Departamento de Informática

Distributed Multi-Threading in GNU Prolog

Nuno Eduardo Quaresma Morgadinho
<nm@di.uevora.pt>

Supervisor: Salvador Abreu (Universidade de Évora, DI)

This thesis does not include appreciation nor suggestions made by the jury.
Esta dissertação não inclui as críticas e sugestões feitas pelo júri.

Évora
2007

Departamento de Informática

Distributed Multi-Threading in GNU Prolog

Nuno Eduardo Quaresma Morgadinho

<nm@di.uevora.pt>

Supervisor: Salvador Abreu (Universidade de Évora, DI)

This thesis does not include appreciation nor suggestions made by the jury.

Esta dissertação não inclui as críticas e sugestões feitas pelo júri.

Évora

2007

Abstract

Although parallel computing has been widely researched, the process of bringing concurrency and parallel programming to the mainstream has just begun. Combining a distributed multi-threading environment like PM2 with Prolog, opens the way to exploit concurrency and parallel computing using logic programming. To achieve such a purpose, we developed PM2-Prolog, a Prolog interface to the PM2 system. It allows multithreaded Prolog applications to run in multiple GNU Prolog engines in a distributed environment, thus taking advantage of the resources available on a computer network. This is especially useful for computationally intensive problems, where performance is an important factor. The system API offers thread management primitives, as well as explicit communication between threads. Preliminary test results show an almost linear speedup, when compared to a sequential version.

Keywords: Distributed, Multi-Threading, Prolog, Logic Programming, Concurrency, Parallel, High-Performance Computing

Resumo

Multi-Threading Distribuído no GNU Prolog

Embora a computação paralela já tenha sido alvo de inúmeros estudos, o processo de a tornar acessível às massas ainda mal começou. Através da combinação com o Prolog de um ambiente de programação distribuída e *multithreaded*, como o PM2, torna-se possível ter computações paralelas e concorrentes usando programação em lógica. Com este objectivo foi desenvolvido o PM2-Prolog, um interface Prolog para o sistema PM2. Tal sistema permite correr aplicações Prolog *multithreaded* em múltiplas instâncias do GNU Prolog num ambiente distribuído, tirando, assim, partido dos recursos disponíveis nos computadores ligados numa rede. Em problemas computacionalmente pesados, onde o tempo de execução é crucial, existe particular vantagem em usar este sistema. A API do sistema oferece primitivas para gestão de *threads* e para comunicação explícita entre *threads*. Testes preliminares mostram um ganho de desempenho quase linear, em comparação com uma versão sequencial.

Acknowledgments

Special thanks to Salvador Abreu, my supervisor at Universidade de Évora, for making this thesis possible. Also thanks to Olivier Aumage, researcher at INRIA, for answering my initial questions about PM2, Marcel and Madeleine programming. I also thank Pedro Martelletto, Nuno Lopes, Rui Marques, Cláudio Fernandes, Paulo Moura and Vasco Pedro for reviewing this thesis. Finally, I have a lot of gratitude for my family and my love, who provided endless support. This work is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation stems from a study made in the Logic Programming field. It describes the implementation of a system that allows distributed multi-threading in GNU Prolog [Diaz and Codognet, 2000].

1.1 Motivation

The motivation for this work came from the conviction that it would be useful to analyse the viability and performance of a system that combined High Performance Computing (HPC) with logic programming, and that it could be achieved by associating PM2 [Namyst and Méhaut, 1996] with GNU Prolog.

PM2 allows distributed multi-threading C applications to be developed and is based on the *Single Program Multiple Data* paradigm, and GNU Prolog, a widely used Prolog system released under GPL license, presents a compiler (gplc) that generates stand-alone binaries, which fits nicely into this paradigm.

PM2 is, above all, a programming environment. It is based primarily

on two distinct libraries: one for thread management (Marcel) and another for communication (Madeleine). Using such libraries in C, we developed a message-passing system, **PM2-Prolog** that allows for the creation of Prolog threads and communication between them.

The HPC community has developed other programming environments besides PM2, such as PVM [Sunderam, 1990], OpenMPI [Gabriel et al., 2004] or TreadMarks [Keleher et al., 1994], building a base for the development of multi-threaded applications in distributed environments. Nevertheless, the viability of combining such environments with logic programming has been confined to a few studies, like PVM-Prolog [Marques and Cunha, 1996].

Such combination is specially suitable when dealing with:

- Applications that potentially have some degree of parallelization whose performance needs to be improved;
- Applications comprised of intelligent agents, hosting one or more agents per machine;
- Scientific and business problems, such as simulation applications, that produce large amounts of data that need to be processed for visualization, data mining or machine learning. For many cases, faster results can be potentially achieved, subdividing the problem and processing each sub-task in a different processor;
- Applications where some loss of accuracy in result can be traded for faster execution times.

When dealing with computationally intensive problems on multi-processor or multi-core architectures, creating and optimizing threaded applications can

improve performance, since each thread can be assigned to a different processor unit or core. And with the arrival of multi-core processors, building up a super computer by assembling several smaller ones, becomes easier and cheaper than ever.

The use of such architectures can reduce the program execution time, sometimes very significantly, although there is a limit on the number of threads that are effectively running in one CPU at the same time, i.e., that are not blocked awaiting re-scheduling by the operating system, since the number of cores or processors in a single machine is finite.

One way to work around this limitation is to use a distributed multithreading environment. PM2, as such an environment, contains features that solve some important problems of distributed computing, to name a few:

- transparent deployment of binaries throughout the network;
- implementation of low-level thread management routines (Marcel);
- implementation of low-level communication routines (Madeleine);
- hard distributed computing problems already approached, such as the distributed termination detection;
- distributed debugging facilities;
- configuration framework for network hosts.

Our motivation was also spurred on by the fact that "Marcel" [Namyst and Méhaut, 1995] and "Madeleine" [Aumage, 2002] are both available for various network hardware and architectures.

1.2 Objectives

While carrying out the work described herein, our goal was to develop a system that:

- Allows distributed multi-threaded applications in Prolog to be developed.
- Achieves faster execution times than sequential Prolog systems for problems that have some degree of parallelization.
- Allows a wider range of Prolog applications to be used.
- Helps to study the viability of combining Prolog and HPC (High-Performance Computing).

1.3 Main Contribution

This work introduces a system that allows the exploitation of explicit parallelism and multi-threading in logic programming, based on a well known ISO-compliant Prolog, GNU Prolog and on PM2, a distributed multi-threading programming environment widely used in academia.

The implemented architecture allows new abstraction layers to be easily defined on top of it, providing a framework to develop parallel and distributed Prolog applications, or as a layer for the support of the execution of other applications that require heavy computational resources or to which applying some degree of parallellization may be beneficial.

1.4 Thesis Organization

The remainder of this thesis is organized as follows: **Chapter 2** provides background material on multi-threading, namely its benefits, models, limitations and gives an example of a multithreaded Prolog program.

Chapter 3 is about parallel systems and logic programming. Concepts needed to understand the thesis continue to be presented in short form and related parallel Prolog systems are subjected to appreciation as well as the most recent level of development of similar systems.

Chapter 4 describes PM2-Prolog in detail, its implementation, design and architecture. The API it provides is listed and an explanation of how to use it is also provided. Observations about ISO compatability are made and PM2-Prolog is compared to a similar system, PVM-Prolog, in terms of usage.

In **Chapter 5** the system is experimentally evaluated and the obtained performance results discussed.

Finally, **Chapter 6** draws conclusions and outlines possible proposals for future work.

Chapter 2

Multi-Threading

Each single-thread process is a *sequential* program, namely, a sequence of statements that are executed one after the other. Whereas a sequential program has a single thread of control, a *concurrent* program has multiple threads of control. The next figure illustrates the difference between a single-thread program and a multithreaded one:

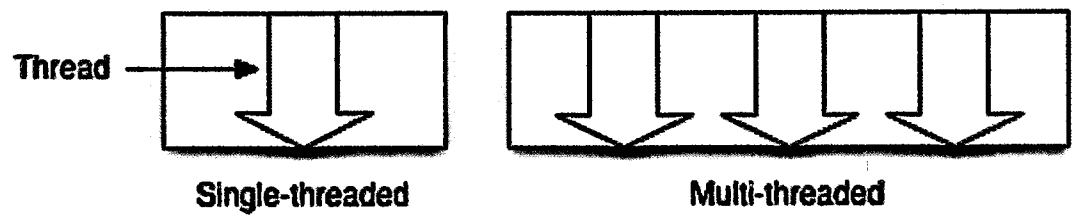

Figure 2.0.1: Single and multithreaded program.

A thread, sometimes called a lightweight process, is different from a process because when creating threads, they are added to the existing process rather than starting in a new process. Processes start with a single thread of execution and can add or remove threads throughout the duration of the program. Also, unlike processes, which operate in different memory spaces, all threads in a process share

the same memory space. Additionaly to this global shared memory space, each thread has a private area for its own local variables.

Threads have the advantage over processes in that multiple threads can cooperate and work on a shared data structure to fasten the computation. By dividing the work into smaller portions and assigning each smaller portion to a separate thread, the total work can be completed more quickly.

Multiple threads are also used in high performance database and Internet servers to improve the overall throughput of the server. With a single thread, the program can either be waiting for the next network request or reading the disk to satisfy the previous request. With multiple threads, one thread can be waiting for the next network transaction while several other threads are waiting for disk I/O to complete.

In a concurrent program the multiple threads work together by communicating with each other. Communication is programmed using shared variables or message passing. When shared variables are used, one process writes into a variable that is read by another. When message passing is used, one process send a message that is received by another.

Independently of the form of communication, often a way to *synchronize* threads with each other is needed. There are two basic kinds of synchronizaion: mutual exclusion and condition synchronization. The first is based on ensuring that critical sections of statements do not execute at the same time. The second consists on delaying a process until a given condition is true.

2.1 Benefits

According to [Silberschatz et al., 2000], the benefits of multithreaded programming can be broken down into four major categories:

- *Responsiveness*: Multithreading an interactive application is essencial to ensure the program continues responding even if part of it is blocked or is performing a lengthy operation.
- *Resource sharing*: By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing allows an application to have several different threads of activity all within the same address space.
- *Economy*: Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and switch the thread context.
- *Utilization of multiprocessor architectures*: The benefits of multithreading can be greatly increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor.

2.2 User-space and Kernel Threads

User-space threads are created, terminated, synchronized, scheduled, and so forth using interfaces provided by a threads library. Because user-space threads are not directly visible to the kernel (which is aware only of the process containing the user-space threads), user-space threads require no kernel support. Any user-level thread performing a blocking system call will cause the entire process to block, even if there are other threads available to run within the application.

Kernel threads are supported directly by the operating system: thread creation, scheduling and management are done by the kernel in kernel space. Because thread management is done by the operating system, kernel threads are generally slower to create and manage. However, since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution.

2.3 Thread States

Typically, a thread is in one of the following states:

- New - execution has started.
- Runnable - running in the system scheduler.
- Blocked - waiting for a mutex or resource.
- Dead - execution is stopped and cannot be resumed.

Figure 2.3.1 illustrates the states in which a thread might be and the actions leading to each state.

2.4 Models

According to [Silberschatz et al., 2000], many systems provide support for both user and kernel threads, resulting in different multithreading models.

The *many-to-one* model maps many user-level threads to one kernel thread, with thread management being done in user space.

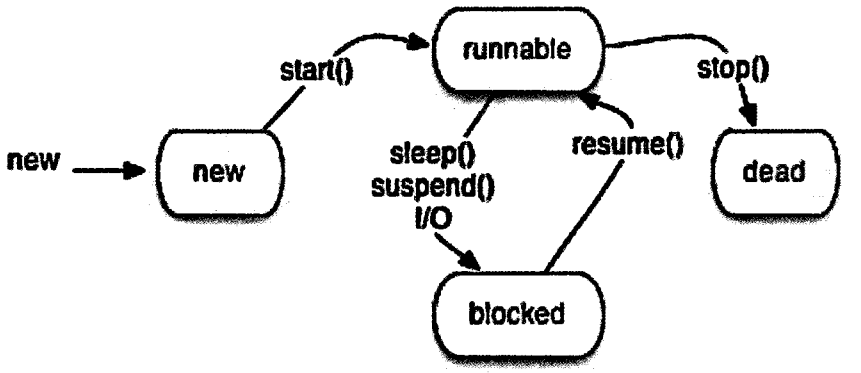

Figure 2.3.1: Thread states.

The *one-to-one* model maps each user thread to a kernel thread. As [Silberschatz et al., 2000] says: *It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.* It also allows multiple threads to run in parallel on multiprocessors. The drawback, also according to [Silberschatz et al., 2000], to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system, for example Windows NT and OS/2.

Other systems like Solaris, IRIX, and Digital UNIX implement a model that suffers from neither of the shortcommings described till now. The *many-to-many* model consists in mapping user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may in fact vary for either a particular application or a particular machine (an application may allocate more kernel threads on a multiprocessor than on a uniprocessor).

2.5 Limitations

To quote [Silberschatz et al., 2000]: *"a thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing, economy, and the ability to take advantage of multiprocessor architectures."*

However, although it is quite practical to have multiple threads with a single CPU or a multiprocessor system, with user-space threads, there is no automatic time sharing [Dowd, 1993], which is useful when threads all want to perform simultaneous CPU-intensive computations. To have automatic time sharing, threads need to be created, managed, and scheduled by the operating system rather than a user-space library.

When the operating system supports multiple threads per process, we can begin to use these threads to do simultaneous computational activity. There is still no requirement that these applications be executed on a multiprocessor system. When an application that uses four operating system threads is executed on a single processor machine, the threads execute in a time-shared fashion. If there is no other load on the system, each thread gets 1/4 of the processor. While there are good reasons to have more threads than processors for non computational heavy applications, it's not efficient to have more active threads than processors for computer-intensive ones because of thread-switching overhead.

With operating-system threads and multiple processors, a program can realistically break up a large computation between several independent threads and compute the solution more quickly. Of course this presupposes that the computation could be done in parallel in the first place.

