OpenMosix for the creation of low-latency, high-efficiency game clusters

Carlo Daffara,
Conecta srl cdaffara@conecta.it

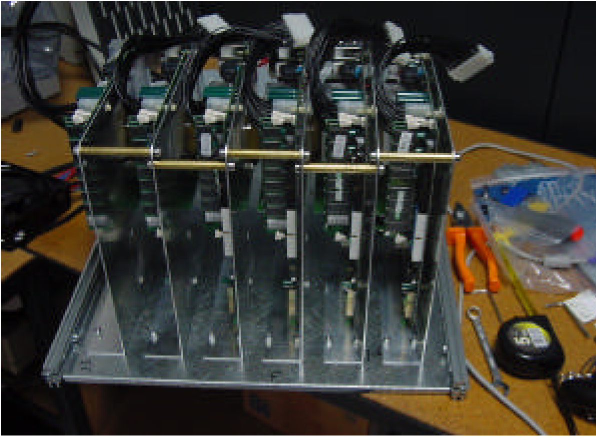
In the variegated world of software there are many niche markets of significant value. One of this niches is online gaming, exemplified by blockbuster games like ID software's Quake III or Sony's EverQuest; popular online gaming sites report more than 10 million players, on 600000 servers, while EverQuest has 433000 paying gamers (for a total of 5 Million dollars/month in revenues).

These games are structured in a strictly client-server style, where the client is the game installed on the player's PC or console, and the server maintains the status, the interaction of players with the environment and the unraveling of the game story. Traditionally, the client/server interaction is maintained using many small UDP packets; most server systems use differential prediction techniques and latency-hiding systems to preserve a smooth playing even in presence of network congestion, packet loss and uneven traffic loads. For a high quality experience, the maximum total ping (round trip) time must be below 150 msec, and the game server must gracefully degrade performance under high load.

The extreme, real-time like requirements and the wide variation in the number of players force the carriers that want to provide this service to host a large number of distinct machines, each hosting a few players per tournament. All the major providers host from 200 to 500 servers, with a players/server ratio of 17. This is a considerable burden, as most of the servers are chronically underutilized, and the large number of servers poses significant maintenance and hardware costs, that makes most of these ventures unfeasible from an economical point of view.

As a system integrator, we got the contract to build a prototype of a high-efficiency (in terms of player/server density) and high-quality (in terms of perceived user experience) game server, designed to host around 1000 concurrent players. We already have experience of Mosix and OpenMosix in the realization of a medium scale video broadcasting system, and so we started development of a custom hardware/software combination that can easily be deployed to telecom carriers.

Our final hardware design consists of 6 Athlon XP 1.8 Ghz systems, built from standard off-the-shelf motherboards. Each board has 768Mb of ram, a dual 100Mbit ethernet connection for performing channel bonding (in a newer version of the cluster we substituted them with single channel copper gigabit boards) and boots through the PXE network boot protocol; the system will be sold in the beginning of 2003 as a universal “tile” for assembling OpenMosix clusters.


The packing system

..and partially assembled cluster

The master node has twin IDE disks in software mirroring mode, under a custom linux distribution originally created for high availability servers. Since some tasks use shared memory and thus cannot be transparently distributed, we replicated most of the distribution to every node's NFS-mounted root. These tasks are then manually started on a user-assigned remote node, and an iptables-based proxy forwards the packets in and out of the master server (the only one with a real IP address). Large files are not copied, but replicated through hard symlinks to avoid unnecessary disk waste, especially considering that large game maps can easily exceed a gigabyte in size.

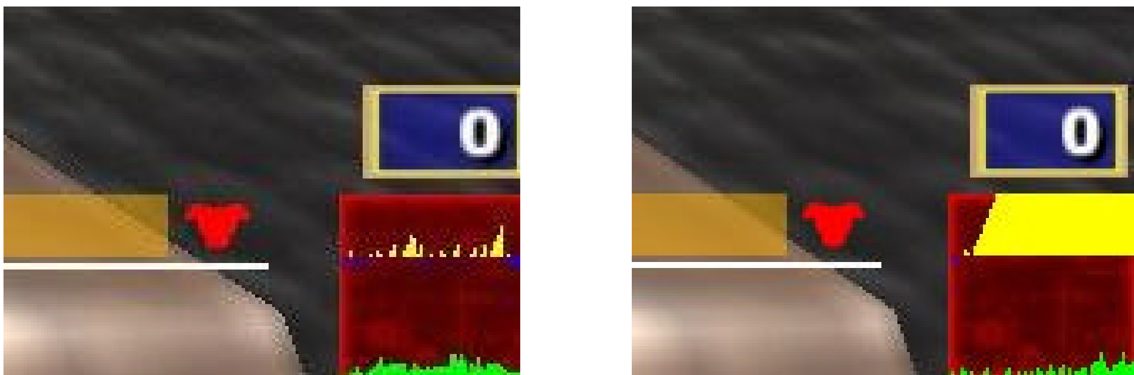
Since the small spacing between the boards, and the high temperature load of the Athlon cpus, we had to implement a simple temperature load monitor for all the boards. To this end, we used the linux lmsensors package, that monitors the onboard temperature and voltage parameters. Each machine has a very simple cron script that execute the sensor check every 5 minutes, and writes the results to a file in the /tmp; thus, on the master node, all temperatures can be checked from the master simply by looking into the /tftpboot/<client_ip>/tmp/temp-out.

Another problem appeared during testing: since the game server memory footprint is large (around 80 Mbytes each), we discovered that the migration of processes slowed down the remaining network activity, introducing significant packet latency (especially perceptible, since packets are very small). So, we used the linux iproute2 queue controls to establish a stochastic fair queuing discipline to the ethernet channels used for internode communications; this works by creating a set of network “bins” that host the individual network flows, marked using hashes generated from the originating and destination IP addresses and the other part of the traffic header. The individual bins are then emptied in round robin, thus prioritizing small packets over large transfer and not penalizing large transfers (like process migration).

The effect of this prioritization process can be seen in the following screenshots of in-game play (with the included network latency display):

Ingame screenshots: actual game (top), with stochastic fairness (bottom left), without SFQ (bottom right)

The top indicator is the server delay in answering to client packets; the bottom indicator is the client ping time (round trip time).
It is quite clear that the packet discipline is considerably improving responsiveness even in presence of substantial internode traffic; this is useful for other realtime-like applications, like video on demand, streaming and large scale instant messaging applications.



