Extension of a parallel library for cellular computing on a cluster of PCs and on computer grids
Radu KÖPETZ

Supervisors:

Stéphane VIALLE
Ph.D. Prof. Eng., SUPÉLEC, France

Nicolae Tapus
Ph.D. Prof. Eng., UPB, Romania

Bucharest
June, 2004
Acknowledgements

My staying at Supélec and consequently 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 offering me permanent support in my work and also for helping me to deal with personal problems related to my adjustment at Supélec. He coordinated attentively every step of my project, always helping me to choose the right direction. I benefited substantially from our meetings and discussions and he managed to turn my attention towards some fields of interest in computer science I barely knew before.

**Nicolae Tapus**, professor and senior researcher, head of the Computer Science Department at Politehnica University of Bucharest, for being my teacher over the last years and for his advice regarding this project. He is one of the best teachers I have had during my five years at Politehnica. To know him for the past two years has been a privilege to me.

My visit to France would have never been possible without the help of **Constantin Iliescu**, professor at Politehnica University of Bucharest. His special care to all the details of my applications helped substantially towards my admission at Supélec.

I want to express my special thanks to **Laurent Casse**, a DEA student at Supélec for all his help. Having the possibility to talk and work with him helped me to better organize and develop my project.
Abstract

The main purpose of this paper is to describe ParCeL6-2, the version for the distributed memory systems of the ParCeL6 language. ParCeL6 is a parallel programming model, cellular oriented, the last of the ParCeL’s family languages. We have also tried to describe the context for this project and the reasons that have led to its development.

ParCeL6-1 is the first version, conceived for the shared memory architectures whereas ParCeL6-2 comes to complete the implementation of the ParCeL6 model with a distributed version.

We have implemented this second version using the SSCRAP library for communication issues. We have found that this library offers all of the communication mechanisms that are very well suited to what we need offering very good performances at the same time.

ParCeL6-2 is now running on shared memory under SSCRAP environment. We are trying to obtain speed performances as close as possible to the shared memory ones before moving further to clusters and grids. The benchmarks we did so far show encouraging results, demonstrating that SSCRAP is a very good option in terms of execution speed and developing times for the kind of communications required by ParCeL6-2.
## Contents

INTRODUCTION .................................................................................................................................................. 5

PARALLEL AND DISTRIBUTED CELLULAR LANGUAGES ................................................................................. 11
  1.1 ParaCell project .................................................................................................................................. 12
  1.2 Synthesis of current cellular languages .......................................................................................... 13

INTRODUCTION TO ParaCell6 .......................................................................................................................... 17
  2.1 Programming model of ParaCell6 ...................................................................................................... 17
  2.2 Software architecture of ParaCell6 .................................................................................................... 20

ParaCell6 on SHM systems ............................................................................................................................... 22
  3.1 Specialization of the programming model ........................................................................................ 22
  3.2 Debugging phase of ParaCell6.1 ....................................................................................................... 29
  3.3 Benchmarks ........................................................................................................................................ 30

SSCrap library ................................................................................................................................................. 34
  4.1 Features of SSCrap ............................................................................................................................ 34
    4.1.1 Synchronization .......................................................................................................................... 34
    4.1.2 Communications ........................................................................................................................ 35

ParaCell6 on Distributed systems .................................................................................................................... 37
  5.1 Programming model of ParaCell6-2 ..................................................................................................... 37
    5.1.1 Input / Output mechanisms .......................................................................................................... 37
    5.1.2 Cell types .................................................................................................................................... 39
    5.1.2 Dynamic evolution of the cell network ....................................................................................... 41
  5.2 Implementation of the programming model ......................................................................................... 41
    5.2.1 Integration with SSCrap library .................................................................................................. 42
    5.2.2 Data types added – “P6chunk” and “P6apointer” ........................................................................ 43
    5.2.3 Missions routing ........................................................................................................................ 45
    5.2.4 Requests ..................................................................................................................................... 45
    5.2.5 Permutation tables ...................................................................................................................... 46
    5.2.6 Input / output data exchange .................................................................................................... 46
    5.2.7 Output values exchange .............................................................................................................. 49

Tests and Benchmarks for Distributed Model .................................................................................................. 54
  6.1 Example of a program using ParaCell6-2 .......................................................................................... 54
  6.2 Benchmarks results ............................................................................................................................ 58

Conclusion .......................................................................................................................................................... 62

Perspectives for Future Development ............................................................................................................ 67

Bibliography ..................................................................................................................................................... 68
Introduction

Parallel computing roots can be traced more than 20 years old. In the years since then, parallel computing has permitted complex problems to be solved and high-performance applications to be implemented both in traditional areas, such as science and engineering, and in new application areas, such as artificial intelligence and finance. Parallel computing has been a challenge for both the hardware and software designers. There are many reasons to use parallel computation: the world surrounding us is inherently parallel, more computational power can be obtained from parallel computers, there are physical limits of sequential computing (like light speed). Although the parallel computation has also a series of disadvantages (costly hardware, harder to develop parallel software packages), it has succeeded to impose itself over the last years.

Steadily increase interest in the parallel computation has lead to the developing of some new computational models, like “cellular automata”. It appeared near the 1950s, when John Von Newman was trying to develop an abstract model of self-reproduction in biology. After that, is has known a continuous rise until the 1970s, when research on systems equivalent to cellular automata had largely petered out [7]. Fortunately, in the middle of 1980s a rapid increase of interest in the field took place, and over the years since then a steadily increasing number of papers have been published on cellular automata.

Cellular automata (CA) are used for designing high performance applications in many areas, like biology, chemistry, physics and artificial intelligence. It offers a powerful modeling approach for complex systems in which global behavior arises from the collective effect of many locally interacting, simple components. Several tools based on CA are providing meaningful results for real-world applications. One of these tools, ParCeL6, is the objective of this paper.

ParCeL6 is a parallel cellular language designed for complex neural networks (cortical systems) and some physical system simulation (based on local equations). It is the result of many years of research and is the last of a family of cellular languages that began to be implemented at the beginning of the 1990s.

ParCeL6 has presently two versions: ParCeL6-1 for shared memory architectures and ParCeL6-2 for the distributed ones. The subject of my research at Supélec was developing the distributed version of this language, with the help of a communication library conceived and implemented at LORIA by Jens Gustedt, Mohammed Essaidi and Isabelle Guérin Lassous. The main reason for developing this new version of ParCeL6 is the increasingly need of computational power from the applications of this language. For the moment, the aim of the ParCeL6-2 language is to run on clusters of PCs, but the long term is to become a grid language in order to benefit as much as possible of this new kind of architecture that knows
such a raising interest nowadays.

We present now a short introduction of the grids. What are they, what their aims are and what languages we can find for grid programming.

**What is a grid and grid computing**

The last decade has seen a substantial increase in commodity computer and network performance, mainly as a result of faster hardware and more sophisticated software. Nevertheless, there are still problems, in the fields of science, engineering, and business, which cannot be effectively dealt with using the current generation of supercomputers. In fact, due to their size and complexity, these problems are often very numerically and/or data intensive and consequently require a variety of heterogeneous resources that are not available on a single machine. A number of teams have conducted experimental studies on the cooperative use of geographically distributed resources unified to act as a single powerful computer. This new approach is known by several names, such as meta-computing, scalable computing, global computing, Internet computing and more recently peer-to-peer and Grid computing [3].

According to Rajkumar Buyya, a grid can be defined as “a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed ‘autonomous’ resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements”. Another definition is given by Ian Foster in [9], by making a simple three point checklist that can be used to verify if a system is a grid or not.

It is very common to compare the grid to the electric power mesh or to the telephony one. This can provide a good image on what is a grid, and how it interconnects the heterogeneous components, allowing them to communicate.

The benefits of using grids can be very numerous, and applicable in many areas, like engineering, economics, market research, biology etc: better utilization of resources, increased user productivity, allowing widely dispersed departments and businesses to create virtual organizations to share data and resources, faster time-to-solution of complex scientific, engineering, and enterprise computational tasks, management and administration of disparate, non-integrated systems as a single unified system, instantaneous access to compute and data resources.

**Programming grids**

Although grid programming can be performed with traditional programming tools and languages, they are not design to work with heterogeneous hierarchies of machines and devices, data and networks with heterogeneous performance. That is why the need for specialized tools and languages for grids increases, and the goals are both efficient but easy to
develop and manage tools.

The current tools and programming languages fall in two main categories: *shared data abstraction* and *message passing*.

The *shared data abstraction* has three main sub-categories [8]:

1. **Global shared address space.** The languages from this category provide a view of memory as if it is shared, although the implementation can be distributed. The main goal of this approach is to minimize the number of messages passed to the number of messages that would pass if the program would have sent them explicitly (TreadMarks, Rthreads, IVY, Munin, and Linda) [8].

2. **Global Data Space.** The users of such a programming language have to define the variables that are mapped in this memory. Data that must be processed in parallel is defined using special keywords, and is the compiler’s job to partition the data and map them onto the different processors of the parallel computer so instructions that operate on these data will be executed in parallel on different processors that execute the same operation on different elements (HPF-High Performance Fortran, OpenMP) [8].

3. **Distributed Data Space.** Data shared among processes is distributed among processors memories, without the existence of a global scope (Orca, Emerald) [8].

Although the shared memory approach seems a very good one, building an efficient DSM for grids has only been moderately successful.

The *message passing* model uses message passing for inter-process communication. The most common standard for message passing is MPI (Message Passing Interface). MPI is a message-passing library specification, meant to be used on parallel computers, clusters, and heterogeneous networks. MPI includes: point-to-point communication, collective operations, process groups, communication domains, process topologies, environment management and inquiry, profiling interface.

There are a number of available *grid-enabled* implementations of MPI that run on a variety of platforms: MPICH-G2, PACX-MPI, Stampi, MPI_Connect, MagPie, LAM etc.

The most important to represent message passing approach perhaps are: MPICH-G2 and PACX-MPI. Both languages are implementation of the MPI standard optimized for Grid-environments, support TCP based protocol for inter-machine messaging and vendor-supplied MPI for intra-machine messaging. MPICH-G2 uses services from the Globus Toolkit (e.g., job startup, security) to allow the coupling of multiple machines, potentially of different architectures, to run MPI applications.