2.6 Multi-Threaded Prolog Example

In this section we will look into a complete multi-threaded program in SWI-Prolog as an example of a multi-threaded application in Prolog.

The program we propose is an implementation for the dining philosophers problem.

According to Jim Plank and Rich Wolski [Plank and Wolski]:

" The dining philosophers problem is a classical synchronization problem. Taken at face value, it seems like a meaningless problem, but it is typical of many synchronization problems that are seen for example when allocating resources in operating systems.[..] The problem is roughly defined as follows: There are 5 philosophers sitting at a round table. Between each adjacent pair of philosophers is a chopstick. In other words, there are five chopsticks. Each philosopher does two things: think and eat. The philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, he/she cannot eat until he/she owns the chopsticks to his/her left and right. When the philosopher is done eating he/she puts down the chopsticks and begins thinking again."

There are several solutions proposed to solve this problem. In the example that will follow we use the easiest one, that consists on having each chopstick be a monitor (mutex), and each philosopher will attempt to pick up the chopstick on his left first, then right, then eat, then put down the right one, and then put down the left one.

The only time that this solution is a problem is if a philosopher's thread gets preempted between picking up the first and the second mutex. For sake of simplicity we have left that case out.

The example was tested to work with SWI-Prolog 5.2.13 multithreaded. One example call could be `init(100)`. The code is presented on Appendix B.

The limitation of this program is that it can only run on a single-machine. Although SWI-Prolog supports basic interaction with the underlying operating system that could be used to implement a distributed multi-threading Prolog system, it would be difficult to support important aspects of parallel and distributed programming, such as portability and fault-tolerance, without decreasing the high-level of Prolog predicates.

Chapter 3

Parallel Systems and Logic Programming

By default, statements inside a computer program execute sequentially, one at a time, one after the other. The goal of multi-threading and parallel programming is to execute a program faster by working around this limitation.

Concurrent programming originated in the 1960s within the field of operating systems. Creating device controllers that operated independently of a controlling processor and allowed an I/O operation to be carried out concurrently with continued execution of program instructions, required that parts of a program could execute in unpredictable order.

The recent several years have witnessed an ever-increasing acceptance and adoption of such systems, both for high-performance scientific computing and for more general purpose applications.

Such systems range from a few hosts to thousands of CPUs and offer enormous computational power that is used for problems such as global climate modelling and drug design.

3.1 Logic Programming

Logic programming differs from other programming languages because problems and algorithms are expressed by using logic instead of constructing sequences of actions that manipulate mutable state information.

A case for using Prolog is presented below:

- An algorithm can be thought as being a combination of logic and control. Due to its logic nature, most control in Prolog is ommitted, given more chance to concentrate on the problem at hand rather than on the behavior of the program.
- Unlike imperative programs, which have only a procedural interpretation, logic programs also have a declarative, logical interpretation, which helps to ensure their correctness.
- Scoping rules are simple and uniform in Prolog, and declaration of variable names and types is not required, thus reducing code size.
- Prolog is a general-purpose programming language with efficient implementations available on most computing platforms today.

[McCarthy, 1959] was the first to publish a proposal that mathematical logic could be used for programming, but it was not till 1972 that Prolog, the still only one widely available language of its kind, was developed.

It was first introduced for natural language processing in French, but it has since then been used for specifying algorithms, searching databases, writing compilers, building expert systems and many other kinds of applications. Prolog is especially suited for applications involving pattern matching, backtrack searching, or incomplete information. A historical perspective of the development of Prolog can be found in [Colmerauer and Roussel, 1996].

As with any programming language, Prolog arguably also has some limitations. Some of them are of technical and others are more of a social nature, such as:

- Non-logical predicates (e.g. writeln/1, findall/3, !/0) reduce the inherent logic of programs because they don't have a direct logic meaning and some have side-effects (output text or modifying the database).
- Steps may be repeatedly derived by the theorem-prover, being possibly redundant.
- The modules system varies from Prolog system to Prolog system and without portable libraries, users are easily trapped into a single Prolog implementation.
- To quote Paulo Moura [Moura, 2006] the *"lack of code sharing means that Prolog programmers cannot bootstrap their applications without being trapped in some proprietary implementation, even for basic tasks. Lack of libraries and bindings for popular APIs makes choosing Prolog as an industrial tool a risky proposition"*, even when the advantages when compared with other languages may seem in favor of Prolog.
- Also, Paulo Moura [Moura, 2006] adds that *"Broad sharing of Prolog code between implementations suffers from both weak ISO Prolog standards and lack of knowledge of the current standards by Prolog programmers."*.

It is possible to overcome some of these limitations, for example, by combining Prolog with other languages. Combining Prolog and C is a trivial task using the foreign language interface that most Prolog implementations offer.

SWI-Prolog and GNU Prolog, for example, offer such interface in which a foreign predicate is a C-function that has the same number of arguments as the predicate represented. C-functions are provided to analyse the passed terms, convert them to basic C-types as well as to instantiate arguments using unification.

Non-deterministic foreign predicates are also supported, providing the foreign function with a handle to control backtracking.

On the C side, it is also possible to call Prolog predicates, providing both an query interface and an interface to extract multiple solutions from an non-deterministic Prolog predicate. For SWI-Prolog, according to [Wielemaker, 1997]: *"there is no limit to the nesting of Prolog calling C, calling Prolog, etc. and it is possible to write the 'main' in C and use Prolog as an embedded logical engine"*.

It is also possible to combine Prolog and other languages besides C. Such an example is the SWI module for Perl programming developed by Robert Barta [Wielmaker, 2000] or one of the libraries currently available to combine Prolog and Java [Calejo, 2001] or P# [Cook, 2001], that allows interoperation between Prolog and C#.

3.2 Parallel Architectures

In a network of computers, each computer (*node*) has fast access to its own local resources, including memory. To access the memory of other nodes, requests have to be made over the network (*distributed memory architectures*). On top of these mechanisms, using the VM's paging system, each node may have access to a large shared memory in addition to its own private memory. Such an organization is called DSM (*distributed shared memory architecture*).

Each node in a network may of course be a *shared memory* multiprocessor, where processors share access to a common memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data. Typically, the number of processors used in shared memory architectures is limited due to the amount of data that can be

processed by the bandwidth of the memory bus connecting the processors.

Several modern parallel computers use a mixed shared/distributed memory architecture. Each node consists of a group of 2 to N processors connected via local shared memory and in turn, those nodes are connected via a high-speed network.

Although parallel architectures are mostly designed with the goal of solving problems too big for any single supercomputer CPU, they also permit sharing other resources, e.g. storage, which are increasingly needed.

3.3 Implicit and Explicit Parallelism

In order to explore the potencial provided by parallel systems, research in logic programming has developed along two major strategies [Gupta et al., 2001, Wielemaker, 2003].

The first approach relies on *explicit* parallelism, where message passing primitives are added to Prolog for concurrency or by modifying the semantics of the logic programming language in a suitable way (Delta Prolog [Pereira et al., 1986] and CS-Prolog [Futo, 1993]). Systems that rely on this approach usually run multiple Prolog processes in parallel and can be classified as follows:

- those that add explicit message passing primitives to Prolog;
- those that add blackboard primitives to Prolog;
- those based on guards and data-flow synchronization.

Another approach exploits *implicit* parallelism in logic programs. This

means parallelization of the execution can (potentially) occur without any input from the programmer.

Two main forms of *implicit parallelism* have been explored in logic programming [Gupta et al., 2001, Wielemaker, 2003]. *And-Parallelism* is based on the parallel evaluation of the various goals in the body of a clause. This form of parallelism is usually further subdivided into *Independent And-Parallelism* in which the goals are independent, that is, they do not share variables, and *Dependent And-Parallelism*, in which goals may share variables. In contrast, *Or-Parallelism* corresponds to the parallel execution of alternative clauses for a given predicate goal. Also there are approaches where both and- and or-parallelism are exploited [Costa et al., 1991b].

The next figure illustrates the several strategies for exploring parallelism in logic programming:

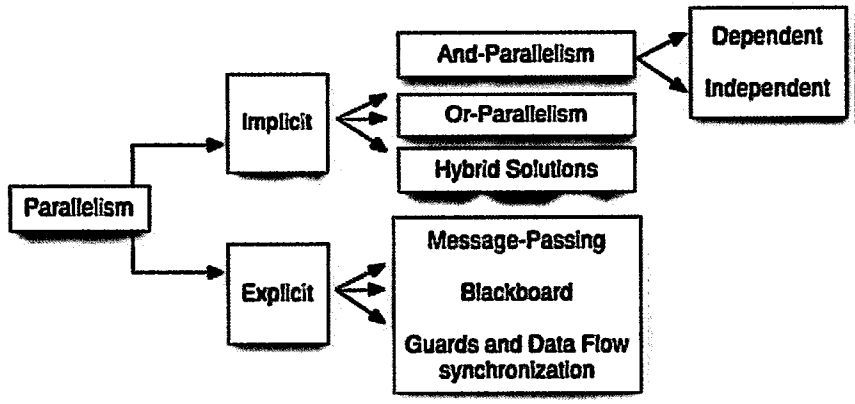

Figure 3.3.1: Strategies for exploiting parallelism in logic programming.

Although implicit parallelism is appealing because the sequential programming model is retained, providing the ability to parallelize legacy code, there is a limit on how much parallelism can be found, and current techniques only work on certain kinds of programs.

Implicit parallelism can be applied in many cases, but current techniques can't achieve maximum parallel potential for all problems. This seems an opportunity to further explore explicit parallelism and increase the range of problems that can be solved using parallel logic programming.

Since explicit and implicit approaches are complementary they can appear together in a single programming language. Such hybrid approach is exemplified, for instance, by the &-Prolog system [Hermenegildo and Greene, 1991].

3.3.1 Message Passing Libraries

One of the basic methods of explicit parallelism is the use of message passing libraries. These libraries manage transfer of data between instances of a parallel program running (usually) on multiple processors in a parallel computing architecture.

The Message Passing Interface (MPI) is the de facto standard[1] for computer program communication in High Performance Computing (HPC) environments. There are many implementations of the MPI standard, created by different groups in industry, academia, and government labs. MPI allows for the coordination of a program running as multiple processes in a distributed memory environment, yet it is flexible enough to be used in a shared memory system as well.

Recently, a merge between three well-known MPI implementations resulted in Open MPI [Gabriel et al., 2004]. These implementations are FT-MPI [Fagg et al., 2003] from the University of Tennessee, LA-MPI [Graham et al., 2003] from Los Alamos National Laboratory and LAM/MPI [Squyres and Lumsdaine,

[1]The MPI standard is comprised of 2 documents: MPI-1 (published in 1994) and MPI-2 (published in 1996). MPI-2 is, for the most part, additions and extensions to the original MPI-1 specification. The MPI-1 and MPI-2 documents can be downloaded from the official MPI Forum web site: http://www.mpi-forum.org/.

2003] from Indiana University. The PACX-MPI team at the University of Stuttgart is also collaborating as part of the Open MPI as well as major companies like Cisco Systems, IBM and Sun Microsystems. Each of the MPI implementations mentioned earlier excelled in one or more areas and the Open MPI effort effectively contains the union of features from each of the previous MPI projects [Graham et al., 2006, Angskun et al., 2006a, Keller et al., 2006, Angskun et al., 2006b, Hoefler et al., 2006].

3.4 Blocking and Non-Blocking Communication

The network communication between two computers can be implemented by using blocking or non-blocking primitives. Blocking primitives, also called synchronous, are those that do not return from the subroutine call until the operation has actually completed. Thus, it ensures that the relevant completion criteria have been satisfied before the calling process is allowed to proceed.

With a blocking "send", for example, the variables sent can safely be overwritten on the sending process. With a blocking "receive", the data has actually arrived and is ready for use.

A non-blocking (or asynchronous) "send" or "receive" returns immediately, with no information about whether the completion criteria has been satisfied. According to [Team RUNTIME, 2001], this has the *"advantage that the processor is free to do other things while the communication proceeds in the background. Tests can be made later to check whether the operation has actually completed"*.

For example, a non-blocking "send" returns immediately, although the operation will not be complete until the reception of the message has been acknowledged. The sending process can then do other useful work, testing later to see if the operation is complete. Until then, however, it can not be assumed that the message has been received or that the variables to be sent may be safely overwritten.

3.5 Problem Decomposition

An approach to design a parallel algorithm is to decompose the problem into smaller tasks. These can then be assigned to processors which will work simultaneously, with some coordination. This decomposition can be made focusing on the problem domain or on the functions used to solve the problem. It is important to distinguish between these two techniques.

In **domain decomposition** the program input is divided into smaller inputs of approximately the same size and then mapped to different processors. Each processor works only on the portion of the input that is assigned to it. Of course, the processes may need to communicate in order to exchange data.

The domain decomposition strategy is usually not very efficient because the data assigned to the different processes may require different lenghts of time to process, making the performance of the program limited by the speed of the slowest process.

In such cases, a more efficient strategy will be to use **functional decomposition** or "task parallelism". This approach consists of parallelizing what is commonly called a **task** or a **job**, that can be identified as a piece of code that is independent and that can be seen as a function, having an input, some processing

time working and an output. The tasks are assigned to the processors as they become available, and processors that finish quickly are assigned more work.

In both scenarios it is also necessary to consider that an overhead in terms of time exists when we are talking about parallel computations. This is due to task coordination and can include factors such as:

- Task start-up time;
- Synchronizations;
- Data communications;
- Software overhead imposed by parallel compilers, libraries, tools, operating system, etc;
- Task termination time.

On the conducted work, we try to have a small glance on the results that can be achieved from exploring Logic programming with both these approaches.

3.6 PM2 - Parallel Multithreaded Machine

PM2 [Namyst and Méhaut, 1996] is a distributed multi-threading programming environment designed to support irregular parallel applications on distributed architectures. A problem/algorithm is considered to be irregular if it involves pointers, such as in algorithms on trees or graphs.

For thread management it uses *Marcel*, a user-level multi-threading library with support for several different platforms. The communication between

threads is made using *Madeleine*, the communication library of PM2, that is available on top of various network hardware such as Myrinet, SCI, Ethernet or VIA and runs on the following architectures: Linux/IA32, Linux/Alpha, Linux/Sparc, Linux/PowerPC, Solaris/Sparc, Solaris/IA32, AIX/PowerPC, WindowsNT/IA32.

