CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Electronic Thesis and Dissertation Archive - Università di Pisa

University of Pisa and Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking
Corso di Laurea Magistrale in Informatica e Networking

Master Thesis

Multithreaded support to interprocess communication in parallel architectures

Candidate
Tiziano De Matteis

Supervisor
Prof. Marco Vanneschi

Academic Year 2010/2011

Alla mia famiglia

There is nothing so stable as change

Bob Dylan

Ringraziamenti

La Laurea è senz'altro uno dei momenti più importanti nella vita di una persona; mi sembra quindi doveroso e giusto condiverla con chi, in tutti questi anni, mi è stato vicino.

Il primo, enorme, grazie, va senz'altro alla mia famiglia: papà, mamma e sorellina. Molte delle difficoltà affrontate in questi anni sarebbero state insormontabili se non avessi avuto voi alle mie spalle, pronti a sorreggermi nel momento del bisogno. Insieme a loro, un pensiero va anche a tutti gli altri parenti: zii, nonne, cugini e cugine.

Un sentito ringraziamento va al prof. Marco Vanneschi, per avermi fatto apprezzare una branca dell'informatica per me nuova e, in generale, a quei professori che in questi anni hanno contribuito al mio amore per questa disciplina scientifica.

Passando al lato amici, qui la lista è lunga, ma permettetemi di iniziare con due ringraziamenti particolari. Il primo va ad Andrea e Bianca, con cui ho condiviso sei anni di vita universitaria e, spero, di condividere tante altre avventure. Incontrare persone di una simile bontà è abbastanza raro e io, in questo, mi ritengo molto fortunato: grazie. Il secondo va ad Andrea e Luigi, compagni fedeli di università. In questi anni, fra le trollate di Andrea e i piani industriali che ho condiviso con Luigi, siete sempre stati pronti ad offrirmi un buon consiglio e un appoggio, per me fondamentale.

Come non ringraziare poi i miei coinquilini del Rosellini, vecchi e nuovi: Momo, Nicola, Luca, Dima, Ivan, Peppe, Diana, Manu, Nicolò, Salvatore, Guido e tutti gli altri che ora mi stanno sfuggendo. Condividere con voi lo stesso tetto, è

come far parte di una grande famiglia allargata e, proprio per questo, l'aver vissuto in una casa dello studente sarà un'Esperienza che mi porterò dentro per sempre, insieme alle vostre amicizie. Al loro fianco, un grazie va anche a Livia e Chiaretta (durighèn!), protagoniste di tante belle serate passate assieme in quel di Pisa.
Lato università, un grazie va senz'altro ai colleghi del Lab (booon per voi): Fabio e Albe, con cui ho condiviso l'avventura "tesistica", il Good, fonte di tanti consigli, quel numerico americano del Gufetto, Dani, Alessio e il Farru. Vanno ringraziati anche tutti gli altri compagni di studio: Ema, Alessio, Ciccio, Stè, Gab, il Boggia, Monica, Mary, il Pucc, Anna, il Fulge, Victor, Elian, Alice, tutti gli MCSN guys (un giorno troverete quella spia, ve lo auguro) e quel mio tempio personale che è la biblioteca di scienze!
Un pensiero sentito, lo dedico anche ai miei amici di giù: Serse, Pino, Tuta grande e Tuta piccolo, Junior, Teddy, i Cabù e Francesco ... nonostante passi con voi poche settimane all'anno, è come se non me ne fossi mai andato da Taviano. Vi porto nel cuore.

Sorrido di gioia nel vedere questa lunga lista e mi scuso con chi, per mia disattenzione, non è stato citato. Tante cose sono successe in questi anni, ma non posso che soffermarmi sul fatto che ognuno di voi mi ha lasciato dentro qualcosa.

Grazie.

Tiziano

Contents

1 Introduction **1**

2 Multithreaded Architectures **7**

2.1 Taxonomy of multithreaded architectures 9

2.1.1 Interleaved Multithreading 9

2.1.2 Blocking Multithreading 10

2.1.3 Simultaneous Multithreading 11

2.2 Intel's Hyper-Threading Technology 12

2.3 Multithreading in Parallel Computing 13

2.3.1 Speculative Precomputation 15

3 Communication in parallel applications **17**

3.1 Overlapping communication and computation 19

3.2 Communication processor 21

3.2.1 Commercial implementation of communication processors 24

4 Multithreaded support for interprocess communication **27**

4.1 Programming in multithreaded architectures 29

4.2 Inter-thread synchronization mechanisms 30

4.2.1 POSIX 31

4.2.2 Spin lock 32

4.2.3 `MONITOR`/`MWAIT` instructions 34

4.2.4 Evaluation of the synchronization mechanisms 36

4.3 Runtime support of LC 39
4.3.1 Zero copy optimization and shared memory segment . . . 39
4.3.2 Channel descriptor 40
4.3.3 Send and receive primitives 44
4.4 Communicator implementation 46
4.4.1 Communicator initialization 47
4.4.2 Delegation queue 47
4.4.3 Check buffer full 48
4.4.4 Send completion notification 49
4.4.5 Communicator finalization 49
4.4.6 Evolution of communication primitives 50

5 Tests and results 53
5.1 Stream computation 54
5.2 Data parallel computation 60

6 Conclusions and future works 67

A The library 71

CHAPTER 1

Introduction

Hardware multithreading is becoming a generally applied and commercially diffused technique in modern processors. Provided that proper architectural supports to program concurrency are available, during the same time slot, or a limited number of time slots, the units of a multithreaded (MT) processor are simultaneously active in processing instructions belonging to distinct programs. Underutilization of a superscalar processor, due to missing instruction level parallelism, can be overcome by this technology: the latency of any long-latency event (such as cache misses or long instruction executions) can be hidden by allowing multiple programs to share functional units of a single processor in an overlapping fashion.
The parallel activities executed by a multithreaded CPU are referred to using the term *hardware thread*: this is a concurrent computational activity supported directly at the firmware level. A specific runtime support of threads is implemented directly at the firmware level in order to be able to execute different threads simultaneously.

Theoretical and experimental results show that multithreading is well suited in a multiprogram environment, achieving a scalability that ranges in the interval 1.4-2.4 for a number of threads in the interval 2-8. Nonetheless, how to ex-

ploit this technology in *High Performance Computing* (HPC), in a systematic manner, is still unknown. Since parallel applications usually exhibit an high exploitable degree of parallelism, an obvious solution seems to be to increase the parallelism degree in order to exploit all the thread contexts offered by a multithreaded machine. Unfortunately this simple solution does not give the expected results: indeed, contention for shared resources limits the performance advantages of multithreading on current processors, leading to marginal utilization of multiple hardware threads and even to a slowdown.
In the literature many researches suggest the use of cooperating threads to increase the performance of a computation. *Speculative precomputation* (SPR, [4]) is an example of such techniques: it utilizes idle hardware thread contexts to execute helper threads on behalf of the main one. These threads, speculatively prefetch data that are going to be used by the main computation thread in the near future, thus hiding memory latencies and reducing cache misses. Experimental tests conducted on commercial multithreaded platforms show that, on a pool of benchmarks, SPR could give some benefits in some cases, while performs comparably to the classical program version in the rest.

The work of this thesis aims at investigating an alternative use of multithreading, by utilizing this technology as a support for efficient interprocess communications in parallel applications.

The working context: communications in parallel applications

The field of HPC is living a new impetus in recent years, thanks to the fact that multi-many core components, or on chip multiprocessors, are replacing uniprocessor based CPU, also at a commercial level. This fact has enormous implications on technologies and applications: in some measure, all hardware-software products of the next years will be based on parallel processing.
In a structured view to parallel programming, a parallel application can be seen as a set of cooperating modules that operate together to achieve a common goal; assuming a *local environment* (or *message passing*) model, this cooperation is

achieved by an exchange of informations between modules through communication channels.
Parallel programs are expressed as structured computation graphs, via high-level parallel programming tools, and are usually compiled into a set of processes in which the mentioned cooperation is expressed using concurrent programming languages. The basic interprocess communication primitives provided by such languages are *send* and *receive* commands; they are used for transferring a message from a sender to a receiver through a communication channel.
Since these processes alternate calculus and communication phases in a sequential fashion, it is evident that considerable advantages can be obtained by allowing the overlap between them. Supposing that proper architectural supports exist, a module can delegate the execution of a communication primitive to a secondary entity, going ahead with the calculus without waiting for its completion; in the meanwhile and in parallel the communication is performed by such secondary entity. In general, hiding the communication latency (at least in part) is allowed, resulting in a performance benefit that depends by various factors such as the structure of the application, the particular architecture, the amount of communications performed with respect to the computation. An example of such supports is the *communication processor*, that could be realized as a central, or as an input-output, coprocessor specialized at the firmware level. In particular, the input/output coprocessor solution is adopted when the communication processor is provided inside the interconnection network box.

The goal of this thesis is to realize a proper runtime support, for shared memory systems, of a basic concurrent language (LC, introduced in the HPC courses of the Master Program [19]), including the emulation of the communication processor facility through multithreading. The idea is to associate to each computational module (mapped as a process or a thread) one (or potentially more) thread that is in charge of executing the communication primitives, following a philosophy similar to the communication processor case. Performing basically a burst of load or store for transferring a message, hopefully the new

thread will not interfere with the computational one.

Research issues

Keeping in mind that the main objective is to implement a proper runtime support for LC, as library, on a MT architecture, through this thesis various problems will be addressed:

- understand how a multithreaded CPU actually works, what could be its limits (with a reference to a real architecture that will be used for the development) and what precautions take in programming in order to use it properly;
- how to efficiently implement the runtime support for the concurrent language, taking into account some optimizations that will be introduced (in particular the *zero copy* technique). It must be designed in a proper way to be executed in user space, saving additional overheads otherwise unavoidable;
- as explained, to each process/thread of the parallel application is associated at least one thread that is in charge of executing the delegated communication primitives. Considering the context of a MT machine, where the threads share the functional units of a processor, there is the need of understanding how properly manage the interactions between them, looking for mechanisms that have properties of high responsiveness and lightweightness.

Even if they will treated with reference to a particular system, these problems and their proposed solutions could be considered quite commons in exploiting these kind of architectures.

This work will allow us to determine whether the proposed use of multithreading is actually a viable alternative to other proposals and what benefits can results in parallel applications.

Structure of the thesis

This documents deal with the mentioned topics and it is structured in the following way:

1. in Chapter 2 the multithreading technique will be introduced in details. A brief taxonomy of multithreaded architectures and a commercial implementation (the Intel's Hyperthreading Technology) will be presented. Finally a brief discussion on the current use of multithreading in HPC is approached;

2. in Chapter 3 the impact of communications in parallel applications and the benefit of overlapping them with computation will be discussed. The communication processor is described, presenting some concrete implementations;

3. Chapter 4 deals with all the implementation details of this work. Measures to be adopted in programming, for efficiently exploit multithreaded system, are presented, with a particular attention on possible mechanisms for efficiently synchronizing two threads. Then the runtime support for a minimal core of LC in shared memory system is introduced and, subsequently, it is described how extend it in order to emulate the communication processor facility through multithreading;

4. in Chapter 5 results obtained from the comparison of the two different versions of the runtime support for LC, with or without the emulation of communication processor facility, are discussed;

5. finally, in Chapter 6 conclusions on the work done are reached and possible ideas for future works are briefly pointed out.

CHAPTER 2

Multithreaded Architectures

In the last few decades there has been an increasing complexity in processors design in order to achieve an ever increasing performance improvement. The earlier scalar pipelined CPUs evolved introducing new techniques such as branch-prediction, out of order and superscalar execution. Nonetheless, the major complexity design, transistor counts and power consumption that this solutions imply, do not correspond to a proportional increasing in processors performance.

Consider, as an example, a superscalar architecture. Superscalar microprocessors are able to issue multiple instructions (two to eight) each time slot (1-2 clock cycles) from a conventional linear instruction stream. To accomplish this goal, proper version of the functional units of a scalar pipelined CPU must be realized, in order to be able to offer a given average bandwidth. As the issue rate of microprocessors increases, the compiler or the firmware architecture will have to extract more *instruction level parallelism* (ILP) from a single sequential program to fully exploit these additional resources. However, ILP found in a conventional instruction stream is limited and this results in performance degradations caused by frequent data dependencies. Optimizations aim to mask latencies caused by such dependencies.
For these reasons, it has been introduced the idea of exploiting parallelism

among instructions of distinct sequential programs running on the same ILP CPU, by means of *multithreaded architectures* (MT architectures). Provided that proper architectural supports to program concurrency are available, this idea implies that during the same time slot, or a limited number of time slots, the CPU units are simultaneously active in processing instructions belonging to distinct programs. Allowing multiple programs to share functional units of a single processor in an overlapping fashion, multithreading can potentially hide the latency of any long-latency event (such as cache misses or long instruction executions).
The parallel activities executed by a MT CPU, on a time slot basis, are referred to using the term *thread* (or *hardware thread* in the literature), with a more specific meaning with respect to its traditional notation. Here a thread is still a concurrent computational activity but is supported directly at the firmware level. For this purpose, the firmware architecture of a MT CPU, able to execute at most m threads simultaneously, has to provide:

- m *independent contexts*, where, as usually, a context is represented by the program counter and the registers visible at the assembler level;
- a *tagging mechanism* to distinguish instructions of different threads within the pipeline;
- a *thread switching mechanism* to activate context switching at the thread level.

This means that there is a specific runtime support of threads implemented directly at the firmware level. From the programmer viewpoint, this may be not visible: a thread can be declared in the usual way, for example as a user or system thread, by means of Posix threads. However the compiler of a MT machine is different from a traditional one: it produces proper code to link the firmware level runtime support for threads.

In the following sections a brief taxonomy of multithreaded architectures will be given. Then a commercial implementation of a multithreading technique,

the Intel Hyperthreading Technology (that will be used also for development and testing), will be described and, finally, common use of multithreading in parallel computing will be discussed.

2.1 Taxonomy of multithreaded architectures

Multithreaded architectures fall into two main categories, according on how instruction emission is performed:

- *single-issue* architectures: during a time slot only instructions belonging to a single thread are issued; instructions belonging to distinct threads are issued in distinct time slots. This class can be implemented either on a scalar or on a superscalar architecture. In this category, a further distinction exists between *Interleaved Multithreading* and *Blocking Multithreading*;
- *multiple-issue* architectures: during a time slot instructions belonging to more than one thread can be issued. This class, which can be implemented only on superscalar architectures, is also called *Simultaneous Multithreading.*

2.1.1 Interleaved Multithreading

In the Interleaved Multithreading (IMT) approach, in every processor time slot a new instruction is chosen from a different thread that is ready and active, so that threads are switched every slot (Figure 2.1). If one of them incurs in a long latency event, it is simply not scheduled until that event completes. The key advantage of the interleaved scheme is that there is no context switch overhead. A disadvantage is that interleaving instructions from many threads slows down the execution of the individual threads (to mitigate this problem, some solutions have been proposed, [18]).

