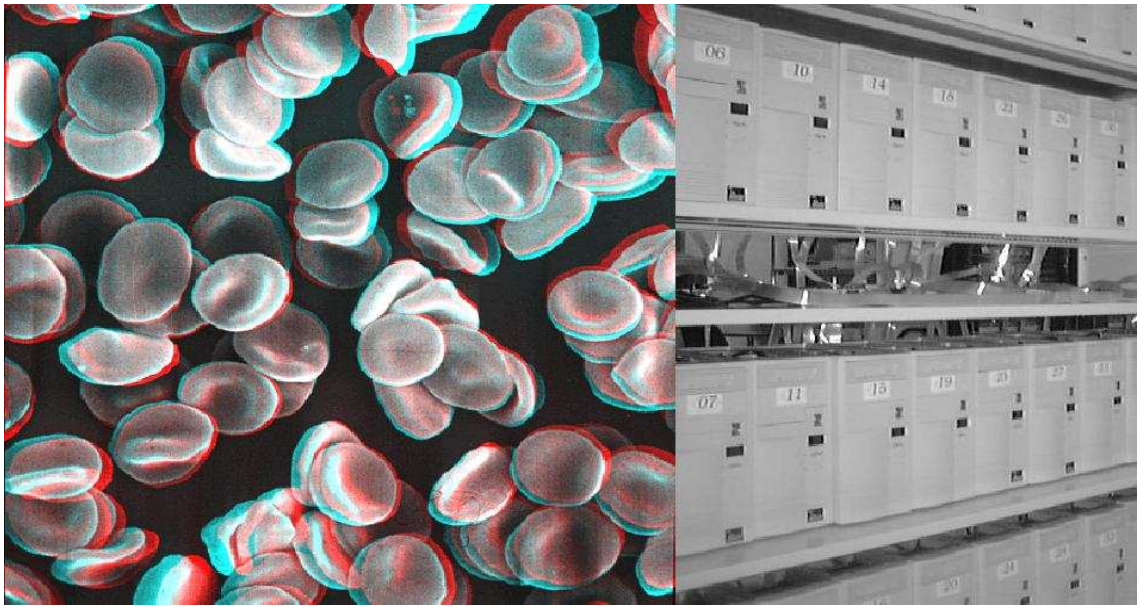
ÉCOLE
SUPÉRIEURE
D'ÉLECTRICITÉ

UNIVERSITATEA
POLITEHNICA
BUCUREŞTI

CONTRIBUTION TO PARCEL-6 PROJECT: DESIGN OF ALGORITHMS MIXING MEMORY SHARING AND MESSAGE PASSING PARADIGMS FOR DSM AND CLUSTER PROGRAMMING.



Author:

Mircea IFRIM

Supervisors:

SUPÉLEC Ph.D. Prof. Eng. **Stéphane VIALLE**

U.P.B. Ph.D. Prof. Eng. **Nicolae ŢĂPUŞ**

2005

Acknowledgements

The development of this project would have never been possible without the help and support of many people.

First I would like to express my special gratitude to my two coordinators for their support and assistance in developing this project:

Stéphane Vialle, professor and senior researcher at Supélec, for letting me experiment with ideas and for giving me great advice on how to present my findings. He offered me permanent support in my work and helped me dealing with personal problems related to my adjustement at Supélec. Our meetings and discussions always helped me to choose the right next step in the development of this project.

Nicolae Ţăpuş, professor and senior researcher, head of the Computer Science Departament at the Politehnica University of Bucharest, for his advice regarding this project and for expressing his confidence in my work. The research group around him was both challenging and resourceful, and this arouse my ambition in searching for a research career. Knowing him for the past tow years has been a great privilege to me.

Mr. **Constantin Iliescu**, professor at the Politehnica University of Bucharest, made possible the development of this project in France. His special care to all the details of my applications helped substantially towads my admission at Supélec.

I would like to express my special thanks to all my great teachers that I had at the Politehnica University of Bucharest.

I could not omit people from Supélec, Metz, who made my life easier: thank you **Mr. Claude Lhermitte**, **Patrick**, **Claudine**, **Hervé**, **Cristian Dan**, **Veronique**, **Gilles**. I would like to express my thanks to my bureau collegues for cheering me up and for their advice: **Lucian Alecu**, **Jacques Henry**, **Didier Vagner**, **Jacques Weidig**. Thanks to all my friends for their support.

Last, but not least, I would like to thank to the people I care about, all of them very encouraging and caring. Thank you **Andreea**, **Ana-Maria**, **Mrs. Viki**, **Iulian** and **Madlena**.

Abstract

The efforts of this project are concentrated on the design and the implementation of a cellular computing library for clusters and Grids: ParCeL-6.

One of the goals of ParCeL-6 is to decrease the development time for fine-grained applications. The implementation is done on coarse grained parallel and distributed architectures in order to take benefit of modern, generic and market available cheap machines. ParCeL-6 has a fine grained parallel programming model that is a successful compromise between the architecture requirements in order to reach high performances and the developer requirements to make quick developments.

Currently, there are two submodels of ParCeL-6: ParCeL-6.1 for architectures supporting memory sharing paradigm and ParCeL-6.2 for architectures supporting message passing paradigm.

We ran ParCeL-6.1 on a Linux cluster after we have installed a Distributed Shared Memory system on it. Kerrighed DSM helped us to transform this cluster in an *almost* shared memory system. The results showed that the Kerrighed DSM prefers the "embarrassingly" parallel computations applications in which cases it exhibits a very good speedup, but in the case of irregular memory acceses in the applications, a performance slowdown can be observed. ParCeL-6.1 worked on the DSM without any problems, but with bad performances.

On the other hand, ParCeL-6.2 was developed on MPI in order to have a library much more scalable. It cannot implement direct communication mode where cells can directly access to one another's output, but instead, it proposes a hybrid communication mode that try to afford the benefits of both direct, and buffered mode.

Finally, we propose two different ways of investigating. The use of an intermediate library, SSCRAP, that groups and manages efficiently MPI's communication is the first way. This library helps providing a high scalable version of ParCeL-6.2. The other way of investigating is the use of ParCeL-6.2 on SSCRAP on MPI using a DSM to take advantage of the cluster-wide shared memory that can be very useful to interactivity and sequential parts of the application.

Contents

Chapter 1

Introduction

ParCeL on Parallel Computers ParCeL is a key component of a project that has the purpose to allow smart design and implementation of complex cortical neural networks, and efficient parallel execution on multiprocessor machines. The cortical networks are implemented using a tool that takes in consideration their natural fine grained formalism, without caring about its mapping on coarse grained parallel computers. For example, all the software suite in this project can be used to control the movements of a robotic arm through cortical neural networks running on multiprocessor PCs. The plans for the future are to run this software, in an efficent and transparent way for the user, on clusters and grids.

In this project, a high level generic and parallel neural model of computation allows a smart programming based on numerous computing units. ParCeL offers an extended cellular programming model and maps the small computing units on the big processors of the parallel machines [60].

A typical ParCeL-6 program is composed of a sequential and classical program that initializes the management of a cellular network, creates a part of this network, calls some net routines, and finally removes this cellular network and its data structures. When a cell is created, a host processor is pointed out and a unique registration number is affected to this cell. This number allows to identify the cell in all the cellular network, and the cell is created directly on its host processor, stays on it and executes its computing cycle [60].

The Clusters Cluster computing has its roots in an idea that was developed in the 1960s by IBM as a way of linking large mainframes to provide a cost-effective form of commercial parallelism. The cluster computing caught the attention in the 1980s when all that was needed was available: high perfomance microprocessors, high-speed networks and standard tools for high performance distributed computing [41].

The term *cluster* is used in computer science to refer to a number of different implementations of shared computing resources. Typically, a cluster integrates the resources of two or more computing devices (that could otherwise function separately) together for some common purpose.

High performance clusters started back in 1994 when Donald Becker and Thomas Sterling built a cluster for NASA [57]. This cluster was made up of 16 Intel 486 DX4 processors connected by 10 Mbit Ethernet, and they named it Beowulf. The Beowulf Project has been joined by other projects trying to provide useful solutions to turning commercial hardware into clusters capable of supercomputer speed. These clusters have been used for everything

from simple data mining, file serving, database serving, or web serving, to flight simulation, computer graphics rendering, or weather modelling [57].

Grids and Clusters Altogether In 1999, Ian Foster and Carl Kesselman attempted to give a definition for grid in the book *The Grid: Blueprint for a Future Computing Infrastructure* - more details in [37]:
A computational grid is a hardware and software infrastructure that provides dependable, consistent, universal, and inexpensive access to high-end computational capabilities.[37]

In the article *The Anatomy of the Grid*, in 2000, signed Ian Foster and Steve Tuecke, another aspect of grid was emphasized - the social and political importance: Grid computing is concerned with *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations* [39].

Ian Foster suggests that the essence of the Grid definitions can be concentrated in one three point list [40]. The Grid is a system that:

- integrates and coordinates resources and users that live within different control domains - different administrative units of the same company, different companies - and addresses the issues of security, policy, payment, membership that arise in these settings;
- is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery, and resource access. It is important that these interfaces be standard and open. Otherwise, the system will be application specific.
- allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating to response time, throughput availability, and security, and/or co-allocation of multiple resources types to meet complex user demands, so that the utility of the combined system is significantly greater than that of the sum of its parts.

Basically, in a grid all the resources can be scattered across the globe and are being managed at the resource itself while in a cluster all the resources are physically near each other and the resources are managed centrally.

SMP, cluster and Grid architectures allow distribution / parallelization of tasks, but support different granularity. Fine grained applications are appropriate for SMP architectures. Coarse grained applications can run on all the three architectures mentioned before. Grid allows to run bigger applications than on one SMP, one cluster, by grouping resources (memory, cpus, etc.). In the Figure 1.1, there can be observed the relation between clusters and grids.

A problem encountered both in grids and clusters is the mapping of applications to parallel machines in a manner that balances the load while minimizing communication. Transparency and efficiency in mapping data to processors is one of the main purposes of ParCeL-6 on both heterogeneous and homogeneous parallel machines and this can be achived only through a solid process of experimentation and implementation.

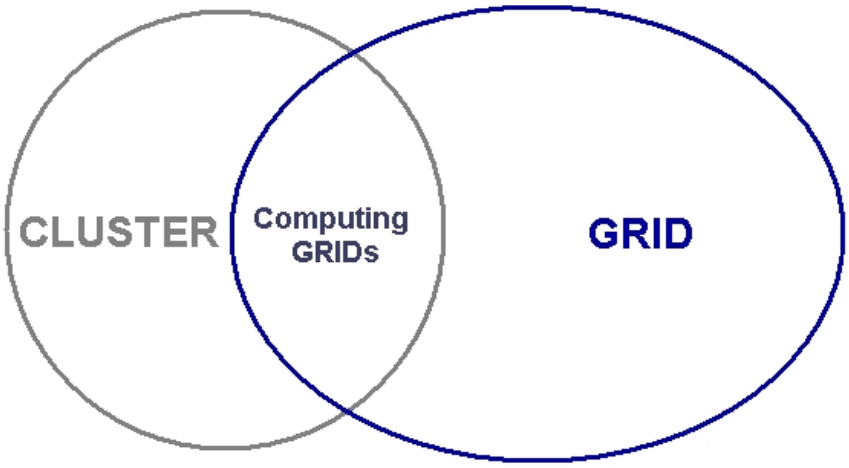


Figure 1.1: Both Grids and Clusters can be used for solving big amounts of computations

Chapter 2

Parallel and Distributed Cellular Languages

2.1 ParCeL Project

The languages of the ParCeL family (*PAR*allel *CE*llular *L*anguages) are destined to provide natural and fast implementation of distributed, cellular oriented algorithms on parallel machines, letting the user to declare and to program the cells, a kind of processing elements. The main purpose of these cellular languages is to efficiently map the fine grained application architectures to the coarse grained architecture of the contemporary computational machines.

The first language (ParCeL-0/MCV) was born around 1989 as a result of the colaboration between Supélec and CRIN (Centre de Recherche en Informatique de Nancy) [12].

Then, a new version (ParCeL-1) greatly improved and optimized appeared around 1993-94 due to the colaboration of Supélec and the University of Orsay-Paris-XI [18]. ParCeL-1 was a tool adapted to artificial intelligence applications and provided transparency on basic parallel architectures. The main objectives of this cellular language were:

- ability to execute on distributed memory architectures
- ability to hide from the programmer most of the hardware constraints
- ability to implement distributed artificial intelligence and neural networks applications

The begining of the ParCeL project and the evolutions it gave birth to, can be observed in the Figure 2.1. The current versions of ParCeL had to be fast adapted to the development of hardware architectures - the law of Moore says that the computers become about twice more powerful at every 18 months, and also to the applications that were mainly specific science fields oriented. For example, the neural networks have a natural parallelism that is not appropriate for the machine parallelism when speaking about granularity and programming paradigms. The implementation of neural networks needed a tool to make possible the adaptation of the neuronal paradigm to the paradigm of machines. The evolutions of ParCeL-1 followed three main directions [47]:

- the need for a scientific computation cellular language gave birth to ParCeL-2 (developped at École Polytechnique Fédérale de Lausanne).

- the multi-agent systems needed to fast adapt the features of the new architectures to their needs with the help of ParCeL-3 [48] and ParCeL-5 [49].

- a direction of development was towards connectionists systems of biological inspiration and this gave birth to ParCeL-4/Hibs [43].

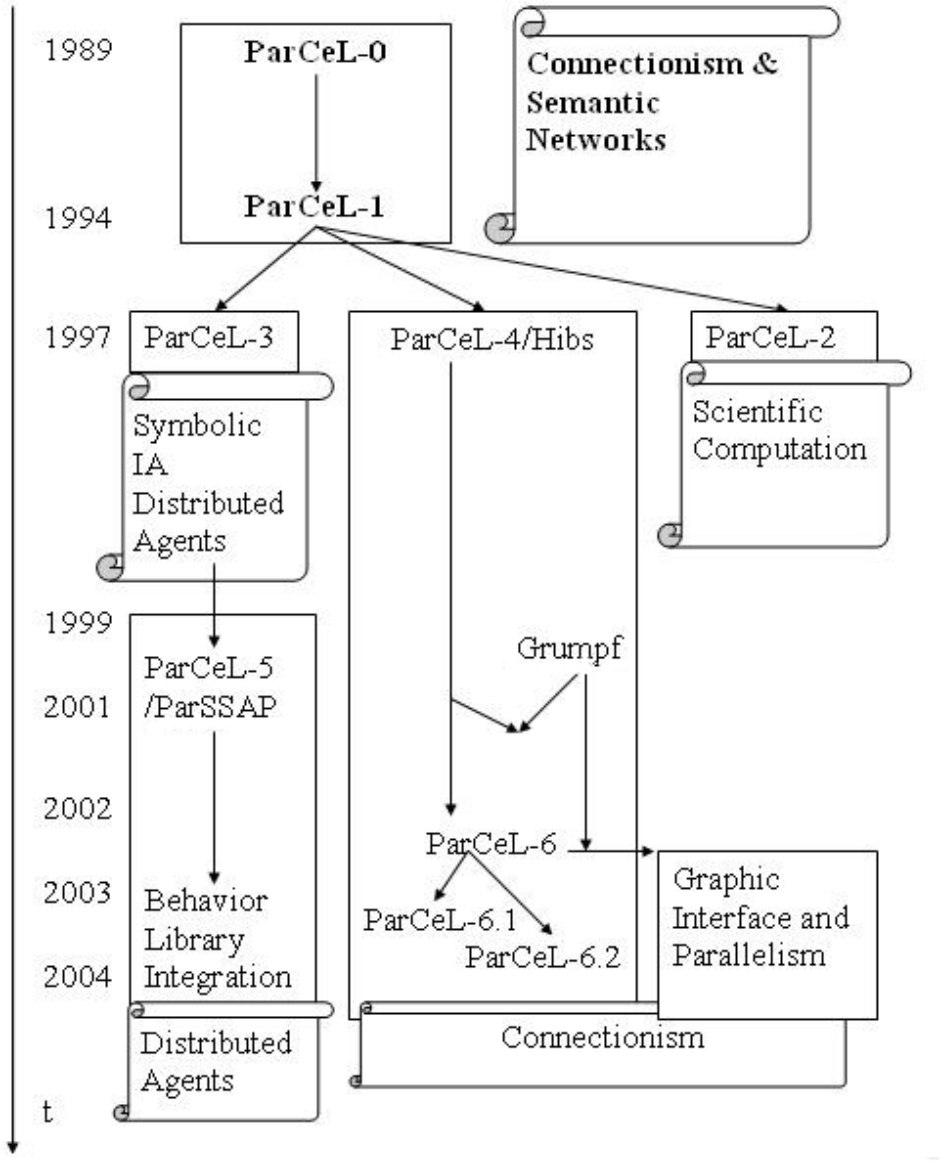


Figure 2.1: General view of ParCel project

In ParCeL-5 - that implements the improved model for multi-agent systems, the cells followed a running cycle that consisted in several steps. These steps followed the interaction model found in the multi-agent systems:

- parallel initialisation step (environment, resources, percepts)

- activation of agent behaviours
- declaration of action intentions
- resolution of agent conflicts (by the arbitrator)
- execution of actions of winners and non-conflicting agents
- execution of parallel or sequential end-cycle function (user)
- update of the environment and agent percepts

With ParCeL-4/Hibs, another evolution of ParCeL project have been towards connectionists systems. In the Figure 2.1 can be observed the previous evolution steps of ParCeL project. ParCeL-4/Hibs has been enriched and evolved giving birth to ParCeL-6. ParCeL-6 is a parallel cellular language designed for complex neural networks and some physical system simulation (based on local equations). Currently, two versions of ParCeL-6 are available: ParCeL-6.1 which is designed for architectures that support memory sharing paradigm and ParCeL-6.2, designed for architectures that support only message passing paradigm. The purpose of ParCeL-6 is mainly inherited from ParCeL-4 and followed the next directions:

- a programming model adapted to neural computation for a quick and natural implementation
- an implementation adapted to coarse grained architectures with few tasks and grouped communications

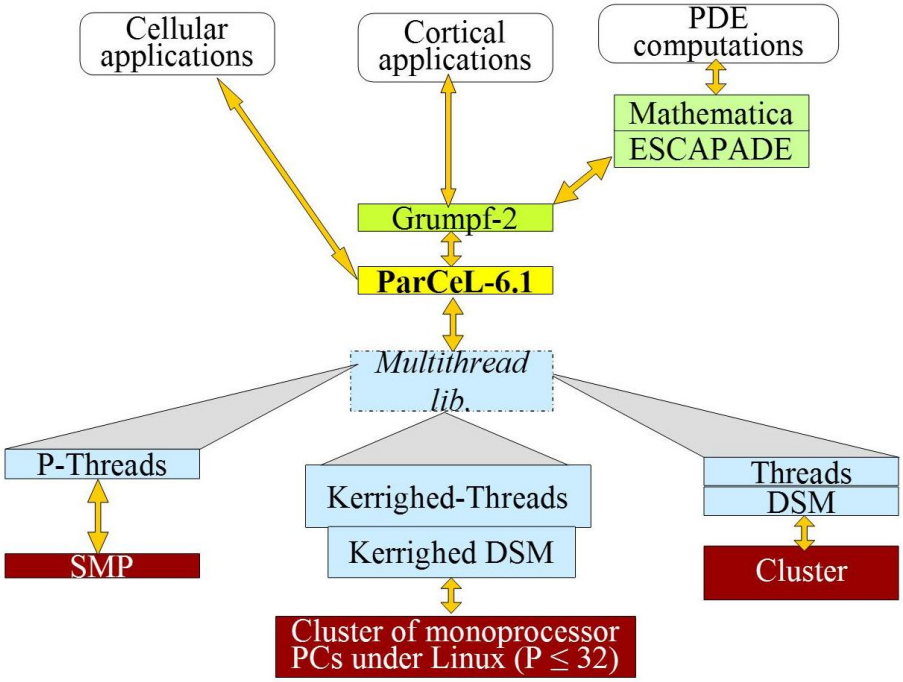


Figure 2.2: ParCeL-6.1 Utilization

ParCeL-6.1 exhibited very good performances on symetrical multiprocessor machines (SMP) due to its cheap and basic mechanisms of data sharing. The high level programming model of ParCeL-6.1 provides a semantics that allows natural and fast development of cellular programs. The programmer can experiment and choose the cell communication mode that is more appropriate for his application from three available modes.

Grumpf is a client-server based environment for developping fine grained applications. The client provides some tools for calcul visualization. The server has fine grained programming model and it makes no compromise with the coarse grained architectures. This is the reason it needs ParCeL-6 in order to adapt its computation units to the machine architectures. Grumpf was developed at Supélec by Hérve Frezza-Buet [60].

Escapade is a tool used to solve Problems with Differential Equations expressed in *Mathematica*. The problems with differential equations are solved calling fine grained Grumpf computations. It is current under development at Supélec with the colaboration of Mops Laboratory.

ParCeL-6.2 had the purpose to increase the scale when running ParCeL-6 programms, therefore it was implemented using MPI and had as target hardware architectures large distributed machines. The way ParCeL-6.2 is used can be observed in figure 2.3.

