pCor: a protoype for resource oriented computing

A. Pina[1], V. Oliveira[1], C. Moreira [1] & A. Alves[1]
[1] *Departamento de Informática, Universidade do Minho, Portugal*

Abstract

In this paper we present CoR a resource oriented computing model that address the question of how to integrate user-level fine-grained multithreading, communication and coordination into a cluster of symmetrical multiprocessor computers. To support the design of complex distributed application using the proposed paradigm we built pCoR a run-time system which has new areas that represents extensions to the strict shared memory and message passing models supported by other platforms: remote operations, dynamic domains, communication ports, multithreading management, shared memory, replication and partition are some of its distinguished features. In addition, it provides a thread-safe transport communication layer to take advantage of modern high-performance commodity hardware/software like Myrinet network.

1 Introduction

The emerging field of parallel computing illustrated by the increased importance of clustering technologies[1] puts strong demands to the problem of organizing computation, communication and coordination. On large-scale projects where the communication patterns cannot be determined in advance the main problem is correctness, due to the non-determinism and concurrent access to resources. Another issue that poses additional requirements to the task of parallel programming and that can have a great impact upon performance is the need to establish a wised relationship between problem specification and decomposition onto the hardware of a parallel computer.

The rise of popular public domain software such as PVM[2] and MPI[3] had a major influence on the wide use of parallel computers, ranging from high performance supercomputer to clusters of workstations or home computers due to

the normalized application development environments. Collections of workstations interconnected by a fast network, allow the farming of jobs over a unified logical virtual parallel machine. Even when the network performance capability is minimal, these tools provide an inexpensive testing vehicle for classes of problems where the ratio computation/communication is very high. Other options are available as high speed networking technology developed for large scale parallel machines migrates to more inexpensive systems and computers of more than one processor that provide multiple simultaneous points of execution.

In what follows, we briefly present some of relevant techniques, and platforms that can be used to identify and compare our approach to the problem of organization computation, communication and coordination to the task of parallel programming. Next, we introduce the principle contributions of CoR a new computing model in the origin of this work. Finally, we present pCoR, a multithread resource oriented library aimed to provide a common framework in which to exploit and evaluate user level fine-grained computation and communication over clusters of shared-memory multiprocessors.

2 Background

Parallel programming is a challenging and complex design space that enables programmers to deal with larger and more sophisticated problems. However, they can no longer rely on the simple and stable programming model of Veumann cause they have to face several new dimensions that usually do not appear in sequential program development.

In areas where clustering are been primarily applied, to cope with the new dimensions of complexity we need new models, tools and environments in order to help programmers in their activity to make the hardware more usable and application more performing. As a matter of fact, in many cases, high-performance is difficult to achieve due to the lack of adequate software development methodologies and tools[4].

2.1 Communication and Multithreading

As parallel distributed computing evolved it has come to evidence that no monolithic system can handle efficiently all the desired computation and communication styles. The issue of threads has been widely discussed and implemented in slightly different ways by various vendors and academics.

The use of threads to perform operations on behalf of the overall application is extremely convenient for several reasons: 1) threads provide a natural implementation of a non blocking operation that may be applied both to communication, sharing and coordination between different threads of control; 2)threads are becoming the dominant parallel programming model for symmetric multiprocessing shared memory computers; 3)threads can improve performance by helping highly latency systems to be more latency tolerant. Fortunately the POSIX standard also known as *Pthreads*[5] seems likely to become the most used programming definitions for

threads. The specification does not impose whether the treads are user or kernel level; it is up to the threads implementation.

In MPI-2 threads were recognized as an important natural programming model for symmetric multiprocessors[6] that separate a process into a single address space and one or more threads of control being an effective way to hide latency in high-latency operations.

Platforms like TPVM[7], LPVM[8], a modified version of P4[9], Chant[10] and Athapascan-0 [11] allow the creation of multiple threads that interact using global identifiers and send/receive primitives. The late adds the concept of ports and requests so that any thread can receive a message sent to a particular port. Panda[12], PM2[13] and Nexus[14] also include threads and remote execution manage communication by executing handlers registered by the user. Nexus also introduces the context as an important concept used to structure applications.

3 Overview of CoR

CoR (Resource oriented Computing) has been primarily motivated by the need to support the design and evaluation of the MC^2 (Cellular Computation Model)[15]. The model combines production systems with Petri Nets as a way to specify and regulate the overall activity of a distributed system viewed as a multi-cellular agent[16].

