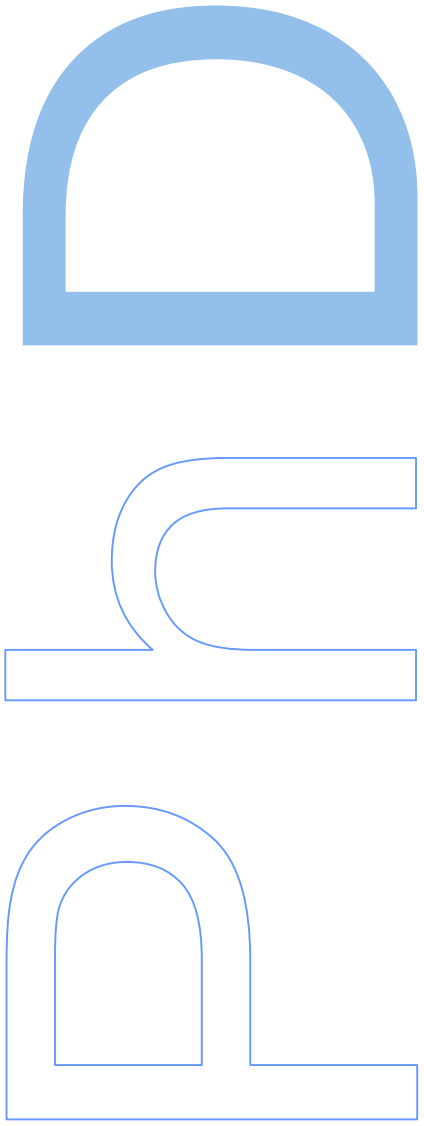
Logic Programming Environments with Advanced Parallelism

João Pedro Barreiros Nunes dos Santos
Programa Doutoral em Ciência de Computadores
Departamento de Ciência de Computadores
2016

Orientador
Ricardo Jorge Gomes Lopes da Rocha, Professor Auxiliar,
Faculdade de Ciências Universidade do Porto



João Pedro Barreiros Nunes dos Santos

Logic Programming Environments with Advanced Parallelism

Thesis submitted to Faculdade de Ciências
of the Universidade do Porto to obtain
the degree of Doctor in Computer Science

Orientador: Prof. Doutor Ricardo Jorge Gomes Lopes da Rocha

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2016

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Ricardo Rocha, for the support and guidance during all this years. He was always there to listen, discuss and help me to carry on this work. I am sure that I am now a better computer scientist because of him.

Besides my advisor, I would like to express my gratitude to Prof. Vítor Santos Costa for his availability and precious help specially during the implementation of the system. I would like to express also a special acknowledgement to Prof. Fernando Silva for his support.

I am thankful to Fundação para a Ciência e Tecnologia (FCT) for funding this work, during 4 years, within the research grant SFRH/BD/76307/2011, and to the Center for Research and Advanced Computing Systems (CRACS) for funding the remaining time.

To my fellow colleagues at DCC-FCUP, specially Miguel Areias, Flávio Cruz and Joana Côrte-Real, I would like to thank for the excellent work environment and to my longtime friends, João Bento and José Vieira, for their unconditional friendship. I wish you all the best for your future.

Finally, I would like to thank my parents, Isolina and Mário, and my son, Pedro Luís, for their support and for always being by my side.

Resumo

Hoje em dia, os *clusters de multicores* estão a tornar-se cada vez mais acessíveis e, embora muitos sistemas paralelos de Prolog tenham sido desenvolvidos no passado, não é do nosso conhecimento, que algum deles tenha sido especialmente concebido para explorar a combinação de arquiteturas de memória partilhada com memória distribuída. Nesta tese, propomos um novo modelo computacional especialmente concebido para tirar partido dessa combinação que introduz um *modelo em camadas* com dois níveis de escalonamento, um para os agentes em memória partilhada, que designamos por *equipa de agentes (team of workers)*, e outro para as *equipas de agentes* (que não partilham memória entre si). No seguimento desta proposta, apresentamos uma primeira implementação do novo modelo que estende o sistema YapOr de forma a explorar parallelismo-ou entre *equipas de agentes*. De modo a ser possível tirar o melhor partido do nosso sistema, propomos ainda um conjunto de predicados *built-in* que constituem a sintaxe para interagir com o sistema. Os resultados experimentais demonstram que o nosso sistema, quando comparado com o YapOr, alcança *speedups* idênticos em memória partilhada e, quando executado em *clusters de multicores*, é capaz de aumentar o *speedup* à medida que aumentamos o número de agentes por equipa, aproveitando assim ao máximo o número de cores em cada máquina, e é capaz de aumentar o *speedup* quando aumentamos o número de equipas, o que permite de tirar partido da junção de mais máquinas ao *cluster* inicialmente disponível. Em suma e nossa convicção que o sistema desenvolvido no âmbito desta tese se apresenta como uma alternativa viável e eficiente para a exploração do parallelismo-ou implícito nos *clusters* de baixo custo que existentes atualmente.

Abstract

Nowadays, clusters of multicores are becoming the norm and, although, many or-parallel Prolog systems have been developed in the past, to the best of our knowledge, none of them was specially designed to explore the combination of shared and distributed memory architectures. In this thesis, we propose a novel computational model specially designed for such combination which introduces a *layered model* with two scheduling levels, one for workers sharing memory resources, which we named a *team of workers*, and another for teams of workers (not sharing memory resources). Starting from this proposal, we then present a first implementation of such model and for that we revive and extend the YapOr system to efficiently exploit or-parallelism between teams of workers. In order to take full advantage of our system, we also propose a new set of built-in predicates that constitute the syntax to interact with an or-parallel engine in our system. Experimental results show that our system, when compared against YapOr, achieves identical speedups for shared memory and, when running on clusters of multicores, is able to increase speedups as we increase the number of workers per team, thus taking advantage of the maximum number of cores in a machine, and to increase speedups as we increase the number of teams, thus taking advantage of adding more computer nodes to a cluster. We thus argue that our system is an efficient and viable alternative for exploiting implicit or-parallelism in the currently available clusters of low cost multicore architectures.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The inherent non-determinism in the way logic programs are structured as simple collections of alternative clauses makes Prolog very attractive for the exploitation of *implicit parallelism*. Prolog offers two major forms of implicit parallelism: *and-parallelism* and *or-parallelism* [16]. And-Parallelism stems from the parallel evaluation of subgoals in a clause, while or-parallelism results from the parallel evaluation of a subgoal call against the clauses that match that call. Arguably, or-parallel systems, such as Aurora [28, 10, 45] and MUSE [4, 3, 5], have been the most successful parallel Prolog systems so far. Intuitively, the least complexity of or-parallelism makes it more attractive and productive to exploit than and-parallelism, as a first step. However, practice has shown that a main difficulty is how to efficiently represent the *multiple bindings* for the same variable produced by the or-parallel execution of alternative matching clauses. One of the most successful or-parallel models that solves the multiple bindings problem is *environment copying*, which has been efficiently used in the implementation of or-parallel Prolog systems both on shared memory [4, 35] and distributed memory [49] architectures.

Another key problem in the implementation of a parallel system is the design of *scheduling strategies* to efficiently assign tasks to workers. In particular, with implicit parallelism, it is expected that the parallel system automatically identifies opportunities for transforming parts of the computation into concurrent tasks of parallel work, guaranteeing the necessary synchronization when accessing shared data. For environment copying, scheduling strategies based on *dynamic scheduling of work* using *or-frame data structures* to implement such synchronization have proved to be very efficient for shared memory architectures [4]. *Stack splitting* [19, 32, 48] is an alternative scheduling strategy for environment copying that provides a simple and

clean method to accomplish work splitting among workers in which the available work is *statically divided beforehand* in complementary sets between the sharing workers. Due to its static nature, stack splitting was first introduced aiming at distributed memory architectures [49] but, recent work, also showed good results for shared memory architectures [48, 47].

Nowadays, the increasing availability and popularity of multicores and clusters of multicores provides an excellent opportunity to turn Prolog an important member of the general ecosystem of parallel computing environments. However, although many parallel Prolog systems have been developed in the past [20], most of them are no longer available, maintained or supported. Moreover, to the best of our knowledge, none of those systems was specially designed to explore the combination of shared and distributed memory architectures.

1.1 Thesis Purpose

This thesis presents the design, implementation and evaluation of a new model conceived for exploring implicit or-parallelism in cluster of multicores. For that, we introduce a *layered model* approach with two scheduling levels, one for workers sharing memory resources, which we named a *team of workers*, and on top of that a scheduler for teams of workers (not sharing memory resources). This approach somehow resembles the concept of teams used by some models combining and-parallelism with or-parallelism, like the Andorra-I [40] or ACE [21] systems, where a layered approach also implements different schedulers to deal with each level of parallelism.

Based on such layered model approach, we then present an implementation that revives and extends the YapOr system [35] to efficiently exploit parallelism between teams of workers running on top of clusters of multicores. YapOr is an or-parallel engine based on the environment copying model that extends the Yap Prolog system [38] to exploit implicit or-parallelism in shared memory architectures. Our implementation takes full advantage of Yap's state-of-the-art fast and optimized engine and reuses the underlying execution environment, scheduler and part of the data structures used to support parallelism in YapOr. On top of that, we have developed a new scheduler based on techniques proposed for distributed memory. In order to take advantage of our implementation, we also propose a new set of built-in predicates that constitute the syntax to interact with an or-parallel engine in our system.

To validate our design and implementation, we set up an experimental environment using a set of 10 well known benchmark programs with several different numbers of workers and different configurations of teams. Our experimental results show that our implementation adds just a small overhead to YapOr when running in shared memory. Furthermore, the experiments also show that our implementation is able to increase the speedups as we increase the number of workers per team, thus taking advantage of the maximum number of cores in a machine, and to increase speedups as we increase the number of teams, thus taking advantage of adding more computer nodes to a cluster.

1.2 Thesis Outline

This thesis is divided in seven major chapters that we briefly describe next:

Chapter 1: Introduction. The present chapter.

Chapter 2: Logic Programming and Parallelism. Presents the basic concepts behind Logic Programming and parallelism with particular emphasis in the Prolog language. It also introduces the key concepts of implicit parallelism in Prolog focusing mainly in or-parallelism, which is the core topic of this thesis.

Chapter 3: Layered Model. Describes the high-level details and characteristics of our parallel layered model aiming to run Prolog code in cluster of multicores and presents the new syntax to interact with an or-parallel engine in our system.

Chapter 4: YapOr System. Presents the key aspects of the execution model of the YapOr system, an or-parallel engine targeting shared memory architectures built on top of Yap, wich is the base for the implementation of our layered model.

Chapter 5: Teams of Workers. Describes in detail the concepts, algorithms and protocols behind our layered model proposal and how we have extended YapOr in order to implement it.

Chapter 6: Performance Analysis. Presents a detailed analysis of the performance of our implementation. Using a set of 10 well known benchmark programs with a different number of workers and several distinct configurations of teams. We have also used a simulator to assess the impact of the network latency in the performance of our system.

Chapter 7: Conclusions. Summarizes the main contributions of the thesis and highlights possible directions for further research.

Chapter 2

Logic Programming and Parallelism

This chapter gives a brief overview of the two main areas of research embraced by this thesis. We begin by discussing Logic Programming with particular emphasis in the Prolog language. Then we introduce the concept of implicit parallelism in Prolog by focusing on the challenges that arise when implementing such systems and by overviewing the most successful models proposed to exploit implicit parallelism in Prolog.

2.1 Logic Programming

Logic Programming allows us to have a high level approach to programming. Such characteristic can be seen in the way on how logic programs are written. Instead of worrying about the details on how to solve the problem, programmers can focus in what to solve. For that reason, Logic Programming is said to adhere to the declarative paradigm.

Logic Programming can be seen as a simple theorem prover where programs are statements defining a certain problem and questions may be asked to them. Then questions are resolved against the program statements, in order to find the set of answers that satisfies them.

Ideally, a logic program should be written as logic statements with the control of execution being tackled as an independent issue by the resolution mechanism. This idea was summarized by Kowalski [26] in:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

However, in practice, it might be a good idea to keep in mind the underlying resolution mechanism if we want to write efficient logic programs.

2.1.1 Logic Programs

A logic program consists in a set of Horn clauses which have the following logic representation:

$$\forall\ v_i\ (B_1 \wedge B_2 \wedge \ldots \wedge B_n) \implies A$$

Or considering Prolog's notation clauses are defined by:

$$A\text{:-}\ B_1,\ B_2,\ \ldots,\ B_n.$$

This can be read as *"if* B_1 *,* B_2*, ...,* B_n *are all true then* A *is true"*. This type of clause is called a *rule* where the literal A is the *head* of the rule and the literals B_i are the *body subgoals*. A clause without a body is called a *fact*, meaning that A is true, and is represented by:

$$A.$$

In order to retrieve information from the program, a clause without head – called *query* – is used:

$$\text{:-}\ B_1,\ B_2,\ \ldots,\ B_n.$$

Each literal in a clause is denoted as:

$$p(t_1,\ t_2,\ \ldots,\ t_n)$$

Where p is the predicate name and the t_i are terms. Each term can either be a *constant* (represented by a word beginning with a lowercase letter), a *variable* (represented by a word beginning with an uppercase letter) or a *compound term*. A compound term

is of the form *f(*s_1*,* s_2*, ...,* s_n*)* where f is the functor name and s_1, s_2, ..., s_n are also terms)

Horn clauses in logic programs are interpreted using the *Selective Linear Definite resolution* (SLD resolution) [27] which was proposed by Kowalski [25] based on the previous work of Robinson [33]. Consider, for example, the query:

$$:\text{-}\ Q_1,\ldots,Q_n.$$

SLD resolution would work on the following way:

- First, an operation called $select_{literal}$, will select a body subgoal Q_i.
- Then, an operation called $select_{clause}$, will select from the program a clause whose head matches with Q_i, if there is any. Assume that the selected clause that matches with Q_i is *"Q:-* $B_1,\ldots,B_n$*."*. The unification process then determines a substitution θ for the variables in Q_i and the head Q such that $\theta Q_i = \theta$Q. Next, Q_i is replaced by the body of the selected clause, resulting in the following new conjunction:

$$\theta\ (Q_1,\ldots,Q_{i-1},B_1,\ldots,B_n,Q_{i+1},\ldots,Q_n)$$

- The process is then repeated to the subgoals in the new conjunction. If, during such process, the conjunction at hand is reduced to *true* the resulting substitution θ is given as a solution. On the other hand, when there are no matching clauses, *backtracking* occurs. Backtracking forces the computation to be restored to the previous selection point in order to try another matching clause. The program ends when there are no more clauses left to try, meaning that all possibilities have been explored.

2.1.2 Prolog

Arguably, the most popular Logic Programming language is Prolog. The name Prolog derives from the abbreviation of *PROgramation en LOGic* and was developed in the first half of the 70's by Colmerauer et al. [11] based on the theoretical work developed by Kowalski [25] and Robinson [33]. In his work, Robinson described two key concepts in the Prolog programming language: the unification and the resolution

process. Kowalski then showed that the Horn clauses together with unification and resolution could have a procedimental meaning.

Till the late 70's, Prolog was restricted to a group of people. But then everything changed when, in 1977, David H. D. Warren presented the first Prolog compiler [51]. Later, in 1983, Warren proposed the *Warren's Abstract Machine* (WAM) [52] capable of running Prolog code even more efficiently.

The operational semantics of Prolog is given by SLD resolution with the operation $select_{literal}$ selecting the subgoals from left to right and the operation $select_{clause}$ selecting the clauses by the order they are written in the program. The exploration of the clauses of a program, done by SLD resolution, can be seen as a tree, where the inner nodes represent choice points, the branches represent the different alternatives (matching clauses) and the leaf nodes represent solutions or failures for the program. In the specific case of Prolog, this tree is explored in a depth-first left to right form with backtracking being used to move back in the tree. Figure 2.1 shows a schematic representation of this process.

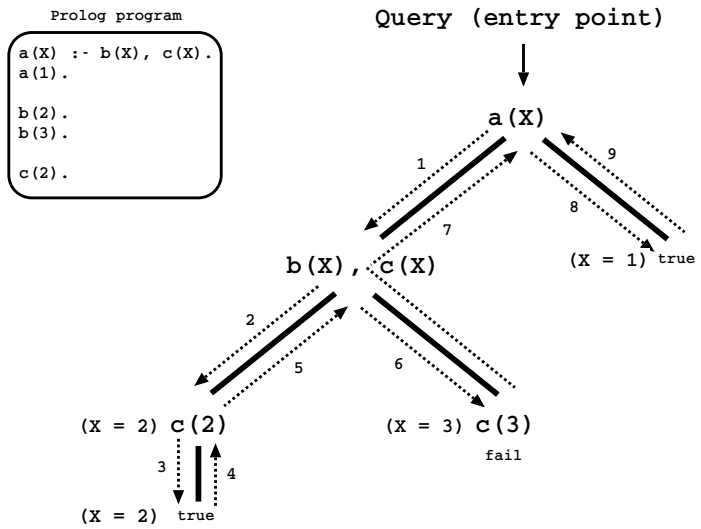


Figure 2.1: Schematic representation of a search tree proof in Prolog

In order to make Prolog more suitable to the everyday use, many extra logical predicates were added to the language. Some of the more important are:

- input/output predicates that allow, for example, to read from and write to files;
- the cut predicate (!) used to control the backtracking mechanism and reduce the search space;

- assert and retract predicates which allow to modify the clauses of the program during execution time;
- meta-logical predicates which allow the programmer to get information about the execution of the current program.

2.1.3 Warren's Abstract Machine

Due to its efficiency, nowadays the WAM is still the standard for most Prolog interpreters. It was designed to efficiently support the two main features of Prolog – unification and backtracking. This abstract machine is composed by two fundamental specifications: the memory layout and the set of instructions.

The WAM memory layout is composed by 5 stacks and by a set of registers as depicted in Fig. 2.2. Next, we briefly describe the functionality of each stack.

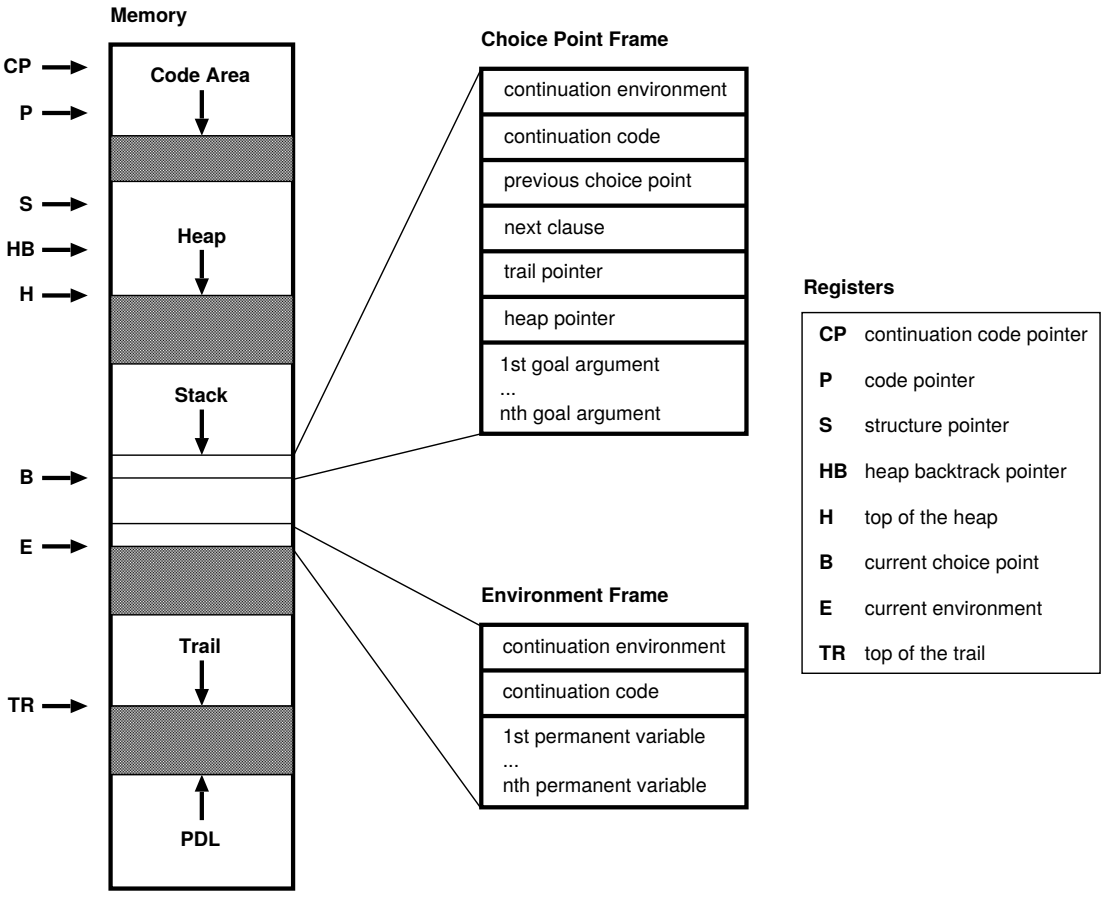


Figure 2.2: WAM memory layout

Code Area: this area stores the WAM instructions of the loaded programs.

Heap: area where the Prolog variables and terms are represented. The register H points to the top of this area.

Stack: used to keep track of the *choice points* and *environment frames* data structures:

- choice points store the state of the computation so that it can be restored later by the backtracking process. They are created whenever a goal has more than one matching alternative. When backtracking occurs they are used to restore the computation state in order to allow other alternatives that are still open to be exploited. The register B points to the current choice point. Each tree node in Fig. 2.1 corresponds, at the engine level, to a choice point stack [52, 2].
- environment frames are created whenever a clause with more than one body subgoal is executed. They are used to store information about the execution of the subgoals and about the *permanent variables*, i.e., variables that appear in more than one subgoal. The register E points to the current environment.

Trail: during execution, variables can be instantiated, but whenever backtracking occurs, their previous state must be restored. Because of that, every binding made to variables is registered in this memory area so that it can be restored when backtracking occurs. Register TR points to the top of this stack.

PDL (Push Down List): is an auxiliary stack used by the unification process.

The other registers in Fig. 2.2 are: the register S, used during the unification of compound terms; the register HB, used to determine the bindings that should be stored in the trail; the register P, that points to the WAM instruction being executed; and the register CP, that points to where to return after a successful execution of the current clause.

Regarding the WAM instructions set it was specially designed to: allow an easy mapping between Prolog instructions and WAM instructions; and to allow an efficient translation to native code. The instructions can be divided in four major groups that are the following:

Choice point instructions: these instructions are responsible for the allocation/deallocation of choice points and recovery of the computation state when applying the Prolog backtracking mechanism.

Control instructions: these instructions are responsible for the allocation/deallocation of environments and for the management of the call/return of subgoals.

Unification instructions: as the name suggests, these instructions are responsible for implementing the Prolog unification mechanism.

Indexing instructions: instead of trying all the clauses of a predicate, these type of instructions allow to efficiently determine which clauses match with a given subgoal, therefore accelerating the execution of the code. In general, these instructions use, the first argument of the subgoal being called to select and jump directly to those matching clauses.

2.2 Parallelism in Logic Programming

As we have seen earlier, the SLD operation $selet_{clause}$ in Prolog choses the clauses by the order which they are written in the program and the $select_{literal}$ operation selects the subgoals from left to right. But in fact, if we consider only pure logic programs, clauses and literals can be selected in any other order without affecting the meaning of the program. This is an important characteristic that we want to take advantage of for exploring parallelism [15].

Exploring parallelism at the level of the SLD operations has one main advantage: we can reuse the programs as originally written for sequential machines without the need of any change. This kind of parallelism is called *implicit parallelism*. There are two main sources of implicit parallelism in Prolog [12], that we discuss next in more detail.

Or-parallelism arises by the parallelization of the $select_{clause}$ operation. It explores in parallel the multiple clauses that match a given subgoal, i.e., it corresponds to the parallel execution of the bodies of the alternative matching clauses. Or-parallelism was implemented with success in many Prolog systems, being the Aurora [28] and the MUSE [4] systems the most well known.

And-parallelism arises by the parallelization of the $select_{literal}$ operation. So, it corresponds to the parallel execution of the subgoals in the body of a clause. And-parallelism can be subdivided itself in two categories:

- The first is called *Independent And-parallelism*, which only exploits in parallel subgoals that are independent. Two subgoals are considered independent if they do not share unbound variables or if the bindings to common variables produced in one subgoal do not interfere with the computation of the other subgoals [23]. That restriction avoids the possible competition in the creation of bindings. This approach was implemented in systems like &-Prolog [22] and &ACE [30, 31].

- The second is called *Dependent And-parallelism* and, as the name suggests, it is the opposite of the previous approach. Dependent And-parallelism can be implemented using two different strategies. In the first one, body subgoals are run concurrently and each one can do its own bindings. After a conflicting binding (a binding to a shared variable) [46] or at the end of the computation [24], an extra step is needed to verify the consistency of the bindings produced. The second strategy consists in allowing only one subgoal to do the bindings to a certain variable (producer), while the other subgoals can only access it in a read only mode (consumers). Several proposals in how to manage consumers and producers can be seen in [20]. Systems like DASWAM [42, 41] and ACE [29], implement support for Dependent And-parallelism.

Another source of implicit parallelism that can be found in logic programs is called *Unification Parallelism* [7] and it arises from the parallelization of the unification process. This kind of parallelism is usually fine-grained and because of that it has not received much attention from the community.

For running a program in parallel using a system that only explores one form of parallelism, first we would need to find out the predominant kind of parallelism present in the program in order to be able to select the system that best fits our needs. Even though this is the case, we would be wasting sources of parallelism since we are throwing away the less predominant type of parallelism. Systems like Andorra-I [40] and ACE [21] were designed to explore the two major types of parallelism simultaneously. In theory they should be able to achieve better speedups than the other systems but, because of their extreme complexity, this goal has not been achieved so far by any system.

The problems of combining the two types of parallelism are more than the sum of the problems of both approaches. For or-parallelism, the most problematic challenge is dealing with the different bindings that can be done to the same variable during the execution. With and-parallelism, the problem is quite different since workers exploring different subgoals of the same clause must be able to freely access the bindings created

by other workers. For that reason, the needs of or-parallelism and and-parallelism in terms of bindings manipulation seem to be antithetical.