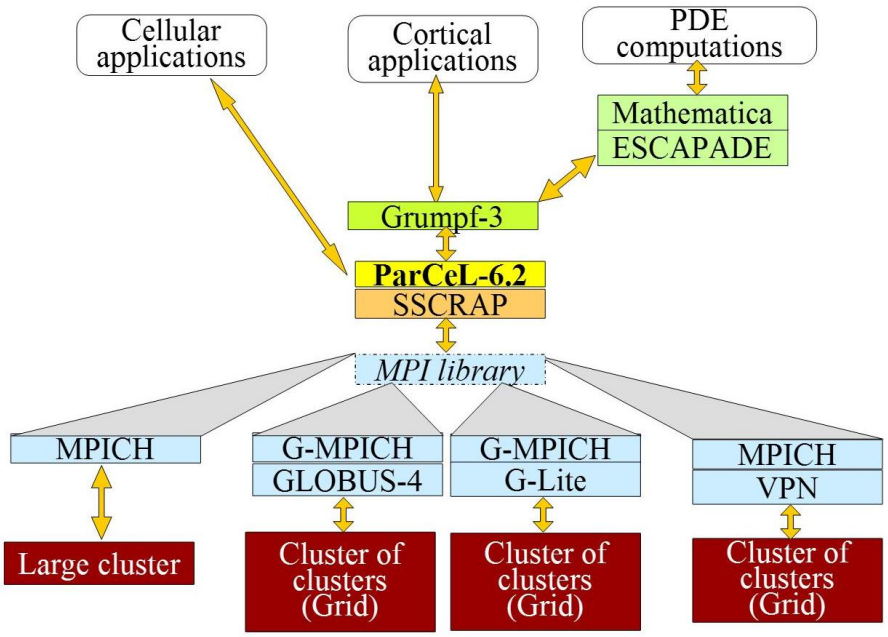


Figure 2.3: ParCeL-6.2 Utilization

2.2 Current cellular languages

The CA model has been proposed in the 1950s by John von Neumann [1]. Its main purpose was to simulate complex dynamical processes using a natural inspired parallelism. The last two decades offered a proper environment for development of several models of cellular automata different from the original one proposed by J. von Neumann. All these models try to simulate more and more complex real-world systems and phenomena in fields like chemistry, biology

and disjunctive domains having in common the need for computational power (see [51] and [50]).

The cellular languages are the programming model that follows the computational model of cellular automata. A general impression on the software development of the CA is that the encountered problems consist in adapting the fine-grained parallelism of the applications to the coarse-grained parallelism of the general-purpose machines.

To achieve high performance in the implementation of CA there are two possible alternatives. The first one is the design of special hardware devoted to the execution of CA - it offers speed but not flexibility and it is very expensive. The second alternative is based on the use of commercially available and coarse grained parallel computers - less expensive, but not adapted to fine grained applications.

The most significant example of a specialized hardware which has been designed to run CA simulations is *CAM (Cellular Automata Machine)* - for details see [32]. Although the CAM offers a high-level environment for programming CA and can run CA simulations in an efficient way, it is limited in the size of the automata which can be simulated and in the number of states per cell. Furthermore, it is a specialized machine which cannot be utilized as a general purpose computer.

In order to use commercially available and general-purpose parallel computers, the point of interest need to be moved towards software research. Several parallel cellular environments have been developed in the latest years. Significant examples of these parallel cellular environments are CAMEL/CARPET - [58], CANL - [59], CAPOW - [33], CaSim - [38], CAT/CARP - [22], CDL - [24], CDM/SLANG - [55], Celip - [15], CelLab - [11], Cellas/Fundef - [16], Cellsim - [2], Cellular/Cellang - [52], CEPROL - [5], DDLab - [31], Flexica - [54], Hical, LCAU - [6], Mathematica, Scamper, SCARLET - [23], Sicela, WinCA - [4], NEMO (Neighbourhood Modelling), CAPE - [17].

About CARPET we can say that it is a high-level programming language based on the biology-inspired cellular automata theory. It offers support for the development of parallel high-performance software with transparent access to the parallel architecture on which programs run. CARPET was used in implementing natural solvers of real-world complex problems, such as forest fire, circuitry simulations, landslide simulation, lava flow models, freeway traffic simulation, image processing and generic algorithms. A portable MPI-based implementation of CARPET and CAMEL (called *CAMELot*) [46] has been implemented on MIMD parallel computers.

Fine-grained mapping is not only a MIMD/cluster problem, but it is a research topic in the domain of embedded systems [61]. These systems require control of many concurrent real-time activities, leading to system designs which feature multiple hardware peripherals with each providing a specific, dedicated service. The peripherals may increase system size, cost, weight, power consumption and design time. A solution to avoid such problems is to use a fine-grained software mapping. Software thread integration provides low-cost concurrency on general-purpose processors by automatically interleaving multiple threads of control into one. This simplifies hardware to software migration [61]. The integration of multiple threads into a single flow of control which executes on a standard uniprocessor is an efficient way to perform hardware to software migration, that leads to avoidance of high hardware costs [61].

Another kind of problems may be encountered when applications such as fully descentralized multi-agent systems need to be mapped on clusters. A solution may be to group agents with similar objectives or data - as it is done in the traditional clustering, but with the constraint that agents must remain in place on a network, instead of first being collected into

a centralized database. The agents may be at the beginning scattered all over the network and have them search in a peer-to-peer fashion for other similar agents. The network traffic with information and the meaning of the exchanged data must be highly corelated in order to obtain performances when running such a system on a cluster.

There also appeared a means of mapping applications in a fine-grained way named *Lattice Gas* [42]. This could be done by using a parallel library. The lattice gas technique is a fast developing numerical tool aimed at simulating, on a computer, various physical phenomena such as complex fluid flows, reaction-diffusion systems or wave propagation processes. In fact, the Lattice Gas systems consist of cellular automata models that are viewed as a serious competitors to traditional computational fluid dynamics techniques. The entity in this model can be an object that must be easy adaptable and must provide reusability. Also the parallelization schemes must be general.

As we can deduce, fine-grained process modelling can be used as an aid to software development. Two levels of granularity may be used in order to do this: one at the level of the individual developer and another at the level of the representation scheme used by that developer. Modelling the software development process at these levels includes some advantages like:

- the production of models that better reflect actual development processes because they are oriented towards the *actors* that *play* these processes
- models that are vehicles for providing *guidance* because they may be expressed in terms of the actual representation schemes employed by those actors.

According to [30], three main aspects can be distinguished regarding the different existing software environments:

- The systems they are intended for
- What do the interfaces to the user and to the machine look like (textual or graphical)
- What features concerning CA are present or missing

It seems that most systems are intended for doing step by step forward simulations of specified CA starting with specified initial configurations in order to find out what the configuration will look after a certain number of steps.

For the simulation of CA on a computer there are at least two basic possibilities. One is to offer a library of routines to be used with a general purpose language. It has the advantage that the user does not have to learn new programming language syntax, but only the names and the parameters of the methods in the library. One possible disadvantage of this approach is that the compiler does not know anything about the special application and hence cannot do any specific optimizations. This can be overcome by the second approach which is to offer a special *CA programming language*.

To simulate a CA on a computer, its main characteristics must be known. In the next section, the general CA features will be introduced to the reader.

2.3 The General Features of a Cellular Automata

2.3.1 Lattice

The structure of a lattice is in most of the programming environments a two dimensional one because in ordinary cases the coordinates to specify a cell consist in two components. Size of the lattice is only bounded by the amount of available free memory for the majority of the programming environments. At least Cellular does some optimizations in the case where the side lengths of the grid are a power of two. CANL requires the grid to be a square [30].

2.3.2 Neighborhood

Most packages do not impose any restrictions on the neighborhood used. Often a coordinate-like notation for referencing relative neighbors is available and the neighborhood is given implicitly by the collection of all relative neighbors referenced. At least CDL allows the (relative) coordinates of neighbors to be computed by the local transition rule and to be stored in a structure. This may simplify the formulation of some algorithms but makes the determination of the neighborhood more difficult [30].

2.3.3 Set of States

In several systems, the size of the set of states is restricted, but in the most cases it is allowed an arbitrary number of states.

If the set of states becomes too large it obviously becomes important to be able to speak about its structure and to have a good notation. Often the memory of a cell can be subdivided into registers which can store values of different data types. This includes numerical as well as enumeration types. Hical allows to distinguish between integers and natural numbers (and to specify the exact number of bits to be used). Cellang and CDL allow integer subtypes of arbitrary ranges. CANL, CARPET, CDL and Hical offer at least one floating point type [30]. Cellang and CDL and also offer the possibility to speak about arrays of values.

2.3.4 Transition Function

In Carpet, CamSim, CANL, CaSim, CAT, CDL, cellsim, Cellang, Ceprol, and Hical the local transition rule is specified in imperative languages offering some usual operators for building expressions and the usual control structures. Furthermore many of them allow the definition of (auxiliary) functions which may be called. In Fundef, SDL and some other languages the local transition rule is essentially specified as a set of rules. These may contain *don't cares* in the place of cells which have no influence on the resulting state in the current local configuration. CDM offers the possibility to update cells asynchronously. Most languages allowing the formulation of probabilistic rules do this via calls to a *random* function (CANL, CARPET, CDL, Cellang) [30].

Chapter 3

DSM systems

This chapter aims at introducing DSM systems to the reader. There are presented research aspects and a brief history of the DSM systems.

3.1 Introduction to DSM systems

3.1.1 The need of having a DSM

Users often use in their processes shared data. The access to this data can be easily obtained by interprocess communication. Sometimes this is not enough - for an user it would be much easier to have transparent access to shared data like programming variables among several processors. This is the main reason the DSMs appeared.

The main goal of a DSM system is to make interprocess communications transparent to end-users. The implementations were software, hardware and hybrid.

3.1.2 What is a DSM system?

A DSM system exploits heterogenous computational machines in a transparent way to the user, as if they were a single and much powerful virtual machine. There are two approaches on a DSM: the **Shared Virtual Memory Model** and the **Object DSM** model.

The first model is similar to the paged virtual memory implemented in mono-processor systems, except that all the distributed memories must be groupped together into a single wide address space. The drawbacks of this approach are given by the fixed size of the pages. The granularity of the shared data is fixed to the page size whatever the type and the actual size of the data.

In the second model - Object DSM, all the shared data are seen as shared objects, variables with access functions. The user has only to define which data objects are shared. The DSM system will take care of the management of the shared objects at creation, modification and access.

When implementing a DSM system, there are many aspects that must be considered such as data location, data access, sharing and locking of data, data coherence.

3.1.3 The features of a DSM

A DSM implies a mechanism, an architecture, an algorithm, a way of managing the system, a data coherence policy and a memory consistency model.

The mechanism of a DSM system can be achieved in software (library, compiler, os/user-space, os/kernel-space), in hardware or it can be a mixture of these two (hybrid).

The Architecture is given by the configuration of the cluster, by the organization of the shared data (non-structured or structured as types or objects) and by granularity of coherence maintenance (word, block, page, object, segment). When **the algorithm** of the DSM is implemented, there must be considered the way the data is accesed. The access on the shared data can be SRSW (Single Reader - Single Writer), MRSW (Multiple Readers - Single Writer) or MRMW (Multiple Readers - Multiple Writers). There also must be determined a way to realize **the Management** of the DSM system; we can have centralized, distributed/fixed or distributed/dynamic management.

There can also must be found a way to maintain the data coherence (**Coherence** means that a value returned by a read operation is the one expected by a programmer, the value of the latest write operation.) - **The Coherence Policy**, for example WI (Write Invalidate) or WU (Write Update). In **Write-Invalidate**, a processor invalidates all other cached copies of shared data and can then update its own copy without further bus operations. The number of write runs observed and not their lengths gives the cost in bus cycles. In **Write-Update**, a processor broadcasts updates to shared data to other caches, maintaing the coherence. Some shared blocks may become unshared during the duration of a write run due to replacement. The updates will therefore be broadcasted only when the data is actively shared.

The Memory Consistency Model can be restricted (strict or sequential consistency) or relaxed (processor, weak, release, lazy release, entry consistency).

The Strict Consistency says that a read returns the most recently written value and requires total ordering of requests which implies significant overhead for mechanisms such as synchronization. Strict consistency isn't always needed.

Sequential Consistency demands that the result of any execution of the operations of all processors is the same as if they were executed in a sequential order and in the same time the operations of each processor appear in this sequence, in the order specified by the program.

In **General Consistency** the condition that must be respected is that all copies of a memory location have the same data after all writes of every processor is over.

Writes issued by a processor are observed in the same order in which they were issued in the **Processor Consistency** model, but ordering among any 2 processors may be different.

The **Weak Consistency** says that synchronization operations are guaranteed to be sequentially consistent (critical regions). To access the shared data there are used the synchronization/mutual exclusion techniques which must be properly used by the programmer.

In **Release Consistency** synchronization accesses are only processor consistent with respect to each other.

3.1.4 The Beginning

IVY(Integreted Shared Virtual Memory At Yale) is one of the first DSM runtime systems. It was implemented by Kai Li and Paul Hudak around 1988 and provides the abstraction of two classes of memory: private and shared. More details about IVY can be found in [7] and

[9]. Using as a starting point the operating system Aegis, the implementation of IVY was made as a set of procedures written in the Apollo DOMAIN Pascal. IVY was installed on a token ring network of Apollo workstations.

The coherence policy used in IVY is the write invalidate update protocol and the algorithm implements multiple reader - single writer semantics. To detect the access to shared memory locations there are used the virtual memory primitives. A page has 1Kbyte. Pages can be in the read-only, write, or nil modes. The write-invalidate protocol maintains consistency by invalidating all the read-only copies of a page before allowing a processor to write to that page. The first read access and the write accesses to a shared page cause page faults; there is a page fault handler that acquires the page from the current holder. IVY provides a strictly consistent memory model. There are integrated three page management policies into IVY:

- centralized manager scheme
- fixed distributed manager scheme
- dynamic distributed manager scheme

In IVY, there is an efficiency problem that is inherited in all the three implementations: successive read and write accesses to a page on a single node cause the page to be transferred twice (double fault problem). The problem was solved using sequence numbers for every shared page.

The synchronization to serialize the concurrent accesses to shared memory locations is implemented using eventcounts - atomic operations on shared counters which are implemented through the system's shared memory semantics.

Mirage[10] extends the IVY mechanisms to avoid page trahshing if two processors reference a single page repeatedly. It was introduced a time interval in which the ownership of the page will not be forwarded to another processor; during this interval a page is pinned to a certain processor.

3.2 Data Management in DSM systems

There are four most important **management algorithms** for DSM systems: Read-remote-Write-remote, Read-migrate-Write-migrate, Read-replicate-Write-migrate, Read-replicate-Write-replicate.

When using a **Migration Algorithm**, data is shipped in the requesting node, allowing subsequent accesses to be done locally; in fact, the whole block/ page migrates to help access to other data. A migration algorithm is susceptible to trashing: a page can migrate between nodes while serving only a few requests. In the Mirage system, we can find a solution to this problem: a page will migrate only after a minimum number of accesses have occured or it has ramained a minimum duration in the local memory of a machine. The migration algorithm can be combined with virtual memory; for example: if a page fault occurs, the memory map table must be checked; if map table points to a remote page, the page will migrate before mapping it to the address space of the requesting process. Locating the remote page can be done by using a server that tracks the page locations, by using some records maintained in nodes that can direct the search for a page towards the node holding the page or by broadcast a query to locate a page.

The Read Replication Algorithm extends the migration algorithm by replicating the data at multiple nodes for read access. If a write operation is requested, all the copies at various nodes of the shared data will be invalidated. There must be maintained an evidence with the location of all the copies of the shared data. The cost of the write operations is higher than the cost of the read operations.

Considering the **Software Based Replication / Migration Management**, there are two problems that must be solved: finding the current owner of a page in the case of migration and finding the set of processors with a copy of the page in the case of replication.

The Full Replication Algorithm is an extension of read-replication algorithm - it allows multiple sites to have both read and write access to shared data blocks. There must be used a mechanism for maintaing consistency, a gap-free sequencer that assigns a sequence number to the request and multicasts it to all nodes that have a copy of the shared data. The receiving nodes will process the requests in order of sequence numbers. Missing requests (gap in sequence numbers) will be reported to sequencer for retransmission.

In the **Read-remote-Write-remote** algorithm there is one or multiple servers which serve the memory pages. This algorithm is easy to implement with the advantage of providing sequential consistency when implemented with a single server which serializes the request and response services, but it has some drawbacks: the servers can become potential bottlenecks.

Read-migrate-Write-migrate implements the migration of a page to the new processors memory upon access. Its main features are: the coherence problem doesn‘t need to be handled separately and exploits program localities well. It has also some drawbacks:

- when a page migrates, all processors sharing the page need to update their virtual page to physical block mapping
- the ping-pong effect

Read-replicate-Write-migrate is a popular algorithm for DSMs because it performs well when reads are dominant operation and many software use the multiple-read/exclusive-write semantic. It maintains strong consistency and the write operations can be costly inefficient when unproper hardware is used. **Read-replicate-Write-replicate** is an algorithm that performs well when reads are not dominant operation and is very important to have hardware support in order the write operations to be costly efficient. To maintain strong consistency, it is necessary to use an aditional protocol.

3.3 Munin

In 1990, a new improved DSM software system appeared - details in [13] and [14]. Munin was implemented at Rice University in Houston, Texas and was based on a shared memory parallel programming environment. It was implemented on an Ethernet network of SUN workstations.

It is the first runtime system for distributed memory machines that uses loosely coherent memory. It has multiple, type-specific coherence mechanisms and among these mechanisms there is a new coherence mechanism: delayed updates. The support for variable-size shared data items is implemented.

Munin was created for programmers that write parallel programs using threads, as they would on a shared memory multiprocessor. The library routines **CreateThread()** and **DestroyThread()** are provided.

The way to handle the memory faults is similar to the one found in virtual memory system; on a memory fault the faulting thread is suspended, the associated server to handle the fault is invoked, the type of the data object is determined, the type-specific fault handler is invoked and after handling the fault, the suspended thread is resumed. In Munin there are multiple shared object classes.

The simplest form of shared data are the **Read-Only objects**. Once they are created, they can‘t be modified. If a thread attempts to write to a read-only object, an error will be generated.

A thread can perform the multiple accesses it needs to a **Migratory object** and only after that other thread can gain access to that object. To maintain the consistency for the migratory objects, before the object will migrate to the next thread, the original copy is invalidated.

If there are many threads that need a concurrent write access to an object, then a **Write-Shared object** must be used. In this case, there is no need for the writes to be synchronized because the updates will modify different words in the object. When two variables reside in the same consistency unit, such as a virtual memory page, we say that we have **false sharing**.

The objects that are written by one thread and read by other one or more threads, are called **Producer-Consumer object**. To maintain the consistency of those objects, the object will be first replicated and then updated when the producer modifies it. In this way the read misses of the consumer threads are avoided. The producer‘s updates will be buffered until the producer releases the lock that protects the object, so that all of the changes can be passed to the consumer threads in a single message.

There are also **Reduction objects** implemented using the fixed-owner protocol. The operations on a such kind of object are equivalent to a lock acquisition, a read followed by a write of the object, and a lock release. Such an object can be used when computing the global minimum in a parallel minimum path algorithm.

When we have an object that is alternately modified in parallel by multiple threads, and after this follows a phase in which a single thread accesses them exclusively, we say we have a **Result object**. In this case, an efficiency problem can appear: when the multiple threads complete their execution, they unnecessarily update the other copies; the updates should be send only to the thread that requires exclusive access.

There can be objects that are replicated on demand and are kept consistent by requiring a writer to be the single owner before it can modify the object - **Conventional objects**. Upon a write miss, an invalidation message is transmitted to all other replicas and the thread that generated the miss is blocked until it has the only copy in the system. A conventional object is a shared object with no annotation provided by the programmer.

When **programming** in Munin, the logical connections between shared variables and the synchronization objects that protect them can be specified by the user call **AssociateDataAndSync()**. If access to a particular object is protected by a particular lock, such as an object accessed only inside a critical region, Munin sends the new value of the object in the message that is used to pass lock ownership, by avoiding some access misses.

PhaseChange() is a routine that purges the accumulated sharing relationship information. This call is useful for problems in which the sharing relationships are stable for long periods of time between problem redistribution phases.

ChangeAnnotation() modifies the expected sharing pattern of a variable and consequently the protocol that is used to keep it consistent - the system will adapt to dynamic

changes. Since the sharing pattern of an object is an indication to the system of the consistency protocol that should be used to mantain consistency, the invocation of this routine may require the system to perform some immediate work to bring the current state of the object up-to-date with its new sharing pattern.

Invalidate() deletes the local copy of an object, and migrates it elsewhere, if it is the only representation of the object, or updates remote copies with any changes that may have occured.

Flush() advises Munin to flush any buffered writes immediately rather than waiting for a release.

SingleObject() advises Munin to treat a multi-page variable as a single object rather than breaking it into smaller page-sized objects.

PreAcquire() routine is used to acquire a local copy of a particular object in anticipation of future use, and consequently avoiding the latency caused by subsequent read mises.

Munin executes a distributed directory-based cache consistency protocol in software, in which each directory entry correponds to a single object. When starting an application program, the Munin root thread starts running. The shared data segment will be initialized, the worker threads to handle consistency and synchronization functions are created, and the root thread registers itself with the kernel as the page fault handler of address space. Then *user_init()* routine must be executed in order to initialize the user‘s environment. If a user thread has an access miss or executes a synchronization operation, the root thread will be invoked and it manages the fault. After that, the user thread resumes. **Object directory entries** contain the fields:

- **Start address** and **Size** which are used as the key for looking up the object‘s directory entry in a hash table, given an address within the object.
- **Protocol parameter bits** that represent the parameters for the consistency protocol.
- **Object state bits** characterize the dynamic state of the object, whether the local copy is valid, writable, or modified since the last flush, and wheather a remote copy of the object exists.
- **Copyset** is used to specify which remote processors have copies of the object that must be updated or invalidated - a bitmap can be enough to do this.
- **Synchq** (optional) is a pointer to the synchronization object that controls access to the object.
- **Probable owner** (optional) is used to reduce the overhead of determining the identity of the Munin node that currently owns the object. The owner‘s identity is used by the ownership-based protocols (migratory, conventional and reduction), and is also used when an object is locked in place (reduction) or when the changes to the object should be flushed only to its owner (result).
- **Home node**(optional) is the node at which the object was created. It is used for a few record keeping functions and as the node of last resort if the system ever attempts to invalidate all remote copies of an object.
- **Access control semaphore** provides mutually exclusive access to the object‘s directory entry.