PM2 adheres to the SPMD (*Single Program Multiple Data*) programming model, in a way very similar to the PVM [Sunderam, 1990] and MPI [Graham et al., 2006] communication libraries. The user writes a single program text, a copy of which is launched by a specific `load` command on each *processing node* of the current configuration. Then, it is up to this common program text to include branching so as to differentiate between the processing nodes, based on a programming scheme exemplified by **Program 1**.

Program 1 Branching to differentiate between processing nodes in PM2.

```
if(pm2_self() == 0) { /* Do something.  .. */ }
else {  /* Do something else... */ }
```

This approach has the advantage of providing a single flow of control. At this level of presentation, a processing node is simply a Unix process. The association between *processing nodes* and *physical nodes* is made by the `pm2conf` command, which defines a static configuration that will be used next time an execution is requested.

3.7 Related Parallel Prolog Systems

Researchers in the field of logic programming have long realized the potential for the exploitation of parallelism present in the execution of logic programs as can be witnessed by the significant number of systems that were implemented with this goal in mind.

Examples of the multitude of such systems are Aurora [Lusk et al., 1988], Muse [Ali and Karlsson, 1990], &-Prolog [Hermenegildo and Greene, 1991], DDAS [Shen, 1992], Andorra-I [Costa et al., 1991a,b], Parlog [Clark and Gregory, 1986], GHC [Ueda, 1985], KL/1 [Ueda and Chikayama, 1990], YapOr [Rocha et al., 1999] to name a few.

In the context of the present work, particular focus is first taken on systems that exploit explicit parallelism based on message passing and then on Prolog systems that support multi-threading. We chose only a few systems that we think are representative enough of the state-of-the-art in the context of parallel and distributed logic programming.

3.7.1 Message Passing

Both Delta Prolog [Pereira et al., 1986] and CS-Prolog [Futo, 1993] present a system where multiple Prolog engines are mapped to processes that are running in parallel and communicate with each other via explicit message passing. These implementations were the first systems that exploited explicit parallelism based on message passing for Prolog.

PVM-Prolog [Marques and Cunha, 1996] introduced a programming interface to the PVM system where multiple distributed Prolog processes cooperate using a message passing model. This is very close to what is presented since PVM is itself very similar to PM2, one fundamental difference being that PM2 can itself sustain more than one entity of the application on each node.

3.7.2 Multi-threading

With respect to multi-threading, Prolog systems commonly offer implementations based on the POSIX threads (pthread) API [Butenhof, 1997]. This is exemplified by Qu-Prolog [Clark et al., 2001], SICStus MT [Eskilson and Carlsson, 1998], IC-Prolog II [Chu, 1994], PMS-Prolog [Wise, 1993] and more recently by SWI-Prolog [Wielemaker, 2003].

In these implementations each Prolog thread is normally a POSIX thread running a Prolog engine and threads communicate among each other either by using FIFO message queues or a blackboard system (an area of shared memory).

3.7.3 Assertions

Regarding assertions, for example in SWI-Prolog, according to [Wielemaker, 2003]: *by default, all predicates, both static and dynamic, are shared between all threads.* Changes to static predicates only influence the test-edit-reload cycle. As for dynamic predicates, a goal uses the predicate with the clause set as found when the goal was started, regardless of whether clauses are asserted or retracted by the calling thread or another thread ([Wielemaker, 2003]). Thread-local predicates are dynamic predicates that have a different set of clauses in each thread. Modifications to such predicates using assert/1 or retract/1 are only visible from the thread that performs the modification ([Wielemaker, 2003]).

3.7.4 Synchronisation

For synchronisation, in most cases, for example between threads in SWI-Prolog, mutexes are used.

3.8 Summary

According to [Silberschatz et al., 2000], we saw that *"a thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing, economy, and the ability to take advantage of multiprocessor architectures."*

Modern parallel computers use a mixed shared/distributed memory architecture. Each node consists of a group of 2 up to N processors connected via local shared memory and in turn, those nodes are connected via a high-speed network.

PM2 is a distributed multithreaded programming environment, that adheres to the SPMD (*Single Program Multiple Data*) programming model, in a way very similar to the PVM and MPI communication libraries. The user writes a single program text, a copy of which is launched by a specific command on each node of the current configuration.

Most Prolog implementations offer libraries and modules to support basic interaction with the underlying operating system, such as access to the file-system, forking a process or TCP/IP communication.

Although these libraries could be used to implement a distributed multithreading Prolog system, they usually offer low-level routines that are difficult to work with to support important aspects of parallel and distributed programming, such as portability and fault-tolerance.

These aspects are supported by distributed programming environments like PVM and PM2, that already account for such problems but rarely offer access to Prolog.

An integration between Prolog and PM2 will allow programmers to exploit

parallelism and develop distributed multi-threaded Prolog applications.

Chapter 4

PM2-Prolog

PM2-Prolog is a system that allows the development of distributed multithreaded Prolog applications, using GNU Prolog and PM2.

Since PM2 programs are developed in C and GNU Prolog is a Prolog engine and compiler, it was necessary to estabilish a model for connecting the two programming environments.

The approach used doesn't involve modifications in GNU Prolog neither modifications in PM2. Instead, it relies on:

- a new program (Tabard[1]), written in C with the PM2 libraries, that manages distributed instances of gprolog engines, and that is transparent for the end-user.
- a new Prolog library (pm2prolog-lib), implemented partly in C and partly in Prolog, which allows the development of distributed multithreaded Prolog applications.

[1]Literally, a tabard is a short coat, that in the late middle ages was worn by knights over their armour. This fits nicely as a name because Tabard will allow "wearing" a Prolog program over PM2 (the armour).

When using a PM2-Prolog program we first generate the binary that results from the compilation of our Prolog program linked with Tabard, the libraries of PM2 and the libraries of GNU Prolog. To ease this compilation task there is a Makefile available in Appendix A.

Before compiling there is a configuration that specifies the list of machines on which the application is going to run. That configurations maps one or more *processing nodes* or *virtual processors* (VPs) to each machine. While it may seem common sense practice to use one virtual processor per physical node, nothing in PM2-Prolog requires such association, as we will see in detail later.

Then, the binary is executed via the PM2 command `pm2load`, e.g.:

```
$ pm2load helloworld
Hello World
```

This command starts the following execution model:

1. The binary is copied to all the machines. The `main()` function of Tabard is called on every VP.
2. In VP 0 (master) a gprolog engine is created calling `Start_Prolog()`, that will start executing the linked Prolog code.
3. In the other VPs (workers) a pthread in C is created and stands awaiting messages. This is done via a call to a blocking read Madeleine function.
4. In the master, now in the Prolog thread, a predicate is called to send a message to every worker, ordering the starting of a Prolog thread by calling `Start_Prolog()`.
5. The workers receive that message, initiate a gprolog engine and the new Prolog thread stands awaiting more messages to come by calling a blocking read pm2prolog-lib predicate. At this time, there are two threads awaiting messages, one in C and another in Prolog, for each worker.

6. In the master, work is distributed throughout the workers through message-passing.
7. The workers receive tasks which they execute locally. As soon as they finish, they send their results back to the master and return to their prior state, awaiting for messages.
8. The master assembles the work results by reading as many messages as the number of previously sent messages.
9. The master redistributes work again (5.) or orders the workers to finish their execution.
10. The workers terminate.
11. The master reiniciates the workers (4.) or terminates itself.

Some of the aspects described in the above model are transparent for the end-user and some require explicit use or the program may not work correctly in the distributed environment.

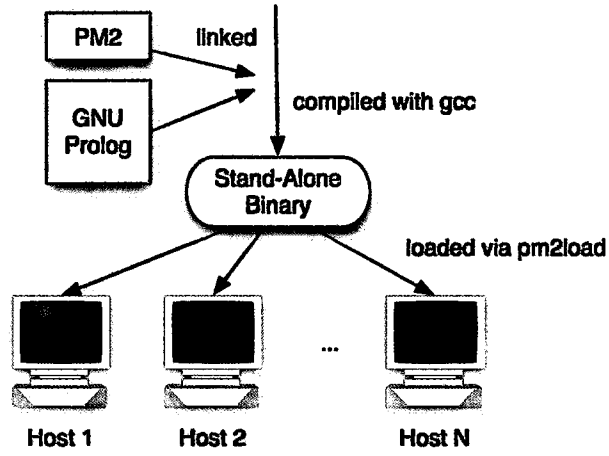

Figure 4.0.1: PM2-Prolog copies the program into each configured host and executes.

A general picture of the execution model as seen by the end-user, independently of the Prolog program itself, is given in above figure.

4.1 Threads in PM2-Prolog

A thread is most commonly described as a flow of execution. A computer program has commonly only one flow of execution, that goes from its first instruction to the last, not necessarily in sequential order. In Program 2 we can see an example of a Prolog program that counts the occurrences of a number in a list. This program, as most computer programs, has only one flow of execution.

Program 2 A Prolog program to count the occurrences of a number in a list.

```
1  :- initialization(init).
2
3  init:-  n_occurs(2, [23, 35, 2, 6, 43, 2, 9], O),
4          write(O),nl,halt.
5
6  % n_occurs(+X, +List, -N)
7  n_occurs(X, L, N):- n_occurs(X, L, 0, N).
8
9  n_occurs(_, [], Acc, Acc):- !.
10 n_occurs(X, [Y|Ys], Acc, N):-
11         X \= Y,
12         n_occurs(X, Ys, Acc, N).
13
14 n_occurs(X, [X|Xs], Acc, N):-
15         Acc1 is Acc + 1,
16         n_occurs(X, Xs, Acc1, N).
```

If we were to execute Program 2 in a parallel computer following the SPMD paradigm, the `init` predicate, where the program starts, would be called on all computers or processing nodes thus originating N equal threads of execution running concurrently. The program would still have a single flow of control but now that flow would be executed once in each node. The list could then be splitted across the processing node which would work with a smaller search-space than before.

In PM2-Prolog each machine in the configuration will have a C thread (*listener*) and a Prolog thread, for each VP. The purpose of the C thread is to control the associated gprolog engine that runs in the Prolog thread, in terms of

creation, termination, monitoring, etc.

Since GNU Prolog doesn't support multi-threading, PM2-Prolog novelty is that it allows to control more than one gprolog thread in the same machine without introducing changes in GNU Prolog itself.

Also, with this approach, we transparently support all predicates of the GNU Prolog libraries, which would not happen if we were to modify the gprolog engine to support multi-threading in which case many predicates would require modifications.

In summary, we achieve a multi-threading that is very appellative from a technical point of view but that for each Prolog thread has an attached C thread. However, once GNU Prolog introduces support for multi-threading it is trivial to change to an architecture where there is only one C thread by machine that controls N gprolog threads.

4.1.1 Task-Farming in PM2-Prolog

Branching is crucial in PM2-Prolog since it allows to differentiate between the different threads and execute different things in each one to our benefit. The case described earlier, in which we have a *master* thread, that distributes tasks to be done by the workers is known as *task farming* and is a commonly used way of parallelizing applications. Also, as described earlier, the communication between master and workers occurs by message-passing, as illustrated on the figure 4.1.1.

The master thread and the workers execute in a VP and what happens inside each VP deserves a closer look. A VP is a Unix process that receives a unique *rank* number.

This rank is an `unsigned int` between 0 and `pm2_max_rank/1`, the configu-

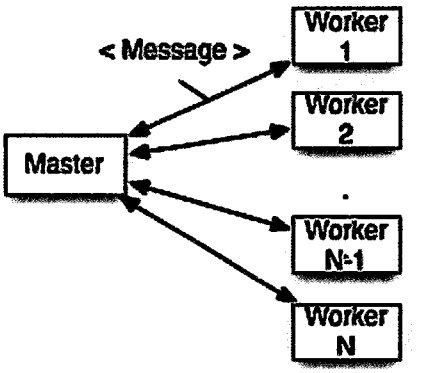

Figure 4.1.1: Task farming strategy to parallelize a program.

ration size. A VP can learn about its own rank by calling `pm2_self/1`.

4.1.2 Listener Thread

Processing nodes are able to execute code and simultaneously check if any messages arrives. The listener thread receives commands or orders in form of messages that can result in different actions being carried out on a specific VP, such as creating another thread or execute specific code.

Two important messages were specified and implemented: 1) create a Prolog thread and 2) terminate the listener thread. All other messages that arrive at a VP will be interpreted not as commands, but as common messages that must be delivered to the running Prolog thread inside that VP. A mechanism of quoting to enable passing messages equivalent to these command is not yet implemented, but is being thought of.

