# A Fast Job Scheduling System for a Wide Range of Bioinformatic Applications

Angelo Boccia, Gianluca Busiello, Luciano Milanesi, and Giovanni Paolella*

***Abstract*—Bioinformatic tools are often used by researchers through interactive Web interfaces, resulting in a strong demand for computational resources. The tools are of different kind and range from simple, quick tasks, to complex analyses requiring minutes to hours of processing time and often longer than that. Batteries of computational nodes, such as those found in parallel clusters, provide a platform of choice for this application, especially when a relatively large number of concurrent requests is expected. Here, we describe a scheduling architecture operating at the application level, able to distribute jobs over a large number of hierarchically organized nodes. While not contrasting and peacefully living together with low-level scheduling software, the system takes advantage of tools, such as SQL servers, commonly used in Web applications, to produce low latency and performance which compares well and often surpasses that of more traditional, dedicated schedulers. The system provides the basic functionality necessary to node selection, task execution and service management and monitoring, and may combine loosely linked computational resources, such as those located in geographically distinct sites.**

***Index Terms*—Cluster computing, job scheduling, PHP, Web interface.**

## I. Introduction

In THE LAST decade a large number of Web sites offering bioinformatic services have been created, and their use is now part of a scientist's daily "routine." In addition to documents and databases, they often provide interactive access to the execution of programs, ranging from simple tasks, such as simple manipulations of nucleotide or protein sequences, to sophisticated database searches and structural predictions, that may require powerful computational resources and that take advantage of the ability of the system to transfer the execution onto specialized computational nodes. In this sense, the availability of clusters of computing units has emerged, as a viable alternative to monolithic multiprocessor servers, for their ability to provide, at a reasonable cost, a highly scalable and operationally fast co mputing engine.

The distribution of the load, generated by a large number of requests coming from the users, requires a specific software architecture, where a central role is played by the job scheduler, a piece of software which is responsible for efficiently distributing the execution of the jobs to the available nodes. There are today several implementations of scheduling systems, which include several batch job schedulers, such as openPBS [1], PBSPro [2], Maui [9], MauiME [10], Torque [11], and LoadLeveler [12]. Specific alternatives are Sun Grid Engine (SGE) [3], a scheduling system oriented to the grid environment, and Condor [7], a high throughput scheduler, tuned to deliver large amounts of processing capacity over long periods of time. An efficient job distribution may also be obtained, but with deeper system administration involvement, with standard cluster software acting at the kernel level, such as openMosix [4]–[6], a load balance system for high throughput computing. More recently, new roads have been explored, which use more unconventional tools, such as OAR [8], an attempt to implement a PERL cluster resource manager operating at the application level.

One problem with batch schedulers is that most of them are optimized for the solution of large scale problems, requiring long computation times, and mainly focus on the completion of job execution. In this sense, these schedulers do not represents the best choice in an interactive situation, such as program execution through a Web interface. As it is not always easy to a priori distinguish between quick and slow tasks, given that the same program may end up by being very fast or really slow, according to the specific combination of parameters and datasets involved, an important point becomes the ability to avoid unnecessary delays in job submission, which would result in long latency, unacceptable for interactive jobs.

Here we present a new scheduling system, named FJS, designed according to the requirements of a typical Web environment, which, by focusing on the essential operations needed to handle the submitted jobs, only require very little activity to launch the requested task. The system, which may easily coexist with other schedulers on the same hardware, has been developed by using PHP [13], a high-level scripting language, and relies on a SQL DBMS, MySql [14], for managing the node status information.

## II. System Architecture

Some schedulers appear, to the user who is trying to submit a job, as a monolithic entity, which takes care of matching the request to the best available node, and then executes the task, often

Manuscript received November 21, 2006; revised March 1, 2007. This work was supported by the Ministero dell'Istruzione dell'Università e della Ricerca (MIUR) under the PON 2004 (SCoPE), FIRB (LITBIO), PRIN 2005, and BioinfoGRID European projects. *Asterisk indicates corresponding author.*

A. Boccia is with CEBSMA, Universià degli Studi di Napoli Federico II, Napoli, Italy, and also with Ceinge Biotecnologie Avanzate, 80145 Napoli, Italy (e-mail: boccia@ceinge.unina.it).