- **Links** used for hashing and enqueueing the object's directory entry.

The **Delayed Update Queue** (DUQ) is used to store the pending outgoing write operations in order to provide release consistency. A write to an object that allows delayed updates, as specified by the protocol parameter bits, is stored in the DUQ. The DUQ will be flushed everytime a local thread releases a lock or arrives at a barrier.

The Synchronization is not provided through shared memory, but through node interaction - each node interact with the other nodes. Munin provides support for distributed locks and barriers. A queue-based implementation of locks is used to allow a thread to request ownership of a lock and then to block awaiting a reply without repeated queries. This DSM uses a synchronization object directory to maintain information about the state of the synchronization objects. A queue identifies for each lock the user threads waiting for the lock, so a release-acquire pair can be performed with a single message exchange if the acquire is pending when the release occurs. To improve scalability, the queue itself is distributed.

The performance of Munin is an agreement between the resulting reduction in program complexity and the execution time of the applications. Munin achieved performance within 5-10 percent of message passing implementations of the same aplications. To improve its performances, it was tried to design higher-level interfaces to distributed shared memory in which the access patterns will be determined without user annotation. The scalability in terms of processor speed, interconnect bandwith, and the number of processors was also an important issue in Munin.

3.4 Other DSM systems

3.4.1 Parallel Virtual Machine - not a DSM

PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource [20]. The individual computers may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations, that may be interconnected by a variety of networks, such as ethernet or FDDI. PVM support software executes on each machine in a user-configurable pool, and presents a unified, general, and powerful computational environment of concurrent applications.

3.4.2 The implementation of Adsmith

The development of the networks of workstations had a result in the domain of the DSM systems. It was needed a low-cost, efficient and portable DSM to use the resources of the networks of workstations (NOW). At this moment Adsmith was implemented. It is build on top of the PVM and it presents to the programmers as a user-level library in C++ [21].

This DSM is an object-based DSM and provides primitives to create and allocate shared objects, accesses to shared objects and operations to synchronize among processes. To improve performance, Adsmith was provided with support for release memory consistency model, different coherence protocols, load/store memory accesses, object-based multiple writer protocol, bulk transfer, prefetch, nonblocking store and other specialized accesses.

3.4.3 Phosphorus

Phosphorus is, like Adsmith, a DSM developed on top of the PVM. The goals of its implementation were [26]:

- to provide a platform to experiment with various features of DSM systems
- educational purposes (for students)
- to use new data sharing features

Because it is built on PVM, this DSM was thought to follow the architecture of the PVM (simple interfaces, portability) and to evolve as PVM evolves.

Phosphorus is comprised of a daemon(phosd) and library functions as PVM is. In order to make use of the shared data, a daemon resides on each machine. This daemon has the role of a shared memory manager who is in charge of keeping the shared data coherent. The sharing unit is the variable. The types supported by PVM through the packing/unpacking functions are also supported by Phosphorus [26]. The programmer can declare shared arrays of these types. The management of the shared variables is distributed among a collection of servers running on the various hosts. A shared variable has a server owner and the ownership can change dynamically. In order to keep simple and efficient access to the virtual address space, Phosphorus was designed to reduce the network traffic used to maintain data coherent. Because there are four different variable access behaviours, four sharing data protocols were implemented:

- Read Only (Multiple Readers/Write once)
- Migratory (Single Reader/ Single Writer)
- Conventional (Multiple Readers/Single Write)
- Write Shared (Multiple Readers/Multiple Writers)

The access interface consists in a simple set of primitives for declaring, reading, writing and synchronizing accesses to shared variables.

3.4.4 Mermera

The Memory Behaviour in Mermera combines the behaviours of Coherent Memory, Pipelined RAM, Slow Memory and Locally Consistent Memory - details in [19]. The memory model is characterized by the fact that the processes - they may be running on different processors, share a region of their address space.

Coherent Memory specifies that all the processes agree on the order of all writes that is consistent with their individual program orderings. In other words, if a process observes some writes in a certain order then no other process observes those writes in a different order.

Pipelined RAM (PRAM) says that the order of all writes by the same process is respected by all processes. For example if a process makes two write operations, then no other process can read them in the reverse order.

Slow memory specifies that when a process has more writes to the same location, those writes must be ordered in all the processes in the order they were written.

Locally Consistent Memory is a consistent memory model much weaker than sequentially consistent memory in which all events appear to be executed on a single processor in an order consistent with the program of *every* process. In the model of locally consistent memory, it appears to *each* process that all the events it observes are executed on a single processor in an order consistent with its program.

The Performance in Mermera was analyzed considering the fact that its implementation mixes coherence with non-coherence [19]. The non-coherent writes have completion times that are about 20-40 times smaller than coherent write. The **completion time** includes the time spent in doing the asynchronous broadcasts and the time spent in executing tasks that are necessary to process incoming updates. For a given number of processes, the completion time grows at a rate sub-linear in the number of messages sent. The explanation to this fact is: the number of messages sent and received is a significant determining factor for completion time and this number depends on the buffer size and on the number of processes. If we have small buffer sizes, then a large number of small messages will be generated. To have a more efficeient transmission, these small messages should be grouped together in larger messages.

The Buffer Size has e significant effect on the performance. If we have a too small buffer size, the frequency at which messages are sent is high which imposes a high overhead of sending and receiving messages on the CPU. If the buffer is too large, then the frequency of messages is low, but the processes will use less recent computed values [19].

3.4.5 CVM

CVM is a DSM system developed at the University of Maryland. This DSM was implemented to be easilly adapted to the needs of the applications. It has **multiple protocol support**: it provides four memory models, single and multiple-writer versions of lazy release consistency, sequential consistency, and eager release consistency. The source code is written in C++ and is freely available, new classes can easily be derived from a master class allowing new protocols to be easily incorporated. Therefore, CVM is **extensible**. CVM supports **multithreading** by implementing context switching. CVM can be **online reconfigurated** - feature that was implemented using thread mobility. The degree of parallelism, the load balancing, the minimization of the communication requirements can be achived by thread migration. Another goal of CVM is **heterogeneity**: CVM can be executed on heterogeneous clusters of workstations. To implement **Race Detection**, it was built a practical online race detection system that is guaranteed to catch all races that occur during an execution, with a small overhead. **The Tapes** are used to allow shared accessed to be recorded, grouped and manipulated at a very high level. They are implemented in some libraries that are layered on top of the consistency protocols and synchronization interfaces. These tapes can be used for improving of the performance: the future accesses can be predicted and the subsequent misses can be eliminated - the data can be easily moved using these tapes.

3.4.6 Rthreads

Rthreads is an object-based DSM system. To read, to write remote data objects or to synchronize remote accesses, it uses a set of primitives. The primitives are syntactically and semantically closely related to the POSIX thread model (Pthreads) and consequently, the precompiler can automatically translate the Pthreads source programs into Rthreads source programs.

A new model of the distributed shared memory was introduced in Rthreads: the global variables of a Pthreads program are transformed into shared variables in the Rthreads program. The operations on the shared data are executed at the locations of the global variables of each node. The information used for data transfer and conversion in heterogeneous environments is retrived from the source code by the Rthreads precompiler. Each node participating in an Rthreads program accesses global variables like a traditional parallel Pthreads program. These accesses are local and don't affect local copies of other nodes. Every participating Rthreads node program may consist of several Pthreads itself that are synchronized by the according primitives of Pthreads.

Rthreads is similar to Adsmith regarding explicit memory accesses and the implementation on top of existing communication systems. Rthreads is more flexible in data accesses because it provides several concepts to vary the granularity of the data sharing. In Rthreads there are also implemented further concepts for data structures and blockwise grouping of array elements. Adsmith only allows access to single data items or to complete arrays.

3.4.7 Quarks

The paper describing Quarks was presented in March 1998 at the "Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments". Quarks was implemented to be simple and also efficient. It consists of a user-level library and associated header files that support DSM on collections of Unix workstations.

The features provided by this DSM system are modern and high-efficient: multiple consistency protocols (a write invalidate protocol providing strict consistency, a delayed write update protocol providing release consistency) and multithreading to mask communication latency. Quarks provides a simple user interface that allows parallel debugging using gdb.

One of the goals of this DSM system is to reduce the impact of DSM computation overhead. The optimizations to provide this were:

- It was employed a release-consistent write-update protocol, similar to that employed by Munin.
- Incoming messages may be handled asynchronously, when the local node is idle or at a convenient blocking point, rather than invoking a high overhead signal.
- Outcoming messages are sent using non-blocking I/O operations.
- The use of multithreading was avoided within the Quarks runtime to support DSM operations, because the impact of context switching were very costly.
- Copy creation and memory protection-related system calls are performed while other pages are being requested.

The Quarks implementation uses as transport layer a Direct Deposit(DD) Protocol. This protocol enables the sender to manage a reserved receive buffer within the receiving process's address space that is obtained when the connection is established. The sender directs placement of messages within that buffer via an offset carried within the message header. DD uses a system call-based interface for sends to provide safety and flexibility at overhead and latency costs that are modest. The semantics of DD allow for asynchronous sends. Given a network interface with DMA capability, the transmission occurs in parallel with continued

computation within the user process. DD supports a completely user mode message reception. On message arrival, a notification object, or note, is written into a circular queue specified by the conection. This queue can be in kernel or user memory, and can be shared by several connections within a single process, at the discretion of the user.

The performance of Quarks is very good for one program and reasonable for a number of others that vary in the emphasis of their sharing behaviour on getpage (producer-consumer). The source of high performance of Quarks is given by the optimizations on the computation overhead of DSM. The time spent performing operations such as page fault handling, synchronous I/O, memory protection manipulation and distributed garbage collection can seriously impact the rate at which useful user computation is performed.

3.4.8 TradeMarks

Treadmarks is a DSM that supports parallel computing on networks of workstations. It provides a global shared address space across the different machines on a cluster. Its architecture took in consideration the fact that a shared memory interface is more desirable from the application programmer's viewpoint, allowing him or her to focus on algorithmic development rather than on managing communication. The challenge in providing a shared memory interface is to do so efficiently. To this end, TreadMarks incorporates several innovative features, including release consistency and multiple-writer protocols.

The research in the TreadMarks project includes the integration of compiler and runtime techniques, the use of multithreading, in particular on multiprocessor nodes, support for large address spaces, heterogeneity, and scalability. TreadMarks was tested on IBM, DEC, SUN, HP, x86 and SGI hardware. A port to WindowsNT has also been completed. C, C++, Java, and Fortran are supported. TreadMarks was developed with support from the Texas Advanced Technology Program.

3.4.9 JIAJIA

In this DSM, physical memories of multiple workstations are combined to form a larger shared space. In other DSM systems from the same generation such as Quarks, TreadMarks, and CVM, the shared address space is limited by the size of local main memory. In JIAJIA, the size of shared space can be as large as the sum of each machine's local memories. To provide consistency, a locked-based cache coherence protocol is proposed to simplify the design. The protocol is lock-based because it totally eliminates directory and all coherence related actions are taken through accessing write notices kept on the lock. Compared to the directory-based protocol, the lock-based protocol is simpler and consequently more efficient and scalable.

A new NUMA-like memory organization scheme which was taken to ease shared memory management. With the simplicity of this shared memory organization scheme and of consistency semantics, JIAJIA totally eliminates the complexity of garbage collection, local address to global address transition, and vector timestamp maintenance. A flexible shared memory allocation call is provided to allow the programmer to control the distribution of shared locations.

A home migration scheme is implemented to migrate home pages adaptively according to the application sharing pattern. A write vector technique is implemented to reduce message amount in home-based software DSMs. With this scheme, the faulting processor fetches only those blocks that are modified since last fetch. An adaptive write detection scheme is

implemented to reduce write faults on read-only pages. An new function call *jia_config()* is provided to turn home migration, write vector, adaptive write detection and other optimization methods on and off in the application program.

JMCL comes from JIAJIA specific Myrinet Communication Library. Myrinet was choose to be used as platform beacuse it has high speed and programming interfaces at multiple levels. JMCL provides reliability and protection in order message delivery for high level JIAJIA system.

3.4.10 JUMP

JUMP is a page-based software DSM system for clusters of PCs or workstations. It adopts the Migrating - Home Protocol (MHP) to implement Scope Consistency (ScC), both of which improve the DSM performance by reducing the amount of data traffic within the network. JUMP was implemented as a user-level C library on top of UNIX, and is able to run on homogeneous clusters of PCs or workstations with SunOS or Linux operating system.

In **Scope Consistency**, we define the concept of scope as all the critical sections using the same lock. This means the locks define the scopes implicitly, making the concept easy to understand. A scope is said to be opened at an acquire, and closed at a release. The definition of Scope Consistency is: when a processor Q opens a scope previously closed by another processor P, P propagates the updates made within the same scope to Q.

The **Home-Based Protocol** in the home-based protocol as adopted by JIAJIA V1.1, a processor is fixed to hold the most up-to-date copy of every page in shared memory. This processor is known as the home of the page. Under the home-based protocol, the updates made by every processor on a page are propagated to the home processor at synchronization time.

The **Migrating-Home Protocol** was used in JUMP because it was proved that the home-based protocol is more efficient than the homeless protocol, the fact that a fixed home may not adapt well to the memory access patterns of many applications. If the home processor itself never accesses the page, then the updates made by other processors must be propagated through the network at synchronization time. If the home can be migrated to the processor which accesses the page, then the updates made by the new home need not be sent anywhere. The migrating-home protocol which allows the home location of a page to be migrated from a processor when serving a page fault.

JUMP vs JIAJIA V2.1 For most applications, the home migration protocol in JIAJIA V2.1, was outperformed by the MHP in JUMP for 5 out of 6 applications. This means MHP is more efficient than the home-based protocols. In was observed that the more aggressive strategy and a wider usage (working on both locks and barriers) accounts for the higher efficiency of MHP.

The performance in communication can be improved by reducing the software protocol overhead. Socket-DP is a low-latency communication package with traditional socket interface to maintain good programmability. The tests showed that the migrating-home protocol is capable of improving the performance of some applications dramatically, while Socket-DP introduces modest performance gain on all applications tested. The two enhancements work together well to improve the performance of DSM applications substantially.

3.5 Optimizing Compiler in Software DSM

One of the goals of such a compiler is to insert valid write commitments as much as possible [36]. First, all the shared memory accesses given in a shared memory program must be enumerated - **shared write detection**. Points-to analysis represents all variables as memory locations. This is a conservative assumption in C. When an input program contains unions or type-castings, they may generate false alias information, which takes many iterations to converge. It is important that the input program is type-safe about pointer values, that is pointer values are not conveyed through non-pointer locations. This prevents the generation of false alias relations in a program with complex structures. Interprocedural points-to analysis calculates symbolic locations where variables may point to [36]. Variables and heap locations are represented with a location set - a tuple of a symbolic base address, an offset and a stride(step). The compiler interprocedurally calculates points-to relations among location sets using a depth-first traversal of the graph. A write commitement is inserted after a write operation using shared address values.

The interprocedural analysis has the merits that the succeeding optimization passes can perform code motion using pointer information and precise shared pointer information can decrease the costs of the redundancy elimination pass.

In release consistency model, a shared write is not transmited to other nodes until the node which had issued the shared write reaches a synchronization. Therefore, it is important that a write commitment to be placed everywhere from the corresponding shared write to the first synchronization thereafter. This can be used to remove redundant write commitments.

If we have:

```
v[x][y]=value1;
if (x==y)
    v[x][y] += value2;
```

Write commitments should be inserted after both assignments. If the first write commitment will be delaied after the conditional, the write commitment within the conditional will be redundant. This optimization was formalized as **redundancy elimination**.

There are many cases in which the write operations are performed into the same contigous region. If we have a loop, for example:

```
for ( k=1; k <=len ; k++ )
   v[k]=a[k+1]*b[k+2];
```

then the write commitment should not be inserted in the innermost loop, but outside the loop - **merging multiple write commitments** [36].

Chapter 4

ParCeL on ShM Systems

ParCeL-6 is a library devoted to cellular programming and to shared memory parallel architectures:

- it allows fast development of cellular programs;
- it allows easy (automatic) parallelization on multiprocessor machines, including cheap multiprocessor PCs;
- it allows easy experiment of different cell output propagation methods, to study future design for cluster and grid architectures.

ParCeL6.1 is the shared memory version of ParCeL-6 for single and multiprocessor machines. This version is available for both Linux (using Posix Threads) and Windows (using Windows native threads).

4.1 Programming model of ParCeL-6.1

In order to facilitate the understanding of how ParCeL-6.1 works, the structure of a ParCeL-6 program must be descrribed. The entities used in ParCeL are called cells - many small computation units, statically or dynamically created and connected, exchanging data, and that can run on parallel machines. A typical ParCeL-6 program consists in a main function beginning with cell definitions: the programmer specifies the functions associated to this cells (for initial, current and final iterations), and the size and kind of its output. The program can be considered like a sequential program until it encounters a *p6_net_creation command*. Then all cells defined are created and automatically distributed on all processors, but cells are not run. The program continue sequentially, until a *p6_net_computation command*. Then all cells are activated, on all processors, and run their activation functions one time. The main function of the ParCeL-6 program continues with a cellular activation loop: the created cellular network is run many times inside a computation loop, calling the *p6_net_computation* command. However, some cells can be dynamically created and killed after the cellular network start, from the main function or from already existing cells. The user can design and implement static or dynamic cellular networks. Finally, the loop computation finishes, and the program ends. A ParCeL-6 program is a kind of sequential program defining and running a cellular network, with cellular net callback.

ParCeL-6.1 is a library written in C language, with an API that is voluntary not too large, and can be linked with C or C++ programs [53].

ParCeL-6.1 uses some concepts that must be known to the reader in order to better understanding its internal structure: *processors, requests, missions, OutConns, permutation tables.*

Processors: ParCeL-6 is structured into multiple *processors.* These virtual processors are the ones that control the flow of the program, by performing every command that they receive - *mission.* A processor has a number of cells, tables and hash-tables to access them, and some other structures.

Requests: As the name says, a request is a demand for something. The demand (command) is called from a cell function with execution delayed to the end of the cycle (after cell computing step). Results are visible at the next cycle. ParCeL-6.1 has requests defined for cell creation (CREATERQ) / destruction (KILLRQ), cells connection (OUTINFOGETRQ) / disconnection (DISCONNECTRQ), response to a connection request (OUTINFOBACKRQ), local link connections (CONNECTRQ), cell death announcements (KILLSIGNALRQ).

The requests are C structures, with properties specific to every type of request, and they are stored in request tables that are kept on every processor. This works as it follows: when a cell wants to create another cell for example, puts a *CellCreateRequest* in the *CellRequestTable.* The request structure contains the destination processor that is meant to execute this demand. At the next cycle, when the mission that says to execute the cell requests is received, every processor will access the tables of request of all the others, and will take the requests that are destinated to it. Then, it executes every one of them, and if it is the case, it stores a response request for the processor that made the request in the first place.

Missions As explained before, a mission should be seen as command to execute some particular piece of code. Every processor waits in an idle state a mission, and when it receives it, it begins to execute the appropriate code for the mission.

From the implementation point of view, a mission is an *int* value that is sent from the main processor to all others.

Waiting for a mission is performed with the help of semaphores. The main processor sends the mission code, and after that it signals the semaphores for every processor that a new mission is ready to be executed.

Missions available:

- INIT - threads make some local init
- INITPERMUT - threads init their local permutation table (when they are defined)
- NETCREATE - threads create their part of cell net
- NETCOMPUTATION - threads run computation functions of their cells
- NETCELLEVOLUTION - threads accomplish cell evolution request destinated to them (ex: threads process cell creation requests)
- NETLINKEVOLUTION - threads accomplish link evolution request destinated to them (ex: threads process cell connection requests)

- NETHYBRIDOUTUPDATE - threads reset the refresh flags of their buffered cell output
- NETBUFFEREDOUTUPDATE - threads propagate their pure-buffered cell output
- HALT - threads terminate

OutConns The "OutConn" states for "Out Connection", and it is a structure that is created when a cell from one processor wants to connect to a cell hosted by another processor or even on the same one. This structure will contain all the information that are further needed to access the connected cell.

Permutation tables When a processor starts to execute the cells that it hosts, it can run them in different order. The user has the option of influencing the order of cell activation, and if he wants a particular order, he has to provide some *permutation tables* for the cell activation (the cell hash-tables will be crossed in the order given by this tables). The order can be one of the following: P6FORWARD (basic cell activation case (forward) from the first cell in the hash-table to the last), P6BACKWARD (reverse run of the cell hash table), P6ALTERNATE (alternate hash table list order of run), P6PERMUT (follow a permutation table), P6PERMUTRANDOM (choose randomly the permutation table), P6PERMUTALTERNATE (follow a permutation table and alternate), P6PERMUTRANDOMALTERNATE (choose randomly the permutation table).

The parallel implementation of ParCeL-6.1 is based on threads. All threads are launched at the initialization time by the main thread. Every thread is associated a ParCeL-6.1 virtual "processor". The processors communicate with each other through a global space (global variables), a very fast and convenient way for a shared memory system (Figure 4.1).