As can be observed on the above figure, the listener thread and the Prolog thread communicate using a shared data structure. The listener delivers messages by writing them to a *message queue* and the Prolog thread accesses them by reading

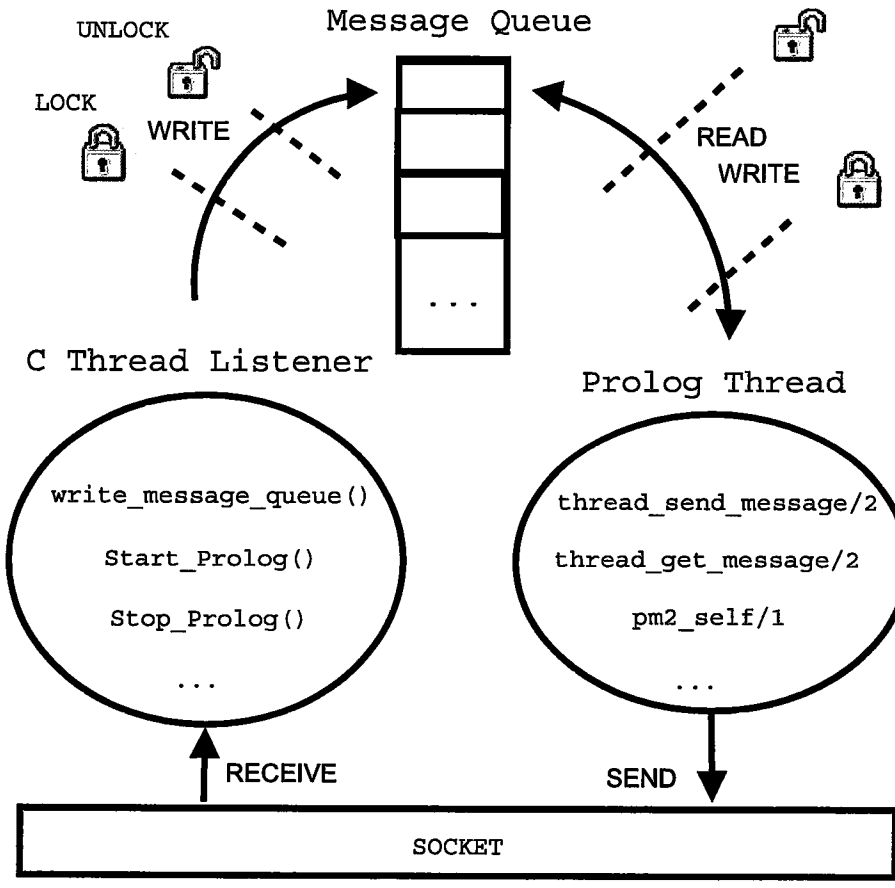

Figure 4.1.2: Inside a processing node or virtual processor.

in *First In, First Out* (LIFO) order from that structure. The message queues provide a means for threads to wait for data without using the CPU. Other means to do this, like checking via a polling loop, would cause *busy-waiting*, that generally should be avoided.

The listener thread receives messages from a socket and the Prolog thread sends messages via its listener "support" thread.

The Prolog thread can also write to its message queue in the special case where the destination VP is the same as the sender.

4.2 Communication Scheme

In terms of communication, the Madeleine layer [Aumage, 2002] provides an API that is similar to POSIX socket. Around this API we've implemented primitives for sending and receiving Prolog terms over the network. As with Madeleine, it is also not possible in PM2-Prolog to know the source address when a message is received. Complementary, we observe that most applications don't have this requirement. That doesn't mean we will only communicate between master and workers, because we can configure any host which is network accessible.

To implement the routines we needed to be able to convert a Prolog term into a C string and also be capable of doing the opposite process. This is required since the Madeleine routines receive a (char *) buffer as an argument.

The approach used consists in transforming the term to a character code list by using the built-in gprolog predicate `writeq_to_codes/2`. This is similar to `write/2` except that characters are not written into a text-stream but are collected as a character code list which is then unified with the first argument.

By using the foreign type `term` a C string will be ready to be sent. On the reception side, once the string is read we use `Mk_Codes()` to convert it again to a character code list and `read_term_from_codes/3` to transform it back into the original Prolog term.

Threads living on distinct nodes may not directly interact together unless by message-passing. When this happens the listener thread receives the message and is responsible for delivering it to the correct local thread.

4.3 Thread Management

Prolog threads are created by calling `marcel_create()`, that is part of the Marcel [Namyst and Méhaut, 1995] library . In terms of managing threads what Marcel offers to the programmer is similar to pthreads, and so in terms of structure the resulting thread can be considered as identical to a regular pthread. The thread starts by executing the function given as argument to `marcel_create()`. In this case, this is `Start_Prolog()`, which initializes and starts the Prolog engine.

The mutexes provided by the Marcel API are used to make the operations on the message queue thread-safe. Their behaviour is also similar to the ones offered by the pthread API.

One thing that also needs to be guaranteed in the message queues is that no initial `read` is made before `write`. For that purpose another mutex has been used, as follows:

1. The mutex is initialized and a lock is made.
2. Some VP send a message. The listener thread receives it, writes it to the message queue and an unlock is made.
3. Since an unlock has been made, the Prolog thread will now acquire the lock and retrieve the message from the queue. Finally, a lock is made and we go back to step 1, starting over again when a new message arrives.

4.4 Programming Model

When developing a PM2-Prolog program, three aspects were taken into account:

1. The SPMD (*Single Program Multiple Data*) programming model.

2. The task-farming model.
3. The message-passing model.

The user writes a single program text, a copy of which is launched by a specific `load` command on each VP. At this point the programmer typically includes branching to differentiate between the different VP and execute different things in each one. One predicate that is included with PM2-Prolog that can help distinguish if we are in the master node (rank equal to zero) or in another node is `pm2_is_master/0`. The following program introduces the use of this and some other predicates introduced by PM2-Prolog's API:

Program 3 Program to identify the Master and Worker threads.

```
1  :- include(lib).
2  :- initialization(init).
3
4  % thread rank = 0
5  init:-
6          pm2_is_master,!
7          pm2_max_rank(MaxRank),
8          start_prolog_workers(MaxRank),
9
10         write('Master thread'),nl,
11         pm2_self(Rank),
12         write('My rank is '),write(Rank),nl,
13
14         stop_prolog_workers(MaxRank),
15         finish_listeners.
16
17 % thread rank != 0
18 init:-
19         write('Worker thread.'),nl,
20         pm2_self(Rank),
21         write('My rank is '),write(Rank),nl.
22
```

The predicate `pm2_max_rank/1` unifies `MaxRank` with the highest rank of the configuration. `pm2_self/1` unifies the rank of the current VP.

The second concern has to do with the task-farming model. Let's continue with Program 2 and now consider a predicate that can be used for distributing

tasks by the workers. We assume that we want to look up the number of occurrences of a number on a set of lists. We send the list and the number to look up over the network and propose the following program:

Program 4 Scheduling work to be done by the workers.

```
1  % all work scheduled(L):- empty(L).
2  send_worker_job([],_,_).
3
4  % round-robin the workers
5  send_worker_job(L, Element, 0):-
6          pm2_max_rank(MaxRank), % obtain the rank of the hightest worker
7          send_worker_job(L, Element, MaxRank). % start again
8
9  send_worker_job([L|Ls], Element, Rank):-
10         thread_send_message(vid(Rank,0), query(Element,L)),
11         NextRank is Rank - 1,
12         send_worker_job(Ls, Element, NextRank).
```

We loop through the workers and when we reach rank zero we call `pm2_max_rank/1` and start over, in a round-robin scheme. The predicate *thread_send_message/2* is also used here, which first argument is a compound term named `vid` indicating the destination `rank` and the thread `id` inside that rank.

The first argument is a list of the existing worker ranks, the second is the element for which the program will trigger a message if it finds it, and the third is the current rank of the iterative process.

4.4.1 Dealing with Message-Passing

Here we briefly analyse, in terms of Prolog, the impact of doing computations based on a message-passing model on program development.

A computation in Prolog is always a process of production of bindings, known as unification.

This unification consists in binding a variable to a value, the scope of which is local to the Prolog process and not visible or accessible to the outside. The binding is lost when the variable is referred in a remote node. Via message-passing, we propose recovering the binding of a variable by reading it from another message that the remote node should sent. As an example, consider the Prolog predicate in Program 5 to be in single-threaded mode.

Program 5 A query call that would presumably unify the variable X.

```
% pred(+A, +B, -C)
pred([1,2], [[2,2], [3,2]], X).
```

The same query, in a multi-threaded application with message-passing, would have to look as Program 6.

Program 6 Master side.

```
thread_send_message(worker_id, pred([1,2], [[2,2], [3,2]])),
thread_get_message(X).
```

And in the worker thread side as Program 7.

Program 7 Worker side.

```
thread_get_message(Pred),
(do processing)
thread_send_message(master, Result).
```

An example `Makefile` that can be used for compiling such a program can be found in Appendix A.

4.5 API

We now present the PM2-Prolog prototype API. Extending PM2-Prolog is similar to extending any GNU Prolog program. New Prolog predicates or new C

functions can be added using the foreign interface.

4.5.1 PM2 Facilities

pm2_self(-*Rank*)

Unifies with the rank number of the processing node, a unique integer number assigned to each machine.

pm2_is_master/0

Will succeed when the rank where it is being called is zero, usually the thread that distributed work.

pm2_max_rank(-*Rank*)

Unifies with the highest rank number of the configuration.

finish_listeners/0

Terminates the listeners threads in each VP. Called upon termination.

4.5.2 Creating and destroying Prolog threads

start_prolog_workers(+*HighestRank*)

Start the Prolog thread in each VP.

stop_prolog_workers(+*HighestRank*)

Stop the Prolog thread in each VP.

4.5.3 Thread Communication

thread_send_message(+*ThreadId*, +*Term*)

Send a message to the thread *ThreadId* with content *Term*. This predicate is non-blocking, meaning it will return immediately after being called. Also, a variable looses any binding it might have when sent to another thread.

Since each thread has by default its own message queue the other threads will be unaffected by this call.

ThreadId is a compound term of the form vid(Rank, Id).

thread_get_message(-*Term*)

Fetches a message from the message queue of the current thread. This predicate is based on a blocking read, meaning the execution will block, if necessary, on this thread until a message can be retrieved. After being retrieved, the message is deleted from the message queue.

read_results(-*Number*)

Calls thread_get_message/1 a *Number* of times.

4.5.4 ISO Compatibility

Timely fulfilling modern software requirements is only possible through the extensive use of libraries. As a result, modern programmers spend a significant

portion of their time creating and using libraries.

The quality and broadness of the accompanying libraries of a particular programming language become an important part on its success, sometimes even more than the language intrinsic characteristics.

For this reason, is it important that broad sharing of these libraries between implementations exists, and that is only possible if ISO Prolog standards are defined and known by current Prolog programmers.

For Prolog systems wishing to implement multi-threading support predicates, there is a draft technical recommendation (DTR) for Prolog multi-threading support [Moura, 2007].

Compatability with the DTR should be reached when the following modifications are made:

- Rename `pm2_self(-Rank)` to `thread_self(-Rank)`;
- Implement `thread_create(@term, -thread, @options)`;
- Modify `start_prolog_workers(+Number)` in such a way that calling it is equivalent to calling `thread_create(@term, -thread, @options)` several times;
- Implement an alias that associates a message-queue with a distributed thread identifier.
- Modify `thread_send_message/2` in order to allow receiving a message-queue alias instead of a distributed thread identifier.

Program 8 PM2-Prolog example: Send a message to each worker thread and read each reply.

```
1  :- initialization(init).
2  :- include('lib').          % include the pm2 interface lib
3
4  % will run on thread with rank 0
5  init:-
6          pm2_is_master,!,
7          pm2_max_rank(MaxRank),    % unify with the highest rank
8
9          start_prolog_workers(MaxRank), % Start_Prolog on each node
10
11         test_prolog_workers(MaxRank), % send a msg to each node
12         read_test(MaxRank),           % read the response
13
14         finish_listeners.
15
16 % will run on thread with rank not 0
17 init:-
18         worker_work.
```

4.6 PM2-Prolog Users Guide

We can observe in the above example the generic structure of a PM2-Prolog program. The execution starts by calling the `init/0` predicate, as indicated by the `initialization/1` directive.

The predicate is called on all nodes and with `pm2_is_master/0` (line 6, Program 8) we distinguish what is executed in the master thread and what is worker code.

Right after this predicate and following the code that is executed on the thread with rank 0, `pm2_max_rank/1` (line 7, Program 8) is used to obtain in runtime the number of workers present in the configuration. This is useful for sending a message to each worker in a round-robin scheme, for example.

`start_prolog_workers/1` will create a new Prolog thread on each worker. This is done by sending a message to each remote listener thread that will order

the creation of a new pthread that will call Start_Prolog().

For sake of simplicity we have encapsulated the logic behind sending a message to each worker in the predicate `test_prolog_workers`. The code for this predicate is in Program 9.

Program 9 test_prolog_workers predicate.

```
1 test_prolog_workers(0):- !.
2 test_prolog_workers(VP):-
3   thread_send_message(vid(VP,0), hello),
4   VP1 is VP - 1,
5   test_prolog_workers(VP1).
```

The program argument is the rank of the highest worker rank, as obtained by calling *pm2_max_rank/1*. We send a message to each worker by decrementing this number until we reach zero (the master rank) on which we stop.

The *read_test/1* is then executed to read the workers response. It consists of the same logic we saw in Program 9 but for reading, i.e. we read from each worker by reading first from the highest rank and then decrementing the rank until we reach the master rank.

Program 10 read_test predicate.

```
1 read_test(0):- !.
2 read_test(VP):-
3   thread_get_message(X),
4   VP1 is VP - 1,
5   read_test(VP1).
```

The `worker_work/0` is the predicate called on each worker thread. It consists of a blocking *thread_get_message/1* predicate that will await for a message, process it, and send back the result back to the master thread. Our processing consists on calling *mytestgoal/2*, a fictional predicate as an example, with the first argument unified to the received message and with a second argument that can be unified or not according to the message.

Program 11 worker_work predicate.

```
1  worker_work:-
2          pm2_self(Rank),
3          thread_get_message(Term),
4          !, mytestgoal(Term, X),
5          write('Worker '),write(Rank),write(' -> '),
6          write(Term), write(','), write(X), nl,
7          thread_send_message(vid(0,0), X),
8          worker_work.
9
10 mytestgoal(hello, ok).
```

finish_listeners/0 will, also by sending a message, order the termination of the Prolog threads.

4.6.1 Compiling a PM2-Prolog Program

Before compiling a program we include the PM2-Prolog library by using the directive `include`, such as:

```
:- include(pm2prolog).
```

This directive assumes the presence of the file pm2prolog.pl on the current directory.

Then, we compile the Prolog program into a object file by using the GNU Prolog compiler *gplc*.

The result object file is then used on another compilation command, this time by issuing a `gcc` compiler command, in which we link the PM2-Prolog library, GNU Prolog and the PM2 libraries with our program.

The arguments for such operation must specify the *static* flag. This makes sure that the resulting program binary is a stand-alone that can be safely deployed

throughout the machines on the network. It must also assure that both the PM2 libraries and the gprolog libraries will be linked together in order to allow the program to access PM2 functions and gprolog predicates.