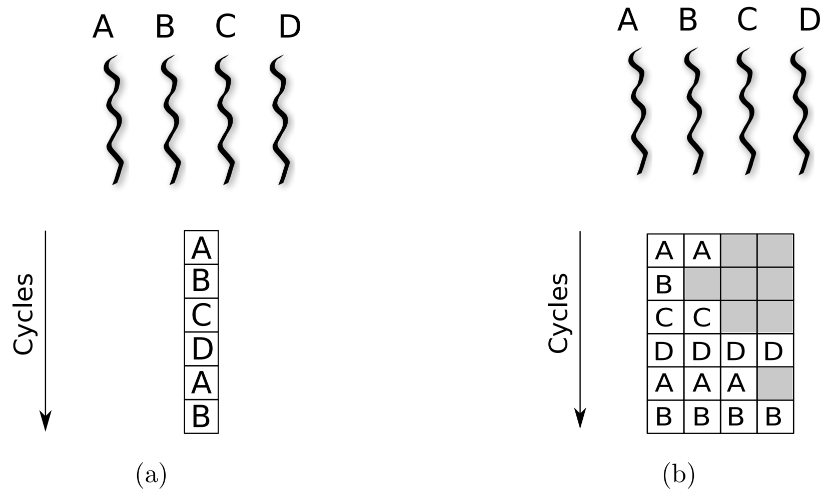

Figure 2.1: Different approaches to Interleaved Multithreading. A, B, C and D are active threads. In (a) IMT technique is applied to a scalar CPU. In (b) it is applied to a superscalar one. Gray boxes represent empty issue slots caused by dependencies in threads instructions execution.

2.1.2 Blocking Multithreading

The Blocked Multithreading (BMT) technique implies that a single thread is executed until it reaches a situation that triggers a context switch, usually a long latency operation (Figure 2.2). Compared to IMT technique, a single thread can now be executed at full speed until a context switch occurs.
The major drawback of this solution is its limited ability to overcome throughput losses when short stalls occur. Since the CPU issues instruction from a single thread, when a stall is encountered its pipeline must be emptied (or frozen) and must be filled up with the instructions of the new thread, resulting in a thread context switch of few clock cycles. Because of this start-up overhead, BMT is much more useful for reducing the penalty of high latency operations, where the cost of pipeline refill is negligible compared to the stall time.

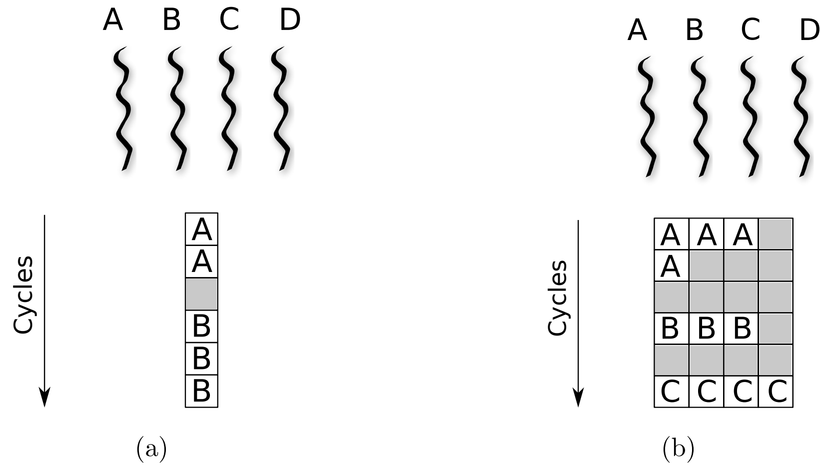

Figure 2.2: Different approaches to Blocked Multithreading, in the case of scalar CPU (a) and superscalar CPU (b)

2.1.3 Simultaneous Multithreading

As already mentioned, simple superscalar processors suffer from two inefficiencies: the limited ability to find ILP in a single program and, consequently, the stalls due to long latency operations. The Simultaneous Multithreading (SMT) approach seeks to overcome the drawback of both superscalar and interleaved/blocked multithreading techniques combining them by issuing instruction from different threads in the same time slot (Figure 2.3). In this way, latencies occurring in the execution of a single thread are bridged by issuing instruction of the remaining threads.
The key insight that motivates SMT is that modern superscalar processors often have more functional unit parallelism available than a single thread can effectively use. For these reasons, SMT seems to be the most promising approach to multithreading and, until now, the most used in research and commercial solutions.

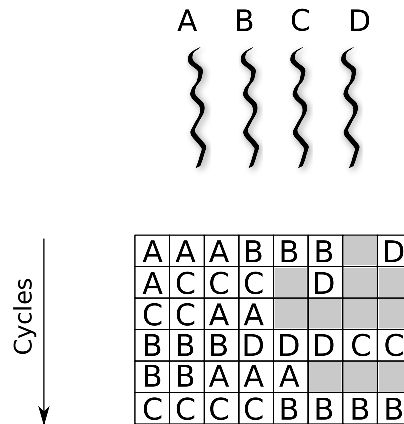

Figure 2.3: Example of Simultaneous Multithreading approach: now, instruction of multiple threads are issued in the same clock cycle

2.2 Intel's Hyper-Threading Technology

Intel's Hyper-Threading Technology (HT) [13] was firstly introduced in 2002 with Intel's Xeon and Pentium 4 processors and proposes a two-threaded SMT approach for general purpose CPUs. As already explained, this means that the firmware architecture of an HT CPU is able to maintain informations for two distinct and independent thread contexts. By doing so, HT makes a single physical processor appears as two logical processors to the operative system.

The processor's resources fall into three categories:

- *replicated*: these include general-purpose registers, the Advanced Programmable Interrupt Controller (APIC), the program counter, the instruction address relocation table (called Instruction Translation Lookaside Buffer);
- *partitioned*: include re-order buffer, load/store buffers and various queues that decouple the major stages of the pipeline from one other. By statically partitioning these resources, if a logical processor is stalled, the other one can continue to make forward progresses. These resources must be recombined in order that, if only one thread is active, it can run at full speed;

- *shared*: comprise caches (Trace Cache, which performs also the tagging of threads instructions, and data caches), data address relocation table (Data Translation Lookaside Buffer), branch predictors, control logics and execution units. Access to them is handled in a round robin fashion.

As highlighted in [16], the primary difference between the HT implementation and the SMT architectures proposed in literature regards the mode of sharing the hardware structures. In fact, SMT researches indicate that virtually all structures are more efficient when dynamically shared, rather than statically partitioned like HT does for some of them. Even if any negative effects of the partitioned approach are minimized with only two hardware contexts, it should be keep in mind that the technique employed by Intel, at least in its first implementation, will not scale well to a larger number of hardware contexts.

Regarding the benefits achieved, according to Intel, the first implementation uses only 5% more die area than a comparable non Hyperthreaded processor, but the performance is 15-30% better. Nowadays, Intel's Hyper-Technology technology is the most available commercially solution for SMT in general purpose CPUs.

2.3 Multithreading in Parallel Computing

Theoretical and experimental results show that SMT is well suited in a multiprogram context, achieving a scalability that ranges in the interval 1.4-2.4 for a number of threads in the interval 2-8 (Figure 2.4). Parallel applications usually exhibit an high exploitable degree of parallelism and SMT could seems to be a good choice also in this case. Unfortunately, the simple solution of increasing the degree of parallelism allocating more computation threads on the same processor, does not give the expected results. Contention for shared resources, indeed, limits the performance advantages of MT on current SMT processors, thus leading to marginal utilization of multiple hardware threads and even a slowdown due to multithreading.
Authors in [6] conduct a study in this sense, evaluating, in various parallel

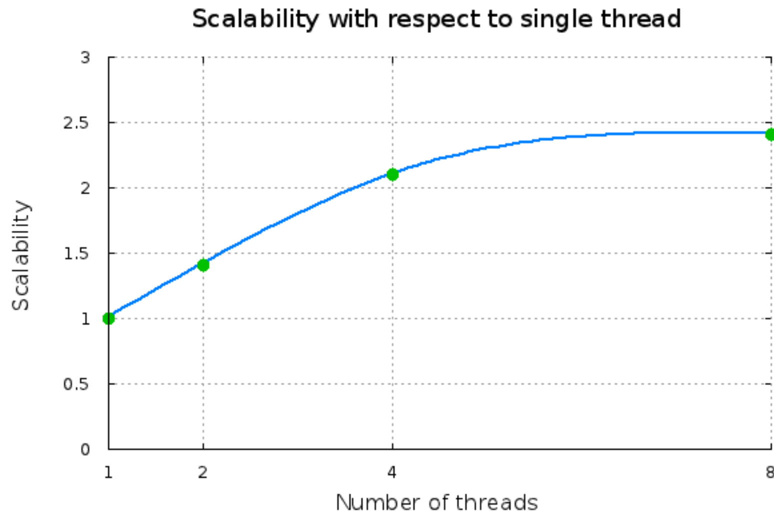

Figure 2.4: Performance evaluation of multithreading on the standard benchmark suite Spec95 and Splash2

benchmarks, the impact of exploiting Intel's Hyperthreading (as mentioned, the Intel's SMT implementation) in a Symmetric Multi Processor (SMP) system. They conclude that it is not always clear whether it is better to use one or two threads per processor. They show that doubling the degree of parallelism with respect to the traditional SMP (thus having two thread run together on a single processor) there is an average increase in performances of merely 7% and in some cases there is also a slowdown, till 30% in particular benchmarks. This is due to the fact that in this case the threads will have similar characteristics, and so, they will put pressure on the same resources that will become bottleneck. In particular the authors experienced an increasing in:

- cache misses: in an Hyperthreaded processor, L2 cache is shared between the two threads. If the working set of both co-executing threads do not fit in L2 cache then cross-thread cache-line eviction significantly increases the number of misses;
- DTLB misses: in many cases, co-executing threads work on different portions of virtual address space and, therefore, cannot share DTLB entries. This, in turn, results in an effective halving of the data TLB

area per thread and, thus, in an increased number of misses.

- stall cycles: the previous considerations and the content for execution units, increase the stall cycles rate (calculated over the total number of execution cycles), that goes up of a factor 3.

In conclusion, how to efficiently exploit multithreading in parallel applications is still unknown. In the literature, many researches suggest the use of co-operating threads to increase the performance of a single thread computation. Speculative precomputation is an example of such techniques and will be briefly discussed in the next section.
The work of this thesis aims to investigate an alternative approach to the problem, by utilizing multithreading to facilitate interprocess communication in parallel applications, by emulating a communication processor when it is not physically available.

2.3.1 Speculative Precomputation

Speculative precomputation (SPR) is a technique to improve single-threaded performance on a multithread architecture. Clearly, such a technique can be also used in the context of parallel applications. It utilizes otherwise idle hardware thread contexts to execute helper threads on behalf of the main one. These additional threads, speculatively prefetch data that are going to be used by the main computation thread in the near future, thus hiding memory latency and reducing cache misses. SPR could be thought of as a special prefetch mechanism that effectively targets load instructions that traditionally have been difficulty to handle via prefetching, such as loads that do not exhibit predictable access patterns ([4]).

In brief, implementing SPR consists of the following steps:

1. identify the so called *delinquent loads*: in many practical cases, only few loads are responsible for the vast majority of cache misses. Their identification is usually done by means of profiling tools;

2. generate code for prefetcher threads: a possible solution is to replicate the original program code and preserve only the backward slices of the target delinquent loads; all other instructions are eliminated;

3. insert synchronizations point between prefetcher and computation threads: this must be done in an accurate way, in order that prefetchers bring right data at the right time.

Results on simulated multithreaded architectures with ideal hardware support (such as multiple contexts, efficient mechanism for thread spawn, special hardware for lightweight synchronization between threads) shows that particular SPR techniques can achieve an average 70% of speedup over various benchmarks ([4]). On the other hand, experimental tests conducted on a commercial hyperthreaded platform (that does not have such ideal mechanisms) show that, on a pool of benchmark, SPR achieve speedups between 4% and 34% in half cases, while performs comparably to single threaded execution in the rest ([2]).

CHAPTER 3

Communication in parallel applications

Parallel applications can be expressed by means of *computation graphs*, whose nodes represent computational modules that cooperate in order to achieve a common goal. Assuming, without loss of generality, a *local environment* (or *message-passing*) cooperation model, the edges of the graph will represent communication channels that will be used to exchange informations between modules (Figure 3.1).

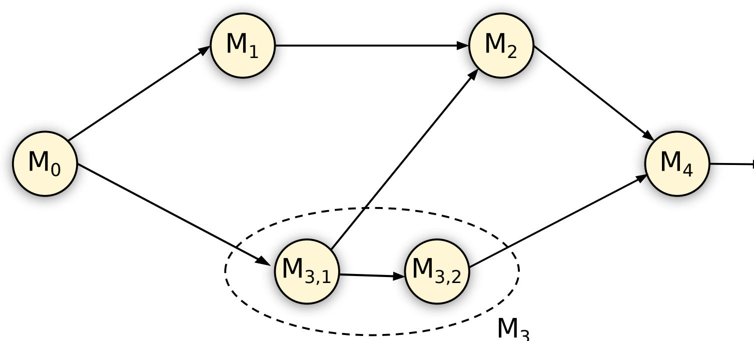

Figure 3.1: An example of computation graph: each module can be seen as a server or a client with respect to another one. For example M_1 is a servant for M_0 but a client with respect to M_2. Inter-node parallelism can be exploited to reduce the effect of bottlenecks, as in the case of M_3.

How to study and organize these systems is usually done according to well-known parallelism paradigms: they are schemes of parallel computations that recur in the realization of many real algorithms and applications. Such paradigms are characterized by formal cost models that allow the programmers of parallel applications to evaluate the performance metrics of the single modules and of their graph composition.

Parallel programs, expressed as structured computation graphs via high-level parallel programming tools, are usually compiled in a set of processes, whose cooperation is defined by means of a concurrent programming language. As reference language will be used LC, a simple concurrent language introduced, with didactic purposes, in HPC courses of the master program ([19]). For the moment, it's important to know that such a language, as any existing message-passing library, is defined according to the general semantics of Hoare's Communicating Sequential Processes (CSP). *Send* and *receive* commands are the basic interprocess communication primitives provided. As usually, their combination has the effect of transferring a message from a sender to a receiver, through a communication channel.
Channels are unidirectional, have a unique name and a type: this is the type of the messages that can be sent over them and the type of the *target variables* to which received messages can be assigned. They can be symmetric or asymmetric and are characterized by an asynchronous degree $k \geq 0$ ($k = 0$ correspond to the case of synchronous communication).
The compiler of the parallel application has several versions of the runtime support of LC and selects one of them in order to introduce optimizations (*e.g.* zero copy communication) depending on the application and/or on the underlying architecture (*e.g.* uniprocessor, shared memory multiprocessor, distributed memory multiprocessor, ...). A LC runtime support suitable for interprocess communication in shared memory systems and its implementation details will be discussed in chapter 4.