In the old years, the research in parallel Prolog has been focused mainly in shared memory models and, for that reason, all the systems/models that we have mentioned above were proposed targeting that kind of architectures. More recently, as distributed memory systems became more affordable, examples of distributed Prolog systems were also proposed [6, 49, 43, 34]. For our research, we are interested in or-parallelism for both shared and distributed memory architectures.

2.2.1 Or-Parallelism

If we view the computation of a Prolog program as a tree, like the one represented in Fig. 2.1, exploiting or-parallelism corresponds to exploiting more than one branch at the same time and, for that reason, we call that tree an *or-tree*. Thus, at a first glance, implementing an or-parallel system seems an easy task since the branches (i.e, clauses), which are tried at the same time are independent from each other. But, in fact, there are important problems that arise when we are extending a sequential Prolog system to support or-parallelism. We begin this subsection by describing these problems and by presenting some of the proposals to solve them. Then, we discuss in more detail some of the or-parallel models introduced before and finally we focus on the Yap Prolog system and its different proposals to support parallelism.

2.2.1.1 Multiple Bindings Problem

A major problem when implementing an or-parallel Prolog system is the *multiple bindings problem*. Figure 2.3 illustrates this problem. At the left side of the figure we have the code of a program and, at the right side, the execution tree for query *p*. The query unifies with the first clause of the program and since variable X appears in the body subgoal *q(X)*, an entry for X is created in the heap. Later, when subgoal *q(X)* is called, there are two clauses that match with *q(X)* and thus, if running the program in parallel, there can be two workers dealing with each clause concurrently. Both workers will try then to do a conditional binding to variable X, one trying to bind it with value 1 while the other trying to bind it with value 2, which leads us to the multiple bindings problem. Note that this problem does not arise in sequential systems. With or-parallelism, variables may have different bindings at the same time and for that reason, each worker should have its *own private area* where it can manage

its conditional bindings without interfering with the work of the other workers.

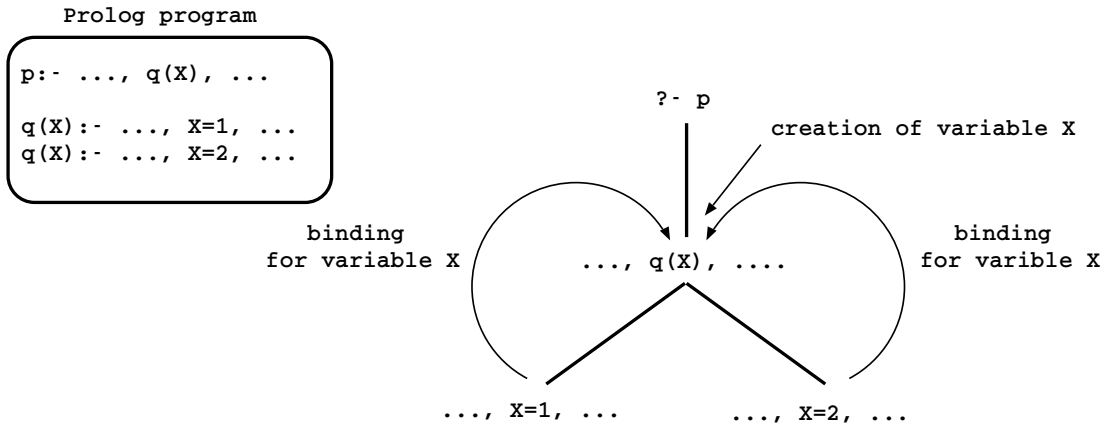


Figure 2.3: The multiple bindings problem

Many models were proposed to deal with the multiple bindings problem. A comprehensive list and explanation for each model can be seen in [16]. An important aspect that differentiates different models is that they can have different computational costs. Gupta and Jayaraman [18] defined the criteria for an optimal or-parallel system:

- The cost of environment creation should be constant-time;
- The cost of variable access and binding should be constant-time;
- The cost of switching from a task to another should be constant-time.

So far, none of the proposals in the literature was able to fulfill these three requirements simultaneously. Despite that, a good implementation should be able to mitigate such problem by avoiding the more expensive tasks. The most well-known proposals are the *binding arrays* [54, 53] and the *environment copy* [4] models.

In the binding arrays model, the system is extended so that each worker has a counter, used to enumerate the variables found during the execution of the program and an auxiliary array where it stores its conditional bindings. To better understand this model, let us see an example. In Fig. 2.4 we have the same example used in Fig. 2.3 but now using the binding arrays model to avoid the binding conflicts. Again, we begin by executing the query p that matches with one single clause in the program. This clause then originates a call to *q(X)* and, since X is a new variable, it is assigned with the current value of the counter, value 0 in this case since it is the first variable being created. Then the counter is incremented. Each worker then uses this value

to make its own bindings independently since the bindings are stored in the auxiliary array in the position assigned for variable X, position 0 in this case.

Later, when a worker moves from one branch to another, as a consequence of not having more work in its current branch, it must update its bindings by deinstalling the old bindings in its binding array and by installing the bindings in the new branch. Because of that and regarding the criteria defined by Gupta and Jayaraman, this model is considered not to have constant cost of task switching, while the other costs – environment creation and variable access – are constant. This model was first introduced in the Aurora system and was later used in other systems like Andorra-I. The binding arrays model was also the base for other models that combine or-parallelism with and-parallelism, such as Shared Paged Binding Array [17] and SBA [13].

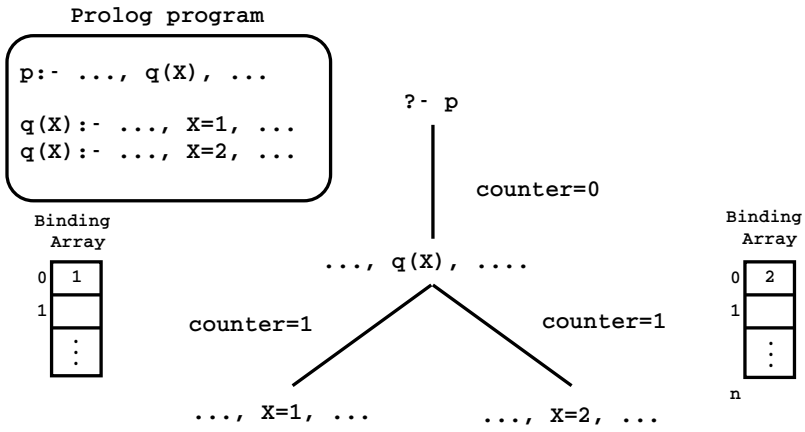


Figure 2.4: The binding arrays model

An alternative successful model for solving the multiple bindings problem is the environment copying model, which was first proposed by the MUSE system and later adopted by many others [50, 35]. In this model, each worker keeps a separate copy of its own environment, thus the bindings to shared variables are done as usual without conflicts, i.e., stored in the private execution stacks of the worker doing the binding. Every time a worker shares work with another worker, all the execution stacks are copied to ensure that the requesting worker has the same environment state down to the search tree node where the sharing occurs. As a result of environment copying, each worker can proceed with the execution exactly as a sequential engine, with just minimal synchronization with other workers. Synchronization is mostly needed when updating scheduling information and when accessing shared nodes in order to ensure that unexplored alternatives are only exploited by one worker. All other WAM data structures, such as the environment frames, the heap, and the trail do not require synchronization. Regarding the criteria defined by Gupta and Jayaraman, this model

has the same characteristics as the previous, constant cost of environment creation and variable access and non-constant cost of task switching.

Both environment copying and binding arrays models were first developed having shared memory architectures in mind. More recently, environment copying has been successfully adapted for distributed memory architectures with only a few minor modifications [9, 49]. Since, in this thesis, we are interested in exploring or-parallelism in both shared and distributed models this is one of the reasons we have focused our study in the environment copying model.

2.2.1.2 Scheduling

A major challenge of any parallel system is the implementation of efficient scheduling strategies to distribute work among workers. An or-parallel Prolog system is not an exception and many strategies exist in the literature [10, 3, 8, 44, 43]. Due to the dynamic nature of Prolog work, it is impossible to assign work in a balanced way to the workers at the beginning of the execution of a program. In the specific case of or-parallelism, open alternatives appear irregularly in the branches of the or-tree. Therefore, we need a scheduler able to distribute work dynamically during execution time. The interaction between workers and the need of workers to switch from task to task are the two situations that the scheduler must minimize since they have a great impact on the performance of the whole system.

The scheduler is also responsible for maintaining the sequential semantics of Prolog, meaning that we should get the same output as in a sequential system. This is a problem when the program contains predicates dealing with I/O, side-effects and the cut predicate (!). In the case of the cut predicate, there is another problem that the scheduler must consider that is known as the *speculative work problem*. In a sequential system, when a cut is executed, all the alternatives at the right side and below the scope of the cut are pruned away. In a parallel system, if one of those alternatives is picked earlier than the cut be executed, it will result in wasted work. Therefore, the scheduler must avoid giving this kind of work by selecting first the available work that is closest to the left side of the or-tree.

There are two major policies for dispatching work in or-parallel execution, namely *topmost* and *bottommost* dispatching of work. The topmost policy gives priority to the exploration of the nodes closer to the root, which are expected to hold more work but are more susceptible of originating more sharing operations. The bottommost

policy gives priority to sharing all the available work, which has the disadvantage of opening large public regions but, on the other hand, it reduces the number of task switching operations.

Two of the most successful or-parallel schedulers, proposed for the Aurora and Muse systems, divide the or-tree in two parts: *public* and *private*. Public nodes of the or-tree are nodes that are shared by more than one worker. The execution of alternatives stored in public nodes requires some type of synchronization, while alternatives in private nodes can be executed as in a sequential system. In these systems, a worker explores first their private nodes and only when there is no more available work left to try, it starts exploring public nodes. When all open alternatives of a worker, private and shared, have been explored, it consults the scheduler in order to discover a busy worker available to publish and share its open alternatives.

Since these kind of schedulers were designed for shared memory architectures, this means that they can easily decide which worker should be selected to share work with an idle worker. Thus, as they have a complete representation of the or-tree in shared memory, they can use such information to make the best decision, for example, by selecting a worker very close to the left side of the tree and with many open alternatives. In systems like PALS [49] and OPERA [9], designed for distributed memory architectures, the schedulers do not have the entire representation of the or-tree, since this would require too many messages exchange between workers in order to have such information up-to-date. Instead, workers send, from time to time, information about their work loads and it is based on these workloads that the scheduler does its choices.

2.2.2 Environment Copying Models

In this subsection, we present two of the most successful parallel execution models based on environment copy. The first one was first proposed for the MUSE system and was developed and designed to take advantage of shared memory architectures. The other model, originally designed for the PALS system, is closely related with MUSE but was specially tailored for distributed memory systems. The main difference between these two models is related with the scheduler and, in particular, with the process of sharing work. In both models a worker enters in *scheduling mode* when it becomes *idle* (without work) and the scheduler is then responsible for finding a *busy* worker that can share its open alternatives. Once it finds such a worker, the sharing

work process begins.

In the MUSE system, the first step of the sharing work process is *publishing the private nodes of the busy worker*. The publishing process involves associating a new data structure, called *or-frame*, to each private choice-point. Figure 2.5 shows the relation between or-frames and choice-points. On the left side of the figure we can see the worker P execution tree with white nodes representing choice points with open alternatives and the black nodes representing choice points fully explored. Open alternatives are represented by dotted lines. On the middle of the figure, we have a common global area where the or-frames are store and, on the right side of the figure, the local stack containing the set of stored choice points. The figure is also divided horizontally in two areas: a private and a shared area. In the private area, choice points point directly to the next open alternative while, in the public area, an indirection is created with the choice points pointing to the corresponding or-frame and then the or-frame pointing to the next open alternative. On the shared area, when a node is fully explored, i.e. it has no open alternatives, its associated or-frame is made to point to NULL. This may happen if, in the meantime, another worker has explored the remaining alternatives in that node.

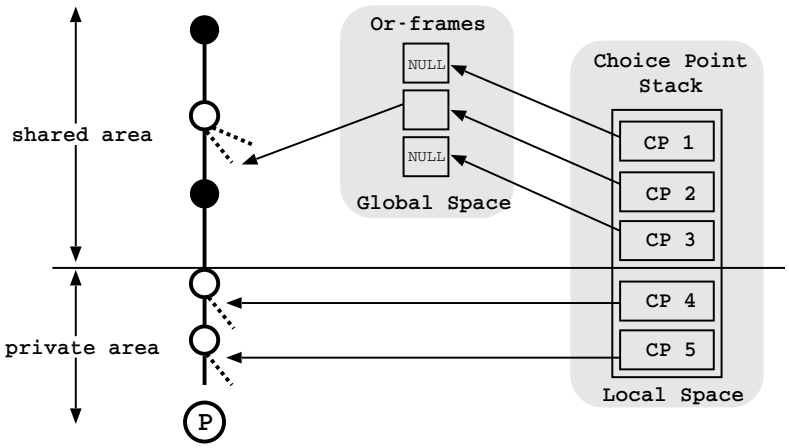


Figure 2.5: Relation between or-frames, choice-points, private and shared areas

Now consider that worker P receives a sharing request from a worker Q. In such case, P would first need to publish his two private choice points as we have seen before. The choice points will be made to point to the new corresponding or-frames and then each or-frame point to the next open alternative previously stored on the associated choice point. Figure 2.6 shows the schematic representation after that operation. Note that the private area disappeared since the bottommost policy implies that all private choice points must be published to be shared with the requesting worker.

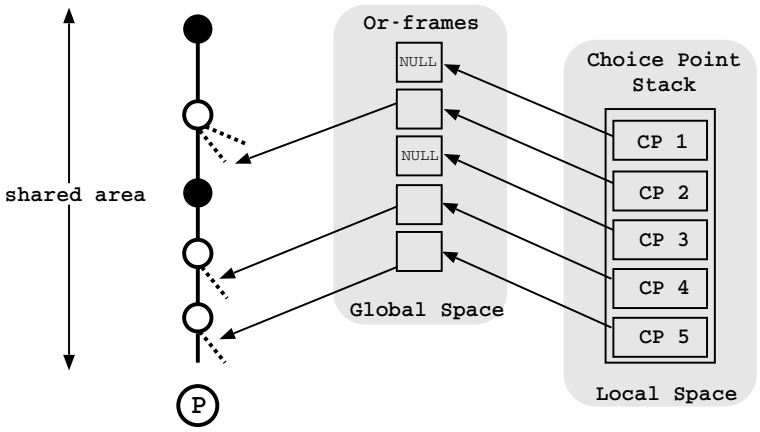
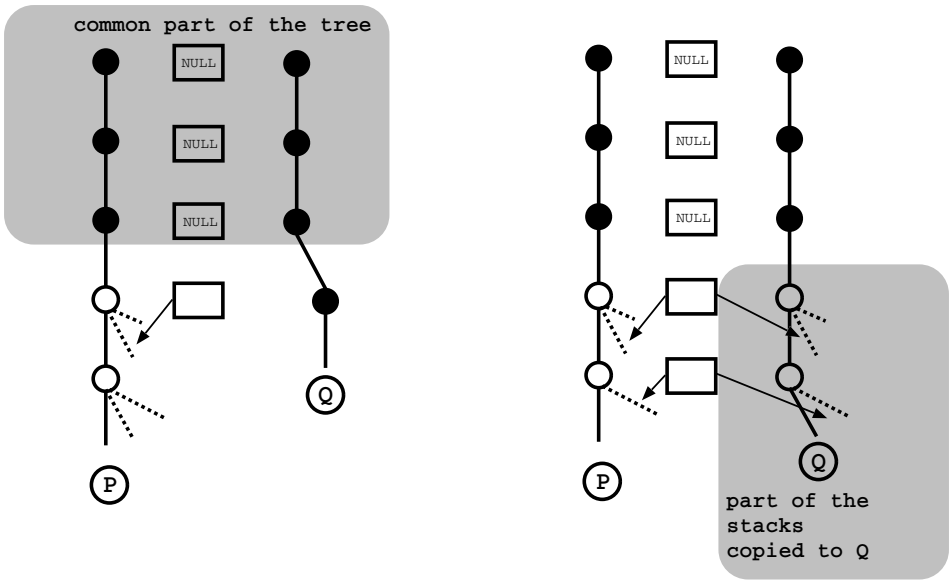


Figure 2.6: Publishing private nodes

The or-frame structure gives support to a dynamic distribution of work by guaranteeing a synchronized access to the open alternatives and by avoiding that two or more workers explore the same alternative. Moreover, each or-frame has information about the workers sharing the corresponding node (i.e, holding a choice-point pointing to the *or-frame*), and includes a pointer to the parent or-frame, which allows for a full representation of the public or-tree. These two characteristics together greatly help the scheduler in its task of distributing work efficiently by the available workers.

After the initial process of publishing the private nodes, the stacks can now be copied from the busy worker to the idle one. The copy process can be optimized using a technique called *incremental copy* [4]. Incremental copy is an optimization of the copy process which avoids copying common parts of the stacks of two workers. Figure 2.7 illustrates this process. In Fig. 2.7(a), we have the state before copying where worker Q is idle and worker P is willing to share its open alternatives with Q. The common choice points of the two workers are depicted in grey background and, as expected, all those choice points have no open alternatives. Worker Q has one more choice point which is private and has no open alternatives either. Worker P has two more choice points, one shared with a third worker and another one private, both with open alternatives. Fig. 2.7(b) illustrates the state after copying. Here, we can see that only the two non-common choice points of P were copied to Q (part depicted in grey background). It is important to note that the previous private choice point in P has now an or-frame associated to it and that the other choice point, that was already shared with a third worker, is now also associated with worker Q.

Despite the good performance that the or-frames based model has shown in shared memory, it is not suitable for distributed memory architectures. Since maintaining



(a) Initial state (before copying) (b) After copying the stacks from P to Q

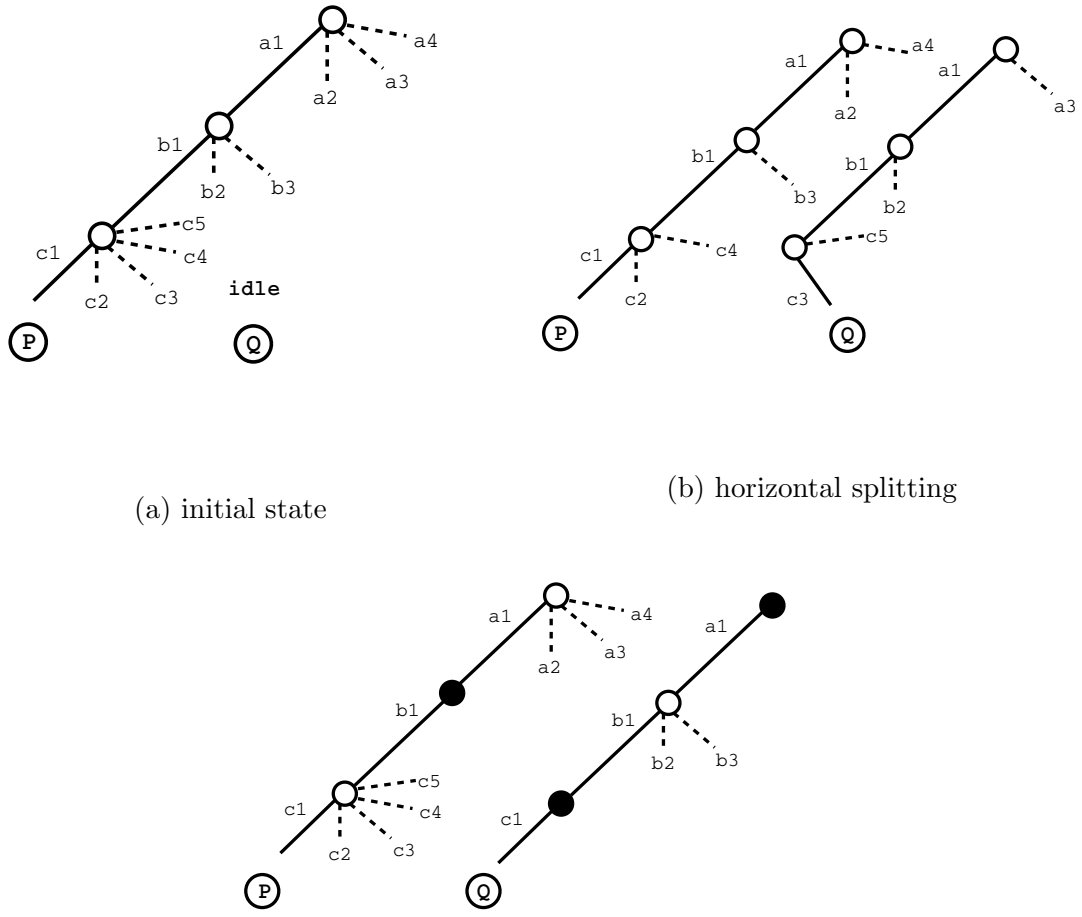
Figure 2.7: Representation of the vertical splitting operation done by a sharing worker

a structure similar to the or-frames in a distributed environment would be very inefficient. Each time a worker would move in the tree or pick a new alternative, it should synchronize such operation with all the other workers sharing the same branch and we know that synchronization messages in a distributed environment results in a great impact for the performance of the system.

To distribute work in distributed memory architectures, the PALS system uses a variation of environment copying, named *stack splitting* [19]. In this model, in order to avoid the multiple access to the open alternatives, the work is assigned statically when the sharing process occurs, so that each worker knows beforehand which alternatives belong to it. This reduces the amount of communication between workers, thus making this strategy more suitable for distributed memory architectures.

Figure 2.8 illustrates two different stack splitting strategies. In Fig. 2.8(a) we have the configuration before sharing, where worker P has 3 choice points with 9 open alternatives while worker Q is idle and waiting for worker P to share work with it. Figure 2.8(b) and Fig. 2.8(c) then show the configuration after sharing. In Fig. 2.8(b), the strategy adopted is called *horizontal splitting* and the open alternatives in a choice point are alternatively divided between both workers. In Fig. 2.8(c), the strategy adopted is called *vertical splitting* and the open choice points are alternatively divided between both workers.

With *stack splitting* workers do not have a complete vision of the or-tree. The scheduler bases its choices in fewer and less updated information, since messages are sent from time to time and, usually, the only information present in that kind of messages is the load of the worker sending it. Nevertheless, results showed that similar or close speedups to those achieved by MUSE are possible and for some of the benchmarks the results are even better [19].



(a) initial state

(b) horizontal splitting

(c) vertical splitting

Figure 2.8: In (a) worker Q is idle and waiting for worker P to share work with it; in (b) P shares work using horizontal splitting; and in (c) P shares work using vertical splitting

2.3 Parallelism in the Yap Prolog system

During the past years, many or-parallel models were developed for the Yap Prolog system. The first one to be implemented was YapOr which is based on the original MUSE approach and uses or-frames to synchronize the access to the open alternatives [35].

Later on, a model named *copy-on-write for logic programs* (COWL) [36] was implemented. This model is similar to environment copy since each worker maintains its own execution stacks and uses or-frames to synchronize the access to the open alternatives. The main difference between the two models resides on how the or-parallel work is shared. In COWL this process works as follows: consider that worker Q has no more alternatives to explore and wants to explore alternatives from worker P. To share work, worker P performs a *fork()* and worker Q is made to execute the new child process created by the system call. After a *fork()*, the operating systems uses a technique, called copy-on-write, that only copies pages of memory when the parent on child processes want to write on them. The COWL model takes advantage of such characteristic in order to reduce overheads of copying the stacks.

Another model is the Sparse Binding Array (SBA) [13] which is based on the original binding array model. In the SBA model each worker has its own shadow of the shared stack space. When a given worker wants to perform a conditional binding to a shared variable he does it on its shadow space. In order to synchronize the access to open alternatives this model also uses or-frames. Comparing the three systems implemented on top of Yap, experimental results showed that the environment copy model is the one showing best performance [39].

More recently, a new approach to YapOr was done. The new system, called ThOr [37], maintains the main concepts of YapOr unaltered but implements workers as POSIX threads rather than processes, as in the original YapOr. Both approaches, YapOr and ThOr, have shown similar speed ups for a set of benchmarks [37].

The static division of work using stack splitting was also implemented in Yap using two different approaches. The first one, called YapDSS, was designed to run in distributed memory in Beowulf clusters [34]. The second one was designed to run in multicores by extending the YapOr model to support stack splitting and it has shown to be very competitive when compared with the original YapOr approach [48, 47].

A summary of the main characteristics and differences of these models can be seen in Table 2.1.

Table 2.1: Main characteristics of the or-parallel models implemented in Yap

Model/Approach	memory	workers	binding scheme	distrib. of work
YapOr	shared	processes	env. copying	or-frames
COWL	shared	processes	env. copying	or-frames
SBA	shared	processes	binding arrays	or-frames
ThOr	shared	threads	env. copying	or-frames
YapDSS	distributed	processes	env. copying	stack splitting
YapOr (stack splitting)	shared	processes	env. copying	stack splitting

2.4 Concept of Teams

The concept of teams was originally proposed by the Andorra-I system in order to be able to exploit the combination of and-parallelism with or-parallelism. The or-parallel component of Andorra-I follows the Aurora system that uses binding arrays to solve the multiple bindings problem, while the and-parallel component was built to exploit dependent and-parallelism.

In Andorra-I, or-parallelism is exploited at the team level meaning that, from the outside, each team behaves (and can be seen) like a single Aurora engine (or worker). Inside a team, and-parallelism is explored by several workers that cooperate in order to solve the subgoals present in the or-branch assigned to the team. This team concept is implemented by defining a master worker inside each team that is responsible for allocating the new choice points and by exploring the or-parallelism, while the remaining workers, called slaves, cooperate by exploiting the and-parallelism in the choice point at hand.

The system is composed by two schedulers, one for each type of parallelism. In the early versions of Andorra-I, the number of workers per team had to be defined statically before the execution begins. Later, a *top-scheduler* [14] was proposed to dynamically adapt, during the execution, the number of workers per team to the type of predominant parallelism being explored at a given instant of time. When or-parallelism is predominant, the slaves may became master and constitute their own team. On the other hand, when and-parallelism is predominant the masters are able to become slaves and join other teams.

The concept of teams was later used by some versions of the ACE system and by the Paged Binding Array (PBA) model [17] to explore the combination of or-parallelism with independent and-parallelism.

The ACE system uses environment copying to deal with the multiple bindings problem and it can be viewed as a combination of the MUSE system with the exploitation of dependent and-parallelism. When a query is executed in the ACE system, at the beginning, only a single team is responsible for exploring its and-parallelism and once a choice point is created the remaining teams may start requesting the or-work present on it. The conceptual sharing process is similar to the one found in MUSE with segments of memory being copied from a busy team to an idle team. After that, the requesting team takes the next untried alternative in the new shared work and, then, the workers inside the team begin exploring the and-parallelism inside that (or-)alternative.