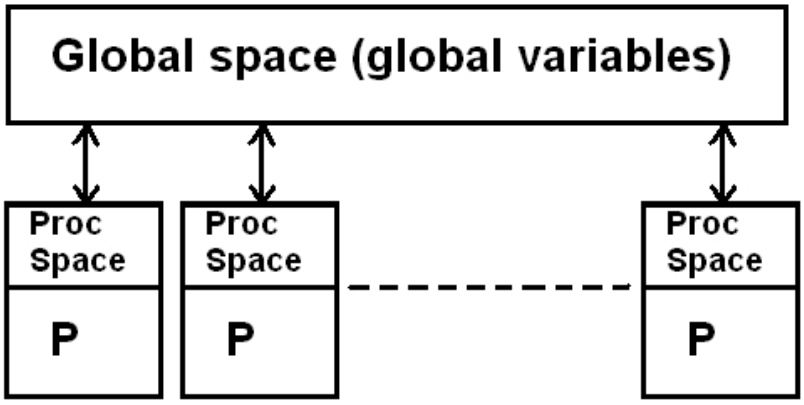


Figure 4.1: Global Space

Global space contains the following:

- table of all structures of the processors
- table of load of processors: how many cells on every processor
- tables of requests: all requests from all processors

- table of mission codes for all processors
- table of all cells on all processors
- hash-table of cell registrations
- permutation of cell execution
- tables for all types of output (direct, hybrid, buffered) for all cells
- table of all OutConn tables
- table of all OutConn values tables
- table of all OutConn refreshed flag tables
- table of all Out Lock tables
- table of Htable of OutConn already on a processor

Every time one processor needs some data from the structures of another processor, it can access it directly through the global space.

The structure of a ParCeL-6 processor is the following:

- processor number: the number used to identify the processor
- the total number of processors
- load balancing management data: used when creating cells to determine the future host of the cell
- cells request tables: contains requests expressed by the cells hosted by this processor. The requests that are kept in this table are for creation, destruction or for announcing the death of one cell. These announces are necessary because when a cell dies, all the cells that have connections open to this cell should close them.
- cell request tables destined to the proc: requests that are issued by cells from all the processors. Before starting to execute the requests, every processor looks in the "cell request tables" of all the others, and copies into these tables the requests that are destined to him.
- cell back-request tables of the processor: after processing the "cell requests", a response may be needed to be sent to the initiator of the request; these kind of responses are of the shape of "back-request", and they are stored in this type of tables (for example, when processing a kill request, a kill signal request is sent to every process in order to announce that this cell has died).
- cell back-request tables destined to the processor: the cell back requests issued for this processor.
- info request tables of the processor: the connection requests made by the cells of this processor

- info request tables destined to the processor: the connection requests issued for the processor
- table of local link requests: when making a connection request to a cell on another processor, a "local" request is also stored, so that when the connection response comes, a local linking is performed.
- cells table of the processor: a table of all cells hosted by this processor
- hash-table of cell registration of the proc hashtable of cells used to increase the access speed to cells.
- current permutation table
- table of direct cell output: output values from the cell that have "direct" output mode
- table of hybrid cell output: output values from the cell that have "hybrid" output mode
- table of buffered cell output: output values from the cell that have "buffered" output mode
- table of old buffered cell output: output values in the previous cycle from the cell that use "buffered" output mode
- tables with the flags for OutConns: this flags say if the OutConn needs to be refreshed or not
- table of OutConns: the OutConn structures for the processor
- table of OutConn values: the output values of the cells that correspond to the OutConns
- hash-table of OutConns: hash-table of OutConns used to increase the access speed to OutConns.

The structures of the *processors* are the most complex ones, because a ParCeL-6 processor has to contain all the other structures that are needed for computation.

Cell structure (Figure 4.2):

- cell registration: combination of host processor number, creator processor number and cell number that uniquely identifies the cell
- init function: function that is run at the initialization of the cell
- iteration function: runs at every cycle
- termination function: runs when the cell is killed; removes the cell from the cells table, cells hash-table and performs other cleaning needed
- current function index: keeps track of the current function from the *initialization*, *iteration* and *termination*
- output information: gives access to the output of the cell; the output contains information for the kind of output (*direct*, *buffered* or *hybrid*), the number of output values and the table to store the values in

– local variables of the cell

– access to the next and previous cell

– *UserData* space: a memory space that can be used by the user to store some cell specific information he may need.

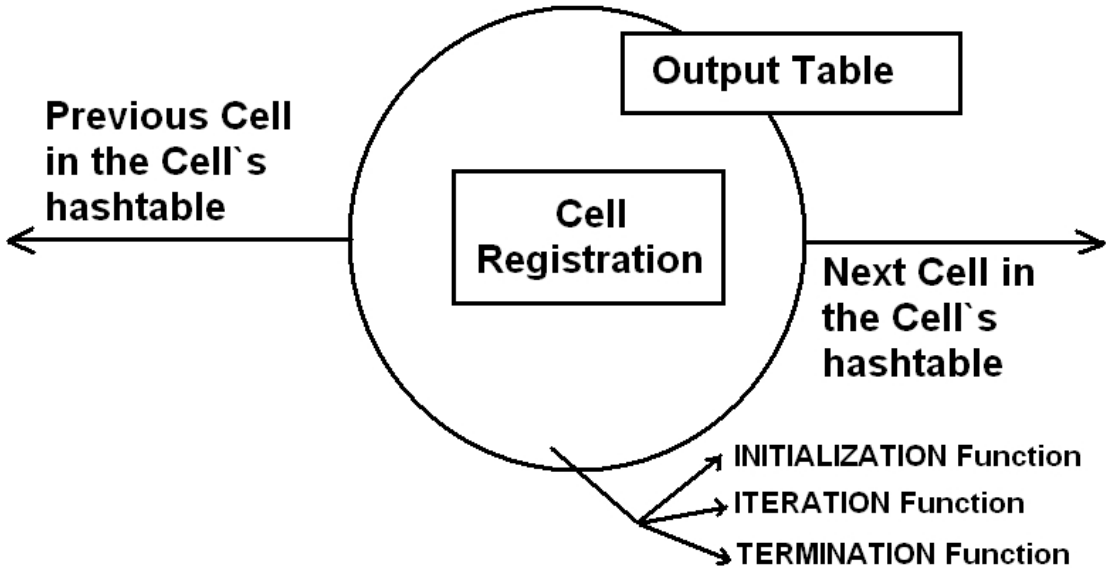


Figure 4.2: The Structure of ParCeL6 Cell

OutConns structure:

- OutConn status: shows if the cell connected to is still alive or not
- cell register: identification of the cell this OutConn refers to
- out kind: the output kind of the cell ("direct", "buffered" or "hybrid")
- number of readers: how many cells read from this OutConn
- number of output values of the cell
- pointer on the remote output table for the cell
- output values of the cell
- next OutConn
- refresh flag: shows if this OutConn should be refreshed or not
- lock flag: insures the mutual exclusion of accesses to this OutConn

For some of the data types it needs to store (e.g. cells, output values of the cells etc), ParCeL-6.1 uses a special type of tables (Figure 4.3). The difference between this kind of

table and a vector for example, is that this table can be chained with other tables of the same kind and that it can also keep track of the free space inside.

For holding the cell structures on a processor for example, ParCeL-6.1 uses this kind of tables. They can be very useful when there are a lot of cell creation and destruction. When a new cell is created, first a search is performed through the free space list, and if the necessary space is found, the cell is stored in the place found. This location is removed from the free space list. If there is not enough space in the free list space, an entire table is allocated (because of high execution cost of memory allocation) and it is chained with the existing tables. The cell is stored in the newly allocated table, and the index for the free space in the table is set to null. When a cell is deleted, it is removed from the table and a free element is put in its place and chained in the free space list. In this way, the space allocation is not performed very often, which saves execution time, and also the space that is no longer in use can be reused. Another technique for reusing the free space would be to rearrange the table after every freeing of space, but that would take a lot of computation time, and when a lot of cell destruction takes place, this would not be a very efficient technique.

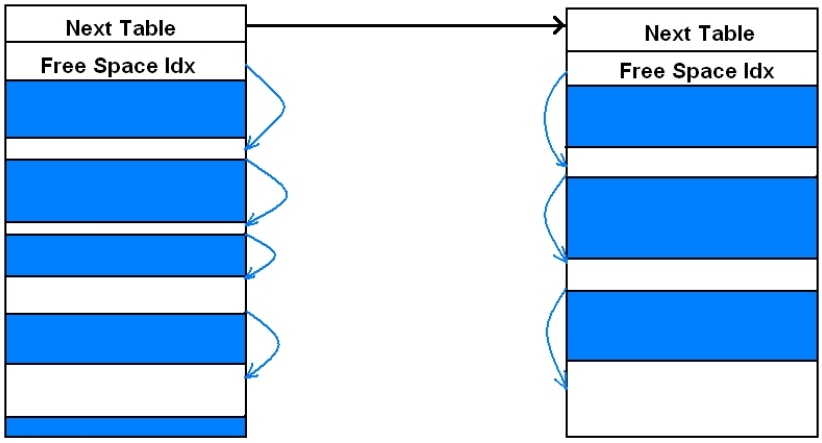


Figure 4.3: The Chained Tables of ParCeL6

Because they are heavily accessed the cells and the OutConns are stored in hashtables (Figure 4.4) to increase the search speed. The entries of the hash-tables are the cell registers, and for one entry in the hash-table, there is a chain of cells / OutConns. The hashtable for the cells contains double-chained elements, making possible to cross the chained cells in the reverse order, while the one for the OutConns is simple-chained.

4.2 Introducing the Posix Semaphores in ParCeL-6.1

4.2.1 A Brief Description of The Source Files of ParCeL-6.1

In the next section the reader can find out the role of the source files of ParCeL-6.1 and the interdependence between them.

p6type.h This file contains the definitions for all the structures and the enumerations used in ParCeL-6.1. The definitions are organized by theme (cell, cell requests, management informa-

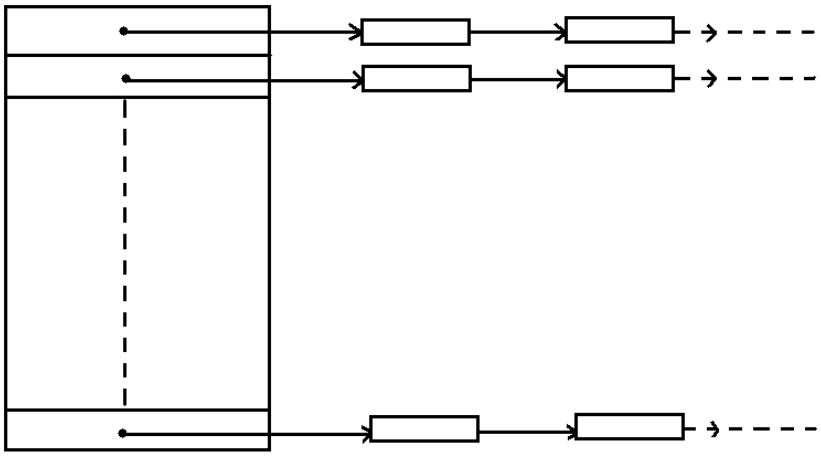


Figure 4.4: The HashTables of ParCeL6

tion used by each processor). Here are defined the structures needed for cell registration, cell output, cell requests, typedefs for processor/thread informations-management, hash-tables, load-balancing informations and some other useful typedefs.

p6const.h Here are defined the constant values used in ParCeL-6.1 (like: number of requests per basic table, number of cells per basic table, number of out values per basic table, number of locks per basic table, number of flags per basic table, number of OutConn per basic table).

p6GlobalVar.h/c These files are in fact an interface that can be used by any of the processes to access the global variables (e.g.: table of processor informations, table of all request from all processors, table of all cells on all processors, hashtable of cell registrations, table of all OutConn tables).

p6CellComput.h/c In these files are defined the handling level functions of the cells (the run of all classic cell computation and the run of cell reactions on net evolution).

p6init.h/c Here are defined some functions to declare/kill cells, to initialize and to free the resources used in the threads and functions that manage the connections.

p6OutConn.h/c In these files are defined the functions for Out Connections management (Refresh a connected output, Operations on the OutConn Hash table, Allocate or Free new OutConn in local table, Allocate or Free new flag in local table, Allocate or Free new OutConn values in local table, Connection to output of a sending cell, Disconnection of an OutConn).

p6Output.h/c Here are defined some functions to write to the ouput or to read someone's output (lock management is included).

p6Mission.h/c When a process wants to broadcast a mission (e.g. Init, Create Cells or Halt) and all processes/threads should cycle on reading a mission and then execute it.

p6Request.h/c The functions for receiving and executing requests are defined here.

p6net.h/c Here are defined the functions for sending and receiving request missions. In one cycle a cell does: receive request tables, read mission and with the mission does the appropriate work on the information provided in the requests.

p6tools.h/c The functions for memory management, the hash function and the function used to get infos on a processor, are defined here.

parcel6.h This file contains the signatures for all the functions used in ParCeL-6.1, tries to give some explanations for each of them and sometimes gives a typical example of how these functions should be used.

p6LowLevel.h/c Here are defined the LowLevel functions like the functions for implementing the synchronization mechanisms on Threads (Mutexes, Semaphores, Barrier) and the randomization functions used to initialize the permutation tables. In this file was added a Linux implementation of the synchronization mechanisms using posix semaphores.

4.2.2 The Implementation with Posix Semaphores

The main purpose of the implementation with Posix Semaphores is to obtain a reliable and portable application conforming to the Posix (Portable Operating System Interface) Standards that can easily run on the Distributed Shared Memory (DSM) systems.

In the file *p6LowLevel.h/c*, the initial implementation on Linux used only the SystemV semaphores (the compilation flag in order to use this implementation is *P6_SYSTEMV_SEM*). We added to this implementation another implementation using posix semaphores (the compilation flag in order to use this implementation is *P6_POSIX_SEM*).

The SystemV semaphores that have now an analogous posix implementation are used as it follows. A table with NbPE (number of processors) semaphores is used for each begining of mission. These semaphores are initialized with zero and are used as mutexes. Four semaphores and a shared variable (a barrier index - int value) are used to implement the SystemV barrier in ParCeL-6.1. The complexity of this barrier is *O(2*P)*, where P is the number of threads.

The functions that were implemented using the posix semaphores, are:

- *p6LowMissionSemaphoreTabInit(int NbPE)*: the mutex table with NbPE semaphores is initialized
- *p6LowBarrierInit(int NbPE)*: initializes the barrier semaphores to zero and the shared variable to zero
- *p6LowMissionSemaphoreTabFree(int NbPE)* is used to free the table with semaphores that are used for missions
- *p6LowBarrierFree(void)* is used to destroy the semaphores used in the implementation of the barrier
- *p6LowSignalMission(int NbPE)*: release each mission semaphore of the mission semaphore table

- *p6LowWaitMission(p6ProcInfo_t *PtProcInfo)*: for a processor/thread to wait at its corresponding semaphore (one from NbPE semaphores) in the mission semaphore table
- *p6LowBarrier(p6ProcInfo_t *PtProcInfo)*: must be called by a processor to wait at the reentrant barrier. When all the processes will arrive to the barrier, the master processor (the processor with the id zero) will command their release.

In *the implementation using posix semaphores* (compilation flag *P6_POSIX_SEM*), the functions have the same meaning, but they are conforming to the posix standard: the Mission Mutex Table was replaced with a table with posix semaphores (NbPE *sem_t* semaphores) and the four semaphores used to implement the barrier were also replaced (with four *sem_t* semaphores).

The code became more concentrated and easy to follow and the main goal of taking advantage of portability (conforming to Posix Standards) was achieved. A simple ParCeL-6.1 application was run on the DSM using succesfully all the Posix implemented synchronization mechanisms.

Chapter 5

Kerrighed

A practical solution to obtain SpeedUp when running a parallel application on a cluster, and in the same time a solution that maintains transparency to the programmer, is to install and to use a software DSM system on the cluster. There exists a much faster DSM solution, but very expensive - this solution would be to use a hardware DSM, but due to its costs, some limitations would be encoutered from the point of view of accessibility and upgrading - often a hardware upgrade can be much cheaper when the old components are totally forgot and replaced.

This chapter is exclusively dedicated to the software DSM we have installed, tested and used with ParCeL6. The introduction has the role to emphasize some main features of this software system. In the second phase, the results that we have obtained with some general tests will be presented to the reader. It is important for the reader knowledge, to specify that all the time values that are mentioned and that appear in the diagrams are in fact the average values of ten measurements [62, 63].

5.1 Description of Kerrighed

Kerrighed is a software DSM that offers the view of a unique SMP machine on top of a cluster of standard PCs. It is implemented as an extension to Linux operating system (a set of Linux modules and a small patch to the kernel). The main features of *Kerrighed* are[62]:

- *Customizable Cluster Wide Process Scheduler:* Processes and threads are automatically scheduled over the cluster nodes to balance the CPU load using the Kerrighed default scheduling algorithm. The global scheduler has to be properly configurated and all its modules must be loaded in the kernel (the names of scheduler's modules are *cpu_scheduler2*, *migration_analyzer*, *mosix_filter*).

- *Cluster Wide Shared Memory:* Threads and System V memory segments can operate through the whole cluster, just like on a SMP machine. All the threads of a process have the facility to migrate on any node of the cluster without any intervention from the programmer.

- *High Performance Stream Migration Mechanism:* Processes using streams (e.g. socket, pipe, fifo, char device) can be migrated with no penalty on communication performance after migration. There is a module in Kerrighed that maintains the efficiency and performance of the streams in their distributed environment.

- *Distributed File System:* A unique file name space is seen over the whole cluster. All cluster disks are merged in an unique virtual disk. A file descriptor will be unique all over the whole cluster.

- *Process Checkpointing:* Processes can be checkpointed and restarted on any cluster node due to the flexibility of this distributed system.

- *Full Posix Thread Interface:* The full Posix Thread interface can operate with threads spread over cluster nodes. In Kerrighed is defined a native posix barrier (*pthread_barrier_t*) for distributed environments.

- *Cluster Wide Unix Process Interface:* All traditional UNIX process management commands (e.g. top, ps, kill) operate cluster wide. Another feature is that the process identifiers (pid) are unique cluster wide, so, when running top for example, we can see the amount of resources used by a process on each machine of the cluster.

- *Customizable Single System Image Features:* Single system image features (e.g. shared memory, global scheduler, migrable streams) can be enabled or disabled on a per process basis. Kerrighed has in fact a large number of primitives that can be applied in order to manage and to configurate the processes that can migrate cluster wide.

There are some features that Kerrighed doesn't offer and it is important for the users to have knowledge of them:

- *An Automatic Parallelizer:* Kerrighed does not parallelize automatically the applications. This means that a big sequential process will not run faster on Kerrighed. To run faster, an application has to be parallelized. The parallelization of an application must be a compromise between the granularity and the performances of the systems that will run it, or, in other words, a compromise between granularity and the level of interprocess (or interthread) communication. The main idea would be that if the application runs faster on a multiprocessor rather than on a uniprocessor machine, it is maybe able to run faster on Kerrighed than on a uniprocessor machine. If the application does not run faster on a multiprocessor, it will certainly not run faster on Kerrighed.

- *A Middleware:* Kerrighed is an operating system, *not* a middleware. It runs inside Linux, not on top of Linux. The mission of Kerrighed is to extend Linux functionalities to manage cluster wide services.

- *A Virtual Machine:* Kerrighed does not create a virtual cluster, it gives the illusion that a physical cluster of PCs is a SMP machine, in order to make possible easier programming.

5.2 The Architecture of Kerrighed

We have worked with Kerrighed version 1.0. In the following section, there are presented to the reader the general architecture of the Kerrighed system and a brief description of the functionalities offered through its different modules (for details see [62]).
Kerrighed architecture can be observed in Figure 5.1 and it is composed from a number of specialized modules.