References

[PW2001] LaPointe D., Winslow J. "Analyzing and Simulating Network Game Traffic", Worchester Polytechnic Institute, Computer science department, MQP MLC-NG01, available at http://www.cs.wpi.edu/~claypool/mqp/net-game/

[Feng2002] Feng W., Chang F., Walpole J., "Provisioning On-line Games: A Traffic Analysis of a Busy Counter-Strike Server", In Proceedings of the Internet Measurement Workshop, November 2002.

Use of openMosix
for parallel I/O balancing
on storage in Linux cluster

Gianluca Argentini
Information & Communication Technology Department, Riello Group, Legnago (Verona) – Italy
gianluca.argentini@riellogroup.com

Introduction and definition of the discussed question

The manifacturing concern Riello Group has centre at Legnago (Verona), produces domestic and industrial burners and markets conditioning equipments. The *Information & Communication Technology* department attends to computing procedures and infrastructures. The elaboration environment is distributed either geographically, being localized at three principle computation centres, or in the matter of hardware resources.

There are about 20 Unix servers of middle-high level and about 60 NT servers, and those used for the administration of the databases for ERP software and business Datawarehouse are multiprocessor with dedicated storage on each single server. The servers's total capacity of space on device is about 2 TeraBytes, distributed on hardware systems which are heterogeneous as technical features and available volume. This fact implies some troubles for the backup services, mainly implemented by three tape libreries, each with only one head for writing, and hence there are problems of performances and of serial queuing of the relative procedures running during the night. Furthermore in some servers a lot of programs, onerous in required time, in CPU waste and in I/O band on storage, necessaries for the administration of the databases and for the night population of the datawarehouse, run during the same temporal window. This fact and the increase of online data imply a progressive reduction of the temporal windows useful for the backup procedures. To find a remedy for this situation, even in a partial manner, some methods of defered startups by the Unix utility *cron* have been implemented for the working night-procedures with the aim of running one process for one processor, but the results have been modest.

For a better strategy, the ICT manager staff is planning for the triennium 2003-2005 a project of storage consolidation, with the aim of centralize the administration of device-space by a SAN for the Unix and NT servers; a project of server consolidation for the requirements of Erp and datawarehouse, and a pilot-project of a cluster Linux. In particular this latter resource will be used for the study and tests of parallel procedures either for business sphere (HA, high availability) as for technical and scientific computations (HPC, high performances computing) for fluidodynamical simulations. Furthermore, in consideration of some critical situations occurred the last summer 2002 as consequence of violent meteorological fenomena, the Linux cluster could be used for implementing local weather forecast with small temporal limits.

The Linux cluster for tests

For reducing the amplitude of the temporal windows devoted to night serial backups, we have planned a project to verify the possibility and the efficiency of rescue copies executed in parallel mode on device for db datafiles. We preferred this logic and not much expensive method

to hardware implementations, less flexible and usually honerous. We remember that the aim was a study of fattibility, not the research of the best performances.

It has been built a small cluster with commodity components: 4 pc monoprocessor, three with Pentium II and one with Pentium III, respectively of 440 and 800 MHz, equiped in medium with a Ram of 256 MB. The computers have been connected to a private 100 Mb network, together with a NAS-like storage system, constituted by 4 disks for a total of about 120 GB; the storage system has been connected to the network by mean of two access channels.

The storage system has been configured for the pubblic sharing, by an NFS-like protocol, of two file systems of equal size.

On the nodes of the cluster it has been installed the operating system Linux, in the RedHat distribution release 7.2; the Oracle 8.1.6 engine and the two file systems exported by the NAS have been mounted at the point /dati and /copie. In every computer it has been built a small database with 2 datafiles of 2 GB, 2 of 1.5, 2 of 1 and 2 of 0.5, with the purpose of doing tests by mean of various-sized files. The tests have been organized for the simultaneous startup on the nodes of cluster for the copies of the files of same dimension, and on the contrary serializing on the same node the files copies for the disequal sizes. At the aim of simulating the load conditions, due to the running of the programs of db management and datawarehouse population, of our present business servers during the night, on the Pentium III node it has been scheduled the startup of some programs of elaborative weight, to be in execution simultaneously to the datafiles copy processes. This further requirement has induced us to consider the use of a software for balancing the total load of cluster, and we decided to test the *openMosix* package ([6]).

First results for copy processes without openMosix

As first tests, copy processes implemented by the system command *cp* have been used. On the Pentium III node simultaneously have been launched the simulation programs, which ran for about 1h 30m. The I/O registered with this method has been of about 20 minutes for each GigaByte on the Pentium II nodes, hence for a total copy time of about 3h 20m for the 10 GB of the database of each single nodes, and of about 30m per GB on the Pentium III, for a total time of about 5h.

The result obtained with these first experiments has been a small degree of parallelism, for the reason that each process has copied only its data, and hence we have had not just a parallelism but a simultaneity of events, and a small balancing of the cluster load.

Using openMosix

Hence we have used the rel. 2.4.16 of openMosix to test the possibility of realize a better load balancing. To each copy process, simply implemented by the system command *cp* too, it has been associated a single datafile. The use of a single shared storage, external to all the nodes, has permitted to avoid the typical problems of cache consistency which rise up using NFS in the case of process migration from one node to another ([3]). The registered times in the presence of the copy programs alone have been of about 13 – 14 minutes per GB. Further some processes, at their startup, have been migrated by openMosix towards the Pentium III node, and this in particular after the copy of higher size datafiles in the Pentium II nodes, hence at a moment when the machine-load was considered even of big entity by the scheduling algorithm of openMosix system ([2]). The total load of the cluster has resulted well balanced.

Subsequently the experiment has been repeated with the simulation programs running on the major node. In this case it has been observed migration of copy processes from this node towards those less power, evidently considered by openMosix in the right condition for to balance the total load of cluster. Effectively the balance has been good, but the further load on the Pentium

II nodes has degraded their performances, and the medium copy time for one GB is resulted of about 25 – 30 minutes. The Tab.1 offers a comparison of the first two experiments done on the use of openMosix.

	No openMosix	*With openMosix*
Pentium II total time per node	3h 20m	4h 30m
Pentium III total time	5h	4h
Balancing	small	good

Tab. 1
A comparison of the copy procedures with and without

From the table the following first conclusions can be derived:

- openMosix has reached the balancing of cluster nodes, that is the primary aim for which it has been projected;
- the functionality of the user-tools associated to the package, *mtop* for the load of a single node, *mps* for the control of a single active process, *mosmon* for the global monitoring of the cluster, has been good;
- the use of this package has implied in the described situation a saving, even if modest, of the time of total elaboration of copy processes;
- without the use of suitable file systems as MFS ([1], [2]), openMosix seems to privilege the balance of the load of computational activity on CPUs respect to that of I/O towards the storage system; this fact is proved by the migration of the copy processes from the Pentium III node, very CPUs-loaded by simulation programs, towards the other nodes, hence the saved hour of total elaboration on the powerful node has been lost as longer activity on Pentium II nodes.