An generic example Makefile that can be used to compile a PM2-Prolog program can be found in Appendix A. The relevant part of such file however is presented below:

Program 12 Example usage of gcc(1) to compile a PM2-Prolog program.

```
tabard: gprolog-pm2.o tabard.o $(OBJECT_PL)
        gcc -static -o tabard
                $(PLL)/obj_begin.o
                $+
                -L$(PLL)
                -lbips_fd -lengine_fd -lbips_pl
                $(PLL)/obj_end.o
                $(LIBS)
                -lengine_pl -llinedit -lm
```

4.6.2 Configuring and Running

The compiled version of our program is automatically placed into the private *build* directory of the user. The final step before execution is to specify the list of hostnames on which the application is going to run. This is done via the `pm2conf` command.

For example, if the current machine is called `ravel`, and two neighboring ones are called `debussy` and `faure`, we can configure our application to run on the three hosts by executing:

```
ravel% pm2conf ravel debussy faure
The current PM2 configuration contains 3 host(s):
```

```
0: ravel
1: debussy
2: faure
```

Each processing node taking part of a given execution receives its own unique *rank* number. In this example, PM2 will consider that processing node 0 is a process run by `ravel`, node 1 by *debussy* and VP 2 by *faure.*

Loading and running the program is done by calling `pm2load`. For example, if Program 11 resulting binary would be called 'hello', then the loading would be done in the following way:

```
ravel% pm2load hello
Worker 1 -> hello
Worker 0 -> ok
Worker 2 -> hello
Worker 0 -> ok
```

We can see that our program generates four messages. The first one is the expected 'hello' message comming from `debussy`. The second one is the reply the processing node 0 received from one of nodes (we don't know which in this example), the third is the 'hello' comming from `faure` and the last one another reply that node 0 received. The order of these messages is irregular between different executions since there isn't any mechanism of synchronization in use.

Behind this simple logic there is a number of internal operations that are spawned and that take care of listening the network and answering to requests that are being made by the PM2-Prolog program.

The Unix standard input/output streams for example, are protected by a

lower code abstraction (at the PM2 layer) against race conditions that could occur from the multi-threading paradigm.

From [Team RUNTIME, 2001] we read that: *"The processing node with rank 0 has a particular status because it is the only one which input/output streams are linked to the terminal from which the application was launched. [..] As a consequence, only the main node of an application can access its standard input stream"*, for example, using `argument_list/1` and `argument_counter/2`.

While it may seem common sense practice to use exactly one virtual node per physical node, nothing in PM2-Prolog requires such association. For example, a valid configuration in which two virtual nodes per physical node exist is presented below:

```
ravel% pm2conf ravel ravel debussy debussy faure faure
The current PM2 configuration contains 6 host(s):
0: ravel
1: ravel
2: debussy
3: debussy
4: faure
5: faure
```

In this example, each hostname will host two VPs. All processes may even be started on the same machine. Also, there is no reason why this machine should be the one which we are logged in. Any machine on the configured network can be used.

4.7 PVM-Prolog vs PM2-Prolog

In this section we compare, in terms of usage, PVM-Prolog and PM2-Prolog. PVM [Sunderam, 1990] is a framework for parallel and distributed computing widely disseminated in the academic community.

Both PVM-Prolog and PM2-Prolog are tools, in the form of a code library, that help in the integration of Prolog with distributed and parallel system. Both approaches are of pragmatic character and have in mind the reach of a functional prototype rather than a fully complete system.

We begin by describing some key PVM-Prolog predicates and concepts that will be used later on the examples.

PVM-Prolog processes correspond to PVM tasks. The unit of parallelism in PVM is a task (often but not always a Unix process), an independent sequential thread of control that alternates between communication and computation.

PVM task identifiers are used to identify PVM-Prolog processes, as atom names. The predicate that allows a PVM task to determine its own unique PVM task identifier is:

pvm_mytid(*-tid*)

Additionally if the process is not already a PVM task, it becomes so. Another important predicate is *pvm_spawn*, that allows for the dynamic creation of new PVM-Prolog tasks.

pvm_spawn(*+progname, +goal, +opt_list, +where, +ntasks, -tid_list*)

In general, *ntasks* are created to solve the given *goal* in the presence of the specified program. *progname* is the name of the file containing the Prolog program and *opt_list* and *where* are PVM specific.

This predicate will spawn a PVM task for the execution of an instance of the Prolog engine (NanoProlog). The specified Prolog file will then be consulted and the specified top goal activated. The newly created process is completely detached from its parent. It is up to the user to control all intended interactions between father and child, e.g. to gather solutions.

For communication, PVM-Prolog offers the *pvm_send* predicate, that allows for sending messages to another task, while *pvm_mcast* allows multicasting of messages, by sending the same message to several recipients, that are identified by *tid_list*.

The API for these predicates follows:

```
pvm_send(+tid, +msgtag, +term)
pvm_mcast(+tid_list, +msgtag, +term)
```

4.7.1 Master/Worker Logic

PM2-Prolog has many parallels with PVM-Prolog. Both are designed primarily as an interface that allows for the development of parallel and distributed multi-threaded applications in Prolog. One fundamental difference is that with PVM-Prolog N tasks are spawned to execute on a virtual single large parallel computer. With PM2-Prolog the number of threads for each machine is controlled.

This is especially useful for taking advantage of SMP systems, where we can, for example, assign two threads for a dual processor machine or four threads to a host with four processors.

Master

Let us continue by translating some PVM-Prolog code into PM2-Prolog. The following predicate is for a simple prime number generator program:

Program 13 Example usage of PVM-Prolog.

```
1  init:-
2          pvm_mytid(Rank),
3          pvm_spawn(examplefile, worker_work, [], [], NWorkers, WorkersId),
4          make_first_primes(FirstPrimes),
5          pvm_mcast(WorkersID, 1, FirstPrimes),
6          read_results(NWorkers),
7          pvm_exit.
```

The equivalent in PM2-Prolog would be the program:

Program 14 Example usage of PM2-Prolog.

```
1  init:-
2          pm2_is_master,
3          pm2_self(Rank),
4          pm2_max_rank(NWorkers),
5          start_prolog_workers(NWorkers),
6          make_first_primes(FirstPrimes),
7          send_workers_loop(NWorkers, FirstPrimes),
8          read_results(NWorkers),
9          finish_listeners.
10
11 init:-
12         worker_work.
```

In PVM-Prolog we begin by initializing PVM, creating the workers and also initialize them. In PM2-Prolog we start by checking if the program is running on the master rank or on a worker by calling *pm2_is_master*. On the node with rank

zero this predicate will succeed. On the other nodes, the predicate *worker_work* will be called.

On the master, by calling *pm2_max_rank* the program will instantiate *NWorkers* with the number of workers present in the configuration. This is relevant for most predicates as it can be used, for example, to loop through each worker.

We see that PM2-Prolog formulation doesn't require specification on the number of threads that are to be created. Such configuration is done outside via the *pm2conf* command-line tool. One advantage of this approach is that it allows for control on how many threads are created on each computer. The drawback is that specification cannot be changed in run-time. Another advantage that could be argued is that configuration changes do not require program re-compilation.

Worker

In PVM-Prolog the worker starts by initializing the PVM environment, and proceeds to do a *repeat-fail* loop, waiting for work to do:

Program 15 Worker code in PVM-Prolog.

```
1  worker:-
2          pvm_setopt(route,routeDirect),
3          pvm_parent(PTID),
4          pvm_mytid(MTID),
5          pvm_recv(PTID, 1, FirstPrimes),
6          repeat,
7                  pvm_send(PTID, 1, more(MTID)),
8                  pvm_recv(PTID, 1, Order),
9                  worker_do(Order, FirstPrimes PTID),
10         fail.
```

In PM2-Prolog, the logic behind the worker code consists in having a call to *thread_get_message* which will perform a blocking read and never fail. The following program is the worker code equivalent in PM2-Prolog:

Program 16 Worker code in PM2-Prolog.

```
1 worker_work:-
2         thread_get_message(Term),
3         !, call_primes(Term, X),
4         thread_send_message(vid(0,0), X),
5         worker_work.
```

Once a message has arrived, *thread_get_message* returns and *call_primes* is called. After the result is unified with *X* it is send via a message to the master and this process is repeated for each message that arrives.

4.8 Summary

PM2-Prolog follows the *Single Program Multiple Data* programming model, that it inherits from the PM2 environment. The same program is executed in every node and actions are taking accordingly to the unique rank number that identifies each node. Commonly, the node with rank 0 distributes work by the nodes with higher ranks, in a task-farming scheme.

The creation of Prolog threads as well as the communication is totally based on explicit primitives. This means everything related to the parallelization is to be controlled by the programmer. This allows to have control over the parallel execution and to prevent as much as possible less-than-optimal parallel efficiency on programs.

Chapter 5

Performance Evaluation

5.1 Evaluation Model

To assess the suitability of PM2-Prolog for a particular purpose, many users will consider it's performance as the most important and indeed critical feature.

The environment on which PM2-Prolog can be used can widely vary. It can be a cluster of networked workstations or a set of workstations wide-spread throughout the Internet. As a matter of fact, these workstations need only to be running Linux and have ssh/rsh access, given that the machines used are of the same architecture.

Our study focuses on a cluster of SMP systems and the speedup that can be obtained from problems that consist of a large task and that can be split into subtasks distributed over a pool of threads.

The guiding principle in reporting performance measurements is reproducibility. A valid exercise of this type must allow others to repeat the benchmark and achieve similar results. We include information about both the hardware and

software environments used, and describe the studied problems and the execution model for each one.

By conducting the benchmark in this way we feel that the results, good or bad, are credible and valid as a basis for studying the applicability of running multiple Prolog engines in a distributed environment.

5.2 Measuring Performance

First of all, we have to define "performance". What interests us here is the elapsed real (*wall-clock*) time used by the process. It represents the total time needed to complete a task, including disk accesses, I/O activity, operating system overhead - everything. We obtain the wall-clock time with the Unix `time(1)` command (not the shell built-in `time` but the one normally found in /usr/bin/time) and with the format set to `elapsed time` only, specified with the parameter `-f %E`, e.g.:

```
/usr/bin/time -f%E ls
```

This time will be composed by the initialization time of the program (T_0) plus the time the program will spend actually doing some processing (T).

The initialization time is not constant. It grows along with the number of workers, while the processing time will decrease until it reaches a point where it is less than the initialization time and by then it no longer compensates to have more workers on the problem.

In order to compare the real processing time between configurations with different number of workers, the initialization time must be calculated and then subtracted from the obtained elapsed time.

The initialization time (T_0) is given by the elapsed time obtained for what is called the *empty problem*, which consists of no more than a program that initializes the system and exits.

Having these measurements we calculate the speedup (S) using the formula:

$$S = \frac{T_1 - T_1(0)}{T - T(0)}$$

where T_1 is the elapsed time obtained with M + 1 worker, $T_1(0)$ the elapsed time obtained with M + 1 worker for the "empty" problem, T the elapsed time obtained with M + N workers and $T(0)$ the elapsed time obtained with M + N workers for the "empty" problem.

5.2.1 Hardware Environment

The hardware used consisted of 7 units of the machine shown below:

Table 5.2.1: Hardware Environment (x7)

CPU	Intel(R) Pentium(R) 4 CPU 2.80GHz each
Hyper-Threading	Enabled
Cache size per CPU	512 Kb
FPU	Yes, integrated
Memory	512 Mg
Filesystem	IDE disk shared via NFS
Filesystem type	Ext2
Network	TCP/IP over Ethernet
Network interfaces	RealTek RTL8139 Fast Ethernet
Background load average	Minimum or none

Note that a Pentium 4 with Hyper-Threading enabled is treated by the

operating system as two processors. This means we have a maximum of 14 processors to work with.

5.2.2 Software Environment

We use a suite of three classic literature problems plus a real-world application to measure the speedup that can be obtained in each one by using PM2-Prolog. The software environment that serves as a basis for the measurement is described below:

Table 5.2.2: Software Environment

Operating System	Debian GNU/Linux kernel 2.6.17.1 SMP support
Prolog Compiler	GNU Prolog 1.2.18 Debian Version
Prolog Compiler options	–with-c-flags=-static –disable-regs
C compiler	gcc 2.95.4

5.3 Benchmark Programs

5.3.1 Parallel Matrix Multiplication

A fundamental numerical problem is the multiplication of two matrices. We use a simple O(N^3) algorithm to compute C = AB, where A, B, and C are NxN matrices. The algorithm follows directly from the definition of matrix multiplication. To compute C_{ij}, we compute the dot product of the i-th row in A with the jth column in B.

Hence we order each worker to compute C_{ij} and send the result back to the master where the final matrix is assembled.

The matrix lines are represented by Prolog lists and the columns are obtained using the nth/2 predicate. Of course, better algorithms exist for matrix multiplication, for example Strassen's algorithm, but our interest here is not to work smarter (e.g. with better algorithms) but to work harder, i.e. adding more processing capacity. We conducted tests with matrices of several sizes but the presented results refer to the multiplication of a 64x64 matrix executed fifty times.

Program 17 Matrix Multiplication: Rows are distributed for processing through the workers.

```
1  mult([], _, 0, _):- !.
2
3  mult(Rows, Matrix, N, 0):-
4    pm2_max_rank(MaxRank),
5    !, mult(Rows, Matrix, N, MaxRank).
6
7  mult([Row|Rows], Matrix, N, Rank):-
8    length(Row,NumRows),
9    thread_send_message(vid(Rank,0), query(N,Row,Matrix,NumRows)),
10   N1 is N - 1,
11   Rank1 is Rank - 1,
12   mult(Rows, Matrix, N1, Rank1).
```

5.3.2 Parallel N-Queens Problem

The N queens puzzle is a well studied toy program in computer science used mainly to study algorithms and perform benchmarks. It consists in finding all the solutions for the placement of N queens on an NxN chess board. The only condition to abide is that no two queens attack themselves. What this means is that no queen can share a row, a column, or a diagonal with any other queen.

The most straightforward way of solving this problem in Prolog is by using constraint logic programming or by using a backtracking algorithm.

Though simple, this is a computationally intensive process. What we present here is a parallel version of N-Queens using the task-farm paradigm. Task-

farming is the most simple and commonly used way of parallelizing applications. A master is set up, which takes care of creating tasks and distributing them among workers. The workers perform the tasks and send the results back to the master, which reassembles them. For simplifying the task, in this case only the solutions are counted for each problem size. This is largely based on [Marques, 2003].