In the following section, the impact of communications in parallel applications

and the benefits of overlapping them with computation will be discussed. Then an architectural support suitable for this scope, the communication processor, is briefly described.

3.1 Overlapping communication and computation

In cost models for parallel paradigms, a fundamental metric is the interprocess communication latency L_{com}. This is intended as the mean time needed to execute a complete communication, that is the mean time between the beginning of the send execution and the copy of the message into the target variable, including synchronization and low level scheduling operations. We can define the latency for transmitting a message of L words as:

$$L_{com}(L) = T_{send}(L) + T_{receive}(L)$$

where T_{send} and $T_{receive}$ are the latencies of the respective operations. In the case of zero copy runtime support, the message is copied directly into the target variables of the receiver, without intermediary copies. Hence, the latency of a receive operation is negligible with respect to the send one, since the receiver runtime support does not perform message copies. Supposing that this optimization is used, the communication latency can be rewritten as:

$$L_{com}(L) = T_{send}(L) = T_{setup} + LT_{trasm}$$

where:

- T_{setup} is the average latency of all the actions that are independent of the message length: synchronization, manipulation of structures of the runtime support, low level scheduling;
- T_{trasm} is the latency to copy one word of the message.

Both of them depend on the concrete architecture and runtime support implementation. Typical order of magnitude for shared memory architectures are $T_{setup} = 10^3 \div 10^4\ \tau$ and $T_{trasm} = 10^2 \div 10^3\ \tau$.

Can parallel applications benefit of overlapping communication and computation? Consider a generic parallel application and, in particular, one of its modules that operates on stream. Suppose that this module during its life cycle alternates computation and communication phases: this is a typical situation in many parallel paradigms that operate on streams. Typically, these two phases follow each other sequentially (Figure 3.2). In this case the service

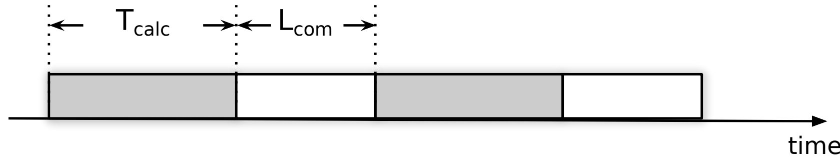

Figure 3.2: Non overlapping case: gray boxes identify calculus phases, white boxes communication ones. In such a situation no overlapping between calculus and communication is present

time of this module will be increased by the communication latency because it is entirely paid, resulting in:

$$T = T_{calc} + L_{com}$$

Consider now the case in which proper architectural supports exist (*i.e.* a communication processor) to which the execution of an asynchronous send could be delegated. In this case the module, once the primitive is offloaded to the communication processor, can start the following calculus phase without waiting; in the meanwhile and in parallel the communication is performed. In general this allow to hide (at least in part) the communication latency (Figure 3.3). Denoting with T_{com} the average communication time not overlapped with internal communication, the module service time can be written as

$$T = T_{calc} + T_{com} = max(T_{calc}, L_{com})$$

resulting in a clear advantage with respect to the previous case if L_{com} is not negligible with respect to T_{calc}. The module latency, instead, is always affected by the communication latency.

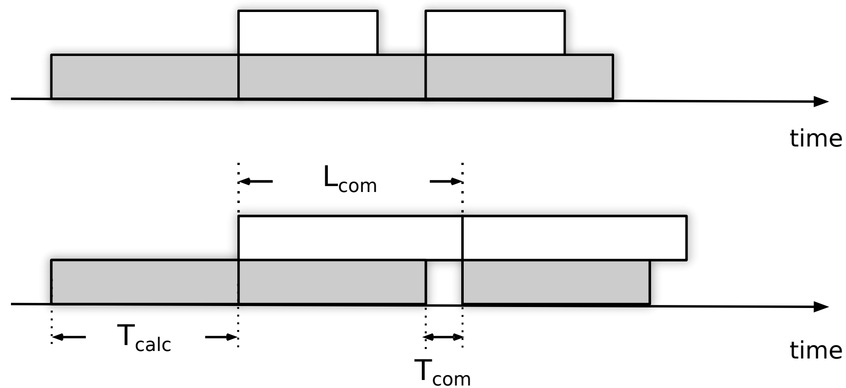

Figure 3.3: In this situation there is the possibility to overlap communications and computation. In the first case, the communication latency is fully masked by the calculation one ($T_{calc} \geq L_{com}$). In the second case, the most general one, calculation is only partially overlapped to communication

3.2 Communication processor

Shared memory system are *MIMD* (Multiple Instruction Stream Multiple Data Stream) general purpose architectures, characterized by a certain number of processing nodes (n, typically homogeneous), that share the main memory physical space and are connected each other and with the main memory by an interconnection structure (Figure 3.4).

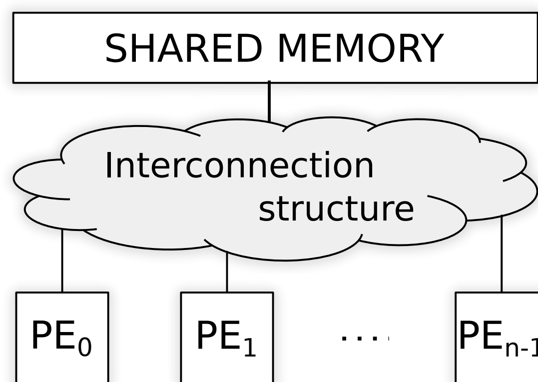

Figure 3.4: Structure of a shared memory system: memory and processing nodes are connected each other via an interconnection structure, usually of limited degree

The general structure of a processor node is depicted in Figure 3.5. This is

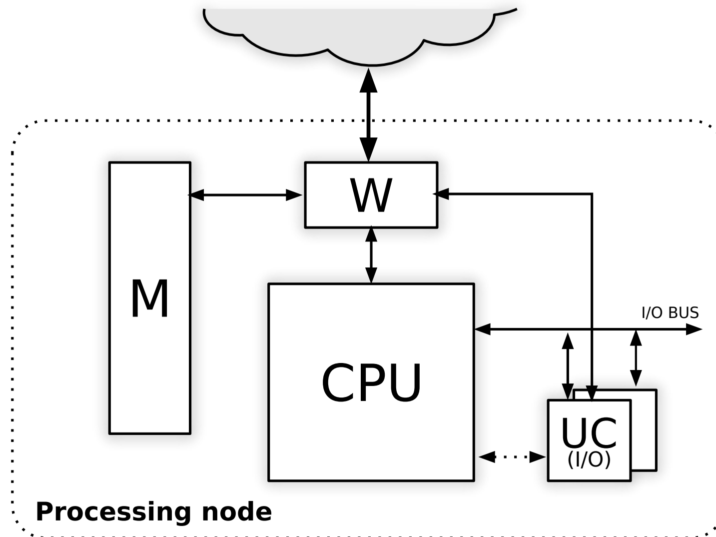

Figure 3.5: The internal structure of a processing node

characterized by:

- CPU: processing nodes are usually based on general-purpose, commercial *off-the-shelf* CPUs or computers. This is true also for *multicore* (or Chip Multi Processor, CMP) architectures, which often integrate existing CPUs into the same chip. An off-the-shelf CPU chip is connected to the "rest of the world" through its primitive external interfaces (*e.g.* Memory Interface, I/O Bus). Such CPUs should be, in any case, suitable to be immersed into a multiprocessor system: for example, should be capable to generate proper physical addresses for addressing the whole shared memory space;

- local memory and I/O units;

- node interface unit (W): this unit is in charge of interfacing the processing node with the rest of the system. In this way, existing CPU can be used thanks to the fact that W masks to the CPU the structure of the shared memory and interconnection network;

- communication unit (UC): it is provided for direct interprocessor communications support, usually performed for processor synchronization (involved in locking mechanism) and process low-level scheduling.

As already said, to allow overlapping between computation and communication there is the need of proper architectural supports, such as a *communication processor*. The structure of a processing node could evolve as illustrated in Figure 3.6.

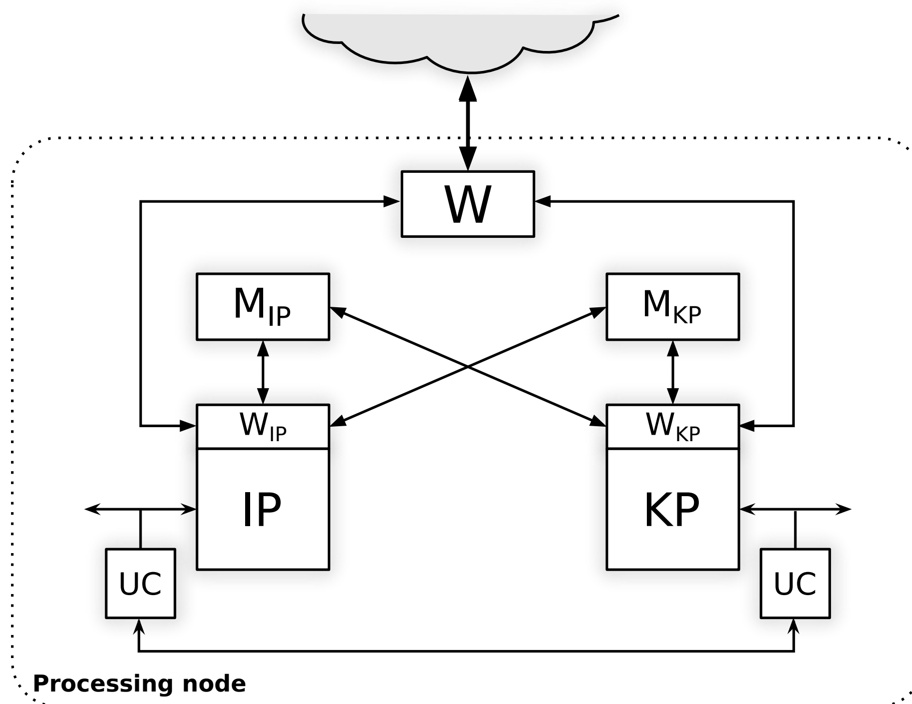

Figure 3.6: The internal structure of a processing node with a communication processor: processing node itself can now be considered as a shared memory multiprocessor

Every node has now two processors: the communication processor (KP) and the main one (IP) with their respective local memories and I/O units. KP is dedicated, or specialized, to the execution of the runtime support of LC and, in particular, of the send primitive. The delegation of the send is performed by passing to KP the information required to execute the primitive (typically the identification of the channel and a reference to the message): these informations are contained into a data structure prepared by IP, whose reference is

passed to KP via I/O (*i.e.* via a communication UC-UC).
Usually, not the entire execution of the send is delegated to KP, but some run-time support related functionalities are still performed by IP. As an example consider the send semantics: when the message is copied, if the asynchrony degree becomes saturated the sender process must be suspended. If IP delegates the send execution entirely to KP, then some complications are introduced in the send implementation because of the management of the waiting state of the sender process. A simple solution to the problem is that IP itself verifies the asynchrony degree of the channel, after the send delegation (clearly this control will not be performed again by KP), suspending the sender process if necessary. The initial phase executed by IP is now not overlapped to the internal calculation. In the interprocess communication cost model, the latency

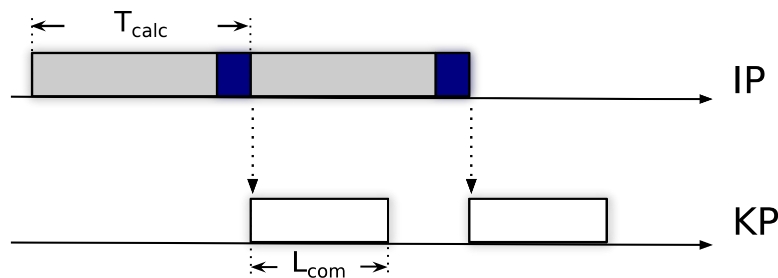

Figure 3.7: Send delegation from IP to KP; blue boxes represent part of the send runtime support executed by IP

of this phase must be included in the T_{calc} parameter (Figure 3.7).

3.2.1 Commercial implementation of communication processors

A concrete example of application of the communication processor can be found in the Intel Paragon XP/S, first shipped in the 1992 ([5]). Even if it is a distributed memory architecture, the same idea till now discussed is applied. In the Intel Paragon systems up to 2048 processing nodes are connected in a 2D mesh: each of them is a shared memory multiprocessor, with two Intel i860XP RISC processors, 16-32 Megabytes of local memory and a Network Interface Card. One of the processors is designated as communication processor and

handle the message-passing primitives, while the other is used as a compute processor for general computing, exactly as illustrated before.

In general, although it is possible to realize KP with the same architecture of IP, the recent trend is to design it as an input-output coprocessor, specialized at the firmware level, sharing memory with IP through DMA and Memory Mapped I/O. In particular, this kind of implementation is adopted when the communication processor is provided inside the interconnection network box. This is the case of specialized networks for HPC, such as Quadrics, Myrinet and Infiniband.
Taking into account the case of Myrinet ([3]), its Myricom/PCI network interface card (Figure 3.8) is characterized by a a communication processor, a local memory and two DMA engines responsible, respectively, for data transfer from host memory to local memory and from local memory to network. The

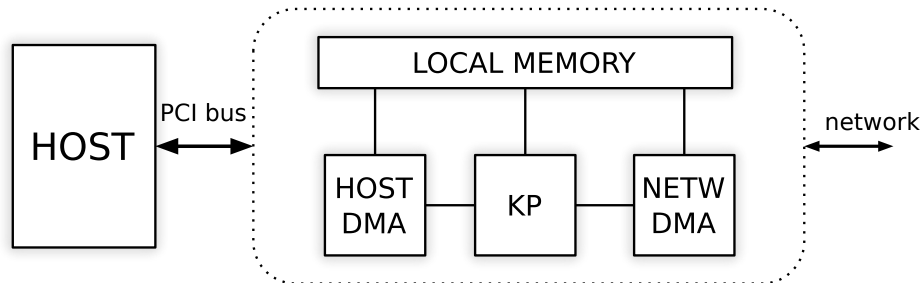

Figure 3.8: Basic structure of the Myricom/PCI network interface card

former is responsible for the execution of message-passing protocol once proper delegation is performed by the host node.

CHAPTER 4

Multithreaded support to interprocess communication

In the previous chapter has been discussed how the parallel applications can benefit of communication-computation overlap. At this point one wants to understand if and how it is possible, in a multithreaded and shared memory architecture, emulate the facility of a communication processor when it is not physically available. This chapter deals with this problem, implementing a minimal core of the concurrent language LC that provides this feature. The approach adopted is to associate to each process/thread of the parallel application (from now on will be referred as *worker thread*) a thread that is in charge of executing the send primitive (*communicator thread*) following a philosophy similar to the IP-KP case.
Although the goal is quite circumscribed, many of the issues that will be addressed are common in approaching this kind of problem, independently from the concurrent language used.