G. Busiello is with Ceinge Biotecnologie Avanzate, 80145 Napoli, Italy (e-mail: busiello@ceinge.unina.it).

L. Milanesi is with Biomedical Technologies Institute (ITB), National Research Council, 20090 Segrate, Italy, and also with CILEA, Segrate, Italy (e-mail: luciano.milanesi@itb.cnr.it).

*G. Paolella is with Dipartimento di Biochimica e Biotecnologie Mediche, Universià degli Studi di Napoli Federico II, 80121 Napoli, Italy, and also with Ceinge Biotecnologie Avanzate, 80145 Napoli, Italy (e-mail: paolella@dbbm.unina.it).

Digital Object Identifier 10.1109/TNB.2007.897474

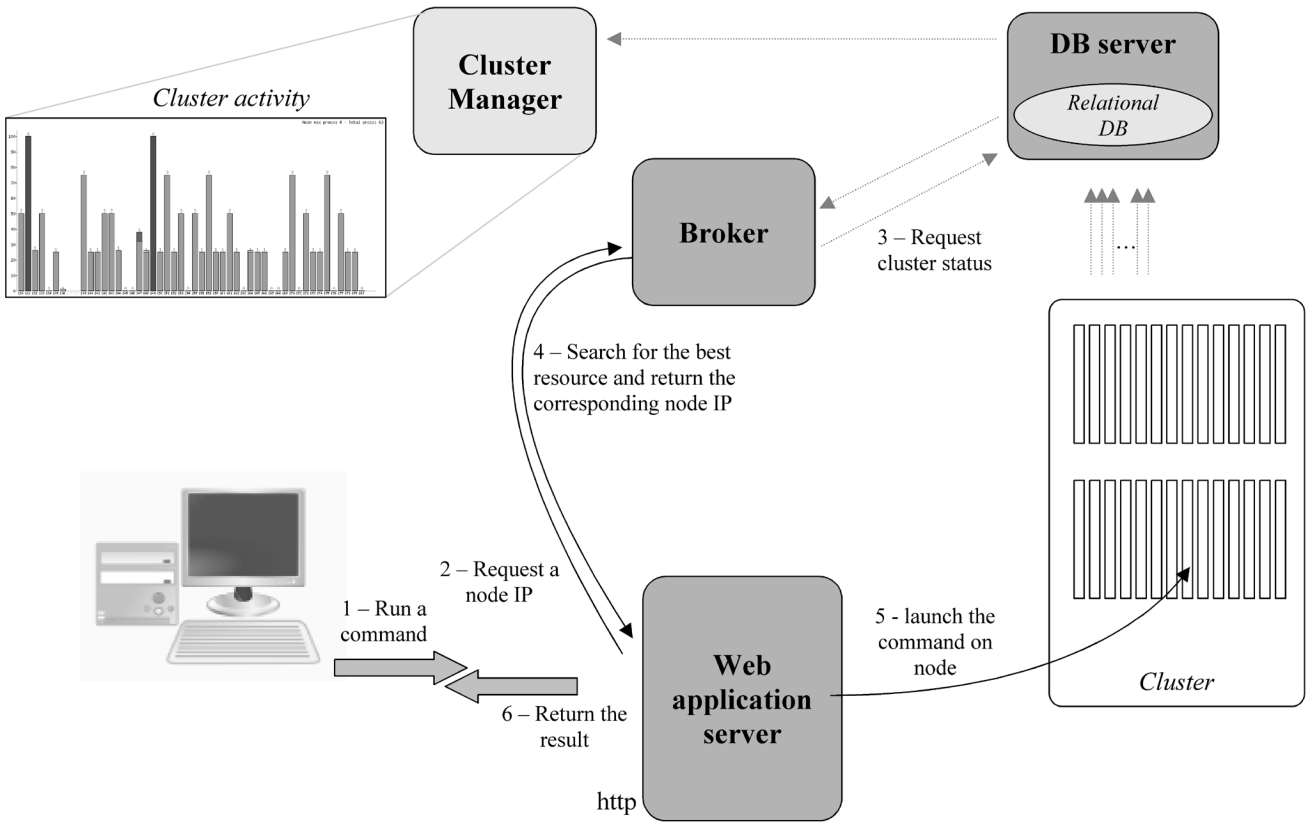


Fig. 1. Global architecture of the scheduling system, showing the central role of the broker, used to provide the requester with the best available resources.