The PBA model extends the binding array model in order to support the combination of or-parallelism with independent and-parallelism. In this model, the binding array present on each team is divided in pages with each page being assigned to a worker of the team, so that each worker can do its own bindings without interfering with the bindings of the other workers.

For the purpose of our thesis, we will assume that, conceptually, a team is a set of workers (processes or threads) who share the same address space and that cooperate to solve a certain part of the main problem.

Chapter 3

Layered Model

In this chapter, we present the key concepts of our proposal in a high level approach. We begin by briefly introducing our model. Next, we describe the syntax developed to interact with it. Finally, we show a small and practical example to highlight the potential of the model.

3.1 Overview

To the best of our knowledge, none of the models proposed in the literature is able to take advantage of architectures which combine shared and distributed memory such as ones based on clusters of multicores. As we have seen in the previous chapter, the shared memory based models take advantage of synchronization mechanisms that cannot be easily extended to distributed environments while the distributed memory based models use specialized communication mechanisms that do not take advantage of the fact that some workers can be sharing memory resources. The goal behind our model is to implement the concept of teams in order to be able to explore such combination while trying to reuse, as much as possible, Yap's existing infrastructure.

We define a team as a set of workers (processes or threads) who share the same memory address space and cooperate to solve a certain part of the main problem. By demanding that all workers inside a team share the same address space implies that all workers should be in the same computer node. On the other hand, we also want to be possible to have several teams in a computer node or distributed by other nodes.

For workers inside a team, we can thus distribute work using both or-frames or

stack splitting. For distributing work among teams, we can apply any of the stack splitting strategies described before. This idea is similar to the MPI/OpenMP hybrid programming pattern, where MPI is usually used to communicate work among workers in different computer nodes and OpenMP is used to communicate work among workers in the same node.

In order to take advantage of our model, we also propose a new syntax which follows two important design rules. The first one is delegating to the user the responsibility of explicitly declare which parts of the program should be run in parallel. The second one is the ability of interacting asynchronously with the parallel engine for fetching for answers. More details about this new syntax are presented in the next section.

To better understand our model, consider the schematic representation shown in Fig. 3.1. On the left side of the figure, we have Yap's console (or *client worker*) which is a sequential Yap engine responsible for interacting with the user. On the right side of the figure, we can see a parallel engine E constituted by a cluster of two computer nodes - host node N1 and host node N2. In host node N1, there are two teams, Team A and Team B, with four workers each while, in N2, there is only one team, with eight workers named Team C. Both Team A and Team B use or-frames to distribute work inside the team while Team C uses stack splitting.

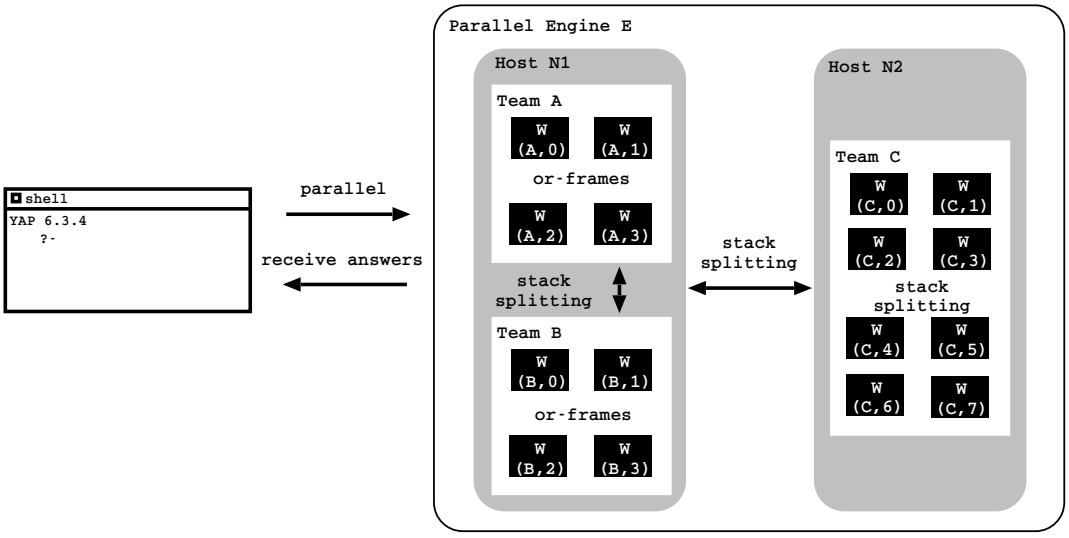


Figure 3.1: Schematic representation of our layered model

The execution of our layered model is very straightforward. At the beginning the client worker is responsible for starting the execution of the user queries and running sequential Prolog code. If during the execution of Prolog code, a goal marked to be

run in parallel is found, it is sent to the parallel engine to be executed and since our syntax allows asynchronous interaction with the parallel engine, the client worker may continue executing code and check, from time to time, the state of the parallel execution.

In our model, each team in the parallel engine has one worker responsible for controlling the execution inside the team, called the *master worker*. Moreover, one of these master workers will be responsible for receiving and starting the execution of the parallel goal sent by the client worker and, for that reason, its team is called *master team*.

After the master worker of the master team starts executing the goal it will inform the remaining workers inside its team and the other teams in the parallel engine that a new parallel execution has begun. After this notification, its teammates will start sending it sharing requests in order to get work. It may then start sharing work with them using the scheduling strategy defined in its own team. Then the execution continues with the workers sharing work between them and cooperating to execute the work available. At this point, the other teams are now also aware that the master team has work, so they will start sending it sharing requests. When the master team receives a request it selects a sharing worker to fulfill it. The work will then be divided using stack splitting and sent to the master worker of the requesting team, which is responsible for starting its execution and informing its teammates about the existence of work inside the team, so that they can cooperate in the exploration of that work.

A team is considered to be out of work when all of its workers are idle. When that happen it must contact a busy team in order to get work and repeat the process described above. If all the teams are idle that means that the parallel goal is completed explored and the client worker is notified that the execution has finished.

In our implementation, that will be described in detail in Chapter 5, we only use exclusively or-frames to distribute work inside the the team while for distributing work between teams both horizontal and vertical splitting are available.

3.2 Syntax

In this section, we present the built-in predicates that constitute the syntax to interact with a parallel engine in our layered model. It includes the following five predicates:

par_create_parallel_engine/2

When the programmer plans to run parallel goals, this is the predicate that should be called first. It is responsible for creating and launching the teams that form a new parallel engine. As arguments it receives the name to be given to the new parallel engine and a list of tuples. Each tuple in the list represents one team and has information that is used by the underlying engine to know in which computer node that team should be launched, how many workers must be spawned and the location of the file containing the program that must be loaded by that team. For example, the following call could be used to create the topology illustrated in Fig. 3.1:

par_create_parallel_engine(E,[(N1,4,'prolog/prog.pl'), (N1,4,'prolog/prog.pl'), (N2,8,'prolog/prog.pl')]).

par_run_goal/3

As the name suggests, this is the predicate used to indicate that a given goal should be run in parallel. The predicate receives as arguments the name of the parallel engine where to run the goal, the goal to be run and a template indicating how the answers to the given goal should be returned. Consider that we want to run, in parallel the goal *do_something(X,Y,Z)* and we are only interested in the answers obtained for variable *Z*. In such case, the template could be defined as *Z*. If we want to run the query in the previous parallel engine, thus we could write:

par_run_goal(E, do_something(X,Y,Z),Z).

par_probe_answers/1

This predicate is used to check if a parallel execution has already found any answer. It receives as argument the name of the parallel engine that we are willing to probe for answers. Using again the current example would be:

par_probe_answers(E).

It succeeds if the parallel engine has found any answer or if the parallel execution has already finished. Otherwise, it fails.

par_get_answers/4

The predicate *par_get_answers/4* allows to retrieve answers asynchronously. It receives four arguments. Again, the first argument is the name of the parallel engine. The second argument specifies options about the reception of answers:

- **max(N)** – this option says that the user is willing to receive a maximum of N answers, meaning that it can receive from one up to N answers, depending on the number of answers that the parallel engine has found till that moment.
- **exact(N)** - this option is more restrictive, the predicate will block and wait until the parallel engine has exactly N answers to return or the execution has finished.

For both options the variable N can be unified with the constant which represents the total of answers in the program at the moment of the call. The remaining two arguments of predicate *par_get_answers/4* are variables, the first one returns the list of the answers found and the second the length of that list. This predicate is not backtrackable, however when called again it will return a new set of answers (if the parallel goal at hand has produced new answers). When all answers have been retrieved and the parallel program has finished, the predicate simply fails. Using the current example, if we want to retrieve exactly four answers we could write:

par_get_answers(E, exact(4), AnswersList, AnswersNumber).

par_free_parallel_engine/1

This predicate is used to free a parallel engine when it is no longer needed. Using again the current example would be:

par_free_parallel_engine(E).

In order to show the potentialities of our syntax model, Fig 3.2 presents a full example of its usage. Again, we are assuming the topology presented earlier. The file containing the program is initially loaded by the Yap's client. The query to be executed is triggered by the last line of code, which runs a *par_computation/1* goal in parallel and, at the end, sums in the client side the set of answers returned by the parallel engine.

Predicate create/0 is responsible for configuring and starting the parallel engine E. While predicate *run/0* launches the parallel execution of the goal *par_computation/1*.

```
create:-
   par_create_parallel_engine(E,[(N1,4,'prolog/prog.pl'),
                                 (N1,4,'prolog/prog.pl'),
                                 (N2,8,'prolog/prog.pl')]).

run:-
    par_parallel_run(E,par_computation(S),S).

wait:-
    not par_probe_answers(E),
    !,
    writeln('waiting for answers...'),
    wait.
wait.

answers(Result):-
    par_get_answers(engine_E, max(all), AnswersList, AnswersNumber),
    !,
    sum_answers(AnswersList, Sum),
    answers(Result1),
    Result is Sum + Result1.
answers(0).

sum_answers([], 0).
sum_answers([Head | Tail], Sum) :-
    sum_answers(Tail, Sum1),
    Sum is Head + Sum1.

free:-
    par_free_parallel_engine(engine_E).

:- create, run, wait, answers(Result), writeln(Result), free.
```

Figure 3.2: A small example showing the usage of our syntax

The code of this predicate is in the file *prog.pl*. After the parallel execution being launched the predicate *wait/0* will be probing for answers and writing to the console 'waiting for answers...'. When the parallel program has found at least one answer the predicate *par_probe_answers/1* will fail and the predicate *answers/0* will be called in the continuation. This predicate will be responsible for summing all the answers found by the parallel engine. In order to do that, it begins by calling the predicate *par_get_answers/4* with the option *max(all)*, which will make the predicate to fetch the maximum amount of answers found until that moment. After that, the answers returned will be summed using the predicate *sum_answers/2*. As we can see, the client program does not need to wait until the parallel computation has finished to start doing some computation with the answers already found. The predicate *answers/1* is called recursively till there are no more answers returned by the predicate *par_get_answers/4* and it fails. Finally, the sum result will be shown in the console and the parallel engine is freed.

Chapter 4

The YapOr System

This chapter gives an overview of the YapOr system [35], an or-parallel engine implemented on top of the Yap Prolog system. We begin by describing the concepts behind its model and then we show the main extensions done to Yap in order to implement the support for or-parallelism.

4.1 Overview

The YapOr system is based on the environment copying model, as originally proposed for the MUSE system, and was designed to run in shared memory architectures. We present next the key concepts and components behind its model.

4.1.1 Basic Execution Model

The YapOr's parallel engine is constituted by a fixed number of workers that should be specified by the user when starting the system. When the execution of a Prolog goal begins, all workers are idle except one (that we will call it worker P). This worker P is responsible for executing code in sequential mode, as a common Prolog engine would do, until it finds a parallel directive which makes it to enter in parallel mode.

After entering in parallel mode, worker P informs the idle workers that a parallel execution is beginning and, after that, starts executing code and generating choice points to the subgoals that have more than one alternative, as usual.

On the other hand, the idle workers will start sending share requests to worker P so that they can begin to cooperate in the computation. Now consider that an idle worker Q requests work to the busy worker P. If P has unexploited work, the sharing process begins, which involves two main steps:

- worker P publishes its private choice points, which consists in associating an or-frame to each choice point (remember that or-frames allow workers to synchronize the access to the open alternatives present in public choice points);
- worker P copies its execution stacks to worker Q.

After these two steps, both workers are exactly in the same computational state. Worker P will then resume its computation from where it was while worker Q simulates a failure in order to force the Prolog backtracking mechanism to enter in action and lead worker Q to take the next open alternative available from the shared state received from P.

At this point, both workers P and Q have work and any other idle worker may request work to them. When a worker is idle, the scheduler is responsible for finding a busy worker with open alternatives to ask for sharing work, as described above. When there is no more work available to explore the computation ends and all workers stay idle waiting for the beginning of a new computation.

4.1.2 Incremental Copying

The operation of copying stacks from one worker to another might have a great impact on the performance of the system. To minimize such impact, YapOr implements an incremental copying technique [4]. With the incremental copying technique, worker Q maintains the parts of the stacks that it has in common with P and only the differences are copied from worker P to Q, thus reducing the total amount of memory to copy.

Figure 4.1 illustrates in detail how the incremental copying technique works. On the left side, we have the execution tree where we can see that the idle worker Q shares the three top nodes (or choice points) with P. On the left side, we have the representation of the worker P's stacks with the segments to be copied from P to Q colored in grey background. At a first glance, we might think that only the parts of the stacks which are not common to Q need to be copied in order to both workers became in the same state. But, in fact we also need to copy the conditional bindings to the variables

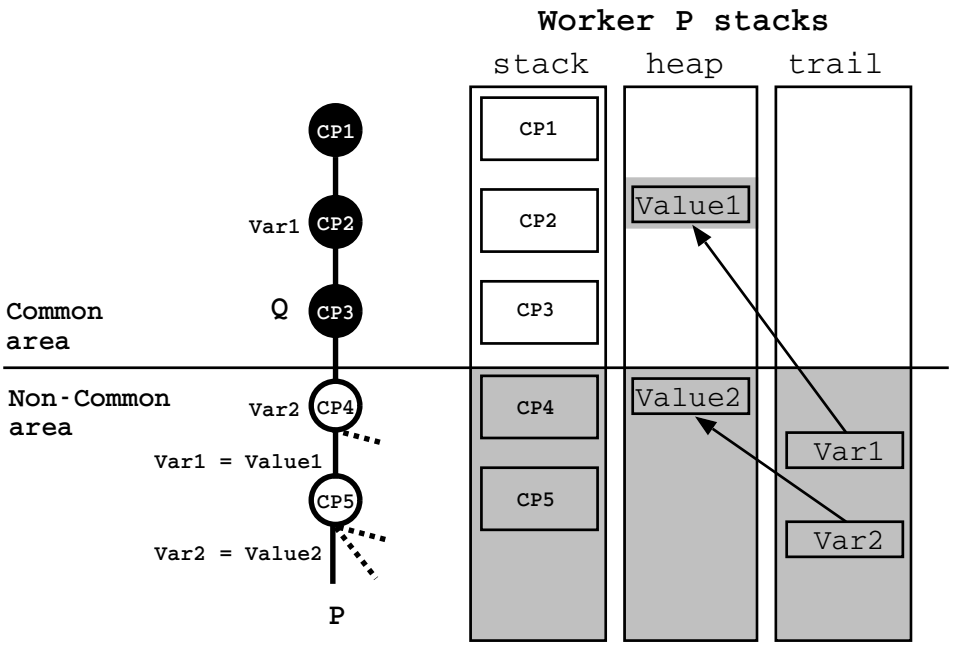


Figure 4.1: Incremental Copying

that belong to (were created in) the common area but were instantiated later in the non-common area. In the figure, *Var1* is one such example. Var1 was created in choice point CP2 but was only instantiated in an alternative of choice point CP4. Remember that when a variable is instantiated, a reference to its position is added to the trail. Therefore, to copy these kinds of bindings, worker Q needs to traverse the copied segment of the trail searching for variables referencing to the common part of the stacks and copy its values in order to became fully consistent and in the same computational state of worker P.

4.1.3 Scheduling

The execution time of a worker in YapOr is divided in two modes: *engine mode* and *scheduling mode*. A worker is said to be in scheduling mode when it is idle and therefore looking for work. After finding new work, a worker enters in engine mode where it runs Prolog code as a sequential engine.

The YapOr scheduler is based on the original MUSE scheduler and it was designed to minimize the execution time while maintaining the original semantics of Prolog. As we have seen earlier, according to the criteria defined by Gupta and Jayaraman, the task switching cost is not constant for the environment copying model. This is

do to the operations done during the sharing work process such as publishing choice points, copying stacks and restoring bindings. For that reason, one of the main goals of the scheduler is to minimize the need for sharing work and, one of the strategies adopted for that, was to force the busy worker to share as much work as possible. That means that a busy worker must share all of its private choice points in order to minimize the possibility that the requesting worker runs out of work too soon. Another strategy adopted by YapOr's scheduler is related with the amount of stacks to copy. It states that the scheduler should give preference to selecting busy workers that are near the requesting worker in the or-tree and which have the highest load (number of unexplored private alternatives). This strategy is tightly related with the incremental copy technique since it also tries to reduce the total amount of memory to copy.

In a nutshell, YapOr's scheduler works in the following way: when a worker runs out of work it tries to find the nearest busy worker with the highest load to share work with it. If there is no such busy worker willing to share work, the scheduler tries to change the position of the idle worker in the or-tree to place it in a position where there are more busy workers near by, thus increasing the probabilities of finding one willing to share its choice points.

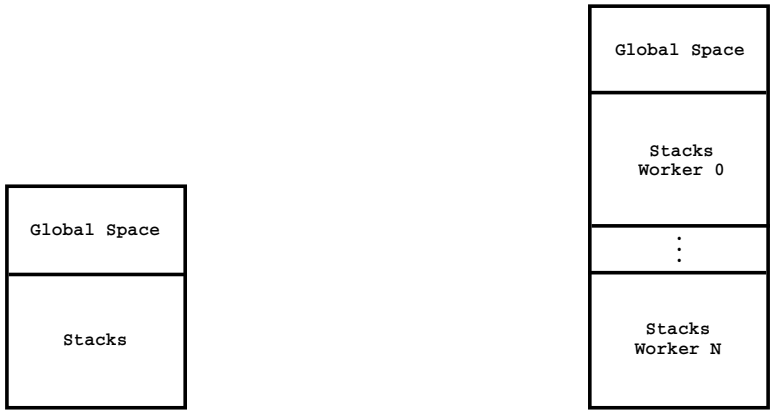
4.2 Implementation Details

In this section, we briefly present the main extensions done to Yap in order to support implicit or-parallelism (for a more complete presentation please see [35]).

4.2.1 Memory Organization

Figure 4.2 shows the differences between the memory layout for the Yap and YapOr systems. In Fig. 4.2(a) we can see Yap's two main memory areas: the global area and the stacks area. The global area includes the code area and the several data structures responsible for supporting the execution of Prolog programs, while the stacks area includes the local stack, the trail and the heap.

YapOr follows the same memory layout of Yap, but with more stacks areas added to accommodate the total amount of workers. This is necessary since with environment copying each worker has it own execution stacks where it can execute code as in a



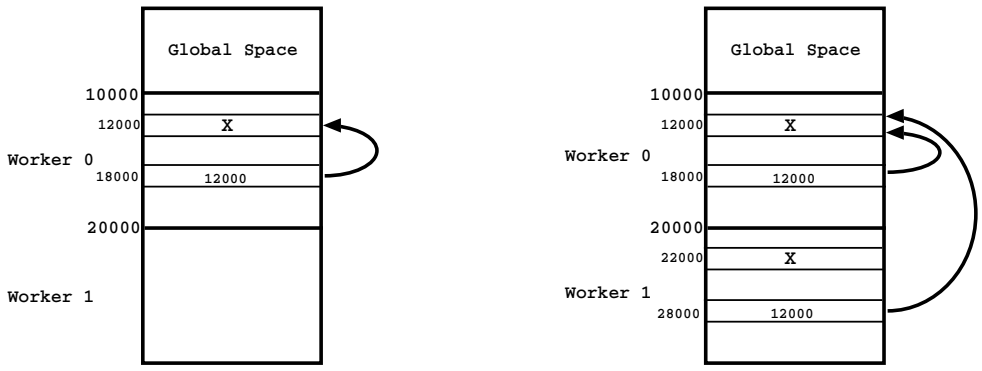
(a) Yap (b) YapOr

Figure 4.2: Memory layout for the (a) Yap and (b) YapOr systems

standard Prolog machine. A schematic view of YapOr's memory layout can be seen in Fig. 4.2(b).

YapOr's memory is allocated as follows. When the system starts, the initial worker (worker 0) is responsible for asking for memory to the operating system using one of the two shared memory schemes available, namely *mmap*, which allows to map a file in memory, and *shmget*, which creates a system shared segment. After that, worker 0 calls the *fork* system call to create the remaining workers which will inherit the previously mapped address space. After that each new worker will need to remap the inherited memory space. The process of memory remapping is necessary in order to allow YapOr to copy memory directly from one worker to another without the need of any post processing.

Let us see first in Fig. 4.3 what would happen if memory is not remapped. On the left side of Fig 4.3 we can see the representation of the stacks of worker 0 and worker 1. The stacks of worker 0 have addresses ranging from 10000 to 19999 while the stacks of worker 1 have addresses ranging from 20000 to 29999. Now consider that part of the stacks from worker 0 will be copied to worker 1, which includes the address at position 18000 that is a pointer to position 12000. On the right side of Fig 4.3, we can see the memory layout after copying. The memory position 18000 of worker 0 was copied to memory position 28000 of the worker 1 and both are pointing to address 12000. From the point of view of worker 1, the value 12000 does not make sense since it is a pointer to the memory space of worker 0. One possible solution to this problem would be to readjust all the addresses in the copied parts of the stacks so that they



(a) Memory before sharing (b) Memory after sharing

Figure 4.3: Copying segments of memory from one worker to another may lead to problems if memory is not remapped

match the address space of the new worker, in the example 28000 would be adjusted to point to address 22000, but this post processing would have a great impact in the performance.

Let us see now how YapOr's remapping process works. In order to do the remapping, each worker rotates the memory so that, from its point of view, the range of addresses of its stacks is the same as the range of addresses of the stacks from the point of view of worker 0. This means that all workers will see their own addresses in the same range of addresses, i.e., the range initially mapped for worker 0. Figure 4.4 illustrates an example with three workers after the remapping process where we can see that all workers, from their point of view, have their stacks ranging between addresses 10000 and 19999 and the remaining worker's stacks addresses are rotated.

Now consider that worker 2 wants to copy its stacks to worker 0. In order to know where to copy it first needs to calculate the offset between it and the receiving worker using the following formula:

$(receiving\_worker - sending\_worker + total\_num\_workers)\%total\_num\_workers * worker\_area\_size$

which in this case would be:

$(0 - 2 + 3)\%3 * 10000 = 10000$

After that, consider that worker 2 wants to copy starting from address 15000, the destination will be calculated by summing the offset to that address, giving as result

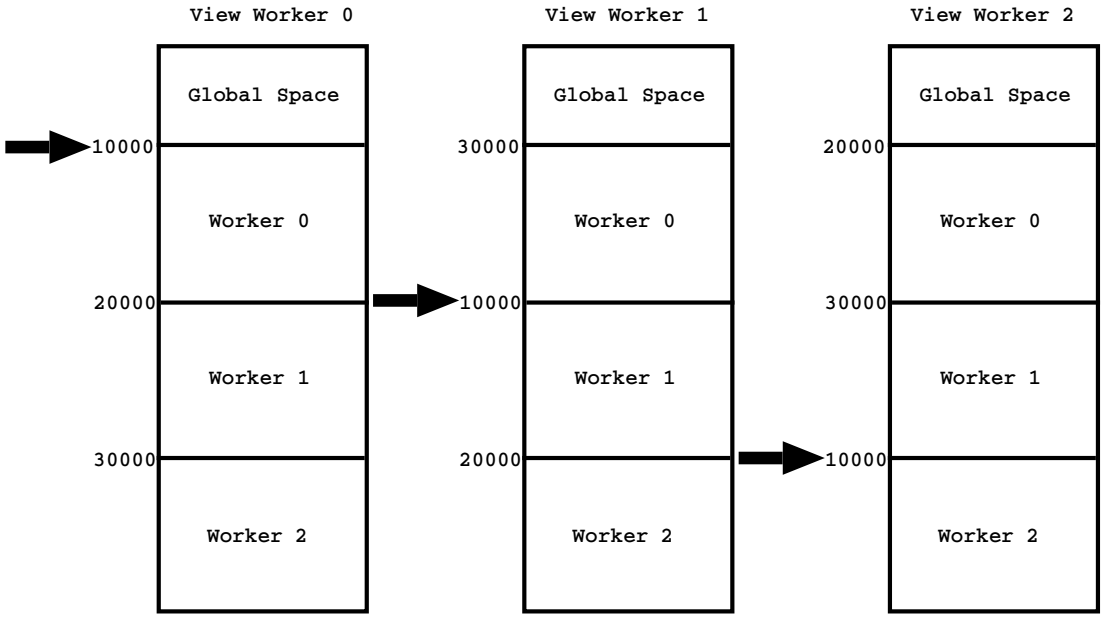


Figure 4.4: Memory addresses from the point of view of each worker after YapOr's remapping process

25000, which from its point of view is an address in the worker 0 memory space.

Now that we have seen how the rotation of memory works, let us see what would happen if we apply it also to the example of Fig 4.3. Figure 4.5 illustrates such situation. On the left side of the Fig. 4.5 we have the worker 0 memory representation before copying the stacks to worker 1. On the right side of the figure, we have the memory representation after the copy, from both the point of view of worker 0 and worker 1. As we can see from the point of view of worker 1, the addresses in its stacks are now ranging from 10000 to 19999, which makes the pointer at address 18000, copied from worker 0 to refer now to a valid address (address 12000). Since each worker only uses its own stacks space, there is no problem that, for example, from the point of view of worker 1, the pointer at address 28000, belonging to the stacks of worker 0 points to address 12000 in the stacks of worker 1.

