Materiały VII Krajowej Konferencji Algorytmy Ewolucyjne i Optymalizacja Globalna, Kazimierz Dolny, 24-26 maja 2004 243-250

A Campus Grid Using Globus and MOSIX

Adam Kozakiewicz[†], Andrzej Karbowski[†‡]

[†] Warsaw University of Technology,
Institute of Control and Computation Engineering,
Nowowiejska 15/19,
00-665 Warsaw, POLAND,
akozakie@ia.pw.edu.pl

[‡] NASK (Research and Academic Computer Network),
Wąwozowa 18,
02-796 Warsaw, POLAND

Abstract: A simple architecture for a campus-wide grid utilizing the resources of computer laboratories is presented. The environment combines the Globus Toolkit with MOSIX, and is mostly designed for MPI applications. A small experimental grid was set up and tested on a real-life example – finding an eigenvector of a large sparse matrix (Google's PageRank algorithm).

1 Introduction

Even though computing power is necessary in many scientific applications, universities often lack sufficient resources for a real, expensive HPC center. In place of a supercomputer, clusters appear as a good solution for most problems, but even they are far from being cheap. At the same time a lot of processing power is lost in computer laboratories, left unused for many hours a day.

Connecting available computers into a grid can offer considerable computing power, although rather restricted to coarse grain computations. Such a solution is cheap and can be quite easy to use. Using laboratories regularly occupied by students is a bit different than connecting regular workstations and some new problems have to be dealt with.

In this paper we show how to build such a grid by combining several available software tools into a single system, which, although complicated in structure, can be used by researchers with reasonable experience almost without preparation or learning new commands. Section 2 deals with the details, including a description of a small proof-of-concept test grid. The grid was tested using a real-life example – an implementation of the Google PageRank algorithm. The problem and test results are described in section 3. We finish the paper with some general notes and observations in section 4.

2 The Grid

The most direct approach to building a grid is to run parts of computing tasks on individual computers, using some form of a remote access tool with a built-in scheduler. Such a solution already exists and is well tested (Condor: [1], [2]).

The problem with such a grid is its heterogeneity. Computer laboratories run different operating systems, often multi-boot. Connecting them into a single grid would reduce the total computing power, unless the user makes the extra effort to make different systems communicate (possible, but often difficult). Moreover, computers in the labs may only be connected to the grid at night and sometimes during the day and it would be a difficult job for a grid-wide migration system.

Similar results can be achieved with a different, two-layer architecture. In this case the computers are organized into separate clusters, dealing with most reliability problems, and the clusters are connected into a grid as larger virtual machines.

2.1 System Architecture and the Choice of Tools

The two layers – the upper, grid layer of the system, connecting computer rooms to create a single virtual machine, and the lower, cluster layer, connecting the machines in one room to offer their resources in the grid as a single virtual machine are completely different. Different problems arise and different software solutions are used in both cases.

Both layers are unified from the end users' point of view. The end user doesn't need to know the details of the implementation, as for him the grid is simply an MPI-based cluster. We concentrate on MPI, since it is a standard, well known environment for parallel computations on distributed memory machines.

2.1.1 The Cluster Layer

In this layer we deal with a set of machines under control of one administrator. The machines need to be detached from the grid during the courses and reattached afterwards. Ideally, the process should not involve the administrator, it should be simple enough to be carried out by the students or teachers leaving the room. The machines cannot be a part of the grid when students work with them, since the machine could be incorrectly rebooted, killing the tasks being processed. If only part of the machines is used during laboratory classes, then the rest can still be connected to the cluster.

As the cluster's computing power changes in time, making individual machines visible from the MPI level would introduce a new problem: how to add and remove machines during computation? Process migration is not a simple task.

The suggested solution uses a MOSIX ([3], [4], [5]) cluster and a dedicated machine in each room. Only this dedicated machine is visible as a part of the grid, and it is the only one with MPI installed. All processes are running on this machine. When other machines are connected during computation, they share the load as MOSIX migrates part of the processes to them. From the grid level the cluster is a single machine with variable computing power. The cluster can also be used separately, disconnected from the grid – tasks can simply be run from the access station, or, as a matter of fact, from any station.

For computers working by default under Linux the configuration is trivial and (dis)attaching them from the cluster is done simply by starting or stopping the `mosix` service. Computers

running other operating systems have to be rebooted to Linux to join the cluster. As the stations, except the dedicated access point, don't need much except the kernel, a simple bootable floppy or CD seems to be the best choice. Although preparing the first one requires some work, copies for other machines need few changes. This approach is not only sufficiently simple and feasible, it also deals with the issue of software heterogeneity by eliminating it.

2.1.2 The Grid Layer

In this layer we have to connect several rooms with different security policies, account names, passwords, and so on. This is a situation typical for a true grid and requires grid software.

