remote computer is available and the frequency these computers contact the server to get new work. Between requests, BOINC clients introduces a delay (*backoff*) that is exponentially increased whenever a request is not fulfilled by the server (no more results to process, network failure, ...). In

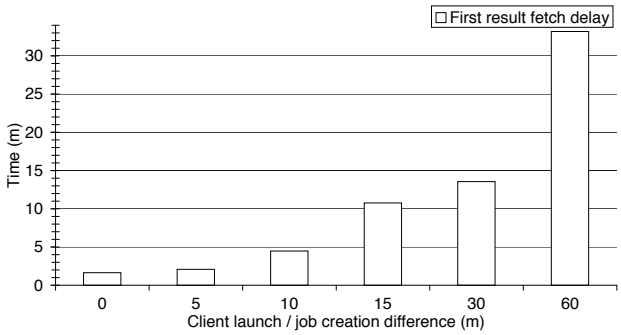

Figure 8. Computations start delay times

Figure 8 we can observer the influence of those delays. For each experiment we delayed the job creation with relation to the clients starting and measured how long it took for the first results to be sent to the clients. We can conclude that the clients, after an high idle time, will have a large *backoff* and may be idle even when some newly created projects have results to be processed. If the projects are short termed or the *backoffs* large, these projects may not take advantage of these clients.

We also experimented the deployment of jobs that used different applications: Java virtual machine and R statistical software. As long as the applications (the java application or R script) read some arguments from the command line its deployment is trivial using the developed job submission user interface. From these experiments we concluded that, along with the use of the task identifier, assigning a different input file (from a batch of files) to each job is also convenient, but a more expressive mechanism to define the parameters can be developed.

6. Conclusions

With the minimal modifications made to the original BOINC infrastructure we managed to extend it into a sys-

tem that allows the efficient execution of user submitted jobs. These jobs must fit in the parameter-sweep or bag-of-tasks categories.

We managed to integrate a job submission user interface to BOINC, allowing the definition of each job input files and parameters without any programming knowledge. Using this user interface, users don't have to develop any application to create the workunits information.

Furthermore, by allowing the execution of commodity applications as the data processing tools, users don't have to develop the BOINC applications to be executed on remote computers. Users only have to define the parameters or configuration files to the applications required to execute the jobs. These commodity applications are well known to the users and are already installed in remote computers.

As these applications have a large user base, it is very likely to get high gains from the execution of some jobs on remote computers, as lots of remote users will be able to donate their CPU cycles. Organized applications user groups can deploy a task distribution infrastructure to be used effortlessly by its members, independently of their computer expertize.

This preliminary evaluation shows that there is a wide range of problems that can be solved in parallel by commodity applications on remote computers and, that with a minimum of participation, good speedups can be obtained.

Furthermore, this system provides a fully functional platform for the development of a "cycle market" were any user can trade remote processing cycles.

6.1. Future Work

As the owner of a client may have jobs to be executed, the scheduling of jobs should be adapted so that a client process his jobs first. The way local [2] (on the client) and server[4] scheduling is performed should be modified. A client that still has uncompleted jobs should be able to execute them: the client scheduler should select the comBOINC project (ignoring other projects that should have its time slice) and the server should deliver to this client those tasks that are owned by him. Furthermore tasks belonging to users that have donated more time to the community should be executed earlier. This will add a sense of fairness and will motivate users to donate processing cycles.

Output integrity is partly handled by the original BOINC infrastructure. Workunits may have redundant results, that are processed in different hosts and later verified. A reputation system should be implemented so that users that process results incorrectly can be identified by the community. A proper scheduling mechanism, that can take into account reputation information, should be able to handle free riders or malicious users.

Even though BOINC is capable of achieving a high task delivery rate [4], it is necessary to study how a centralized BOINC server handles bursts of project creations. It may be necessary to replicate the server or use some other data distribution mechanisms [6].

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parametised simulations using distributed workstations. In *The 4th IEEE Symposium on High Performance Distributed Computing*, August 1995.

[2] D. P. Anderson. Local scheduling for volunteer computing. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 26-30 March 2007.

[3] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2006.

[4] D. P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*. IEEE Computer Society, 2005.

[5] M. Cieslak. Boinc on jxta (thesis from Technical University of Wroclaw, Poland). http://boinc.berkeley.edu/cieslak.pdf, June 2007.

[6] F. Costa, L. Silva, G. Fedak, and I. Kelley. Optimizing the data distribution layer of boinc with bittorrent. Technical report, CoreGRID Technical Report TR-0139, June 2008.

[7] C. Germain, V. Néri, G. Fedak, and F. Cappello. Xtremweb: Building an experimental platform for global computing. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*. Springer-Verlag, 2000.

[8] R. Mason and W. Kelly. G2-p2p: a fully decentralised fault-tolerant cycle-stealing framework. In *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*. Australian Computer Society, Inc., 2005.

[9] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet - the popcorn project. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1998.

[10] U. of California. BOINC WarapperApp – legacy applications. http://boinc.berkeley.edu/trac/wiki/WrapperApp, 2007.

[11] U. of Leiden. Leiden classical. http://boinc.gorlaeus.net/.

[12] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. Netsolve: Grid enabling scientific computing environments. In *Grid Computing and New Frontiers of High Performance Processing*. Elsevier, 2005.

[13] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*. Springer-Verlag, 2002.