The chapter is structured in the following way:

1. *programming in multithreaded architectures*: measure to be adopted in programming, for efficiently exploit these kinds of systems, are briefly

discussed;

2. *inter-thread synchronization*: the problem of how efficiently synchronizing two threads, with particular attention to the specific work context, is approached;

3. *LC runtime support*: the runtime support of a minimal core of the concurrent language in shared memory architectures is introduced;

4. *communicator implementation*: finally, how to extend the traditional support of LC for emulating a communication processor is described.

Through the rest of the thesis, as reference architecture will be considered a shared memory multiprocessor with Hyper-threading technology, whose structure is shown in Figure 4.1. It is composed of 2 CPUs Intel Xeon E5520

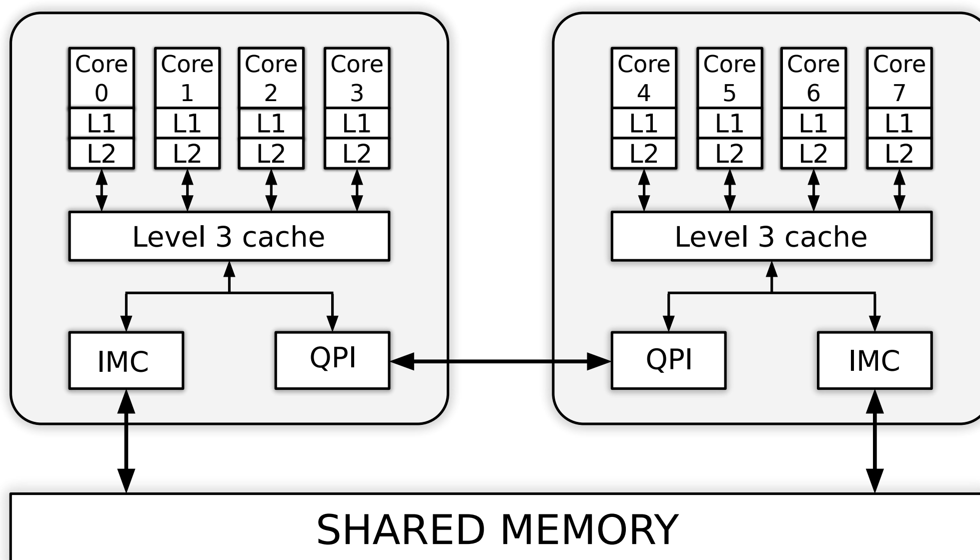

Figure 4.1: Internal structure of the reference architecture: the *Integrated Memory Controller* (IMC) and *Quick Path Interconnect* (QPI) constitute the interconnection structure of the system

(running at 2.27 GHz), each of them with four physical cores supporting the Hyper-threading technology (2-way Simultaneous Multithreading). Level one and level two cache (respectively of 32 and 256 Kb) are private for each core,

while the level three cache (8 Mb) is shared by all the four cores of a CPU. The machine is equipped with 12 Gb of RAM and runs a GNU/Linux operating system, based on the Linux Kernel 3.0.

4.1 Programming in multithreaded architectures

In the case of a multithreaded architecture, a thread is viewed as a firmware level supported thread which can be a single process, a user defined thread, a compiler-generated thread such as micro or nanothreads.
Since the reference environment is a Linux-based operating system, the obvious solution seemed to exploit multithreaded architecture through used-defined threads created with the features offered by the Native POSIX Thread Library (NPTL, compliant with POSIX standards). With the POSIX Threads, a process can have multiple threads each of which has its own flow of control and its own stack. Everything else about the process, including global data, heap and resources, is shared.

Even if the methodologies applied are the usual of multithreaded programming, here the application programmer (or, as in this particular case, the runtime support designer) should be well aware that he/she is working on a multithreaded architecture and how it is organized.
As explained in sect. 2.2, Hyper-threading technology makes the single two-threaded physical processor/core appears as two logical processors (referred as *siblings processors*), each of them identified by a unique ID. HT-aware Linux schedulers are clever enough to manage this situation efficiently, for example trying to schedule threads/processes on sibling processors only if it is strictly necessary, taking into account that the majority of the resources is partitioned or shared between them. Nevertheless, in programming the runtime support of LC with communication processor facilities there are more stringent requirements, for example assuring that the worker and communicator threads run on two sibling processors to better exploit the cache (since the low levels are shared between sibling processors). This is usually done by providing directives

to the scheduler regarding the binding thread-logical processor, the so called *processor affinity*. NPTL provides proper function for this purpose, such as the `pthread_setaffinity_np()`.
The only thing that the support user must know is the processors topology, that indicates the association between logical and physical processors. In Linux systems such informations are published through the `sys` pseudo file system ([7]). For the testing architecture, this association is sketched in Figure 4.2

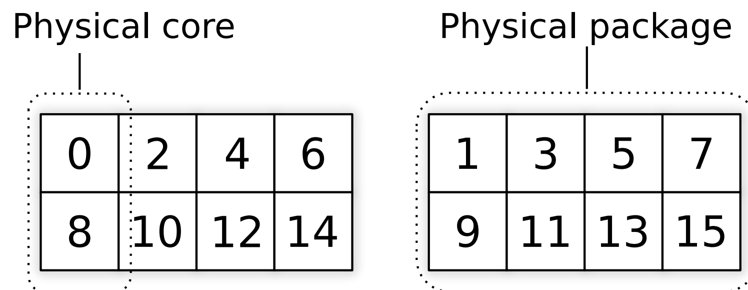

Figure 4.2: Processors topology of the reference architecture: there are two physical CPU (also called packages) each of which has four physical cores. The numbers represent the ID of a logical processor associated by the operative system; ID on the same column identify sibling logical processors

4.2 Inter-thread synchronization mechanisms

In the working scenario, the communicator thread has to wait, even for a large period of time, that a send is delegated to it by the worker thread; therefore, there is the need for inter-thread synchronization mechanisms provided in the form of classical `wait/signal` primitives. Considering the asymmetric workload that characterizes these two threads and, especially, the context of an Hyper-threaded machine where many of the computational resources are dynamically shared or statically partitioned, such mechanisms must have properties of high responsiveness and lightweightness.
In the literature, proper hardware extensions have been proposed to meet these requirement (consider as an example the *lock-box* introduced in [17]), while many brand new commercial products (such as the IBM wirespeed processor)

provide appropriate synchronization primitives.
An Hyper-threaded processor does not provide such mechanisms for low-latency and resource-friendly synchronization directly at the firmware level.

In the following, classical method (POSIX mechanisms and spin locks) and a new one (proposed in [1] and based on proper Intel's assembler instructions) for synchronizing two threads are discussed. In conclusion, an evaluation of such mechanisms is presented. Except where indicated otherwise, the code that will be shown is written in a *C-like* pseudocode.

4.2.1 POSIX

POSIX standard provides well-known mechanisms for synchronizing threads. To signal and wait for a condition, three things are necessary: a mutex M, a condition variable C and a predicate P. The first two are proper data structure offered by POSIX Threads; the predicate depends from the particular context of application. Primitives that work on condition variables must be called in critical sections, as shown in the example.

```
Thread A:

    mutex_lock(M);
    <...>
    cond_signal(C);
    mutex_unlock(M);
```

```
Thread B:

    mutex_lock(M);
    while(!P)
        cond_wait(C,M);
    <...>
    mutex_unlock(M);
```

The call at **cond_wait()** do not imply busy-waiting: it actually releases the associated mutex, causes the thread to be descheduled (passing in waiting state) and, assuming that there is no other runnable thread, the processor resources previously occupied are released in favour of the worker thread.

Latest versions of the NPTL Library makes use of *futexes* (Fast User muTEX), a mechanism provided by the Linux Kernel (2.6 or greater) as building block

for fast userspace locking. In this case, multiple threads communicate locking state through shared memory regions and atomic operations; a futex based lock works in user space unless there is a lock contention ([10]). In this case, system calls are required to wake up a process or to put another one into waiting state. Even if this solution could save many user-kernel-user space switches compared with old implementations, it should be considered that some overhead is unavoidable due to the presence of system calls.

4.2.2 Spin lock

Synchronization primitives based on spin lock have been commonplace in traditional multiprocessor systems, due to their simple implementation and high responsiveness; unlike to the previous solution, they imply busy waiting. In the case of spin lock, the lock variable can be a simple boolean that indicates whether the lock is held by someone. Each processor repeatedly use an atomic `test-and-set` or `exchange` instruction in attempt to change the flag from false to true, thereby acquiring the lock. A processor releases the lock by setting it to false.
The principal shortcoming of this first version is contention for the flag: each waiting processor accesses the single shared flag as frequently as possible using the indicated instructions. This can be particularly expensive on cache-coherent processors, since each execution of such instructions can cause a cache line invalidation (resulting in a ping-pong effect). A simple modification to the presented algorithm is to use a `test-and-set` instruction only when a previous read indicates that this can succeed.

In modern out-of-order processors a spin loop is dynamically unrolled multiple times since there are no data dependencies and the branch that it contains can be easily predicted. In an Hyper-threaded environment, a synchronization primitive based on spin locks can incurs significant performance penalty since the spinning thread, even though not performing any useful work, inserts a significant number of instructions that compete with other threads executing

on the sibling logical processor (the worker thread in this context) causing a slow-down of both of them.
Intel recommends the use of `PAUSE` instruction in spin loops ([12]): these instruction introduces a slight delay in the loop and de-pipelines its execution, preventing it from aggressively consume processor resources. The cost of spinning, however, is reduced but not entirely eliminated, because the resources designated to be statically partitioned are not released by the spinning thread. Considering the use of the `PAUSE` instruction, a simple spin lock, written in assembler language, is the one reported in Listing 4.1.

```
SLOCK:      mov [LV],eax
            cmp 0,eax
            je GET_LOCK
            pause
            goto SLOCK
GET_LOCK:   mov 1,eax
            xchg [LV],eax
            cmp 0,eax
            jne SLOCK
```

Listing 4.1: Spin lock assembler implementation using the pause instruction. *LV* represents the address of the locking flag

Spin locks can constitute a building block for simple synchronization primitives. Spin lock itself can be used as primitive for the access in critical sections while a condition variable can be implemented via a boolean flag. Maintaining a style similar to the POSIX standards, the signalling and waiting procedures must be called into a critical section. The former simply set the flag; the latter first of all resets the flag and then, cyclically, check it values, releasing the lock if founded still unset. The resulting wait primitive is shown in Listing 4.2.

```
bool notified=false;
c=UNSYNCHR_VALUE; //reset the flag
spin_unlock(l); //free the lock variable
do
{
    spin_lock(l); //lock
    if(c==SYNCHR_VALUE) //the partner notified
```

```
        notified=true;
    else
        spin_unlock(l); //unlock and loop
} while(!notified);
```

Listing 4.2: Wait primitive: c represents the condition variable. It must be called into a critical section

Clearly, the behaviour is slightly different compared to the POSIX case:

- a condition variable can be used to synchronize entities only in a "one-way" fashion. This means that if it used by thread A to signal a condition to thread B then it cannot be used to synchronize these entities in the opposite way, otherwise dead-locks can occur;
- since wait and locking primitives require busy waiting, no fairness is guaranteed to the contenders. Thereby, it is better to use this solution to synchronize only two entities (like in the study case).

4.2.3 `MONITOR/MWAIT` instructions

`MONITOR` and `MWAIT` are two assembler instructions introduced with the Intel's Prescott architecture. The `MONITOR` instruction sets up a memory address that is monitored for write activities. `MWAIT`, instead, places the calling processor in a "performance-optimized" state (which may vary between different implementations) until a write to the monitored region occurs [12]. `MWAIT` still causes a busy waiting but behaves as a `NOP` and execution continues at the next instruction in the execution stream. Moreover, on a Hyper-threaded processor, a thread that calls `MWAIT` causes its logical processor to relinquish all its shared and partitioned resources.
The address provided to `MONITOR` instruction effectively defines an address range, within which a store will cause the thread blocked to the `MWAIT` to wake up. Thus, to avoid *missed wake ups*, the data structure that will be used as condition variable must fit within the *smallest monitor line size* and must be

properly aligned so that it does not cross this boundary. Otherwise, the processor may not wake up after a write intended to trigger an exit from `MWAIT`. Similarly, write operations not intended to cause an exit from the optimized state should not write to any location within the monitored range. In order to avoid *false wake ups* the data structure should be padded to the *largest monitor line size*. Both of this parameters (smallest and larger monitor line size) are architecture dependent and should be properly checked; in the reference system are equal to 64 bytes.
Furthermore, multiple events, other than a write to the triggering address range, can cause a processor that executed `MWAIT` to wake up, such as interrupts or context switches. This means that, after wake up, the value stored in the triggering address should be checked and, if necessary, return in a waiting state: therefore `MONITOR/MWAIT` need to be executed in a loop.

In their initial implementation, `MONITOR` and `MWAIT` instructions are allowed to be executed only in kernel space, with the maximum privilege level[1]. Therefore, in order to be able to implement synchronization primitives based on these two instructions, there are two possible solutions:

- extend the Linux Kernel with a pair of system calls through which access to these instructions is granted;

- run the program that use them directly at kernel level.

The second option has been chosen as best representing a probable future situation where these instructions are released by Intel to be executed in user level, avoiding the overhead that the system call will have introduced. To run the entire programs in kernel mode, a proper patch of the Linux Kernel will be used (*Kernel Mode Linux*, [15]).

---

[1] Actually, during the study of this instructions, some machines were found in the computer center facility of the university, based on Pentium 4 HT (the first commercially available processor with Hyper-threading technology) that can run this instruction also at user level

As before, simple synchronization primitives based on these instructions are provided. Access to critical sections is still granted through spin locks. The condition variable is now a vector of 64 bytes that has to be properly aligned on memory. The sketch of the wait primitive, that addresses all the previously stated problem, is shown in Listing 4.3.

```
bool notified=false;
c=UNSYNCHR_VALUE;
do //monitor/mwait cycle
{
    _mm_monitor(&c,0,0); //set the monitor region
    spin_unlock(l);
    _mm_mwait(0,0); //wait for a store on it
    spin_lock(l); //access to the condition variable and check
    if(c==SYNCHR_VALUE) //the partner notified
        notified=true;
    //otherwise loop
}while(!notified);
```

Listing 4.3: Wait primitive based on MONITOR/MWAIT instructions: c represent the condition variable. _mm_monitor and _mm_mwait are the GCC intrinsics counterparts of the MONITOR/MWAIT instructions

The considerations done for the primitives based on spin lock, are valid also for this case.