also allowing monitoring and optional interruption of execution. An alternative model, used for example in Condor [7], uses a broker mechanism in order to find the most suitable node; all subsequent interactions between requesting and executing hosts is not mediated by the broker, resulting in lighter load on the broker itself and better scalability. This second approach was used in FJS, as illustrated in Fig. 1, where at the core there is a broker, able to quickly respond to requests for the best resource, available in the distributed computational system. Submission of a job consists of a request for a node from the broker, followed by remote execution of the command on that node. To match the request to the current load and to the resources available on the nodes, the broker makes use of the information contained in the DB server, a relational database continuously updated with information coming from the nodes. A small script, running on each node, periodically carries out the updates. The information contained in the DB is also used to manage the configuration of nodes and services and to interactively display the status of the cluster.

### A. Node Status Collector

Each node in the cluster is independently responsible for getting its own state, and communicating it to the DB server. The state consists of static information, describing the hardware configuration and software version, the presence in active form of specific services and the continuously changing load of the node. In detail the state consists of:

— machine type;
— number and type of processors;
— RAM configuration;
— OS version;
— list of installed services;
— load level;
— memory usage;
— swap memory state.

Most information is directly taken from the /proc virtual file system, made available by the kernel; for installed services a specific directory, maintained by the administrator, contains a file for each installed service, whose content is used to record the current state.

A simple script, running as a daemon on each node, periodically collects the described information, and immediately stores it in the DB server, by using an SQL update command. This periodic update is also used as a heartbeat signal, which flags the node itself as active to the system.

### B. DB Server

A relational SQL server contains the configuration and status information provided by each node, together with additional info on the recent usage of each node by the scheduling system. For each node, the DB stores the information indicated in Table I, which reports the organization of the "nodes" table.

Another table, "hosts," is used to keep track of the node requests, granted to each client host. The table is used to limit the number of node assignments granted to each host and is periodically flushed to guarantee acceptable performance. Additional tables on the DB-server are used to contain the descriptive information, relative to the various available services and other.

TABLE I
INFO STORED IN "NODES" TABLE

| Field | Description |
|---|---|
| total_mem | total memory installed |
| free_mem | free physical memory |
| total_swap | total configured swap memory |
| free_swap | free swap memory |
| total_procs | total number of processes |
| running_procs | number of running processes |
| available | if node is available for computation |
| rsh_connections | nr of rsh connections requested |
| sec_ips | nr of times the node is already assigned in the last second |
| load_us | load of code executed in user space |
| load_ni | load of code executed with low priority |
| load_sy | load of code executed in kernel space |
| load_id | idle |
| load_id_delta | idle variation |
| free_swap_perc | free swap memory percentage |
| last_update_db | time of last sql update |
| services | list of available services |
| max_sec_ips | maximum number of times the node can be assigned each second |

### C. Broker

The broker has been implemented in the form of a Web service and operates by retrieving the load and configuration data from the DB server and by giving back the IP address of the selected node. The node is chosen from a list of nodes matching a number of requirements, which take into account parameters such as system load, memory and disk space resources, number of times the node has been used in the last second, and, optionally, additional service requirements as described below. When no suitable node may be given back, the search is repeated several times, at short defined intervals, until a node is obtained or a timeout is reached, thus providing a short term, internal queuing system. The approach, involving simple SQL requests and fast computation, results in very fast response times.

### D. Job Execution

A number of alternative methods may be used to submit the job to the node. As for other systems, rsh may be used in a closed environment, with ssh as a slower, but more secure alternative for communication over an open connection. Another approach we used is to create a php wrapper for program execution, which is installed on the nodes and used as a Web service.