Using MPI

For a better comprehension of the performances offered by openMosix, we have realized a copy procedure of datafiles by mean of the parallelization library *MPI*, Message Passing Interface, using the Argonne National Laboratories MPICH implementation, rel. 1.2.4 ([5]). For this aim it has been found as optimal technique for obtaining good performances the use of multiprocess views, that is the possibility offered by some specific functions of the package MPI-2 of reading and writing a fixed portion of a single file for each process (see Fig.1, [4]). The copy procedure has been wrote in C language and compiled by mean of GNU C rel. 2.3 .

A remarkable characteristic of this library is the possibility of using a buffer for reading or writing, and after some tests it has been chosen one of 25 MB size for each node. A bigger value could be better for a faster I/O but it could cause overflow of nodes memory.

Some tests have been conducted with 1, 2 and 4 processes for each datafiles, registering the copy times even for every single files size. It has not made use of 3 processes for avoiding problems of no perfect divisibility of the file size by multiples other then those of 2.

Fig. 1 *A datafile segment with views for each process, identified by its own rank (ID code in MPI). The process number 4 is shown with a view of size not equal to the other because of the possibility of a non integer division by 4 between the datafile size, the buffer size, and the number of segments necessary for the covering of the entire file.*

Using only the MPI library and not openMosix, the medium time for the copy of 1 GB on Pentium II nodes has been of 13 – 15 minutes, and of 15 –17 on Pentium III where ran simultaneously the simulation programs too. The complete load of the cluster has been satisfactory even if not optimized. Using instead openMosix too, some migrations of copy processes from Pentium III to the other nodes has been risen up; this fact seems to confirm the CPU-oriented nature of the internal algorithms of openMosix, which tends to balance with priority the computational load on processors. During the temporal period of processes migration, the cluster general load has been very well balanced, but the Pentium II nodes, which were heavier respect to the previous case, have registered an increment of about 25% of their local copy time. Instead the total backup time on the cluster is remained almost the same. The following Tab.2 offers a summary of the executed experiments:

	Only openMosix	*Only MPI*	*MPI-openMosix*
Pentium II total time	4h 30m	2h 30m	3h
Pentium III total time	4h	2h 50m	2h 30m
Balancing	good	sufficient	very good

Tab. 2 *Comparison of the copy procedures and simulation programs times*

Comparison tests between openMosix and MPI

To obtain some results on the degree of parallelism and on the balancing of the load by mean of a comparison between openMosix and MPI, has been executed three copy experiments with a node datafiles set of about 8 GB, and respectively with size of 0.4 GB (20 files), 1 (8), 1.4 (6), 2 (4). On every node have been run, with openMosix started (*openmosix start*), at first *cp* copy processes simultaneously in number of 1, 2 and 4 on three successive experiments, and then on the cluster, with openMosix stopped (*openmosix stop*), have been launched copy procedures managed by MPI with 1, 2 and 4 processes per node on other three experiments.

We have not used on the Pentium III the simulation programs for not to dirty the test making the results difficult to analyze. In this manner on the cluster at each instant was running, in the two situations, the same number of processes, which have acted on every node on the same data quantity. Hence the two analized situations have been based on two distinct philosophies: with openMosix we attempted to parallelize the total procedure on the cluster associating to each process one file for the copy, with MPI associating to each file one multiprocess program. The reached conclusions, graphically shown in Fig.2, can be so summarized:

- openMosix has registered a less copy time in the case of one process per node and in the multiprocess case for small size files;
- openMosix has shown a performances degrade increasing respect the used processes number, demonstrating that the load of the autonomous concurrent processes on the resources of the single nodes has had a negative weight bigger than the obtained parallelism;
- for MPI the use of its native parallelism and the possibility of buffers management have shown an increase of performances, even if with decreasing influence, respect the number of used processes.

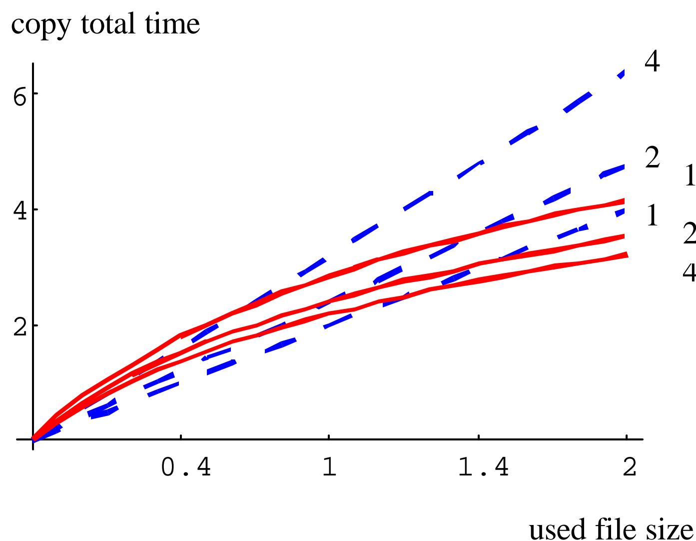


Fig. 2
Comparison between the copy times of openMosix (dashed lines) and MPI. The numbers beside the lines correspond to the processes used for obtain the graphic. The growth of MPI temporal graphics is resulted to be logarithmic respect the used files size.

Final considerations

The used cluster Linux has had evident limits: diversity of the nodes hardware characteristics; the nodes itself were only monoprocessor; the original files to be read and those to be written were localized on the same disks set; it has not been possible to conduct tests with files of size bigger than 2 GB, for an intrinsic limit of the used storage system; it has not been possible to use the file system MFS, Mosix File System, which in analogous studies has provided very good results with openMosix ([1]). From the executed tests the following considerations can be extracted:

- openMosix has evidenced correct and good functionalities of processes balancing;
- its use with MPI has not reduced the cluster global performances of the library;
- it has shown inferior results compared to MPI in the case of multiprocessor procedures parallelizing the I/O of big size files.

Acknowledgments

The author wish to thank *Stefano Martinelli* and *Carlo Cavazzoni* of CINECA for the

invitation to present this work and for useful explanations on the use of functionalities of MPI-2; *Giorgio Colonna*, Riello Group ICT Department manager, and *Guido Berardo*, ICT Projects Planning manager, for to allow me the studies on HPC and Linux Clustering; *Ernesto Montagnoli*, ICT Infrastructures manager; *Claudio Gastaldo*, *Giuliano Nava* and *Mirco Trentin*, ICT Unix administrators, for their useful contribution of ideas on Linux.

