University of Modena and Reggio Emilia

DISMI - Technical Report: 23/2002

Open Source Solutions for Optimization on Linux Clusters

Roberto Aringhieri
DISMI - University of Modena and Reggio Emilia
Viale Allegri 13, 42100 Reggio Emilia - Italy
Email: aringhieri.roberto@unimore.it

Latest revision: August 5, 2002

Abstract: Parallel implementation of optimization algorithms is an alternative and effective paradigm to speed up the search for solutions of optimization problems. Currently a *Linux Cluster* is probably the best technological solution available considering both the overall system performance and its cost. Open Source community offers to researchers a set of software tools for setting up clusters and to optimize their performance. The aim of this paper is to review the open source tools are useful to build a cluster which efficiently runs parallel optimization algorithms. Particular attention is given to the OpenMosix approach to scalable computing.

Keywords: Open Source, Linux Cluster, Optimization, OpenMosix, Dynamic Load Balancing

1 INTRODUCTION

Although (sequential) optimization algorithms have reached a sophisticated level of implementation allowing good computational results for a large variety of optimization problems, usually the running time required to explore the solution space associated to optimization problems can be very large [33]. This is due to the fact that the complexity of (real) problems dealt by Operations Research community increased as well as the computational capacities (e.g. faster CPU, larger memory spaces, and so on).

With the diffusion of parallel computers and fast communication networks, parallel implementation of optimization algorithms can be an alternative and effective paradigm to speed up the search for solutions of optimization problems.

Klabjan, Johnson and Nemhauser [21] have proposed a parallel primal-dual simplex algorithm ables to solve linear programs with thousands of rows and millions of columns. The basic idea is to solve several linear programs in parallel combining all dual solutions to obtain a new dual feasible solution.

Although its computational capacities are continuously increased, PC and its components are cheaper than some years ago. By consequence, a great impulse to the proliferation of *clusters* has been given. A cluster is a set of PCs which communicates together through a network in order to carry out some computations. Currently a *Linux Cluster* is probably the best technological solution available considering both the overall system performance and its cost.

Recently, the Department of Energy's Pacific Northwest National Laboratory has ordered a 24.5 million Linux-based supercomputer. Consisting of 1,432 next-generation Intel Itanium, the new supercomputer will have an expected total peak performance of more than 9.1 teraflops. The supercomputer will be roughly 9,100 times faster than a current personal computer and it will be used to answer questions such as how radioactive waste can be processed and stored, and how proteins interact and behave in order to model a living cell. Once fully operational, the supercomputer should be the world's most powerful Linux based supercomputer and one of the top supercomputers in the world [22].

In order to implement parallel algorithms, the researcher usually deal with two main issues which are the *communication overhead* and the *load balancing*.

The first one is to find a good compromise between communication and processing avoiding communication overhead, that is the situation in which the algorithm is slow down by too much communication between CPUs. This problem can be solved at software engineering level by determining the degree of parallelism, or *granularity*, of the application. Several approaches to determine a good granularity of metaheuristic algorithms and the corresponding implementations are reviewed by Cung, Martins, Ribeiro and Roucairol in [11].

The second one is to find a policy giving a good load balancing. A good load balancing means a distribution of computation through the parallel computer which optimizes the utilization of the computational resources. As pointed out in [11, 33], a good load balancing is a crucial point to have good computational performance. Barak, Guday and Wheeler in [8] have proposed a dynamic load balancing approach based on preemptive process migrations.

The Open Source community offers to researchers interested in parallel computing a large set of software tools which helps to build an efficient and easy-to-use cluster. Recently, an article appeared in `FreshMeat.net` [17] counts more than 100 projects concerning clustering softwares which are related to High Performance Computing (HPC), High Availability, Load Balancing, File Systems, Monitoring and Management and Programming and Execution Environments.

The classical approach to setting up a cluster of PC, the Beowulf approach [36] consist of a set of Linux PCs connected by a private network (Fast Ethernet or Myrinet) and communicating through some libraries such as the Parallel Virtual Machine (PVM) [13] and the Message Passing Interface (MPI) [18, 30].