### E. Web Application or Other Clients

The role of the Web application server is to provide Web access to the bioinformatic tools and to remotely launch the task, by using the IP previously obtained from the broker. For each command, a handler is present on the Web application server, which transparently translates the incoming requests into command execution on the selected node. If no suitable node is found available, within the requested amount of time, the client is expected to handle the negative reply in the best possible way, for example by trying again, if possible, or submitting to a batch queue or returning a "Not enough computational resources, please try later" message.

Other applications, such as scripts in PHP or other languages, may also directly request nodes from the broker Web service, and then submit the request to the node. Also in this case it is requested that the client handle the "no node available" situation.

### F. Virtual Nodes

The system is currently installed on a 56 node cluster, and is expected to scale up easily within a factor of at least ten times. However, as at some point the system is going to hit a ceiling, an additional mechanism was introduced, aimed to organize the nodes in a hierarchical way. This turns out to be flexible enough to virtually join the system to sets of nodes located in a separated site, or not directly accessible from the scheduler, or characterized by fundamentally different performance. The resulting system is able to transparently handle requests, without any need for the requester to know where the program is being executed.

The hierarchy may be produced by grouping together nodes, located, for example, in an external network, and making them available to the local cluster as a single, more powerful, but slower to respond node. The scheduler has now the option to distribute job requests to the virtual node, when the local resources are not suitable to execute the job or simply not enough for the currently requested load.

The system merges distinct computational resources by using the following strategy.

- The nodes of the remote cluster are configured on the local cluster as a single "virtual" node with extended resources.
- A "status collector" on the remote cluster gathers the status statistics from each node, summarizes them as a global cluster status, and store this information as the virtual node status.
- The scheduler, in computing the best available resources, takes into account the limitations introduced by virtual nodes, such as higher network latency, and an additional delay, due to the transfer of input and output files.
- An http service, installed on a cluster gateway, listens for a remote job submission and translates the remote submission request into a standard job submission to the local cluster. It is in charge of getting all the command parameters, store the input and output data as temporary files, set up the execution environment, launch the command, and return the results.

Considering the expected scheduling and execution overhead, the job submission to the virtual nodes is of course preferred in case of batch jobs and in case of overload of local resources. This approach is currently being tested to combine two different clusters located in geographically distinct sites, as in Fig. 2.

### G. Configuration and Monitoring

Several aspects of the system can be configured by means of a Web application (*clusterManager*), which also doubles as a monitoring tool, used to display the current status of the cluster.

*clusterManager* (Fig. 3) is used to store the necessary configuration information into the DB. For each node the DB stores parameters such as the maximum number of jobs that can be submitted per second per node, maximum number of nodes to grant per host, and other settings used for performance tuning. *clusterManager* allows to temporary disable one or more nodes in the cluster, to remove them from the pool of usable nodes. In

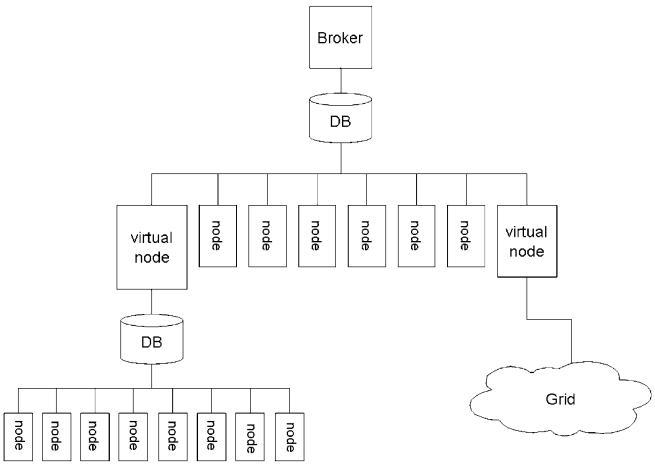


Fig. 2. Hierarchical architecture of the nodes, as viewed from the broker.