4.2.2 Choice Points and Or-frames

Choice points have a critical role in the Prolog backtracking mechanism by allowing the computation to be restored to a previous saved state. After that the next unexplored alternative stored in the choice point, is picked in order to explore a new branch in the search tree. With or-parallelism, we can have more than one worker owning the

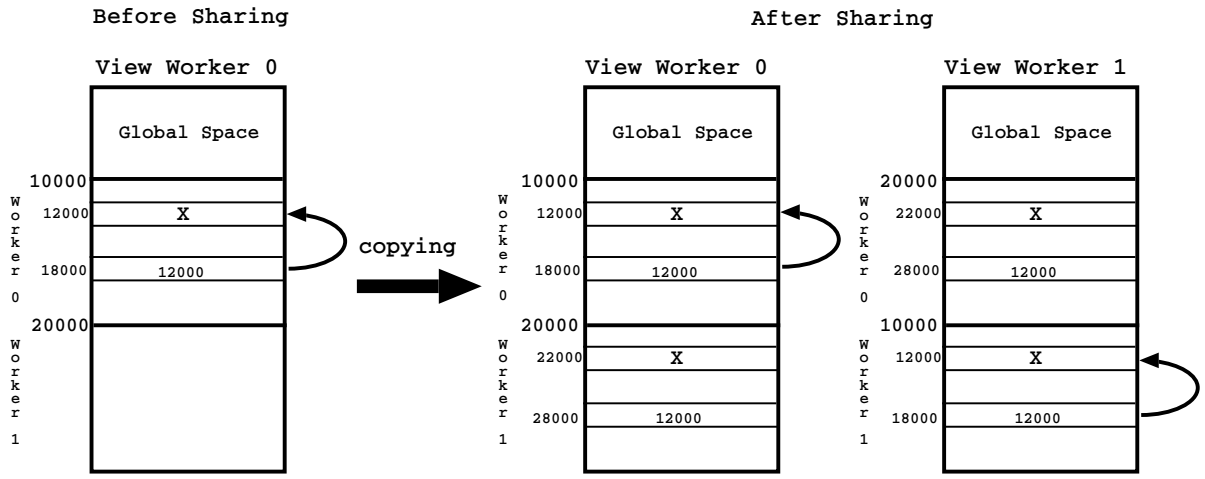


Figure 4.5: Using memory rotation to solve the problem found in Fig 4.3

same choice point and thus we need some sort of synchronization to avoid situations where more than one worker starts exploring the same alternative. In order to do that YapOr uses a shared structure, called or-frame to synchronize the access to the open alternatives in a shared choice point. Figure 4.6 shows how a private choice point is transformed into a public choice point and associated with an or-frame data structure.

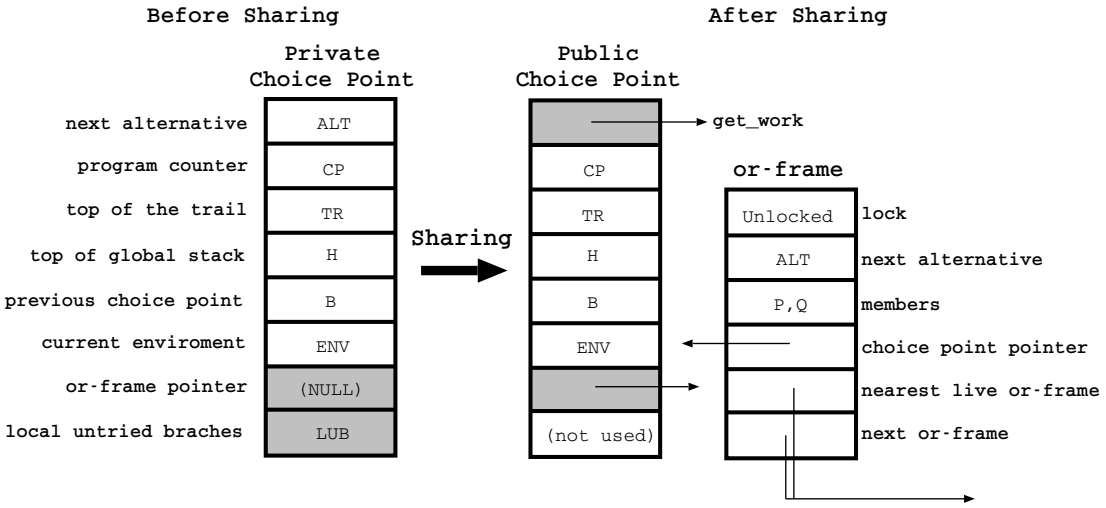


Figure 4.6: Sharing a private choice point

On the left side of Fig. 4.6 we can see a representation of a (private) Yap choice point. In order to support or-parallelism two new fields were added, which are colored in grey in the figure. The first one is a pointer to the corresponding or-frame, which is made to point to NULL when the choice is not shared. The second one is the local untried branches which corresponds to the number of private alternatives in the

current branch.

On the right side of the figure we can see the choice point after being shared and associated with an or-frame. The next alternative field in the shared choice point is made to point to a special *get_work* instruction, the or-frame field is made to point to the or-frame associated to it and the local untried branches field is no longer needed since the above alternatives are now public.

The right side of Fig. 4.6 also shows that an or-frame structure is composed by six fields. The *lock* field, as the name suggests, implements a lock that enable workers to synchronize the access to the or-frame. The *next alternative* field was inherited from the choice point and stores the pointer to the next untried alternative. The *members* field stores information about which workers are currently sharing that choice point. The *choice point pointer* field is a back pointer to the associated choice point. The *nearest live or-frame* field is a pointer to the next or-frame corresponding to a choice point with open alternatives. Finally, the *next or-frame* field points to the parent or-frame in the current branch.

4.2.3 Worker Load

As we mentioned before, the YapOr scheduler selects busy workers according to the position of the workers in the or-tree and their workload. Having the load of each worker updated is thus very important but, updating it often, may have a great impact in the performance so it is necessary to define a compromise between both. In YapOr, the workload of a worker is updated when a new choice point is created.

We can define the load of a given worker as the sum of all the open alternatives in its private choice points. Figure 4.7 illustrates how the process of calculating the load of a worker is done. The figure represents an execution tree divided in shared and private regions. With the private choice points showing the number of local untried branches (the CP(LUB) field). The first private choice point has CP(LUB) = 0 since the above shared choice points are not taken into account. The second private choice point has CP(LUB) = 1, which is number of alternatives in the first private choice point. The last choice point has CP(LUB) = 3, which is the sum of the previous CP(LUB) plus the two open alternatives in the previous choice point. To calculate the total workload of a worker we simply need to sum the CP(LUB) of its top choice point with the number of alternatives present in that choice point. In this case, the workload of the worker represented in the figure is thus 5.

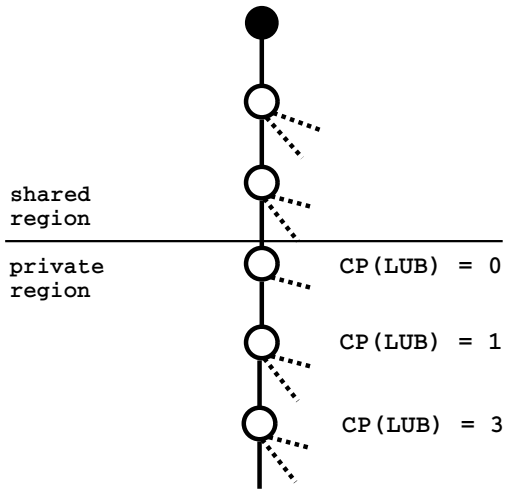


Figure 4.7: Local untried branches

4.2.4 Sharing Work Process

When a worker Q runs out of work, the scheduler is responsible for finding a busy worker P. Once it finds it, the idle worker Q should send a sharing request to P. The busy worker P may then accept and start the sharing work process or decline the solicitation if, for example, it has too few work that it is not worth sharing. This is done in order to avoid situations where worker P would spend more time preparing and sending the work than it would spend executing the work itself. Figure 4.8 shows the most important steps and communication signals involved in a sharing work operation.

After accepting a share request, worker P starts by computing the areas to copy to Q and since YapOr implements incremental copy only the non-common parts of the stacks will be copied. After this initial step, worker P sends a *sharing* signal to worker Q. Next, worker Q can start copying the trail and the heap stacks whilst worker P starts publishing its private nodes. Worker Q may only start copying the local stack when it receives a signal from P saying that its private nodes are made public. This happens because publishing nodes involves changing the choice points which are located in the local stack. After publishing the nodes, if needed, worker P may cooperate with worker Q in copying the stacks. Then, the workers synchronize by sending a *copy done* signal to each other. After that, the incremental copy mechanism requires that worker Q installs the bindings for the variables in the non-copied segments, while worker P can return to Prolog execution although it can not backtrack to the shared choice points in order to avoid undoing bindings that are being copied by Q. In such case, P must wait to receive a *ready* signal from Q confirming that all bindings are copied.

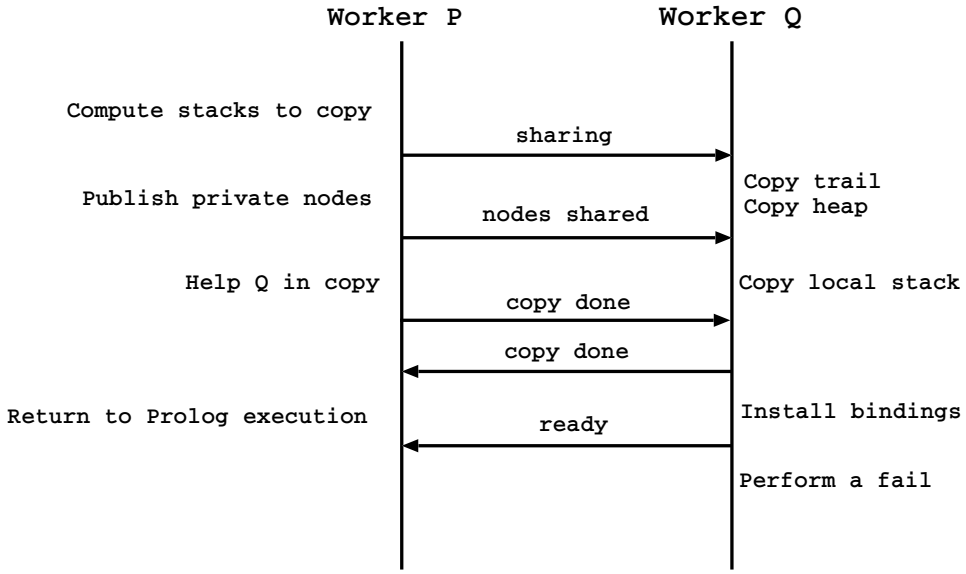


Figure 4.8: Steps and communication protocol between the two workers involved in a sharing work operation

At the end, worker Q simulates a failure in order to pick an open alternative from the youngest shared choice point.

4.2.5 New Pseudo-Instructions

YapOr introduces four new pseudo-instructions, three of them are related with the parallel execution and one is related with the execution of the cut predicate.

Before the first execution or at the end of the execution of a previous parallel goal all workers, except worker 0, execute the *getwork_first_time* instruction. The main purpose of this instruction is to block the workers until worker 0 informs that a new parallel goal is available.

As we have seen, when a choice is published the next alternative field is made to point to a *get_work* instruction. This instruction is then executed when a worker backtracks to a shared choice point and it allow workers to synchronize the access to the corresponding or-frame in order to pick the next open alternative, therefore guaranteeing that only one worker gets access to a shared alternative.

A variant of the getwork instruction, is called *getwork_sequential*, used when a predicate is declared as *sequential*. This instruction guarantees that the alternatives in a sequential public choice point are executed as if they were being interpreted by

a sequential engine, i.e., an alternative is picked for execution only when there is no available work in any younger choice point, which means that, the previous alternative is fully explored.

A last instruction, called *synch* instruction, used to implement the cut predicate in YapOr. The *synch* instruction implements a delay that delegates the execution of any other operation shared area until the alternative at hand (the one executing the cut) becomes the leftmost alternative in the or-tree. This is done in order to ensure that the execution of the cut follows the same order as if it were being performed in a sequential engine.

Chapter 5

Teams of Workers

This chapter discusses in detail our layered model, which aims to run Prolog in clusters of multicore processors, and describes the changes done to Yap/YapOr in order to efficiently support it.

5.1 Execution Model

At the beginning of the execution only one standard Yap engine, called the *client worker*, is running. Before executing any parallel goal it is necessary to launch, at least, one parallel engine. Predicate *par_create_parallel_engine/2* allows to create and launch a new parallel engine. A parallel engine is composed by teams of workers with each team behaving as an independent YapOr engine.

The worker 0 of each team is considered to be the *master worker* of the team and it is responsible for controlling the execution inside the team and for the communication with the other teams. Moreover, the first team to be launched is considered to be the *master team* and its *master worker* is responsible for receiving and launching the execution of the parallel goals sent by the *client worker*.

The predicate *par_run_goal/3* allows to define goals to be run in parallel. When this predicate is called a message with the goal to be run is sent from the client worker to the master worker of the master team. This worker will then start the execution of the received goal and notify all the master workers of the other teams belonging to the same parallel engine about that. Inside the master team, the execution is similar to the one seen for YapOr, with the master worker starting to share work with its

teammates. Outside the master team, the other teams are now aware that a parallel computation has begun and thus they enter in scheduling mode. Soon after, they will start contacting the master team in order to get work.

When a team A receives a sharing work request from a team B, the *team sharing work process* begins. In the team sharing work process, first a worker W from inside team A will be designated to answer the request. Then, worker W may reject or accept the request based on its current conditions. If the request is accepted, W proceeds in the following way:

- W starts by copying its stacks to an auxiliary area assigned to it;
- a stack splitting strategy is applied to its stacks and the stacks in the auxiliary area;
- the stacks in the auxiliary area are sent to the master worker of the requesting team B.

When the master worker of team B receives the stacks from team A, they are installed on its own work space. At this point, the master worker of team B must inform the remaining teammates that the team has now work. After that, the execution inside team B evolves as a standard YapOr execution with the master worker performing a fail, in order to take the next open alternative, and with its idle teammates starting to ask it for work.

A team is considered to be out of work when every worker inside the team is idle. When that happens, the team enters in scheduling mode in order to choose a busy team to request work and the same sharing process, as described above, is repeated. The execution ends when all teams are idle and, in the continuation, the client worker is notified that the parallel execution is finished.

Our model can be seen as a layer implemented on top of the YapOr engine in order to combine distributed memory with the already existing shared memory approach. In our model, the communication between teams is done using MPI messages. The MPI implementation chosen was Open MPI [1], although our model should be compatible with any other recent MPI standard implementation. In the next sections, we present the main implementation details of our model.

5.2 Starting a Parallel Execution

As we have seen in section 4.1, launching a parallel goal requires two steps. The first step involves the creation of, at least, one parallel engine using the predicate *par_create_parallel_engine/2*, and then the predicate *par_run_goall/3* can be used for sending goals to be run in the available parallel engines. Next, we describe the key implementation details behind these two steps.

5.2.1 Creating a Parallel Engine

When the predicate *par_create_parallel_engine/2* is called in the client worker, it initiates the process of creating a parallel engine. The Prolog code for this predicate can be seen in Fig. 5.1.

```
par_create_parallel_engine(EngineName,TeamList) :-
    '$c_parallel_engine_create'(TeamList,EngineID),
    assertz(parallel_engine(EngineName,EngineID)).
```

Figure 5.1: Prolog code for the predicate *par_create_parallel_engine/2*

The predicate receives two arguments: the name to be given to the parallel engine, represented by the variable *EngineName*, and a list of tuples defining the set of teams to be created as part of the parallel engine, represented by the variable *TeamList*. Each tuple $< h, n, p >$ in *TeamList* includes a host h, the number of n workers to be spawned on that team and the path p to the file program to be loaded by default.

When the predicate is called, the subgoal *'$c_parallel_engine_create'/2* is the first to be executed. The *'$c_parallel_engine_create'/2* predicate is written using the C language interface of Yap[1] and is responsible for doing the most critical part of the job – spawning the master workers of each team. In order to do that, it uses the *MPI_Comm_spawn_multiple()* function of the MPI API. Then a broadcast is performed informing those master workers about the number of workers that will be present in each team. During this process, a new *EngineID* (which is an integer) is assigned to that new parallel engine, allowing to internally identify it. This *EngineID* corresponds to the position of the *engine_frame* data structure where information about that parallel engine is stored. The *EngineID* is also used to perform an assert in order to associate it with the name given to the new parallel engine.

[1] In what follows, all predicates that start with '$c_' are defined using Yap's C interface.

In the continuation, each master worker must then allocate the shared memory which will support the parallel execution of its team and launch the remaining workers of the team using the *fork()* system call, in a process similar to the one in YapOr. Each worker will then remap its memory according to the process described in the next section and, at last, jump to the *getwork_first_time* instruction. At that point, the parallel engine is ready to run parallel goals.

A schematic representation summarizing the process of spawning processes can be seen in Fig. 5.2. For this example, we consider again the topology presented in Chapter 3 that we have been using, which can be created with the call:

par_create_parallel_engine(E,[(N1,4,'prolog\prog.pl'), (N1,4,'prolog\prog.pl'), (N2,8,'prolog\prog.pl')]).

In the figure, we can see that the master workers (workers 0) of each team are first spawned by the client worker using the *MPI_Comm_spawn_multiple()* function and only after the other workers are launched using the *fork()* system call.

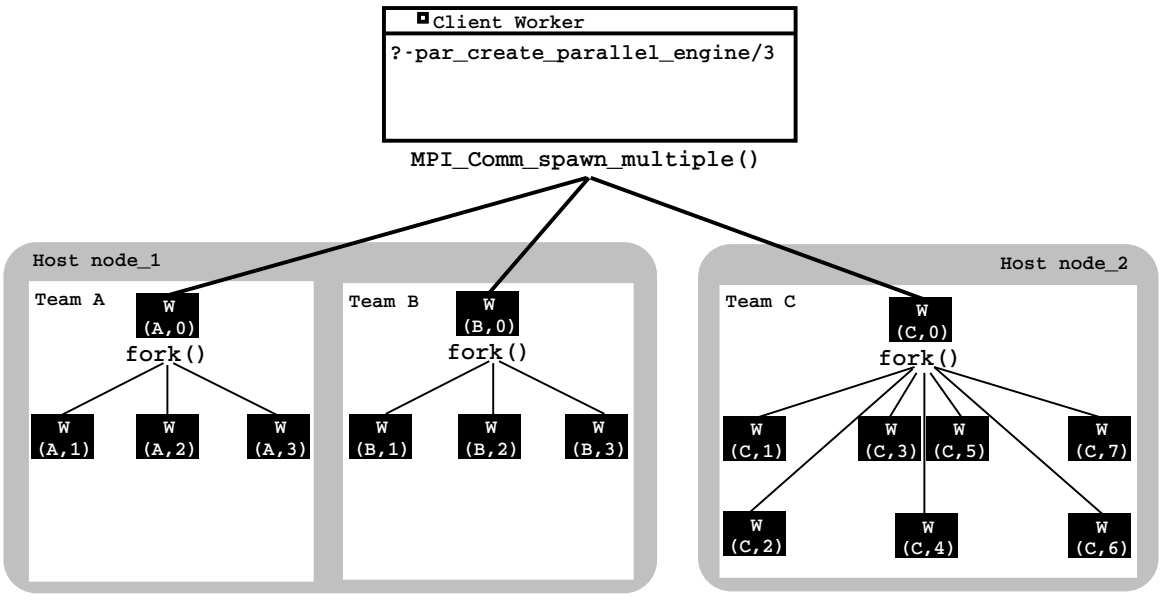


Figure 5.2: Schematic representation of the process of spawning workers that constitute a parallel engine

5.2.2 Running a Parallel Goal

After the creation of a parallel engine, the user can now run parallel goals and for that it must use the predicate *par_run_goal/3*. The predicate receives as arguments the name of the parallel engine where to run the goal, the goal to be run and a template indicating how the answers to the given goal should be returned. Consider that we want to run, in parallel the goal *par_computation(X,Y,Z)* and we are only interested in the answers obtained for variable *Z*. In such case, the template could be defined as *Z*. The Prolog code for this predicate can be seen in Fig. 5.3.

```
par_run_goal(EngineName,Goal,Template) :-
    engine_name_id(EngineName,EngineID),
    '$c_run_parallel_goal'(EngineID,Goal,Template).
```

Figure 5.3: Prolog code for the predicate *par_run_goall/3*

The predicate *par_run_goal/3* begins by translating the name of the team to the corresponding *EngineID* so that it can access the information stored in the *engine_frame* data structure. After that, the predicate *'$c_run_parallel_goal'/3* is executed with the *EngineID* as argument. This predicate is responsible for sending the goal and the template to the parallel engine in order to be run. This is done by first converting the goal and the template terms to strings (using the *YAP_WriteBuffer()* function that allows the terms stored in the heap to be converted into strings), and then by sending a message containing such strings to the master worker of the master team.

In Fig. 5.4 we have a fragment of code, extracted from the *getwork_first_time* instruction, that shows what happens when a worker jumps to that instruction waiting for the execution of a new parallel goal. As we can see, each master worker first waits that all other workers inside its team became ready to initiate the computation, and, after that, all master workers guarantee that all teams are in the same condition (all master workers wait for each other). It is important to note that the first wait function is implemented in shared memory by marking the ready workers in a bitmap while the second waiting function is implemented using an *MPI_Barrier()*. After this initial synchronization, the master worker of the master team waits for a new parallel goal to be sent by the client worker. When that happens, it begins by converting the goal and the template strings into terms with the help of the *YAP_ReadBuffer()* function, and then the parallel computation starts by running the predicate *parallel_run/2* with the goal and the template as arguments.

```
if (is_master_worker(worker)) {
    wait_team_mates()
    wait_for_master_workers()
    if (is_master_team(team)) {
        msg = Recv(client_worker)
        goal = get_goal(msg)
        template = get_template(msg)
        goalTerm = YAP_ReadBuffer(goal)
        tamplateTerm = YAP_ReadBuffer(template)
        RunPred(parallel_run(goalTerm,templateTerm))
    } else { //not the master team
        wait_for_work_in_parallel_engine()
        make_root_choice_point()
        TS_team_idle_scheduler()
    }
} else { //not a master worker
    set_as_ready(worker)
    wait_for_work_in_team()
    make_root_choice_point()
    LS_local_scheduler()
}
```

Figure 5.4: Fragment of pseudo-code from the *getwork_first_time* instruction

The code for the *parallel_run/2* predicate can be seen in Fig. 5.5. It begins by calling the predicate *'$c_yapor_start'/0* which is responsible for initializing auxiliary execution variables, setting up the first shared choice point and, more important, signalizing the other master workers that the execution has begun. After that the predicate *'$execute'/1* starts the computation of the parallel goal. Each time the parallel goal yields an answer the predicate *'$c_parallel_new_answer'/1* is responsible for storing it.

```
parallel_run(Goal,Template) :-
    '$c_yapor_start',
    '$execute'(Goal),
    '$c_parallel_new_answer'(Template),
    fail.
parallel_run(Goal,Template).
```

Figure 5.5: Prolog code for the predicate *parallel_run/2*

Returning to the code of the *getwork_first_time* instruction, the remaining master workers wait for the message to be sent by the master worker of the master team.

That signal is done in a form of a message that contains information, such as the position of memory of the first choice point, that will allow the workers to create the first shared choice point using the *make_root_choice_point()* procedure. Then those master workers must call the function *TS_team_idle_scheduler()* which is responsible for finding a busy team to request work to.

On the other hand, all (non-master) workers wait for a notification from their master workers saying that their team has work. After receiving such signal, they allocate also their first shared choice point using the *make_root_choice_point()* procedure, this time the first choice point can be consulted directly, through shared memory, in the workspace of the master worker of their team. Then the workers call the local scheduler function *LS_local_scheduler()* as a YapOr worker would do in order to search for work inside its team. Every time a team runs out of work, the non-master workers return to this instruction waiting again for its master worker to get work from another team. From their perspective, every time the master worker gets work from the outside, is like beginning a new parallel computation.

5.3 Memory Organization

A team in our model can be viewed, from the outside, as a standard YapOr engine. For that reason, the management of memory and the memory layout is quite similar in both systems.

Regarding the memory layout, the main difference is due to the new process of sharing work between teams. Since this process requires auxiliary memory areas, they were included in the new memory layout for a team. The size of each one of these areas is the same of a worker area and the number of areas can be statically defined when compiling the system. A schematic view of this new layout can be seen in Fig. 5.6, with the new memory areas colored in grey.

The process of mapping memory inside the team is also very similar to YapOr. When a new master worker is launched, it is responsible for allocating the memory that will support the parallel execution of its team, which is done by calling the *mmap()* system call. After that, using the *fork()* system call, it launches the remaining workers of the team. Then, it is necessary to remap the memory, but now this process is done in a slightly different way from YapOr's original strategy.

In the remapping process of YapOr, each worker rotates the memory to guarantee that,

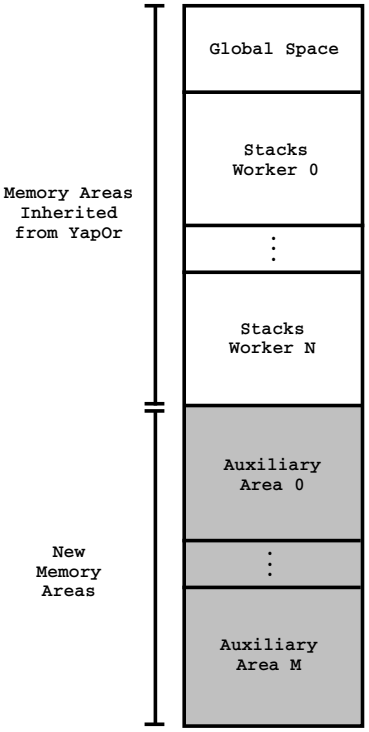


Figure 5.6: Team memory layout

from its point of view, all workers see its own stacks area starting at the same address space [35]. Now instead of rotating the memory, we reserve an address space that will be used by all workers to remap their own areas without the need of remapping the areas assigned to the other workers. In order to understand this process, let us consider the example in Fig. 5.7. On the left side, it presents the initial mapping of the memory space done by the master worker and, on the right side, the view, from the perspective of each worker, and after applying the remapping process. In the example, the reserved address space ranges from address 10000 to address 19999. Comparing the diferents views of each worker with the initial mapping, we can observe that each worker only remapped its area using the reserved address space thus maintaining the remaining addresses intact.