Although the message passing does not address all the challenges posed by Grid computing, it allows the coupling of several platforms, potentially of different architectures, in order to run MPI applications. The Grid programmer is still responsible for some burdensome tasks such as decomposing the computation, load balancing, layout of data and latency management. This makes the message passing a too low level option, compared to assembly language for conventional computer.
On a higher level of abstraction of grid programming we can find the middleware. The main role of middleware is (from the Grid programmer’s point of view) to offer some additional useful abstractions to alleviate the heaviest programming tasks, eliminating the need for low-level tools (like the ones presented above). Grid-based middleware includes tools belonging to the following subclasses: Network-enabled servers, Frameworks and Component-based technology (CBT).

**Network-enabled servers**

Tools like NetSolve [10] and Ninf [11] belong to the Network-enabled Server subclass, also called GridRPC, because those systems are both Grid-enabled and RPC-based systems that provide access to remote libraries and task-parallel programming model on the Grid. NetSolve and Ninf are very similar to each other in their design and purpose. Essentially, they are remote computing systems which are oriented to provide numerical computations. Adapters have been implemented to enable each system to use numerical routines installed on the other [11].

Ninf and Netsolve are asymmetrical systems in that the client and server side software packages being different. Client function call obtains all the interface information regarding the called library function at runtime from the server. Although this will cost an extra network round trip time, the burden of code/library maintenance on client side is considerably reduced. By contrast, CORBA and other typical RPC systems are symmetrical, in that the same software packages are used for both client and the server.

Other GridRPC systems are DIET, Neos and RCS.

**Frameworks**

According to [8] a computational framework is an integrated collection of software tools that facilitates the development and execution of an application. A framework is the core feature of some Problem Solving Environments (PSEs). There are several tools which currently come under the framework definition. Two good examples are Cactus and Meta-Chaos.

Cactus [8, 16] is an open source framework designed for scientists and engineers. Its modular structure easily enables parallel computation across different architectures and collaborative code development between different groups. Cactus originated in the academic research community, where it was developed and used over many years by a large international collaboration of physicists and computational scientists. The name Cactus comes from the design of a central core (or flesh) which connects to application modules (or thorns) through an extensible interface. Thorns can implement custom developed scientific or engineering applications, such as computational fluid-dynamics. Other thorns from a standard computational toolkit provide a range of computational capabilities, such as parallel I/O, data distribution, or check pointing. Cactus allows the computational application scientist, such as a physicist or engineer, to assimilate the most appropriate available components into a solution tailored specifically to the problem at hand, and to execute a simulation using the best computer architecture available. Developers build Cactus applications dynamically, using a meta-code with a new object-oriented language. This describes how the different
pieces of code, written in any common computational language such as C, C++, Fortran 77, and Fortran 90, interweave. Cactus runs on many architectures. Applications, developed on standard workstations or laptops, can be seamlessly run on clusters or supercomputers. Cactus provides easy access to many cutting edge software technologies being developed within the academic research community. These include the Globus Metacomputing Toolkit, HDF5 parallel file I/O, the PETSc scientific library, adaptive mesh refinement, web interfaces, and advanced visualization tools [16].

In a way Cactus architecture is similar with component base architecture, because anybody can build and reuse thorns for solving a specific problem, most like components model.

Meta-Chaos [8, 16] is a framework that allows interoperability between various data parallel runtime libraries. The role of Meta-Chaos is to allow efficient and transparent interoperability between data structures distributed by a user (or compiler) using multiple data parallel libraries and/or data parallel languages. The multiple data parallel libraries can be used within a single program, or can be used in multiple programs. In either case, Meta-Chaos is able to transfer data between the various data parallel libraries directly and efficiently. Meta-Chaos hides the distribution of data across the processors by one data parallel library from other such libraries. The applicability of Meta-Chaos is not limited to communicating between two data parallel libraries in the same program. Meta-Chaos can also be used to communicate between data parallel libraries in two different programs [16]. Therefore a tool like Meta-Chaos is needed to allow existing parallel applications to interoperate and also to allow new applications, written using whatever data parallel library or language the application programmer feels is appropriate, to interoperate.

Component-based technology (CBT)

There is a growing consensus that CBT could represent a promising new way for higher-level Grid programming. A composite application is defined as an application that is developed from the integration of smaller components (applications or parts of applications) under the control of a work-flow description. A component is an independent unit of software deployment whose functionality is described by behavior rules and interfaces. A component architecture specifies interfaces, interaction rules, and other tools such as component repositories and component composition tools to build component-based applications. Systems like Common Object Request Broker Architecture (CORBA Component Model [11]), Distributed Component Object Model (DCOM, Microsoft, [12]), Enterprise JavaBeans (EJB, Sun, [13]), .NET (Microsoft, [14]), etc were developed using component technology.

CORBA Component Model was included in CORBA 3 specification, and extends the CORBA object model by defining features and services that enable application developers to implement, manage, configure, and deploy components that integrate commonly used CORBA services, such as transaction, security, persistent state, and event notification services, in a standard environment.

CCM is based on EJB and COM, but has some advantages compared to EJB for example, and a significant one is the fact that CCM is platform independent: not limited to
Java or Windows. EJB development is limited to Java, as it is a SUN technology. CCM has EJB support and inter-working, supports the “ejb-jar” deployment information, and bridges uses RMI-IIOP and translate messages at run-time.

EJB servers reduce the complexity of developing middleware by providing automatic support for middleware services such as transactions, security, database connectivity, and more. By hiding such complexities, the EJB architecture enables component developers to focus on business logic.

DCOM extends the Component Object Model (COM) to support communication among objects on different computers on a local area network (LAN), a WAN, or even the Internet. With DCOM, an application can be distributed at the most suitable locations. DCOM handles low-level details of network protocols so that programmers can focus on their real business. One of the biggest disadvantages compared to CORBA and EJB is that DCOM is platform independent, so components must be deployed on Windows machines.
Chapter 1

Parallel and distributed cellular languages

Although massively parallel computing can be very useful to engineers and scientists, the lack of models and tools that simply and automatically permit the exploitation of parallelism in complex engineering and scientific problems prevents a widespread use of this technology by industry. Issues as the decomposition of physical problems in parallel parts, efficient communication between processors, computation load balancing across multiple processors, and fast movement of data in and out of a computer are major factors that created the perception that parallel computers can be used only by experts [3]. Consequently to the development of computer science in the latest years, its application boundaries have been enlarged considerably because of the continuous rise of computing power. At the same time research in parallel computing evidenced the significant potentialities of parallel computing models, such as cellular automata and neural networks, in representing a valid alternative to differential calculus in the description of complex phenomena. This occurs especially when differential equations cannot be solved because of their complexity or when the problem being solved cannot be described in terms of differential equations.

Cellular automata (CA) represent both a powerful tool for modeling complex dynamic problems and an abstract parallel computing model [3]. It is used to model parallel computation for a large number of applications in biology, physics, chemistry and artificial intelligence.

A cellular automaton consists of one-dimensional or multi-dimensional lattice of cells, each of which is connected to a finite neighborhood of cells which are nearby in the lattice. Each cell in the regular spatial lattice can take any of a finite number of discrete state values. Time is discrete, as well, and at each time step all the cells in the lattice are updated by means of a local rule called transition function, which determines the cell’s next state based upon the states of its neighbors. That is, the state of a cell at a given time depends only on its own state in the previous time step and the states of its nearby neighbors at the previous time step. Different neighborhoods can be defined for the cells. The most common neighborhoods in the two-dimensional case are the von Neumann neighborhood consisting of the North, South, East, West neighbors and the Moore neighborhood composed of 8 neighbor cells. In the three dimensional case up to 26 neighbors can be taken in consideration. All cells of the automaton are updated synchronously. The global behavior of the system is determined by the evolution of the states of all cells as a result of multiple interactions [4].
A standard cellular automaton is usually specified by a lattice of cells, a finite set of states for the cells, a finite neighborhood and a local transition function.

Due to the fact that CA are intrinsically parallel, they can be simulated onto parallel computers with high efficiency, as the communication flow between processors can be kept low. Experiments showed that whenever a sequential computer is used to support the simulation, the execution time might become very high since such computer has to perform the transition function for each cell of the automaton in a sequential way [4]. This comes to strengthen the fact that parallel computers represent the most natural architecture where CA environments might be implemented.

In the following sections of this chapter, the ParCeL languages family is described, along with a synthesis of current cellular languages.

1.1 ParCeL project

The different languages of the ParCeL family (PARallel CEllular Languages) are mainly destined to make very easy the implementation of distributed, cellular oriented algorithms on parallel machines, letting the user to declare and to program the cells, a kind of very light processors [2]. The first language (ParCeL-0/MCV) was born around 1990 during the UHP thesis of Thierry Cornu made between Supélec and CRIN. Then, a new version (ParCeL-1) greatly enriched and optimized appeared around 1993-94 due to the thesis of Stéphane Vialle of the University of Orsay-Paris-XI, done at Supélec. The applications taken into consideration for these languages were the connectionist ones, the semantic networks and the hybrid symbolic-connectionism systems. The last ones are developed with the collaboration of Loria, during the UHP thesis of Yannick Lallement. The performances are encouraging, but the hardware architectures evolved, and the computational model of ParCeL-1 couldn’t take full advantage of this, without deep modifications. This first period matches the upper part of the Figure 1.1.

Later other parallel languages of that family appeared, however it seemed that is very hard to conceive an efficient parallel languages for such a great variety of applications that exists. More, at that time, the authors of these languages wanted to develop also some scientific computational aspects. So, the evolutions of ParCeL-1 followed three main directions [2]:

- one evolution towards the scientific computation had given birth to ParCeL-2
- one evolution towards the Artificial Intelligence and multi-agent systems using the features of the new architectures took place at Supélec with the name of ParCeL-3. This version was evaluated during the development of a multi-agent system.
- one precise evolution towards connectionists systems of biological inspiration gave birth to ParCeL-4/Hibs.
another precise evolution towards multi-agents systems was finished in 2002, with the name of ParCeL-5/ParSSAP.

These evolutions, towards connectionists systems, multi-agent systems and scientific computation correspond to the three lower cadres that can be seen in the Figure 1.1.