Parallelizing the algorithm is straightforward. In this case, a job consists in a valid placement of queens up until a certain column. A result consists in the number of solutions found for that particular job. A worker must find all the solutions for that board prefix, then send the number back to the master.

Program 18 Worker code example for finding all the solutions for the placement of N queens on a NxN chess board.

```
1 worker_work:-
2   thread_get_message(Termo),
3   calc_solutions(Termo, X),
4   thread_send_message(vid(0,0), X),
5   worker_work.
6
7 calc_solutions(q(N, List), NumSolutions):-
8   findall(L, solution_queens(List, N, L), G),
9   length(G, NumSolutions).
```

5.3.3 Parallel Number of Occurrences

Consider the program presented in section 3.1 but now for running on a parallel computer. We've seen that such program, to count the number of occurrences of a number in a list of numbers, is rather simple in Prolog.

In the parallel version, the program will now find all occurrences of a certain integer by dividing the list for processing by the available workers and doing a local search. Then, on the master, the number of occurrences found is assembled.

The conducted tests refer to a parallel search on a list of 10000 elements executed one hundred times in each worker:

Program 19 Parallel search executed one hundred times in each worker.

```
1  count(q(X,List), 0):-
2    repeat(100, number_of_occurences(X, List, _)).
3
4  repeat(0):-!,fail.
5  repeat(_).
6  repeat(N):- N1 is N-1, repeat(N1).
7
8  repeat(N, G):-
9    repeat(N), G, fail.
10 repeat(_, _).
```

This could also be done by sending a hundred messages to the worker but that would not generate the CPU intensive work as this approach.

5.3.4 Case Study: Speedup on a Real-World Application

So far, we have only considered synthetic benchmarks. That means we have only considered artificial programs that try to match the characteristics of large programs. Although they are fine for testing, real benchmarks can only be obtained from testing real-world applications.

This section describes the results of distributing OpenArp [Aires et al., 2004]. OpenArp is a tool for linguists and computer scientists interested in natural language processing (NLP) and related areas that implements the Centering theory for enhanced pronominal anaphora resolution in Portuguese language documents. An anaphor in a text is an entity that refers to another entity (*antecedent*).

The main algorithm in OpenArp relies on searching a space of possible general pronoun candidates for the one that scores best with respect to several

constraints, e.g. proximity of the pronoun to the anaphor and the current subject.

Since the algorithm is idealized to run as a series of sequencial steps, we had to overcome this fact. The approach used consists in doing a paragraph marking on the input texts.

The search-space of the algorithm was then changed to treat each paragraph as the complete text that is being targeted for anaphora resolution, the drawback being an anaphor that occurs at the beginning of a paragraph will never be resolved to a candidate that occurs in the previous paragraph.

Since different paragraphs usually deal with distinct subjects we argue that this kind of separation will allow to distribute the anaphora resolution process without interfering with the foundations of the algorithm.

Indeed we observed that the difference between the original OpenArp implementation and the newly created distributed version is only about less 6% accurate, in a test set consisting of a total of 313 anaphora.

5.4 Benchmark Results

In this section we present the results obtained for each problem. The tasks were carried out using 2, 6 and 12 CPUs. The values are averaged over 6 runs.

5.4.1 Parallel Matrix Multiplication

In the matrix multiply program the system obtained a speedup of 2.92 times with 6 CPUs and of 4.71 times with 12 CPUs, comparing to the same program running in a single processor.

Table 5.4.1: Obtained times for 64x64 matrix multiplication executed fifty times

Workers	CPUs	Elapsed Time	Speedup
1	2	01:09.7s	1.00
3	6	00:23.9s	2.92
6	12	00:14.8s	4.71

5.4.2 Parallel N-Queens

In the N-Queens program a speedup of 2.98 times with 6 CPUs and 4.78 times with 12 CPUs was obtained.

Table 5.4.2: Obtained elapsed time for the parallel nqueens problem

Workers	CPUs	Elapsed Time	Speedup
1	2	03:23.2s	1.00
3	6	01:08.1s	2.98
6	12	00:42.5s	4.78

5.4.3 Parallel Number of Occurrences

In the parallel array search program a speedup of 3 times with 6 CPUs and 5.20 with 12 CPUs was obtained.

Table 5.4.3: Obtained elapsed time for the parallel number of occurrences problem

Workers	CPUs	Elapsed Time	Speedup
1	2	03:23.2s	1.00
3	6	01:08.1s	3.00
6	12	00:42.5s	5.20

5.4.4 Parallel Anaphora Resolution

In the parallel anaphora resolution program a speedup of 2.24 times with 6 CPUs and 2.50 times with 12 CPUs was obtained.

Table 5.4.4: Obtained elapsed time for parallel anaphora resolution

Workers	CPUs	Elapsed Time	Speedup
1	2	00:05.0s	1.00
3	6	00:02.2s	2.24
6	12	00:02.0s	2.50

5.5 Summary

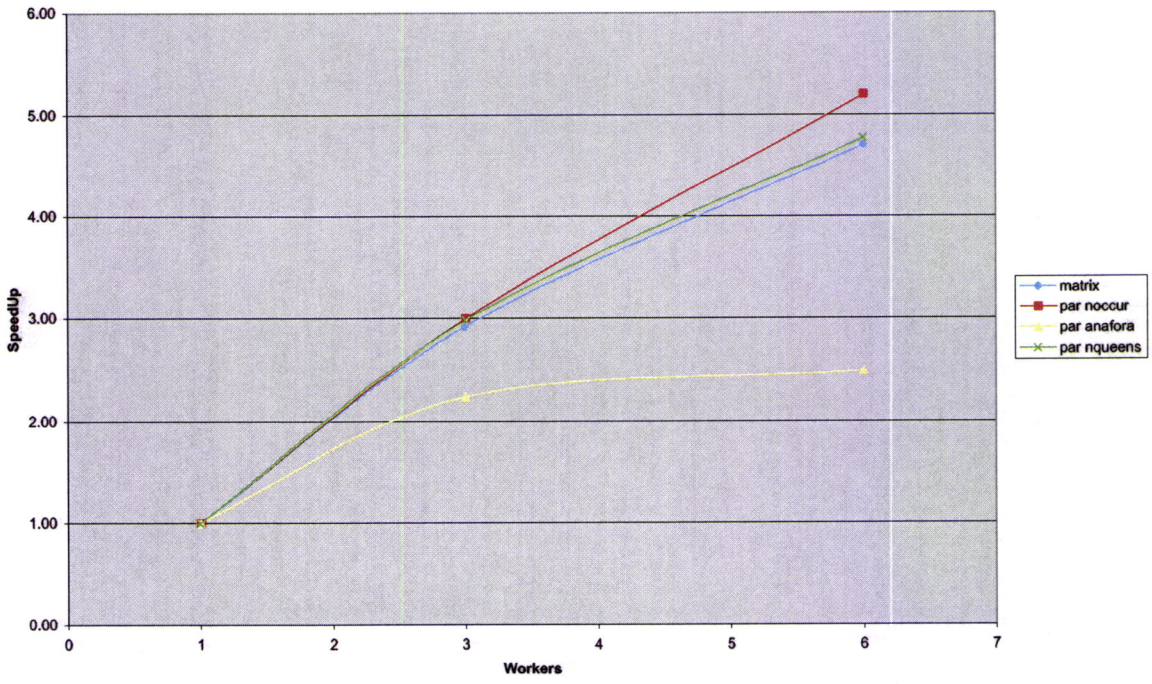

Figure 5.5.1: Speedup with an increasing number of workers defined as elapsed time using one worker divided by elapsed time using N workers.

Quantifying the obtained speedup is important but very dependent on the

problem we are dealing with and on its degree of parallelization. By measuring the speedup we can verify if the system scales.

Looking at Figure 4.5.1 we observe that the speedup is almost linear for the first three examples. That shows that the system scales without problems, at least until the considered number of workers.

Since the cluster used in our benchmarking exercise is relatively small (7 nodes) we can't always observe the point of speedup convergence for all the tested programs, but in the case of the anaphora resolution system we can observe that the speedup is unlikely to reach more than 3 times, independent of the number of workers used.

If more machines were added to the cluster, it is believed that the speedup would also converge for the other three examples to a value where it no longer compensates to have more CPUs for the problem, because the initialization time surpasses the processing time that the task requires. It depends on the problem when this occurs.

Another issue that should be taken into account is the correctness of the results. What happened while distributing OpenArp, in which we chose to modify the search-space of the algorithm in order to distribute the problem, affecting herewith the correctness of the results, will probably happen with other real-world applications. If this separation affects somehow the algorithm of the application, by e.g. modifying the search-space, the accuracy of the system will also be affected. Being so, the accuracy of the results must be assessed in order to verify if the obtained speedup compensates the loss.

In summary, the results show the model is valid and can obtain good performance gains, even when the number of distributed machines is low.

Chapter 6

Conclusions

A system that allows the development of distributed multi-threaded applications in GNU Prolog is now developed.

The system works by executing in each processor a copy of the same program, which is capable of determining it's identity and run different actions accordingly.

The applications can then be submitted for distributed parallel processing using Prolog predicates, via a developed abstraction of MPI functions. Included in this abstraction is the mechanism for remote Prolog job submission. Work is distributed to processors in form of messages, that are received and processed. The results are then sent back or forwarded to other processors.

The processors run inference engines (provers) on native pre-emptive POSIX threads. Each processor has only one thread, but in a computer N processors (virtual or real) can co-exist.

The abstraction is developed on top of the PM2 programming environment and is primarly composed in two parts: thread management and thread

communication.

The Prolog implementation used is GNU Prolog. We've chosen GNU Prolog due to the relatively small stand-alone binaries it produces. The main drawback related with this choice was that gprolog doesn't support local multi-threading (in a shared memory space). For that reason, in each listener thread it is only possible to create one attached thread. However, since the same machine can hold multiple listener threads, it becomes possible to run N local threads in each processor.

The developed system is at a prototype state. It will need further development and testing. However, we decided to conduct tests to evaluate the preliminary performance of our system. We used three classic literature problems plus a real-world application and measured the obtained speedup in each one using one worker (sequential version), three workers and six workers. The programs used were:

- Parallel matrix (64x64) multiplication;
- Parallel N-Queens problem;
- Parallel array search;
- Parallel Pronominal Anaphora Resolution;

In the matrix multiply program the system obtained a speedup of 2.92 times with 6 CPUs and of 4.71 times with 12 CPUs, comparing to the same program running in a single processor.

In the N-Queens program a speedup of 2.98 times with 6 CPUs and 4.78 times with 12 CPUs was achieved.

In the parallel array search program the speedup reached 3 times with 6 CPUs and 5.20 with 12 CPUs.

In the parallel anaphora resolution program the speedup reached 2.24 times with 6 CPUs and 2.50 times with 12 CPUs.

In summary, the results show the model is valid and can obtain good performance gains, even when the number of distributed machines is low.

The conducted tests obtained an almost linear speedup on the first three problems. On a more real-world application, OpenArp, we obtained speedup but relatively less comparing with the other tested programs. Our results also showed how the accuracy of an application might be affected by distributing it's algorithm. In OpenArp this happened because we modified the search-space of the algorithm instead of using a parallel algorithm for anaphora resolution, if such algorithm is even possible. The case has been made to show that if a parallel algorithm isn't possible, then accuracy might be sacrified in favor of a speedup in execution time.

We noticed that concurrent Prolog programs perform very good and we feel encouraged to test bigger configurations. Performance, however, degrades quickly when using predicates that require synchronization or that make intensive use of the network. A solution for this issue might reside in the duplication of what is going to be passed over the network and send only a reference over the network. This might not be possible to execute in several scenarios, such as problems where the messages are created at runtime.

Other issues that are associated with the current implementation include:

- Many situations, if not handled carefully, can lead to deadlock, e.g. a thread not receiving the terminate message, due to an error, will cause the main thread to deadlock.

- Theads have to be terminated explicitly. It would increase performance if the threads terminated as soon as no more jobs are available. This would release CPU for other threads.

While working towards improving our proposal, solving these issues, we also want to pursuit several traits. These are:

- Extend the API with introspection and monitoring predicates. That will permit programmers to control better the running distributed program.
- Test the system with bigger configurations, namely with GRID, and more powerful applications.
- Use distributed multi-threading to build and control intelligent agents capable of taking specific actions.
- Test the combination of GNU Prolog with OpenMPI.
- Work closely with Prolog developers extending the Prolog multi-threading support ISO standard to account for distributed multi-threading.
- Implement an abstraction for distributed shared-memory system in Prolog using MPI.

In a non-distributed multi-threaded environment, powerful applications are limited by the number of threads that can effectively run concurrently.

In a distributed multi-threaded environment resources are pooled and a scheduler sets the rules for routing the jobs to help optimize resources automatically, for accelerated results and help reducing processing time.

Prolog can play a fundamental part in the next generation of applications that will exploit multi-core architectures and bring concurrency to the masses. It

will permit on many cases programs to have a declarative, logic interpretation and will allow for the programmer to omit most control, helping the expression of complex applications and algorithms. The developed system is a tool to help in this process.

Bibliography

Ana Aires, Jorge Coelho, Sandra Collovini, Paulo Quaresma, and Renata Vieira. *Avaliação de Centering em Resolução Pronominal da Língua Portuguesa.* 2004.

Khayri A. M. Ali and Roland Karlsson. *The Muse Or-parallel Prolog model and its performance.* MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-54058-4.

Thara Angskun, Graham E. Fagg, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra. *Self-Healing Network for Scalable Fault Tolerant Runtime Environments.* Springer-Verlag, Innsbruck, Austria, September 2006a.

Thara Angskun, Graham E. Fagg, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra. *Scalable Fault Tolerant Protocol for Parallel Runtime Environments.* Lecture Notes in Computer Science. Springer-Verlag, Bonn, Germany, September 2006b.

Olivier Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes.* Thèse de doctorat, spécialité informatique, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, September 2002. 154 pages.

David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-63392-2.