An alternative way is that proposed by MOSIX team [28] which is a set of enhancements of Linux/OS with adaptive resource sharing algorithms and a

mechanism for preemptive process migration that are geared for efficient cluster computing [8]. These algorithms are designed to respond dynamically to variations in resource usage among the nodes, by migrating processes from one node to another, preemptively and transparently, to improve the overall performance. Although MOSIX became proprietary software in late 2001, the OPENMOSIX project starts their activities in February 2002 under GPLv2 license providing new releases and support to Open Source community.

In the rest of the paper, we consider a cluster composed by a server which provides a series of services to a set of computer nodes connected together by a fast Ethernet private network. The server play also the role of workstation for local users. This configuration is the simplest available for a cluster.

In this paper we review some of the optimization tools provided by Open Source community. The main contribution is to introduce the OPENMOSIX approach to scalable clustering in the field of optimization problems which seems, to the best of our knowledge, unknown or not used implementing parallel algorithms.

In Section 2 we report how to setting up a cluster. The two different approach to parallel computing on a cluster of PC, introduced above, are discussed in Section 3 and 4, respectively. The target of Section 5 is to highlight the *pros* and *cons* of these approaches with respect to the development of algorithms for solving optimization problems. Conclusions and future works are outlined in Section 6.

2 SETTING UP A CLUSTER

Setting up a cluster could be a very difficult and long task for each system administrator [37]. The Open Cluster Group provides its open source solution called OSCAR standing for Open Source Cluster Application Resources.

OSCAR [19] is a package of RPM's, perl-scripts, libraries, tools, and whatever else is needed to build and use a modest-sized Linux cluster. The administrator download a single package, install it and the users are ready to start their parallel computation. In other words, OSCAR is a snapshot of current, *best-known-practices* in cluster computing. The main packages of OSCAR are:

- SIS, the System Installation Suite [34], is a Linux installation tool and it is used to install heterogeneous Linux cluster over a network. SIS uses an image based model for each cluster meaning that a copy of all the files that are installed on the nodes resides on the server. This also provides an option for maintenance as well as nodes uses `rsync` to match the image every time it is updated on the server.

- C3 is cluster management packages which allow to see the nodes of the cluster as single computer. For example, it provides `cshutdown` which allow to reboot or alt all the nodes of the cluster at the same time, or `cpush` which allow copying a file (typically a configuration file) in a given path to all nodes or to a subset of them. For example, `cpush -s=/usr/ilog/`

`-e=/usr` copies the ILOG packages to all nodes of the cluster putting them in the `/usr` path.

- the programming environment provided by OSCAR is based on message passing libraries. It provides both PVM and MPI. PVM library is that distributed through the project's home page [32]. Both MPICH [29] and LAM/MPI [24], two different open source implementation of MPI, are installed by OSCAR. MPICH is the default for MPI. We deeply discuss them in Section 3.
- OpenSSH guarantees secure connection which is a key feature for each system administrator whilst PBS manages workload ensuring that every computation job gets fair share of the cluster, and all resources get used efficiently.

Figure 1: The 8 processors cluster at DISMI

OSCAR requires to access (in some place of our local network or from CD) to a list of packages distributed within RED HAT suite. The current distribution of OSCAR, the release 1.3, works with distributions 7.1 and 7.2.. Usually, some standard and important packages are continuously updated from its distributor. In this case, before starting the installation process, the administrator can upgrade or modify the list of packages required to build the image which it will be copied to the nodes. For example, the administrator can prefer to install the latest versions of OpenSSH, kernel, and so on.

Our experience with OSCAR has been done with a cluster composed by a server (which also serves as workstation) and 4 dual processor nodes. The whole installation process (considering also the modification of the list of the packages required to build the image) takes about 1 workday. The operating cluster is that reported in Figure 1.

3 PARALLELIZATION USING MESSAGE PASSING LIBRARIES

The common programming environments available to develop algorithms on a cluster are certainly PVM and MPI. They are two Application Programming Interfaces (API) which provide different features to application developers of parallel and distributed programs.

The general goals of PVM project are to investigate issues in, and develop solutions for, heterogeneous concurrent computing. PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. The overall objective of the PVM system is to to enable such a collection of computers to be used cooperatively for concurrent or parallel computation. Briefly, the main principles upon which PVM is based include the following:

Process-based computation: the unit of parallelism in PVM is a task which often corresponds to a Unix process. It is an independent sequential thread of control alternating between communication and computation.

Explicit message-passing model: a collections of tasks cooperate by explicitly sending and receiving messages to one another which size is limited only by the amount of available memory.