5.4 Team Scheduler

In order to be able to efficiently distribute and balance the work among teams we have developed a team scheduler. This new scheduler, can be seen as an extra layer on top of the local scheduler (the scheduler used by YapOr) which in our model continues

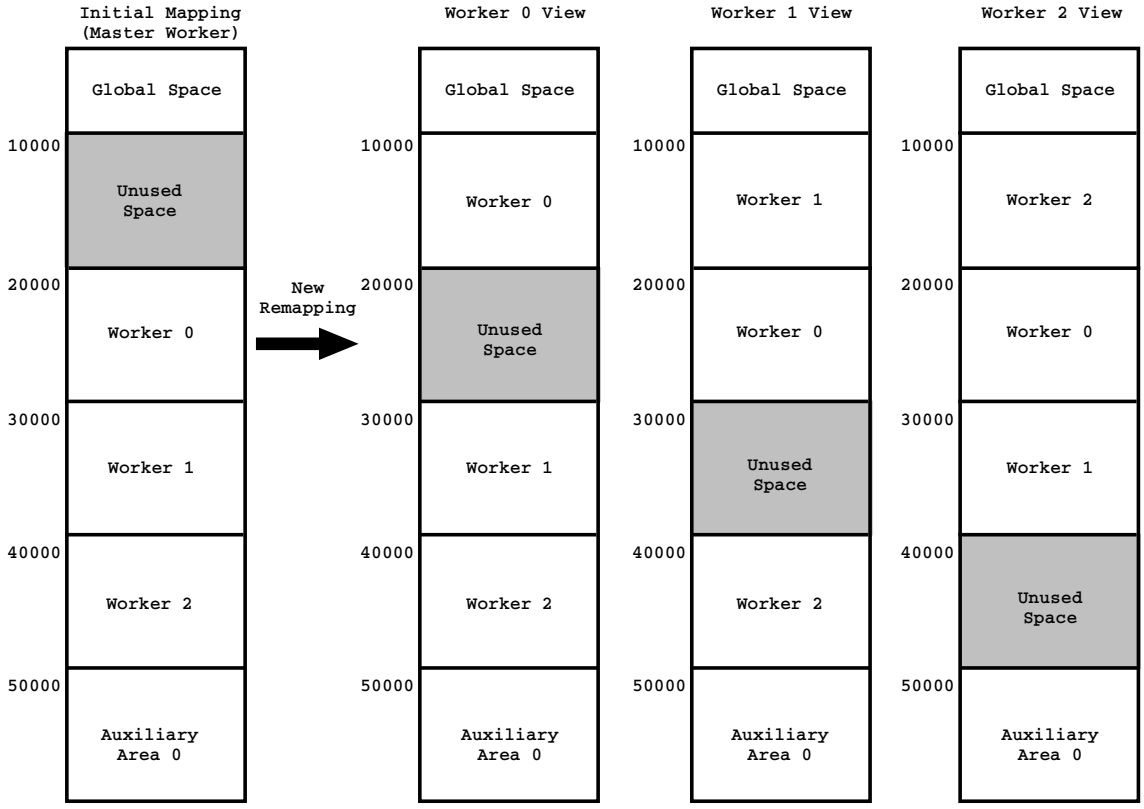


Figure 5.7: Remapping process inside a team

to be responsible for the distribution of work inside the teams. The team scheduler implements several functionalities, such as: (i) the handling of the communications between teams; (ii) the sharing work process, which allows a team to share work with another team; and (iii) the termination process, which determines the end of the execution of a parallel goal by ensuring that all search space was fully explored.

The team scheduler can be divided in two different modules. The first module, called *team idle scheduler*, runs when a team is idle. A team is considered to be idle when all of its workers run out of work. Once it happens, the master worker of that team enters in scheduling mode and starts running the team idle scheduler which is responsible for finding a busy team willing to share work. The second module, called *team busy scheduler*, is run from time to time by all workers inside a busy team and it is responsible for answering to the sharing requests sent by the idle teams.

In order to better understand how the team scheduler works let us consider the example in Fig. 5.8, where we have two teams represented. On the left side of the figure, we have an idle team A formed by four workers and, on the right side, we have a busy team B also formed by four workers. The boxes inside both teams show the procedures

called during the execution of the two modules of the team scheduler. The arrows represent information exchanged during scheduler execution – if they are exchange between workers of the same team we call them notifications (represented as dotted arrows), otherwise, if they correspond to information exchange between workers from different teams, we call them messages (represented as solid arrows).

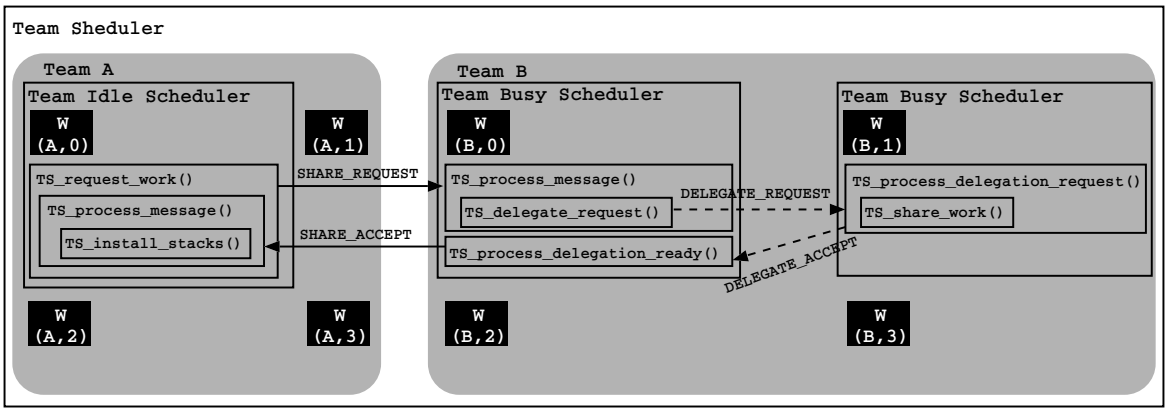


Figure 5.8: Team scheduler and its major components

Since team A is idle, its master worker is running the team idle scheduler. The procedure *TS_request_work()* is responsible for finding a busy team to request work to. In the example, the procedure's scheduling algorithm decided to send a SHARE_REQUEST to the busy team B. When the master worker in team B notices that there is a pending message coming from other team it uses the procedure *TS_process_message()* to process that message. The SHARE_REQUEST from team A will then cause the procedure *TS_delegate_request()* to be called in order to find the worker inside the team with the best conditions to successfully answer the request. In this example, the selected worker was worker 1 (or W(B,1)) and therefore a DELEGATE_REQUEST notification is sent to it. Worker W(B,1) also runs the team busy scheduler from time to time looking for delegation requests and once it receives one, the procedure *TS_process_delegation_request()* is called. If the request is accepted, it then calls the procedure *TS_share_work()* that is responsible for performing the stack splitting, preparing the stacks to be shared and then send a DELEGATE_ACCEPT notification to the master worker. Otherwise, if the request is refused, it sends a DELEGATE_REFUSE notification to the master worker informing that the request was rejected. When the master worker receives a DELEGATE_ACCEPT notification, it sends a SHARE_ACCEPT message together with the stacks to the requesting team. The procedure *TS_process_message()* in team A is responsible for receiving the SHARE_ACCEPT message and by calling the procedure *TS_install_stacks()* that will

install the stacks in the master worker. At that point, team A is no longer considered to be idle and the master worker can now share work with its teammates. On the other hand, if the request had been denied with a SHARE_REFUSE message, team A would try to initiate the termination process by calling the procedure *TS_try_termination()*. The code for the *team idle scheduler* can be seen in Fig. 5.9 while the code for the *team busy scheduler* can be seen in Fig. 5.10.

```
TS_team_idle_scheduler() {
    while (TRUE) {
        if (TS_request_work())
            fail() // backtracks to the top choice point to get work
        else
            TS_try_termination()
    }
}
```

Figure 5.9: Pseudo-code for the team idle scheduler

```
TS_team_busy_scheduler() {
    if (is_master_worker(worker)) {
        if (probe_message())
            TS_process_message()
        if (probe_notification())
            TS_process_delegation_ready()
    } else {
        if (probe_notification())
            TS_process_delegation_request()
    }
}
```

Figure 5.10: Pseudo-code for the team busy scheduler

In the next subsections, we discuss in more detail the two team scheduler's modules introduced here.

5.5 Communication

The communication between teams is done using messages. Those messages are sent and received only by the master worker of each team, using the MPI functions *MPI_Send()* and *MPI_Recv()*, respectively.

Team messages are divided in two groups: *sharing messages* and *termination messages*. In Table 5.1, we can see the messages along with content sent in each one. The sharing messages, as the name suggests, are used to implement the team sharing process. When a team is out of work, the SHARE_REQUEST message is used to request work to a busy team. A busy team might reply with a SHARE_ACCEPT message, to accept the request, or may decline it by sending a SHARE_REFUSE message. The extra content included in these messages is the current state of the load array of the sending team and, in the specific case of the SHARE_ACCEPT message, the stacks to be installed in the requesting team. The load array is only used for scheduling purposes and will be detailed in another section. The TERMINATION messages are related with the termination process and they indicate when the current parallel goal is fully exploited meaning that a team must end the current computation. The termination process will be discussed in detail in a later section.

Table 5.1: Messages used for communication between teams

Group of Message	Type of Message	Extra Content
Sharing	SHARE_REQUEST	[LOAD_ARRAY]
	SHARE_ACCEPT	[LOAD_ARRAY] [STACKS]
	SHARE_REFUSE	[LOAD_ARRAY]
Termination	TERMINATION	

Checking for messages regularly is vital to maintain the system updated but may also be very costly. When a given team is busy, the master worker must run the team busy scheduler. However, the master worker may be itself in two states: also busy or idle. If the master worker is busy, the team busy scheduler is run one time per each RUN_TEAM_BUSY_SCHEDULER_THRESHOLD times it performs the WAM *call* instruction. Otherwise, if the master worker is idle, the outside team messages are checked inside the main loop of the local scheduler. Another possibility is the team being idle, which implies that the master worker is also idle and running the team idle scheduler and thus checking for outside team messages. Figure 5.11 shows the pseudocode for the *TS_process_message()* procedure that is responsible for receiving and processing messages in both team busy scheduler and team idle scheduler. Please note that in the team busy scheduler only SHARE_REQUEST messages are expected to be received.

The code begins by receiving the next pending message and the variables *type_of_message*, *team* and *load_array* are initialized with the correspondent content in the message received. After that, the loads of the teams are updated with the new information re-

```
TS_process_message() {

    msg = Recv(any_team)
    type_of_message = get_type_of_message(msg)
    team = get_team(msg)
    load_array = get_load_array(msg)
    if (load_array)
        update_load_array(load_array)
    switch (type_of_message) {
        case SHARE_REQUEST:
             if (is_team_busy())
                 TS_delegate_request(team)
             else // team is idle
                 Send(team, SHARE_REFUSE)
        case SHARE_ACCEPT:
             stacks = get_stacks(msg)
             TS_install_stacks(stacks)
             notify_teammates()
        case SHARE_REFUSE:
             // in this case the calling function will be responsible for dealing with it
        case TERMINATION:
             goto getwork_first_time
    }

    return type_of_message

}
```

Figure 5.11: Pseudo-code for the *TS_process_message()* procedure

ceived in the load array followed by the actions to be taken for each specific type of message received (switch case statement in Fig. 5.11). When a SHARE_REQUEST is received, we first check if the team is busy and, in such case, the function *TS_delegate_request()* is called. This function is responsible for finding the worker inside the team with the best conditions to answer the request. Otherwise, if the team is idle, a SHARE_REFUSE message is sent immediately to the requesting team.

The *TS_process_message()* function also treats the replies to previously sent SHARE_REQUEST messages. If the reply to a sharing request is SHARE_ACCEPT, we begin by calling the *TS_install_stacks()* procedure which is responsible for installing the stacks sent by the busy team. After that, the master worker informs their teammates that their team has now work and starts computing that work. On the other hand, if the answer to a

share request is SHARE_REFUSE the calling function in the *team idle scheduler* will deal with it after the return of this procedure. A TERMINATION message is only sent when the termination process detects that all teams are idle. For that reason, when this message is received by a master worker, it simple jumps to the *getwork_first_time* instruction which is responsible for synchronizing the workers and prepare them to a new execution. At the end, the procedure returns the type of message received.

5.6 Load Balancing

When a team is idle, its master worker enters in scheduling mode where the function *TS_request_work()* is used to select the team to which a SHARE_REQUEST message should be sent. When the request is received by the master worker of the busy team, it calls the function *TS_delegate_request()* which is responsible for choosing the sharing worker from the team. Even though the sharing request involves two teams, in practice, the sharing process is done between two workers. The master worker of the requesting team and the chosen sharing worker from the busy team. Next, we will see in more detail the two functions involved in the selection of the busy team and in the selection of the sharing worker.

5.6.1 Selecting a Busy Team

The *TS_request_work()* function is responsible for selecting a team with available work to share. The pseudo-code for the *TS_request_work()* function can be seen in Fig. 5.12. The function begins by selecting the busy team with the highest load and, if a busy team exists, a SHARE_REQUEST is sent to it. After that, it waits for an answer from the busy team (function *TS_process_message()*) returning true or false if it receives a SHARE_ACCEPT or a SHARE_REFUSE, respectively. While the *TS_request_work()* is waiting for a response to the share request it may continue receiving and answering messages from the outside. This constant updating of information might be useful, if the current sharing request is rejected.

The team's load information is stored in a bi-dimensional array, the load array, which contains for each team its load and a timestamp. We define the load of a team as the sum of the particular loads of each worker in the team. The load of each worker is still computed as in YapOr (remember from Chapter 4 that the load of a worker is given by the sum of all open alternatives in its private choice points).

```
TS_request_work() {

    max_load = -1;

    for (i = 0; i < number_teams; i++) {
        if(get_load(load_array,i) > max_load){
            max_load = get_load(load_array,i)
            selected_team = i
        }
    }

    if (max_load > -1) {
        Send(selected_team,SHARE_REQUEST)
        while (TRUE) {
            if (probe_message()) {
                answer = TS_process_message()
                if (answer == SHARE_ACCEPT)
                    return TRUE
                else if (answer == SHARE_REFUSE)
                    return FALSE
            }
        }
    } else // no busy team found
        return FALSE

}
```

Figure 5.12: Pseudo-code for the *TS_request_work()* procedure

The information in the load array is updated whenever a new team message is received. When a worker sends a new message, it includes a copy of the load and timestamp information in its load array, so that the receiving worker can update its array with such information. This is done by comparing the timestamps in its load array with the timestamps on the received load array and when a received timestamp is younger, then the load array must be updated. Timestamps are implemented using an integer which is incremented every time the corresponding team sends a new message. Thus, whenever we refer that a timestamp A is younger than a timestamp B that means that the timestamp A has a higher value than the timestamp B. It would be possible to update the load array more often by sending specific LOAD_INFO messages whenever certain operations are done, for example, after a successful sharing process the requesting team could send a broadcast message informing all the others teams about

```
TS_delegate_request(team) {

    sharing_worker = master_worker
    max_load = load(sharing_worker)

    for (i = 1; i < number_workers; i++) {
        if (!has_delegations(i) && worker_load(i) > max_load) {
            max_load = worker_load(i)
            sharing_worker = i
        }
    }

    if (max_load < THRESHOLD_FOR_SHARING || !available_sharing_areas()){
        Send(team,SHARE_REFUSE)
        return
    }

    frame = find_auxiliary_sharing_area()
    if (sharing_worker == master_worker) {
        new_load_requesting_team = TS_share_work(frame)
        update_load_array(team,new_load_requesting_team)
        Send(team,SHARE_ACCEPT)
    } else{ //sharing_worker != master_worker
        initialize_delegation_frame(frame,sharing_worker,team)
        set_as_has_delegations(sharing_worker)
        Send_notification(sharing_worker,DELEGATE_REQUEST,frame)
    }
    return
}
```

Figure 5.13: Pseudo-code for the *TS_delegate_request()* procedure

its new load. However, this will increase the number of messages circulating which could have a negative performance impact.

5.6.2 Selecting a Sharing Worker

When a team is busy and receives a SHARE_REQUEST, the *TS_delegate_request()* function is used to select the worker with the highest load inside the team and to delegate to such worker the sharing work task. This scheduler procedure can be seen in Fig. 5.13.

The *TS_delegate_request()* procedure receives as argument the requesting team. Initially, it starts by selecting the worker with the highest load that currently is not dealing with any other delegation request. After that, two conditions must be checked. The first condition is if the load of the selected worker is lower than a defined threshold. This is done in order to avoid sharing small tasks that may take more time sharing them than eventually computing them. The second condition is related with the availability of an auxiliary sharing memory area, as introduced in the memory organization section, which is required for the sharing process. If one of these two conditions fails, the procedure sends a SHARE_REFUSE message to the requesting team in order to reject the share request and then returns.

Otherwise, we must first find a free auxiliary sharing area by calling the function *find_auxiliary_sharing_area()* which will return the frame associated to that area. If the selected sharing worker is the master worker, we begin by calling the function *TS_share_work()* in order to do the stack splitting and preparing the stacks to be sent in the auxiliary sharing area. After that, we update the load of the requesting team with the load returned by the function *TS_share_work()*, which corresponds to the work that will be sent. Then a SHARE_ACCEPT message with the stacks is sent to the requesting team. The master worker can then leave the team busy scheduler and return to its work since the following steps in the sharing process are responsibility of the requesting team.

On the other hand, if the selected worker is not the master worker, the stacks can not be sent right away to the requesting team. First, we need to initialize the delegation frame associated with the selected auxiliary sharing area which will further allow the communication between the sharing worker and the master worker. After that, the sharing worker is notified about this delegation by sending a DELEGATE_REQUEST together with the selected auxiliary sharing area. After this the master worker can leave the team busy scheduler and return to its work. In the next section, we explain in more detail this process and which are the next steps taken in a delegation process.

5.7 Sharing Process

As we have seen, we can distinguish two types of sharing: the sharing process done by the master worker and the delegated sharing process. In the case of the master worker, the stacks are sent immediately to the requesting team. In the delegated sharing process, a worker diferent from the master worker is choosen for sharing its

stacks and therefore more steps are needed in order to send the stacks to the requesting team. Next, we will discuss delegated sharing in more detail.

5.7.1 Delegated Sharing Process

Remember that, when a worker receives a delegated sharing request its stacks may not be sent directly to the requesting team. This happens because Open MPI only allows MPI processes to send messages and, in our model, the non-master workers are non-MPI processes since they are launched using the *fork()* system call rather than using a specific MPI procedure. When we are dealing with a delegated sharing, the function *TS_delegate_request()* assigns an auxiliary sharing area to that particular sharing operation. Each one of the auxiliary sharing areas has a delegation frame associated to it that must be initialized in order to later allow the communication between the master worker and the sharing worker.

The delegation frame structure includes the following nine fields (Figure 5.14, shows a schematic representation of this structure): *DgFr_is_free* which indicates if the auxiliary area associated to that particular frame is being used or not; *DgFr_ptr_area* is a pointer to the first position of the auxiliary sharing area associated to that frame; *DgFr_sharing_worker* represents the id of the sharing worker; *DgFr_requesting_team* which is the idle team who is requesting for work; *DgFr_notification_to_master_worker* is a field used by the sharing worker to communicate with the master work of its team; *DgFr_stacks_size* informs the master worker about the total amount of memory used in the auxiliary sharing area; *DgFr_new_load_requesting_team* informs the master worker about the new load of the requesting team after the stacks being splitted.

Consider again the example in Fig. 5.8 that a message requesting work is sent from team A to team B. When the scheduler on team B receives that message it starts the process of selecting a sharing worker, which in our example is worker 1, and then it must find a free auxiliary sharing area and initialize its delegation frame. On the right side of Fig. 5.14, we can see the result of this initialization for the situation described above. The field *DgFr_is_free* is updated to FALSE, meaning that the frame is currently in use. The fields *DgFr_sharing_worker* and *DgFr_requesting_team* are now matching this hypothetical sharing scenario with values 1 and A, respectively. Finally, the field related with the communication, *DgFr_notification_to_master_worker* has no messages yet. Next, with the frame initialized, it is time to notify worker 1 to start the team sharing process.

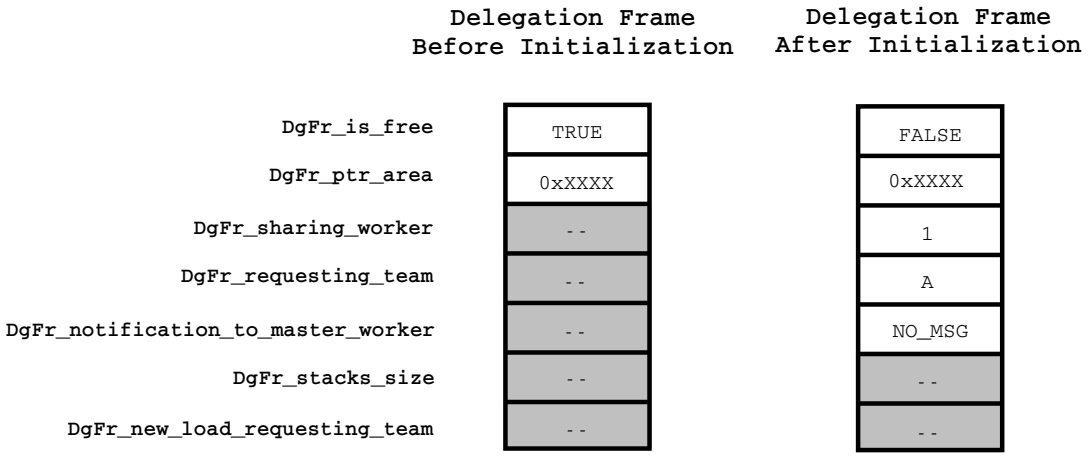


Figure 5.14: On the left side, we can see a delegation frame before being initialized and, on the right side, the same structure after the initialization to be used in a delegation request

The non-master workers always check if they have any sharing delegation request before executing any WAM *call* instruction. The pseudo-code for processing such a request is shown in Fig. 5.15. The worker starts by receiving the delegation request and then the variable *frame* is initialized. After that, the worker checks if its current load is lower than a defined threshold (THRESHOLD_FOR_SHARING) that allows workers to refuse sharing requests when its load is not enough. This condition was already checked by the master worker but since the communication is not immediate, the load might have changed. If the previous condition holds, the sharing worker denies the request by sending a DELEGATE_REFUSE notification to the master worker using the field *DgFr_notification_to_master_worker* of the delegation frame. Otherwise, the worker accepts the request and calls in the continuation the *TS_share_work()* procedure that will perform the stack splitting, store the resulting stacks in the assigned auxiliary sharing area and fill the fields *DgFr_stacks_size* and *DgFr_new_load_requesting_team* of the delegation frame. After that, a DELEGATE_ACCEPT notification is sent to the master worker informing it that the stacks can now be sent to the requesting team.

The master worker also checks for answers to its delegation requests when it executes the WAM *call* instruction. It does that by checking if there are any new notifications in the *DgFr_notification_to_master_worker* field in each position of the delegation frame array. When it finds one, it runs the function *TS_process_delegation_ready()* which uses the excerpt of code of Fig. 5.16 to process the request.

When the master worker receives a DELEGATE_REFUSE notification, it sends a

```
TS_process_delegation_request() {

    notification = Rcv_notification(master_worker)
    frame = get_delegation_frame(notification)

    if (worker_load(worker_id) < THRESHOLD_FOR_SHARING) {
        DgFr_notification_to_master_worker(frame) = DELEGATE_REFUSE;
    } else {
        TS_share_work(frame);
        DgFr_notification_to_master_worker(frame) = DELEGATE_ACCEPT;
    }

}
```

Figure 5.15: Pseudo-code for the *TS_process_delegation_request()* function responsible for processing a delegation request

SHARE_REFUSE message to the requesting team, removes the sharing worker from the list of workers with delegations and frees the delegation frame. If the notification received is a DELEGATE_ACCEPT, first it sends a SHARE_ACCEPT message to the requesting team together with the stacks in the auxiliary sharing area. Then, it updates its load array with the new load of the receiving team in order to be able to propagate the new load in its following messages. At this point, the sharing worker can be removed from the list of workers with delegations and the delegation frame is made free again.

5.7.2 Preparing the Stacks to be Sent

The function *TS_share_work()* is responsible for preparing the stacks to be sent to the requesting team. The first step taken by this function is to determine the segments of the different stacks to be copied. In order to better understand how this is done, observe the left side of Fig. 5.17, which presents the stacks in a worker area. The area of the heap to be copied is delimited by the pointer to the heap in the root choice point, which represents the first shared choice point, and the register H, which points to the top of that stack. For the local stack, the area to be copied is delimited by register B, that points to the last choice point, and by the root choice point. For the trail, the area to be copied is given by register TR that points to the top of the trail, and by the pointer to the trail in the root choice point.

```
team = DgFr_requesting_team(frame)
sharing_worker = DgFr_sharing_worker(frame)

if (DgFr_notification_to_master_worker(frame) == DELEGATE_REFUSE)
    Send(team,SHARE_REFUSE)
else if (DgFr_notification_to_master_worker(frame) == DELEGATE_ACCEPT) {
    Send(team,SHARE_ACCEPT)
    update_load_requesting_team(team,DgFr_new_load_requesting_team(frame))
}
unset_as_has_delegations(sharing_worker)
free_auxiliary_sharing_area(frame)
```

Figure 5.16: Excerpt of code from the function *TS_process_delegation_ready()* responsible for receiving and processing a delegation response

The values that delimit those areas are stored in the header region of the auxiliary sharing area together with the load of the new team that will be determined during the stack splitting operation. Following the header region are the heap, local stack and trail segments as determined before. Since the information in the auxiliary sharing area will be sent to the requesting team, we try to reduce the size of that message by copying those segments in such a way that there is no gaps between them. A schematic view of the steps discussed above is depicted on the right side of Fig. 5.17.