Another strong motivation is the need to provide adequate models and programming tools to assist parallel programmers, with varying degrees of understanding and skills, in the job of producing structured and performing software. A third motivation is the exploiting of fine-grained user-level computation, communication and coordination in large scale application designed to run on clusters of shared memory multiprocessors. The final motivation is to investigate and construct a common frame-work in which to understand and evaluate architectural and logical trade-offs of software/hardware interaction.

3.1 Resources

As a new computation model CoR introduces the **resource** as a generic user metaphor that directly incorporates the notion of state, concurrency, locality and distribution. It also supports composition and coordination between applications to deal with the growing complexity of contemporary systems and applications.

The resources are the abstractions we use to simplify program development and execution by freeing the user of the burden of explicitly managing the complex relationships between the two phases typically involved on the development of large distributed parallel application programs: *structuring* and *computing*.

Structuring is a series of separate stages ranging from problem analysis and specification to the naming and design of the entities on the application. Computing is the continuous process of mapping the entities onto the hardware and executing the instructions of the distributed multi-threaded control parallel program to inquire, transform and communicate the state of the named entities.

Tasks and domains are physical resources that dim the traditional boundaries between the physical hardware and logical software components. A task is the most elemental unity of execution while domains delimit regions of addressable physical space where interaction between logical and physical resources occurs. Logical resources includes: ports) a communication mechanisms used to route the information between domains, data) a general class of resources used to contain structured and unstructured pieces of information. Synchronizers, barriers, transitions and topologies area other logical resources suited to the tasks of synchronization and coordination of large scale multi-threadead applications.

A logical domain is a special case of logical resource derived from MC^2 to support the design of large, complex and modular applications. It is used to organize resources in a hierarchy tree of dependencies, rooted by the first domain in the application, where nodes are *structured resources* (domains) and leaves *simple resources*. Several existing systems use groups, the equivalent of logical domains, for slightly different reasons[17, 3] allowing either for static or dynamic membership allocation.

The aggregation of resources inside a domain may be viewed as a first form of composition equivalent to a program constituted by a bunch of modules. Higher level of composition may be achieved through the association and integration at runtime of several applications through a designated logical domain (see section 3.2).

3.2 Identification

An application is viewed as a system of distributed domains where domains provide for the identification and representation of the resources they encompass.

The identification assigned to every resource is obtained from the *meta-domain*, a special logical domain that acts as a central maintainer and generator of identifiers belonging to the same application. It is a universal identifier that comprehends a *macro-identification* and a *micro-identification*; the late being used to uniquely identify the resource in the application (its principal identification *idp*), whereas the first encodes an index used to join and integrate on-the-fly pluggable resources, belonging to different applications.

Resources are directly represented at the domain level. They may be constituted by one or more built-in objects related with **computer resources** like *containers*, *memory*, *port* or **operon** (made of one or more *executors*), and **body**, like *synchronizer*, *mail-box*, *organizer* or *data* (see figure 1).

3.3 Exchange of identity

With the exception of domains and tasks all other resources are considered passive agents. Domain autonomy is related with the management and representation of the computer resources used to support the entities of an application. Tasks are autonomous agents whose activity results of the actions performed by an *executor* – a thread of control that executes a function – viewed at the implementation level

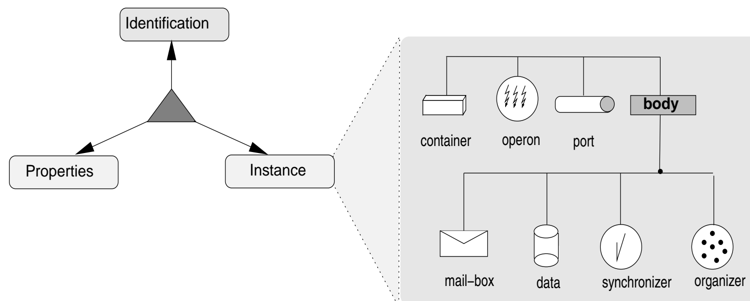

Figure 1: Resource anatomy.

as a computer resource.

All resource management and state transformation operations are performed in the context of a user executor or a proxy executor delegated by the run-time system. Executors are allowed to pursuit their action after assuming (or borrowing) the principal identification of a target resource. Executors may be dynamically allocated or de-allocated at run-time, or statically assigned to a newly created resource as it happens every time a simple task is created.