Heterogeneity support: PVM supports heterogeneity in terms of machines, networks, and applications. It is also allowed the exchange of messages containing more than one data type between machines having different data representations.

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory and each vendor of Massively Parallel Processor (MPP) has implemented its own variant. In order to give an answer to the need of portability between different MPP, MPI is intended to be a standard message passing specification that each MPP vendor would implement on their system. The main features of MPI-1 specification are:

- a large set of point-to-point communication routines among two processes,
- a large set of collective communication routines for communication among groups of processes,
- the ability to specify communication topologies.

Additional features are then added to MPI-2 specification, including:

- a function allowing to start both MPI and not-MPI processes,
- one-sided communication functions,
- nonblocking collective communication functions.

A deeply comparisons of PVM and MPI has been done by Geist, Kohla and Papadopoulos in [14]. The authors conclude their work observing that MPI has "the advantage of expected higher communication performance" and a "much richer set of communication functions". From this point of view, MPI seems more suitable than PVM in order to avoid communication overhead. On the other side, MPI lacks of "the interoperability between any of the MPI implementation" and "the ability to write fault tolerant applications". Moreover, PVM "contains process control functions that are important for creating portable applications that run on cluster composed by workstations and MPPs. Furthermore, "the ability to write long running PVM applications that can continue even when hosts or tasks fail" is a very important feature of PVM .

The aim of PVMPI is to interface the flexible process and virtual machine control from the PVM system with the enhanced communication system of several MPI implementations [12].

Different projects developing Open Source packages both for PVM and MPI are available.

PVM is supported by the original team of developers as research projects and its distributed through `NetLib.org`. Although its popularity, the projects seems to reach a final point with the latest version released in the late September 2001.

On the other side, the two main open source implementation of MPI, MPICH and LAM/MPI seems to be more vital than PVM. Currently, the stable versions of MPICH and LAM/MPI are respectively 1.2.4 and 6.5.6. Both MPICH and LAM/MPI implement all the features defined in MPI-1.2. The two projects differ from the implementation state of MPI-2 standards. In particular, LAM/MPI implements those functions concerning dynamic process spawning and one-sided communication which are not yet implemented in MPICH. On the other side, MPICH seem more robust than LAM/MPI.

An alternative project concerning high performance messaging is SPREAD [35]. SPREAD is an API that provides a high performance messaging service that is resilient to faults across external or internal networks. Is is designed to encapsulate the challenging aspects of asynchronous networks and enable the construction of scalable distributed applications, allowing application builders to focus on the differentiating components of their application.

4 PROCESS MIGRATION FOR HPC

A *process* is an operating system abstraction representing an instance of a running computer program. Process migration is the act of transferring a process between two machines during its execution. Process migration enables:

- **dynamic load distribution**, by migrating processes from overloaded nodes to less loaded ones,

- **fault resilience**, by migrating processes from nodes that may have experienced a partial failure,
- **improved system administration**, by migrating processes from the nodes that are about to be shut down or otherwise made unavailable,
- **data access locality**, by migrating processes closer to the source of some data.

A very detailed survey on process migration research has been done by Milojicic and others in [27].

MOSIX is one of the most famous research project concerning process migration. MOSIX is a set of enhancements of Linux/OS with adaptive resource sharing algorithms and a mechanism for preemptive process migration that are geared for efficient cluster computing [8].

MOSIX was originally developed by students of Prof. Amnon Barak at the Hebrew University of Jerusalem and was in the eighties used by the US Air Force to cluster its PDP11/45 computers. Scientific applications of MOSIX are protein sequences, quantum simulation of large molecules, molecular dynamics simulation and others [9]. Recently, MOSIX has been successfully applied to improve the performance of a compiler farm [26].

Although MOSIX is proprietary software from late 2001, the OPENMOSIX project starts their activities in February 2002 under GPLv2 license providing new releases and support to Open Source community. In this Section, we refer to the OPENMOSIX project.

An OPENMOSIX cluster of Linux machines is a Single System Image cluster in which the set of nodes cooperates in such a way that they emulate a bigger computer in terms of computational resources. The most noticeable properties of executing applications on OPENMOSIX are its adaptive resource distribution policy and the symmetry and flexibility of its configuration. The combined effect of these properties implies that users do not have to know the current state of the resource usage of the various nodes, or even their number. Parallel applications can be executed by allowing OPENMOSIX to assign and reassign the processes to the best possible nodes, almost like an SMP.