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: ParCeL6-1 which is designed for shared memory systems and ParCeL6-2, designed for distributed architectures.

1.2 Synthesis of current cellular languages

Cellular languages, the languages where the programming model follows the
computational model of cellular automata, existed before 1995. Although, most of the cellular languages implemented on MIMD architectures have appeared after 1995 [2].

For the CA model, there are two possible alternatives which allow achieving high performance in the implementation of CA. The first one is the design of special hardware devoted to the execution of CA. The second alternative is based on the use of commercially available parallel computers where the state of cells can be updated simultaneously.

CAM (Cellular Automata Machine) is the most significant example of a specialized hardware which has been designed to run CA simulations [5]. 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. On the other hand, highly parallel computers offer the most natural architecture for a CA machine. These systems are based on a number of interconnected processing elements (PE) which perform a task concurrently. Both SIMD and MIMD architectures are suitable to support CA implementation achieving high parallel efficiency, but MIMD computers are preferred because they are more flexible and allow also efficiently implementation of heterogeneous systems with a lower cost when few machines are built.

In the latest years several parallel cellular environments have been developed. Significant examples of these parallel cellular environments are CAMEL, NEMO, CAPE, Cellang, Ceprol, CDL, CAM, DDLab and CAML.

According to [6], three main aspects can be distinguished according to the different software packages:
1. The systems they are intended for
2. What do the interfaces to the user and to the machine look like – textual or graphical
3. What features concerning CA are present or missing

As far as (1) is concerned is seems that most systems are intended for doing step by step forward simulations of specified CA starting with specified initial configurations, i.e. for finding 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”.

We will present further some differences between several systems, organized by the characteristics of a general CA.

Lattice

The structure of a lattice is in most of the programming environments a two-
dimensional one. This means that 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 [6].

**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 [6].

### 3.3 Set of States

Several systems restrict the size of the set of states to 256. DDLab only allows one bit per cell. This has to do with the special kind of explorations it is intended for. All other systems (including CANL, CARPET, CDL, Cellang and Hical) allow an arbitrary number of states.

If the state 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 [6]. Cellang and CDL and also offer the possibility to speak about arrays of values.

### 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 (more or less) the 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 (e.g. CANL, CARPET, CDL, Cellang) [6].
Chapter 2

Introduction to ParCeL6

As introduced before, ParCeL6 is the most programming tool of the ParCeL family. It is a parallel cellular language designed for complex neural networks (cortical systems) and some physical system simulation (based on local equations). These applications consist in many small computation units (the cells), statically or dynamically created and connected, exchanging data, and are run on parallel machines. [1]

Like all other ParCeL programming models and languages (or libraries) it consist in a compromise between a high-level semantics to make easier cellular program development and a programming model that can be efficiently implemented on general purpose parallel architectures.

Currently, two versions of ParCeL6 are available: ParCeL6-1 which is the shared memory version and the distributed version, ParCeL6-2. This chapter will focus on programming model and software architecture of ParCeL6, common for both versions, with some specializations for the distributed version that are presented in detail in Chapter 6.

2.1 Programming model of ParCeL6

ParCeL6 objectives can be seen from two points of view: the user’s point of view and the theoretical point of view. From the user’s point of view, ParCeL6 has to be easy enough to use, in order to minimize as much as possible the times needed to implement a cellular model of a specific problem, and also to be fast enough to minimize the execution time for the simulations of the problem. As for the theoretical point of view, ParCeL6 has to find the best approach to adapt the fine grain problems (cortical applications, physical simulations) on the coarse grain architectures (Figure 2.1). The programming model of ParCeL6 is conceived in the way to suit the best this objectives.
Main characteristics of the programming model are:

1. **Three types of cells**

   Being a cellular language, ParCeL6 has the “cell” as fundamental computing element. A cell has some characteristics: unique identification number, parameters and Input/Output operations with the sequential program (getting and sending data), local variables for the cell (static or dynamic), three functions that control the life cycle of the cell (initialization, iteration and termination function), channels of communication with other cells, output of numerical type with variable size but fixed on definition of the cell.

   ParCeL6 has three types of cells based on their output type, and these can be easily exchanged during experimentations: direct, buffered or hybrid. The difference between these types of cells is the moment when their new output value is available. We will call further a “writer cell” a cell that just produces some output values, and a “reader cell” one that reads the output of another. Of course the writer can be itself a reader of another cell.

   For the direct output mode, the output values of a writer cell are accessible to any reader cell immediately. Therefore any cell that needs to read an output of a direct one can get the most recent value, by accessing directly the output of the writer cell. In opposite, the reader of a buffered cell will always get the old output value (produced in the previous cycle) from the writer. The hybrid cells are a combination of the direct and
the buffered ones, meaning that when accessing the output of the writer, the reader can get either the new or the old values. For the ShM model, the state of the values read (new or old) is given by the order of the readers: the first cell from one processor that will access the writer’s output will get the new value, and all other further accesses from the cells hosted by the same processor as the first will result in getting the old value. On distributed memory, “direct” model would be not efficient; therefore a new model that is close to the direct one should be imagined.

The experiments showed some differences when using the three output kinds of the cells (Figure 2.2)

![Diagram showing output types of the cells](image)

**Figure 2.2: Output types of the cells**

This mechanism allows to quickly experiment different kind of output propagation methods, and to study their impact on the result values and on the algorithm convergence speed. Moreover, this is useful to begin to design a cellular programming model for distributed architectures: all output propagation methods can not be efficiently implemented on a distributed architecture, due to too many messages passed [1].

2. **All cells are executed in parallel**

The computation model of ParCeL6 states that all the cells are executed in parallel, although that this is not applicable in practice. The user cannot presume any order of their activation and the applications of ParCeL6 should never depend on the order of cell execution.
3. The topology of the cellular network is dynamic

At any moment during the execution of the program, cells can be created or killed, therefore the topology of the cellular network changes. Cells can be created statically (before running the cellular network) or dynamically (by other cells during their activations). Cells are dynamically connected, and can be disconnected and re-connected when needed. Finally, unused cells can be killed at any time. A ParCeL6 program can be entirely static: all cells are created before running the cellular network, and connections are establish at initialization step of cells and remain unchanged until the end of the program. Or, a ParCeL6 program can be dynamic, with very few cells created before to run the cellular network, and many cells created and connected at run time during iteration steps of already existing cells or between any two cycles by the main program (running on processor 0).

4. Functioning is based on cycles (BSP model influence)

Like Bulk Synchronous Parallel (BSP) model, ParCeL6 operates on cycles. During a cycle, processors compute and they exchange values (not necessary at the end of the cycle). Finally, a barrier synchronization phase is performed to wait for all computing and communication actions to complete.

5. Memory sharing between cells is possible

Cells can connect / disconnect from other cells dynamically during the execution of the program. After a cell connects to another, it can read the output values from that cell, in this way making possible data transmission between cells.

6. Neighborhoods’ shape is irregular

The user has the option to connect to a cell to any other cell disregarding their positions on the cell lattice. All connections are established by the readers, and are unidirectional.

2.2 Software architecture of ParCeL6

The software architecture of the ParCeL6 is a Single Program-Multiple Data (SPMD)
one, with the global data to be processed partitioned and a portion of the data assigned to each processing node. When the cells are created, a load balancing technique is used, ensuring that the processors will get approximately the same amount of data.

ParCeL6 has “n” cells / process, with “n” >> 1. It is essential that the number of the cells is much greater than the number of processes, because of the fine-grained applications ParCeL6 is designed for. Also ParCeL6 is designed to have one process / physical processor, and the first benchmarks of the ParCeL6 clearly show that the best performance is obtained in this way.

One of the main concepts of ParCeL6 is the concept of missions. A mission is something that says what code to be executed at a certain moment. After the program starts, every process waits to receive the mission from the main process. The main processor sends the mission to everybody (including itself). After the mission is received, it is interpreted, and the proper code is executed according with the mission code received. After every mission, all processes are synchronized with a barrier, so that the possibility that two processors execute different missions at the same time is excluded (*Figure 2.3*).

*Figure 2.3: Missions*

When cells are created, they are assigned to a certain processor, which will be the host processor for those cells. The host of a cell cannot be further changed, so that a cell once assigned to a certain processor, remains hosted by that processor for its entire life cycle (until it is killed).

On a processor, the activation order for the cells can be impacted by the user by using “permutation tables”. In this way, the user can choose the activation order to be from the first to the last hast-table entry, from the last hast-table entry to the first, or by giving the precise permutation to be used.
Chapter 3

ParCeL6 on ShM systems

The shared memory version of ParCeL6 for single / multiprocessor machines is ParCeL6-1. This version is available for both Linux (using P-Threads) and Windows (using native Windows threads), and we think that it is a stable version, as it ran through multiple tests and the bugs that were revealed were corrected.

3.1 Specialization of the programming model

Before explaining the internal structures of ParCeL6.1, some definitions should be stated first. ParCeL6.1 has the concepts of processors, requests, missions, OutConns, permutation tables. We describe them in the next paragraph.

Processors

ParCeL6 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 that will be later detailed in this chapter.

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. ParCeL6.1 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 destined 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 appropriated 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 destined to them (ex: threads process cell creation requests)
- NETLINKEVOLUTION - threads accomplish link evolution request destined 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 (chose randomly the permutation table).

ParCeL6.1 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 [1].

The parallel implementation is based on threads. All threads are launched at the initialization time by the main thread. Every thread is associated a ParCeL6.1 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 3.1).

![Figure 3.1: Global Space](image)

Global space contains the following:
- table of all processors structures
- 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 ParCeL6 processor:

- **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 request 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 announce 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** – hash-table 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 of OutConns’ flags** – 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 ParCeL6 processor has to contain all the other structures that are needed for computation.

**Cell structure (Figure 3.2):**
- **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 space that can be used by the user to store some cell specific information he may need.
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), ParCeL6.1 uses a special type of tables (Figure 3.3). 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, ParCeL6.1 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 and that 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.
Because they are heavily accessed the cells and the OutConns are stored in hash-tables (Figure 3.4) 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 hash-table 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.
3.2 Debugging phase of ParCeL6.1

The development of the project also included a debugging phase. As a result, ParCeL6 evolved to its version 1.6 to its version 1.10. We consider that now it is a stable and robust version for shared memory environments. In this section, we present the list of the bugs we found and corrected.