Whenever an executor needs to operate a resource, distinguished from the one it gets the current principal identification, it executes an *exchangeOfIdentity* primitive that when successful gets the idp of the target resource. To return to the original identity the executor must evocate a *returnOfIdentity* primitive.

3.4 Replicas and partitions

Naming is the most elemental mechanism that can be used to identify each individual resource, thus promoting spatial concurrency. Another mechanism based on *alias* is used to increase the concurrency at the identification level that fails to produce real concurrency. In this case we are sharing the body of a solely resource using different identifications.

However, in a system of distributed domains the need for efficiency may determine the existence of multiple *replica* of the same body; as it happens when the alias operation spawn multiple domains. In this situation the system may automatically manages both local and remote representations of the solely resource, ensuring the consistency of replicas, according to a default or elected policy.

As an alternative to replication, a complex resource (the equivalent of a structure in C) may be partitioned in such a way that each *partition* may be recognized as an individual resource on itself, also represented by a principal identification and a body. This powerful mechanism has the potential to increase the concurrency and reduce the communication overhead by maintaining the consistency of one sole representation of the distributed shared body.

To manage all forms of user and system identification the micro-identification (see section 3.2) may be seen as comprehending several distinct fields that may

be used to distinguish between the different idps of the same resource, irrespectively of the representation scheme used to address a resource – alias, replicas or partitions.

As an example consider the use of a task whose mailbox is partitioned between multiple alias as an alternative to a task that shares a simple mail-box between its alias (see figure 2). The advantages of the first approach are evident when it is necessary to identify the real destination of a message. In case of data decomposition over a system of distributed domains the first approach also favors concurrency and has the potential for better efficiency when dealing with large volumes of data.

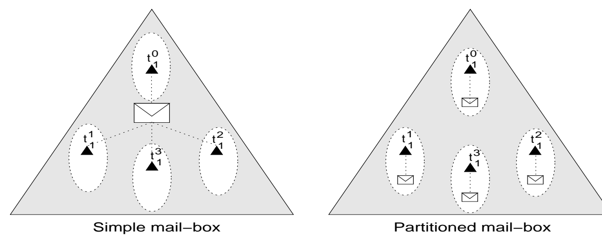

Figure 2: Task alias.

3.5 Asynchronous operation model

CoR offers programmers the choice between synchronous or asynchronous models of operation for most of the calls it supports. If the initiator of a call selects the synchronous model it always suspends waiting the operation to terminate to resume execution. This mode of operation using *blocking* calls are best suitable to guarantee the determinism of a multithreaded parallel program. When selecting the asynchronous model the operation immediately returns a handler to the initiator that may later be used to inquire the status of the requested operation or to revert to synchronous model of operation.
Asynchronous operation model provides the means to enhance the concurrency within each domain to increase system responsiveness and efficiency and in some situations to avoid deadlocks. In a system of distributed domains, most non-local operation are natural candidates to immediate calls due to the considerable overhead introduced by the transport communication layers between domains.

3.6 Message passing contexts

CoR communication model assumes that any resource, not only tasks, may send a message to any other resource with the guaranty that messages sent in the same path are received in the order they are sent, without loss. Messages are labeled with a user-level supplied tag and a system-level tag along with the identification of the sender. The user-tag allows discriminating between multiple messages sent to the

same destination. The system-task may be used for library writer and user alike in a way which is broadly equivalent to the use of a communicator in MPI[3].

CoR uses the idps of domains (as they are unique and generated by the system at-runtime) as system-tags. This way messages are addressed or received through a domain context does allowing for the creation of private communication spaces where messages may not be sent/received outside the designated domain.

3.7 Ports

The aim of management in the world of clusters is the utilization of computers resources at the highest possible degree. In respect to communication sub-systems CoR assumes that at each cluster node communication adapters may be divided by the runtime library into a certain number of disjoint communication end-points called *ports*. Each port is a communication path to other hardware compatible ports used to communicate with remote domains.
At run-time, inside each domain, fully abstracting from the specific communication medium and protocols, several *soft-ports* can be open to share or exclusively use the underlying communication adapters, and then closed to release the ports. In figure 3 we show several domains each one having two soft ports sharing the available ports in the respective node.