References

[1] *Roberto Aringhieri, Open source solutions for optimization on Linux clusters, University of Modena and Reggio Emilia, DISMI, Technical Report 23/2002;*

[2] *Moshe Bar, openMosix Internals, article in openMosix Web site;*

[3] *Moshe Bar, Oracle RAC on Linux, BYTE, July 2002;*

[4] *William Gropp – Ewing Lusk, Advanced topics in MPI programming, in T. Sterling, Beowulf cluster computing with Linux, 2002, MIT Press;*

[5] *www.mcs.anl.gov/mpich, Web site for MPI libraries;*

[6] *www.openmosix.sourceforge.net, Web site for openMosix.*

(Open)Mosix Experience in Naples

Rosario Esposito1, Francesco M. Taurino1,2, Gennaro Tortone1
1 (Istituto Nazionale di Fisica Nucleare, via Cintia – 80126 Napoli, Italy)
2 (Istituto Nazionale di Fisica Nucleare – UdR di Napoli, via Cintia – 80126 Napoli, Italy)

This document is a short report of the activities carried out by the computing centre of INFN, INFM and Dept. of Physics at University of Naples "Federico II", concerning the installation, management and usage of HPC Linux clusters running (open)Mosix [1][2].
These activities started on small clusters, running a local copy of the operating system and evolved through the years to a configuration of computing farms based on a diskless architecture, simplifying installation and management tasks.

Introduction

Thanks to high performances offered by low cost systems (common personal computers) and availability of operating systems like Linux, we were able to implement HPC clusters using standard PCs, connected to an efficient LAN.
This solution offers several advantages:

- lower costs than monolithic systems;
- common technologies and components available trough the retailers (COTS, Components Off The Shelf);
- high scalability.

Mosix in Naples

Our first test cluster was configured in January 2000.
At that time we had 10 PCs running Linux Mandrake 6.1, acting as public Xterminals.
They were used by our students to open Xsessions on a DEC-Unix AlphaServer to compile and execute simple simulation programs.
Those machines had the following hardware configuration:
- Pentium 200 Mhz
- 64 MB RAM
- 4 GB hard disk

Each computer, with the operating system installed on the local disk, was connected to a 100 Mb/s Fast Ethernet switch.
We tried to turn this "Xterminals" in something more useful. Mosix 0.97.3 and kernel 2.2.14 were used to convert those PCs in a small cluster to perform some data-reduction tests (mp3 compression with *bladeenc*[1] program).
Compressing a wav file in mp3 format using *bladeenc* could take up to 10 minutes on a Pentium

[1] *http://bladeenc.mp3.no/*

200. Using the Mosix cluster, without any source code modification, we were able to compress a ripped audio cd (14-16 songs) in no more than 20 minutes.

Once verified the ability of Mosix to reduce the execution time of those "toy" programs thanks to preemptive process migration and dynamic load balancing, we decided to implement a bigger cluster to offer a high performance facility to our scientific community.
The first step was to allow our students to directly log on the cluster to submit their jobs. In this way the machines started to be used as computing nodes rather than simple Xterminals.

Majorana

The students' Linux farm was a success, then we decided to build a more powerful Mosix cluster, named "Majorana", available to all of our users, in line with the nowadays trends in operating systems and development platforms (Linux + X86), using low cost solutions and opensource tools (MOSIX, MPI, PVM).

The farm was composed by 6 machines.
We bought the following hardware:
5 computing elements with:

- Abit VP6 motherboard
- 2 Pentium III @800 Mhz
- 512 MB RAM PC133
- a 100 Mb/s network card (3com 3C905C)

1 server with:

- Asus CUR-DLS motherboard
- 2 Pentium III @800 Mhz
- 512 MB RAM PC133
- 4 IDE HD (os + home directories in RAID)
- a 100 Mb/s network card (3com 3C905C) - public LAN
- a 1000 Mb/s network card (Netgear GA620T) - private LAN

All of the nodes were connected to a Netgear switch equipped with 8 Fast Ethernet ports and a Gigabit port dedicated to the server.

Configuring a farm presents some difficulties due mainly to the high number of hardware elements to manage and control.
Farm setup can be a time-consuming and difficult task and the probability to make mistakes during the configuration process gets higher when the system manager has to install a large number of nodes.

Moreover, installing a new software package, upgrading a library, or even removing a program, become heavy tasks when they have to be replicated on every computing node.
A farm, at last, can be a complex "instrument" to use, since executing parallel jobs often requires an "a priori" knowledge about hosts status and a manual resources allocation.

The software architecture of the clusters we have implemented tries to solve such problems, offering an easier way to install, manage and share resources.
The model we have chosen is based on diskless nodes, using no operating system disk. One of the machines (master node) exports to all the other nodes the operating system and the applications.

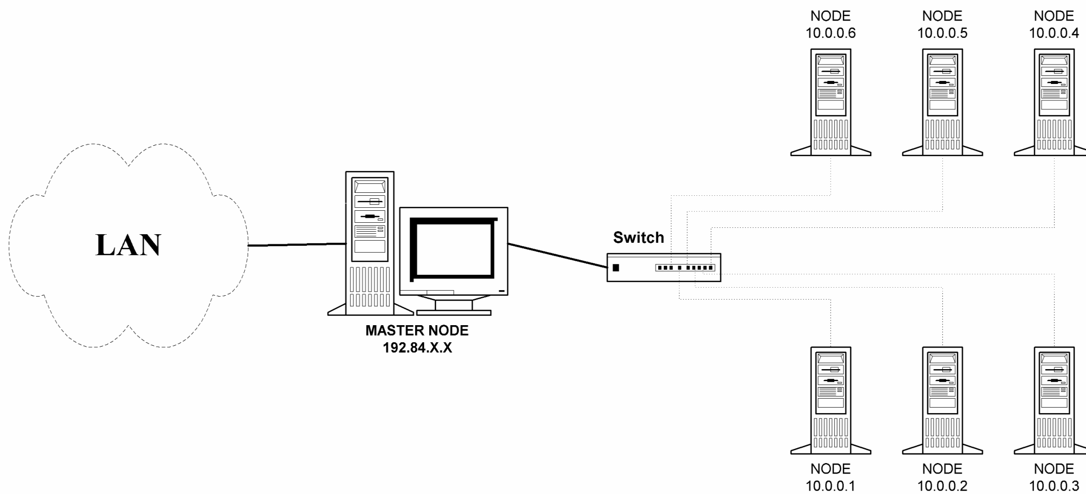


Figure 1: ***Majorana* farm configuration**

A private network is used by slave nodes to remotely execute the bootstrap procedure and to exchange data with the master node (NFS, process migration, MPI).
Each machine, except the master node, has no system disk. This solution offers several advantages:

- installing and upgrading software on the farm gets very simple as every modification involves only the master node;
- increased reliability, because there are less mechanical components;
- reduced cost, because only the master node mounts a local hard disk;

This type of setup significantly reduces the management time for “slave” nodes.