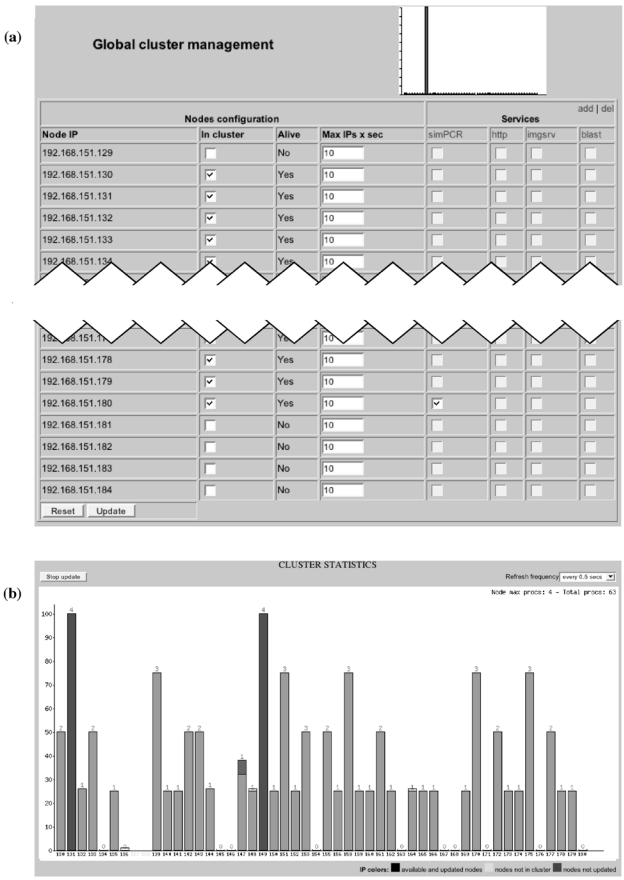


Fig. 3. *clusterManager*, a tool for configuring and monitoring the system. (a) Node configuration page. (b) Dynamic graphical representation of the current cluster status.

this way, for example, system maintenance may be carried out on the cluster without interfering with overall system activity.

The information describing the available services is stored in the DB, and is accessed through the same Web interface described above. However service availability is not directly configured, but independently controlled by the nodes themselves. In detail, the daemon running on each node periodically checks for the presence of a configuration file—one for each service—in a specific directory of the file system. Flags in this file indicate if the service is installed and active on that node. During the updates, the daemon transfers this information to the central DB. This approach provides different modes of control. In most cases the service configuration file is directly managed by the administrator. In a more complex environment, with the various services administered by different people, each administrator signals the availability of the service by modifying his own file. A script might, for example, set the file to "not active," carry out the maintenance and then reset the file content to "active." The file might even be substituted by a custom script, which, by itself might verify a number of conditions, such as available ports, running processes, and so on.

Monitoring consists of querying the DB, and presenting the information obtained in tabular or graphic form. A graphical representation of the current status of the cluster is provided in the form of a bar graph [Fig. 3(b)] which reports node availability, processor load, and number of running processes, together with some historical information. The refresh frequency can be directly modified by the user from the Web interface. In order to make it accessible by many contemporary users, the system provides image caching, to reduce the load on the system and to keep low the resources dedicated to monitoring.

## III. Job Scheduling

The scheduling routine returns a suitable node on the basis of computational resources and hardware and software configuration requirements. The scheduler tries to avoid overloading the nodes, by keeping track of the number of nodes assigned and the number of requests from each client within one second. This granularity was chosen by taking into account the update frequency of the status on the DB.

### A. Node Availability

In order to be selected, nodes must be configured as active by the administrator and must be running regularly. The DB update executed from the node doubles as a heartbeat signal, with longer delays since the last update interpreted as a dead or temporarily unavailable node.

### B. Job Execution Type

Processor load and usage of swap memory are probably the parameters that affect the performance most, once the job has been scheduled and submitted. However, different jobs have different requirements in terms of interactivity. We defined three typical situations, which correspond to different interactivity requirements:

- — fast, interactive jobs, which require the best possible performance from the system;
- — slower, but still interactive jobs, which, being intrinsically slower, can tolerate somewhat lower performance;
- — batch jobs, which do not pose specific requirements, but must be executed in all cases, no matter how long the whole process will take.

The behavior can be modulated by means of the configuration options; the standard parameters correspond to the selection of:

- — nodes with system load <= 25% and free swap memory >= 90% for faster jobs;
- — nodes with system load <= 75% and free swap memory >= 90% for slower jobs;

— all available nodes without special constraints on system loading and swap memory for batch jobs.