Memory management

This problem we found had a great impact on memory allocation. To obtain a dynamic structure, most of the data is stored in the chained tables we presented in the previous section. When a table is full, a new table is allocated and chained with the others. Because allocation and de-allocation of memory space cost a lot in terms of execution time, the tables are never freed after there are not used anymore – they are empty. This space can be reused whenever is needed. The problem came exactly from this re-usage. Actually, every time more space was needed, a new table was allocated instead of using the ones that existed. This led to a big memory leak.

We modified the function that was returning the pointer to a free memory space, p6GetPtRqBody. Every time the current table is full, the function will perform a verification through all the tables in front of this one before allocating new space. Even more, for avoiding a search though all the tables, the new table is placed in front of the list for the next memory space search. In this way, we save the search time that can be quite long when there are a lot of tables allocated.

Hash-tables management

This problem appeared during the tests for cells death, by counting the number of the death-signaling requests was sent by the killed cell to other cells. Actually, this problem revealed two bugs.

For obtaining a better access time to the cells, these are put in a hashtable with an entry key that is computed from their registration number. The hashtable is actually a table of elements that contain two pointers: one to the first cell that is connected to that element, and one to the last one. The cells themselves have the pointers to their neighbors. This way, a chained list of cells corresponds for every entry in the hashtable. When the user calls p6_kill_cell, a kill-request is created. After that, the request is executed by the function p6KillCellRqExec. This function will put the current iteration function of the cell the Term function, which will cause eventually the p6_end_term function of ParCeL6 to be called. At
its execution, the killed cell is removed from the hashtable, and all the space related to it is deleted. The problem appeared at the next cycle, when an iteration was performed though the hashtable to activate every cell: some of the cells were not activated anymore. It seemed that when removing the cell from the hashtable, the links from the chained list of this cell were broken, so we couldn’t access the cells that had the same key with the killed one.

We solved this problem by modifying the `p6_end_term` that was taking care of cell removal, and we took extra precautions not to alter the hashtable anymore.

The second problem concerned the usage of a memory space that was had already been deleted. This caused us to use some randomly memory values, completely different from the ones we were expected. We solved the problem by using a “buffer” variable for holding the values after their space was deleted.

**Output values exchange in hybrid mode**

We discovered this bug the time we were trying to became more familiar with the library implementation.

The “OutConn” structure has a flag that says if it was refreshed or not. This is used only in `direct` and `hybrid` mode, because in `buffered` the “OutConns” are never refreshed. In `hybrid` mode, if the “OutConn” is not refreshed and the cell is the first to call the refresh, then the update of the values should be performed. In the `direct` mode, whenever a cell wants to refresh the values, they are updated. The problem we discovered was the fact that the flag for the state of the “OutConn” was always false. This means that the fresh took place every time, this way making the `hybrid` mode to work exactly as the `direct` one. We corrected it by simply updating the flag correctly.

### 3.3 Benchmarks

ParCeL6.1 has run through several benchmarks on different kind of applications. For the moment, two series of benchmarks have been realized. The first concerns the speed convergence (e.g. for the Jacobi is the time needed for the cell in the middle to have the value close to 5 – 4.9 for ex) for the three output types of cells (“direct”, “buffered” and “hybrid”). The second series of benchmarks illustrate the execution speed (time needed to run a certain number of cycles) for different number of processors (threads) and for different output type of cells.

The first series of benchmarks try to measure the efficiency of the different output types for the convergence speed of the applications tested. The time measurements were realized for a Jacobi relaxation program of 130x130 size and on a P4 2.8MHz with 1MB cache memory. For the benchmark, we have measured the number of cycles needed for the Jacobi program to converge (the difference between the ideal value – 5 – in the center and the value obtained to be less than 0.001) in every of the three modes (Figure 3.5). For every
processor, we have run the Jacobi program four times, and the points on the graphic are the average.

![Table and Graphic](attachment:Figure_3.5.png)

**Figure 3.5: ParCeL6-1 convergence speed**

The convergence speed of the buffered mode, as expected, doesn’t depend on the number of processors.

As we can see from the benchmark results, the hybrid mode behaves almost the same as the buffered for a small number of processors, but when the number of processors increases, it approaches the direct mode behavior. The best convergence speed is when using the direct output mode of the cells (almost two times faster than the buffered).

The second series of benchmarks is divided in two sections:

a) *Measurements of the execution time of ParCeL6-1 applications for different number of processors*

We have made two series of benchmarks, with two different applications and on different computers. The first we made using cortical application (Grumpf-2.28, Nono-2.16, Bijama 1.10) and are shown in *Figure 3.6*. We ran the application for 1000 cycles on three different machines (Quadx1, Quadx2, Quadx3), and for 1 to 8 threads (ParCeL6.1 processors). Every point on the graphic is the average of four values. The characteristics of the machines used are:

- **Quadx1**: 4 x PIII 700MHz, with 1MB cache memory
- **Quadx2**: 4 x PIII 700MHz, with 2MB cache memory
- **Quadx3**: 4 x PIV 2.5GHz, with 1MB cache memory

When using less than 3 processors, seems that the cache difference between the Quadx1 and Quadx2 machines doesn’t influence so much the execution speed, but when going for more than 3, the execution curve changes a lot. Very interesting is that when running on more than 6 threads, we get better results on Quadx2 than on Quadx3, although there is a big difference in the frequency of the physical processors of the two machines. We can see that the best results are obtained with 4 threads for all the computers, and performance decreases when using hyper-threading.
Figure 3.6: ParCeL6-1 execution time for cortical applications

b) Measurements of the execution time of ParCeL6-1 applications for different number of processors using different types of cell outputs

First benchmark in this section (Figure 3.7) measured the cycle time for the three types of cell outputs. It is drawn from the same values as in Figure 3.5 and shows not a very big difference in the cycle time when using different cell outputs.

Figure 3.7: ParCeL6-1 cycle time for different kind of cells for Jacobi relaxation

The benchmark from Figure 3.8 was also done on Quadx2, using cortical application.
We can see that the curve for the *hybrid* is almost identical with the one for the *buffered*, and that they have better execution times than the direct mode.

The results from this benchmarks shows that when using the *hybrid* mode with enough number of processors, we can obtain the same convergence speed as the *direct* mode with the execution time of the *buffered*.

**Figure 3.8: ParCeL6-1 execution time for different kind of cells**
Chapter 4

SSCRAP library

“The Soft Synchronized Computing in Round for Adequate Parallelization (SSCRAP) library is a C++ communication and synchronization library for coarse grained algorithms. SSCRAP is the first library supporting all known coarse grained models that are based on message passing. Indeed, it allows the implementation of BSP, CGM and PRO algorithms by supporting, at the same time, their respective execution models. Providing a high abstraction level, SSCRAP makes the real evolved communications transparent for the user and handles efficiently data exchanges and inter-process synchronizations. Thus it guarantees a good portability and ensures efficiency”[15].

4.1 Features of SSCRAP

We can classify the features of SSCRAP library into four categories [15]: communication routines, synchronizations tools, process management and instrumentation. Because this paper focuses mainly on features that are used by ParCeL6-2, and is not intended to offer a detailed description of SSCRAP, we will focus mainly on describing the communication and synchronization models.

4.1.1 Synchronization

SSCRAP provides two types of synchronization: bulk synchronization and data synchronization. The data synchronization can be considered one of SSCRAP main innovations. It provides both send and receive synchronization. The data synchronization implemented by SSCRAP provides the possibility of handover of data control. This way the computation can be efficiently separated from the communication.

Send synchronization: each processor is blocked until the communication layer has taken over all sending messages. Thereafter local memory allocated for send buffers can be reused: data control is handed over to the communication layer.

Receive synchronization: each processor is blocked until all the messages coming from the communication layer are loaded in their respective reception buffers. Then processors access to the data for their computation: data control is handed over to the processors. Being data based, send and receive synchronizations allow "soft synchronization". Indeed, compared to bulk synchronization as used in the BSP libraries
where every processor has to wait until all communications of all processors is accomplished, they avoid unnecessary idle time.

## 4.1.2 Communications

In coarse grained algorithm execution, we distinguish two types of exchanged data. The first corresponds to the local data directly used in the algorithmic treatment. In this case, the data size is proportional to the input size and relatively important. Its performance is mainly restricted by the bandwidth of the communication layer. The second type of exchanged data corresponds to control messages. These messages inform the processors about the state of the coarse grain machine, the exchanges and the dynamical distribution of the data during the execution. Having generally a fixed and reduced size, their performance is usually dominated by the latency of the communication layer. To simplify the exchange of the control data, SSCRAP implements DRAM mechanisms. Using DRAM users can specify and use shared data structures used to make the control data remotely accessible without message passing [15].

From the communication methods provided by SSCRAP library, we mainly used two of them in order to implement the data communication in ParCeL6-2: `bulk communication` and `one_to_all communication`.

### Bulk Communications

The `bulk communication` is the part of SSCRAP library where most of the efforts were concentrated. The main purpose of the bulk communication is to provide a simple and unique tool to cover all the different types of data exchange.

When using bulk communication, the data is represented as a distributed array over the processors. Each processor has its subpart of the array allocated, which it can access. The bulk communication is about redistribution of this array. For controlling what data to send from the array and to what processor, SSCRAP provides two methods: `extract` and `identify`. The `identify` is used by each processor to determine the destination for the data elements. It is called for each element in the array. `Extract` transform the data item as it is on the sending side to what should be received. The user has the option to hold elements in the array of a specific data type, and to use `extract` to transform each element in a different data type.

*Figure 4.1* illustrates the `bulk` communication mechanism.

### One_to_all communications

Another communication type we used from SSCRAP is the `one_to_all` communication. It is very similar with the method that has the same name in MPI. Basically,
is a broadcast from one processor to all others. It can be used to send any type of data.

Figure 4.1 – Bulk communication in SSCRAP
Chapter 5

ParCeL6 on distributed systems

ParCeL6-2 is the distributed version of ParCeL6, and the description of this second version is the main objective of this report. It was developed starting from ParCeL6-1, and it conserves most of the features and characteristics of this initial library. Main reasons of developing this new version are due to the performances of the distributed architectures comparing to the ShM ones. Although we can find on the market very powerful ShM machines (multi-processor computers especially), we can obtain the same performances or even better with multiple mono-processor machines coupled in a network, at a much more reasonable price. At the same time, on the ShM machines the number of processors is limited by the performance of the communication “medium” it uses for memory access, a component that frequently reduces the power of these systems comparing to the distributed ones (mainly for low cost multiprocessors PCs).