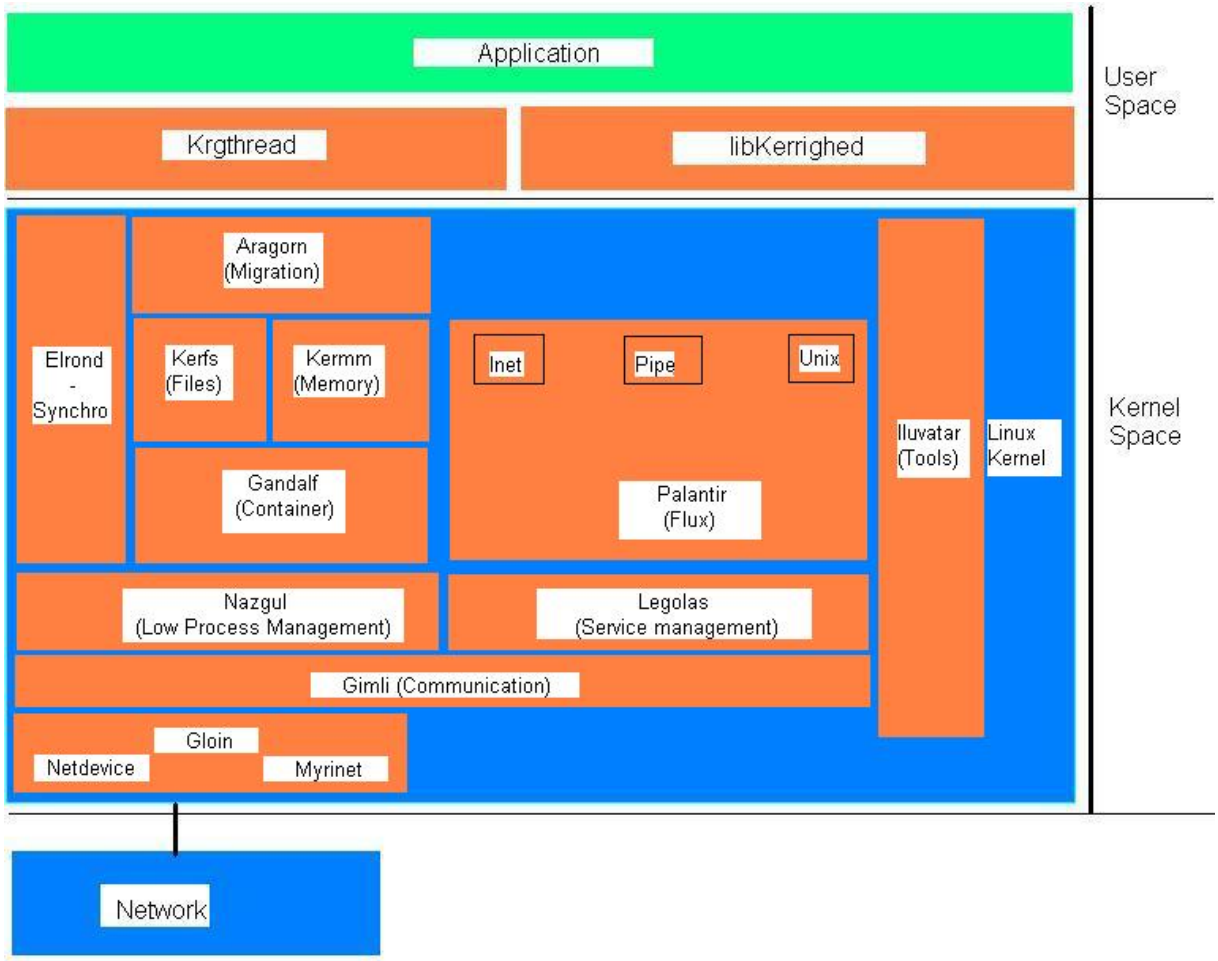


Figure 5.1: The general Architecture of Kerrighed System Version 1.0

The module *Iluvatar* is the first module loaded when starting the system. It offers basic functionalities that are used by the other modules. It has the role to manage the data structures of the processes that run on Kerrighed.

The modules *Gimli* and *Gloin* are in charge with the communication between nodes: *Gloin* takes care of the network access, while *Gimli* offers the communication interface. It is known that Linux offers two types of mechanisms for communication: the *sockets* and the *RPC*. The disadvantage of these mechanisms is that they offer low performances and a not very simple interface for the distributed programming. This is the reason that lead to the implementation in Kerrighed of the modules *Gimli* and *Gloin* that provide a communication library with high performances in a distributed environment.

Nazgul offers the main tools for replication, usage of checkpoints and migration of processes. This module also offers the mechanism for remote function invocation. The data are imported and exported using a mechanism named *ghost*. This mechanism provides an interface for storing the data - on the disk, in the memory, or for sending the data in the network. It is used to register and to activate some of the services used by the other modules. Its functionalities can't be directly accesed from the user space.

Legolas makes possible the access to the services of Kerrighed. A service is a mixture of a folder with distributed data and function calls on the entries of this folder.

Palantir manages the *flux* in Kerrighed. The extension modules *Palantir/Inet*, *Palantir/Pipe* and *Palantir/Unix* provide the interfaces for the access from the modules above Palantir.

Elrond provides the synchronization primitives: locks, barriers, semaphores, condition variables and atomic functions. These synchronization functions can be directly used, or they can be used through the *pthread interface* offered by the library *krgthread*. The interprocesses synchronization is provided by using a structure for each process with pointers to the synchronization objects.

The module baptized *Gandalf* provides the sharing of data, and maintains its coherence by using a container mechanism. Each shared object (file, memory segment, IPC) has associated a container. In order to mantain easy access to the containers for the high-level services, Gandalf provides a data coherent interface.

Kermm provides the global access to the memory by using the container mechanism provided by module *Gandalf* to keep available the sharing of data (threads, systemV memory segments) and to mantain the coherence. All the management of the physical memory is done by using only two Linux functions: *get_free_page* and *free_page*.

Kerfs offers a system for the parallel and distributed management of distributed files. It uses the containers in order to provide sharing and coherence of the cached data and also for some internal data structures (e.g. i-nodes, file structure).

Aragorn is responsible with the replication mechanisms, the checkpoint management and with the migration of the processes. The migration of a process consists in two main steps: the migration of the execution context of the process and the maintaining of the relation between the migrated process and the resources that it used on the initial execution node. The module *Aragorn* makes possible the first phase of the migration using the *ghost* mechanisms offered by Nazgul, while the second phase is automatically provided by the other modules.

The library Krgthread provides an almost complete Posix interface to the kernel mechanisms offered by Kerrighed. The thread synchronization functions (e.g. *pthread_mutex_lock*, *pthread_cond_wait*) use the mechanisms provided by the module *Elrond*. The function *pthread_create* uses the mechanism of remote process creation offered by the module *Aragorn*. The memory sharing, default for the threads, is provided by the module *Gandalf*.

The library *libKerrighed* offers some Kerrighed primitives (e.g. *migrate_self()*, *aragorn_create_gthread()*), implemented in the modules *Iluvatar*, *Gimli* and *Aragorn*.

5.3 Kerrighed Installation

5.3.1 Available Hardware

We have installed and run Kerrighed 1.0.0 on a cluster at Supélec containing four machines P4 Xeon 2.4GHz with 1GB main memory connected through Gigabit Ethernet.

Detils about the motherboard:

- FSB 533MHz
- chipset Intel E7501
- 2 onboard channels Ultra320 SCSI

- harddisk ST336607LC 36,7 Go Ultra320, max. 3 disks
- onboard 10/100/1000Base-T Gigabit Ethernet dual port

The machines were connected using a *HP 2724 Procurve (J4897A)* switch. Other details about this switch:

- 24 RJ-45 10/100/1000 ports (IEEE 802.3 Type 10Base-T, IEEE 802.3u Type 100Base-T, IEEE 802.3ab 1000Base-T Gigabit Ethernet)
- Dimensions: 44.2 x 23.62 x 4.32 cm
- Switching capacity: 48 Gbps
- Address table size: 32000 entries

5.3.2 Building the Kernel

The Linux distribution that we used was RedHat9. At the moment we have installed Kerrighed, we were more or less obliged to use this distribution because the latest version of this software DSM available at that time (v1.0.0) supported only the Linux 2.4.24 kernel that was not compatible with FedoraCore3 (the other distribution that was also installed on some machines).

In order to succesfully run and install Kerrighed 1.0.0 there are some steps that must be followed [63].

Firstly, the Kerrighed package and the Linux 2.4.24 source code must be downloaded from:

- Kerrighed package: *www.kerrighed.org/download.html*
- Linux 2.4.24 source: *www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.24.tar.bz2*

Secondly, we had to apply the Kerrighed patch on the kernel.

We have installed Kerrighed in the shared directory */usr/local/kerrighed*.

To uncompress the Kerrighed packages, we have executed the following:

```
$cd /usr/local/kerrighed
$tar zxvf kerrighed-1.0.0.tar.gz
$ln -s kerrighed-1.0.0 Kerrighed
$export KERRIGHED_PATH='pwd'/Kerrighed
```

Then we uncompressed the kernel source:

```
$cd /usr/local/kerrighed
$tar jxvf linux-2.4.24.tar.bz2
$mv linux-2.4.24 linux-2.4.24-krg
$export LINUX_PATH='pwd'/linux-2.4.24-krg
```

To apply the patch on the kernel the following must be executed:

```
$cd $KERRIGHED_PATH/config/kernel/2.4.24
$make
```

Configuring the kernel is the next step of the installation:

```
$cd $LINUX_PATH
$make menuconfig
```

The Kerrighed version that was available at that time was not completly SMP-safe, so the kernel must be configured to be *non SMP*, but to be able to normally boot each node in the cluster and to have network communication. The version 1.0.0 of Kerrighed didn't support high memory option so this feature should be deactivated when configuring the new kernel. The high memory support feature is supported in Kerrighed version 1.0.1 and 1.0.2, versions that appeared good time after our first installation. There are other important features that can be useful when troubleshooting Kerrighed in order to "catch" the bugs:

- the Kernel Debugger must be compiled and activated in the kernel
- the section "Kernel Hacking" in the kernel configuration menu should look like this:

```
[*] Compile the kernel with frame pointers
[*] Built-in Kernel Debugger support
< > KDB modules
[ ] KDB off by default
(0) KDB continues after catastrophic errors
```

To build the kernel the following command should be used:

```
$make dep clean bzImage
```

For a Lilo boot loader, in order to install the kernel, the following steps must be followed (as root):

```
#cd /usr/local/kerrighed/linux
#cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.24-krg
#cp System.map /boot/System.map-2.4.24-krg
#cp .config /boot/config-2.4.24-krg
```

The next lines should be added to */etc/lilo.conf*:

```
image=/boot/vmlinuz-2.4.24-krg
label=2.4.24-krg
read-only
```

The configuration is applied to lilo by issuing:

```
#lilo
```

In order to choose the next kernel to boot without changing the default configuration, the following command can be used:

```
#lilo -R 2.4.29-krg
#reboot
```

In order to install a kernel that was built on one node on the other three nodes of the cluster, we have used the following script that takes as argument the name of the target machine:

```
#
#
if [ -z "$1" ]; then
        exit -1
fi
rdist -c /boot $1
rdist -c /lib/modules/2.4.24-krg-root $1
rdist -c /usr/local/kerrighed $1
rdist -c /etc/profile.d $1
rdist -c /etc/sudoers $1
ssh $1 lilo
```

5.3.3 Starting Kerrighed Cluster

In order to install Kerrighed the following commands must be executed:

- in the Kerrighed directory:

  ```
  $ cd $KERRIGHED_PATH
  $ make
  $ make install
  ```

- if someone wants to use the additional tools that came with Kerrighed, the GTK and GDK environments must be installed, and the following should be executed:

  ```
  $ make tools
  $ make install
  ```

To start Kerrighed cluster, the following configuration files should be created:

- */etc/kerrighed/kerrighed_nodes*; Our file (for a cluster with four nodes) looks like this:

  ```
  monox1 eth0
  monox2 eth0
  monox3 eth0
  monox4 eth0
  ```

- */etc/kerrighed/kerrighed_session*; This file is used to store an unique number in order to identify distinct Kerrighed clusters that can be found in the same network. In our file we have used the value *1*.

- In the */root* directory - in our case the *root will be the user that runs Kerrighed*, the files *.kerrighed_nodes* and *.kerrighed_session* have the same content as the files above.

- The user that starts Kerrighed must be able to run insmod/rmmod commands using the command *sudo*. In order to do this, our file */etc/sudoers* looks like this on *all* the four nodes of the cluster:

```
Host_Alias CLUSTER=monox1,monox2,monox3,monox4
Cmnd_Alias MODULE=/sbin/insmod,/sbin/rmmod
root            ALL=(ALL) ALL
ifrim_mir       CLUSTER=NOPASSWD: MODULE
vialle          CLUSTER=NOPASSWD: MODULE
demo            CLUSTER=(ALL) ALL
```

- In order to be able to use the KerFS filesystem, the following directories had to be created on *all* the Kerrighed nodes:

```
# mkdir /.KERFS_ROOT
# mkdir /mnt/kerfs
# ln -s /mnt/kerfs/chkpt /var/chkpt
```

- In the file */etc/fstab* on all the Kerrighed nodes, we have added the line:

  none /mnt/kerfs kerfs noauto,defaults,user 0 0

Now Kerrighed must be started by running

```
krgreboot
```

on only *one* node in the cluster. This loads the Kerrighed environment on *every* node in the cluster. In order to use the distributed KerFS filesystem, on only one node in the cluster, the following should be executed:

```
mount /mnt/kerfs
chmod 777 /mnt/kerfs
```

At the time we have installed Kerrighed, there was not available any mechanism to stop successfully the Kerrighed services, and consequently it was not available a mechanism to restart successfully the machines of the Kerrighed cluster.

5.4 Limitations in Kerrighed v1.0.0

There are some limitations and problems that a user of this Kerrighed version may encounter and it is important to have knowledge of them:

- A process cannot create more than 32 synchronization objects (e.g. lock, barrier)
- No SMP support
- No 64bits support
- No consistent cluster wide time management

- Kerrighed modules cannot be properly unloaded
- Do not manage cluster bigger than 32 nodes
- mremap is not supported within migrated or deployed threads
- Cluster wide System V semaphores are not supported
- Hot node addition or removal is not supported
- Cannot list more than 128 files in /proc/<pid>/fd
- Some process crash can occur on some corner cases when using OpenMP
- Small memory leaks
- *pthread_cond* may block (very unlikely)
- A migrated socket does not send SIGIO to peer socket. A alternative to select is to let the kernel inform the application about events via a SIGIO signal. For that the FASYNC flag must be set on a socket file descriptor via *fcntl* and a valid signal handler for SIGIO must be installed via *sigaction*.
- One node crash is likely to crash or dead-lock the all cluster.

Another important observation is that in our case the applications that we run on Kerrighed had to be launched from the directory */usr/local/kerrighed/tests*, either the binary is there or not, but the prompt must be */usr/local/kerrighed/tests* (in our case Kerrighed installation directory is */usr/local/kerrighed*). If we didn't executed the applications in this way, one of the nodes in the cluster would have gone in kernel panic.

5.5 Benchmarks And Strategies on Kerrighed

We first established some strategies and thought to some implementation of algorithms in order to test the performances of the DSM on our cluster. In the beginning we wanted to see what Kerrighed can do and what it can't do from the point of view of performances. In order to do this, we have decided to find the answer for some key critical situations:

- optimal and worst cases from the point of view of the frequency of the memory page misses - independent computations in threads or the opposite
- what is the most efficient mechanism of thread synchronization on cluster: the barrier implemented in ParCeL-6.1 exhibits better performances when running the application on the DSM than the native Kerrighed barrier?; the native barrier synchronization mechanisms of Kerrighed are really efficient on the distributed environment of the cluster?

The next step had the role to determine the performances and the limitations of the current version of ParCeL for shared memory systems - ParCeL-6.1, when running a program implemented with the help of its easy to use semantics.

The final step would be to obtain the conclusions and to think about the perspectives. This will be done by taking into consideration all our results on the cluster:

- the results and performances of the DSM Kerrighed (Page Misses Impact and Synchronization Barrier)
- the results and performances of ParCeL-6.1 when running on Kerrighed
- if necessary, the results and performances exhibited when running other ParCeL versions on clusters and SMPs

The next topics will present the results achieved with ParCeL on the cluster.

The first tests we ran on Kerrighed had the role to show its performances in the case of some classical situations when speaking of a software DSM - *Page Miss Impact* and *Synchronization Barrier*.

At the beginning, we had to take a decision on the number of threads that lead to the most significant results in the case of our cluster (four machines P4 2.4GHz with 1GB main memory connected through Gigabit Ethernet). We did this by running a classical pthread Jacobi Relaxation. The diagram in Figure 5.2 shows that the significant results in our case are obtained for one thread per node in the cluster (and implicitly for one thread per processor) - in other words: when we have *multithreading* and we are sure that we don't have *hyperthreading*. Maybe sometimes will be intersting to find out what is happening when running two threads per node in the cluster to see if any *hyperthreading* influence.

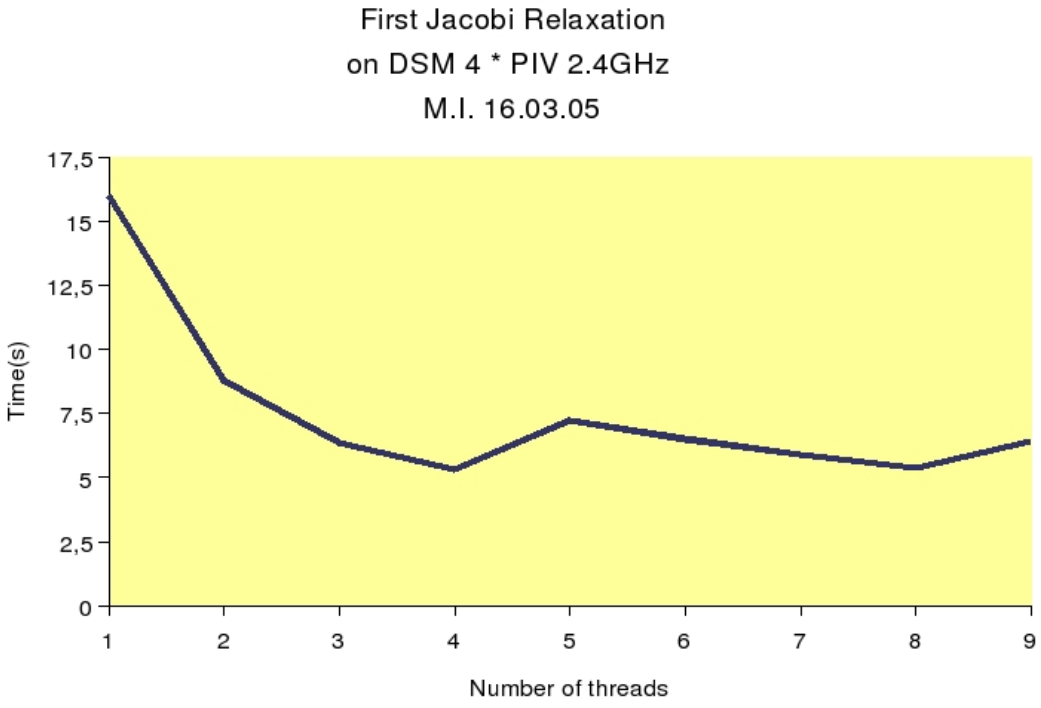


Figure 5.2: First Jacobi Relaxation that Ran On Kerrighed

Between 4 and 8 number of threads used in the application, we can see that the simple fact that there is no load balancing - we don't have the same number of threads on each node of the cluster, may introduce some delay due to the scheduling algorithms that are applied.

Assigning an inequal number of threads to the nodes in the cluster may be combined with some delays in page miss handaling and this could be the reason for the fact that we don't have any fall in the exacution times between 4 and 8 threads.

The next test had the role to determine the behaviour of Kerrighed in the case of an *optimal* parallelism - an embarrassingly parallel application. In the application there were no memory allocations (Figure 5.3), and each thread made processing on some local variables without any interaction with the other threads. The behaviour of Kerrighed in this case showed that a high SpeedUp value can be achieved when increasing the number of threads. The execution time is deacreasing as much as the number of threads increases.

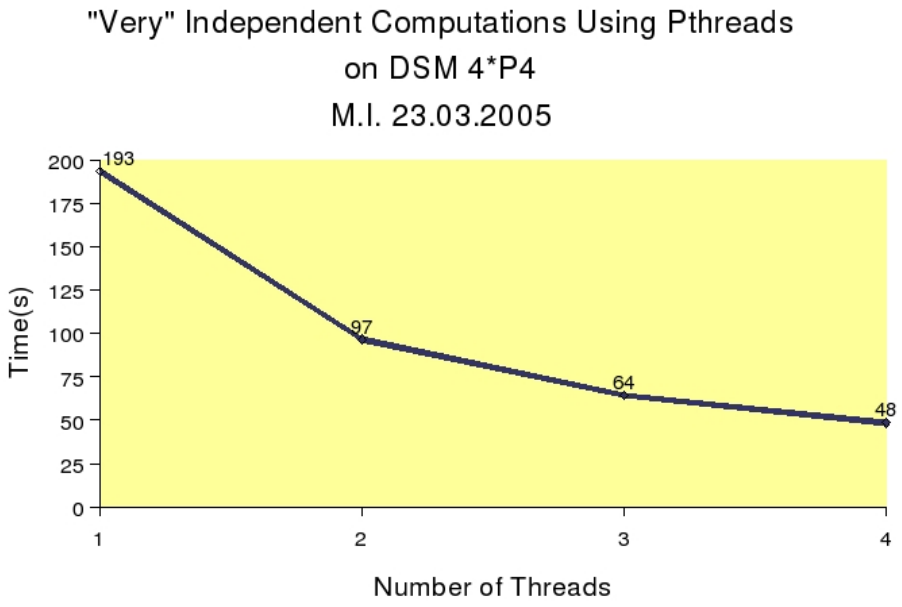


Figure 5.3: An Application with Independent Pthread Computations that Ran On Kerrighed

In the Figure 5.4 there are presented the results that we have obtained when running an application that resembles Jacobi Relaxation, excepting threads execute independent computations and we have no page memory misses. The results were very encouraging, showing that in the case of analogus (very parallel) applications, SpeedUp can be efficiently obtained on Kerrighed. A hyperSpeedUp is obtained in this case: the ideal value of the SpeedUp equals at most the number of running threads, but in the case of 2 threads, for example, it is almost 3.

The next tests were made to evaluate the performances in handaling a large number of memory access requests - a very high frequency of memory page misses (Figure 5.5). The application was a kind of Jacobi Relaxation in which each pthread read the memory pages managed by the others and had the role to demonstrate the efficiency or the inefficiency of Kerrighed to handle the page faults. No SpeedUp was achieved in this case.