4.2.4 Evaluation of the synchronization mechanisms

For comparing the illustrated synchronization mechanisms a proper test scenario has been set up. There are two threads allocated on two sibling processors: the first one (*worker thread*) performs, cyclically, heavy calculus in one case and, in a second one, make also random accesses to a data structure in order to generate a large amount of cache faults. The second (*waiting thread*) just waits until notified by the former each time that a calculus cycle is completed.
The two tests were performed multiple times for each synchronization primi-

tive.
The gathered metrics of interest are:

- T: the completion time of the worker thread: the larger this time is, the more disturbing is the co-existence of the waiting thread on the physical processor;
- T_{call}: the time (measured in processor cycles) that the worker thread spends in invoking the notification primitive;
- T_{wakeup}: the time (measured in processor cycles) between the notification to the waiting thread and the moment that it is actually awakened.

Results of the two tests ran on the reference architecture are shown in Table 4.1 and Table 4.2.

Primitive	$T(msec)$	$T_{call}(cycles)$	$T_{wake}(cycles)$
Spinlock	4420.5	193	318
`MONITOR/MWAIT`	2934.12	131	815
Posix	2861.5	3918	162094

Table 4.1: Test case with only calculus

Primitive	$T(msec)$	$T_{call}(cycles)$	$T_{wake}(cycles)$
Spinlock	37881.101	602	779
`MONITOR/MWAIT`	36721.699	719	1701
Posix	35472.199	9640	72154

Table 4.2: Test case with calculus and cache faults

As expected, spin lock based primitives provide minimum response time and call overhead (comparable with the one exposed by `MONITOR/MWAIT`) but they are the most aggressive in term of resource consumption. In the first test case, on average the spinning thread decelerates the worker thread of more than 50% compared with the other two solutions. This difference decreases in the second test case, resulting in a slowdown of only 4%: this is probably due to

the fact that the latencies introduced by cache faults allow to amortize the additional workload caused by the waiting thread. POSIX based primitives present the worst wakeup and call times due to the passage at kernel level and rescheduling of waiting thread. However they present a completion time that is less compared to the one obtained with `MONITOR/MWAIT` primitives, that perform on average 3% worse. This indicates that the "performance-optimized" state of `MWAIT` still introduces some impairments on the sibling thread.
As stated in Intel's documentation, this state is implementation dependent. In fact, the same tests run on a newer Intel's processor compared to the reference one (Intel Core i7-2677M dual core, running at 1.80GHz, 32 Kb for L1 cache and 256Kb for L2 cache, same operative system of the reference architecture) show that the difference between the completion time obtained with the POSIX mechanisms with respect to the one resulting by the usage of `MONITOR/MWAIT` based primitives decreases of one order of magnitude (Table 4.3 and Table 4.4).

Primitive	$T(msec)$	$T_{call}(cycles)$	$T_{wake}(cycles)$
Spinlock	3662.8	186	261
`MONITOR/MWAIT`	2381.6	62	615
Posix	2358.899	1421	69803

Table 4.3: Test case with only calculus

Primitive	$T(msec)$	$T_{call}(cycles)$	$T_{wake}(cycles)$
Spinlock	29656.099	658	744
`MONITOR/MWAIT`	28613.599	561	1327
Posix	28525.3	8452	96752

Table 4.4: Test case with calculus and cache faults

Hopefully this gap will become smaller and smaller as new architectures refine these instructions. Regarding the false wake ups caused by event external to the computation, it has been noticed that they are exclusively due to the interrupts send by the system timer to the processor for indicating that the time quantum is expired; in any case, this don't impact on the completion time.

In conclusion, it is clear that the spin lock based synchronization mechanisms are not suitable for synchronizing two threads running on the two sibling processors. On the other hand, a single winner from the comparison between POSIX mechanisms and `MONITOR/MWAIT` based synchronization primitives can not be determined both for the simplicity of the test cases considered (which are, in any case, extreme situations) and, as pointed out, because these instructions seem to be subject to potential future improvements. For this reason, the support user will be allowed to chose which one use. In chapter 5, tests will be performed with both type of primitives. Moreover, given the generality of this mechanisms, they will not used only in the case of inter-thread synchronization but also to regulate the access to the runtime support data structures.

4.3 Runtime support of LC

The minimal core of LC that will be implemented refers to the case of symmetric, synchronous or asynchronous, deterministic channel (*i.e.* not referred in alternative commands) for shared memory architectures, with zero copy optimization. The interface offered to the user is inherited by a previous work of the research group ([8]) and it is maintained for compatibility reasons. Since in the case of communication processor the interest is focused on asynchronous communications, the runtime support for this kind of communication is now discussed. The synchronous case is obtained with some minor modifications.

4.3.1 Zero copy optimization and shared memory segment

The zero copy technique allows the sender to copy a message directly into the target variable of the receiver without copies in intermediary data structures. The number of copies is just one, *i.e.* the minimum independently of the receiver progress state (zero copy stands for *zero additional copy*). This means that the processes involved into the communication must share part of their virtual memory, in particular the target variables are shared objects them-

selves.
Moreover, one objective of this work is to design the runtime library in a proper way to be executed in the user space. In this way the additional overhead caused by the execution of a language primitive in supervisor (or kernel) state is saved.

Unix-based operative systems offer a basic mechanism for sharing memory between different processes: the *XSI Shared Memory Segment*. A segment is created by one process and, subsequently, written and read by any number of processes that acquire it in their address spaces. This means that the logical addresses, generated by the different processes that operate on the segment, will be relocated to the same physical addresses. Once that the process finishes to work with it, it is detatched from its virtual memory and, when none of the processes is willing to use it, it is finally marked to be destroyed. The memory segment is referred (hence shared and acquired) by a key that unequivocally identifies it in the system.
A peculiar characteristic, that will influence some implementation choices, is that a process can not dynamically share, via an XSI memory segment, an already allocated portion of its virtual memory. In the case of the runtime support for LC, this implies that a receiver can not dynamically share the target variables. These must be created at channel creation time by allocating the necessary space into a memory segment that will be shared by sender and receiver.

4.3.2 Channel descriptor

The basic data structure of the runtime support is the channel descriptor that will contains all the informations necessary for an efficient implementation of LC functionalities. Along with the target variables, this structure is shared by sender and receiver. For consistency reasons, the procedures that manipulate it must be executed in an indivisible way. This requirement is achieved by means of locking primitives, such as the ones previously discussed. Moreover

the communication primitives could require some synchronization between the involved entities. In particular:

- when executing a receive, if the communication channel does not contains any message, the receiver must wait until a send is executed;
- when executing a send, if the communication channel is full (the asynchrony degree has been reached), the sender must wait until a receive is executed.

Depending of what type of synchronization primitive is used, these situations could imply busy waiting (such as in the case of `MONITOR/MWAIT`) or not (POSIX mechanisms).

The type of the channel is specified at creation time, therefore the dimension of the target variables is known and fixed. If the asynchrony degree is equal to K, $K+1$ target variables, handled as a circular vector, will be created and shared between sender and receiver. The solution $N = K+1$ avoids that the sender process re-executes the send primitive once that the asynchrony degree is reached: it completes the actual send operation and then is suspended. In this way, once waked-up, it is resumed at the return logical address of the send procedure.
By the definition of the zero copy method, the receiver process works directly on the target variable instances (without copying them). Therefore, a critical race could occurs when the sender tries to copy a message into a target variable and the receiver is still utilizing it. To overcome this problem, a simple solution is to allocate J additional target variables. J is chosen at channel creation time ($J \geq 0$) and this results in having a total of $N = K + J + 1$ target variables; in many practical cases, J is equal to 1. The send procedure will copy the messages in this N variables, handling them as a circular buffer, respecting the asynchrony degree constraint. On the other hand, the receiver can execute at most J receive and, simultaneously, use the respective target variables, being sure that the execution of a send will not overwrite their content.
An alternative solution to this problem could be to associate a *validity bit* to

each target variable that indicates whether or not it is safe to overwrite it. Thus the sender process will now be suspended also if the validity bit of the target variable to which this message must be copied is set to false. On the receiver side a proper primitive (*e.g.* set_validity_bit()) must be provided in order to allow the receiver process to indicate that it is no more willing to utilize a specific target variable.
Even if a validity bit mechanism (with a slightly different semantics) has to be included in the channel descriptor for consistency reasons that will be clear in the next section, the solution of the additional J target variables has been chosen, mainly for reasons of backward compatibility.

Taking into account these considerations, in the case of asynchronous channel (the synchronous case is simply derived from this one) the channel descriptor data structure contains:

- *lock variable*: used to assure the exclusive access to the descriptor; it will be used by all the procedures that intend to manipulate the data structure;
- *key*: the identifier of the shared memory segment that contains this channel descriptor;
- *condition variable*: used for synchronizing sender and receiver processes; if necessary, as in the case of `MONITOR/MWAIT` based synchronization primitives, two condition variables are present;
- *size*: the dimension (bytes) of the messages transferred through this channel;
- *messages*: the number of messages currently present into the channel;
- K, J: respectively the asynchrony degree and the number of additional target variables;
- *buffer infos*: insertion and extraction pointer for the circular vector of target variables; they assume relative values, that is between 0 and N,

indicating a position within the buffer.

As can be seen, for implementation reasons, the vector of target variables and the vector of the respective validity bits are kept outside the channel descriptor. However they are allocated in the same memory segment of the channel descriptor, contiguously to it. In this way, only one memory segment per channel is necessary and, each time that an operation on the buffer (and, consequently, on the validity bits) must be performed, the address of the interested target variable can be easily computed knowing the value of the current insertion/extraction pointers, the size of the channel descriptor data structure and the base address of the memory segment. Figure 4.3 shows the internal organization of a memory segment.

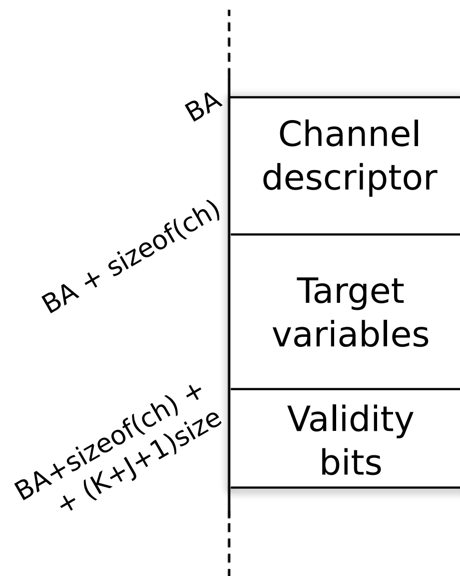

Figure 4.3: Shared memory segment organization: supposing that it is allocated into the virtual memory of a process starting from a base address BA, the addresses of all the interesting structures can be easily computed

4.3.3 Send and receive primitives

At this point, a sketch of the send and receive primitives implementation, for the asynchronous case, can be given. Clearly, some modification to them will be necessary when the communicator thread will be introduced in the runtime support. For the moment, the validity bit mechanism is not used.

The send procedure (Listing 4.4) is totally executed in mutual exclusion by using the lock variable present in the channel descriptor.

```
void send(Channel ch, char *msg )
{
   lockChannel( ch );
   //copy the message in the next target variable.
   memcpy( VTG(ch,ch->insVtg), msg, ch->size );

   //update the number of messages and targ. variables vector
      infos
   ch->messages ++;
   ch->insVtg = ( ch->insVtg + 1 ) % ( ch->k + ch->j +1 );

   //signal to the receiver
   signal(&(ch->cond_toreceiver));

   //if the buffer is full wait
   while( ch->messages > ch->k )
      wait(&(ch->cond_tosender), &(ch->lockVar));

   unlockChannel( ch );
}
```

Listing 4.4: Send primitive

First of all the message is copied into the proper target variable: its address is computed by using the macro VTG that takes as parameter the base address of the channel and the pointer of the target variable (its position within the variables buffer) and compute its address:

```
#define VTG(ch,i) ( (char*) ch + sizeof(*ch) + ch->size*i )
```

Then the number of messages and the insertion pointer are updated according to the circular handling of the buffer. A `signal` synchronization primitive is performed to notify to the receiver, that is possibly waiting, that a new message is present in the channel and, finally, if the buffer is full, the sender process have to wait (will be suspended or will enter in a busy waiting state).
On the other hand, the receive code is shown in Listing 4.5; as expected, it is a very lightweight operation.

```
char *receive(Channel ch)
{
   lockChannel( ch );

   //wait if buffer is empty
   while( ch->messages == 0 )
      waitNotify(&(ch->cond_toreceiver), &(ch->lockVar));

   //get the pointer to the message
   char *vtg = VTG(ch, ch->extrVtg);

   //update infos
   ch->extrVtg = ( ch->extrVtg + 1 ) % (ch->k + ch->j+1);
   ch->messages --;

   //signal to the sender
   signal(&(ch->cond_tosender));

   unlockChannel( ch );
   return vtg;
}
```

Listing 4.5: Receive primitive

If no message is present into the channel, the receiver will wait upon sender notification. Otherwise, the pointer to the target variable is computed and will be returned to the caller. The information are properly updated and a notification is sent to the sender for indicating that a slot was released in the target variables buffer.

4.4 Communicator implementation

The runtime support of LC will be modified and extended in order to allow the emulation of the facilities of a communication processor through a multi-threaded architecture.
As seen in section 3.2 the IP processor does not delegate the entire send execution to the KP, but the buffer full control is executed by itself. The same consideration hold also in the case of worker and communication thread:

```
Worker Thread:
Send:
   <delegate send>
   if(buffer full)
      wait();
```

```
Communicator Thread:

   wait_delegation();
   <execute rest of the send>
   <loop>
```

A set of problems have to be addressed:

- how to synchronize the two threads: the communicator thread has to wait until a delegation arrives from the worker one. Independently of what mechanism will be used, from now on the existence of the well known `wait` and `signal` primitive is assumed;
- how consistently check whether the buffer is full or not;
- the worker, when delegating a send, has to pass to the communicator a reference to the channel and to the message to be sent. Furthermore, more than a send (to different channels) could be delegated without waiting that the previous one is terminated. A data structure, designed specifically for this purpose, must be shared by these two entities;
- the worker thread can overwrite a message subject of a send, only when it is safe to do it *i.e.* the message has been copied into the target variable. This means that the completion of a send must be properly notified to the worker.

4.4.1 Communicator initialization

Assuming that each thread of the parallel application is bound to a unique logical processor (usually this is the case), when initializing a communicator a new thread is created and scheduled on the sibling processor of the worker's one. The creation procedure is invoked by the worker itself; in this way the two threads will share all the informations necessary, such as messages to be sent and channel descriptors. Moreover, being allocated on the same physical core, the shared cache can be fully exploited. The creation procedure will return a data structure that contains all the informations necessary to the support to allow a meaningful cooperation between worker and communicator. It should be noticed that this solution allow the creation of an arbitrary number of communicator, which can be useful in the case of architecture *n-threaded* with $n \geq 2$.