Developing of the distributed version of ParCeL6, made possible that at this moment we dispose of a ParCeL6 that can be deployed on both types of architectures: shared memory and distributed.

This chapter will present the changes in the programming model of ParCeL6.1 in order to run on distributed systems, along with the implementation details of ParCeL6-2.

5.1 Programming model of ParCeL6-2

Most of the programming model of ParCeL6 was kept in ParCeL6-2. Although, some changes were necessary in order to adapt it to a distributed environment. So we had to eliminate some of the elements of the ShM model (ParCeL6-1) and to modify others.

5.1.1 Input / Output mechanisms

Some of the target applications of ParCeL6 need the possibility of data exchange between the application and the ParCeL6 library. A good example of this data is the images from video camera that needs to be accessed by the cells managed by the library. This type of data exchange in ShM environment was very simple to implement in ParCeL6.1. We just have to put the data in the right place indicated by the library, and after that, every thread of ParCeL6.1 can simply just read the data by accessing directly the memory space where it is stored. Now, being on a distributed architecture, a process running on another machine
cannot access the memory where we put the data on the main processor, and so we have to route it to every process.

In ParCeL6.1 we have the possibility to share data that is needed by all cells (global data), data that is particular to every cell (local data) and the results of cell computation (result data). For ParCeL6-2 we needed all of these three data types to be exchanged, so we imagined three mechanisms for every one of them. All three input / output data types are defined by the user in the way he needs to, the library doesn’t introduce any constrains at all regarding data types. Even more, we introduced a mission for every of these data types exchange so that the user has the possibility of refreshing any of them at any moment during the execution of the application.

The applications written in ParCeL6-2 can choose to use one, two, all three or none of this data types.

**Mechanism for global data exchange**

The “global data” is the data that is destined to all the cells of the process. All the cells on the process can read the data when they need to, but can never change it. This data can be for example a camera image that is refreshed from time to time. The data is sent from the sequential program (main process) to all other processes whenever the mission for this data refresh is performed. Every process gets this global data, and after receiving it, the cells can use it in the next steps.

**Mechanism for local data exchange**

The “local data” is a set of data and every data element is destined to just one cell. For example, the local data can contain the cell coordinates in the cell matrix, if the application needs it. It is the user choice to decide for each cell what data to send. It is possible that not all of the cells need this kind of data, and so the user can choose not to send it for some cells. The local data is read-only, that means that the cell cannot change the data after it had received it.

The local data is sent from the sequential program to all other processes whenever the mission for this data refresh is performed. Every process gets this local data for its cells, and after receiving it, the cells can use further in the next steps.

**Mechanism for result data exchange**

This mechanism allows for the data resulted from some computational steps to be gathered from all the processors in the sequential program.

After some computational steps, the user may decide that for continuing with the next steps, he needs the values computed so far from some cells. So using the appropriate refresh
mission for the result data type, the output of all cells from all the processes can be gathered on the main process. Every process, when receiving this mission, will send the current output values of its cells to the main process.

In ParCeL6-1 this was very simply achieved by directly accessing (through pointers) the output space of the cells.

### 5.1.2 Cell types

Although the ParCeL6 computational model contains three types of cell differentiated by their output type (direct, buffered and hybrid), the computational model of ParCeL6-2 restricted the cells only to buffered and hybrid. In ParCeL6.1 it was possible to implement also the direct mode, because the simple access to the memory space offered by the ShM systems is very efficient. But for the distributed systems (ParCeL6-2), using this kind of cell output would imply a great amount of communication between the processes when trying to refresh the input data of the cells on demand. If we would have implemented this kind of cells, the performances of the ParCeL6-2 applications would have suffered an important decrease. For this reason, the direct mode was removed from the computational model of ParCeL6-2.

**Buffered mode**

The buffered mode of the cells in ParCeL6-2 hasn’t been changed very much from the way it works in the ParCeL6-1. Actually, the model of the buffered remained unchanged, the only changed is that the user is able to divide each data refresh in multiple sub-steps. He can choose to refresh the data for all the processors at the same time, or only for a subset of processors, repeating the refreshing until every processor has been refreshed. This feature can be used by the user to tune the application, choosing between the quantity of memory used and the computational time spent to refresh the buffered input.

**Hybrid mode**

Unlike the buffered model, the hybrid exact model has changed in this version of ParCeL6. For the hybrid approach we found two possible solutions. First is inspired by the direct mode, and the second is inspired by the buffered one.

The model inspired from the direct one corresponds to the one described in Figure 5.1, and is very similar with the hybrid approach in ParCeL6-1. In this model, on-demand refresh used in ParCeL6-1 is kept. The idea is to create a second thread which will deal only with the communication problem. In this way, every process will have two threads: one for the cell computation and the other for communication with the other processes. When a cell is computed, it can attempt to refresh the output values of the cells at which it is connected to.
At that moment, the cell will send a refresh message to the process that host the cell it wants to refresh. The communication thread of that processor will receive the message and will send back the requested value. So the cell will get the value it requested, and will continue its execution.

![Image of hybrid model inspired from the direct model](image)

Figure 5.1: Hybrid model inspired from the direct model

The other way the hybrid can be modeled can be seen in Figure 5.2. It was inspired from the buffered model used in ParCeL6-1.

For this model, the on-demand refresh is not present. To replace it, we introduced sub-cycles of refresh in the computational cycle. Every sub-cycle is divided in two steps. The first step is a pure computational one, where output values of a part of the cells are computed without performing the refreshes demanded, like in the buffered model. In the second step, the same number of cells as the cells that were computed is refreshed, but not necessarily the same cells. Even the group of cells refreshed is not the same on every cycle, being randomly chosen. This is important in order to conserve the asynchronism required for the convergence of the applications of the ParCeL6-2. If the cells that are refreshed are in the same order on every sub-cycle and cycle, we risk having the same behavior as the buffered mode.
5.1.2 Dynamic evolution of the cell network

One of the main characteristics of ParCeL6 is the ability to modify the cell network dynamically. At any moment, cells can be created and killed both from the sequential program (main process) and from others cells. This is fully supported by ParCeL6-1, but for the distributed version we modified and reduced a little this feature. In fact, in ParCeL6-2 the cells have no possibility to create a sub-network by creating cells in their activation functions. The cell creation is only reserved to the sequential program. The reason for this change is a quite simple one: if we would have implemented this in ParCeL6-2 the exchange mechanism for the three input / output data types would have became very complicated and very costly in execution time. Anyway, the current applications of ParCeL6 are not affected by this change, because mainly they create cells from the sequential program.

5.2 Implementation of the programming model

The implementation of ParCeL6-2 was done with the help of the SSCRAP library. We
start the implementation from ParCeL6-1. During development, we tried to test every change we made just to be sure it is working properly. Therefore, every modification to ParCeL6-1 was tested at the time it was finished, and not when everything was ready. This helped us to be sure that what was changed so far was working, and also to observe the differences in execution time between the two implementations. For doing this, we used the facility of SSCRAP to run exactly the same program on both shared and distributed architectures. This allowed us to run the different intermediate versions we build, versions that contained both new code written for the distributed version and old shared memory code.

In the next sections, we will present the implementation steps of ParCeL6-2 exactly in the order they were performed.

5.2.1 Integration with SSCRAP library

The first step in ParCeL6-2 implementation was to successfully integrate ParCeL6 with the SSCRAP library, and run some applications in this environment.

For this to be possible, we had to make some changes. The first one was to compile the ParCeL6-1 with the “g++” compiler, because being a pure C library we used to compile it with “gcc”. For this, no special changes were required. We just had to rename the files’ extension to “cc” instead of “c” and to change the “Makefiles” accordingly, also replacing the C compiling flags with the C++ ones and adding the required path for SSCRAP.

The creation of threads needed also to be changed. We had to modify the main process of ParCeL not to create any threads and leave this to the SSCRAP library, which will create the number of threads specified in the command line arguments, transparently to the library client.

Another change we had to do was changing the global structures of ParCeL6 (Figure 5.3). We had to somehow put the global structures of every thread in a private space available only to it. Because we ran under shared memory with pthreads, this was very easy to implement with the help of the routines and structures provided by the pthreads library: “pthread_key_t”, “pthread_setspecific”, “pthread_getspecific”. This way, we were able to simulate the private memory space of every processor, exactly as in distributed memory environments.

![Figure 5.3: Changes of the global structures](image-url)
5.2.2 Data types added – “p6chunk” and “p6apointer”

SSCRAP’ „chunk” is a simple handle to a part of a memory segment, defining an offset and a length of what part should be mapped. Such a chunk is not mapped on construction, therefore it has to be mapped explicitly when used (the implementation provides methods for mapping and un-mapping, as well as for the access to the mapped memory).

SSCRAP’ „apointer” provides you a pointer to the mapped data of a chunk. It is usually better to access the data by means of the class „apointer”. It will ensure that the data is mapped into address space and can then be used almost like a normal C pointer. Afterwards, on destruction, it also does the un-mapping of the memory.

In spite of the large range of features they provide, when working with these data types („chunk” and „apointer”), we realized that we need more than their implementation provides. Therefore we had to create our derived data types. We built a p6chunk class inherited from chunk and a p6apointer class. As they inherit from template classes, they are also templates.

The features added by p6chunk are the possibility to know at every moment the number of elements stored in the chunk (the number can be also reset and changed anytime) and it offers also the possibility to define three types of chunks:

- **CLASSIC**
  As the name says, this is a classic chunk, meaning that is exactly as the chunk defined by SSCRAP, with the same functionality.

- **ORDERFREE**
  The “ORDERFREE” p6chunk adds some extra features to the normal chunk. It is used to hold elements for which the order in what they are stored doesn’t matter, and nothing will happen if the order of storage is changed. When deleting an element, the last element in the p6chunk will take its place, and the chunk size will be decremented. At adding, it works as CLASSIC chunk and adds the element at the end.