Network boot of diskless nodes is achieved using EtherBoot[2] [3], an opensource tool that allows to create a ROM containing a startup code, depending on the Ethernet card installed on the diskless machines. This code can be loaded on a boot device, such as a floppy, a hard disk, or a network adapter EERPOM.

We can shortly explain the boot sequence of a generic diskless node:

1. startup code, generated by EtherBoot, is loaded and executed at boot time
2. the node sends on the network (via bootp or dhcp) a request to obtain an IP address from the server
3. once obtained the IP address, the node downloads its MOSIX kernel from the server, using TFTP protocol
4. the MOSIX kernel is loaded and the node begins the real operating system boot
5. kernel, previously compiled with the *“root filesystem over NFS”* option, mounts its root filesystem via NFS from the server
6. once mounted the root filesystem, the node completes its boot sequence initialising system services and, eventually, mounting local disks

[2] *http://www.etherboot.org*

Since every diskless node has to mount its root filesystem via NFS, master node should export an independent root filesystem image to each client.
Exporting the same root filesystem to every client is not convenient for obvious reasons, for example configurations and log files have to be different for each machine.

To solve this problem we chose to install ClusterNFS[3] [4], an enhanced version of standard NFS server.
ClusterNFS is a an opensource tool which allows every machine in the cluster, (included the master node), to share the same root filesystem, paying attention to some syntax rules:

- all files are shared by default;
- an "xxx" file common to every client, but not to the server, has to be named "xxx$$CLIENT$$";
- an "xxx" file, for a specific client has to be named "xxx$$HOST=hostname$$" or "xxx$$IP=111.222.333.444.", where "hostname" and "111.222.333.444" are the real hostname or the ip address of that node.

This diskless cluster, currently running OpenMOSIX 2.4.19, is used by our users (mainly students and researchers) as a test platform to develop scientific applications and to submit cpu-intensive jobs, such as Monte Carlo simulations.

VIRGO

VIRGO [5] is the collaboration between Italian and French research teams, for the realization of an interferometric gravitational wave detector.
The Virgo project consists mainly in a Michelson laser interferometer made of two orthogonal arms being each 3 kilometres long. Multiple reflections between mirrors located at the extremities of each arm extend the effective optical length of each arm up to 120 kilometres. Virgo is located at Cascina, near Pisa on the Arno plain.
The main goal of the VIRGO project is the first direct detection of gravitational waves emitted by astrophysical sources.

Interferoretric gravitational wave detectors produce a large amount of "raw" data that require a significant computing power to be analysed.
Moreover a large number of computing resources has to be allocated to analyse simulated data; this is a very important task to develop new efficient analysis procedures that will be used on the real data.

To satisfy such a strong requirement of computing power we decided to build a Linux cluster running MOSIX.
The prototype of this Linux-farm [6][7][8] is located in Naples and consits in 12 computers (SuperMicro 6010H). Each machine contains two 1Ghz Pentium III processors, 512Mby RAM, 2 fast ethernet adapters and one 18Gbyte SCSI disk.
One of the machines, acting as a "master" node, distributes via a private network the MOSIX kernel and the operating system to the other machines, using the same diskless architecture shown in section 3.
Client nodes can be considered as diskless machines; their local disk is only used as a "scratch" area.
Each node is connected to a second private network to access applications and data, stored on a 144Gbyte SCSI disk array, attached to an Alphaserver 4100 (dual 500Mhz alpha processors + 256Mb RAM) and exported via standard NFS.
The following picture shows the entire architecture:

[3] *http://clusternfs.sourceforge.net*

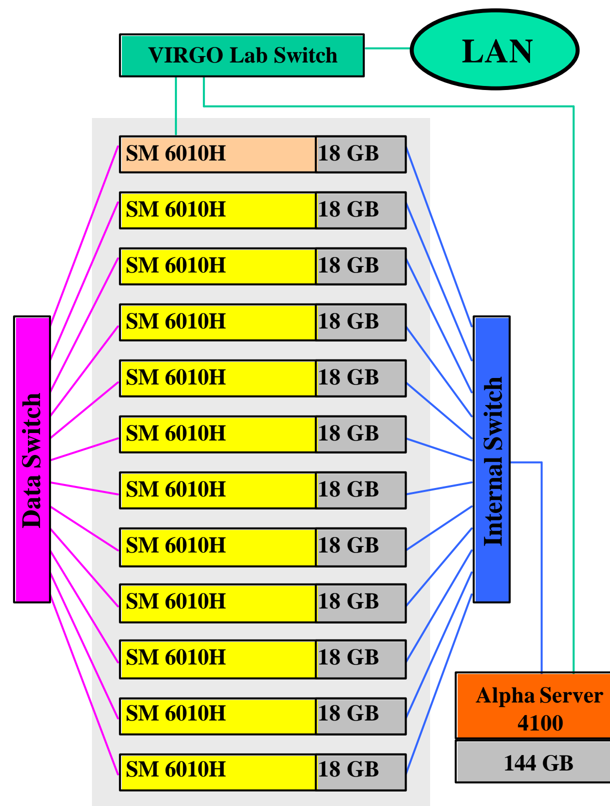


*Figure 2: **VIRGO farm architecture***

Each node currently runs Linux RedHat 7.2 and OpenMOSIX 2.4.17, while the AlphaServer runs Compaq Tru64 (5.0a).
Most of the applications running on the cluster use some math libraries such as Numerical Recipes and Grasp, a library for gravitational signal processing.
Some applications use parallel implementation of filtering algorithms for gravitational signals based on MPI.

One of the main purposes of the Linux farm in Naples is the detection of a particular class of gravitational waves; those emitted by coalescing binaries.
This type of analysis is performed using specific libraries, written by the Virgo researchers, and some public domain math tools, such as FFTW and ROOT.
The Linux farm has been strongly tested by executing intensive data analysis procedures, based on the Matched Filter algorithm, one of the best ways to search for known waveforms within a signal affected by background noise.