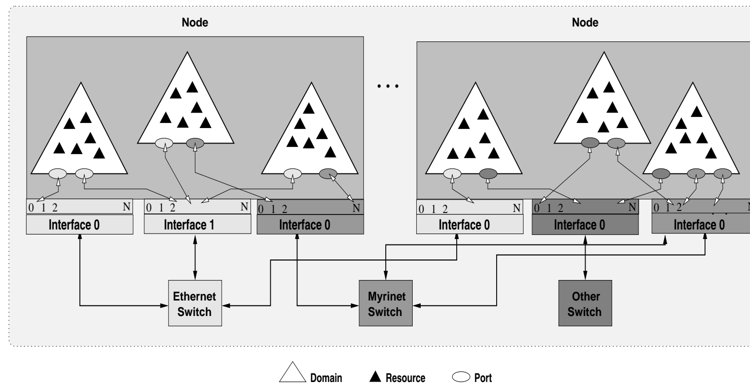

Figure 3: Transport layers in a system of distributed domains.

4 pCoR

pCoR is intended as a general purpose run-time system built to support and evaluate the proposed resource oriented computing paradigm. It is based on a former effort to combine multithreading, message passing and shared memory[18]. Currently is been used under SIRe (Scalable Information Retrieval Environment) a research project partially supported by contract POSI/CHS/41739/2001, FCT/MCT

Portugal, and in classes by students of University of Minho and Polytechnics Institute of Bragança.

The overall design enhances portability and extensibility by providing a programmatic interface to static and dynamic linker which can be used to bind and unbind on-the-fly system shared libraries or user dynamic executables. By combing ELF Dynamic Linking with POSIX threads, GNU compiler and linker we ensure the portability of the code and easy support for a wide range of heterogeneous environments and operating systems.

4.1 Design issues

A pCoR application is a system of distributed domains translated into user programs and scheduled as processes at the cluster nodes. pCoR resources are functional implementations of CoR concepts, slightly deviated from the original, that are created in the computing space delimited by physical domains.

Domains are containers that organize or structure other resources and provide for safe contexts and the execution on-the-fly of pluggable modules. Tasks are user functions scheduled as autonomous threads of control that exploit dynamic libraries to allow the binding of local functions by name at runtime. Data are regions of contiguous memory viewed by programmers as distributed shared memory (object-oriented) protected by the traditional acquire and release operations of the release consistency model.

Other resources were added to the runtime: ports, as transport definitions, *barriers* for synchronization purpose and *mutexes* for defining mutual exclusion regions of shared code.

4.2 System architecture

The core of domains consists of several independent subsystems implemented as an hierarchy of layers from the low-level interface of system dependent services – POSIX threads, dynamic libs and communication – to the higher level interfaces of system and application APIs (see figure 4.2).

Taking advantage of the shared address space inside each domain most of the local operations and services are executed directly by pCoR library, reducing the cost of scheduling system threads.
Each domain includes a *controller*, a thread responsible for the interconnection of the application domains, and one or more *mailers* which are responsible for moving messages from the transports communication layers to the internal messaging system. Other system threads may be spawned in response to external service requests or events.

4.3 Shortcuts and replication

All the resources including tasks have a principal identification that makes them visible outside the ascendant domain (usually a process), along with a local name

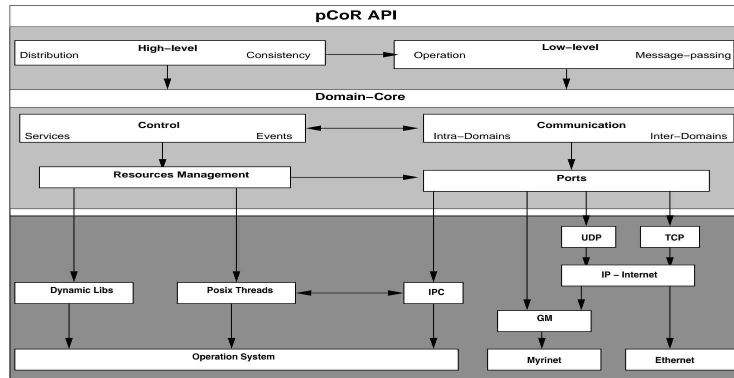

Figure 4: The core domain layers.

and membership index. In addition any resource can become a member of another ascendant (a domain) under a different name or *alias* – a proper new membership index and name plus and a new principal identification. However original resources and their alias share the same body.