The OPENMOSIX technology basically consists of two parts: a Preemptive Process Migration (PPM) mechanism and a set of algorithms for adaptive resource sharing. Both parts are implemented at the kernel level, using a loadable module, such that the kernel interface remains unmodified. Thus they are completely transparent to the application level.

4.1 Preemptive Process Migration

PPM is the main tool for the resource management algorithms. PPM can migrate any process, at anytime, to any available node. Usually, migrations are based on information provided by one of the resource sharing algorithms, but users may override any automatic system-decisions and migrate their processes manually. Each process has a Unique Home-Node (UHN) where it was created.

OpenMosix is a cache coherent cluster, in which every process seems to run at its UHN, and all the processes of a user's session share the execution environment of the UHN. Processes that migrate to other (remote) nodes use local (in the remote node) resources whenever possible, but interact with the user's environment through the UHN. For example, if the user executes `ps`, it will report the status of all the processes, including processes that are executing on remote nodes. To implement the PPM, the migrating process is divided into two contexts: the user context (*remote*) and the system context (*deputy*) (see figure 2). While the remote can migrate many times between different nodes, the deputy is never migrated.

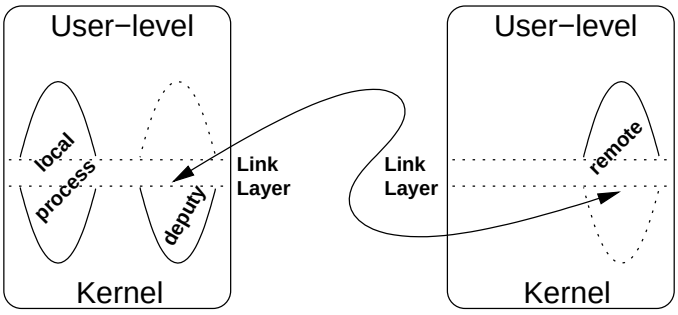

Figure 2: A local process and a migrated process [10]

The migration time has a fixed component, for establishing a new process frame in the new (remote) site, and a linear component, proportional to the number of memory pages to be transferred. In [9] are reported some experimental data: to migrate a process of about 8 Mb is required less than 0.9 seconds on a Ethernet-100 LAN and less than 0.3 seconds on a Myrinet LAN.

4.2 Resource Sharing Algorithms

The main resource sharing algorithms of OpenMosix are the load balancing and the memory ushering.

The dynamic load balancing algorithm continuously attempts to reduce the load differences between pairs of nodes, by migrating processes from higher loaded to less loaded nodes. This scheme is decentralized and all the nodes execute the same algorithms, and the reduction of the load differences is performed independently by pairs of nodes. The number of processors at each node and their speed are important factors for the load balancing algorithm. This algorithm responds to changes in the loads of the nodes or the runtime characteristics of the processes. It prevails as long as there is no extreme shortage of other resources such as free memory or empty process slots. Determining the optimal location for a job is a complicated problem since the resources are heterogeneous and so forth are incomparable. The algorithm employed by OpenMosix tries to reconcile these differences based on economic principles and competitive

analysis. The key idea of this strategy is to convert the total usage of several heterogeneous resources, such as memory and CPU, into a single homogeneous cost. This economic strategy provides a unified algorithm framework for allocation of computation, communication, memory, and I/O resources. It allows the development of near optimal online algorithms for allocating and sharing these resources [3].

The memory ushering (depletion prevention) algorithm is geared to place the maximal number of processes in the cluster-wide RAM, to avoid as much as possible thrashing or the swapping out of processes. The algorithm is triggered when a node starts excessive paging due to shortage of free memory. In this case the algorithm overrides the load balancing algorithm and attempts to migrate a process to a node that has sufficient free memory, even if this migration would result in an uneven load distribution [6].

4.3 Information Collection Statistics, Decentralized Control and Autonomy

Statistics about a process' behavior are collected regularly, such as at every system call and every time the process accesses user data. This information is used to assess whether the process should be migrated from the UHN. These statistics decay in time, to adjust for processes that change their execution profile. Each process has some control over the collection and decay of its statistics. For instance, a process may complete a stage knowing that its characteristics are about to change, or it may cyclically alternate between a combination of computation and I/O.