Matched Filter analysis requires a high computational cost as the method consists in an exhaustive comparison between the source signal and a set of known waveforms, called “templates”, to find possible matches. Using a large number of templates the quality of known signals identification gets better and better but a great amount of floating points operations has to be performed.
Running Matched Filter test procedures on the OpenMOSIX cluster have shown a progressive reduction of execution times, due to a high scalability of the computing nodes and an efficient dynamic load distribution.
Figure 3 shows the measured speed-up of repeated Matched Filter executions:

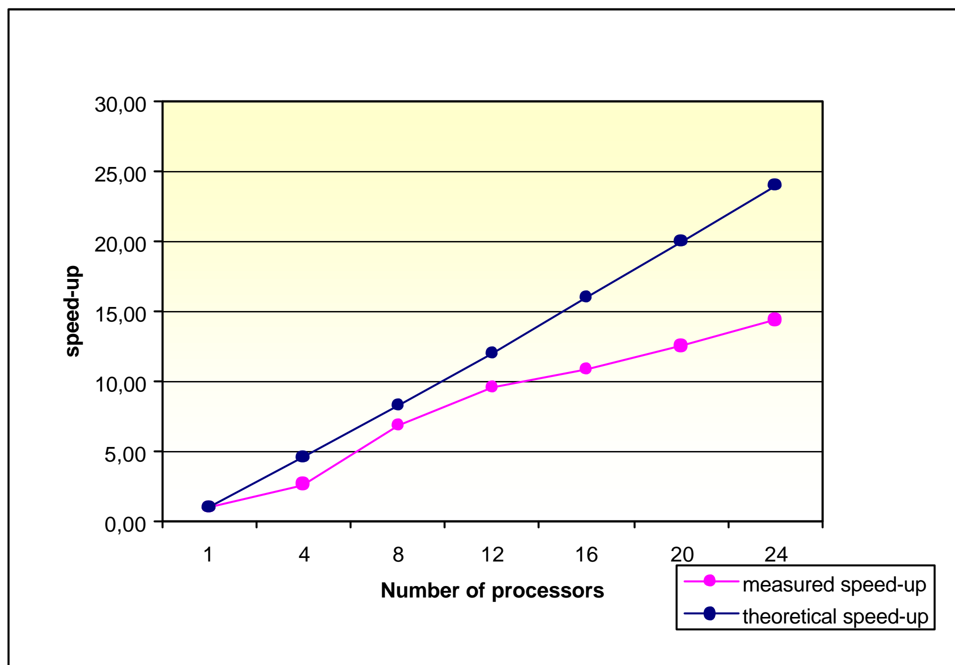


Figure 3: ***Speed-up of parallel execution of Matched Filter program***

The increase of computing speed respect to the number of processors doesn't follow an exactly linear curve; this is mainly due to the growth of communication time, spent by the computing nodes to transmit data over the local area network.

ARGO

The aim of the ARGO-YBJ [9] experiment is to study cosmic rays, mainly cosmic gamma-radiation, at an energy threshold of ~100 GeV, by means of the detection of small size air showers.
This goal will be achieved by operating a full coverage array in the Yangbajing Laboratory (Tibet, P.R. China) at 4300 m a.s.l. .

As we have seen for the Virgo experiment, the analysis of data produced by Argo requires a significant amount of computing power. To satisfy this requirement we decided to implement an OpenMOSIX cluster.
Currently Argo researchers are using a small Linux farm, located in Naples, constituted by five machines (dual 1Ghz Pentium 3 with a total amount of 4.5 Gbyte RAM) running RedHat 7.2 and Mosix 2.4.13.
The cluster has been set up using the same diskless architecture we have previously illustrated (Etherboot+ClusterNFS).
At this time the Argo OpenMOSIX farm is mainly used to run Monte Carlo simulations using *Corsika*[4], a Fortran application developed to simulate and analyse extensive air showers.
The farm is also used to run other applications such as *GEANT*[5] to simulate the behaviour of the Argo detector.
The OpenMOSIX farm is responding very well to the researchers' computing requirements and we already decided to upgrade the cluster in the near future, adding more computing nodes and starting the analysis of real data produced by Argo.

---

[4] *http://ik1au1.fzk.de/~heck/corsika/*
[5] *http://www.fisica.unile.it/~argo/analysis/argog/*

Conclusions

For our purposes, the most noticeable features of OpenMOSIX are its load-balancing and process migration algorithms, which implies that users don't need to have knowledge of the current state of computing nodes.

Parallel application can be executed by forking many processes, just like in an SMP, where OpenMOSIX continuously attempts to optimise the resource allocation.

The "(Open)MOSIX+EtherBoot+ClusterNFS" approach is one of the best solutions to build powerful computing farms with minimal effort.

In our environment, the choice of (open)Mosix has been proven to be an optimal solution to give a general performance boost on implemented systems.

References

[1] OpenMosix homepage
http://www.openmosix.org
[2] A. Barak, O. La'adan: "The MOSIX Multicomputer Operating System for High Performance Cluster Computing", Journal of Future Generation Computer Systems, Vol. 13, March 1998
http://www.mosix.cs.huji.ac.il/ftps/mosixhpcc.ps.gz
[3] Ken Yap, Markus Gutschke: "Etherboot User Manual", 2002
http://www.etherboot.org/doc/html/userman.html
[4] G. R. Warnes: "Recipe for a diskless MOSIX cluster using ClusterNFS", 2000
http://clusternfs.sourceforge.net/Recipe.pdf
[5] "General overview of the VIRGO Project"
http://www.virgo.infn.it/central.html
[6] F. Barone, L. Milano, R. Esposito et al.: "Evaluation of Mosix-Linux Farm Perforamces in GRID Environment", Proc. of CHEP (International Conference on Computing in High Energy and Nuclear Physics), pag. 702-703, September 2001
[7] F. Barone, L. Milano, R. Esposito et al.: "Mosix Linux Farm Prototype for GW Data Analysis", 6th Gravitational Wave Data Analysis Workshop, December 2001
http://gwdaw2001.science.unitn.it/abstracts/abstractsgwdaw.pdf
[8] F. Barone, L. Milano, R. Esposito et al.: "Preliminary tests on the Napoli farm prototype for coalescing binary analysis", VIR-NOT-NAP-1390-196, March 2002
[9] Argo experiment homepage
http://www1.na.infn.it/wsubnucl/cosm/argo/Argo_index.html

Consideration on OpenMosix

Gian Paolo Ghilardi
Responsabile progetto Openmosix@Crema
D.T.I. dell'Università di Milano, presso il Polo didattico e di ricerca di Crema

This short document is the report of several experiences about the opportunity to use Java and C programs on OpenMosix.

Why I wrote this document.

The purpose of this document is to highlight some experiences about the possibility to create standard programs to use with OpenMosix.