Any time a pCoR primitive references a remote resource the run-time provide for the automatically creation of a local resource replica ensuring the consistency of all the scattered replicas according to a release consistency multiple reader-single writer protocol. Within a domain the life time of every local replica is a system dependent variable. However the programmer may influence system behavior by asking for a local identifier of a resource a *shortcut* – which has the effect of create a consistent local replica of the target resource which is made permanent until being explicitly discharged. Shortcut may not be used outside the local domain.

4.3.1 Resource tree

pCoR promote a unified approach to the task of distributed parallel programming by representing computing nodes and target architectures through the same metaphor that is used for the entities on the application. In addition, the same set of primitives may be used to manipulate resources in the application model and in the computer nodes. For example, the addition/deletion of a node of a parallel distributed computer is equivalent to the creation/destruction of a resource.

Along with the representation of nodes, architectures and the virtual machine by using logical domain, several other proxy resources can be created to registry information about each node of the cluster like the amount of memory or the number of processors. This kind of information may be of relevant interest to programmers as it may be used as an input, for example, to select the nodes where to spawn new physical domains.

Every pCoR running application draws a dynamic resource dependency tree that may be used to determine or trace the established relation-ship between all the liv-

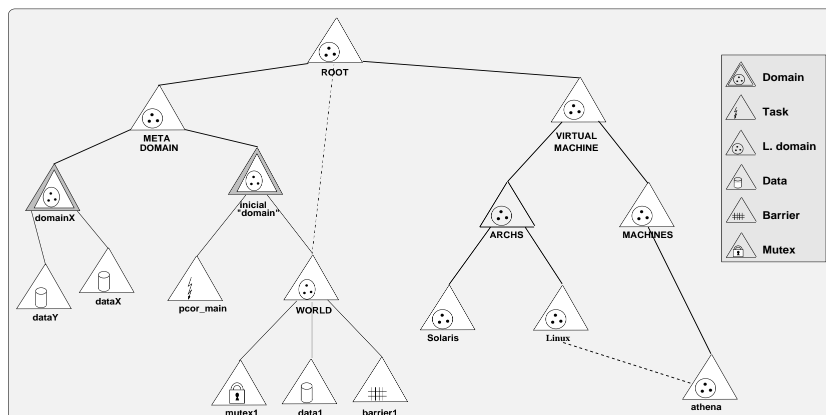

Figure 5: The hierarchy tree of dependencies.

ing resources. By following an absolute path initiated on ROOT or a relative path initiated at any domain in the tree it is also possible to identify resources by index or name.
In figure 5 the root of the tree represented by the constant ROOT is the ascendant of both the METADOMAIN, that contains all domains in the application, and the VIRTUALMACHINE used as a sub-tree of logical resources that represent cluster resources. A logical domain named WORLD is always created in the first application domain, to ease the creation and access to global application variables.

5 Evaluation

In this section we present results of some preliminary studies that should be viewed as suggestive of pCoR performance behavior and are no way conclusive. The experiments that we describe were designed to evaluate the run-time system and determine the potential gain of using shared memory, local message passing and remote message passing. The experiments were realized using two dual Pentium III workstations (733MHz) running Linux RedHat connected by Myrinet and Fast Ethernt networks.

The first experiment accounts for the time spent to synchronize two tasks that accesses a shared variable, by using mutexes and condition variables. Next, we measure the time spent to synchronize and to send data between two tasks in the same domain. Finally we evaluate the performance of task to task communication between two nodes of the cluster using both UDP/Ethernet and GM/Myrinet transport communication layers.

The picture (see figure 6) shows that **1)** the time to perform a synchronized memory transfer between two tasks in the same domain is above 25μs, independently of the message size; **2)** the time to post and get a message from the internal mailer message structure of an exchange of data (round trip) between two tasks in

the same domain, takes about 30μs . The overhead raises significantly when data content is added (1 byte - 47μs, 32 bytes - 50μs, 1kbyte - 96μs); **3)** the time to exchange data between two tasks in different nodes, over UDP/Ethernet is about 221μs for a 0 byte message, 267μs for a 32 byte message and 1201μs for a 1kbyte message; **4)** the time to exchange data between two tasks in different nodes, over GM/Myrinet is 72 μs for small messages and 107 μs for 1kbyte messages.