- **STRICTORDERED**
  This p6chunk was introduced to offer the possibility of keeping elements strictly order by some criteria. For example, we use it to keep the requests for the output values refresh, and we want to have the elements ordered by the processor number whom they are destined. Whenever an element is inserted, the right place for it is found, and all elements that are on the right of this space are shifted one position to the right (Figure 5.4). The delete operation is performed in the same manner, meaning that when an element is deleted, all elements in its right are shifted one position to left in order to fill the empty space created.
Figure 5.4: Insert operation for a STRICTORDERED p6chunk

- FREESPACECHAINED

This is the most complex structure of the p6chunks (Figure 5.5). It was introduced in ParCeL6-2 in order to replace the chained tables with an equivalent mechanism. It contains the data that needs to be stored mixed with a ParCeL6-2’s specific data type for the free space. The free space elements in the chunk contain references to the next free space element, in this way creating a chained list of free space.

When an element is inserted, first a search is performed through the free space list to see if there can be found a place to put the new element. If it is, the free space element is removed from the free space list, and the new element takes its place in the chunk. If it is not, the new element is inserted at the end of the chunk.

When deleting an element, in its place in the chunk is put a free space element which is chained in the free space list for later usage.

Figure 5.5: FREESPACECHAINED p6chunk

The p6apointer class is a different implementation of a “pointer” used for accessing p6chunks. First when we start developing ParCeL6-2, we used the classical apointer provided by SSCRAP. But we realized that the way this class works gives a very bad impact on the execution speed. This is because when declaring an apointer for a chunk, it will map the chunk into memory, and when getting out of scope it will remove it. So, using the apointer will result in a lot of mapping and un-mapping memory operations, which are very time-consuming. This is why we decided to use a more appropriate method for chunk accessing. p6apointer maps the chunks into memory when it is first declared, and it will never un-map it, which got us a good speed improvement. Also defining our own apointer class allowed us to implement all the operations for chunk manipulation for all types of p6chunk. This means that when using this class in the ParCeL6-2, we didn’t care of the type of p6chunk we were using. We simply performed “add” and “remove” operations on the p6chunk with the help of the p6apointer, and the implementation of the p6apointer took care of calling the right
operations according to the type of \textit{p6chunk}.

\section*{5.2.3 Missions routing}

The mission routing (Figure 2.3) is a key concept in ParCeL6, which is also in charge with the synchronization of all the processors. The missions are sent from the main process to all others. This is a classical example of the one-to-all communication that is present in MPI for example. SSCRAP also provides a method called \textit{one_to_all} for this type of data exchange, which we used for implementing the missions’ exchange. A very good feature of this method is that it insure the synchronization of the threads that called it. The threads that call this method will wait until every thread from the program calls it. This means that we just have to call this method from every thread, and we obtain the synchronization needed without the use of synchronization barriers employed by ParCeL6-1.

\section*{5.2.4 Requests}

The requests tables in ParCeL6-2 were completely changed from the ones in ParCeL6-1. Now, all the requests (creation, destruction, connection, disconnection, kill-signaling) are stored in \textit{p6chunks}. The storage in the ParCeL6-2 was changed because now the requests have to be routed from a processor to others. In ParCeL6-1 version, a thread simply accessed the request tables of all others. In the distributed version, this is not possible anymore, and there has to be a request sending and receiving between processors. Because SSCRAP uses for most of the communication mechanisms the chunks, it is better to store them directly into these structures than to loose computational time transferring from tables to chunks whenever a communication is needed.

For all the requests, the communication method chosen was the \textit{general communication} also called \textit{bulk communication}. It is basically the exact exchange type the \textit{bulk} communication was designed for: every processor has a chunk containing the requests (part of them for one processor, part for other) generated in the computational cycle and he wants to send every one of them to the correct destination. There is a chunk for the creation, destruction and kill-signaling requests, a chunk for connection of the cells, one for the connection responses and one for disconnection. When the requests need to be exchanged, a \textit{bulk} communication is performed and every processor gets the requests destined to him. The \textit{bulk} communication ensures also the synchronization needed: the threads will call the \textit{bulk} and the operation will end only when every thread receives its requests.

A picture describing the request exchange mechanism can be seen in Figure 5.6. Every process P1, P2, P3 holds requests for all other processes, and after the \textit{bulk} communication all processes have received their data.
5.2.5 Permutation tables

The permutation tables allow the user to impose a specific execution order for the cells. The user can specify his permutation tables in the main sequential program, and with a special mission they are updated on every processor.

Being defined in the main sequential program, the permutation tables have to be sent from the main processor to all others. We chose for this operation SSCRAP’s one_to_all communication. The tables are defined line by line by the user with the help of the malloc function. We kept this type of definition for these tables in ParCeL6-2. Anyway, as the communication methods need a contiguous memory space to operate, we have to send as many messages as the number of lines in the permutation tables.

5.2.6 Input / output data exchange

The input / output mechanisms allow the user to transfer data between the sequential main program and the other processes.

For ParCeL6-2 we had to implement this mechanism from scratch, because in ParCeL6-1 there is no need for such features. The user simply declares the data types he wants to, then declares some variables and put them in the place indicated by the library. After that he uses them without any problem. In ParCeL6-2 this was not possible anymore, because the data declared by the user has to be transferred to all processors.

Because when building the library we could not make any assumptions about the data types the user is going to send us, we had to transform a part of the ParCeL6-2 library in a template library. This way, we can support any data type the user needs to use, and we can safely perform the data exchange between processors. The template approach was possible to implement because also SSCRAP is a template library, and so it can operate with any data type.

Because we wanted to put everything that is related to this data (we can called it “UserData”) – storage, methods for adding/extracting, for exchange – we built a template class called p6UserData that encapsulates everything that is related to “UserData”. The class
also contains three types of chunks, one for every type of “UserData”: “global” data, “local” data and “result” data.

The user has the possibility to refresh at any moment these kinds of data for all the processors, by calling the appropriate refresh method. We implemented this by introducing three new missions in ParCeL6-2: REFRESH_LOCAL_USERDATA, REFRESH_GLOBAL_USERDATA and REFRESH_RESULT_USERDATA.

Further, we will try to present every mechanism we used for exchanging the three types of input/output data.

**Local data exchange**

The local data is a sort of data that is specific to every cell, for example the cell coordinates inside the cellular network. It is sent from the sequential program to all processors ([Figure 5.7](#)).

The cells are all created by the sequential program in ParCeL6-2. When a cell is created, the individual parameters are stored in the correspondent chunk using an instance of the `p6UserData` class. Every piece of data from this chunk has to be sent to the processor that hosts the corresponding cell. Because we have a chunk filled with different pieces of data for different processors, we used the bulk communication to exchange it. When the user calls `p6_net_refresh_local_user_data`, the corresponding REFRESH_LOCAL_USERDATA mission is executed, a bulk communication takes place and every processor gets the data for its cells. In particular, a method of the `p6UserData` is called to effectuate the communication. Thus, after receiving the data, every cell can access its individual parameters its host processor. The data can be accessed only in read-only mode. It is not meant to be modified (like parameters, not variables).

For the bulk communication to work properly, every data element that is involved in such communication has to hold the necessary information from which the destination processor can be extracted. For this reason we had to impose a condition to the data type the user declares for local I/O. The data type is compulsory to be a structure that has an element called “destinationProc”. We do not restrict the type of the rest of the data contained in the structure. The user can use it to store whatever he thinks is necessary.
Global data exchange

Global data is the information that is used by all cells of a processor. A copy is sent from the sequential program to all other processors (Figure 5.8). We can think of this data for example to be an image from a camera that needs to be processed by the cells.

In the sequential program, the user declares the global data of the type he wants to use, and stores it in a chunk from a \textit{p6UserData} instance. To send it from the main processor to all others, he calls a function defined for this operation, \textit{p6\_net\_refresh\_global\_user\_data} which will invoke the mission \texttt{REFRESH\_GLOBAL\_USERDATA}. The mission will call a method from \texttt{p6UserData} class which will take care of data transfer. So on every processor we will have a copy of the data.

In the implementation, we used for the communication the SSCRAP’s \texttt{one\_to\_all} function.
Result data exchange

The result data is used whenever the user wants to have the results of the computation. Any time at the inter-cycles from the main program in the main process during the execution he can gather the output values (values that were computed) from all the cells. In the shared memory it was no need for such a mechanism, as the user was able to access directly the output tables of all processors.

Letting the user perform the gathering operation whenever he needs to, determined us to provide a method also for this kind of refresh. The method is `p6_net_refresh_result_user_data`, and as for all other user refresh methods, this method invokes a mission, `REFRESH_RESULT_USERDATA`. Again, a similar mechanism to the one presented above, will call a method of the `p6UserData` class that will take care of communication. Also the result data is stored in a chunk of the `p6UserData`.

An important difference from the other data types refreshing is the SSCRAP method we used in this case. SSCRAP provides a unique `gather` mechanism which we used here only in the ParCeL6-2 implementation (we call this `gather` because the computed values from all the processors have to be sent to the main processor – Figure 5.9). We declared an `array` on the main process large enough to hold all the data that will be sent, and on all other processors we declared an empty array (size 0). After that, we used the “<<” overloaded operator with the arrays declared which performs the gather operation, and all data gets to the main processor.

5.2.7 Output values exchange
The output values (computed values of the cells) exchange was the most difficult to implement, especially because of its impact on the efficiency of the library. Being an operation that is executed at the end of every cycle, which means a lot of times if we think that the number of cycles could be equal to some thousands, it has to be very fast. Otherwise, the ParCeL6-2 would not be a very interesting solution to cellular computation.

The need for output values exchange comes when the cells that connect to other cells need their output. In fact, this is the reason for which a cell A would connect to a cell B – A needs the output of B for performing the computation.

The way the output values are exchanged makes the difference between the two models of ParCeL6-2: buffered and hybrid.

Data exchange in buffered mode

When cells work in buffered mode, means that the output values are not exchanged inside the computational phase cycles, but only between cycles. So, we can imagine that cycle is composed by two steps: computation and output values exchange for the connected cells. This means that if during the computation a cell “A” needs the output value of another cell “B”, it will always use the old output value received, and not the current output of “B”, which might be different if the cell “B” has been computed at that time.