In the Figure 5.5, the results show that the passing from local memory to global memory of the cluster (from local buses to Gigabit Ethernet) (from 1 thread to 2 threads) increases the execution time for about 5 times. Moreover, if continuing with increasing the number of threads, the execution time doesn't decrease under the execution time obtained for 2 threads. We hoped that running ParCeL-6.1 on Kerrighed won't generate such a high frequency of

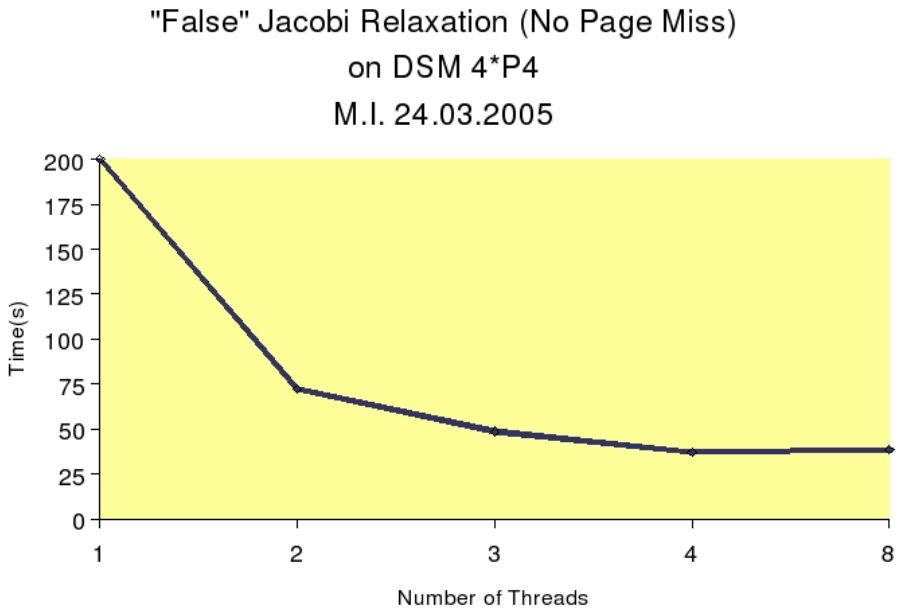


Figure 5.4: Pthread Application on Kerrighed with No Pege Misses

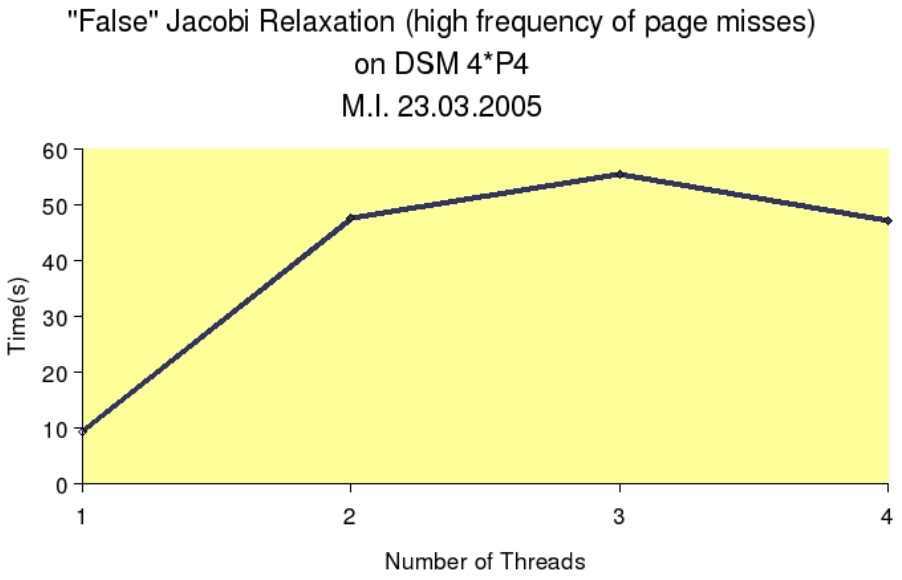


Figure 5.5: Pthread Application on Kerrighed with a lot of Page Misses

page memory misses.

Another important aspect in benchmarking Kerrighed was to see the importance of the *false sharing* effect - if there is any influence given by the fact that the memory used in the program is page aligned or not (Figure 5.6 and Figure 5.7). The importance of the false sharing and its effect in increasing the execution times is more obvious for a number of threads equal or less then 75% from the number of nodes in the cluster. It seems that using the page aligned memory really increases the performances by avoiding *false sharing*, but with the

disadvantage of some memory waste.

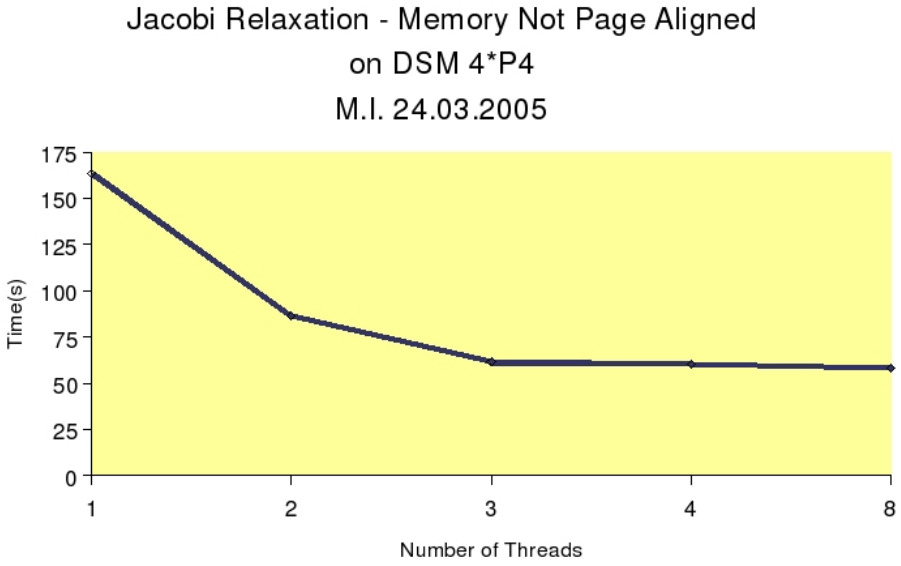


Figure 5.6: Pthread Application on Kerrighed using Memory that is Not Page Aligned

Using the page aligned memory will eliminate the *false sharing* effect.

True sharing between threads happens when two threads access the same data in a concurrent manner. Even if access to this data is protected by mutexes for correctness of the program, this sharing should be avoided where possible.

False sharing between two threads happens when two threads access different data situated in the same memory page. When speaking about performances, this is the same problem as *true sharing*. Avoiding *false sharing* is much difficult than true sharing, because examination of the logical structure of the code is not sufficient to detect when it will occur. If possible, variables used by different threads should not be located in the same memory page. The memory padding (creating unused memory zones so that data-structures are page aligned) is often a good solution.

We have calculated the SpeedUp for a program that does a Relaxation in the most significant situations for the page memory misses impact (no page memory misses, page aligned memory, memory not page aligned - in the last two cases, the application becomes a classical Jacobi) and we have centralized the results in the Figure 5.8. The best SpeedUp values are obtained in the case of no page misses and in the case of using page aligned memory as we expected. The results were very encouraging and they showed that a parallel application with a low frequency of page memory misses can be transparently and efficiently parallelized on a cluster with Kerrighed. As the reader can see in the 5.8, in the case of no page memory misses, when running the application with 4 pthreads, a high value of SpeedUp can be observed in the diagram - the value of the SpeedUp is greater than the ideal SpeedUp value that is 4, so we have obtained *HyperSpeedUp*.

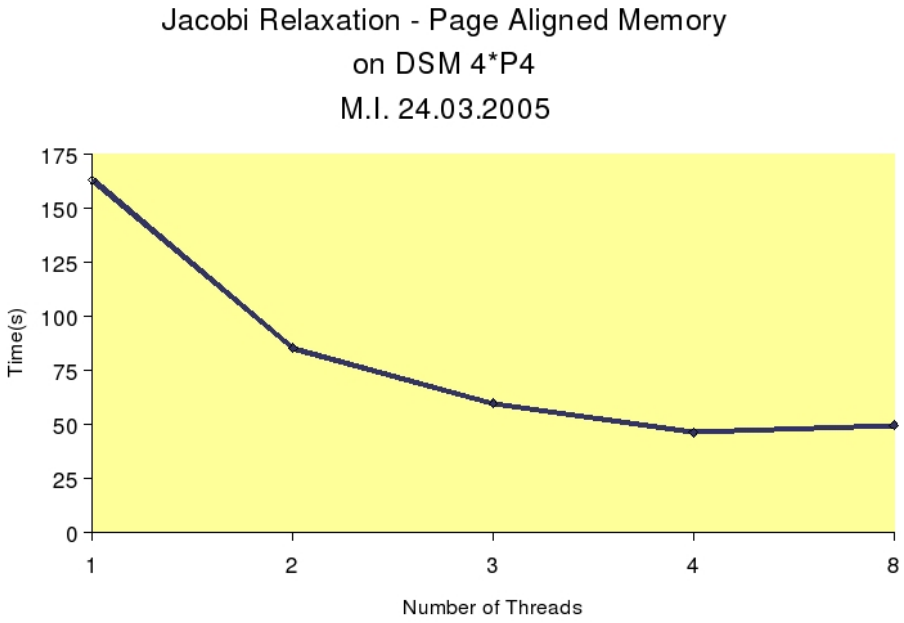


Figure 5.7: Pthread Application on Kerrighed using Page Aligned Memory

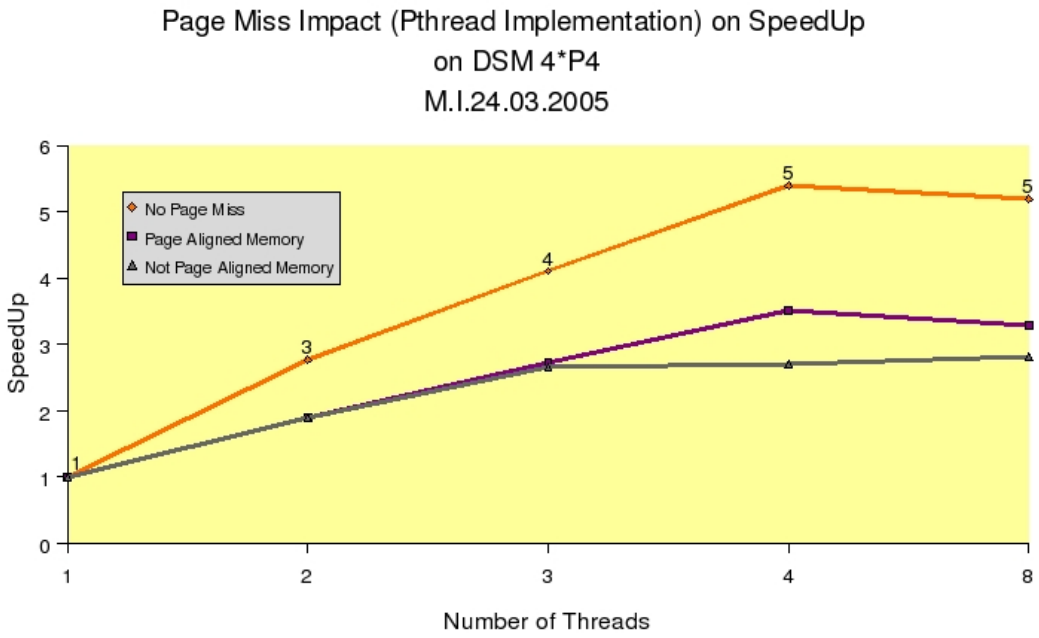


Figure 5.8: The SpeedUp for a Relaxation Application Running on Kerrighed

5.6 MPI Compatibility

The current sub-models of ParCeL-6 (ParCeL-6.1 and ParCeL-6.2) are optimized and implemented for architectures supporting memory sharing paradigm and respectively for architectures supporting only message passing paradigm. Kerrighed may offer the posibility to take

benefit of both paradigms on the same parallel or distributed architecture. This is the reason that we were interested to see if Kerrighed is completely MPI compatible.

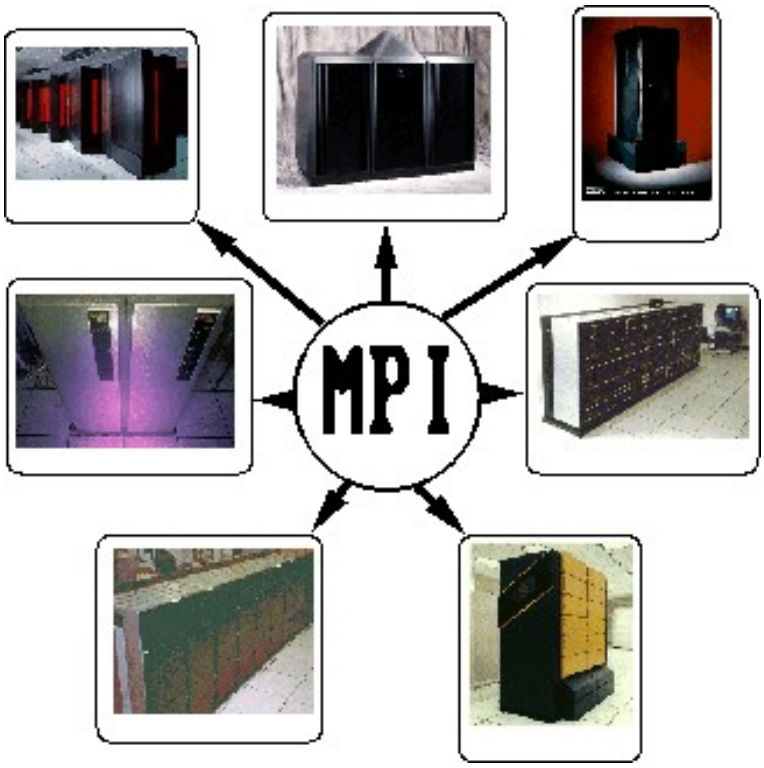


Figure 5.9: Large utilization of MPI

Because MPI uses (by default) sockets in order to allow the communication between processes on different machines, Kerrighed installed system must be instructed with the available Kerrighed capabilities to link the created sockets to *Kernet streams* and to use a special *rsh* program for process deployment - *krg-rsh*. The MPI library and the runtime will be used unchanged. With the configurations mentioned above, Kerrighed scheduler will menage the deployment of the MPI program.

In order to install MPICH to be menaged by the Kerrighed Scheduler, the usr sould follow the steps:

- download and unpack MPICH. We have donloaded *MPICH-1.2.6* tar archive and we have unpacked it in */usr/local/src/*
- set the RSHCOMMAND to

  `KERRIGHED_PATH/bin/krg-rsh`

- install MPICH. There are no other Kerrighed requirements.
- We have first tried to run an MPI application with the hints given with Kerrighed: we created a *machine file* (machine.file for example), containing one line:*localhost*; this solution was not satisfactory in our case because the processes were executed on

only one node of the cluster. When we launched the application with a simple command like *mpirun -np 4 mpi_test*, the processes were deployed on all the four nodes of the cluster. In all the cases, the executable file *mpi_test* was placed in the directory */usr/local/kerrighed/tests/* on all the four machines of the clusters.

- in the terminal in which are launched the MPI applications, in order that *Kernet streams* to be used, the following command must be executed:

```
krg_capset -d +USE_INTRA_CLUSTER_KERSTREAMS
```

- the application should be launched with a command like:

```
mpirun -np 4 -machinefile ~/machine.file mpi_program
```

We have noticed that the processes were equally deployed on all the nodes of the cluster when launching the MPI applications (we have used this way often) with a command like:

```
mpirun -np 4 mpi_program
```

The Kerrighed engineers say that there may be some critical cases when running MPI on the cluster that must be considered:

- MPI processes are deployed and managed by the current scheduler: they therefore theoretically can be migrated to balance the load on all nodes, and be deployed in an unexpected way if other programs are running on the cluster.
- Simple schedulers may not be very robust to process deployment using hierarchical methods.

The first test that we made with MPI on Kerrighed had the role to see if Kerrighed and MPI can run independently and if Kerrighed introduces some great overhead in the MPI communication. The application did only a lot of broadcasts and message sending between processes. MPICH was installed and configured to use the Linux provided *rsh* program and no *krg_capset* command was executed in the terminal in which the MPI applications were launched. In other words, the first MPI test was ran by totally ignoring the fact that Kerrighed is installed on all the four nodes of the cluster.

In order to run the application, we have fallowed the steps:

- *normal* installation of MPICH on all the four nodes of the clusters. The fact that Kerrighed was installed on the cluster was ignored.
- the fallowing *machine.file* was created on the machine *monox1* in */usr/local/kerrighed/tests*:

```
monox1.grid.metz.supelec.fr
monox2.grid.metz.supelec.fr
monox3.grid.metz.supelec.fr
monox4.grid.metz.supelec.fr
```

- the executable file - *mpi_test*, was placed on all the four machines of the cluster in */usr/local/kerriged/tests/*.
- the application was launched on *monox1* with commands like:

```
mpirun -np 4 -machinefile machine.file mpi_test
```

The application was ran several times in two cases:

- when Kerrighed was not running on the cluster
- when Kerrighed was running on the cluster

The results for running an intensive MPI communication test can be observed in Figure 5.10. In the diagram, we can observe that the delay introduced by the fact that Kerrighed

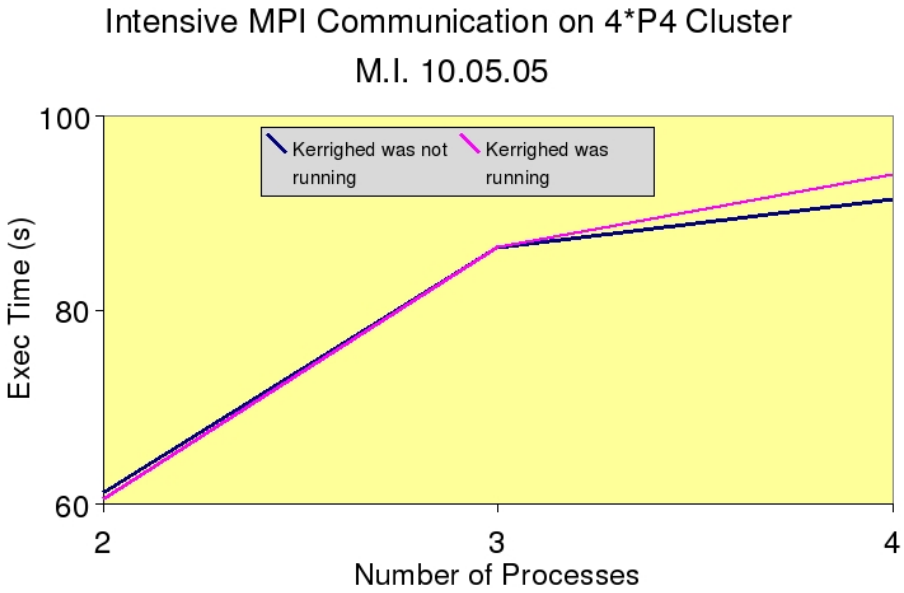


Figure 5.10: The first MPI test on the cluster where Kerrighed was installed

is running is parctically insignificant for the cases when the application is ran with 2 or 3 processes. When the application is ran with 4 processes, this delay has the value of about 2 seconds. We recommand the studying of this difference between the execution times when the increasing of the Kerrighed cluster will be needed.

5.7 IPC Compatibility

One of the advantages provided by Kerrighed is that System V memory segments - and consequently IPCs like shared memory, can operate through the whole cluster, just like on a SMP machine. We wanted to test the usage of the IPC shared memory in order to take knowledge of all the details that are needed for proper use on the cluster. Another test

was done in order to see if we can take benefit of the combination of MPI and IPC on the Kerrighed cluster.

In the first test - a short MPI application that was a data integrity test for the IPC shared memory of Kerrighed, the next steps were followed:

- the executable file *mpi_test* was copied on all the four machines of the cluster in */usr/local/kerrighed/tests*. The application was launched as it fallows and the processes were automaticaly deployed on all the nodes of the cluster:

```
krg_capset -d +USE_INTRA_CLUSTER_KERSTREAMS
mpirun -np 4 mpi_test
```

- the process with the process identifier 0 (named in the fallowing *root process*) created the shared memory with the instruction

```
shmid=shmget (IPC_PRIVATE, GLOBAL_SIZE, IPC_EXCL|IPC_CREAT| 0777)
```

- the id obtained by the root process was then broadcasted to all MPI processes. This processes attached to the shared memory that they received from the root process. Each process used exclusively an equal part of the shared memory. They attached themselves to the shared memory with the instruction:

```
shmat(shmid, 0, 0 )
```

- the process with the id 0 send with *MPI_Scatter* some data to the other processes.
- the MPI processes copied the data received from the root process into their corresponding portion of the shared memory. The root process attached himself to the shared memory.
- all the processes executed *MPI_Barrier*.
- the root process tested if the data that was send to the MPI processes with *MPI_Scatter* was identical with the data that was now in the shared memory and it was, so the data integrity test for the IPC shared memory of Kerrighed was succesfully passed.
- The shared memory was destroyed by the root process with the instruction:

```
shmctl(shmid, IPC_RMID, &result);//result is (struct shmid_ds)
```

The second test had the role to see if we can take advantage of both MPI and IPC shared memory in the same time. In other words, we wanted to see if the IPC shared memory can be directly used by the MPI processes as receiving buffers.

The application was ran as it was previously mentioned, except the fact that in *MPI_Scatter* all the processes used as receiving buffer their corresponding part of the shared memory buffer. The root process verified then that the data in the shared memory is identical to the data that it send to the MPI processes. The data was identical and the test was successfully passed.

Conclusion Now we can easily emphasize one important advantage of using Kerrighed on a cluster: we can take benefit in the same time of both message passing paradigm and shared memory paradigm, and consequently we can have a transparent cluster wide access on an IPC shared memory.