M. Calejo. *InterProlog, a declarative Java-Prolog interface.* in Procs. Logic Program-

ming for Artificial Intelligence and Information Systems (thematic workshop of the 10th Portuguese Conference on Artificial Intelligence), Porto, 2001.

D. Chu. *I.C. Prolog II: A Multi-Threaded Prolog System.* Kluwer, Dordrecht, 1994.

Keith Clark and Steve Gregory. *PARLOG: parallel programming in logic*, volume 8. ACM Press, New York, NY, USA, 1986.

Keith L. Clark, Peter J. Robinson, and Richard Hagen. *Multi-threading and Message Communication in Qu-Prolog*, volume 1. 2001.

Alain Colmerauer and Philippe Roussel. *The birth of Prolog.* ACM Press, New York, NY, USA, 1996. ISBN 0-201-89502-1.

Jon Cook. *P#: Using Prolog within the .NET Framework.* 2001.

Vítor Santos Costa, David H. D. Warren, and Rong Yang. *The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model.* 1991a.

Vítor Santos Costa, David H. D. Warren, and Rong Yang. *The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model.* 1991b.

Daniel Diaz and Philippe Codognet. *The GNU Prolog systems and its implementation.* In ACM Symposium on Applied Computing, Como, Italy, 2000.

Kevin Dowd. *High performance computing.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1993. ISBN 1-56592-032-5.

Jesper Eskilson and Mats Carlsson. *SICStus MT — A Multithreaded Execution Environment for SICStus Prolog*, volume 1490. 1998.

Graham E. Fagg, Edgar Gabriel, Zizhon Chen, Thara Angskun, George Bosilca, Antonin Bukovsky, and Jack J. Dongarra. *Fault Tolerant Communication Library and Applications for High Performance Computing.* 2003.

Ivan Futo. *Prolog with communicating processes: from T-Prolog to CSR-Prolog*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-73105-3.

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. Budapest, Hungary, September 2004.

Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. *A network-failure-tolerant message-passing system for terascale clusters*, volume 31. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. *Open MPI: A High-Performance, Heterogeneous MPI*. Barcelona, Spain, September 2006.

Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. *Parallel Execution of Prolog Programs: a Survey*, volume 23. 2001.

Manuel V. Hermenegildo and K. J. Greene. *The &-Prolog System: Exploiting Independent And-Parallelism.*, volume 9. 1991.

T. Hoefler, P. Gottschling, W. Rehm, and A. Lumsdaine. *Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations*. Lecture Notes in Computer Science. Springer-Verlag, Bonn, Germany, September 2006.

P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*. 1994.

Rainer Keller, George Bosilca, Graham Fagg, Michael Resch, and Jack J. Dongarra. *Implementation and Usage of the PERUSE-Interface in Open MPI*. Lecture Notes in Computer Science. Springer-Verlag, Bonn, Germany, September 2006.

E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. *The Aurora or-parallel Prolog system*, volume 7. Ohmsha, Tokyo, Japan, Japan, 1988.

Paulo Marques. *Task Farming & The Message Passing Interface*, volume 28. Dr. Dobbs Journal, September 2003.

R. Marques and J. Cunha. *PVM-Prolog: A Prolog interface to PVM*. 1996.

John McCarthy. *Programs with Common Sense*. Her Majesty's Stationary Office, London, 1959.

Paulo Moura. *ISO/IEC DTR 13211-5:2007 Prolog Multi-Threading Support*. 2007.

Paulo Moura. *Logtalk development: Porting Prolog programs to Logtalk*. 2006. URL `http://logtalking.blogspot.com`.

Raymond Namyst and Jean-François Méhaut. *Parallel Computing: State-of-the-Art and Perspectives. Proceedings of the Intl. Conference ParCo '95, Ghent, Belgium, 19–22 September 1995*, volume 11 of *Advances in Parallel Computing*, chapter PM^2: Parallel Multithreaded Machine. A Computing Environment for Distributed Architectures, pages 279–285. Elsevier, February 1996.

Raymond Namyst and Jean-François Méhaut. *Marcel : Une bibliothèque de processus légers*. LIFL, Univ. Sciences et Techn. Lille, 1995.

Luis Moniz Pereira, Luis Monteiro, Jose Cunha, and Joaquim N Aparicio. *Delta Prolog: a distributed backtracking extension with events*. Springer-Verlag New York, Inc., New York, NY, USA, 1986. ISBN 0-387-16492-8.

Jim Plank and Rich Wolski. *C560 Lecture notes - Dining Philosophers*. URL `http://www.cs.utk.edu/~plank/plank/classes/cs560/560/notes/Dphil/lecture.html`.

Ricardo Rocha, Fernando M. A. Silva, and Vitor Santos Costa. *YapOr: an Or-Parallel Prolog System Based on Environment Copying*. 1999.

Kish Shen. *Exploiting Dependent And-Parallelism in Prolog: The Dynamic Dependent And-Parallel Scheme (DDAS)*. 1992.

Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts – 1st ed.* John Wiley & Sons, 2000.

Jeffrey M. Squyres and Andrew Lumsdaine. *A Component Architecture for LAM/MPI*. Number 2840 in Lecture Notes in Computer Science. Springer-Verlag, Venice, Italy, September / October 2003.

V. S. Sunderam. *PVM: a framework for parallel distributed computing*, volume 2. John Wiley & Sons, Chichester, West Sussix, 1990.

LaBRI Team RUNTIME. *Getting Started with PM2*, 2001.

Kazunori Ueda. *Guarded Horn Clauses*. 1985.

Kazunori Ueda and Takashi Chikayama. *Design of the Kernel Language for the Parallel Inference Machine.*, volume 33. 1990.

Jan Wielemaker. *Native Preemptive Threads in SWI-Prolog*. Springer Verlag, Berlin, Germany, december 2003. LNCS 2916.

Jan Wielemaker. Swi-prolog reference manual, 1997.

Jan Wielmaker. *A C++ Interface to SWI-Prolog*. 2000.

Michael J. Wise. *Experience with PMS-Prolog: a distributed, coarse-grain-parallel Prolog with processes, modules and streams*, volume 23. John Wiley & Sons, Inc., New York, NY, USA, 1993.

Appendices

Appendix A

PM2-Prolog Example Makefile

```
1  OBJECT_PL=intarray.o
2
3  CFLAGS= -C -static $(shell pm2-config --cflags | tr ' ' '\n' | awk '{printf " -C %s", $$1}')
4  LIBS= $(shell pm2-config --libs | tr ' ' '\n' | awk '{printf " -L %s", $$1}')
5  LIBS= $(shell pm2-config --libs)
6  CCFLAGS= $(shell pm2-config --cflags)
7  LDFLAGS= -L -static $(shell pm2-config --libs)
8  # prolog libs
9  PLL=/home/nm/extended_stack_gprolog/gprolog-1.2.16/lib
10 # gcc-2.95
11 L= -C -I/usr/lib/gcc-lib/i486-linux-gnu/2.95.4/include
12
13 all: tabard
14
15 gprolog-pm2.o: gprolog-pm2.c
16         gplc -c $(L) $(CFLAGS) gprolog-pm2.c
17
18 tabard.o: tabard.c
19         gplc -c $(L) $(CFLAGS) tabard.c
20
21 %.o: %.pl
```

```
22          gplc -c $+
23
24  tabard: gprolog-pm2.o tabard.o $(OBJECT_PL)
25          gcc -static -o tabard                       \
26                  $(PLL)/obj_begin.o                  \
27                  $+                                  \
28                  -L$(PLL)                            \
29                  -lbips_fd -lengine_fd -lbips_pl     \
30                  $(PLL)/obj_end.o                    \
31                  $(LIBS)                             \
32                  -lengine_pl -llinedit -lm
33          mv tabard /home/nm/build/$(PM2_FLAVOR)/examples/bin/
34
35  clean:
36          rm -f *.o *~ tabard
37
38  run:
39          @pm2load tabard
```

Appendix B

SWI-Prolog Multi-thread Example: Dining Philosophers

```
1  %
2  % Dining Philosophers in Prolog
3  % Based on:
4  % http://www.cs.utk.edu/~plank/plank/classes/cs560/560/notes/Dphil/lecture.html
5  %
6
7  %
8  % chopstick<->atom correspondece
9  % (so that a named mutex is created later)
10 %
11 chopstick_id(1, chopstick1).
12 chopstick_id(2, chopstick2).
13 chopstick_id(3, chopstick3).
14 chopstick_id(4, chopstick4).
15 chopstick_id(5, chopstick5).
16
17 %
18 % chopstick(+Philosopher_Id, +Chopstick_Left_Id, +Chopstick_Right_Id)
```

```
19 % Positions of philosophers and chopsticks
20 %
21 chopstick(1, 5, 1).
22 chopstick(2, 1, 2).
23 chopstick(3, 3, 2).
24 chopstick(4, 4, 3).
25 chopstick(5, 5, 4).
26
27 %
28 % init(+Number)
29 % Number - run for Number times
30 %
31 init(Number):-
32   mutex_create(chopstick1),
33   mutex_create(chopstick2),
34   mutex_create(chopstick3),
35   mutex_create(chopstick4),
36   mutex_create(chopstick5),
37   thread_create(run(1, 5, 0, Number), A, []),
38   thread_create(run(2, 5, 0, Number), B, []),
39   thread_create(run(3, 5, 0, Number), C, []),
40   thread_create(run(4, 5, 0, Number), D, []),
41   thread_create(run(5, 5, 0, Number), E, []),
42   thread_join(A, _),
43   thread_join(B, _),
44   thread_join(C, _),
45   thread_join(D, _),
46   thread_join(E, _).
47
48
49 %
50 % pickup(+Philosopher, -Secs)
51 % Philosopher = philosopher id
52 % Secs = number of seconds blocked
53 %
```

```
54 pickup(Philosopher, Secs):-
55   % T0 = time just before the block started
56   my_get_time(M0,S0),
57
58   % get the chopstick at its left and right first
59   chopstick(Philosopher, LeftStickId, RightStickId),
60   chopstick_id(LeftStickId, LeftStick),
61   chopstick_id(RightStickId, RightStick),
62
63   % pickup left and then right chopstick
64   mutex_lock(LeftStick),
65   mutex_lock(RightStick),
66
67  % sleep(2), % test with fixed blocktime
68
69   % T = time just after the block ended
70   my_get_time(M,S),
71   Secs is ((M*60+S) - (M0*60+S0)).
72
73 %
74 % putdown(+Philosopher)
75 %
76 putdown(Philosopher):-
77   chopstick(Philosopher, LeftStickId, RightStickId),
78   chopstick_id(LeftStickId, LeftStick),
79   chopstick_id(RightStickId, RightStick),
80
81   % putdown right and then left chopstick
82   mutex_unlock(RightStick),
83   mutex_unlock(LeftStick).
84
85 %
86 % my_get_time(-Minutes, -Seconds)
87 %
88 my_get_time(Minutes,Seconds):-
```

```
89    get_time(Time),
90    convert_time(Time,_,_,_,_,Minutes,Seconds,_).
91
92                                  %
93                                  % MAIN
94                                  %
95
96  % The philosophers basically go through the following steps.
97  %
98  % while(1) {
99  % think for a random number of seconds
100 % pickup(p);
101 % eat for a random number of seconds
102 % putdown(p);
103 % }
104 %
105
106 %
107 % run(+Philosopher, +MaxSleepTime, +Blocktime, +Counter)
108 %
109 % List - a list containing the philosophers id
110 % Max_Sec_Time - max sleep time
111 % Blocktime - accumulator for block time
112 % Counter - number of times to run this goal
113 %
114
115 run(Philosopher, _, Acc_Blocktime, 0):- !,
116   write(Philosopher),write('\t'),
117   write(Acc_Blocktime),nl,
118   flush_output.
119
120 run(Philosopher, Max_Sec_Time, Acc_Blocktime, Count):- !,
121   /* First the philosopher thinks for a random number of seconds */
122   think(Philosopher, Max_Sec_Time),
123
```

```
124   /* Now, the philosopher wakes up and wants to eat. He calls pickup
125         to pick up the chopsticks */
126   pickup(Philosopher, Secs),
127
128   % Accumulate blocktime
129  % write('Debug: '),write(Philosopher),write(' Acc = '),
130  % write(Acc_Blocktime),write(', Secs = '),write(Secs),
131   Acc_Blocktime1 is (Acc_Blocktime + Secs),
132  % write(', Blocktime = '), write(Acc_Blocktime1),nl,
133  % flush_output,
134
135   /* When pickup returns, the philosopher can eat for a random number of
136         seconds */
137   eat(Philosopher, Max_Sec_Time),
138
139   /* Finally, the philosopher is done eating, and calls putdown to
140         put down the chopsticks */
141   putdown(Philosopher),
142
143   Count1 is Count - 1,
144
145   !, run(Philosopher, Max_Sec_Time, Acc_Blocktime1, Count1).
146
147                                %
148                                % THINK & EAT
149                                %
150
151  % think(+Philosopher, +Max_Sec_Time)
152  % eat(+Philosopher, +Max_Sec_Time)
153  %
154  % Calculate a random number between 0 and Max_Sec_Time and then sleep
155  % for that time.
156  %
157
158  think(Philosopher, Max_Sec_Time):-
```

```
159   random(1, Max_Sec_Time, Sleep_Time),
160   write('Philosopher '),
161   write(Philosopher),
162   write(' thinking for '),
163   write(Sleep_Time),
164   write(' seconds.'),nl,
165   flush_output,
166   sleep(Sleep_Time),
167   write('Philosopher '), write(Philosopher),
168   write(' no longer thinking - calling pickup'),nl,
169   flush_output.
170
171 eat(Philosopher, Max_Sec_Time):-
172   random(1, Max_Sec_Time, Sleep_Time),
173   write('Philosopher '),
174   write(Philosopher),
175   write(' eating for '),
176   write(Sleep_Time),write(' seconds.'),nl,
177   flush_output,
178   sleep(Sleep_Time),
179   write('Philosopher '),write(Philosopher),
180   write(' no longer eating -- calling putdown'),nl,
181   flush_output.
182
183
```

