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'ye 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... :))