My tests started some week after the publication on the Net of the first stable release of Mosix (1.0). Later I found a FAQ talking about compatibility between Java and the cluster.

So I decided to create small Java programs hoping to succeed distributing the load on four PC that had been assigned to me for my tests.

This document is summary of my experience from that moment .

Targets of my tests.

All the performed tests want to show how to develop a program that can “cohabit” with OpenMosix and, in a particular way, they want to evidence the ability to OpenMosix to distribute the created processes using Java language.
I’ll use C (the king of the languages on Linux) also.

Processes and Threads in the JVM and OpenMosix

OpenMosix can distribute processes from low-resources-nodes to node with a large amount of resources at “favorable price”: in fact, OpenMosix algorithms adopt methods and strategies from the economy.

We must observe that a process, migrated from a node to another one, maintains an “umbilical cord” with the departure node.
Each process (that is a program in execution) has normally two “faces”: an "user space" one and a "kernel space" one. OpenMosix can distribute only the first face therefore the part in "kernel space" remains on the “home” node and answers to the requests coming from the destination node.

Therefore the execution context is "shared" between the home node and that node of remote execution. However it has been noticed that a narrow number of system-calls (we speak about "kernel space") can be executed directly on the destination node because they aren't specific for that node.
This possibility reduces the overhead related to data transfer between the nodes.

However it is possible, in some particular situations, to distribute also thread (light-weight processes).

The Java Virtual Machine ("JVM") is the responsible of thread creation in Java so thread distribution problem of Java code is relative to the JVM implementation.
The official JVM of Sun supports various thread typologies: native threads (JVM uses thread from the Operating System below; "classic" threads (the specific JVM thread library); "green" threads are "special" because Java can be distributed on single OpenMosix only in presence of a JVM those ones.

"Green" are " IO-bound" threads, are strongly tied with the input/output and they are not fundamental for the system life: it means that they are intrinsically predisposed to being migrated.

Processes and Threads in Linux

Linux, like many other Operating Systems, integrated complete management of processes and threads (lightweight process), according to many standards (POSIX, X/Open, Open Group amongst the others). Moreover following a Unix tradition, all the processes come down from other processes with the exclusion of the fundamental process, created at the system start (the "init" process), from which all the others are originated.

Linux supports support either "user space" threads, either " kernel space" ones.

Before starting...

The purpose of the tests is to show how to produce programs distributable on OpenMosix using C, C++ and Java.
We need to find at least an effective method of distribution on OpenMosix: it would clear the path for OpenMosix diffusion and represents the maximum gratification for me.

Test of Java on OpenMosix (bytecoded classes).

Targets of the tests.

The tests want to answer to a request I received: to use a JVM and to show in a practical and immediate way that the Java "bytecode" classes can be distributed without particular changes or optimizations.

The "bytecode" it is "machine-neutral" code, loaded dynamically at runtime from a JVM and not compiled natively for a specific architecture/platform. JVM translate dynamically the bytecode into the comprehensible language of the OS below.
This is Java's philosophy, synthesized in the famous sentence "compile once, run everywhere" (you compile once the code and you execute it everywhere a JVM is available).

Preparation to the tests.

For my tests (started December, the 21st 2001) I've used the following Java Virtual Machines.

- JDK 1.2.X from Sun for Linux on i386 (2001/2002).
- JDK 1.3.X from Sun for Linux on i386 (2001/2002).
- JDK compatible with v1.3.x specifications, available on www.blackdown.org (2001/2002).
- JDK 1.4.X from Sun for Linux on i386 (2002).
- JDK compatible with v1.2.x specifications, "Kaffe" (2001/2002).

At the end of 2001 and the beginnings of 2002 I've created a small program Java, PiBenchmark constituted by 3 only classes (bytecode format).
Such program creates a narrow number of thread each of which it calculates the value of the Pi.

Every thread computes its own Pi, independently.

The algorithm I used is largely known (somebody told me it has been used by various mathematicians, among which Archimedes): departing from a raw value, at every footstep it gets closer to the value of the Pi.

Results?

In that period I was a neophyte with Mosix (at the time it was still a "free" project: OpenMosix didn't exist yet) and I tried my program on the cluster at the University, hopeful.

The result was (and it is still) a failure.

What doesn't it work?

OpenMosix is able to distribute Java threads only in presence of "green" ones, (they are threads tied intrinsically to the I/O and therefore they use do not stress the CPU too much).

We need to underline some things:

This typology of thread is provided from special libraries: OpenMosix doesn't if you create some "common" threads and to do so that for long periods performs some I/O.

Linux (the whole Operating System) doesn't provide intrinsic support to this typology of thread.

We need special libraries not "re-mapping" the used threads on "native" ones (libraries in kernel-space like the LinuxThreads only manage native one).

On OpenMosix site, among the FAQs ("Frequently Asked Question") there is a note talking about the possibility to distribute Java threads using "JVM green-thread compliant" only.

I tested many JVM (official and alternative ones) but I got no positive result because there is no JVM providing green threads.

Conclusion.

My experiments of the use of Java "bytecoded" on OpenMosix you/they can define then him concluded. The result is negative.

Test of Java (compiled classes) on OpenMosix.

Objective of the tests.

Frustrated by the failure had with the JVMs and the relative classes "bytecode" of Java, I have tried the card of the so-called JITs compiler (compilers Just In Time), or rather that family of compilers that compiles the bytecode of Java for a specific platform/architecture.

Obviously compiling the classes "bytecode" means to lose the portability typical of Java but in this case such characteristic doesn't serve (OpenMosix runs only on Linux and, for the time being, only on architecture x86 even if the porting is in progress for the architecture IA64).

Particularly I used the JIT compiler for Java that is part of the "GCC suite" of Gnu.
The compiler in object is the GCJ (Gnu Compiler for Java): it is released under license GPL v2.0 and it is therefore "free."

In particular I have tried two versions of the GCJ.

GCJ included in the Gnu Suite v2.96

GCJ included in the Gnu Suite v3.1.0

I have compiled the classes of PiBenchmark.
Both the versions of the compiler have correctly completed the compilation producing a perfectly working executable.

The problem is that the compilation by itself did not result in the hoped effect.
If we launch two or more instances of the program the CPU is constantly used to a 100% (each instance is, to be true, able to saturate the CPU by itself) but OpenMosix doesn't intervene.

What doesn't work?

The problem has been just moved. Before we had a JVM that loaded the code and executed the threads using an own library of thread either OS based or non- but with the same results.

We now have compiled code, surely faster than that "bytecode", but tightly tied up to the OS… Too tied to the OS.