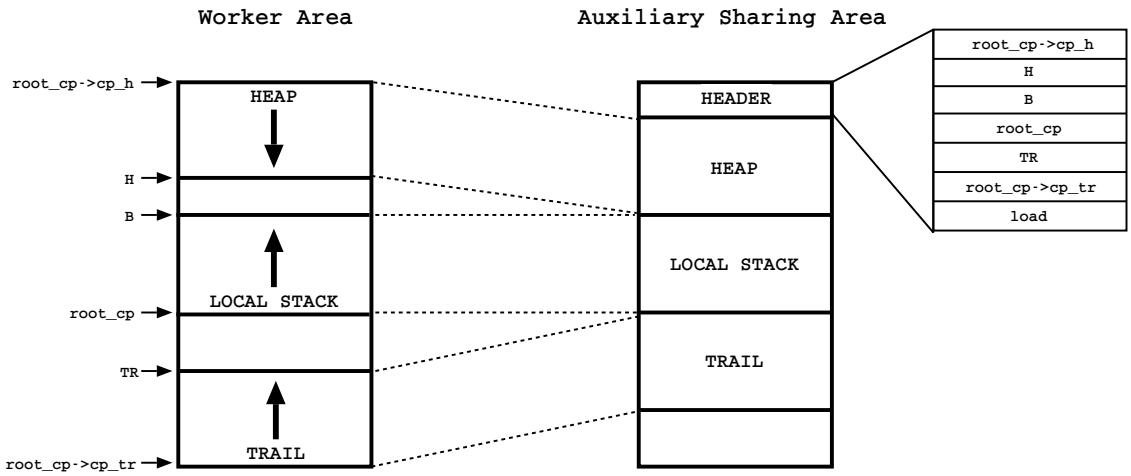


Figure 5.17: On the left side, we have the schematic representation of the segments of the stacks to be copied and, on the right side, we have the representation of the auxiliary sharing area

Now we have two copies of the stacks, one in the sharing worker stacks and another

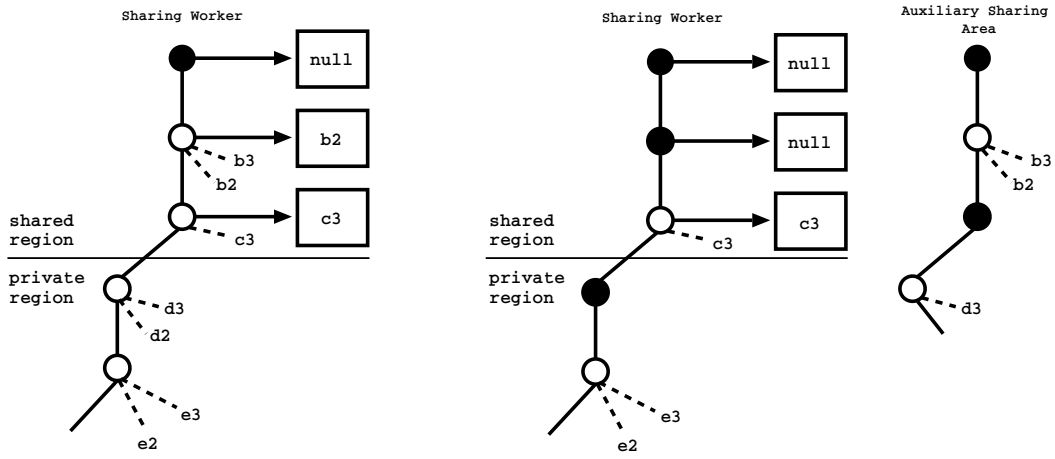
in the auxiliary sharing area. So we are now able to perform the stack splitting operation between both. Our system implements two different stack splitting strategies – vertical splitting and horizontal splitting – which are described in detail in the next two subsections. After the stack splitting operation completes, the LOAD of the sharing worker and the load in the header are both updated to reflect the changes done and the auxiliary sharing area is ready to be sent to the requesting team. Once the master worker of the requesting team receives it, it just needs to install the stacks in its own worker space with the help of the information present in the header.

5.7.3 Vertical Splitting

Before seeing how vertical splitting is implemented, let us consider the example in Fig. 5.18a where we can see the schematic representation of the execution tree of the sharing worker, which is the same as the stacks in the auxiliary sharing area since they were copied as explained before. The execution tree is divided in two regions, the shared and the private region. In the shared region, we have choice points that are shared between the sharing worker and some of its teammates (nodes that have or-frames associated to them in the figure). On the private region, we have the choice points produced by the sharing worker, that have not been shared yet. In Fig. 5.18b, we can see the representation of the execution trees in the sharing worker and in the auxiliary sharing area after performing the vertical stack splitting operation. It is important to note that the nodes (choice points) in the shared region were also splitted. However, since the auxiliary area will be sent to another team, the or-frames associated to those shared choice points can be removed. Therefore, we can say that in the auxiliary sharing area there is only private work.

Figure 5.19 shows in detail the *TS_vertical_splitting()* procedure which implements the vertical splitting operation in our model. The function receives as arguments the pointer to the beginning of the local stack in the auxiliary sharing area. First we initialize the variable *stack_cp* with the current choice point that is given by the register B and then we initialize the variable *sharing_state*.

Next, inside the while loop, it will traverse the local stack in the sharing worker area and in the axillary sharing area, starting in the youngest choice point. Initially, it calculates the corresponding position in the auxiliary area of the choice point in the local stack. This is done by first determining the offset between the choice point in the local stack and the one pointed by register B and then by adding this offset to



(a) Before vertical splitting (b) After vertical splitting

Figure 5.18: Representation of the vertical splitting operation done by a sharing worker

the base of the local stack in the auxiliary area. The code inside the loop follows by checking if the choice point at hand has work and, if there is available work, then it is checked the sharing state of the algorithm. There are two states: AUXILIARY_AREA and SHARING_WORKER, which are used to define the division of the choice points in the vertical splitting fashion. The AUXILIARY_AREA state means that the choice point at hand should be given to the auxiliary sharing stack and nullify in the local stack of the sharing worker. In order to do that, we begin by first checking if the choice point is shared or private. If it is shared, we must lock the or-frame associated with it, copy the next alternative in the or-frame to the choice point in the auxiliary sharing stack, nullify the access to the next alternative in the or-frame, and unlock the or-frame. Otherwise, if the choice point at hand is private, we just invalidate the available work in the choice point in the sharing worker stack by putting the next alternative field (*cp_ap*) pointing to the NO_WORK instruction and leave intact the copy of that choice point in the auxiliary sharing stack. The NO_WORK instruction is a pseudo-instruction used to mark the choice points that become without work during stack splitting. Therefore this instruction does not perform any relevant work, it simply redirects the computation to the above choice point. Returning to the code, the next step is changing the sharing state to SHARING_WORKER, meaning that the next choice point with work should be owned by the sharing worker. Thus, when the sharing state is SHARING_WORKER, the corresponding choice point in the auxiliary sharing stack is updated to the NO_WORK instruction and after that the sharing state is set again to AUXILIARY_AREA. When the choice point at hand has no work but is shared, we simply put the next available alternative of the choice point in the auxiliary

```
TS_vertical_splitting(aux_area_local_top) {

    stack_cp = B // B is a register pointing to the current choice point
    sharing_state = AUXILIARY_AREA

    while(stack_cp != ROOT_CP){
        aux_area_cp = aux_area_local_top + (stack_cp - B)
        if (has_work(stack_cp)) {
            if (sharing_state == AUXILIARY_AREA) {
                if (is_shared_cp(stack_cp)) {
                    or_fr = stack_cp->cp_or_fr
                    lock_or_frame(or_fr)
                    aux_area_cp->cp_ap = OrFr_alternative(or_fr)
                    OrFr_alternative(or_fr) = NULL
                    unlock_or_fram(or_fr)
                } else // private cp
                    stack_cp->cp_ap = NO_WORK
                sharing_state = SHARING_WORKER
            } else { // sharing_state = SHARING_WORKER
                aux_area_cp->cp_ap = NO_WORK
                sharing_state = AUXILIARY_AREA
            }
        } else //cp without work
            if (is_shared_cp(stack_cp))
                aux_area_cp->cp_ap = NO_WORK
        stack_cp = stack_cp->cp_b
    }

}
```

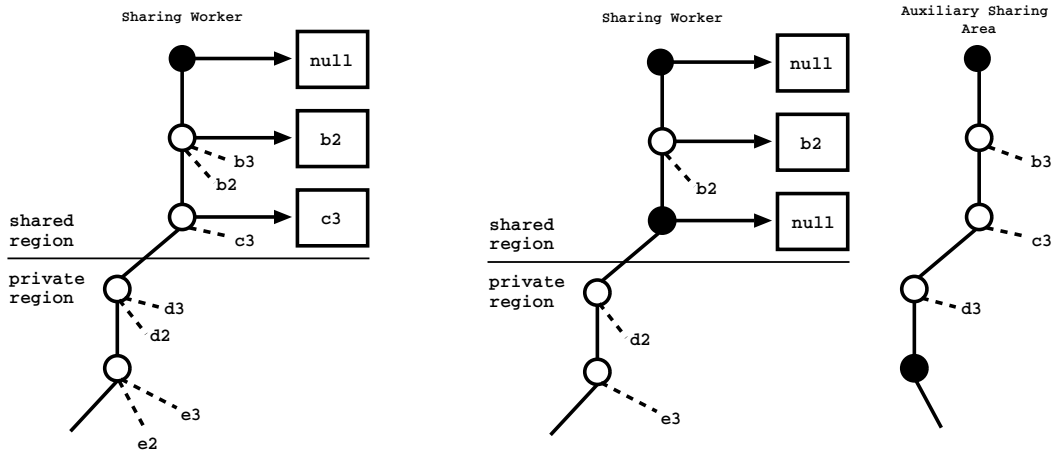
Figure 5.19: Pseudo-code for performing vertical splitting between teams

stack pointing to the NO_WORK instruction. Then we update the *stack_cp* and the *aux_area_cp* and proceed to the next iteration of the while loop.

During this process, the CP_LUB field of each choice point is also updated in order to maintain its coherence. For the sake of simplicity, we have omitted that part from the pseudo-code in Fig. 5.19.

5.7.4 Horizontal Splitting

In Fig. 5.20a we have again the same example of Fig. 5.18a but now using horizontal splitting to divide work between the stacks of the sharing worker and the stacks in the auxiliary sharing area. In this strategy, instead of splitting the choice points we split the unexplored alternatives in the choice points. The final result can be seen in Fig. 5.20b.



(a) Before horizontal splitting (b) After horizontal splitting

Figure 5.20: Representation of the horizontal splitting operation done by a sharing worker

In order to implement this splitting strategy, a new field called *split offset* was added to the choice point and or-frame data structures. This new field will help us to calculate the alternatives belonging to each team after a sharing operation. It is initialized with a value of one when a choice point is created and its value is doubled each time the choice point is splitted with another team. When a choice point is turned public the value in the *split offset* of the choice point is simply copied to the field with the same name in the corresponding or-frame. With the *split offset*, the difference is that, when backtracking, instead of trying the next alternative, as usual, now, we use the *split offset* field to calculate that alternative. For example if the *split offset* is two, then instead of trying the next alternative n, we jump n and we try the next alternative after n (offset of two).

Figure 5.21 shows in detail the *TS_horizontal_splitting()* procedure responsible for implementing the horizontal splitting operation in our model. As before with *TS_vertical_splitting()*, the function also receives as argument the pointer to the top of the local stack in the auxiliary sharing area. Again it begins by initializing the variable *stack_cp* with the

current choice point and then the variable *sharing_state*.

```
TS_horizontal_splitting(aux_area_local_top) {

    stack_cp = B // B is a register pointing to the current choice point
    sharing_state = AUXILIARY_AREA

    while(stack_cp != ROOT_CP){
        aux_area_cp = aux_area_local_top + (stack_cp - B)
        if (has_work(stack_cp)) {
            if (is_shared_cp(stack_cp)) {
                Or_fr = stack_cp->cp_or_fr
                lock_or_frame(stack_cp->or_fr)
                next_alt = next_alternative(OrFr_alternative(or_fr),
                OrFr_split_offset(stack_cp->or_fr))
                if (sharing_state == SHARING_WORKER) {
                    aux_area_cp->cp_ap = OrFr_alternative(or_fr)
                    OrFr_alternative(or_fr) = next_alt
                    sharing_state = AUXILIARY_AREA
                } else { // sharing_state == AUXILIARY_AREA
                    aux_area_cp->cp_ap = next_alt
                    sharing_state = SHARING_WORKER
                }
                aux_area_cp->cp_split_offset *= 2
                OrFr_split_offset(or_fr) *= 2
                unlock_or_frame(or_fr)
            } else { // private cp
                next_alt = next_alternative(stack_cp->cp_ap,stack_cp->cp_split_offset)
                if (sharing_state == SHARING_WORKER) {
                    stack_cp->cp_ap = next_alt
                    sharing_state = AUXILIARY_AREA
                } else { // sharing_state == AUXILIARY_AREA
                    aux_area_cp->cp_ap = next_alt
                    sharing_state = SHARING_WORKER
                }
                aux_area_cp->cp_split_offset *= 2
                stack_cp->cp_split_offset *= 2
            }
        }
        stack_cp = stack_cp->cp_b
    }
}
```

Figure 5.21: Pseudo-code for performing horizontal splitting between teams

After this initialization, the while loop, will traverse the local stack in the sharing worker area and in the axillary sharing area, starting in the youngest choice point. Then the code inside the while loop checks if the choice point at hand has work and then if it is shared. In such case, we must lock the or-frame associated with that choice point and then we call the function *next_alternative()* that receives, as arguments, the current alternative in that or-frame and the horizontal splitting offset. The function will then return the next alternative based on the given offset. At this point, we must check the sharing state of the algorithm. There are two sharing states: SHARING_WORKER and AUXILIARY_AREA, which are used to define which one of the areas will have the choice point at hand moving to the next alternative. Thus, if we are in SHARING_WORKER state, it means that it is the choice point in the sharing worker area that will be updated to the next alternative. This is done by putting the choice point in the auxiliary area pointing to the alternative in the or-frame and by updating the or-frame alternative to point to the alternative returned by the *next_alternative()* function. Finally, we change the sharing state to AUXILIARY_AREA.

Otherwise, if we are in AUXILIARY_AREA state, the choice point in the auxiliary area is put to point to the alternative returned by the *next_alternative()* function, while the alternative in the or-frame is left untouched. After that the sharing state is changed again to SHARING_WORKER. At the end, we update the splitting offset in the or-frame and in the choice point by doubling their values and we unlock the or-frame.

Our procedure also deals with private choice points. In that case, the idea behind the algorithm is the same but without or-frames associated. We simply put the choice point in the sharing worker area or in the auxiliary sharing area pointing to the next alternative returned by the *next_alternative()* function, depending if we are dealing, respectively, with the SHARING_WORKER or with the AUXILIARY_AREA state. Then we update the offset in the choice point in both stacks. At the end, we update the *stack_cp* and the *aux_area_cp* and we proceed to the next iteration of the while loop.

Like with the *TS_vertical_splitting()* procedure, the CP_LUB field of each choice point is also updated but for the sake of the simplicity we omitted that in the explanation.

5.8 Termination

Every sharing message in our system includes the load array of the team sending it. When a team receives a sharing message, it uses that information to update its own load array. From a conceptual point of view, the load array can be seen as the view that a team has about the other teams in the parallel engine. Therefore, it is not only useful for selecting teams with work but also for initiating the termination process.

When a team runs out of work, the team idle scheduler, first uses the information in the load array as a way to select the team with the highest load in order to make it a share request. Otherwise, if no such team is found, the termination process begins.

In our model, the load of a team is the sum of the loads of all workers in the team, while the load of a worker is given by the number of untried private alternatives in its execution tree. Therefore, in an extreme scenario we may have a team with load 0 but that is still busy. This may happen if, for example, all the unexplored alternatives, are in the shared region of the team. In order to distinguish these two situations, we defined that when a team is completely out of work, its load is represented as -1, instead of 0.

The termination process thus ensures that all teams are idle by traversing the load array in order to check if all have a load value of -1. If that condition is verified, a TERMINATION message is sent to all the other teams signalizing that the computation has ended. Otherwise, we restart the process of requesting work by sending sharing requests to the workers with higher load or load 0. Meanwhile, as other teams refuse sharing work, they will send their load array which may contain newer information that could help to decide if the computation has ended or not.

5.9 Fetching Answers

As we have seen before, predicates *par_probe_answers/2* and *par_get_answers/4* can be used to deal with the answers yielded by the parallel computation. The first one checks if there are new answers available while the second one is used to return answers according to a defined criteria. In order to implement these predicates, we propose a protocol that allows to request and receive a bunch of answers from the parallel engine.

5.9.1 Protocol

In order to implement this protocol, we introduced a counter per team that keeps track of the number of answers currently stored in the team and we extended the load array so that it also includes information about the number of answers per team. Remember that the load array is sent in every sharing message in our system and, thus, we will use it to propagate also this information.

The master team has an important role in this protocol, being responsible for establishing a bridge between the parallel execution and the client worker. The master team informs the client worker about the availability of answers in the parallel engine and then it is responsible for receiving and trying to fulfill the requests sent by the client worker. Sometimes, to fulfill these requests, it needs to contact the other teams in the parallel engine. The communication between the client worker, master team and the other teams is accomplished by four new messages which can be seen in Table 5.2.

Table 5.2: Messages used by the fetching answers protocol

Type of message	Extra Content
ANSWERS_FOUND	
ANSWERS_REQUEST	[NUMBER_OF_ANSWERS] (TYPE_OF_REQUEST)
ANSWERS_REPLY	[ANSWERS][(HAS_MORE_ANSWERS) || (LOAD_ARRAY)]
END_EXECUTION	

The first message is the ANSWERS_FOUND message which is sent by the master team to the client worker in order to inform that there is at least one available answer in the parallel engine. The next one, the ANSWERS_REQUEST message, is used for requesting answers and can be sent by the client worker to the master team or by the master team to the other teams. As extra content, it includes the number of answers being requested and, if sent by the client worker, information about the type of request (as stated in Chapter 3 when presenting the predicate *par_get_answers/4*). The type of request can be MAX, meaning a maximum of NUMBER_OF_ANSWERS answers must be returned, or EXACT, which indicates that NUMBER_OF_ANSWERS answers should be returned. The ANSWERS_REPLY message is used to reply to a previous ANSWERS_REQUEST message and contains the list of answers being returned. If sent by a common team to the master team, it includes the updated load array. Otherwise, if sent by the master team to the client worker, it includes the actual number of remaining answers in the parallel engine. Finally, the END_EXECUTION message is sent by the master team to the client worker to inform it that the execution

has ended. Figure 5.22 shows an example of the fetching answers process which occurs when the predicate *par_get_answers/4* is called on the client worker side.

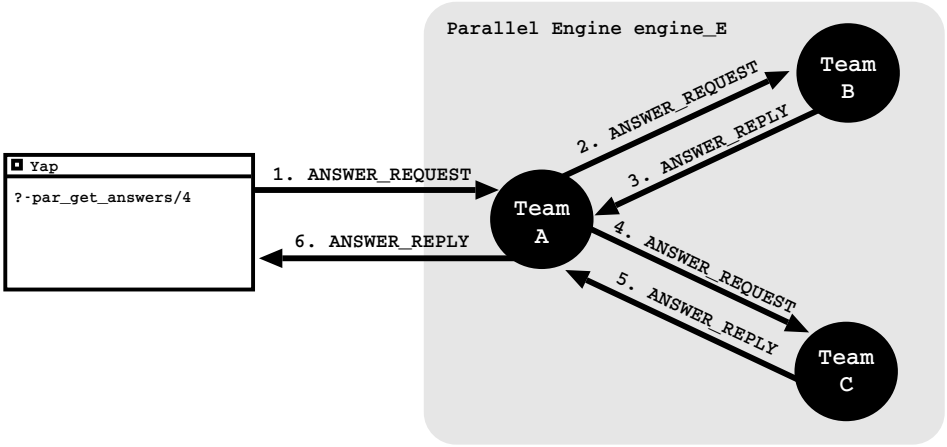


Figure 5.22: Representation of the fetching answers process

On the left side of Fig. 5.22, we have the client worker and, on the right side, we have a parallel engine composed by three teams. The client worker starts by sending a ANSWERS_REQUEST message to the master team A (messages are represented by arrows and numbered) and if the master team does not have enough answers in its team to fulfill the request, it starts contacting the other teams. In this example, the master team A starts by contacting team B. Then, after receiving a reply from team B, team A checks if the answers received from B together with the answers in its team are enough to fulfill the request. If not, it must contact the next team. In this example, the next team is team C. After team C's reply, the master team found that it already has enough answers to fulfill the request and therefore a ANSWERS_REPLY message is sent to the client worker. This message includes not only the set of answers but also information about if there are more answers in the parallel engine. In case no more answers exist in the parallel engine then, later when a new answer is found, an ANSWERS_FOUND message should be sent to the client worker informing about the availability of new answers. Finally, when the execution ends an END_EXECUTION message is sent to the client worker.

5.9.2 Implementation Details

In order to support the implementation of the two client side predicates, we extended the *engine_frame* structure with two boolean fields. The first one, named

has_answers, is made true when the parallel engine has answers. The other one, called *is_running*, is set to true by the predicate *parallel/2* and to false when the message END_EXECUTION is received. In order to receive and process the messages from the parallel engine, both predicates call the *process_message_from_parallel_engine()* procedure that can be seen in Fig. 5.23.

```
process_message_from_parallel_engine(engine_id) {

    msg = Recv(engine_id)
    type_of_message = get_type_of_message(msg)
    answers = NULL
    switch (type_of_message) {
        case ANSWERS_FOUND:
             has_answers(engine_id) = TRUE
        case END_EXECUTION :
             is_running(engine_id) = FALSE
        case ANSWERS_REPLY:
             answers = get_answers(msg)
             has_answers(engine_id) = get_has_more_answers(msg)
    }
    return answers

}
```

Figure 5.23: Pseudo-code for the *process_message_from_parallel_engine()* procedure

The function receives as argument the *engine_id* of a given parallel engine and processes the three types of messages that can be received from it by the client worker. If the message received is ANSWERS_FOUND then the corresponding *has_answers* field is set to TRUE. In the case of an END_EXECUTION message, the corresponding *is_running* field is set to false. For an ANSWERS_REPLY message, the variable *answers* is set to point to the list of answers returned by the parallel engine, *has_answers* field is set to the value received in the message that indicates if the parallel engine has more answers or not. At the end, the function returns the answers received, which has a non-NULL value when an ANSWERS_REPLY message is received.

The predicate *par_probe_answers/2*, used to check if the parallel engine has yielded any answer is implemented in the C language by the *c_probe_answers()* procedure as shown next in Fig. 5.24.

The function begins by checking if there is any pending message in which case it calls the function *process_message_from_parallel_engine()* in order to update the state of the

```
c_probe_answers(engine_id) {

    if (probe_message(engine_id)) // only ANSWERS_FOUND and END_EXECUTION messages can exist
        process_message_from_parallel_engine(engine_id)

    if (is_running(engine_id) && !has_answers(engine_id))
        return FALSE
    else
        return TRUE

}
```

Figure 5.24: Pseudo-Code for the *c_probe_answers()* procedure that implements the predicate *par_probe_answers/1*

parallel engine at hand. Then, it checks if the computation is running and it has no answers and, if so, it returns false and the predicate fails. Otherwise, it returns true and the predicate succeds.

The core of the predicate *par_get_answers/4* is also written in the C language and its pseudo-code can be seen in Fig. 5.25.

Again, first we check if there is any pending message and, if so, we call the function *process_message_from_parallel_engine()* to update to the state of the parallel engine at hand. Next, if the parallel engine is not running and it has no answers, the function returns NULL which indicates that the predicate *par_get_answers/4* must fail. In the next block of code, we wait until we know that the parallel engine has answers. While waiting, if an END_EXECUTION message is received, it returns NULL and the predicate fails. On the other hand, when we know that the parallel engine has answers, we send it an ANSWER_REQUEST message and we wait for the answers.

Now that we have seen the implementation on the client side, let us see how we have extended the parallel engine to also support this protocol. In order to do that, a new case for dealing with ANSWERS_REQUEST messages was added to the switch-case statement of the original *TS_process_message()* function (as presented before in Fig. 5.11). Figure 5.26 shows the extended code.

When an ANSWERS_REQUEST is received, the function starts by checking if it is the master team running the code. If it is the master team, it begins by determining the number of answers being requested and the type of request that we are dealing with. After that, the number of answers available in the the parallel engine is determined

```
c_get_answers(engine_id,mode,n_answers) {

    if (probe_message(engine_id))
        process_message_from_parallel_engine(engine_id)

    if (!is_running(engine_id) && !has_answers(engine_id))
        return NULL

    while (!has_answers(engine_id)) {
        process_message_from_parallel_engine(engine_id)
        if (!is_running(engine_id))
            return NULL
    }

    Send(engine_id,ANSWER_REQUEST,n_answers)
    do
        answers = process_message_from_parallel_engine(engine_id)
    while (answers == NULL)
    return answers

}
```

Figure 5.25: Pseudo-code for the *c_get_answers()* function that implements the predicate *par_get_answers/4*.

by consulting the information in the load array. Then, we check if the type of request is EXACT and if the number of answers in the parallel engine is enough to satisfy the requested number of answers. If not, the request is marked as pending and the function returns. Later, when the master team verifies that the request can be fulfilled, it reactivates the request and a response with the answers is sent to the client worker. Whenever the master team finds a new answer or receives a new message it checks if there are any pending requests and, if so, it checks if now there are enough answers in the parallel engine.

Otherwise, if at least one of the two previous conditions is not verified, we can prepare the response to be sent to the client worker. We begin by initializing the variable *ans_available* with the number of available answers in the master team and the variable *answers_list* with the list of answers in the master team. The variable *next_team* is then initialized with team B, which is the first team to be contacted to send its answers. The condition in the while loop states that the loop continues until we have the number of answers needed to fulfill the request or all the teams have been contacted.

```
msg = Recv()
type_of_message = get_type_of_message(msg)
team = get_team(msg)
...
switch (type_of_message) {
    ...
    case ANSWERS_REQUEST:
    if (is_master_team(team)) {
        n_ans_requested = get_number_of_answers(msg)
        type_request = get_type_of_request(msg)
        n_ans_parallel_engine = count_answers(load_array)

        if (type_request == EXACT && n_ans_requested > n_ans_parallel_engine){
            put_pending_request(n_ans_requested)
        } else {
            ans_available = count_answers_in_team()
            ans_list = answers_in_team()

            while (ans_available < n_ans_requested && has_next_team_to_be_called()) {
                next_team = next_team_to_be_called(next_team)
                Send(next_team, ANSWERS_REQUEST, n_ans_requested - ans_available)
                msg = Recv()
                answers = get_answers(msg)
                ans_available = ans_available + count_answers(answers)
                ans_list = ans_list + answers
            }

            Send(team,ANSWERS_REPLY,ans_list)
        }
    } else { // team != master_team
        ans_list = answers_in_team()
        Send(team,ANSWERS_REPLY)
    }
    ...
}
```

Figure 5.26: Pseudo-code extending the *TS_process_message()* procedure to support answers request messages

Inside the loop, we basically keep contacting teams sending ANSWERS_REQUEST messages, requesting the missing answers. When a non-master team receives an ANSWERS_REQUEST message, it simply sends a response with its answers. The number of answers send must be always less or equal than the number of answers requested.