4.4.2 Delegation queue

The structure shared by worker and communicator is responsible for maintaining the send delegations waiting to be served. A delegation will contains at least the channel and the message references. Obviously, it must have a queue structure in order to guarantee that the ordering of communications is preserved. This structure will be referred as *delegation queue*.
A problem that must be solved is how to size this structure. In many practical cases, a limited number of send delegations could be enqueued, due to the structure of many common parallel paradigms. However, imposing a fixed size to the queue *a priori* could result in a violation of send semantics: a sender process can be suspended if there is no space into the delegation queue even if the buffer is not full. The solution adopted is to implement a queue whose size is *potentially infinite*, in the sense that can dynamically grows when required. Access to it take place in a mutual exclusion fashion.
Since the delegation queue is the only shared data structure between worker

and communicator, the synchronization issues between these two entities can be solved elegantly by regulating in an appropriate way the accesses to this structure. The delegation queue is made blocking while extracting an element:

- the worker thread, will delegate the send by inserting an element into the queue. This operation is not blocking and when, executed, will notify the event to the communicator via a `signal` primitive;
- the communicator extracts the delegation from the queue. If it is empty, it will use the `wait` primitive to wait until a new delegation is inserted.

4.4.3 Check buffer full

Since the check on the channel asynchrony degree saturation is performed by the worker itself, a consistency problem arises. The worker can not simply check the messages field of the channel descriptor because this information could be inconsistent (send delegations may be enqueued but not yet performed by the communicator). A simple solution could be to let the worker modifies the field but not releases the lock of the channel; it is in charge of the communicator, once the communication is completed, to unlock it. The simplicity of this solution has its counterpart in the fact that the channel could remain locked for a long time, preventing the receiver to access to it.
The solution adopted is to release the lock variable after that the message field is updated and the control is performed; it will be acquired again by the communicator when the send will be actually performed. In this way, however, the channel could remain in an inconsistent state: the receiver can find a non empty channel even though it is not so (the send has yet to be executed). For this reasons the validity bit associated at each target variable has been introduced: it indicates to the receiver whether or not the next target variable to be read has a meaningful content.
Summarizing, the actions performed by the various entities are:

- worker: accesses in mutual exclusion to the channel descriptor, updates

the message field, checks the asynchrony degree (eventually is suspend); in any case, releases the lock on the channel descriptor;

- communicator: executes the copy of the message in the target variable, accesses in mutual exclusion to the channel for updating the informations regarding the insertion pointer and sets the validity bit;

- receiver: this time must check not only the presence of message through the proper field, but also the relative validity bit. If a new message is actually present, it is returned and the relative validity bit is reset to false.

4.4.4 Send completion notification

For the send notification, a solution similar to the one of MPI specification is adopted: for each send delegated to the communicator, a *ticket* is returned to the caller. The ticket is part of the delegation message. The communicator, when a send is completed, will validate the correspondent ticket; on the other hand the worker, before modifying the message, must check the ticket. There is the need, again, of synchronizing two entities.

4.4.5 Communicator finalization

Once that the worker thread finishes its job, it must properly finalize the communicator. This is done by inserting into the delegation queue a particular value and wait for communicator termination; the communicator, when read the message, will exit. This mechanism will ensure that, once the worker is terminated, all the delegated communications have been completed.

4.4.6 Evolution of communication primitives

Taking into account all the discussed solutions, the communication primitives for asynchronous channel must be modified.
Regarding the send on the worker side, the main operations executed are shown in Listing 4.6. In this case, the send creates a proper delegation and inserts it into the delegation queue shared with the communicator. Then the buffer full check is performed as explained.

```
Delegation *send(Communicator KP, Channel ch, char *msg)
{
    //create a delegation
    Delegation *dg= createDelegationStructure();
    dg->ch=ch;
    dg->msg=msg;

    //update the channel
    lockChannel( ch );
    ch->messages ++;

    //enqueue the delegation
    enqueue(kp->delegationQueue,dg);

    //wait if buffer full
    while( ch->messages > ch->k )
        wait(&(ch->cond_tosender), &(ch->lockVar));
    unlockChannel( ch );
    return dg;
}
```

Listing 4.6: Send primitive worker side

On the communicator side, the send is actually the main function of the thread, that is executed until it is stopped (Listing 4.7). The VALB macro, computes the address of the validity bit.

```
void CommunicatorMainFunction()
{
    Delegation req;
```

```
    //Extract a delegation from the queue. If it is equal to NULL
    //it means that we have to terminate
    while((req=(Delegation *)dequeue(delegationQueue))!=NULL)
    {
        Channel ch=req->ch;

        //copy the message to the target variable
        memcpy( VTG(ch,ch->insVtg), msg, ch->size);

        lockChannel( ch );
        //set the validity bit
        VALB(ch,ch->insVtg)=true;
        ch->insVtg = ( ch->insVtg + 1 ) % ( ch->k + ch->j +1);

        //wakeup partner
        signal(&(ch->cond_toreceiver));
        unlockChannel(ch);

        //signal that the send has been completed
        lock(&(req->lV));
        signal(&(req->ticket));
        unlock(&(req->lV));
    }
}
```

Listing 4.7: Send primitive communicator side

The receive primitive, is very similar to the previous one, except for the operations on the validity bit that must be performed. Due to its lightweightness, the receive will be executed by the worker itself.

CHAPTER 5

Tests and results

This chapter deals with the results obtained from the comparison of the two different versions of the runtime support for LC, with or without the communicator thread facility; for the sake of simplicity, these two will be also referred as *KP* or *non-KP* version. The tests were performed on common types of parallel applications:

- a stream computation, structured according to the farm paradigm;
- a data parallel computation (stencil based) on stream.

Moreover, in the case of runtime support with communicator the two different synchronization mechanisms (POSIX or `MONITOR/MWAIT`) illustrated in section 4.2 will be analysed. The main purposes of this comparison are:

- to evaluate if the KP version of the runtime support is really able to overlap main part of the communications with the computation;
- to understand whether or not is convenient use all the thread contexts to increase the degree of parallelism or use half of them for supporting a communicator thread.

In general, to exploit the overlap between computation and communications, the programs must be slightly modified. These interventions will be anyway minimal, negligible in the case of stream based computations.

5.1 Stream computation

The considered computation is composed by a process P that operates on a stream of matrices. Each element of the stream represents a frame coming out from a video source; the generic element of the matrix is a triplet that identifies the three color component (red, green and blue, 1 byte each) of the respective frame's pixel. A simple filter, sketched below, is applied to each frame for enhancing the image colors.

```
void purify(Frame f)
{
    for(i=0;i<f.height;i++)
        for(j=0;j<f.width;j++)
            if(sq(f.r-f.g) + sq(f.r-f.b) + sq(f.g-f.b) > thresh)
                <assign to each color the avg of the previous ones>
}
```

The resulting frames are sent to another device, such as a monitor. Hence, the process P is defined as follows:

```
while(true)
{
    receive(input_stream, frame);
    purify(frame);
    send(output_stream, frame);
}
```

Since it represents a pure function, P can be parallelized according to the *farm paradigm* (Figure 5.1). In particular:

- the workers W_i are essentially replications of the process P;
- the emitter E schedules the frames of the input stream to the various workers; this is done by distributing the frames according to a round robin strategy;
- the collector C collects the resulting frames from each worker; applying also in this case a round robin strategy (*i.e.* the collector performs a

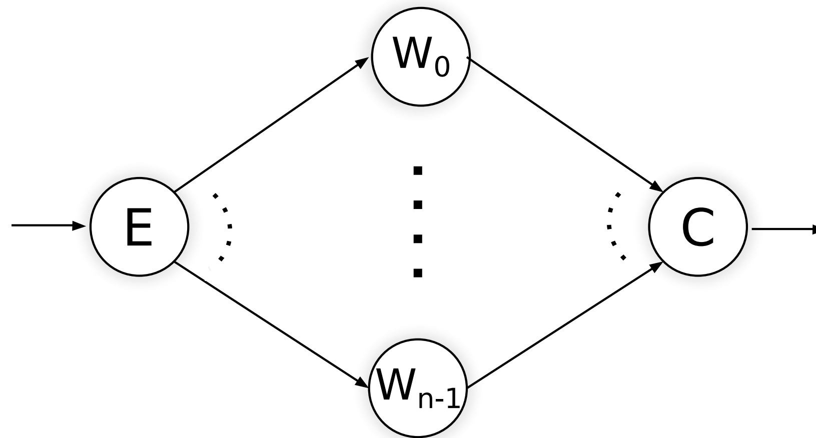

Figure 5.1: General structure of a computation parallelized according to the farm paradigm

receive from the channel relative to W_0, hence a receive from W_1, and so on) the frame ordering is preserved in a simple way.

In such a situation, where emitter and collector do not perform other work than communications, the benefit of an overlap between computation and communication will be due to the sends made by a worker toward the collector. In this case, the logic of the program does not need to be further modified in order to exploit the KP version of the runtime support.
On the reference architecture (constituted by 8 physical cores, each of them two-threaded; see chapter 4), the emitter and the collector will be bound to two different physical processors. The workers (whose number identifies the parallelism degree) will be:

- up to 12 if no communicator is used: they are scheduled first of all on different physical processors and, if the parallelism degree is greater than 6, on sibling logical processors;
- up to 6 if communicators are used, because for each worker a communicator thread is created and bound to the sibling logical processor.

Different test cases have been created, by using frames of different sizes and, as usually in the case of stream computations, the parameter of interest will be

the *service time*. Each of the test cases was run multiple times, with a stream of fixed length (2400 frames). The results obtained are shown in Figure 5.2 and 5.3: in these cases the stream is composed by squared frames of size 800×800 or 1000×1000 pixels. In Table 5.1 the calculus time needed for apply the filter on a frame (T_{calc}) and the communication latency involved in sending an element of the stream through a communication channel (L_{com}) are reported lo.

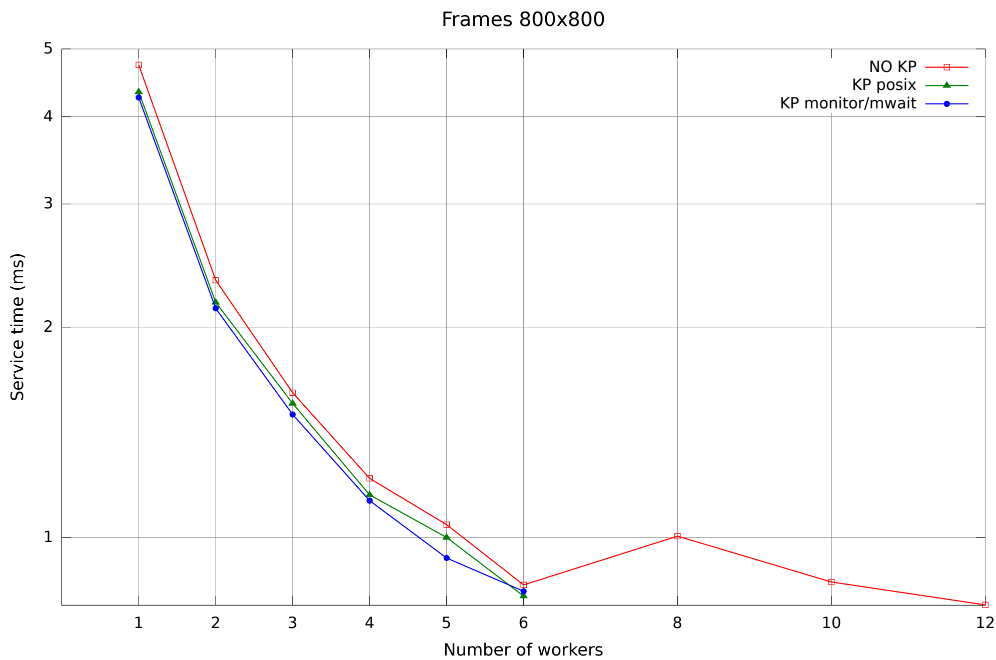

Figure 5.2: Results with a stream composed by frames of 800×800 pixels. KP posix and KP monitor/mwait refer to the KP versions of the runtime support that use the respective synchronization primitives. The plot is in *log-lin* scale

Frame size	T_{calc}	L_{com}
800×800	4.17	0.41
1000×1000	6.39	0.579

Table 5.1: Calculus and communication times (expressed in milliseconds) experimentally evaluated

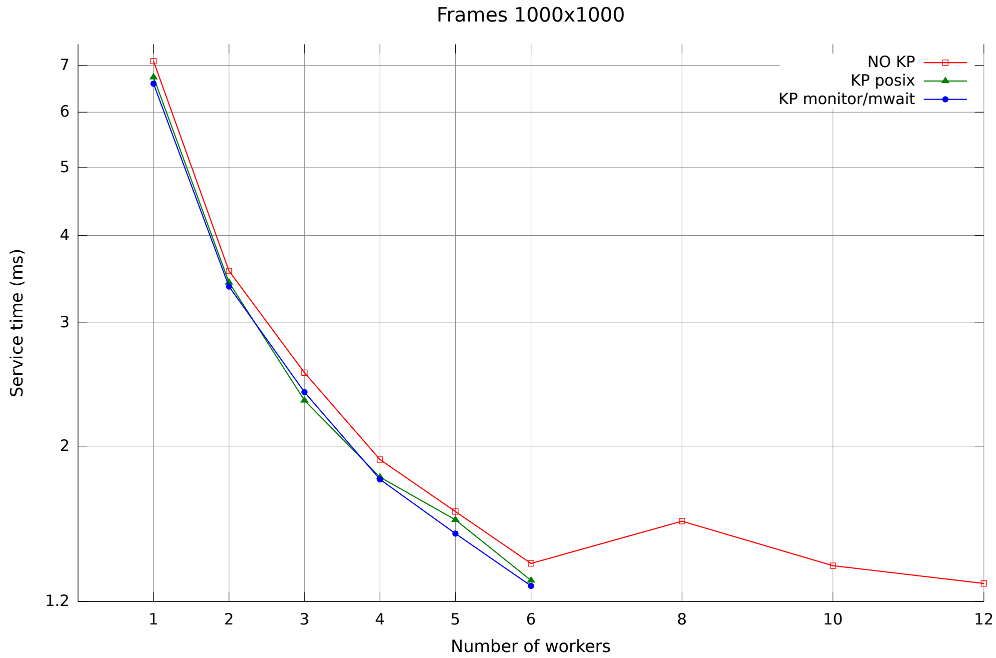

Figure 5.3: Results with a stream composed by frames of 1000×1000 pixels

In the case of the runtime support without the communicator facility, increasing the parallelism degree over the number of physically available processors (thus allocating a worker for each logical processor) does not lead to sensible benefits: on average the service time decreases of the 6%, passing from 6 to 12 workers. For intermediate values of the degree of parallelism the program performs worse, due to the round robin scheduling strategy used. On the other hand, especially in the case of one worker, the KP version of the runtime support allows to overlap most part of the communication with the computation performed by the workers (as explained in section 4.4, part of the runtime support of the send is still executed by the worker thread). The difference between the two versions of the runtime support vanishes as the parallelism degree grows. Nevertheless, this is a logical consequence of the parallelism paradigm applied. From the cost model of the farm paradigm, the service time of a worker is known to be equal to:

$$T_W = T_{calc} + L_{com}$$

where L_{com} is equal to zero if the communications are overlapped with the computation; the time necessary for the additional controls performed by the worker thread should be considered in T_{calc}. For the sake of simplicity, T_{calc} will be considered equal in both cases. For zero copy communications, both the emitter and the collector service times are equal to the communication latency, since they do not perform other work:

$$T_E = T_C = L_{com}$$

Moreover, also the inter-arrival time to the emitter will be considered equal to L_{com}. Thus, the service time of the farm is equal to:

$$T = max(\frac{T_W}{n}, L_{com})$$

where n is the parallelism degree. In this particular case, since $n \leq 6$ and $L_{com} \leq \frac{T_W}{6}$, it can be concluded that:

$$T = \frac{T_W}{n}$$

Considering the two different cases, the service time will be:

- in the case of non-KP version of the runtime support:

$$T = \frac{T_{calc} + L_{com}}{n} = \frac{T_{calc}}{n} + \frac{L_{com}}{n} \tag{5.1}$$

- in the case that the KP version is used:

$$T = \frac{T_{calc}}{n} \tag{5.2}$$

This computation is characterized by the property $L_{com} << T_{calc}$. This is not unusual in the case of parallel computation on shared memory systems. As n grows, the fraction L_{com}/n in (5.1) will become smaller and smaller and the difference with (5.2) disappears.