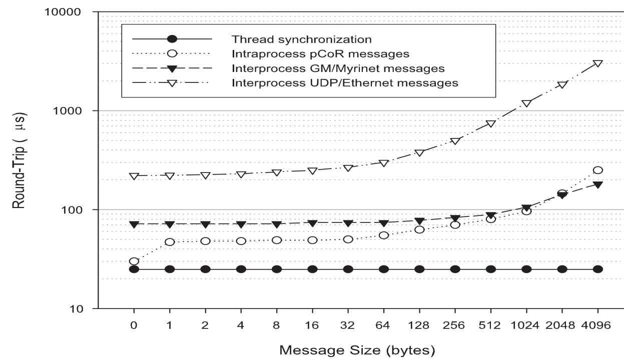

Figure 6: Performance evaluation.

6 Conclusion

We have proposed a resource oriented computing model aimed to help programmers in the task of develop large and complex parallel distributed application to run in clusters of symmetrical multiprocessors. The prototype run-time provides an integrated address space based on domains that integrates distributed multithreading and synchronization. It incorporates an autonomous sub-system[19] that allow resources to interact through a thread-safe communication library which supports the message passing style and allows for safe contexts communication. In addition, it takes advantage of existing commodity hardware/software high-performance networks like Myrinet while maintaining compatibility with UDP/TCP over Ethernet. In future work we plan to extend the basic pCoR design to incorporate the full functionality of the proposed paradigm. More significant performance can be achieved by reducing the cost of thread creation and scheduling introduced by current implementations of POSIX threads on multiprocessor. We also plan to extend the underlying communication sub-system with the goal to support Gigabit Ethernet using VIA.

References

[1] Buyya R. *High Performance Cluster Computing*. Prentive Hall PTR, 1999.

[2] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., and Sunderam V. *PVM: Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Pres, 1994.

[3] MPI Forum. MPI: A Message-Passing Interface Standard. *Internacional Journal of Supercomputer Application*, 8(3/4):165–416, 1994.

[4] Cunha J.C., Kacsuk P., and Winter S.C. *Parallel Program Development For Cluster Computing*. Nova Science Publisher Inc, 2001.

[5] Technical Committee on Operating Systems and Aplication Environments of the IEEE. *Portable Operating Systems Interface*, 1996.

[6] Gropp W., Lusk E., and Thakur R. *Using MPI-2*. The MIT Press, 1999.

[7] J. Ferrari and V. Sunderam. Tpvm: Distributed concurrent computing with lightweight processes. In *4th IEEE Int. Symposium on High Performance Dist. Computing - HPDC 95*, 1995.

[8] Zhou H. and Geist A. LPVM: A Step Towards Multithread PVM. *Concurrency: Practice and Experience*, 10(5):407–416, 1998.

[9] Chowdappa A., Skjellum A., and Doss N. Thread-safe message passing with p4 and MPI. Technical report, Computer Science D. and NSF E. R. C. Mississippi State University, 1994.

[10] Haines M., Cronk D., and Mehrotra P. On the Design of Chant: A Talking Threads Package. In *Supercomputing '94*, 1994.

[11] Briat J., Ginzburg I., and Pasin M. *Athapascan-0 User Manual*, 1998.

[12] Bhoedjang R., Rühl T., Hofman R., Langendoen K., and Bal H. Panda: A Portable Platform to Support Parallel Programming Languages. In *USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, 1993.

[13] Namyst R. and Méhaut J. PM^2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95*, 1995.

[14] Foster I., Kesselman C., and Tuecke S. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[15] Pina A. *MC^2 – Modelo de Computação Celular. Origem e Evolução*. PhD thesis, Dep. Inf., Universidade Minho, Braga, Portugal, 1997.

[16] Pina A., Fernandes J., and Machado R. Genetic regulatory mechanism by means of extended petri nets. In *IEEE International Conference on Systems, Man and Cybernetics (SCM'97)*, Hyatt, Orland, California, 1997.

[17] Beguelin A., Dongarra J., GeistA A., Manchek R., and Sunderam V. Recent Enhancements to PVM. *International Journal of Supercomputing Applications and High Performance Computing*, 1995.

[18] Pina A., Oliveira V., and Moreira C. Domains, Threads and Shared Memory in a message passing environment. Technical report, Dep. Inf., Universidade Minho, Braga, Portugal, May 1997.

[19] Pina A., Alves A., Oliveira V., and Moreira C. CoR's Faster Route over Myrinet. In *MUG '00 - First Myrinet Users Group Conference*, pages 173–17. INRIA, 2000.