Chapter 6

Running ParCeL-6.1 on Kerrighed DSM system

We have installed and run Kerrighed 1.0.0 on a cluster containing four machines P4 2.4GHz with 1GB main memory connected through Gigabit Ethernet. In order to take benefit of the advantages provided by this DSM (in this particular case - migration of threads on all the nodes of the cluster), we had to compile our ParCeL-6.1 applications using the library *krgthread* instead of *pthread*. When using this library, the function *pthread_create* uses the mechanism of remote process creation offered by the module *Aragorn*. In this way, all the pthreads in the application will be deployed by Kerrighed on all the nodes of the cluster providing load balancing and process/thread migration.

6.1 Barrier Comparison: *Native* Kerrighed barrier and ParCeL-6.1 *Handmade* barrier

The first main interest was to determine which barrier has better performances when used in an application that runs on Kerrighed. We have created a small program that just made 1 million barrier calls for a certain number of pthreads given as argument. This application had the posibility to run either with the barrier implemented with posix semaphores in ParCeL-6.1, either with the barrier implemented in Kerrighed (*pthread_barrier_t*) - the option was chosen by the value of a compilation flag.

In ParCeL-6.1 we have implemented a barrier that uses four posix semaphores and a shared variable (an integer); therefore the barrier is portable and confoming to the Posix standard. This barrier is in fact a barrier for the Linux kernel version 2.4 that we have used - this kernel did not provide any barrier. The access times for this barrier are linear and their complexity is *O(2*P)*. The efficiency of this barrier was proved by the tests that we have made on a four processor SMP. The processors of this SMP were Pentium3 at 700MHz and the main memory had 1GB. The results for 1 million calls on this efficient and portable barrier for a certain number of threads can be seen in the Figure 6.1.

The benchmarks on Kerrighed had the role to establish if the native synchronization barrier provided by this DSM has better performances (and if so, how much better?) in the distributed environment, then the efficient SMP barrier implemented in ParCeL-6.1. The results of running the ParCeL-6 posix semaphores barrier and the Kerrighed barrier can be

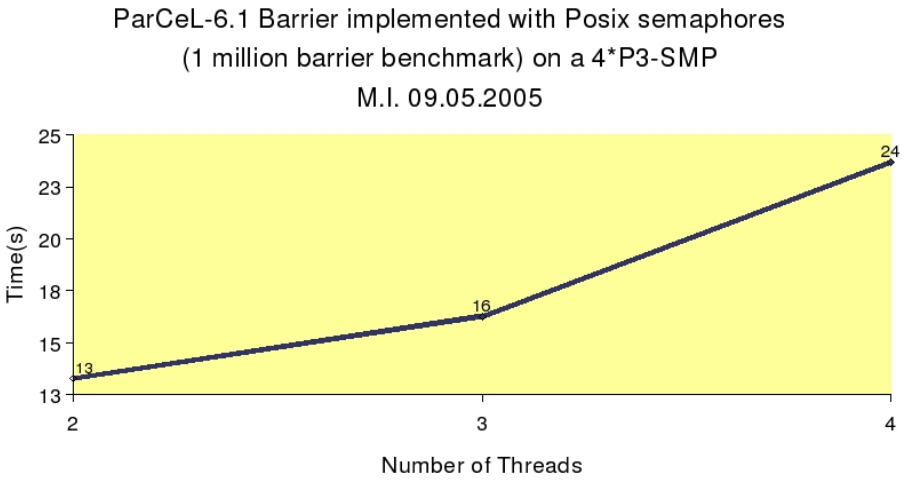


Figure 6.1: The Behaviour of the ParCeL-6.1 Barrier Implemented with Posix Semaphores on a 4*P3-SMP

observed in the Figure 6.2 for the barrier implemented in ParCeL-6.1 and in the Figure 6.3 for the barrier implemented in Kerrighed.

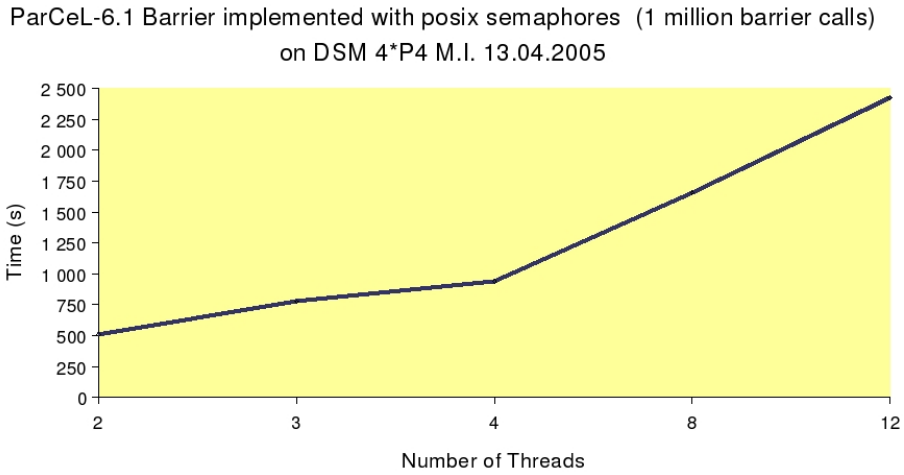


Figure 6.2: The Behaviour of the ParCeL-6.1 Barrier Implemented with Posix Semaphores on Kerrighed - 4*P4 cluster

As it was more or less expected, the barrier implemented in Kerrighed for applications that run on Kerrighed was faster, and in fact a lot faster, than the barrier implemented with Posix semaphores (that was efficient on a SMP). The execution times for the two barriers are compared in Figure 6.4. A very important observation could be emphasized here: the native Kerrighed barrier showed an almost constant execution time when increasing the number of threads, and this happens for sure in the case of *multithreading* and maybe this happens also in the case of *hyperthreading* (running enough threads on a single node of the cluster in order to activate hyperthreading). Some benchmarks could be done to study the behaviour in hyperthreading in order to see if we can have both multithreading and hyperthreading on

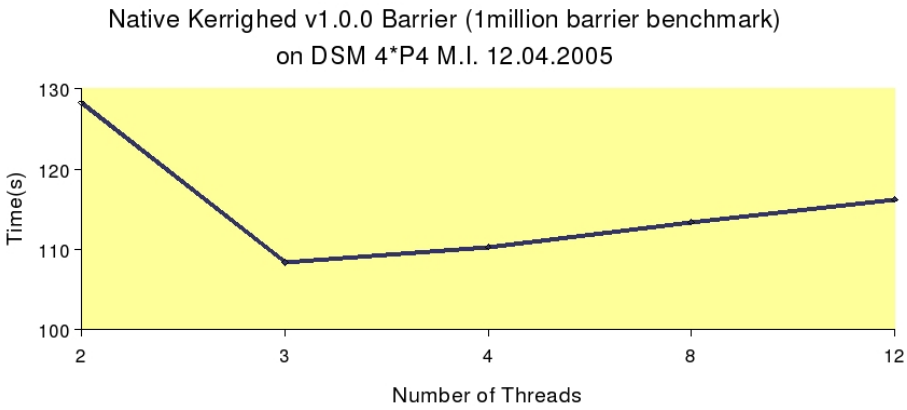


Figure 6.3: The Behaviour of the Barrier *pthread_barrier_t* Implemented in Kerrighed - 4*P4 cluster

the Kerrighed cluster, but this is not one of our interests in this phase.

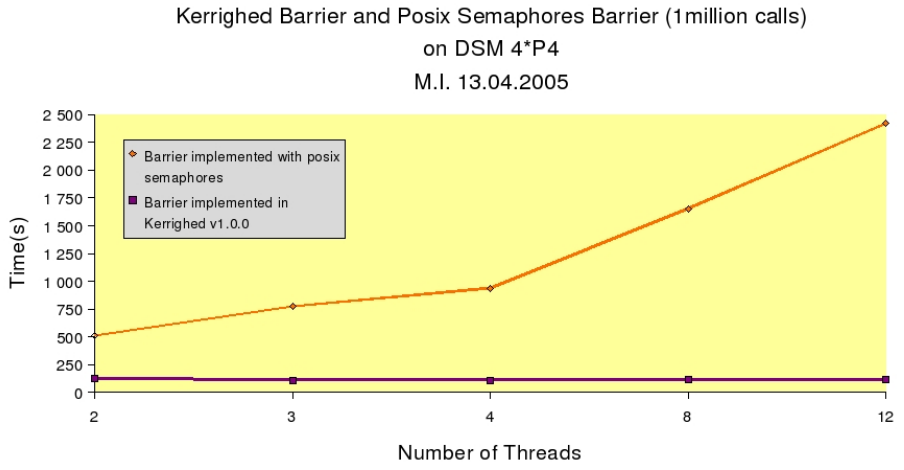


Figure 6.4: Comparison between the Behaviour of the Barrier Implemented with Posix Semaphores in ParCeL-6.1 and the Barrier *pthread_barrier_t* Implemented in Kerrighed

We can formulate the conclusion for our synchronization case study.

- The portable barrier implemented by us when running on the 2.4 Linux kernel exhibited good performances when running on a four multiprocessor machine, even if the processors were much older and less fast than the processors that were on the four machines in the cluster.
- The Kerrighed barrier showed very good performances and achieved constant execution times, even if the number of threads increased.
- The cost for running the native Kerrighed barrier in the case of our cluster (4*P4-DSM) is about *five* times higher than running the efficient posix barrier implemented in ParCeL-6.1 on a multiprocessor machine (4*P3-SMP).

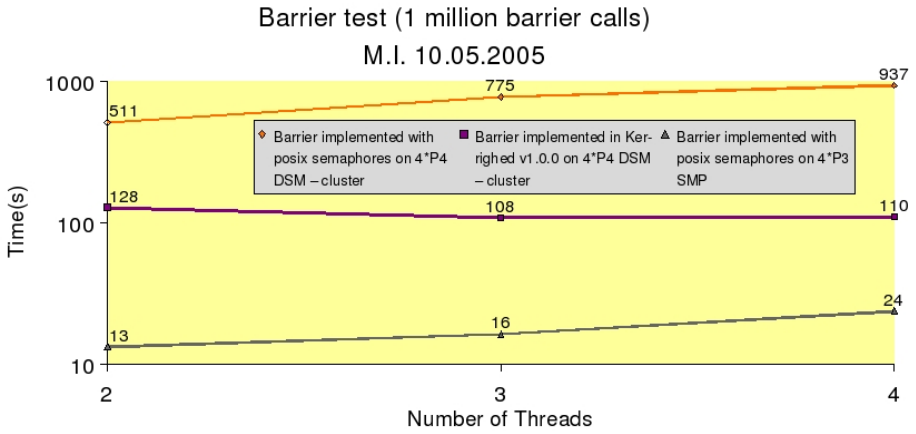


Figure 6.5: Execution Times of the Barrier Implemented with Posix Semaphores in ParCeL-6.1 and the Barrier *pthread_barrier_t* Implemented in Kerrighed on 4*P4-DSM and 4*P3-SMP

6.2 Benchmarking ParCeL-6.1 on Kerrighed DSM

The next tests had the role to see the behaviour of a ParCeL-6.1 application on the DSM Kerrighed, knowing its behaviour when ran on a single machine of the cluster. We used for these tests, when performed on the DSM, the barrier implemented in Kerrighed because it showed better performances than the barrier implemented in ParCeL-6.1 as mentioned in the previous section. For the tests performed on a single machine of the cluster, we have used the ParCeL-6.1 barrier (because Kerrighed v1.0.0 couldn't run on a single machine and we couldn't use the *pthread_barrier_t* in this case).

As the results in the Figure 6.6 show, the execution times when using a single machine are not very good for a small number of threads, but they are satisfactory when increasing the number of threads used in the application and hyperthreading is activated.

When running the same application on Kerrighed V1.0.0 (on the the four machines of the

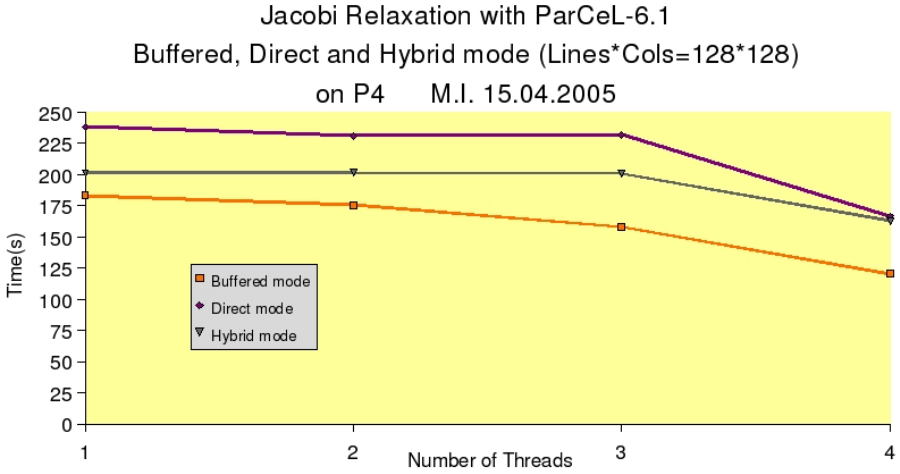


Figure 6.6: Jacobi Relaxation Implemented with ParCeL-6.1 on One Machine of the Cluster (15000 iterations)

cluster), the results were not good at all, showing that ParCeL-6.1 on Kerrighed is not a good solution to obtain SpeedUp, as we can see in Figure 6.7. We could easily observe that the execution times were much greater than in the case of using a single machine of the cluster for the same application. The conclusion is that the memory model used in ParCeL-6.1 is more complex than we expected from the point of view of a DSM and we should take into consideration for the future to adapt ParCeL-6 to run efficiently on a DSM.

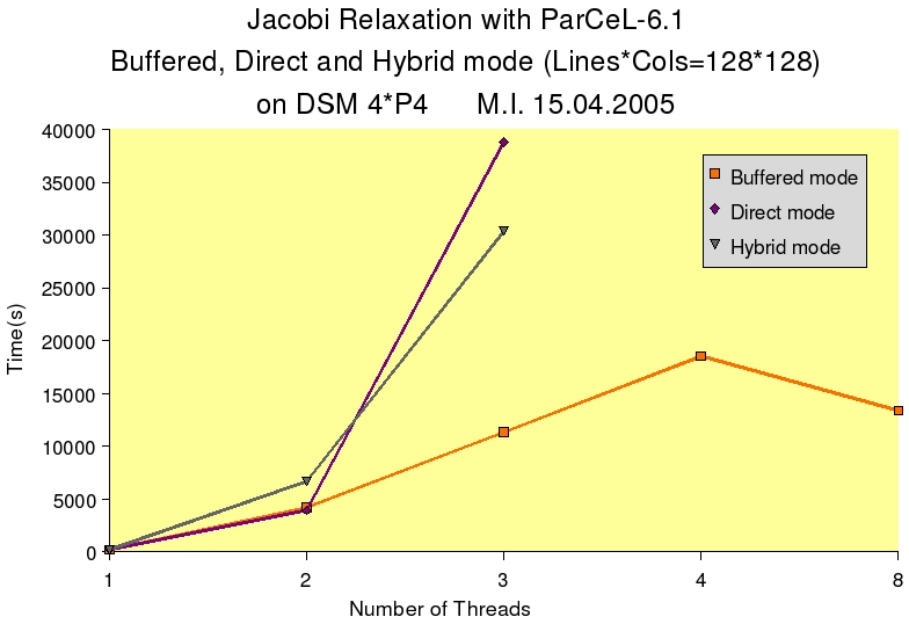


Figure 6.7: Jacobi Relaxation Implemented with ParCeL-6.1 on The Cluster 4*P4 DSM Kerrighed V1.0.0 (15000 iterations)

We did the same tests on Kerrighed V1.0.2 installed on the kernel 2.4.29. In this case, the application ran about 40% faster, but we didn't obtain SpeedUp either and the execution times maintained at a very high level in comparison with the execution times of the same application on Kerrighed V1.0.0. The results are ilustrated in Figure 6.8.

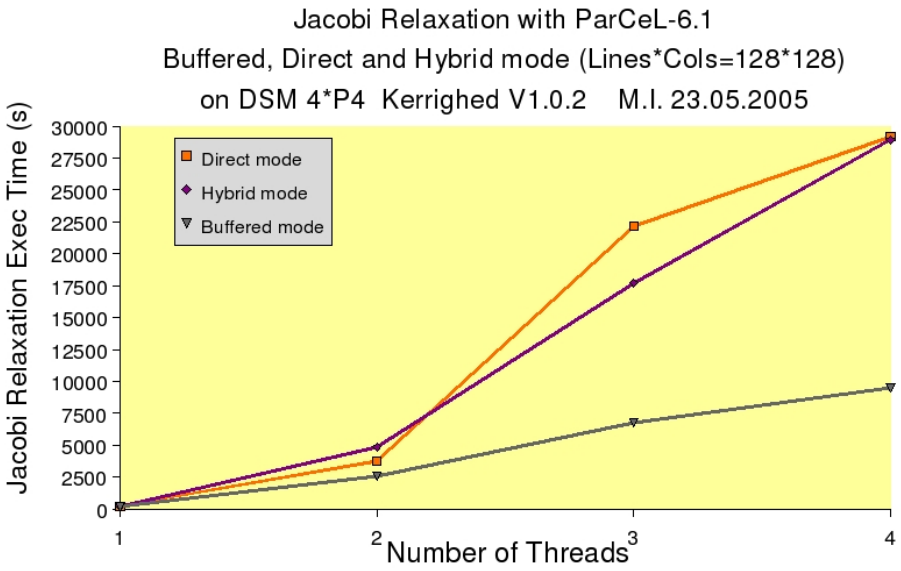


Figure 6.8: Jacobi Relaxation Implemented with ParCeL-6.1 on The Cluster 4*P4 DSM Kerrighed V1.0.2 (15000 iterations)

Chapter 7

ParCeL-6 Project from DSM's point of view

7.1 Considerations on future development

A typical ParCeL-6/Grumpf application has both sequential and slow interactivity parts and intensive computations parts. ParCeL-6 with message passing can run on the DSM handaling massive parallel calculus parts as can be observed in the Figure 7.1, while the slower and sequential parts can run directly on the DSM without introducing any delay in the execution and taking benefit of the advantages provided by this DSM such as cluster wide memory space and consequently, cluster wide interactivity.

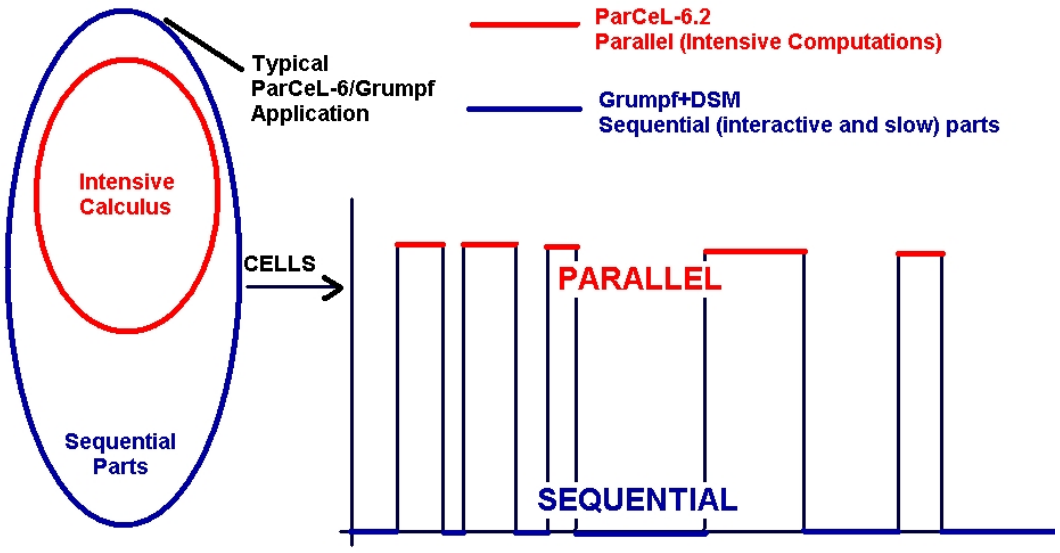


Figure 7.1: Running a typical ParCeL-6/Grumpf Application on the DSM Kerrighed

A generic application was written. The main objectives of this application were:

- taking benefit of the cluster wide memory space in the way needed by ParCeL-6, by using the shared memory IPC provided by Kerrighed

- taking benefit of the high scalability provided by the usage of MPI
- finding new openings for future development of ParCeL-6

A typical ParCeL-6 application consists in sequential and parallel parts. In the sequential parts, the process with the id zero (*master* process) needs to easy access the global variables used in a ParCeL-6 program. These are the slower parts which can be executed without a meaningful delay over the cluster wide memory space provided by the DSM. The parallel parts will be executed by accessing only the local memory of each process and will mantain the high prallelism performances over the cluster provided by the MPI implementation.

A short description of the application is provided in the following:

- each process has a linked list of shared memory IPCs
- an entry in the list contains a pointer to the next entry, an id of a shared memory IPC and a *void* pointer to a memory space of constant size where a table used in ParCeL-6 can be stored. This pointer indicates the address in the space of the process where the shared memory with the memorised id is attached.
- each process can allocate as many shared memories it needs in order to store the tables used in ParCeL-6
- at the end of an intensive computations cycle, the master process needs to access the global variables used in the program. First, it receives the identifiers of the shared memories created by each process in the computation cycle and then it can read the data in these memory spaces by attaching them in its own memory space.

In the figure 7.2, there can be observed the means of handling the shared memory IPCs on a DSM like Kerrighed.