OpenMosix has no central control: each node operates as an autonomous system, and all its decisions are independently. This design allows a dynamic configuration, where nodes may join or leave the network with minimal disruptions. Additionally, this allows for a very great scalability and ensures that the system runs well both on small and large configurations. Scalability is achieved by incorporating randomness in the system control algorithms, where each node bases its decisions on partial knowledge about the state of the other nodes, and does not even attempt to determine the overall state of the cluster or any particular node. For example, in the probabilistic information dissemination algorithm, each node sends, at regular intervals, information about its available resources to a randomly chosen subset of other nodes. At the same time it maintains a small window, with the most recently arrived information. This scheme supports scaling, even information dissemination and dynamic configurations.

4.4 Direct File System Access

When a process migrates, it has to be able to continue any I/O operations on its UHN. A typical options is the mounting of all file systems needed by using the network file systems protocol (NFS).

Taking the best of modern file system research such as Global File System [38], OpenMosix proposed the Direct File System Access (DFSA) [2]. DFSA is designed to reduce the extra overhead of executing I/O oriented system-calls

of a migrated process. The basic idea is simple: unlike all existing network file systems, i.e. NFS, which bring the data from the file server to the client node over the network, OpenMosix attempts to migrate the process to the node in which the file actually resides. This is done by allowing the execution of most such system-calls locally, i.e. in the process's current node. In addition to DFSA, an algorithm that takes into account I/O operation was added to the OpenMosix process distribution policy in such a way that a process that performs moderate to high volume of I/O is encouraged to migrate to the node in which it does most of its I/O.

A file system which guarantees *cache consistency* is needed by DFSA. Mosix File System (MFS) is a prototype file system which guarantees cache consistency and provides an unified view of all files on all mounted file systems on all the cluster's nodes.

Table 1: File systems access time (avg. time for 10 executions in sec.) [2]

access method	Data transfer block size 64B	512B	1KB	2KB	4KB	8KB	16KB
local	102.6	102.1	100.0	102.2	100.2	100.2	101.0
MFS with DFSA	104.8	104.0	103.9	104.1	104.9	105.5	104.4
NFS	184.3	169.1	158.0	161.3	156.0	159.5	157.5
MFS without DFSA	1711.0	382.1	277.2	202.9	153.3	136.1	124.5

Table 1 reports the results of different access methods simulating heavy file system loads using PostMark benchmark [31]. The benchmark has been executed between a pair of identical Pentium 550 MHz with IDE disks using four different file access methods and ranging block sizes from 64 bytes to 16K bytes. The results show that MFS with DFSA is 1.8-4.9% slower than local access method. Moreover, it is 51-76% faster than NFS.

4.5 Performance

Performance of PVM with the Mosix PPM are investigated in [7]. The authors compares the execution of sets of identical CPU-bound processes under PVM, with or without process migration, in order to highlight the advantages of the PPM mechanism and its load balancing scheme.

The results (see the most significative in Table 2) show that at least Mosix is not worse than PVM . When the number of processes increases, the average slowdown of PVM vs. Mosix is over 15%. This slowdown can become very significant, e.g. 32.3% for 17 processes and 28.4% for 33 processes. In contrast, the measurements show that PVM with the Mosix PPM is slightly better than Mosix itself, due to residency period that is imposed to any processes by Mosix.

Table 2: Comparing PVM and MOSIX execution times in secs. [7]

No. of processes	Optimal time	MOSIX time	PVM time	PVM slow-down %	PVM on MOSIX
8	300	305.73	308.57	0.9	301.88
16	300	310.83	317.12	2.0	303.40
17	450	456.91	604.36	32.3	452.84
27	525	549.07	603.86	10.0	559.81
32	600	603.17	603.14	0.0	604.64
33	700	705.93	906.31	28.4	707.39
40	750	767.67	905.39	17.9	771.71
48	900	916.11	908.51	-0.8	907.71

5 WHICH IS BETTER FOR OPTIMIZATION?

The answer strictly depends on the type of application that we are interested to develop.

As already discussed in Section 1, the development of an efficient parallel algorithm depend on the techniques used to avoid communication overhead and the policy used to have a good load balancing.