The buffered mode works in a query-reply mode: every processor will send to other processors the identifications of the cells for which it needs the output values, and will get back the values requested.

The output values of the cells are stored in chunks – p6chunks. Every process has a chunk called “OutputValuesChunk” that holds the output values for all the cells. This change appeared in ParCeL6-2 for speed improvement - now we use indexes for chunk access which are very fast. A cell knows its index in this chunk where it is allowed to store its computed values. Different cells may have different number of outputs, and so they have to store also the output’s number. For storing the refresh requests, also a chunk is used: “OutConnsRefresh”. Elements stored in this chunk contain: destination processor for the request, remote cell output index and the number of outputs for that cell. After the output values are received from other processors, they are stored also in a chunk, “OutConnValues”. In the computational phase, the cells will use this chunk to read the output values of their connections.

Before starting to explain how the buffered mode was implemented, we will present the way connections are performed in ParCeL6-2. When one cell from a processor “Px” wants to connect to a cell hosted by processor “Py” several steps have to be performed (Figure 5.10):

1. Processor “Px” requests a connection to a cell hosted by processor “Py”. The request contains the identification number of the cell to connect to.
2. Processor “Py” responds to the connection request with the index of the cell in the “OutputValuesChunk” and with its number of outputs.
3. Processor “Px” stores the response received in the “OutConnsRefresh” chunk. Then, when it will need to update its OutConn, it will be able to identify the corresponding remote output.

![Cell connection steps diagram](image)

**Figure 5.10: Cell connection steps**

From the communication point of view, we used *bulk* communications for all data exchange regarding the output values in the buffered mode. Exchanging of the output values is performed in two steps (*Figure 5.11*):

1. Every processor sends the elements of his “OutConnsRefresh” chunk to destination processors for those elements. After a *bulk* communication is performed, on every processor will be the requests for the output values of the cells hosted by it.

2. In this step, all processes process the requests receive and build a chunk containing the response. The response chunk is filled with elements containing double values and destination process. The second *bulk* communication is performed and after that the data requested in the first place will be on every process. Then a match is performed between the initial requests and the data received in order to store the double values in the right place in the “OutConnValues” chunk.
As presented in the description of the programming model of ParCeL6-2 we implemented also a “tuning” feature for the step 2 (Figure 5.11) of the outputs exchange. We introduced the possibility for the user to control the amount of data exchanged in the step 2. For example, he can choose that not all the data to be exchanged, but only the one for processor 1 and 2, and after that for processor 3. Actually he can define the number of processors that will get their requested data. The number can be from 1 to the total number of processors. The default value is the number of processors meaning that all the data is exchanged in one single communication. If he chooses the number lower than the number of processors, the communication will be divided in small steps, as many needed to exchange the data for all processors. This option was introduced for the case when the machines running ParCeL6-2’s applications don’t have enough memory; dividing the communication in smaller steps, the same memory space for every sub-step can be the same for all of them, and obviously is smaller.

**Data exchange in hybrid mode**

For the hybrid mode, the version inspired from the buffered model was chosen in the implementation. In the ParCeL6-1, for refreshing the connection’s values in direct and hybrid mode, we used a function named `p6RefreshOutConn` (called “on demand”). This function is no longer needed in ParCeL6-2. The direct mode has disappeared, and the hybrid one has been changed. Now the refreshing of the hybrid cells is done for a group of cells and not cell by cell as in the previous version. Also the flags that showed the state of the “OutConn” –
already refreshed or not in the current cycle, were eliminated in this version.

The computation on every process is performed by a function called
\textit{p6ThreadCellComputation}, which takes every cell in a turn and calls its iteration function. The \textit{p6ThreadCellComputation} function was changed in ParCeL6-2. We introduced inside it the notion of sub-cycles. In a sub-cycle, the process will compute a certain number of cells (given by the user), and once the computation is finished, it will begin to refresh the same number of cells. Choosing the output cells to refresh is done randomly at the beginning of the sub-cycles (among cells not yet refreshed) and then at every sub-cycle we refresh another part of the cells. The randomly chosen cells for refreshing insure the fact that not exactly the computed cells are the ones refreshed, which preserves the asynchronism of the application. 

The \textit{hybrid} mode was implemented by another Supélec student, Laurent Casse.
Chapter 6

Tests and benchmarks for distributed model

After we finished developing ParCeL6-2, we made some tests and benchmarks on different machines, single and multi processors, to see the difference in execution time between this version and the shared memory one. During development, we used for testing the Jacobi relaxation program. Jacobi relaxation is a very simple application which uses most of the features of ParCeL6, so it was a very good tool to use at the time we were writing ParCeL6-2.

In this chapter, we will present the Jacobi relaxation program for ParCeL6-2 as an example of library usage, along with some benchmarks we did.

6.1 Example of a program using ParCeL6-2

Jacobi relaxation is an iterative algorithm, which given a set of boundary conditions, finds discrete solutions to differential equations of the form $\Delta^2 A + B = 0$. It uses a matrix of elements. At every iteration step, the Jacobi updates the value of each node with the result of computing the average of its four neighbors. For example, at step “k+1”, for the element being at “i”, “j” coordinates, we will have the following formula:

$$A_{i,j}^{k+1} = \frac{A_{i+1,j}^k + A_{i-1,j}^k + A_{i,j+1}^k + A_{i,j-1}^k}{4}$$

A Jacobi relaxation program of ParCeL6-1 was available at the time we start the development for ParCeL6-2. We adapted it step by step while we developed ParCeL6-2, so now we have a complete version of it. We will present it in this section, as it is very representative for the usage of the library we developed.

A simplified scheme of a generic ParCeL6 program is shown bellow:

Main() // Example of ParCeL6 program
{

// Example of ParCeL6 program

Cell network definition (sequential)

Cell network creation (parallel)

LOOP:
   I/O operations – send data to the cells
   Cellular computation (parallel)
   Evolution of the cellular network (parallel)
   I/O operations – gathering computed values
   Destruction of cells (sequential)
   New cells definition (sequential)

Requests for network destruction (sequential)
Auto-destruction of cellular network (parallel)

The Jacobi follows this scheme. We will present the necessary code for writing the Jacobi program, and after that we will explain every step performed in the program:

//definition of LocalUserData, GlobalUserData, ResultUserData
//data structure
//definition of JacoInit, JacoIter, JacoTerm functions

int sscrap_main(…)
{
   ...
1. if (mynum()==0) {
   ...                     
   p6_init_mission();

2. // Permutation table definition
   PermutTab = (int **) 
   p6_malloc_or_exit(sizeof(int*)*PERMUTTABNB);
   p6_basic_srand(1123);
   for (i = 0; i < PERMUTTABNB; i++) {
      PermutTab[i] = (int *) 
      p6_malloc_or_exit(sizeof(int)*PERMUTTABSIZE);
3. // Define a Kohonen map (ask for future creation)
for (lin = 0; lin < LINE+2; lin++)
    for (col = 0; col < COLUMN+2; col++) {
        LocalUserData[lin][col].col = col;
        LocalUserData[lin][col].lin = lin;

        //the user data for this cell
        int cellIndex =
            userData.addLocal
                (LocalUserData[lin][col], p6_get_main_proc_info_pt());

        GlobalUserData.PtJacobi[lin][col] =
            p6_define_cell(1, P6BUFFEREDOUT, FALSE, cellIndex,
                JacoInit, JacoIter, JacoTerm, NULL, NULL,
                p6_get_main_proc_info_pt());
    }

    userData.addGlobal(GlobalUserData);

4. // User Data refresh
    p6_net_refresh_local_user_data<TEMPLATE_PARAMETERS>(P6NODEBUG);
    p6_net_refresh_global_user_data<TEMPLATE_PARAMETERS>(P6NODEBUG);
    p6_net_refresh_result_user_data<TEMPLATE_PARAMETERS>(P6NODEBUG);

4. // Net creation
5. p6_net_creation(P6NODEBUG);

6. // Net activation
7. p6_net_computation(P6FORWARD, 0, P6NODEBUG);
8. p6_net_link_evolution(P6NODEBUG);

8. for (cycle = 1; cycle <= CYCLENB; cycle++) {

    p6_net_computation(P6ALTERNATE, cycle%PERMUTTABNB, P6NODEBUG);
G);
        p6_net_refresh_out_conns(P6NODEBUG);
    }

    // Ask to kill all cell of the Jacobi map

9. for (lin = 0; lin < LINE+2; lin++)
    for (col = 0; col < COLUMN+2; col++)
        p6_kill_cell(GlobalUserData.PtJacobi[lin][col],
                    p6_get_main_proc_info_pt());

10. //executes kill rq
    p6_net_cell_evolution(P6NODEBUG);
    //execute term function
    p6_net_computation(P6ALTERNATE,cycle%PERMUTTABNB,P6NODEBUG);
    //execute kill signal rq
    p6_net_cell_evolution(P6NODEBUG);

    // Stop threads and free global variables
    p6_finalize();

    // Return result of the run
    fprintf(stdout,"p6: End!\n");

} else{

11. p6TotalMission<TEMPLATE_PARAMETERS>(&p6GlobalVars->p6ProcInfo);
}

return 0;
}

12. P6_TEMPLATE_INSTANTIATION(LocalUserDataElt_t, GlobalUserDataElt_t, ResultUserDataElt_t);

At the beginning of the program, the structures for the I/O data exchange have to be defined, along with the three functions for cell initialization, iteration and termination. For this example, the Init function of a cell performs the necessary connections with the four neighbors: NORTH, WEST, SOUTH and EAST. The Iter function will just compute the output value of the cell by performing the average of its neighbors, and the Term function runs some memory cleaning and also sends to all other cells that it has been terminated, in order to avoid connection to a dead cell.
As we can observe from the listing, every operation is coordinated by the processor 0 (line 1). All other processes just run \texttt{p6TotalMission} (11) which wait in a loop for missions and execute them. The \texttt{p6\_init\_mission()} will send the INIT mission to all process for internal allocations of needed structures. After the initialization, the user can define the permutation tables (2) on processor 0 and broadcast them with \texttt{p6\_init\_permute}. The next step is to define the Jacobi map, by defining all the cells (3). Defining the cells puts a cell creation request in the correspondent request chunk of processor 0, and when running (5) the requests are routed to the destination process that will create and host it. The process that will host the cell is also specified by a load balancing technique at cell definition. At the end of cell definition, the I/O data is completely filled with cell registration and so it can be sent to the other processes (4).