In the future developments, some problems might appear when it is necessary to access the global variables used in ParCeL-6, not in the sequential parts, but in the intensive computations parts. In this case, it might appear some delays ought to the unexpected communications between the processes. A solution would be to plan in advance the handling of such communications and to solve them efficiently in the sequential parts. When global access is needed only in the sequential parts, no delay will be introduced and the master process can easily read all the data from the other processes.

The Resulting Structure We thought to use in the future, as entry in the list with shared memories, a structure with the following members:

- *table* - table with the structures used in ParCeL-6.2
- *crt_shm_id* - identifier of the current shared memory
- *shm_id* - identifier of the next shared memory
- *adr* - address of the shared memory *shm_id* in the current address space of the process

Obs: Each entry of the list will be allocated in a shared memory.

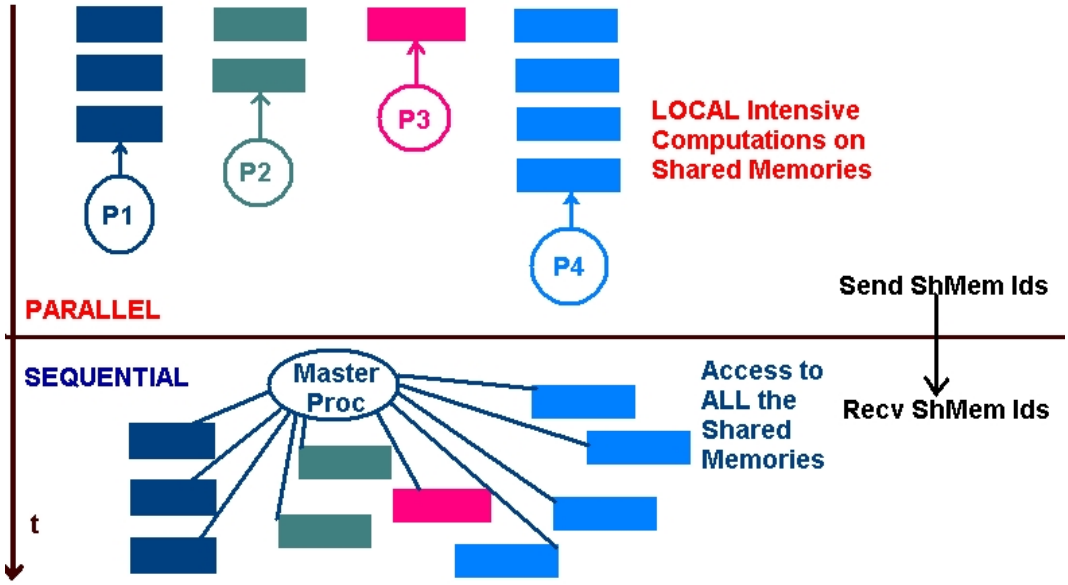


Figure 7.2: Running a ParCeL-6 Application taking benefit of the cluster wide shared memory provided by the DSM

7.2 Example of a DSM Application that uses ShMems IPC

```
/.....................................
int main (int argc, char *argv[])
{
/.....................................

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&no_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&proc_id);
/*each process creates its own shared memories and it memorises in a linked list their ids
  for( i=0 ; i < ShMems_PER_PROC ; i++)
  {
    shmid=createShmId();
    addShMemToList ( shmid, TRUE);
  }

  for( i=0 ; i < ShMems_PER_PROC ; i++)
  {
    buffer = (int*) getTable(i);
    buffer[0] = proc_id;
    for ( j=1 ; j < TABLE_SIZE ; j++ )
/*write some data that will be read by the master process, in the shared memories*/
      buffer[j] = i;

    list_size = getListSize();
```

```
/*each process will have an array with all the ids of the shared memories*/
    if ( proc_id == 0 )
      shmids = returnShmIds(FALSE);
    else
      shmids = returnShmIds(TRUE);
/*the master process receives the number of shared memories created by each process*/
    if (proc_id==0)
      if((no_shms_proc=(int *)malloc(no_procs*sizeof(int)))==NULL) error();
    else
      if((no_shms_proc=(int *)malloc(sizeof(int)))==NULL) error();
    MPI_Gather (&shmids[0], 1, MPI_INT, no_shms_proc, 1,MPI_INT, 0, MPI_COMM_WORLD);
/*only in the master process*/
    if (proc_id==0)
    {
      no_all_shmids=0;

      for ( i=1 ; i<no_procs ; i++ )
        no_all_shmids+=no_shms_proc[i];

      if((all_shmids_aux=(int *)malloc(no_all_shmids * sizeof(int)))==NULL) error();
/*the master process receives the ids of all the shared created by the other processes and
      for ( i=1 ; i<no_procs ; i++ )
      {
        if(no_shms_proc[i]>0)
{
          MPI_Recv ( all_shmids_aux, no_shms_proc[i], MPI_INT, i, 0, MPI_COMM_WORLD, &stat
          for ( j=0; j<no_shms_proc[i] ; j++ )
            addShMemToList ( all_shmids_aux[j], TRUE);

          free(no_shms_proc);
          free(all_shmids_aux);
}
        else
        {
/*the worker process sends the list with its previous created shared memories, to the mast
          if ( list_size > 0 )
            MPI_Send( shmids+1, list_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
          free(no_shms_proc);
        }

        if(proc_id==0)
        {
/*the master process reads the data received in the shared memories from the other process
          list_size = getListSize();

          for ( i=0 ; i < list_size ; i++ )
          {
```

```
            buffer = (int*) getTable(i);
            printf("table=%d;proc_id=%d;page_no=%d;\n", i,buffer[0], buffer[1]);
            buffer[0] = proc_id;
            buffer[1] = i;
          }
/*the master process destroies all the shared memories*/
          deleteAllShMemsList();
        }

        MPI_Finalize();

        return 0;
}
```

Chapter 8

Conclusions and Perspectives

The DSM experiment done at Supélec helped to the elaboration of new points of view when speaking about what can be done with parallel and distributed cellular languages on parallel and distributed architectures.

8.1 Conclusions regarding the DSM experiment

A DSM, in our case the Kerrighed DSM, has some limitations. In case of *Embarassingly Parallel Computations*, a good value of the SpeedUp can be achieved - practically, the ideal SpeedUp value, but in the case of *Irregular Memory Accesses* and High Frequency of Page Memory Misses, an obvious performance slowdown is exhibited.

ParCeL-6.1 worked on the DSM without any problems, but with poor performance showing that this submodel of ParCeL-6 is not really appropriate for running DSM applications. Though the applications implemented in ParCeL-6.1 can run in a transparent manner for the user on a Linux cluster with the help of the Kerrighed DSM.

8.2 Perspectives for ParCeL-6 Project

The benchmarks with ParCeL-6.2 - the ParCeL-6 submodel for architectures that support message passing paradigm, showed that very good performance can be achived on clusters. In the Figure 8.1 can be observed that the usage of the pure MPI version of ParCeL-6.2 can lead to important performances when running applications over clusters. The Figure 8.1 ilustrates the execution times that were obtained for creation of a large number of cells, when increasing the number of the machines used in the cluster. The machines that were used were 32 Pentium4 machines connected through 2 switches Gigabit Ethernet.

The other direction of research would be to take advantage of the high level of interactivity offered by a DSM in the future developments of ParCeL-6.

8.3 Previous and Future Steps for ParCeL-6 project

In the Figure 8.2 can be observed the phases followed by the ParCeL-6 project. The first phase of ParCeL-6 development was to run ParCeL-6.1 on a SMP with the help of posix threads. This phase was accomplished successfully. Though, the solution provided was not

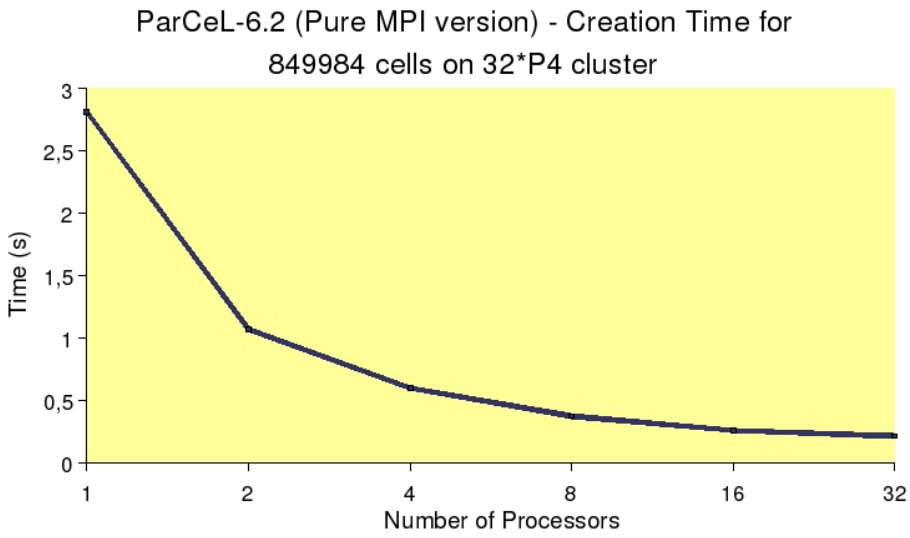


Figure 8.1: The SpeedUp value that can be achieved for Creation of Cells with ParCeL-6.2

scalable. The following phase was to increase the scale for using ParCeL-6.1, and we passed - with the help of the Kerrighed DSM, to the execution of the applications on a Linux cluster. The scale was increased, because now we had the possibility to run ParCeL-6.1 on an entire cluster, but with the limitation of 32 nodes in the cluster. The scaling level was satisfactory, but the performance was poor and this solution proved to be inefficient.

The performances exhibited by ParCeL-6.2 on parallel machines showed that this can be a good basis for future development. When running this submodel of ParCeL-6 implemented only in MPI, it proved to be an efficient solution but liable to become less efficient on large scale machines that are not grouped and optimized. This disadvantage can be prevented by using the SSCRAP library developed by the AlGorille team at LORIA together with MPI. SSCRAP communications appear faster on large scale systems.

The perspectives are to run ParCeL-6.2 implemented with SSCRAP and MPI and in the same time to take benefit of the advantages provided by the Kerrighed DSM. The cellular calculus will be efficiently handled on SSCRAP/MPI, meanwhile the user interactivity will be facilitated at DSM Kerrighed level.

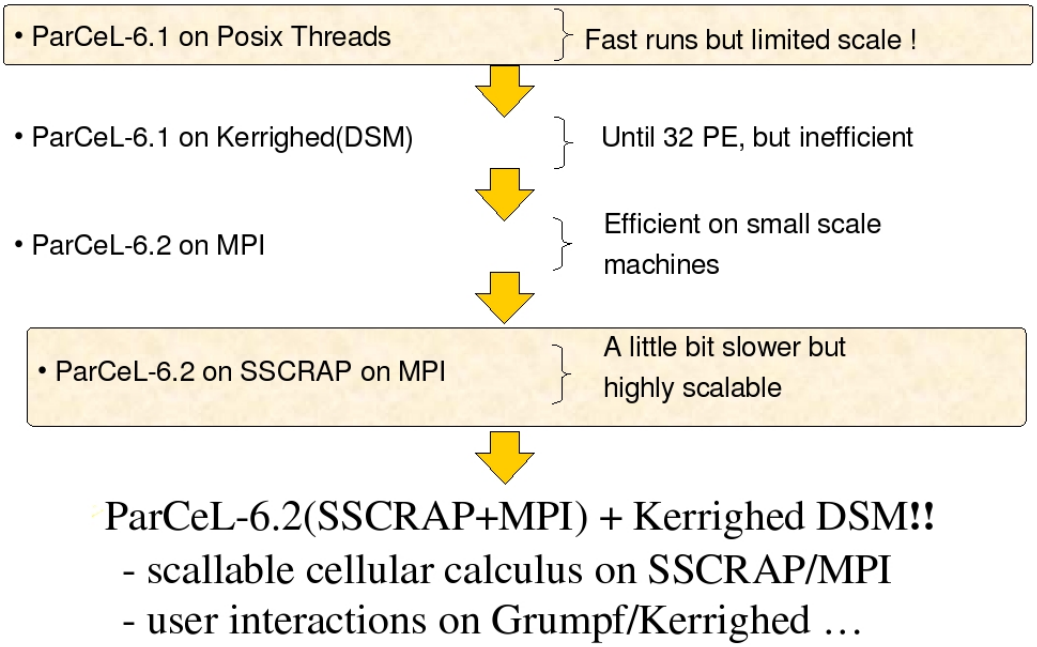


Figure 8.2: The steps followed by the ParCeL-6 project

Bibliography

[1] *Theory of Self-Reproducing Automata*. Edited and completed by A. W. Burks, University of Illinois Press, Illinois, 1966

[2] *CELLSIM II User's Manual*, C.E. Donaghey, U Houston, September 1975

[3] H. Szwerinski, H.-J. Brede, *Sicela Simulation Zellularer Automaten*, Technische Universität Braunschweig, 1979

[4] *A fast cellular automata simulator with Windows GUI*, Bob Fisch, David Griffeath

[5] Friedhelm Seutter, *CEPROL: A cellular programming language. Parallel Computing*, 1985

[6] *Linear Cellular Automata*, Harold V. McIntosh, Universidad Autonoma de Puebla,Mexico, May 20, 1987

[7] Kai Li, *IVY: A Shared Virtual Memory System for Parallel Computing*, Proceedings of the 1988 International Conference on Parallel Processing, August 1988

[8] Bal, H., E., Tanenbaum, A., S., *Distributed Programming with Shared Data*, International Conference on Computer Languages '88, October 1988

[9] Li, K., Hudak, P., *Memory Coherence in Shared Virtual Memory Systems*, ACM Transactions on Computer Systems, November 1989

[10] Fleisch, B., Popek, G., *Mirage: A Coherent Distributed Shared Memory Design*, Proceedings of the 14th ACM Symposium on Operating System Principles, ACM, New York, 1989

[11] Rudy Rucker, John Walker, June 1989, *Rudy Rucker's Cellular Automata Laboratory*

[12] UHP (University Henry Poincaré, Nancy) Master and PhD thesis of Thierry Cornu

[13] Bennet, J. K., Carter, J. B., Zwaenepoel, W., *Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence*, Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming, March 1990

[14] Bennet, J., Carter, J., Zwaenepoel, W., *Adaptive Software Cache Management for Distributed Shared Memory Architectures*, Proceedings of the 17th Annual International Symposium on Computer Architecture, June 1990

[15] *CELIP: A cellular Language for Image Processing*, W. Hasselbring, 1990

[16] Cellware, 1991

[17] MG Norman, JR Henderson, G. Main, DJ Wallace, *The Use of the CAPE Environment in the Simulation of Rock Fracturing*

[18] PhD thesis of Stéphane Vialle, *ParCeL-1: A Parallel Language synchronous autonomous actors*

[19] Himanshu Shekhar Sinha, *Mermera: non-coherent distributed shared memory for parallel computing*, Boston University, Boston, MA, 1993

[20] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam , V., *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994

[21] Wen-Yen Liang, Chun-Ta King, Feipei Lai *ADSMITH: An efficient object-based distributed shared memory system on PVM*, Proceedings of the 1996 International Symphosium on Parallel Architecture (ISPAN 96), June 1996

[22] *Cellular Automaton Tool User Manual*, Georg Junger, 1994, GMD, Sankt Augustin

[23] Kutrib, Martin, *Parallele Automaten*, Bericht Nr. 9401, AG Informatik, Univ. Giessen, 1994

[24] Christian Hichberger and Rolf Hoffmann, *CDL - a language for cellular processing*, Proceedings of the Second International Conference on Massively Parallel Computing Systems, IEEE, 1996

[25] Protic, J., Tomasevic, M., Milutinovic, V.,*Tutorial on Distributed Shared Memory (Lecture Transparencies)*, IEEE CS Press, Los Alamitos, California, USA, 1997

[26] Demeure, I., Cabrera-Dantart, R., Meunier, P., *Phosphorus: A Distributed Shared Memory System on Top of PVM*, In Proceedings of EUROMICRO'95, September 1995

[27] Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W., *Treadmarks: Shared memory computing on networks of workstations*, IEEE Computer, February 1996

[28] Keleher, P. *CVM: The Coherent Virtual Machine*, University of Maryland, Departament of Computer Science, 1996

[29] Khandeher, D., *Quarks: Distributed shared memory as a basic building block for complex parallel and distributed systems*, Technical Report Master Thesis, University of Utah, March 1996

[30] Thomas Worsch - *Programming environments for Cellular Automata*, Karksruhe University, Informatics Departament, November 1996

[31] Wuensche,A., *Discrete Dynamics Lab (DDLab)*, 1996

[32] Domenico Talia - *Cellular automata + Parallel Computing = Computational Simulation*, 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, Berlin, August 1997

[33] *Continuous-Valued Cellular Automata for Non-Linear Wave Equations*, by Daniel Ostrov and Rudy Rucker, published Fall 1997

[34] Dreier Bernard, Zahn Markus, Ungerer Theo *Parallel and Distributed Programming with Pthreads and Rthreads*, Third International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'98)

[35] W. Hu, W. Shi and Z. Tang. *JIAJIA: An SVM System Based on a New Cache Coherence Protocol* In Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99), April 1999.

[36] Inagaki Tatsushi, Junperi Niwa, Matsumoto Takashi, Hiraki Kei *Supporting Software Distributed Shared Memory with an Optimizing Compiler*, Department of Information Science, Faculty of Science, Univeristy of Tokyo, ICPP 1998

[37] I. Foster, C. Kesselman,The Grid: Blueprint for a New Computing Infrastructure Morgan Kaufmann Publishers, San Francisco, California, 1999.

[38] Freiwald, U. and J. Weimar, *CASim: Java Environment for Simulating Cellular Automata*, 1999

[39] Ian Foster *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, IJSA, 2001

[40] Ian Foster *What is the Grid? A Three Point Checklist*, July 20, 2002

[41] *High Performance Cluster Computing: Architectures and Systems*, Rajkumar Buyya, Prentice Hall, USA, 1999

[42] *An object oriented approach to lattice gas modeling*, Alexandre Dupuis, Bastien Chopard, University of Geneva, Switzerland, August 1999

[43] 2000, october 22 - Yann Boniface. *Etude et développement d'une bibliothèque d'adaptation du parallélisme neuromimétique au parallélisme MIMD - Design and implementation of a library to adapt neuromimetic parallelism to MIMD one* University Henri Poincaré - Nancy I. Collaboration with Loria-Cortex team (PhD Thesis also supervised by Frederic Alexandre), and Charles Hermite Center.

[44] Stéphane Vialle - *Parallélisation de réseaux de neurones VS Parallélisation de systèmes multi-agents*

[45] B. Cheung, C. L. Wang and K. Hwang. *JUMP-DP: A Software DSM System with Low-Latency Communication Support* In the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), Las Vegas, Nevada, USA.

[46] Spezzano G., Talia D., *CAMELot: A Parallel Cellular Environment for Modelling Complexity*, June 2001

[47] Stéphane Vialle - *Synthèse des recherches et perspectives en: Parallélisation de Systèmes de Calculus Distribués dÓrientation Cellulaire, sur Architectures MIMD*, 28 Nov. 2002, Supélec, France

[48] Stéphane Vialle, Eugen Dedu, Claude Timsit, *ParCeL-5/ParSSAP: A Parallel Programming Model and Library for Easy Development and Fast Execution of Simulations of Situated Multi-Agent Systems*, 2002

[49] 2002, March 8 - Eugen Dedu, *Parallelisation of situated multi-agent systems*, University de Versailles St-Quentin. (PhD Thesis supervised by Claude Timsit and Stéphane Vialle). Collaboration with Charles Hermite Center

[50] *Cellular automata models for synchronized traffic flow*, Rui Jiang, Qing-Song Wu, University of Science and Technology of China, November 2002

[51] *Cellular Automata Models and MHD Approach in the Context of Solar Flares*, Anastasios Anastasiadis, Institute for Space Applications and Remote Sensing, National Observatory of Athens, 2002

[52] Eckart, J.D., *Cellular/Cellang environment*, March, 2002

[53] A description of ParCeL6, available from *http://www.metz.supelec.fr/ ersidp/Projects/Parmodel/Parcel-6/Root.html*

[54] Chris Gordon-Smith 2003, *Flexica - User Manual*

[55] *Cellular Automata*, Klaus Sutner, Carnegie Mellon University, Fall 2003

[56] Radu Kopetz *Extension of a parallel library for cellular computing on a cluster of PCs and on computer grids, 2004*, Supélec and University "Politechnica" of Bucarest (UPB), supervised by Prof. Stéphane Vialle and Prof. Nicolae Ţăpuş

[57] *Building High-Performance Linux Clusters, Sponsored by Appro*, Logan G. Harbaugh, Networking Consultant for Mediatronics

[58] Naumov L., *CAMEL - Cellular Automata Modeling Environment and Library*, Saint-Petersburg, 2004

[59] Carotenuto Dario, *Definizione ed implementazione di CANL: Cellular Automata Network Language*, 15.7.2004

[60] Olivier Ménard, Stéphane Vialle, Hervé Frezza-Buet *Making Cortically-Inspired Sensorimotor Control Realistic for Robotics: Design of an Extended Parallel Cellular Programming Model*

[61] Alexander G. Dean, *Compiling for Fine-Grain Concurrency: Planning and Performing Software Thread Integration*, Center for Embedded Systems Research, Department of Electrical and Computer Engineering, NC State University, Raleigh

[62] *Kerrighed V1.0 Reference Manual*, IRISA/INRIA, 28.02.2005

[63] Pascal Gallard, *Kerrighed 1.0.1 - Installation notes*, 23.03.2005