Consider now the case in which the communications require a time comparable to the computation one (at least same order of magnitude). The process

considered operates on a stream of vectors whose elements are of type double; a simple arithmetic transformation is applied on each vector element. Also in this case, the parallelization is done according to the farm paradigm. Even if the computation is not meaningful per se, it represents a typical situation that could occur in a distributed memory system, where the T_{setup} and T_{trasm} parameters (involved, as explained in section 3.1, in the evaluation of L_{com}) are typically one order of magnitude, or more, bigger compared to the shared memory case. The results are shown in Figure 5.4 and 5.5, while the values of T_{calc} and L_{com} are reported in Table 5.2.

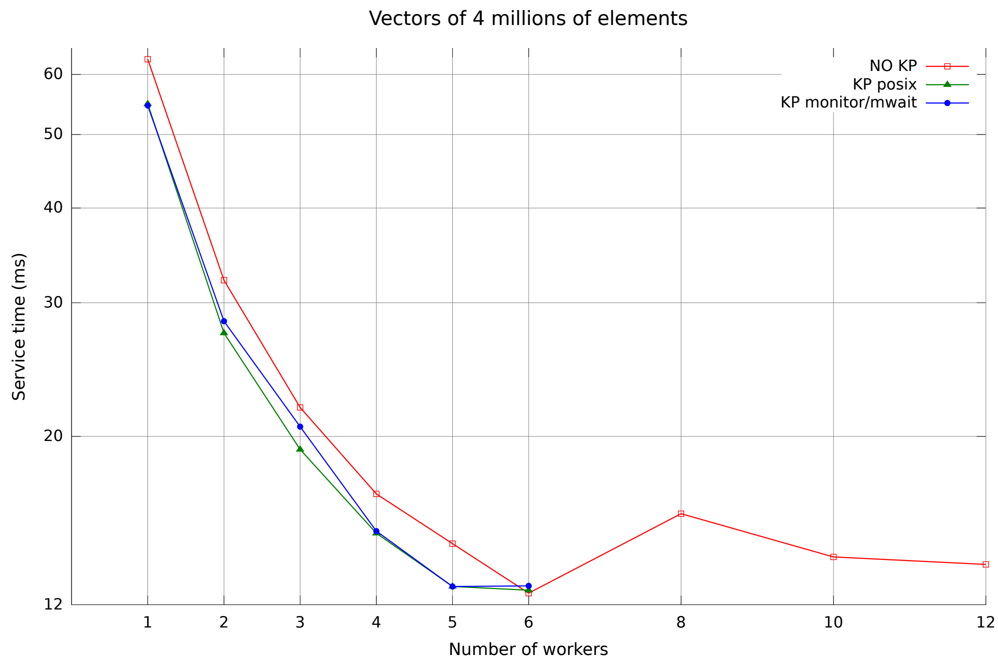

Figure 5.4: Results obtained with a stream composed by vectors of 4 millions of elements

Vector size	T_{calc}	L_{com}
$4M$	51.93	10.04
$8M$	102.12	19.84

Table 5.2: Calculus and communication times (expressed in milliseconds) experimentally evaluated

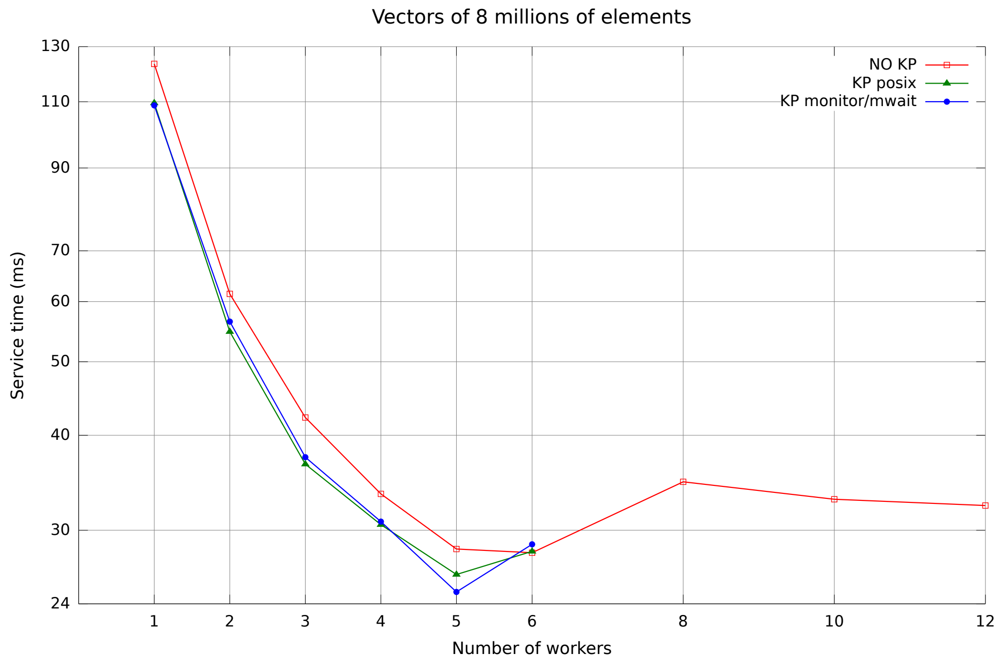

Figure 5.5: Results obtained with a stream composed by vectors of 8 millions of elements

In this case, the overlap is more visible thanks to the fact that L_{com} has an heavier impact. Moreover, using the communicator threads, there is the possibility to exploit the optimal degree of parallelism in the reference architecture ($n = 5$), resulting in a lower service time compared to the non-KP version. Again, use all the logical processors for doubling the parallelism degree do not leads to any benefits: there is even a degradation of the performances, passing from $n = 6$ to $n = 12$, probably due to the high cache line contention between two threads allocated on two sibling processors.

5.2 Data parallel computation

A process P is in charge of apply a convolution computation on a stream of matrices. A typical application of such calculus can be found in edge detection algorithms, that allow to recognize the border of the objects present into an image.

The value of every point in a discrete space (an element of the two dimensional

matrix) is updated by a function applied to the point itself and to its neighbour points (Figure 5.6). This is repeated until a given convergence condition is sa-

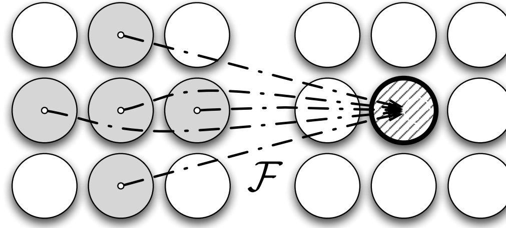

Figure 5.6: Functional dependences of the considered computation

tisfied or a maximum number of iterations has been reached. The convergence condition is a proper predicate that must be satisfied by all the elements of the matrix: in the computation examined it requires that, for all the points, the absolute difference between the current and the previous iteration value is less than a given threshold.

```
while{true}
{
    receive(input_stream,A);
    do{
        for(i=0;i<N;i++)
            for(j=0;j<M;j++)
                A'=C*(A[i,j]+A[i-1,j]+A[i+1,j]+A[i,j-1]+A[i,j+1]);
        A=A';
    }while(convergence(A) || #iter < MAX_IT);
    send(output_stream,A);
}
```

Listing 5.1: In the computation considered the process works on a matrix stream of dimension NxM. C is a program dependent constant

The process P is parallelized obtaining a stencil-based computation that operates on a stream of matrices. Such computations are characterized by workers that operate in parallel and cooperate through data exchanges. The *stencil* is a data dependence pattern implemented by inter-worker communications. In this study case, the stencil will be static and fixed, meaning that the communication pattern is known at compilation time and does not change throughout

the computation. Like any data parallel computation, there will be a prior

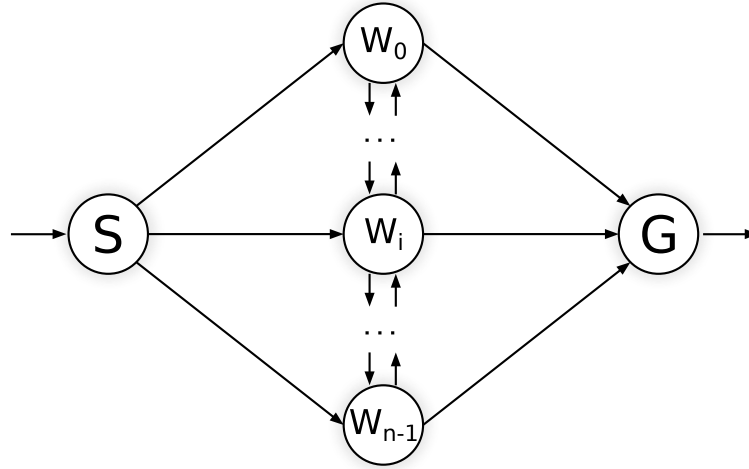

Figure 5.7: Stencil based computation that operates on stream

phase for partitioning the data to all the workers, the *scatter*, and a final phase for collecting the result, the *gather* (Figure 5.7). These two forms of collective communications will be implemented by means of multiple sends. The process S will be in charge of performing the scatter of the input matrices toward the workers: the partitioning adopted will be *by rows*, meaning that S will send a bunch of consecutive matrix rows to each of them. In this way, neighbour workers need to exchange the first and the last row (the borders) of the matrix each other. Process G will perform the gather operation, collecting all the row partitions hold by the different workers, building the final matrix. The convergence condition, is expressed now by a reduce computation. Given the limited degree of parallelism exploitable in the reference architecture, it is realized without particular optimizations.
The process P, in the case that communicator threads are not used, evolves as represented in Listing 5.2.

```
while{true}
{
   receive(input_stream,B); //receive a partition of the matrix A
   send(w(i-1)_out,B[0]); //first row to the upper neighb.
   send(w(i+1)_out,B[N/n]); //last row to the lower neighb.
```

```
    receive(w(i-1)_in, B[-1]); //last row from the upper neighb.
    receive(w(i+1)_in, B[N/n+1]);//first row from the lower neighb.
    do{
        for(i=0;i<N/n;i++)
            for(j=0;j<M;j++)
                B'=C*(B[i,j]+B[i-1,j]+B[i+1,j]+B[i,j-1]+B[i,j+1]);
        B=B';
    }while(reduce(B) || #iter < MAX_IT);
    send(output_stream,B);
}
```

Listing 5.2: Pseudo code of the generic worker i. B is the partition of the matrix assigned

Clearly the communication channels must be asynchronous to avoid deadlocks. If the KP version of the runtime support is utilized, the process must be further modified: after that the partition of the matrix is received by the worker, two sends are delegated to the communicator in order to forward the first and the last row to the neighbours. Meanwhile, the computation of the inner part of the matrix (that does not require informations additional to the ones already hold) is executed and, once finished, the receive of the borders from the neighbours is done in order to compute also the new elements of the first and last row. In this situation there are two possible sources of overlapping:

- the send of the resulting matrix done by the workers toward the process G for the gather;
- the send of the borders between neighbours workers.

The communications involved in the reduce phase are not feasible to be overlapped with the computation.

The tests were performed with a stream of fixed size composed by matrices with the same size; also in this case different sizes were used. For the termination of the inner loop, in order to perform a fair comparison, only the condition on the number of iterations will be considered. In any case, even if its result is not actually used, the reduce operation is executed.
The results are shown in Figure 5.8, 5.9 and 5.10. Because there were no

significant differences, for the KP version is shown only the one that uses POSIX based synchronization mechanisms.

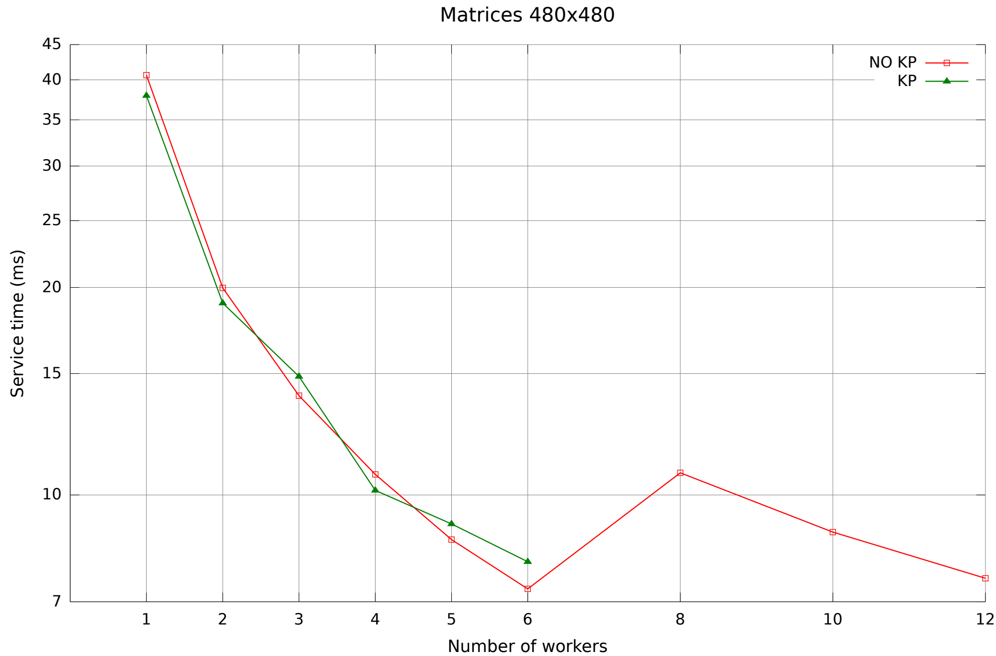

Figure 5.8: Results obtained with a stream of matrices 480×480

As can be seen, in the first test case the two versions of the runtime support perform in a similar way; increasing the size of the stream elements the benefits of the communicator threads becoming more significant. This is pretty obvious, thanks to the fact that more time will be necessary to perform the gather operation and the borders exchange between neighbour workers. Moreover, since it is adopted a row partitioning of the matrices, the communication latencies of the stencil are constant independently from the parallelism degree. The effect of this situation is visible especially in the last test case were the gain due to the KP version of the runtime support over the non-KP version remains almost constant as the parallelism degree grows.
Also in this case, it should be noticed that exploiting the multithreading by simply doubling the parallelism degree does not bring to any considerable benefit.