According to the documentation of the GCJ (and having observed the code of the compiler), it is noticed that every standard class of Java has its own "translation" in a C++ class so that the compiler (in reality it is a front-end for G++, the C++ compiler of GNU) doesn't employ a lot of time to create an executable.

The class java.lang.Thread is translated using PThreads (POSIX threads) and really in this lays the problem.

The standard PThread library remaps in turn its threads on LinuxThreads, that is the “native” Linux threads.

The library of the Linux native threads, as already said before, is not "green-thread-compliant." Such threads are "general purpose", they share memory and resources of the system; they have a common user area and files descriptors with the "parent".

In practical OpenMosix is not able to distribute these so tightly tied up, among themselves, threads (because of the various sharings) and therefore to the node that entertains them.

PThread and OpenMosix

Once failed the experiment with Java, and before knowing the reality about threads in Linux, I have made some tests with the PThreads.
The purpose was to obviously succeed in distributing the code on more nodes.

The granularity of distribution in OpenMosix is to the level of process. The distribution of threads is not guaranteed instead.

At the time of my first tests, believing that the standard PThreads library was totally a in user-space (as such no tightly connected to the kernel and therefore potentially usable to distribute the code), I created some small programs to try this one supposition of mine: the programs that use the PThreads can be distibuted with OpenMosix.

In my tests, once a simple algorithm had been selected (and relative implementation), every thread executed it in an independent way. This implicates that shared variables don't exist.

About one year ago I have completed different tests with the PThreads: every test ended up with negative result.
I didn't know in fact that the PThreads in Linux are created and managed from the LinuxThreads library, directly included in glibc.
This means that compiling with Linux GCC a program that asks for the PThreads, under standard conditions, such program is linked to the LinuxThreads (flag of the GCC "- lpthread").

To be clearer: when a program (compiled with the library glibc) makes the standard POSIX call "pthread_create", the Linux system-call clone () is instead executed.

The use of the LinuxThreads, kernel level thread, obviously involves (to the actual state) the impossibility of distribution on OpenMosix.
Nevertheless we must underline that currently OpenMosix has granularity to level of process and not of single thread.
This means that is that the use of libraries of user-space threads does not automatically guarantee to exploit the power of OpenMosix: both kernel-space and user-space threads are not totally independent. As for its definition a thread shares with the others the resources and this fact sets a serious tie if one tried to separate and distribute the load of job.

One might think about special libraries that "remap" the calls on precesses and not threads (as every “green-thread” compliant JVM should do) in an entirely transparent way to the programmer. Obviously this would cause more than a few problems during the implementation

and would however penalize the performances obtained from these libraries.

The other solution is... to wait! In the roadmap of OpenMosix there is shared memory to level of cluster ("cluster shared memory"). Having the opportunity "to see" and "to manage" the memories of all the nodes as if they were a single memory could allow a granularity level of down to a thread. The threads, not tightly dependent to the system or to the life of the other thread/processes, they could be "unhooked" from a nodeand be "rehooked" on a destination node in a transparent way.

An updated list on the compatible and not compatible software with OpenMosix can be recovered on the site looking for the "wiki pages".

Solutions?

Waiting for the implementation of the “cluster shared memory”, the applications, to correctly work on Openmosix, don't have to resort to threads, both kernel-space and user-space threads. Currently the only practicable road is that of the usage of the system-call fork ().

A first alternative would be to create a small library (similar to that of PERL) that creates and manages processes using the calls typical of threads, maybe conforming to the POSIX drafts.

Reflecting on the differences between processes and threads I have noticed that to level of definition it all comes down to seeing and implementing in different way the concept of "context of execution."

A process (according to A.S.Tanenbaum) it nothing else other than a program in execution "with it’s current values of program counter, registers and variables; conceptually every process has it’s own virtual CPU". This means that a process comes to constitute a context of execution virtually independent from the others.

The threads (we speak of those descending from a process, not the process itself intended as the main thread) "allow a lot of executions in the environment of a process, in wide measure independent the one from the others" (again a definition of Tanenbaum).

To a good insight threads are not so independent among them: the sharing of the resources (common addressing space to the parent process,...) makes difficult to distribute the job on the nodes of a cluster.

Let’s tighten the field...

Let’s say that we have a multithread application in which the threads are able to work without need to communicate among them, without sharing variables and they don't communicate, either among them or with the parent process. In practice these are "launch & forget" threads that, opportunely initialized, are able "to live" and to operate in loneliness and only when in "stings of death" let themselves tell they are alive to leave a "will" (the result of their job).

The concept around eventual shared open files is more complex.
For simplicity we admit that our threads don't write on file, instead the writing is prerogative of the principal thread. Otherwise we can do so that every thread writes on temporary files and only at the end the result is communicated to the principal thread (remember that the threads are “virtually” independent among them).

Acknowledged to be able to realize this model, we would have some threads with a lot of analogies with the processes.

A new model of thread.

From here one computer madness/fantasy of mine is born according to which this new typology of thread (that is very limited/limiting under the profile of use) represents the correct compromise among the independence typical of processes and the performances of threads.

If we now meld this model of thread with that of "popup" (illustrated on the book of Tenenbaum), we can perhaps find a practical system to distribute threads on OpenMosix.

A "popup" thread is a thread that is created on the fly on a node following aincoming message to that machine. Such threads don't have history and each of them "starts fresh and it is identical to all the others."

The model that I have in mind comes to be a distributed system in which every thread of the described type can be "disconnected" (detached) from a node and "connected again" (re-attached) to destination node.

This is just a theoretical proposal!

I perfectly know that this type of solution is of difficult (if not unlikely) realization, nevertheless to theoretical level it would allow a distribution of threads on OpenMosix that doesn't require for cluster shared memory.

Nevertheless the model, opportunely extended, would allow to migrate processes with more than one thread.

The problem is to divide the resources shared from more threads on the node of departure.

The cluster should be able to:

- ask for "division of the common inheritance" (the shared resources) to a thread in whatever moment
- move the part of "inheritance" of one or more threads on other nodes

Besides, the typology of threads above exposed sets big limits in the realization of the programs. Anyways some scientific applications, that effect practically only calculations (and therefore aim a lot at performances), could benefit from this solution (which I repeat: is only theoretical).

Conclusions

All the tests that I have conducted, although with no positive results, underline what the "practical" priorities are for the "general purpose" development of OpenMosix.

In the meantime I will begin to work again on PVM/MPICH on OpenMosix and, if it will be granted to me, I will try again to use Amber on OpenMosix (the first time was a failure for a series of misunderstandings that now I have solved).

I am confident in the advancement of the project and I don't hide my curiosity and wish to try as

soon as possible all the news that come to light.

(Sooner or later Java and OpenMosix will get along... :))