The (6) line will send the mission for running the computation. Being the first computation, the cells will run their \texttt{Init} function and not the \texttt{Iter} one. After running the \texttt{Init}, current function to be executed by the cells on the computational phase will be the \texttt{Iter} one. The \texttt{Init} in Jacobi deals with the connection requests, so after all cells have run this function, all connection requests are placed in the corresponding chunks. In line (7) these requests are sent to their destinations, and executed by each process.

Until step (8), basically only the initialization of the Jacobi program is performed. After that, in the (8) step the real computation takes place. The computational loop has two functions: \texttt{p6\_net\_computation} that will compute the cells, and \texttt{p6\_net\_refresh\_out\_conns} for refreshing the “OutConns”. Refreshing the “OutConns” means that the output values of the cells will be sent to the ones that are connected to them. This will ensure that at the next computational step, the cells will have new values for their neighbors to whom there are connected.

After the computation has finished, requests for killing all the cells are made (9) by the processor 0 and they are sent and executed by all processes (10).

ParCeL6-2 being partially transformed in a template library, we have to make some template instantiations. For this, we supply the user a macro that will perform the entire job (12).

### 6.2 Benchmarks Results

We made some benchmarks of ParCeL6-2 and also of ParCeL6-1 in order to make some comparisons between the two versions. The benchmarks we did were both on shared memory machines to be able to compare the two implementations in the same environment, and also on distributed environment (cluster of PCs). We benchmarked ParCeL6 both with the SSCRAP’s ShM version and with the MPI version.

#### 6.2.1 Benchmarks done with the ShM version of SSCRAP
All the benchmarks that we present in this section were done on a multiprocessor machine called Quadx2. Quadx2 hardware characteristics are the following: 4 PIII processors at 700MHz with 2MB of cache memory each. The benchmarked program was the Jacobi relaxation, with a matrix size of 500x500 and during a number of cycles equal to 1000. For every point in the graphics we made four benchmarks and compute their average.

Initially, when we first finished the developing of ParCeL6-2, we did some benchmarks. As is can be seen from Figure 6.1, the iteration time we obtained initially was way to big compare to the one of ParCeL6-1. The lowest difference we obtain was for three processors, and this was 179% of ParCeL6-1 time. For more than three processors, the time increases a lot. For 8 processors for example we have in ParCeL6-2 375% iteration time of ParCeL6-1. The initialization time though (Figure 6.2) we obtained was a very good one, almost the same as shared memory version. Because the initialization contains a lot of communications (both one_to_all for missions and bulk for the requests) and with all these we obtained such a good time, we draw the conclusion that SSCRAP is an efficient library (on shared memory at least), but some of the computation loop needed to be improved. So we looked at the function that was in charge with the output values exchange to see if we can make some optimizations, because only this function could have led to such a big computational time. We noticed a sorting we were performing in that function, sorting that meant to iterate multiple times through a big chunk, so we imagined a quicker solution and we implemented it. As it can be seen from Figure 6.1, the times obtained after we did the optimization are better, but still not very close to the ParCeL6-1 ones. The initialization time (Figure 6.2) kept quite the same curve until 8 processors, but increased dramatically after that. This is not such an important thing, because with more than 4 processors we have hyper-threading for our 4 processors testing machine, and ParCeL6 main objective is to obtain the best performance with 1 process / physical processor. We also made some curves for the total time (initialization time + iteration time) - Figure 6.3.

After a meeting with the persons that built SSCRAP where we discussed the results obtained so far, we reach the conclusion that the Jacobi example is not the best to evaluate the times of ParCeL6-2 with SSCRAP because the computational phase of Jacobi is just an average of four values, which is too little compared to the time for values exchange. So we decided to make another series of benchmarks, with a modified Jacobi. Inside the iteration function, we introduced a loop of 2000 cycles that was only dividing two numbers. The results are presented in the Figure 6.4 – iteration time, Figure 6.5 – initialization time and Figure 6.6 the total time. The graphics are drawn for the ParCeL6-1 and for the optimized version of ParCeL6-2. The results obtained are encouraging, although is still an important difference in the times of executions for the two versions. The initialization time (Figure 6.5) didn’t change because we only modified the computational loop. As we can observe from Figure 6.4 comparing with Figure 6.1, when using more computation, the results are better, and we even obtain a speed-up for 4 processors, which is not true for Figure 6.1. The curves tend to be parallel for the two implementations.
Figure 6.1 – Iteration time of Jacobi in three implementations: ParCeL6-1, ParCeL6-2 and ParCeL6-2 after some optimizations

Figure 6.2 – Initialization time of Jacobi in three implementations: ParCeL6-1, ParCeL6-2 and ParCeL6-2 after some optimizations
Figure 6.3 – Total time of Jacobi in three implementations: ParCeL6-1, ParCeL6-2 and ParCeL6-2 after some optimizations

Figure 6.4 – Iteration time of Jacobi with ParCeL6-1 and ParCeL6-2 -> more computation added in the iteration loop
6.2.1 Benchmarks done with the MPI version of SSCRAP

The first benchmark we did for the ParCeL6-2 MPI version was meant to show the serial execution time compared to the ParCeL6-2 ShM version and also with ParCeL6-1
We used “monox1” (P4 XEON 2.4 GHz, with 512MB cache and 1Gb RAM memory) for the hardware and Jacobi relaxation (200x200 matrix, 2000 cycles) problem for the testing software. The results obtained are close to what we would have expected. The execution time for the ShM version is smaller than the MPI one, but the difference is not very significant. A bigger difference can be observed when comparing with ParCeL6-1.

![Figure 6.7 – Total time of 200x200 Jacobi with ParCeL6-1 and ParCeL6-2 (ShM and MPI)](image)

Another series of benchmarks we did on a cluster of 8 machines, monox1 to monox8, theoretically all with exact the same hardware configuration (P4 XEON 2.4 GHz, with 512MB cache and 1GB RAM memory). The curves for the initialization time (Figure 6.8) are very good and very close to the perfect deviation. A small time increase can be observed from 8 processes to 16. But this is normal, because the cluster has only 8 physical machines. All graphics for different size of the Jacobi matrix (big, medium and small) keep the same slope and are almost parallel, which means the behavior remains the same when modifying the size of the problem.

The cycle time (total computational time divided by the number of cycles) we measured (Figure 6.9) also has a very good curve with the first 4 of the cluster’s machines. The graphics is good for all three sizes of the matrix. The reason for the shape of the graphic when using machines 5,6,7,8 we think is a hardware problem, most probably related to the network layer. This is the reason we kept doing benchmarks (medium and small size of the matrix) only with the first four machines of the cluster.
Interesting results we obtained on Quadx2, when comparing the ParCeL6-2 SSCRAP ShM and MPI versions. For the initialization time (Figure 6.10), both versions gave almost the same results, with small exceptions, but for the cycle time (Figure 6.11) the ShM version is much faster. Probably this depends on the hardware used, because on the first benchmark (Figure 6.7) the difference is very small. The slope is very good for the MPI version, but the times are not.
Figure 6.10 – Init time of Jacobi for ParCeL6-2 SSCRAP ShM & MPI on Quadx2

Figure 6.11 – Cycle time of Jacobi for ParCeL6-1 and ParCeL6-2 SSCRAP ShM & MPI on Quadx2
Conclusion

During my internship at Supélec, I participated to every stage of development of ParCeL6-2. When I first arrive at Supélec and start working, I begin with the debugging of the shared memory version ParCeL6-1. This helped me to fully understand the computing model of this library and to have a complete knowledge about its implementation.

Concurrently with the debugging phase, a series of discussions and meetings led to the defining of the ParCeL6-2 model, and some ideas about the implementation. All my work has been done in parallel with Laurent Casse, a DEA student of Supélec. We both worked at the same thing, but each following its own way of implementation. This helped us to do a quicker and better development of the library – both from efficiency and quality of written code points of view. Our work has split when we reach the implementation of the two models of cell outputs: buffered and hybrid. My work was to write the outputs exchange in buffered model.

ParCeL6-2 is a mixture of the work and research of several people from both Supélec and LORIA laboratory. First, the two different output models were developed by me and Laurent using the SSCRAP library acquired at LORIA by Jens Gustedt, Mohammed Essaidi and Isabelle Guérin Lassous. The cortical applications of ParCeL6 are written by Hervé Frezza-Buet and Olivier Menard. We managed to put together successfully all this people’s work in the process of creation and testing of ParCeL6-2. For the user’s point of view, we tried to maintain exactly the same features as the ones offered by the shared memory version (e.g. cell computed values available on-demand), in order to make possible the migration of the applications between the two versions.

ParCeL6 has evolved from the shared memory implementation – April 2004 to shared memory under SSCRAP at the end of June 2004. The very next steps are running on distributed architectures (clusters) and library optimizations in this scope. In the future, we hope that ParCeL6-2 will successfully run on computer grids.
Perspectives for future development

ParCeL6-2 is now a fully distributed version of ParCeL6 library. Although, because the SSCRAP library is under development right now (20 of June 2004), the only tests and benchmarks of ParCeL6-2 were performed on shared memory machines. A series of possible optimizations were discovered when running benchmarks. Part of them were implemented, another part remains to be done.

In short term (1 month), the objectives are to obtain approximately the same performance running under SSCRAP in shared memory as the classical shared memory version, ParCeL6-1, and to insure fully integration with all the applications of this library.

As a medium term objective (2-3 months) is running ParCeL6-2 on distributed architectures – clusters. One of the features of SSCRAP is to run the same application under shared or distributed memory (over MPICH, LAM) without any change to the written code. This means that when a SSCRAP version that works properly over MPI will be available, no change to ParCeL6-2 will be necessary to run it on clusters.

The time ParCeL6-2 will be available on clusters, a more concluding series of benchmarks will be possible to be done. That will show if the implementation can offer the necessary computing power and speed to be really valuable for current physical and cortical applications.

A long term objective (1-2 years) is running on computer grids. We think that this started to be the only architecture for parallel computation which offers the computation power needed at reasonable costs.
Bibliography

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