Chapter 6

Performance Analysis

In this chapter, we assess, evaluate and analyse the performance of our system. In order to understand its behaviour in different scenarios, we have run several experiments using from one up to 32 workers organized in different configurations of teams, using different different stack splitting strategies and different network latency conditions.

6.1 Benchmark Programs

For benchmarking, we used a set of ten programs that we briefly describe next:

Arithmetic Puzzle – Given a list of N integers, the program finds how to place the arithmetic signs *, +, - , / and = between the N integers so that the result is a correct equation. We used a version with 10 integers, which we named *arithmetic(10)*.

Cubes – This program consists of stacking N colored cubes in a column in such a way that no color appears twice in the same column for each given side. We used a version with 10 cubes, which we called *cubes(10)*.

Ham – A program that finds hamiltonian cycles in a given graph. We used a version with 40 edges, which we called *ham(40)*.

Knight Move – Given a initial point (i,j) in a chessboard and a number N this program finds a path of length N using the allowed knight moves in chess. We used a version with paths of length 13, which we called *knight_move(13)*.

Magic Cube – A program for solving the Rubik's magic cube problem.

Map Colouring – This program checks if a given map can be colored using only three colors in such a way that no two adjacent countries have the same colour. We used a map with 46 countries, which we called *map_colouring(46)*.

Nsort – A program that sorts an array of N integers by brute force. We used a version with an array with 12 integers, which we called *nsort(12)*.

Puzzle – A program that solves a maze problem in a N*N grid by moving an empty square. We used a version with a 4*4 grid which we called *puzzle(4)*.

Queens – A program that solves the problem of placing N queens in a N*N chessboard so that no two queens may attack each other. We used a version with 14 queens, which we called *queens(14)*.

Send More Money – A program that uses brute force to find the correct substitution of letters by numbers so that SEND + MORE = MONEY.

The benchmark programs presented above find all the possible solutions for their problems.

For measuring the execution time in YapOr we used the code shown next in Fig. 6.1. Predicate *go/0* is the top query goal that will start measuring the execution time and then call the *parallel/1* predicate. The *parallel/1* predicate launches the execution of our benchmark program in parallel and executes it until finding all answers. After that the total elapsed time is calculated and printed to the screen.

```
go :- statistics(walltime, [Start,_]),
      parallel(benchmark),
      statistics(walltime, [End,_]),
      Time is End-Start,
      write('WallTime is '), write(Time).
```

Figure 6.1: Prolog program used for measuring the execution times in YapOr

For measuring the execution time in our system we run on the client worker the code shown in Fig. 6.2. The code begins by creating the parallel engine, that will be responsible for running the benchmark, using the predicate *par_create_parallel_engine/2*[1].

After that, it is safe to call the go/0 predicate that starts by measuring the execution time and then calls the predicate *par_run_goal/3* in order to run the benchmark in the parallel engine. After that a *par_barrier/1* predicate is called. This predicate was

[1]This process is not necessary in YapOr since the number of workers is passed using a flag.

```
:- par_create_parallel_engine(engine_E,(N1,4,'prolog\benchmark.pl',
                                        N2,4,'prolog\benchmark.pl').

go :- statistics(walltime, [Start,_]),
      par_run_goal(engine_E,benchmark,_),
      par_barrier(engine_E),
      statistics(walltime, [End,_]),
      Time is End-Start,
      write('WallTime is'), write(Time).
```

Figure 6.2: Prolog program used for measuring the execution times in our system

specially developed to help us in the benchmarking process and it will wait until the computation on the parallel engine side has finished. After this the elapsed time is calculated and printed to the screen.

6.2 Performance Evaluation

The environment for our experiments included two parallel machines, each one with four AMD SixCore Opteron TM 8425 HE @ 2.1 GHz (24 cores per machine, 48 cores in total) and 64 GBytes of main memory each, both running Fedora 20 with the Linux kernel 3.19.8-100 64 bits. The two machines are connected through a one Gbit router shared with other servers. In the experiments that follow, we have not collected results for more than 16 workers per machine (32 workers in total) because we do not had full access to the machines and since other users could be using the machines simultaneously, thus interfering with our results, we decided to be safer to go only until 16 workers per machine.

The Yap and YapOr versions used in our experiments are based on Yap's 6.3.4 engine which was also the base for our team implementation. The MPI implementation used was OpenMPI version 1.7.3.

The results presented next were obtained by executing each benchmark 10 times and by calculating the average of that executions. For simplicity of presentation, in most tables, we only present speed ups or ratios against the base case. Full results including also the execution times and the coefficient of variation can be seen on Appendix A. The coefficient of variation is defined as the ratio of the standard deviation to the mean and gives an idea of the dispersion of the results in the 10 runs.

In the next sections we use the following convention to refer to systems and their configurations. Our system is represented by *Teams(T,W)* where T represents the number of teams and W the number of workers per team (for a total number of T*workers). Standard stack splitting (without teams) is represented by SS(W) and YapOr by YapOr(W) where W is the total number of workers.

6.2.1 Overheads over YapOr

In order to measure the impact of our system we start by analyzing the overheads introduced over YapOr. For that, we ran our set of benchmarks with YapOr with one worker (YapOr(1)) and compare it against the two versions of our system, using vertical splitting (VS) and horizontal splitting (HS), when executing with a single team with one worker (Teams(1,1)).

Before seeing the overheads of our implementation, let us see the overheads introduced by YapOr over sequential Yap. Table 6.1 shows the execution times in milliseconds for Yap and YapOr(1) and the corresponding overheads of YapOr(1) over Yap.

Table 6.1: Overheads added by YapOr with a single worker to sequential Yap

Program	**Yap**	**YapOr(1)**	**YapOr(1)/Yap**
arithmetic(10)	304.455	361.439	1.19
cubes(10)	52.172	67.503	1.29
ham(40)	95.375	121.444	1.27
knight_move(13)	321.563	395.602	1.23
magic_cube	34.339	46.872	1.36
map_colouring(46)	140.376	178.729	1.27
nsort(12)	317.252	406.292	1.28
puzzle(4)	12.708	17.538	1.38
queens(14)	466.543	552.275	1.18
send_more	53.171	69.684	1.31
Average			1.28

The overhead of YapOr over Yap is nearly 28% ranging from 18% in the *queens(14)* benchmark to 38% in the *puzzle(4)* benchmark. These results are very different from those presented by Santos Costa et al. [37] using the same machine which were around 3% but with a different version of Yap and YapOr (version 6.0.1) and with a different set of benchmarks. We thus decided to repeat the experiments done by Santos Costa

et al. and we were able to reproduce their results. But with version 6.3.4 of Yap we got again higher overheads, this time around 31%. Another interesting detail is that comparing both versions of Yap, version 6.3.4 is around 45% slower than version 6.0.1 for our set of benchmarks.

Now, let us see the overheads introduced by our team implementation over YapOr. Table 6.2 shows the overheads for both the vertical splitting and horizontal splitting versions.

Table 6.2: Overheads added by our team implementation to YapOr when running with a single worker

	Teams(1,1)		**Teams(1,1)/YapOr(1)**	
Program	VS	HS	VS	HS
arithmetic(10)	366.632	361.453	1.01	1.00
cubes(10)	64.04	71.753	0.95	1.06
ham(40)	124.301	121.586	1.02	1.00
knight_move(13)	377.642	390.299	0.95	0.99
magic_cube	53.049	49.358	1.13	1.05
map_colouring(46)	175.58	182.361	0.98	1.02
nsort(12)	400.239	378.670	0.99	0.93
puzzle(4)	18.533	18.508	1.06	1.06
queens(14)	540.263	542.387	0.98	0.98
send_more	71.506	68.889	1.03	0.99
Average			1.01	1.01

As we can see, the overheads added are on average 1% for both scheduling strategies. Observing the results in more detail, we can see that there are some benchmarks which have no overhead and, in fact, they are faster in our implementation than in YapOr. For example, the *nsort(12)* benchmark in horizontal splitting is 7% faster in our team implementation than in YapOr. In theory this should not happen since our implementation adds an extra layer to YapOr. Furthermore, we were expecting that horizontal splitting had higher overheads than vertical splitting since horizontal splitting performs extra operations, even when executing with a single worker. One extra operation is the initialization of the choice point offset field whenever a new choice point is created. Another extra operation is related with the backtracking process, that needs to determine the next alternative to be tried based on the offset field.

After looking carefully to the code we believe that the results we got in the overheads added by YapOr to Yap and between the two versions of our implementation might be related with compilation issues. Comparing Yap version 6.0.1 with Yap version 6.3.4 we can see that more code was added to Yap's abstract machine which may explain the difference of 45% we saw previously. YapOr also adds more code to the abstract machine, therefore we think that the compiler might have problems optimizing the code which may explain the differences between Yap and YapOr in version 6.3.4.

We also measured the overheads added to YapOr when we execute our system with vertical splitting and horizontal splitting with more than one worker. For that, we run YapOr with 4, 8 and 16 workers and we compare it with our team implementation running with the same number of workers in one team. It is important to note that in configurations with just one team, the stack splitting strategies will not be used for distributing work and thus the execution should be identical to YapOr. In Table 6.3 we show the execution times obtained for YapOr and the overheads added by the versions of our system.

Table 6.3: Overheads added by our, implementation to YapOr when running with 1 team with the same number of workers

	4 workers			**8 workers**			**16 workers**		
Program	**YapOr(4)**	**Teams(1,4)**		**YapOr(8)**	**Teams(1,8)**		**YapOr(16)**	**Teams(1,16)**	
		VS	HS		VS	HS		VS	HS
arithmetic(10)	167.731	1.01	0.98	117.421	0.95	0.95	106.649	0.99	0.95
cubes(10)	16.829	0.96	1.08	8.449	0.96	1.08	4.253	0.96	1.09
ham(40)	30.498	1.04	1.01	15.162	1.06	1.03	7.586	1.07	1.05
knight_move(13)	99.862	0.96	0.98	50.350	0.96	0.98	25.192	0.96	0.99
magic_cube	11.862	1.13	1.04	5.945	1.12	1.05	2.971	1.14	1.07
map_colouring(46)	45.027	0.98	1.01	22.499	0.99	1.02	11.304	1.00	1.03
nsort(12)	101.896	1.01	0.95	51.310	1.00	0.96	25.571	1.02	0.98
puzzle(4)	4.421	1.06	1.07	2.206	1.07	1.09	1.113	1.06	1.12
queens(14)	138.120	1.00	0.98	69.280	0.99	0.98	34.329	1.01	1.00
send_more	17.450	1.03	1.01	8.766	1.02	1.01	4.425	1.01	1.02
Average		1.02	1.01		1.01	1.02		1.02	1.03

The overheads for vertical splitting are on average 2%, 1% and 2% for 4, 8 and 16 workers, respectively. For horizontal splitting are 1%, 2% and 3% for the same numbers of workers, respectively. In summary, our results show that our system adds a small overhead to YapOr which makes it also adequate for running in shared memory.

6.2.2 Teams in the Same Machine

In the previous subsections, we have seen that our system achieves comparable results to YapOr when we run it with just one team. Here, we want to assess the impact in terms of speed ups when the number of teams increases. So, we run again experiments for 4, 8 and 16 workers with different configurations of teams using only one of the machines available. It is important to note that when two processes in the same machine exchange messages, by default, OpenMPI uses shared memory instead of using the loopback interface. Such characteristic guarantees that the communication latency is minimal allowing us to have a more accurate and clear idea about the cost of increasing the number of teams.

Table 6.4 shows the speed ups obtained by our experiments using the different configurations of teams when compared with the execution of YapOr with a single worker. The table is divided in three parts: execution with 4, 8 and 16 workers. The columns represent the configurations of teams tested where the number of teams in which workers are divided increases from left to right and all the teams have the same number of workers. For each team configuration, we show the results for both vertical and horizontal splitting, columns VS and HS, respectively.

As expected, in general, the benchmark programs see their speed ups decrease when we increase the number of teams. This decrease is more visible in the configuration of 16 workers. By comparing the speed ups obtained for configuration Teams(1,16) with configuration Teams(16,1) (both with 16 workers in total) we can see that there is a decrease of around 2 for vertical splitting (from 14.30 to 12.27) and around 1 for horizontal splitting (from 14.14 to 13.00). The only benchmark that increases the speed ups when the number of team increases is the *arithmetic(10)* benchmark, which seems to benefit from the higher number of stack splitting operations when we have more teams.

Regarding the stack splitting strategies, horizontal splitting seems to have a slightly advantage over vertical splitting when the number of teams increase. In the case of 16 workers, the difference starts to be in favor of vertical splitting for configuration Teams(1,16) (14.30 against 14.14) and then for configuration Teams(16,1), the difference turns in favor of horizontal splitting (13.00 against 12.27).

As we stated earlier, since all teams are running on the same machine these experiments show how teams behave in an optimal scenario, i.e., which allows us to have a clearer view on the impact of teams in terms of performance. In general, when reading

Table 6.4: Speed ups comparing our implementation running in a single machine against YapOr with one worker

4 workers	Teams(1,4)		Teams(2,2)		Teams(4,1)	
Program	VS	HS	VS	HS	VS	HS
arithmetic(10)	2.14	2.20	2.83	2.82	3.86	3.86
cubes(10)	4.17	3.73	3.76	3.48	3.42	3.31
ham(40)	3.82	3.94	3.66	3.77	3.57	3.69
knight_move(13)	4.13	4.04	3.70	3.73	3.39	3.51
magic_cube	3.50	3.80	3.41	3.69	3.40	3.61
map_colouring(46)	4.04	3.92	3.86	3.80	3.76	3.71
nsort(12)	3.97	4.20	3.68	4.03	3.68	3.85
puzzle(4)	3.74	3.70	3.70	3.70	3.61	3.50
queens(14)	4.02	4.07	3.67	3.69	3.23	3.46
send_more	3.88	3.97	3.76	3.84	3.68	3.79
Average	3.74	3.76	3.60	3.66	3.56	3.63

8 workers	Teams(1,8)		Teams(2,4)		Teams(4,2)		Teams(8,1)	
Program	VS	HS	VS	HS	VS	HS	VS	HS
arithmetic(10)	3.23	3.23	4.57	4.42	6.20	6.11	7.42	7.47
cubes(10)	8.30	7.42	7.74	7.04	7.43	6.85	6.54	6.49
ham(40)	7.57	7.77	7.38	7.46	7.12	7.45	7.03	7.27
knight_move(13)	8.22	8.02	7.67	7.62	7.38	7.42	6.79	6.97
magic_cube	7.01	7.53	6.76	7.41	6.74	7.18	6.56	7.21
map_colouring(46)	8.03	7.77	7.84	7.64	7.76	7.56	7.43	7.31
nsort(12)	7.93	8.28	7.29	7.98	7.29	7.83	7.29	7.67
puzzle(4) :	7.44	7.26	7.30	7.03	7.21	7.06	6.51	6.88
queens(14)	8.04	8.10	7.53	7.71	7.19	7.42	6.73	6.90
send_more	7.78	7.89	7.64	7.70	7.48	7.55	7.17	7.51
Average	7.35	7.33	7.17	7.20	7.18	7.24	6.95	7.17

16 workers	Teams(1,16)		Teams(2,8)		Teams(4,4)		Teams(8,2)		Teams(16,1)	
Program	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
arithmetic(10)	3.43	3.58	6.03	6.37	9.53	9.62	11.63	11.80	12.93	13.13
cubes(10)	16.52	14.58	15.60	13.90	15.10	13.68	14.09	13.26	11.61	11.46
ham(40)	14.99	15.24	14.67	14.66	14.56	14.66	12.63	14.16	13.20	13.53
knight_move(13)	16.36	15.85	15.38	15.22	15.19	14.94	14.49	14.50	13.20	13.52
magic_cube	13.88	14.76	13.24	14.30	13.21	14.55	12.44	14.20	9.57	12.46
map_colouring(46)	15.88	15.40	15.60	14.90	15.41	14.92	15.12	14.65	13.45	13.25
nsort(12)	15.64	16.29	14.42	15.76	14.38	15.43	14.38	15.38	14.01	15.04
puzzle(4)	14.84	14.12	14.19	13.63	13.91	13.26	12.67	12.01	9.17	10.94
queens(14)	15.88	16.16	15.21	15.56	14.97	15.29	14.22	14.76	12.97	13.36
send_more	15.58	15.42	15.13	14.83	15.02	14.96	14.45	14.71	12.53	13.29
Average	14.30	14.14	13.95	13.91	14.13	14.13	13.61	13.94	12.27	13.00

the results horizontally, we can see that when the number of teams increases the speed ups show only a slightly decrease we thus argue that our system is well design and that teams have a small impact in terms of performance.

6.2.3 Teams in Distributed Machines

In real environments, the network latency has a greater impact on the performance of a distributed program and thus, in this subsection, we want to assess how teams behave in an environment with higher latencies. Since we had only two parallel machines available, we tried to emulate the existence of more machines, such that, we could create scenarios where each team of workers always runs on a separate machines. In order to do that, we acted in the following way: (i) we configured OpenMPI to use the loopback interface and the TCP protocol for communications between processes even if they are in the same machine (by default OpenMPI uses shared memory for this type of communications); and (ii) we used the *tc* command to add more 0.06 milliseconds to the 0.02 milliseconds of latency in the loopback interface in order to simulate the latency that we have observed between the two physical machines, which is about 0.08 milliseconds.

To assess how teams behave in this environment we have run experiments for 16, 24 and 32 workers with different configurations of teams. As before, teams were created with the same number of workers but now they were divided equitably between the two physical machines. For example, consider the case of 24 workers and 4 teams. In such case, we launch two teams in each machine and each team is then created with six workers each.

Table 6.5 shows the speed ups results achieved by the different configurations of teams when compared with the execution of YapOr with a single worker. The table is divided in three parts: execution with 16, 24 and 32 workers. The columns represent the configurations of teams tested where the number of teams in which workers are divided increases from left to right. For each configuration, we show the results for both vertical and horizontal splitting, columns VS and HS, respectively.

Please note that the experiments with 16 workers were already run in the previous subsection but in one single machine. By comparing both, we may have a clear view on how the latency affects the performance. With the workers divided in just two teams, configurations Teams(2,8), the impact is limited. In Table 6.4, we had on average speed ups of 13.95 and 13.91 for vertical and horizontal splitting, respectively, and

Table 6.5: Speed ups comparing our implementation running in several machines against YapOr with one worker

16 workers	Teams(2,8)		Teams(4,4)		Teams(8,2)		Teams(16,1)	
Program	VS	HS	VS	HS	VS	HS	VS	HS
arithmetic(10)	6.10	6.08	8.07	8.04	9.91	10.20	8.31	7.79
cubes(10)	14.29	13.31	12.94	12.65	9.82	10.43	5.31	6.45
ham(40)	14.17	14.15	13.49	13.74	6.50	11.60	6.92	6.39
knight_move(13)	15.36	14.94	14.86	14.73	13.33	13.30	10.51	10.24
magic_cube	12.03	13.95	11.26	12.73	9.00	11.77	4.76	7.32
map_colouring(46)	15.17	14.72	14.54	14.06	12.23	12.15	7.11	7.17
nsort(12)	14.24	15.48	13.91	14.76	12.98	13.75	10.23	9.91
puzzle(4)	11.61	12.56	10.02	9.94	7.36	7.62	3.36	4.56
queens(14)	15.00	15.46	14.33	15.03	11.88	14.10	8.09	11.23
send_more	14.02	13.97	13.02	13.59	9.87	12.60	5.82	8.36
Average	13.20	13.46	12.64	12.93	10.29	11.75	7.04	7.94

24 workers	Teams(2,12)		Teams(4,6)		Teams(6,4)		Teams(12,2)		Teams(24,1)	
Program	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
arithmetic(10)	6.65	6.56	10.52	10.41	12.02	12.10	11.69	11.72	8.33	8.36
cubes(10)	20.36	19.09	18.61	17.61	16.66	16.59	11.35	12.00	5.54	6.83
ham(40)	20.83	20.49	19.81	20.06	16.86	18.25	6.66	14.33	7.01	6.47
knight_move(13)	22.92	22.06	22.07	21.54	21.12	20.49	17.59	17.02	12.68	11.50
magic_cube	17.26	20.51	16.20	18.09	14.52	17.04	9.43	13.54	4.11	7.34
map_colouring(46)	22.39	20.91	20.79	20.29	20.11	18.58	14.07	13.88	7.44	7.29
nsort(12)	21.02	22.42	20.28	21.58	19.42	20.82	16.78	17.92	11.52	9.88
puzzle(4)	15.34	16.85	13.00	13.78	11.34	11.83	7.71	8.18	3.49	4.76
queens(14)	21.97	23.09	21.30	22.48	20.07	21.83	13.71	20.06	9.95	12.74
send_more	19.67	18.84	19.04	18.97	16.91	18.05	12.51	14.84	5.79	8.52
Average	18.84	19.08	18.16	18.48	16.90	17.56	12.15	14.35	7.59	8.37

32 workers	Teams(2,16)		Teams(4,8)		Teams(8,4)		Teams(16,2)		Teams(32,1)	
Program	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
arithmetic(10)	6.70	6.63	11.07	10.61	15.47	15.14	13.45	12.69	8.94	8.79
cubes(10)	26.04	22.99	23.59	21.27	19.27	19.65	11.72	12.63	5.61	7.27
ham(40)	26.18	26.65	25.31	26.28	20.13	21.52	6.91	15.36	7.52	7.42
knight_move(13)	30.08	28.10	28.97	28.16	26.81	26.16	20.30	18.09	13.39	12.27
magic_cube	21.65	26.03	20.28	22.92	15.94	20.81	10.02	14.20	4.57	7.85
map_colouring(46)	29.00	25.75	27.46	26.14	23.26	21.96	14.72	14.29	7.88	7.16
nsort(12)	27.80	28.38	26.50	27.23	24.07	25.05	17.56	18.03	11.55	11.24
puzzle(4)	19.95	21.21	15.25	17.73	13.77	13.21	7.77	8.54	3.40	5.05
queens(14)	28.67	30.62	27.55	29.42	23.99	27.99	15.59	23.64	8.74	13.29
send_more	24.91	22.99	23.86	23.74	19.93	22.55	12.80	15.98	5.45	8.23
Average	24.10	23.94	22.98	23.35	20.26	21.40	13.08	15.35	7.70	8.86

now in Table 6.5 we have speed ups of 13.20 and 13.46, respectively. As we increase the number of teams the impact is more clear. For example, in the most extreme configuration, Teams(16,1), the decrease is from 12.27 to 7.04 in vertical splitting and from 13.00 to 7.94 in horizontal splitting. The *arithmetic(10)* which, in the previous experiment, was the only benchmark to increase the speed ups when increasing the number of teams now also decreases in configuration Teams(16,1).

For 24 workers, we start with speed ups of 18.84 and 19.08 for vertical and horizontal splitting, respectively, when we divide the workers in two teams, configuration Teams(2,12), and then the speed ups start decreasing until they reach 7.59 for vertical splitting and 8.37 for horizontal splitting for configuration Teams(24,1). For 32 workers, the speed ups begin in 24.10 for vertical splinting and 23.94 for horizontal splitting and then they decrease to 7.70 and 8.86, respectively. In general the decrease seems to be more clear when we have more than 10 teams.

Overall, horizontal splitting achieves better results for distributing work as we increase the number of teams. This trend is in line with the results from the previous subsection but now it became even more clear. Another interesting detail is that the coefficient of variation is now much higher than that we have seen on the previous experiments. This means that there is a higher flotation among the results obtained in the ten executions.

6.2.4 Scalability

Looking at the previous results we may draw two conclusions: (i) the configurations with all workers in the same team (YapOr approach) have the best speed ups on average; and (ii) the configurations with all workers in different teams (standard stack splitting approach) clearly show the major scalability problems. For example, for configuration Teams(16,1) we got speed ups of 7.04 and 7.94 for vertical and horizontal splitting, respectively, and when we double the number of workers to configuration Teams(32,1) the speed ups barely increased to 7.70 and 8.86, respectively.

Although the YapOr approach has the best speed ups, it is limited to one machine. On the other hand, the standard stack splitting solves that limitation but it has scalability problems. By combining both approaches, our approach has the best of both worlds. Table 6.6 compares the possible usage of the three or-parallel approaches for different scenarios of clusters of multicore machines. On the columns, we have the number of machines per cluster, which increases from left to right and, on the rows, we have the

number of cores per machine, which increases from top to bottom.

Table 6.6: Possible usage of the three or-parallel approaches for different scenarios of clusters of multicore machines

		clusters with 1 machine	clusters with 2 machines	clusters with 4 machines	clusters with 8 machines
4 cores	per machine	YapOr(4) SS(4) Teams(1,4)	SS(8) Teams(2,4)	SS(16) Teams(4,4)	SS(32) Teams(8,4)
8 cores	per machine	YapOr(8) SS(8) Teams(1,8)	SS(16) Teams(2,8)	SS(32) Teams(4,8)	SS(64) Teams(8,8)
16 cores	per machine	YapOr(16) SS(16) Teams(1,16)	SS(32) Teams(2,16)	SS(64) Teams(4,16)	SS(128) Teams(8,16)

As mentioned before YapOr is limited to the configurations with a single machine while standard stack splitting and our team's approach can run taking advantage of all the cores available.

In order to compare the three approaches, we used again the simulation method described in the previous subsection that enable us to simulate different machines and we run our set of benchmarks for the configurations presented in Table 6.6 that use at most 32 cores. Tables 6.7 and 6.8 show the results.

Table 6.7 is similar to Table 6.6, the number of machines is represented in the columns and increases from left to right and the number of cores per machine is represented in the rows and increases from top to bottom. For the configurations with a single machine with N cores, it shows the execution times for Teams(1,N) (which is the equivalent to YapOr(N)). For the other configurations, it shows the ratio between Teams(1,N)/Teams(T,N). For example, for 2 machines with 4 cores each it shows the ratio Teams(1,4)/Teams(2,4). So, in practice, we want to see how much the execution time reduces when we add more machines to the cluster.