We chose the Globus Toolkit 2.4 ([6]), as it is rather popular, well tested and well integrated with MPI (MPICH-G2, see [7], [8]). The Globus Toolkit provides uniform login using certificates, simplifying the construction of the grid.

The communication between processes is provided by the MPICH-G2 package – an implementation of MPI 1.1 using Globus as the communication device. Unfortunately, this type of communication is not very efficient. However, our tests show that the speed is sufficient to make the grid useful, and the efficiency cost of using Globus may be negligible in the case of very long computations (hours, days, weeks) with minimal communication.

While the cluster layer is limited to one architecture (x86), using Globus it is possible to also connect other machines to the grid if desired. It may be possible to propose different cluster-layer solutions for other architectures, alternatively machines can be connected directly to the grid layer.

2.2 Test Grid

For testing we have used 9 computers from our small MOSIX cluster (a total of 11 × Pentium III/450MHz/128MB, running RedHat Linux 7.3, kernel 2.4.22 + MOSIX 1.10.1). We have reconfigured those machines into two separate clusters – 4 and 5 machines.

On one machine from each group we installed the Globus Toolkit 2.4 – resource management client, server and SDK. On top of it we built MPICH-G2, only those two machines were connected on the MPI level. This configuration allowed us to test running programs on one machine, two machines and two clusters. The setup is shown in Figure 1. The clusters "MOSIX A" and "MOSIX B" are separate. Only standard migration takes place, all messaging is done as if all processes still were on the same computer. The "MPI" section is the test "grid". Job execution is done using the Globus Toolkit.

Some configuration problems were found when connecting the two layers. The Globus Gatekeeper, the daemon responsible for running remote jobs, is started by the xinetd daemon. Under MOSIX, xinetd and most other daemons, are "locked", i.e. cannot migrate. This state is inherited by child processes, so without changes the processes would never leave the section "MPI". The first solution was to have the programs unlock themselves. It helped, but the MPI-enabled stations still were overwhelmed by processes spawned by the Globus tools. The solution is to configure the gatekeeper service as migratable (running it via `mosrun -l`) – in this case not only can the programs skip unlocking themselves in this case, but the communication processes will migrate too! As can be seen in the next section, after this change the cluster tolerates many processes well.

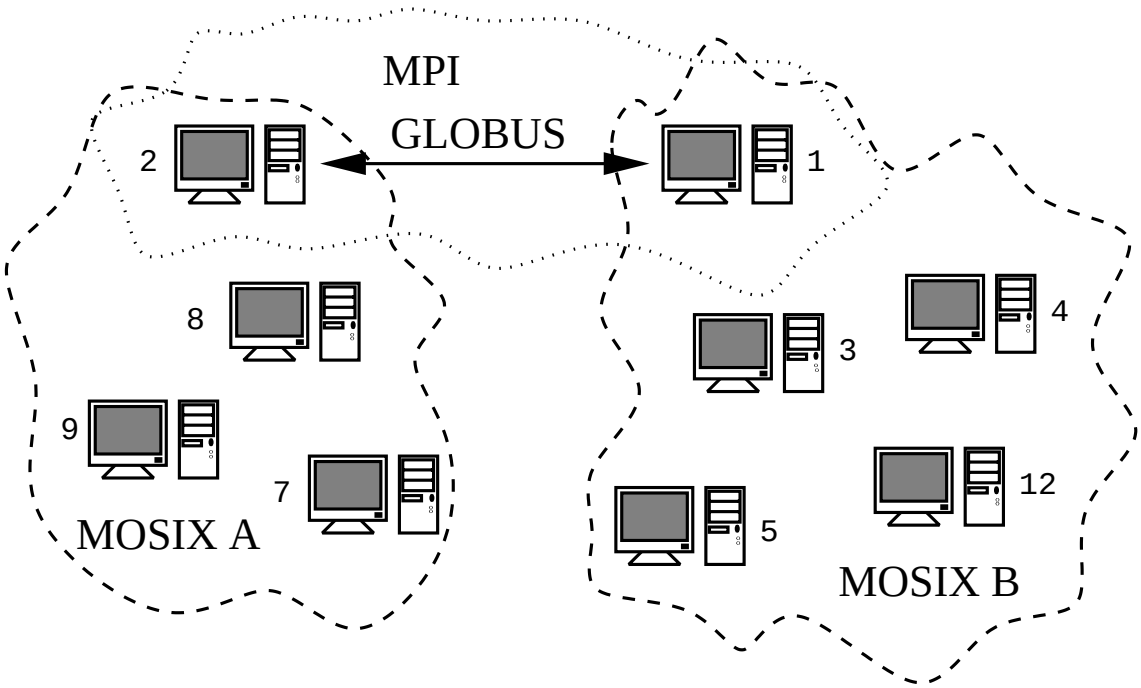

Figure 1: The setup of the test grid.