While communication overhead can be avoided both defining the right granularity of the application and using efficient communication libraries, a good load balancing is a difficult task since we do not have any information concerning the loads of other cluster machines. For example, let us consider the Knapsack Problem (KP) (for a detailed presentation see [25]), whose mathematical model is the following

$$\begin{aligned} \text{KP: min} \quad & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} p_i x_i \geq M \\ & x_i \in \{0,1\}, \end{aligned}$$

and a simple Branch and Bound algorithm in which, at each branch decision, a binary variable is fixed to 0 or 1. The subproblems obtained by fixing a binary variable can be harder or easier than we will expect, but we do not know anything about their running time before fixing that variable. In general, this means that we can assign high loaded processes to machine in the same way as we assign the lighted ones. Clearly, we can implements a sort of dynamic load balancing in which, every period of time, information about running processes are collected and then an assignments of new jobs to machines can be made.

In the most simple case, the computation carried out by a parallel optimization algorithm can be described as a set of tasks, which explores a part of the solution space, usually communicating at the beginning and at the end of their search. This is the case of neighborhood exploration done by some parallel metaheuristics. Typically, high loaded processes can be generated by large scale neighborhoods [1] or by the evaluation of hard objective functions arising in some stochastic problems, e.g. the stochastic routing problem [15].

Usually there are also periodically synchronization steps in which some global and crucial information are updated. For instance, during Branch and Bound exploration, one process, usually the master, can collect all the current best solutions, computes the best ones and then distributed it to all processes which can continue their computation with an improved bound.

In most cases, we can suppose that an optimization algorithm does not require so much sophisticated communication libraries as those defined by MPI standards, MPI-1 and MPI-2. Actually, these libraries are developed keeping in mind the efficiency of typical parallel computation, e.g. matrix operations, which are the basic elements for computation in Physics, Computer Graphics and so on.

On the contrary, optimization algorithms usually do not perform so complicated computation and their difficulty depends on the magnitude of the solution space which is required to be explored.

From our point of view, a good load balancing is the first issue which is needed to be addressed. The second issue is the possibility of dynamic spawning of new processes. This issue corresponds to the `fork( )` system call provided by C language. At more high level, both PVM and MPI (MPI-2 specification) provides this issue. The third and last issue is that to have good communication performance between processes.

In conclusion, we think that an appropriate programming environment should be based on OpenMosix integrated by PVM or MPI depending on the needs of good dynamic spawning procedures or more efficient and communication libraries. A well tuned OpenMosix cluster can guarantee a near optimal resource sharing whilst PVM and MPI provide the required basic function to implement an efficient parallel algorithm.

It is our opinion that, in most cases, the process control and the communication libraries provided by POSIX C language are all we need to implement efficient parallel optimization algorithm running under OpenMosix cluster.

For instance, suppose to implement a Variable Neighborhood Search (VNS) algorithm [20]: VNS is a metaheuristic in which, at each step, more than one neighborhood are evaluated in order to chose the next solution among those computed by exploring all the neighborhoods. A parallel VNS can assign the evaluation of each neighborhood to a different process and then collect all the results to compute the next solution.

In this context, we observe that `fork( )` creates a child process which inherits all the data structures (and their values) from the parent avoiding the communication needed for starting the communication. The result of each single exploration can be easily collected by using the interprocess communication

library which allows both asynchronous communication and messages with priority.

Although it requires to make a further programming effort, the computer programmer has the complete control of its computer code and he is able to generate simpler and smaller codes than those including communication libraries. We observe that a small process migrates faster than a larger one.

6 CONCLUSIONS AND FUTURE WORK

Different Open Source tools for parallel programming on Linux clusters are reviewd in this paper. These tools ranges from the installation and management of a Linux cluster to programming environments such as PVM and MPI.

Great emphasis is given to OpenMosix approach to scalable computing highlighting all the peculiarity which can be used to successfully address the implementation of efficient optimization algorithms. In particular, the resource sharing algorithms provided by OpenMosix seem to be very helpful to obtain a good load balancing policy.

Current and future research tries to explore the potentiality of parallel optimization under OpenMosix environment. In particular, we are addressing the problem of solving a class of stochastic optimization problem arising from the design of SONET network for which efficient metaheuristics, based on Tabu Search and Scatter Search methodologies [16, 23], have been already developed [4, 5].

Acknowledgments

This research was supported by University of Modena and Reggio Emilia, Italy under "Progetto Giovani Ricercatori 2001".