### C. Available Memory

If a job needs that a minimum amount of RAM is available on the execution node, it may be indicated as a requirement, that will be honored while choosing the node to be returned.

### D. Service Availability on Node

Many tasks do not require specific configuration or installation on the node, but some require that specific libraries or datasets are available. This is taken into account by allowing the job request to specify a service tag, that will force checking for the availability, in active state, of the specific service on the node.

### E. Job Distribution

By default the scheduler controls the load balance by distributing the processes to nodes with the highest free resources. A random factor is added to the load status in order to avoid reselecting always the same one, when more nodes share the same load status.

### F. Node Reuse

Balanced distribution is not always desirable. Sometimes, for example when a large dataset has to be read from disk at the beginning of a job, the shortest execution times are achieved by reusing the same node (up to a point). In this case a requirement can be specified to force the usage of the nodes already allocated, before a new node is selected. A typical example is represented by multiple searches with blast against the same dataset; in that case, the reuse of preloaded datasets, considerably improves the performance.

### G. Limiting the Number of Jobs Per Second

Considering the low latency provided by the broker in the selection of the node, especially if compared to the granularity of the status DB updates, additional control is used to try to avoid overloading a given node, by repeatedly selecting it during the relatively large time window before the execution is started and the load becomes high enough to prevent its selection. To do this, the system keeps track of the total number of times each node has been used within the current second, by storing this information in the DB as part of the node status. When this number reaches the predefined maximum number of processes to be submitted per second, the node is excluded from subsequent selections. This value depends on the number and type of processors, and may be separately configured for each node.

Similarly, the number of processes launched from a specific client may be limited, in order to obtain a better job distribution and to prevent a single user from blocking the system with a flood of repeated requests.

## IV. Analysis and Discussion

FJS was created as a result of a specific need for low latency execution of interactively requested jobs on a collection of nodes, as typical of current clusters. This emphasis is visible in Table II, where the main architectural features of FJS are shown, compared with other job schedulers.

TABLE II
FJS Architectural Features, Compared With Other Job Schedulers

| | FJS | openPBS | SGE | Condor | OAR |
|---|---|---|---|---|---|
| Batch/Interactive Mode | b/I | B/i | B/i | B/i | B/I |
| Queues | no | yes | yes | yes | yes |
| Broker based | yes | no | no | yes | no |
| Hierarchical node organization | yes | no | yes | no | no |
| Node specialization | yes | yes | yes | yes | yes |
| Preferential node reuse | yes | no | no | no | no |
| Integrated monitoring | yes | no | no | no | yes |

TABLE III
Broker Performance Test—Average of 10000 Requests

| Nr of requesters | Time (secs) | Latency (secs) | Nodes x sec | DB load (%CPU) |
|---|---|---|---|---|
| 0 | - | - | | 9 |
| 1 | 15.6 | 0.016 | 64.1 | 34 |
| 2 | 20.3 | 0.020 | 98.6 | 47 |
| 3 | 26.8 | 0.027 | 111.8 | 49 |
| 4 | 34.1 | 0.034 | 117.3 | 51 |
| 5 | 40.8 | 0.041 | 122.6 | 52 |
| 6 | 48.2 | 0.048 | 124.4 | 54 |
| 7 | 53.1 | 0.053 | 131.9 | 54 |

The system was tested in a typical environment where a single http server was acting as a frontend, submitting jobs to a cluster of 56 3 GHz P4 biprocessor nodes. The hardware used for the DB server consisted of a 1000 MHz biprocessor PIII with 1 GB RAM, and a 3 GHz P4 broker server. With the described configuration, the nodes updated the DB server once per second, with each update taking around 2 ms. The resulting load on the DB server was around 9% CPU time. In order to test the throughput and the latency of the system, 1000 requests for nodes were carried out from one to seven contemporary requesting clients. The results are reported in Table III, where it appears that by increasing the number of clients, the response time becomes longer, but even at the heaviest load, the time to obtain a node is well below 100 ms, for a total of over 130 nodes per second. Of course with more reduced loads, the time is better, up to only 10 ms in the best case.