3 The Test Problem

For testing we have chosen a large scale linear algebra problem. We implemented the numerical core of Google PageRank algorithm for ranking WWW pages, which is in fact a calculation of an eigenvector using the power method (see [9], [10], [11]).

3.1 The Google PageRank Algorithm

The PageRank algorithm attempts to rank pages so that the most often referred to pages get the highest ranking. Additionally, it attempts to give more weight to links from pages with few links.

The links are represented by a binary $n \times n$ matrix G, usually huge (at the moment Google is said to use $n \approx 4.3 \cdot 10^9$) and very sparse. Each page is characterized by two values: in-degree $c_j = \sum_i g_{ij}$ and out-degree $r_i = \sum_j g_{ij}$.

The model describes a motion of a random user, following with probability p random links from each page he visits, and with probability $1 - p$ jumping to a different address, ignoring existing links.

The matrix A, defined as follows:

$$a_{ij} = pg_{ij}/c_j + (1 - p)/n \tag{1}$$

is a probability matrix of a Markov chain. The sums of all columns are equal to 1. According to the Perron-Frobenius theorem the largest eigenvalue of this matrix is equal to one. The respective eigenvector is the solution to the equation $x = Ax$. The elements of this vector, normalized so that $\sum_i x_i = 1$, are the steady-state probability distribution of the Markov chain and are interpreted as PageRank of respective pages.

The simplest (and scalable) algorithm of solving this problem is the iteration:

$$x_{k+1} = Ax_k. \tag{2}$$

The algorithm is partially asynchronously convergent, see e.g. [12].

3.2 Implementation

The algorithm has been implemented in a simple way, without excessive optimization. The matrix A can potentially be huge, so it is always stored in a sparse form. The program is written using MPI, according to the SPMD (Single Program Multiple Data) rule. The matrix A is split between processes, each computes a different subvector of x. To minimize communication penalties and to utilize the asynchronous properties of the algorithm, the multiplication of a subvector in each process is repeated until a local solution is found, only then messages with results are exchanged and the next iteration begins.

To get the maximum gain from MOSIX process migration, our program declares itself as a cpu-intensive task. Without it, the migration takes a bit more time – the statistics gathered during the loading of data from disk, discourage migration, as the process is identified as an I/O-oriented task. Processes collide on their home machine for several seconds in this case, until the old statistics decay and new ones are collected.

3.3 Test Results

During the tests the PageRank algorithm was run for 25000 pages and for 1000 pages, in different configurations, with the same data (matrix A). The larger task was well suited for distribution, the smaller one was too short – a stress test of the MPI's remote job management and communication procedures.

On a single machine and without decomposition (no communication, local and global convergence are the same) the larger task was done in 8 minutes. The time can be further reduced to 6 minutes and 8 seconds by removing all MPI-related code and, unnecessary in this case, the final iteration (verifying convergence). Decomposition introduced the need for communication and multiple iterations, slowing the computation down to 17 minutes, more if more processes were created. As we were testing the grid infrastructure, not the effectiveness of the decomposition itself, this is the reference time for calculating gains in parallel configurations.

To make use of the network, we connected two machines with MPI, but without activating MOSIX on them. We ran two tests, one with 2 processes, 1 per machine, and the other with 9 processes, our planned decomposition level for full cluster. The results for 9 processes are good, but definitely worse than without decomposition.

Finally, we activated the MOSIX clusters. The computation was done in under 5 minutes. The 9 machines not only ran the program much faster than one machine, but decomposition actually proved useful – the cluster was much faster than a single machine with optimized, non-distributed code. At the same time the cluster's total memory potentially allows solving even larger tasks.

All results are summarized in Table 1.

We also tested turning individual machines on and off (of course except the two nodes using MPI). As long as the system was closed properly, the calculation was not seriously hampered by this.

Task size	25000 pages	1000 pages
A single machine, 1 process	8:00	12.7
A single machine, 2 processes	17:00	14.2
A single machine, 9 processes	20:43	40.0
A single machine, 18 processes	26:54	1:22.6
2 machines via MPI, 2 processes	8:01	13.3
2 machines via MPI, 9 processes	11:45	28.8
2 machines via MPI, 18 processes	12:40	46.2
9 machines total, 2 via MPI, the rest share the load via MOSIX. 9 processes	4:42	32.0
Same as above, 18 processes	5:52	56.9

Table 1: Times (minutes:seconds) required to solve an example problem on different configurations. The same problem can be solved without decomposition or MPI in 6 minutes 8 seconds (25000 pages) and just under 2 seconds (1000 pages).