References

[1] R.K. Ahuja, Ö. Ergun, J.B. Orlin, and A.B. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

[2] L. Amar, Barak A., Eisenberg A., and Shiloh A. The MOSIX scalable cluster file systems for linux. Technical report, www.mosix.org, 2000.

[3] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Tran Parallel and Distribuited Systems*, 11(7):760–768, 2000.

[4] R. Aringhieri and M. Dell'Amico. Intensification and diversification strategies for the SONET network design problems. Working Paper, August 2002.

[5] R. Aringhieri and M. Dell'Amico. Solution of the SONET ring assignment problem with capacity constraints. In C. Rego and B. Alidaee, editors,

Adaptive Memory and Evolution: Tabu Search and Scatter Search. Kluwer Academic Publishers, 2002.

[6] A. Barak and A. Braverman. Memory ushering in a scalable computing cluster. *Journal of Microprocessors and Microsystems*, 22(3-4), 1998.

[7] A. Barak, A. Braverman, I. Gilderman, and O. Laden. Performance of PVM with the MOSIX preemptive process migration. In *Proc. 5-th Israeli Conf. on Computer Sys and Software Eng.*, pages 38–45, 1996.

[8] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distribuited Operating System, Load Balancing for Unix*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.

[9] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, 1998.

[10] A. Barak, O. La'adan, and A. Shiloh. Scalable cluster computing with MOSIX for linux. In *Proc. 5-th Annual Linux Expo*, pages 95–100, 1999.

[11] V. Cung, S. L. Martins, C. C. Ribeiro, and C. Roucairol. Strategies for the parallel implementation of metaheuristics. In C.C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 263–308. Kluwer Academic Publishers, 2001.

[12] G. E. Fagg and J. J. Dongarra. PVMPI: An integration of the PVM and MPI systems. *Calculateurs Parallèles*, 8(2):151–1660, 1996.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, MA, 1994.

[14] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Parallèles*, 8(2):137–150, 1996.

[15] M. Gendrau, G. Laporte, and R. Seguin. A tabu search heuristic for the vehicle routing problem with stochastic demands and customers. *Operations Research*, 44(3):469–447, 1996.

[16] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[17] J. Greenseid. linux clustering software. Technical report, FreshMeat.net, June 2002.

[18] W. Gropp, E. Lust, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[19] The Open Cluster Group. *OSCAR: A packaged Cluster software stack for High Performance Computing*. http://www.openclustergroup.org, February 11 2002.

[20] P. Hansen and N. Mladenović. Variable neighborhood search. In P. M. Pardalos and M. G. C. Resende, editors, *Handbook of Applied Optimization*. Oxford Academic Press, 2001.

[21] D. Klabjan, E. L. Johnson, and G. L. Nemhauser. A parallel primal-dual simplex algorithm. *Operations Research Letters*, 27(2):47–55, 2000.

[22] P.N.N. Laboratory. http://www.pnl.gov/news/2002/computer.htm, Aprile 2002.

[23] M. Laguna. Scatter search. In P. M. Pardalos and M. G. C. Resende, editors, *Handbook of Applied Optimization*. Oxford Academic Press, 2001.

[24] LAM/MPI. http://www.lam-mpi.org.

[25] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, Chichester, 1990.

[26] S. McClure and R. Wheeler. MOSIX: How linux clusters solve real world problems. In *Proc. 2000 USENIX Annual Tech. Conf.*, pages 49–56, 2000.

[27] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration survey. *ACM Computing Surveys*, 32(3):241–299, 2000.

[28] MOSIX. http://www.mosix.org.

[29] MPI. http://www-unix.mcs.anl.gov/mpi/.

[30] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Pub. Inc., 1996.

[31] PostMark. http://www.netapp.com.

[32] PVM. http://www.csm.ornl.gov/pvm/.

[33] C. Roucairol. Parallel processing for difficult combinatorial optimization problems. *European journal of Operational Research*, 92(3):573–590, 1996.

[34] SIS. http://sisuite.org.

[35] Spread. http://www.spread.org.

[36] T. Sterling, G. Bell, and J. S. Kowalik. *Beowulf Cluster Computing with Linux*. MIT Press, 2001.

[37] T. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese. *How To Build a Beowulf: A Guide to the implementation and Application of PC Clusters*. MIT Press, 1999.

[38] Global File Sytem. http://www.gfs.org.