Appendix C

Parallel Matrix Multiplication

```
1  :- initialization(init).
2
3  % include the pm2 interface lib
4  :- include('lib').
5
6  rows(64).
7  cols(X):- rows(X).
8
9  determine(Rows, Rows):-
10   argument_counter(2),
11   argument_list(Args),
12   append([RowsString|_], [], Args),
13   write_to_chars(Chars, RowsString),
14   number_chars(Rows, Chars).
15
16 determine(Rows, Cols):-
17   rows(Rows),
18   cols(Cols),
19   write('Going for default row number of '),write(Rows),write('.'),nl.
20
21 % thread 0
```

```
22 init:-
23   pm2_is_master,!,
24   pm2_max_rank(MaxRank),
25   pm2_config_size(ConfigSize),
26
27   % read number of columns from console
28   % or go with the default value
29   determine(Rows, Cols),
30
31   % Start_Prolog(0, 0) on each node
32   start_prolog_workers(MaxRank),
33
34   fill_matrix(Matrix1, Rows, Cols),
35   fill_matrix(Matrix2, Rows, Cols),
36
37   % give workers work
38   mult(Matrix1, Matrix2, Rows, MaxRank),
39   read_results(Rows),
40
41   stop_prolog_workers(MaxRank),
42
43   finish_listeners.
44
45 % thread != 0
46 init:-
47   worker_work.
48
49 init.
50
51 worker_work:-
52         !, thread_get_message_loop,
53         worker_work.
54
55 worker_work:-
56   write('worker_work falhou'),nl,!,fail.
```

```
57
58 % tirar todas as mensagens da queue
59 thread_get_message_loop:-
60   thread_get_message(Termo),
61   !, do_query(Termo, X),
62   thread_send_message(vid(0,0), X),
63   thread_get_message_loop.
64
65 % For NxN matrix will have to read N lines
66 read_results(0):-!.
67 read_results(X):-
68         thread_get_message(R),
69         X1 is X - 1,
70         !, read_results(X1).
71
72 read_results_loop:-
73         thread_get_message(Result),
74         !, read_results_loop.
75
76 do_query(query(NumberOfRow,Row,Matrix,NumRows), (NumberOfRow,NL)):-
77   repeat(50, multiply_row_per_matrix(Row, Matrix, 1, NumRows, _)),
78   multiply_row_per_matrix(Row, Matrix, 1, NumRows, NL).
79
80 repeat(0):-!,fail.
81 repeat(_).
82 repeat(N):- N1 is N-1, repeat(N1).
83
84 repeat(N, G):-
85   repeat(N), G, fail.
86 repeat(_, _).
87
88 %
89 % multiply_matrix(+Matrix1, +Matrix2, +NumRows, -Result)
90 %
91 % for each row in Matrix1 call multiply_row_per_col for all the
```

```
92 % columns in Matrix2
93 %
94
95 mult([], _, 0, _):- !. % N messages were sent for a NxN matrix
96
97 % round-roubin the machines
98 mult(L, M, N, 0):-
99   pm2_max_rank(MaxRank),
100  !, mult(L, M, N, MaxRank).
101
102 mult([Row|Rows], Matrix, N, Rank):-
103   length(Row,NumRows),
104   thread_send_message(vid(Rank,0), query(N,Row,Matrix,NumRows)),
105   N1 is N - 1,
106   Rank1 is Rank - 1,
107   mult(Rows, Matrix, N1, Rank1).
108
109
110 % the maximum number when generating the matrix content.
111 maximum(7).
112
113 %
114 % multiply_row_per_matrix(+Row, +Matrix, +NumCol, +NumRows, -Result)
115 %
116 % multiplies a line of the first matrix per all the columns of the
117 % second and obtains the first line of the solution matrix.
118 %
119 multiply_row_per_matrix(_, _, N, M, []):- N > M, !.
120
121 multiply_row_per_matrix(Row, Matrix, NumCol, NumRows, [RowCol|Res]):-
122   col_items(Matrix, NumCol, Col),
123   multiply_row_per_col(Row, Col, RowCol),
124   NumCol1 is NumCol + 1,
125   multiply_row_per_matrix(Row, Matrix, NumCol1, NumRows, Res).
126
```

```
127 %
128 % multiply_row_per_col(+Row, +Col, -Result)
129 %
130 % multiply a row (list) per a column (list) like this:
131 % Elem*Elem + Elem*Elem + .. + Elem*Elem
132 %
133 multiply_row_per_col(Row, Col, Res):-
134   multiply_row_per_col(Row, Col, 0, Res).
135
136 multiply_row_per_col([], [], V, V):- !.
137 multiply_row_per_col([First|Rest], [Second|SecRest], Acc, Res):-
138   Acc1 is (Acc + (First*Second)),
139   multiply_row_per_col(Rest, SecRest, Acc1, Res).
140
141 %
142 % col_items(+Matrix, +NumCol, -Col)
143 %
144 % given a matrix unify Col with a list of all the elements in that
145 % col on the matrix.
146 %
147 col_items([], _, []):- !.
148 col_items([First|Rest], NumCol, [X|Col]):-
149   nth(NumCol, First, X),
150   col_items(Rest, NumCol, Col).
151
152 %
153 % fill_matrix(-Matrix, +Rows, +Cols)
154 %
155 % given N and M return a list of lists (matrix) filled randomly.
156 %
157 fill_matrix([], 0, _):- !.
158 fill_matrix([First|Rest], Rows, Cols):-
159   fill_row(First, Cols),
160   Rows1 is Rows - 1,
161   fill_matrix(Rest, Rows1, Cols).
```

```
162
163 %
164 % fill_row(-List, +N)
165 %
166 % Given N unifies List with a list of length N filled randomly.
167 %
168 fill_row([], 0):- !.
169 fill_row([First|Rest], Cols):-
170   maximum(Max),
171   random(2, Max, First),
172   Cols1 is Cols - 1,
173   fill_row(Rest, Cols1).
174
```

Appendix D

Parallel N-Queens

```
1  :- initialization(init).
2
3  % include the pm2 interface lib
4  :- include('lib').
5
6  size(8).
7
8  determine(Size):-
9    argument_counter(2),
10   argument_list(Args),
11   append([SizeString|_], [], Args),
12   write_to_chars(Chars, SizeString),
13   number_chars(Size, Chars).
14
15 determine(Size):-
16   size(Size).
17
18 % thread 0
19 init:-
20   pm2_is_master,!,
21   pm2_max_rank(MaxRank),
```

```
22
23    determine(N),
24
25    % Start_Prolog(0, 0) on each node
26    start_prolog_workers(MaxRank),
27
28    column(N, I),
29    findall(ParcialSolution, mkmaxlist(I, N, ParcialSolution), G),
30    length(G, GL),
31
32    send_job_worker(G, GL, N, MaxRank),
33    read_results(GL, NumSolutions),
34    write(NumSolutions),nl,
35
36    stop_prolog_workers(MaxRank),
37
38    finish_listeners.
39
40  % thread != 0
41  init:-
42    worker_work.
43
44  worker_work:-
45          !, thread_get_message_loop,
46          worker_work.
47
48  thread_get_message_loop:-
49    thread_get_message(Termo),
50    !, do_query(Termo, X), !,
51    thread_send_message(vid(0,0), X),
52    thread_get_message_loop.
53
54
55  read_results(X, N):-
56    read_results(X, 0, N).
```

```
57
58 read_results(0, N, N):-!.
59 read_results(X, Acc, N):-
60         thread_get_message(Msg),
61         Msg == 1,
62         Acc1 is Acc + 1,
63         X1 is X - 1,
64         !, read_results(X1, Acc1, N).
65
66 read_results(X, Acc, N):-
67         X1 is X - 1,
68         !, read_results(X1, Acc, N).
69
70 do_query(q(N, List), NumSolutions):-
71   findall(L, solution_queens(List, N, L), G),
72   length(G, NumSolutions).
73
74 send_job_worker([], 0, _, _):- !.
75
76 send_job_worker(L, C, N, 0):-
77   pm2_max_rank(MaxRank),
78   !, send_job_worker(L, C, N, MaxRank).
79
80 send_job_worker([H|Tail], Counter, N, Rank):-
81   thread_send_message(vid(Rank,0), q(N, H)),
82   Counter1 is Counter - 1,
83   NextRank is Rank - 1,
84   send_job_worker(Tail, Counter1, N, NextRank).
85
86
87 column(N, X):-
88   X is N - 4.
89
90
91 go(N):-
```

```
92    findall(Y, teste(N, Y), G),
93    length(G, GL),
94    write(N),write(': '),write(GL),write(' total solutions.'),nl,fail.
95
96
97  % solution_queens(+ParcialSolution, -Solution)
98  solution_queens([], _, _):- fail.
99  solution_queens(L, N, PossibleSolution):-
100   column(N, I),
101   MissingQueens is N - I,
102   mkemptylist(MissingQueens, EL),
103   append(L, EL, NL),
104
105   mkmaxlist(MissingQueens, N, PossibleParcialSolution),
106
107   cross_lists(NL, PossibleParcialSolution, PossibleSolution),
108   queens(N, PossibleSolution).
109
110 cross_lists([], _, []):-!.
111 cross_lists(_, [], []):-!.
112 cross_lists([X|Xs], [Y|Ys], [Y|R]):-
113   var(X),
114   nonvar(Y),
115   cross_lists(Xs, Ys, R).
116 cross_lists([X|Xs], [Y|Ys], [X|R]):-
117   nonvar(X),
118   cross_lists(Xs, [Y|Ys], R).
119
120
121 mkmaxlist(S, N, [X|R]):-
122   S > 0,
123   for(X, 1, N),
124   S1 is S - 1,
125   mkmaxlist(S1, N, R).
126 mkmaxlist(0, _, []).
```

```
127
128 mkemptylist(N, [_|R]):-
129   N > 0,
130   M is N - 1,
131   !, mkemptylist(M, R).
132 mkemptylist(_, []).
133
134
135 % queens(+BoardSize, -ResultBoard)
136 queens(N, Qs):-
137   range(1,N,Ns),
138   permutation(Ns,Qs),
139   safe(Qs).
140
141 safe([Q|Qs]):- safe(Qs), \+ attack(Q,Qs).
142 safe([]).
143
144 attack(X,Xs):- attack(X,1,Xs).
145 attack(X,N,[Y|_]):- X is Y + N.
146 attack(X,N,[Y|_]):- X is Y - N.
147 attack(X,N,[_|Ys]):- N1 is N + 1, attack(X,N1,Ys).
148
149
150 range(N,N,[N]):- !.
151
152 range(M,N,[M|Ns]):-
153         M < N,
154         M1 is M+1,
155         range(M1,N,Ns).
```

Appendix E

Parallel Number of Occurrences

```
1  :- initialization(init).
2
3  % include the pm2 interface lib
4  :- include('lib').
5
6  size(8).
7
8  determine(Size):-
9    argument_counter(2),
10   argument_list(Args),
11   append([SizeString|_], [], Args),
12   write_to_chars(Chars, SizeString),
13   number_chars(Size, Chars).
14
15 determine(Size):-
16   size(Size),
17   write('Searching for '),write(Size),write('.'),nl.
18
19 % thread 0
20 init:-
21   pm2_is_master,!,
```

```
22    pm2_max_rank(MaxRank),
23
24    determine(El),
25
26    % Start_Prolog(0, 0) on each node
27    start_prolog_workers(MaxRank),
28
29    see('/home/nm/devel/tabard-0.1/input.txt'),
30    read_chunk(El, 1000, MaxRank, NumberSentMessages),
31    read_results(NumberSentMessages, Num),
32    write(Num),nl, % number of total occurrences
33
34    stop_prolog_workers(MaxRank),
35
36    finish_listeners.
37
38  % thread != 0
39  init:-
40    worker_work.
41
42  worker_work:-
43    thread_get_message(Termo),
44    !, do_query(Termo, X),
45    thread_send_message(vid(0,0), X),
46    worker_work.
47
48  read_results(X, N):-
49    read_results(X, 0, N).
50
51  read_results(0, N, N):-!.
52  read_results(X, Acc, N):-
53          thread_get_message(Msg),
54          Acc1 is Acc + Msg,
55          X1 is X - 1,
56          !, read_results(X1, Acc1, N).
```

```
57
58 do_query(q(X,List), 0):-
59   repeat(100, number_of_occurences(X, List, _)).
60
61 repeat(0):-!,fail.
62 repeat(_).
63 repeat(N):- N1 is N-1, repeat(N1).
64
65 repeat(N, G):-
66   repeat(N), G, fail.
67 repeat(_, _).
68
69
70
71 number_of_occurences(X, L, N):-
72   number_of_occurences(X, L, 0, N).
73
74 number_of_occurences(_, [], Acc, Acc):-!.
75 number_of_occurences(X, [Y|Ys], Acc, N):-
76   X \= Y,
77   number_of_occurences(X, Ys, Acc, N).
78
79 number_of_occurences(X, [X|Xs], Acc, N):-
80   Acc1 is Acc + 1,
81   number_of_occurences(X, Xs, Acc1, N).
82
83
84 read_chunk(El, ChunkSize, Rank, N):-
85   read_chunk(El, ChunkSize, Rank, 0, N).
86
87
88
89 read_chunk(El, ChunkSize, 0, Acc, N):-
90   pm2_max_rank(MaxRank),
91   !, read_chunk(El, ChunkSize, MaxRank, Acc, N).
```

```
92
93 % read_chunk(+ChunkSize) reads ChunkSize numbers from input to a list
94 read_chunk(El, ChunkSize, Rank, Acc, N):-
95   read_lista(ChunkSize, ChunkList),
96   length(ChunkList, ChunkSize),
97   ChunkList \= [],
98   thread_send_message(vid(Rank, 0), q(El, ChunkList)),
99   NextRank is Rank - 1,
100  Acc1 is Acc + 1,
101  read_chunk(El, ChunkSize, NextRank, Acc1, N).
102
103 read_chunk(_, _, _, N, N).
104
105 read_lista(N, [X|R]):-
106  N > 0,
107  read_token(X),
108  integer(X),
109  read_token(_),
110  N1 is N - 1,
111  read_lista(N1, R).
112 read_lista(_, []).
```