4 Conclusions

The approach presented in this paper works well. The combination of MPI working on few dedicated machines, physically isolated from users, and MOSIX spreading the load to other stations, connected only when available, allows for very comfortable and fast computation and utilization of computing reserves. Only a relatively small gain was achieved in comparison with the non-distributed algorithm, but the distributed version showed huge increase in speed with parallelization. The weak point is obviously the decomposition method, not the grid stucture.

Spreading tasks from one or two nodes to the rest of the cluster by using MOSIX process migration mechanisms is not optimal. It would theoretically be better to use MPI to spread the tasks to all machines from the beginning, and only let MOSIX manage the loads if tasks differ in complexity. Our setup has the advantage of being far more elastic – all machines in the cluster except the "access point", where MPI is installed, can be detached and reattached during computation, for example during courses. The cost of nonoptimal initial dislocation of processes is negligible for large tasks.

Neither MOSIX nor MPI is a high reliability environment. If a computer crashes for any reason, or is simply turned off without closing the system correctly, the processes running on it will be lost. The design encourages detaching computers from the grid before classes, but a danger still exists. Therefore, programs for the grid should be written for reliability and fault tolerance, if they are expected to run excessively long. Using advanced capabilities of the Globus Toolkit is advised.

The use of Globus as an MPI device to connect the clusters is a good choice from the security standpoint and allows for better administration of resources, but the penalties are noticeable. Using Globus inside a single, isolated cluster would be a waste of resources. It is only needed, when several clusters with different user names, policies and system setup are connected into a grid. A good way of dealing with the slowness of Globus in our example might be to use an MPI flavor of Globus (a version of Globus using an implementation of MPI for local passing of messages). This would allow faster local (in-cluster) communication. Unfortunately we couldn't achieve such a setup due to a lack of a suitable MPI implementation

(non-MPICH based and not LAM).

The MOSIX clusters should be relatively small – up to about 10 machines. There are two reasons for it. First, while MOSIX scales well, with size of the cluster the amount of communication necessary for intelligent migration, grows. More importantly, however, when the cluster is deactivated, all the tasks reside on the access station and slowly continue running. With the processes started by Globus it might be enough to almost freeze the machine, leading to many problems. The number of processes on one cluster should be kept small. In large laboratories there might be a need for multiple access stations.

The grid built this way is also very easy to use. The user only needs to know that he should use the `grid-proxy-init` command to login and `grid-proxy-destroy` to logout. That, and familiarity with MPI, especially the MPICH implementation, suffices. The system behaves mostly like a simple cluster.

The structure shown in this paper is easy to set up. It can be suggested, wherever a large MPI computation is necessary and several smaller clusters can be connected. Because MOSIX offers dynamic load balancing, such a setup handles tasks with wildly varying sizes remarkably well, much better than typical MPI environments, where tasks are statically allocated to machines. The tasks cannot be migrated between clusters (by MOSIX), so the initial distribution is still important.

Summing up, we have shown how Globus and MOSIX can be combined to create an elastic and efficient grid for MPI calculations. We have shown, that it can indeed speed up large calculations, while using it for smaller tasks is a mistake – the costs of spawning remote processes are too high.

5 Acknowledgement

The research presented in this paper was partially supported by the Foundation for Polish Science.

References

[1] M. Litzkow, M. Livny, M. Mutka, "Condor – A Hunter of Idle Workstations", Proc. 8-th International Conference on Distributed Computing Systems", pp. 104-111, June 1988

[2] The Condor project homepage, `http://www.cs.wisc.edu/condor`

[3] A. Barak, O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing", Journal of Future Generation Computer Systems, Vol. 13, No. 4-5, pp.361-372, March 1998

[4] A. Barak, O. La'adan, A. Shiloh, "Scalable Cluster Computing with MOSIX for LINUX", Proc. 5-th Annual Linux Expo, pp. 95-100, Raleigh, May 1999

[5] The MOSIX project homepage, `http://www.mosix.com`

[6] The Globus project homepage, `http://www.globus.org`

[7] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface", Journal of Parallel and Distributed Computing (JPDC), Vol. 63, No. 5, pp. 551-563, May 2003.

[8] The MPICH-G2 project homepage, `http://www.hpclab.niu.edu/mpi/`

[9] L. Page, S. Brin, R. Motwani, T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web", `http://dbpubs.stanford.edu/pub/1999-66`

[10] A. Arasu, J. Novak, A. Tomkins, J. Tomlin, "PageRank Computation and the Structure of the Web: Experiments and Algorithms", World Wide Web 2002 Conf., `http://www2002.org/CDROM/poster/173.pdf`

[11] C. Moler, "The World's Largest Matrix Computation", MathWorks Company Newsletter, `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/oct02_cleve.shtml`

[12] D. P. Bertsekas, J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*, PrenticeHall, Englewood Cliffs, NJ, 1989