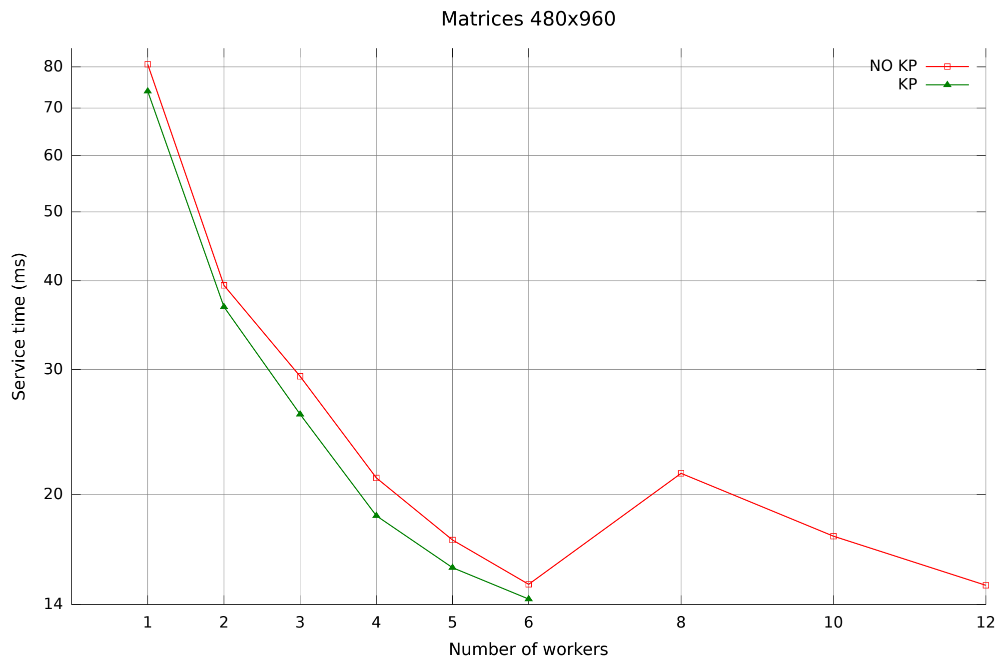

Figure 5.9: Results obtained with a stream of matrices 480×960

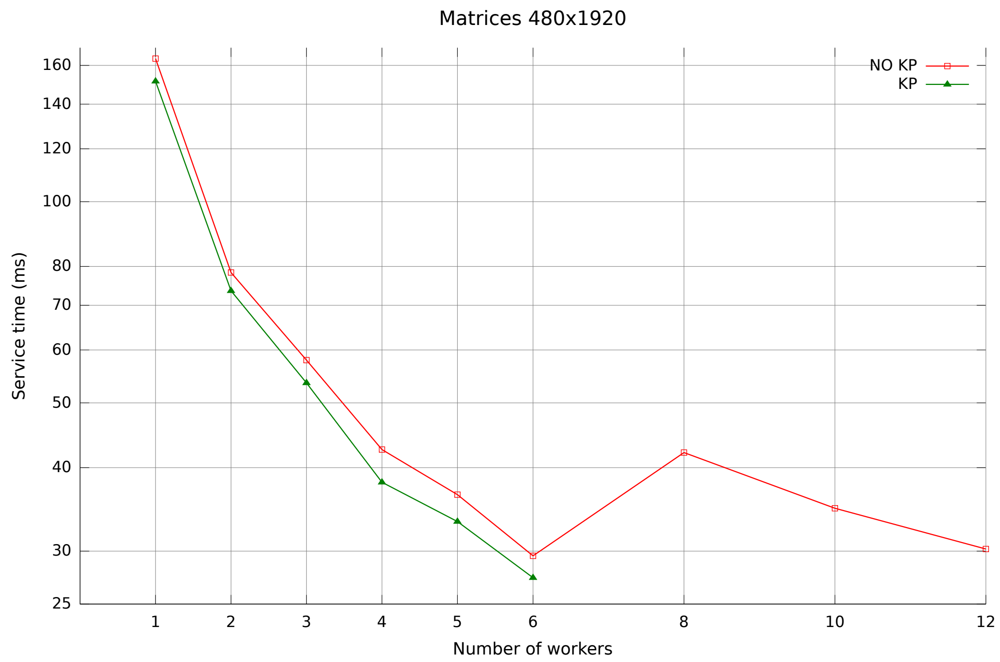

Figure 5.10: Results obtained with a stream of matrices 480×1920

CHAPTER 6

Conclusions and future works

The work of this thesis was aimed at investigating a possible way of exploiting the hardware multithreading technique for supporting interprocess communication in shared memory systems, emulating the facility of a communication processor when it is not physically available. For this purpose, an appropriate runtime support of the concurrent language LC has been realized. The approach adopted is to associate to each process/thread of the parallel application (the *worker thread*) a thread that is in charge of executing the send primitive (the *communicator thread*). Once that a send execution is delegated to the communicator, the worker thread can continue the main computation while, in parallel, the communication is performed.
The produced results are encouraging: the communicator thread allows to overlap main part of a communication (on average the 75% on the considered test cases), without interfere too much with the worker one. This was made possible thanks to an efficient implementation of the runtime support, that exploits features offered by Unix-based operating systems (such as the *shared memory segment* for implementing zero copy communications) and uses appropriate inter-thread synchronization mechanisms. Regarding the latter, two types of mechanisms have been considered: the ones based on classical POSIX mechanism (*i.e.* mutexes and condition variables) and the ones based on the

MONITOR/MWAIT Intel's assembler instructions. In practical terms, in the reference architecture it has been shown that it is almost equivalent to use one of them. However, since the MONITOR/MWAIT instructions appear to be subject to future improvements, probably they will become the preferred choice in newer systems, assuming that they will be released for use at user level.

The use of the produced runtime support, introduces some advantages already in the shared memory case, where typical applications could be characterized by a communication latency one (or more) order of magnitude smaller than the calculus time. Nevertheless in particular context where this is not true, or, a fortiori, in the case of distributed memory systems, the benefits will be more visible, as a simple analysis based on the cost models can demonstrate.

Future works

This work can be considered a starting point to efficiently exploit multithreaded systems for supporting interprocess communication in parallel architectures. Further improvements could go essentially in two directions:

- expansion of the library: the runtime support implemented regards a minimal core of LC: in particular, it refers to the case of symmetric, synchronous or asynchronous, deterministic channel. Clearly this could be expanded, including the support for non deterministic channels (hence alternative commands) and collective communications;

- implementation of a runtime support for distributed memory systems: in this class of architectures, no memory sharing is physically possible among processes allocated onto distinct processing nodes. The only primitive architectural mechanism for node cooperation is the communication by value, that is the cooperation via input-output mechanisms and interface units. Interprocess communications must be implemented on top of

such mechanism. Since the communication latencies in this kind of systems are one or two order of magnitude bigger compared to the shared memory case, the benefits of interprocess communication overlappable with the computation will be more relevant. Clearly the support must be heavily modified, but many consideration done in this work will be still valid.

APPENDIX A

The library

The runtime support of LC has been implemented as a library written in *C*, designed to be directly used in user space. The programmer that wants to use it needs to include the `channel.h` header. In the following, a brief description of the offered functionalities and of the library internal structure is given.

Data types

The user of the library will handle three different kinds of data types:

Channel: represents a reference to a generic communication channel;

Communicator: communicator thread descriptor. It contains useful informations about a communicator thread;

Delegation: represents the delegation of a send from a worker thread to a communicator one.

All these kind of data types are substantially pointers and represent informations that will be returned from, respectively, the creation of a communication channel, the creation of a communicator thread, the offload of a send to the communicator.

Functions

The functions offered by the library can be grouped into four main categories: communication channels management, communicator thread management, communication primitives and utilities. These will be now briefly described, pointing out precautions that have to be taken while using them.

Channel management

`LCcreate_ss( key_t key, size_t size, int j )`
: Creates a synchronous communication channel, with a given data size *size* and implemented using *j* additional target variables. *key* is the key of the shared memory segment that will contain all the data structures relative to the channel; in this case is used also to identify the channel. It can be generated using proper System V functions, such as `ftok`. On success a reference to the created channel is returned. On failure `NULL` is returned.

`LCcreate_as( key_t key, size_t size, int k, int j)`
: Creates an asynchronous communication channel, with a given data size *size*, asynchrony degree *k* and implemented using *j* additional target variables, in a similar way to the previous function.

`Channel LCattach( key_t key )`
: This function must be called by a process that intends to use the channel identified by *key*. In practice it is used to acquire the shared memory segment that contains all the data structures of the particular channel into the addressing space of the caller process. On success a reference to the channel is returned, otherwise `NULL`.

`void LCdetatch( Channel ch )`
: Detaches the channel identified by the passed reference. By doing so, the

respective shared memory segment is removed from the virtual memory of the caller.

`void LCdestroy( Channel ch )`

Destroys the channel. Its relative shared memory segment will be removed from memory only when the last process detaches it.

Communicator management

`Communicator initCommunicator(int core_mapping[])`

This function will create a communicator thread, allocating all the data structures necessary for the interaction with the worker. The thread must be scheduled on the same physical processor where the caller resides: for this reason, `core_mapping` is an integer vector whose size is equal to the number of logical processors present in the machine. `core_mapping[i]`, where i is the processor ID of the worker thread, indicates the ID of the logical processor where the communicator thread must be scheduled in order to be on the same core of the partner. It can be built according to the informations published through the *sys* pseudo file system (see section 4.1). On success a reference to the communicator descriptor is returned to the caller. On failure `NULL` is returned.

`void finalizeCommunicator(Communicator kp)`

This function must be called by the worker once that it no longer need the communicator and before that the used channels are detached from its address space. It will stop the communicator thread, waiting for its termination (that will occurs when all the delegated send have been served).

Communication primitives

`void LCssend(Channel ch, char *msg )`

Sends the message pointed by *msg* through the synchronous channel *ch*. Being a *synchronous communication*, this will be executed directly from the caller.

`Delegation LCsend( Communicator kp, Channel ch, char *msg )`

Delegates the send of the message *msg* through the channel *ch* to the communicator thread referenced by *kp*. The procedure can be blocking, if the channel is full (the asynchrony degree has been reached). A reference to the send delegation is returned to the caller.

`Delegation LCbsend( Communicator kp, Channel ch, char *msg )`

Buffered send: a copy of the message is created and sent to the partner. The send is anyway offloaded to the communicator. The copy of the message is done by communicator, if it is not working, or by the caller itself. In this way, it will be possible to modify the message sooner with respect to the traditional send. Also in this case a reference to the send delegation is returned to the caller.

`void LCwait(Delegation dg)`

This procedure imposes to the caller to waiting until it is safe to modify the message object of the delegation *dg*. This occurs when the message has been sent through the channel or, in the case of buffered send, a it has been copied

`char *LCreceive( Channel ch )`

Receives a message from the channel *ch* and returns it to the caller. The receive, being in the case of zero copy communications, is executed directly by the worker. It can be blocking if the channel is empty.

Utility functions

`size_t LCgetChannelTypeSize( Channel ch )`
Returns the size of the data type transmittable with the channel.

`bool LCreadyToReceive( Channel ch )`
Returns whether or not the channel *ch* contains at least one valid message.

`bool LCreadyToSend( Channel ch )`
Returns whether or not the channel *ch* is full.

Internal structure of the library

The library is composed (apart from the header `channel.h`) by various files that are in charge of defining or implementing all of its aspects.
Regarding the header file, they are:

- `synchr.h`: this header file contains the definitions of locking and synchronization primitives. The offered synchronization primitives recall the bahaviour of POSIX condition variables. Two different mechanisms are used: POSIX based and `MONITOR`/`MWAIT`instructions. The choice of which one to use is done at library compilation time, by means of proper flags;
- `shm_common.h`: contains the definitions of the aforementioned data types (channel, communicator and delegation descriptors) and internal functions for managing the delegation queue and synchronization on the channel descriptor.

The implementation of the various functionalities is partitioned in different source files:

- `synchr.c`: regards the implementation of synchronization primitives;
- `inf_blkqueue.c`: contains the definition of the queue that will be used as delegation structure between the worker and the communicator thread;

- `uni_shm.c`: implements the channel management and communication (worker side) functionalities;
- `kp_comm.c`: contains the communicator thread related functionalities (creation, finalization, definition of the main function executed by the thread).

Bibliography

[1] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE Int'l Parallel & Distributed Processing Symp*, 2008.

[2] Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis, and Nectarios Koziris. Exploring the performance limits of simultaneous multithreading for memory intensive applications. *J. Supercomput.*, 44:64–97, April 2008.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15:29–36, February 1995.

[4] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 14–25, New York, NY, USA, 2001. ACM.

[5] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[6] Matthew Curtis-Maury and Tanping Wang. Integrating multiple forms of multithreaded execution on multi-smt systems: A study with scientific applications. In *Proceedings of the Second International Conference on the*

Quantitative Evaluation of Systems, pages 199–, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Ulrich Drepper. *What every programmer should know about memory.* Technical report, Red Hat Inc., 2007.

[8] Paolo Giangrandi. *Supporto alle comunicazioni su piattaforme commerciali orientato all'high perfromance computing.* Relazione di tirocino, Universitá di Pisa, 2009.

[9] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[10] Rusty Russell Hubertus Franke and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the 2002 Ottawa Linux Summit*, 2002.

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, January 2011.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2011.

[13] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6, 2002.

[14] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[15] Maeda Toshiyuki. Kernel mode linux: Toward an operating system protected by a type theory. In Vijay Saraswat, editor, *Advances in Computing Science – ASIAN 2003. Programming Languages and Distributed Computation Programming Languages and Distributed Computation*, volume

2896 of *Lecture Notes in Computer Science*, pages 3–17. Springer Berlin Heidelberg, 2003.

[16] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 26–, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 54–, Washington, DC, USA, 1999. IEEE Computer Society.

[18] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35:29–63, March 2003.

[19] Marco Vanneschi. *Architettura degli elaboratori.* Edizioni PLUS, 2009. Also: Course Notes of High-performance Computing Systems and Enabling Platforms, Master Program in Computer Science and Networking, University of Pisa, 2011.

[20] Marco Vanneschi. *Instruction level parallelism: scalar, superscalar and multithreaded architectures.* Integration to Course Notes of High-performance Computing Systems and Enabling Platforms, Master Program in Computer Science and Networking, University of Pisa, 2011.