The reported performance indicates that, although the chosen language is more typical of less critical scripts, such as dynamic Web pages, the overall behavior of the system results in very fast answers, with a job typically able to start within a few milliseconds of the request, even under relatively high load. This delay is much shorter than that observed in other, batch oriented, systems, which may typically imply waits of up to a few seconds before execution is started.

The various options for tuning node selection help to optimize resource usage when parallel execution of many jobs is requested. Scheduling execution of increasing number of BLAST searches, a typical bioinformatics application, which depends on the preload of a large dataset from disk, is reported in Fig. 4. For a small number of processes (up to five to ten), execution on a single node may be even faster than by using random distribution on cluster nodes, thanks to the effect of dataset preload; only when the number of processes goes beyond ten, the effect of parallel execution on cluster nodes becomes relevant. Selection of the "node reuse" feature allows to take advantage of both

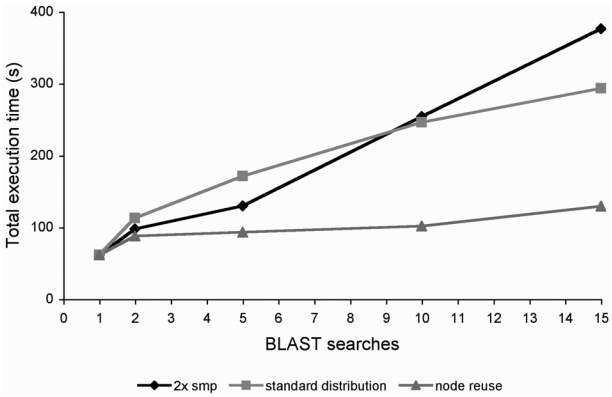


Fig. 4. Performance of the system on BLAST runs. A variable number of simultaneous BLAST searches was run on a single node (diamonds), with standard distribution (squares) or with node reuse (triangles). Total execution time is reported.

preloading effect and cluster distribution, and results in the best performance for both low and high numbers of simultaneous searches.

The system is organized around a MySQL database, which is known to be tuned for good performance, especially with small datasets that can easily be kept in memory, as in this type of application. Scaling up to around 500 nodes or a little more should not be a problem, especially if better hardware is used for the DB server, but well before reaching these numbers, it would probably be better to take advantage of the hierarchical option. This would also allow to include in the system machines of different level or location, especially considering that the requirement on the node is minimal in terms of installation, and that the system does not take over the calculating node or impose specific limitations. In this sense it is fully compatible with other job scheduling software tools. Finally a useful benefit of this approach is that the DB server, based on SQL language, can also be used for data analysis, thus allowing easy internal system monitoring and management.

## ACKNOWLEDGMENT

The authors would like to thank M. Petrillo for suggestions and useful discussions, and L. Cozzuto and C. Cantarella for testing the system.

## REFERENCES

[1] OpenPBS [Online]. Available: http://www.openpbs.org
[2] PBSPro [Online]. Available: http://www.altair.com/software/pbspro.htm
[3] Sun Grid Engine [Online]. Available: http://www.sun.com/software/gridware/index.xml
[4] OpenMosix [Online]. Available: http://openmosix.sourceforge.net
[5] OpenMosix, presented by Dr. Moshe Bar and MAASK [Online]. Available: http://openmosix.sourceforge.net/linux-kongress_2003_openMosix.pdf
[6] openMosix past, present and future, a peek at 2.6 [Online]. Available: http://howto.x-tend.be/openMosix.UKUUG2005/
[7] Condor [Online]. Available: http://www.cs.wisc.edu/condor/
[8] OAR [Online]. Available: http://oar.imag.fr
[9] Maui [Online]. Available: http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php
[10] Maui Scheduler Molokini Edition (MauiME) [Online]. Available: http://mauischeduler.sourceforge.net
[11] Torque [Online]. Available: http://www.clusterresources.com/pages/products/torque-resource-manager.php
[12] Tivoli Workload Scheduler LoadLeveler [Online]. Available: http://www-03.ibm.com/systems/clusters/software/loadleveler.html
[13] PHP [Online]. Available: http://www.php.net
[14] MySql [Online]. Available: http://www.mysql.com

Authors' photographs and biographies not available at the time of publication.