We can see that we are able to reduce the execution time significantly by adding more machines. When we first add one machine we are able to almost double the ratio for the three types of machine for both vertical splitting and horizontal splitting with values ranging from 1.70 to 1.88, which is very close to the theoretical optimal linear

Table 6.7: Execution times in milliseconds for the clusters with 1 machine and the corresponding ratios for the clusters with 2, 4 and 8 machines for the case of machines with 4, 8 and 16 cores each

		1 machine		2 machines		4 machines		8 machines	
4 cores	**Program**	VS	HS	VS	HS	VS	HS	VS	HS
	arithmetic(10)	169.015	164.563	1.90	1.89	3.77	4.38	7.23	6.97
	cubes(10)	16.182	18.096	1.78	1.84	3.10	3.67	4.57	5.37
	ham(40)	31.791	30.789	1.87	1.86	3.53	3.72	5.03	5.60
	knight_move(13)	95.698	97.945	1.84	1.87	3.60	3.70	6.39	6.54
	magic_cube	13.374	12.333	1.83	1.90	3.21	3.83	4.65	5.52
	map_colouring(46)	44.189	45.600	1.91	1.92	3.60	3.81	5.65	5.60
	nsort(12)	102.433	96.799	1.82	1.90	3.51	3.68	6.14	5.98
	puzzle(4)	4.688	4.734	1.71	1.81	2.68	3.58	3.45	3.64
	queens(14)	137.521	135.580	1.87	1.89	3.57	3.75	6.32	6.93
	send_more	17.982	17.564	1.88	1.92	3.36	3.77	4.91	5.78
	Average			1.84	1.88	3.39	3.79	5.44	5.79
8 cores	**Program**	VS	HS	VS	HS	VS	HS		
	arithmetic(10)	111.728	111.740	1.88	1.88	3.42	3.28		
	cubes(10)	8.130	9.098	1.72	1.79	2.84	2.87		
	ham(40)	16.050	15.639	1.87	1.82	3.34	3.38		
	knight_move(13)	48.124	49.349	1.87	1.86	3.52	3.51		
	magic_cube	66.88	6.227	1.72	1.85	2.89	3.04		
	map_colouring(46)	22.267	22.995	1.89	1.89	3.42	3.36		
	nsort(12)	51.213	49.062	1.80	1.87	3.34	3.29		
	puzzle(4)	2.358	2.415	1.56	1.73	2.05	2.44		
	queens(14)	68.690	68.155	1.87	1.91	3.43	3.63		
	send_more	8.961	8.828	1.80	1.77	3.07	3.01		
	Average			1.80	1.84	3.13	3.18		
16 cores	**Program**	VS	HS	VS	HS				
	arithmetic(10)	105.305	101.033	1.95	1.85				
	cubes(10)	4.085	4.629	1.58	1.58				
	ham(40)	8.102	7.969	1.75	1.75				
	knight_move(13)	24.174	24.957	1.84	1.77				
	magic_cube	3.378	3.176	1.56	1.76				
	map_colouring(46)	11.256	11.604	1.83	1.67				
	nsort(12)	25.975	24.946	1.78	1.74				
	puzzle(4)	1.182	1.242	1.34	1.50				
	queens(14)	34.778	34.177	1.81	1.90				
	send_more	4.474	4.520	1.60	1.49				
	Average			1.70	1.70				

value of 2. When we increase the number of machines to four, we got ratios of 3.39 and 3.79 for vertical splitting and horizontal splitting, respectively, for machines of 4 cores and of 3.13 and 3.18 for machines of 8 cores. For 16 cores machines, we were no longer able to simulate the 4 machines scenario since it surpases our establish limit of 32 cores. For the same reason, we only run an experiment with 8 machines with 4 cores. In general the results show that we are able to benefit from the addition of more machines and benefit from all the cores available which it would not be possible with YapOr.

In the next experiment, we want to put in perspective our system with the standard stack splitting approach. For that, in Table 6.8, we present the ratios SS(T*W)/Teams(T,W), again for the same previous configurations of machines and cores. For example, for two machines with four cores each, we show the ratio SS(8)/Teams(2,4). The execution times for standard stack splitting were taken by running our system with all workers in different teams. However, when using our simulation we can not simulate the situation where some teams are located in the same machine and others don't, we are only able to simulate situations where all teams are in the same machine or all teams are in different machine. Thus, to present the results for standard stack splitting we have decided to run it in two different conditions. The first one is using our simulator as we did before this could lead to worse results if compared to a real situation where we can have teams in the same machine and thus benefit from the MPI implementation. The second one is with no simulation at all but also dividing the teams equitably by our two machines this could lead to better results if compared to a real situation where we have to pay the cost of having more teams in different machines. In Table 6.8 the results are presented as an interval [X, Y] where X is related with the ratio obtained without simulation and Y with the ratio obtained with simulation.

By observing the results in Table 6.8, we can see that our system is significantly faster than the standard stack splitting approach. Overall, as we expected, standard stack splitting shows best results when running without simulation. Comparing both stack splitting strategies, horizontal splitting is the one presenting lower ratios. The biggest differences between our team approach and stack splitting is seen in the configuration of 2 machines of 16 cores where we got ratios of [3.06, 3.57] for vertical splitting and [2.59, 2.85] for horizontal splitting. This is because standard stack splitting can not benefit from the fact of having 16 cores in the same machine and use shared memory to synchronize them as in our team approach.

Experimental results showed that our implementation, when compared against YapOr, achieves identical speed ups for shared memory and, when running on clusters of

Table 6.8: Comparison between our approach and the standard stack splitting approach

		1 machine		2 machines		4 machines		8 machines	
4 cores	**Program**	VS	HS	VS	HS	VS	HS	VS	HS
	arithmetic(10)	[0.58, 0.59]	[0.60, 0.60]	[0.66, 0.65]	[0.65, 0.65]	[0.89, 0.97]	[1.19, 1.23]	[1.67, 1.73]	[1.76, 1.74]
	cubes(10)	[1.30, 1.34]	[1.17, 1.19]	[1.53, 1.51]	[1.26, 1.33]	[2.15, 2.44]	[1.88, 2.12]	[3.16, 3.40]	[2.51, 2.76]
	ham(40)	[1.12, 1.15]	[1.12, 1.14]	[1.19, 1.36]	[1.18, 1.33]	[1.47, 1.95]	[1.65, 2.29]	[1.93, 2.56]	[2.58, 2.98]
	knight_move(13)	[1.23, 1.23]	[1.17, 1.17]	[1.17, 1.21]	[1.15, 1.18]	[1.37, 1.41]	[1.41, 1.46]	[1.81, 1.97]	[2.12, 2.15]
	magic_cube	[1.14, 1.18]	[1.08, 1.10]	[1.37, 1.45]	[1.12, 1.17]	[2.08, 2.37]	[1.87, 1.99]	[3.23, 3.57]	[2.63, 2.67]
	map_colouring(46)	[1.11, 1.11]	[1.09, 1.10]	[1.25, 1.28]	[1.19, 1.25]	[1.78, 2.05]	[1.93, 2.08]	[2.65, 2.90]	[2.72, 3.06]
	nsort(12)	[1.10, 1.11]	[1.11, 1.11]	[1.07, 1.08]	[1.09, 1.17]	[1.26, 1.36]	[1.35, 1.56]	[1.95, 2.11]	[1.89, 2.23]
	puzzle(4)	[1.23, 1.28]	[1.17, 1.23]	[1.61, 1.97]	[1.41, 1.59]	[2.25, 2.98]	[2.31, 2.90]	[2.87, 3.79]	[2.32, 2.67]
	queens(14)	[1.23, 1.22]	[1.19, 1.19]	[1.22, 1.29]	[1.17, 1.17]	[1.46, 1.77]	[1.34, 1.36]	[2.47, 2.91]	[1.96, 2.12]
	send_more	[1.15, 1.14]	[1.08, 1.09]	[1.36, 1.48]	[1.15, 1.17]	[2.22, 2.23]	[1.70, 1.79]	[2.83, 3.49]	[2.54, 2.79]
	Average	[1.12, 1.14]	[1.08, 1.09]	[1.24, 1.33]	[1.14, 1.20]	[1.69, 1.95]	[1.66, 1.88]	[2.46, 2.84]	[2.30, 2.52]
8 cores	**Program**	VS	HS	VS	HS	VS	HS		
	arithmetic(10)	[0.52, 0.51]	[0.51, 0.51]	[0.68, 0.73]	[0.75, 0.78]	[1.19, 1.24]	[1.22, 1.21]		
	cubes(10)	[1.71, 1.68]	[1.36, 1.44]	[2.37, 2.69]	[1.83, 2.07]	[3.91, 4.20]	[2.67, 2.93]		
	ham(40)	[1.26, 1.44]	[1.24, 1.41]	[1.55, 2.05]	[1.59, 2.21]	[2.55, 3.37]	[3.07, 3.54]		
	knight_move(13)	[1.26, 1.30]	[1.22, 1.25]	[1.42, 1.46]	[1.41, 1.46]	[1.99, 2.16]	[2.26, 2.29]		
	magic_cube	[1.49, 1.58]	[1.17, 1.22]	[2.22, 2.53]	[1.79, 1.91]	[4.02, 4.44]	[2.88, 2.92]		
	map_colouring(46)	[1.29, 1.33]	[1.23, 1.29]	[1.85, 2.13]	[1.90, 2.05]	[3.18, 3.49]	[3.24, 3.65]		
	nsort(12)	[1.17, 1.19]	[1.14, 1.21]	[1.30, 1.39]	[1.36, 1.56]	[2.13, 2.29]	[2.05, 2.42]		
	puzzle(4)	[1.87, 2.29]	[1.52, 1.72]	[2.60, 3.46]	[2.19, 2.75]	[3.40, 4.49]	[3.06, 3.51]		
	queens(14)	[1.31, 1.39]	[1.23, 1.23]	[1.52, 1.85]	[1.35, 1.38]	[2.68, 3.15]	[2.04, 2.21]		
	send_more	[1.45, 1.57]	[1.19, 1.22]	[2.39, 2.41]	[1.59, 1.67]	[3.55, 4.38]	[2.63, 2.88]		
	Average	[1.33, 1.43]	[1.18, 1.25]	[1.79, 2.07]	[1.58, 1.78]	[2.86, 3.32]	[2.51, 2.76]		
16 cores	**Program**	VS	HS	VS	HS				
	arithmetic(10)	[0.38, 0.41]	[0.41, 0.41]	[0.72, 0.75]	[0.76, 0.75]				
	cubes(10)	[2.74, 3.11]	[1.83, 2.01]	[4.31, 4.64]	[2.89, 3.16]				
	ham(40)	[1.64, 2.17]	[1.78, 2.05]	[2.63, 3.48]	[3.11, 3.59]				
	knight_move(13)	[1.51, 1.56]	[1.27, 1.29]	[2.07, 2.25]	[2.25, 2.29]				
	magic_cube	[2.56, 2.92]	[1.85, 1.88]	[4.29, 4.74]	[3.27, 3.31]				
	map_colouring(46)	[1.94, 2.23]	[1.91, 2.15]	[3.36, 3.68]	[3.19, 3.60]				
	nsort(12)	[1.42, 1.53]	[1.23, 1.45]	[2.23, 2.41]	[2.14, 2.53]				
	puzzle(4)	[3.32, 4.41]	[2.44, 2.80]	[4.44, 5.87]	[3.66, 4.20]				
	queens(14)	[1.61, 1.96]	[1.12, 1.22]	[2.79, 3.28]	[2.12, 2.30]				
	send_more	[2.66, 2.67]	[1.71, 1.87]	[3.71, 4.57]	[2.54, 2.79]				
	Average	[1.98, 2.30]	[1.55, 1.71]	[3.06, 3.57]	[2.59, 2.85]				

multicores, is able to increase speed ups as we increase the number of workers per team, thus taking advantage of the maximum number of cores in a machine, and to increase speed ups as we increase the number of teams, thus taking advantage of adding more machines to a cluster. Furthermore, it has the advantage of not being limited to only a single machine, as it happens with YapOr, and it does not suffers from scalability problems like a pure stack splitting approach. We thus argue, that our approach combines the best of both worlds.

Chapter 7

Conclusions

In this last chapter, we summarize the main contributions of this thesis and we highlight possible directions for further research aiming to bring new functionalities and improvements to our system.

7.1 Main Contributions

This thesis proposes and discusses the implementation of a new computational model designed to explore implicit or-parallelism in clusters of multicore. Next, we summarize the main contributions of this work:

Novel computational or-parallel model. We have proposed a new layered computational model combining techniques for shared and distributed memory approaches with the aim of running Prolog code efficiently in clusters of multicores. To the best of our knowledge, this is the first model specially designed to explore such combination. In our proposal, we have introduced concepts and presented algorithms that may be used as guidelines to others willing to implement a similar model. Next we enumerate the most relevant contributions of the new model:

- The concept of teams which was borrowed from previous and/or-parallel systems but redefined by us in order to be able to combine techniques for shared and distributed memory approaches;

- Scheduling algorithms and sharing protocols to efficiently distribute work between teams;
- A distributed termination algorithm that ensures the complete execution of the program;
- A protocol that allows answers to be fetched during the execution of the program.

Implementation of our or-parallel model. We showed all the important and relevant details about how we have extended YapOr with an extra layer combining the existing shared memory approach with the new distributed one and how MPI was used for launching the teams of workers and to enable the communication between those teams.

A new syntax. Set of built-in predicates designed to allow the user, to manage and interact asynchronously with an or-parallel engine in our model.

Performance study. We have tested the system implementing the new computational model with several different configurations and under different conditions. From the results obtained, the following conclusions can be enumerated:

- The overheads added to YapOr by the new model (i.e., when running with configurations with just one team) are insignificant. Our experiments showed that they range from 1% to 3%.
- Our experiments also show that we are able to increase speedups when we increase the number of machines involved in the computation, thus taking advantage of the totality of cores that are available. This is a clear advantage over YapOr, that was built for running shared memory thus being limited to the cores present in a single machine.
- Although the standard stack splitting approach is also able to take advantage of the cores in more than one machine, our results show that our approach incurs in less overhead, thus being significantly faster.

Our model showed that it is able to seamlessly combine shared and distributed approaches and take advantage of the best of both worlds. In what follows we discuss possible paths of further research regarding other functionalities and improvements to our system.

7.2 Further Work

The implementation of our model has reached its primary goal of being able of taking advantage of the combination of shared memory with distributed memory. Even though, it lacks some important features that may limit its usage in some realistic applications. We hope that these limitations could result in further improvements and further research in this area, such as:

More experimentation. It would be important to test our implementation more intensively and with a wider range of benchmarks so that we have a more clear view on how to tune some scheduling parameters and refine the system as a whole. In addition to that, it would be also important to assess how our system behaves when some network parameters, such as bandwidth and latency, change.

Scheduling strategies. In our system the stack splitting technique is only used to distribute work between teams. It would be interesting to allow workers to share work inside the team also using stack splitting as proposed by Vieira et al. [48]. Furthermore, we could also implement alternative stack splitting splitting strategies, such as diagonal and half splitting [49], for team scheduling of work.

Support for full Prolog. Our system does not support the cut predicate, order sensitive predicates (such as the assert and retract predicates) and side-effects (such as the write predicate). Regarding the cut predicate, at the team level, we already have all the data structures and mechanisms to support it since they were inherited from YapOr. In YapOr when a cut is performed, the tree formed by the or-frames is used to know which workers are on the scope of a given cut. In a distributed system that is not possible since we do not have a full representation of the or-tree and auxiliary mechanisms to maintain a description of each worker would have to be studied and implemented. The order sensitive predicates and side-effects do not prune branches of the or-tree but they also require information about the position of the other workers in the or-tree since they need to be executed in the way they would be in a sequential system. To implement side-effect predicates, it would be also required to support concurrent updates to the internal database.

Support dynamic code compilation. By default Yap generates indexing code dynamically during execution [38]. If we allow Yap to generate dynamic code during

the execution, such behaviour constitutes a problem when sending work from one team to another since the stacks may be pointing to code that was not generated in the receiving team. To bypass this limitation, in our current implementation, we begin by running the program sequentially in the master worker of each team to ensure that all the indexing code is generated and accessible to everyone. Only after that we run the code in parallel. This situation is a clear limitation of our system. Solving this problem is not trivial and it would involve creating an internal database that guarantees that each team may generate its own dynamic code without interfering with the code generated by other teams. When the sharing process occurs this database would also need to be copied. An easier alternative is to disable dynamic indexing code generation, but we have chosen to avoid such alternative.

Support for incremental copy between teams. Incremental copying is a technique that allows to reduce the total amount of memory to be copied during the sharing work operation by avoiding to copy the common parts of the stacks [4]. Incremental copying between workers of the same team is already supported as it was inherited from YapOr and it has showed to have a positive impact in terms of performance. This technique was also implemented in distributed systems such as YapDSS [34] and PALS [49]. In these systems, when sharing work, information about the relative position of the workers in the or-tree is exchanged and based on that information, that the incremental copying algorithm then decides which parts of the stacks must be copied. We think that this approach could be adapted to our model too.

7.3 Final Remark

The research work we have presented in this thesis is based on the work developed by the parallel Prolog community during many years. Unfortunately, in the recent past years, the work in these research field suffered a substantial decline. With this thesis, we hope to bring a new breath to the field.

Appendix A

Results

Table A.1: Execution times in seconds and coefficient of variation for the vertical splitting results presented in Table 6.4

4 workers	Teams(1,4)		Teams(2,2)		Teams(4,1)					
arithmetic(10)	169.015	0.01	127.770	0.00	93.708	0.00				
cubes(10)	16.182	0.00	17.933	0.00	19.745	0.00				
ham(40)	31.791	0.00	33.214	0.00	34.001	0.00				
knight_move(13)	95.698	0.00	106.854	0.00	116.809	0.00				
magic_cube	13.374	0.00	13.761	0.00	13.790	0.00				
map_colouring(46)	44.189	0.00	46.279	0.00	47.537	0.00				
nsort(12)	102.433	0.00	110.379	0.00	110.512	0.00				
puzzle(4)	4.688	0.01	4.744	0.00	4.856	0.02				
queens(14)	137.521	0.00	150.458	0.00	170.737	0.00				
send_more	17.982	0.00	18.546	0.00	18.923	0.00				
8 workers	Teams(1,8)		Teams(2,4)		Teams(4,2)		Teams(8,1)			
arithmetic(10)	111.728	0.00	79.118	0.00	58.271	0.01	48.715	0.01		
cubes(10)	8.130	0.00	8.718	0.00	9.082	0.00	10.329	0.01		
ham(40)	16.050	0.00	16.455	0.00	17.062	0.01	17.277	0.01		
knight_move(13)	48.124	0.00	51.608	0.00	53.582	0.00	58.272	0.00		
magic_cube	6.688	0.00	6.933	0.00	6.951	0.00	7.142	0.01		
map_colouring(46)	22.267	0.00	22.802	0.01	23.031	0.01	24.063	0.01		
nsort(12)	51.213	0.00	55.750	0.00	55.699	0.00	55.721	0.00		
puzzle(4)	2.358	0.01	2.403	0.00	2.432	0.01	2.695	0.04		
queens(14)	68.690	0.00	73.334	0.00	76.864	0.00	82.082	0.00		
send_more	8.961	0.00	9.125	0.00	9.311	0.01	9.724	0.02		
16 workers	Teams(1,16)		Teams(2,8)		Teams(4,4)		Teams(8,2)		Teams(16,1)	
arithmetic(10)	105.305	0.00	59.912	0.01	37.946	0.02	31.086	0.02	27.953	0.03
cubes(10)	4.085	0.00	4.328	0.00	4.471	0.02	4.791	0.02	5.813	0.08
ham(40)	8.102	0.00	8.276	0.00	8.339	0.01	9.613	0.05	9.197	0.04
knight_move(13)	24.174	0.00	25.723	0.00	26.041	0.00	27.301	0.01	29.961	0.01
magic_cube	3.378	0.01	3.540	0.01	3.548	0.01	3.767	0.04	4.896	0.08
map_colouring(46)	11.256	0.01	11.459	0.01	11.596	0.03	11.819	0.01	13.288	0.03
nsort(12)	25.975	0.00	28.183	0.00	28.253	0.00	28.258	0.00	29.007	0.02
puzzle(4)	1.182	0.01	1.236	0.02	1.261	0.02	1.384	0.05	1.913	0.08
queens(14)	34.778	0.00	36.310	0.00	36.893	0.01	38.830	0.01	42.573	0.01
send_more	4.474	0.00	4.607	0.00	4.638	0.01	4.822	0.01	5.562	0.06

Table A.2: Execution time in seconds and coefficient of variation for the horizontal splitting results used in Table 6.4

4 workers	Teams(1,4)		Teams(2,2)		Teams(4,1)					
arithmetic(10)	16.456	0.00	128.114	0.00	93.666	0.00				
cubes(10)	18.096	0.00	19.409	0.00	20.407	0.00				
ham(40)	30.789	0.00	32.209	0.00	32.890	0.00				
knight_move(13)	97.945	0.00	106.063	0.00	112.788	0.00				
magic_cube	12.333	0.01	12.701	0.01	12.984	0.01				
map_colouring(46)	45.600	0.00	47.030	0.00	48.225	0.00				
nsort(12)	96.799	0.00	100.879	0.00	105.522	0.00				
puzzle(4)	4.734	0.01	4.738	0.01	5.009	0.01				
queens(14)	135.580	0.00	149.495	0.00	15.9845	0.00				
send_more	17.564	0.00	18.128	0.01	18.367	0.00				
8 workers	Teams(1,8)		Teams(2,4)		Teams(4,2)		Teams(8,1)			
arithmetic(10)	111.740	0.00	81.777	0.01	59.127	0.01	48.379	0.01		
cubes(10)	9.098	0.00	9.595	0.00	9.848	0.02	10.409	0.01		
ham(40)	15.639	0.00	16.290	0.01	16.296	0.01	16.705	0.01		
knight_move(13)	49.349	0.00	51.924	0.00	53.339	0.00	56.758	0.00		
magic_cube	6.227	0.01	6.326	0.02	6.524	0.01	6.505	0.03		
map_colouring(46)	22.995	0.00	23.386	0.00	23.655	0.01	24.445	0.01		
nsort(12)	49.062	0.00	50.930	0.00	51.879	0.00	52.975	0.00		
puzzle(4)	2.415	0.01	2.494	0.01	2.485	0.01	2.548	0.02		
queens(14)	68.155	0.00	71.666	0.00	74.411	0.00	80.029	0.00		
send_more	8.828	0.00	9.049	0.01	9.224	0.02	9.276	0.02		
16 workers	Teams(1,16)		Teams(2,8)		Teams(4,4)		Teams(8,2)		Teams(16,1)	
arithmetic(10)	101.033	0.00	56.718	0.01	37.588	0.02	30.641	0.03	27.536	0.04
cubes(10)	4.629	0.01	4.858	0.01	4.933	0.01	5.090	0.02	5.890	0.05
ham(40)	7.969	0.00	8.286	0.01	8.284	0.01	8.574	0.02	8.977	0.05
knight_move(13)	24.957	0.01	25.994	0.00	26.475	0.01	27.277	0.00	29.268	0.02
magic_cube	3.176	0.01	3.278	0.01	3.222	0.01	3.301	0.01	3.762	0.03
map_colouring(46)	11.604	0.01	11.994	0.01	11.981	0.02	12.200	0.00	13.484	0.02
nsort(12)	24.946	0.00	25.786	0.01	26.324	0.01	26.412	0.01	27.021	0.02
puzzle(4)	1.242	0.01	1.287	0.03	1.323	0.02	1.460	0.07	1.603	0.08
queens(14)	34.177	0.00	35.496	0.00	36.116	0.00	37.422	0.00	41.353	0.01
send_more	4.520	0.00	4.699	0.02	4.658	0.01	4.738	0.01	5.245	0.02

Table A.3: Execution time in seconds and coefficient of variation for the vertical splitting results presented in Table 6.5

16 workers	Teams(2,8)		Teams(4,4)		Teams(8,2)		Teams(16,1)	
arithmetic(10)	59.293	0.01	44.802	0.02	36.474	0.05	43.471	0.11
cubes(10)	4.724	0.02	5.218	0.04	6.872	0.11	12.720	0.15
ham(40)	8.568	0.02	9.004	0.03	18.672	0.20	17.541	0.09
knight_move(13)	25.759	0.01	26.619	0.01	29.674	0.04	37.631	0.07
magic_cube	3.895	0.01	4.162	0.05	5.207	0.11	9.857	0.13
map_colouring(46)	11.784	0.01	12.291	0.02	14.616	0.04	25.138	0.08
nsort(12)	28.525	0.00	29.218	0.01	31.294	0.04	39.713	0.11
puzzle(4)	1.510	0.05	1.750	0.07	2.384	0.12	5.218	0.12
queens(14)	36.829	0.01	38.550	0.02	46.500	0.07	68.270	0.12
send_more	4.972	0.03	5.353	0.04	7.063	0.11	11.963	0.15

24 workers	Teams(2,12)		Teams(4,6)		Teams(6,4)		Teams(12,2)		Teams(24,1)	
arithmetic(10)	54.328	0.01	34.359	0.02	30.065	0.04	30.917	0.11	43.412	0.14
cubes(10)	3.315	0.05	3.628	0.04	4.053	0.06	5.948	0.14	12.187	0.13
ham(40)	5.831	0.02	6.130	0.04	7.205	0.06	18.248	0.12	17.334	0.11
knight_move(13)	17.260	0.01	17.925	0.02	18.735	0.02	22.486	0.05	31.188	0.08
magic_cube	2.716	0.04	2.894	0.05	3.227	0.09	4.968	0.15	11.395	0.16
map_colouring(46)	7.984	0.02	8.597	0.03	8.886	0.05	12.702	0.04	24.014	0.09
nsort(12)	19.330	0.01	20.038	0.01	20.918	0.03	24.220	0.07	35.269	0.11
puzzle(4)	1.143	0.06	1.349	0.15	1.546	0.10	2.275	0.10	5.029	0.11
queens(14)	25.137	0.01	25.926	0.01	27.518	0.03	40.291	0.11	55.514	0.16
send_more	3.543	0.03	3.659	0.04	4.121	0.06	5.572	0.13	12.042	0.22

32 workers	Teams(2,16)		Teams(4,8)		Teams(8,4)		Teams(16,2)		Teams(32,1)	
arithmetic(10)	53.960	0.01	32.656	0.02	23.358	0.05	26.873	0.08	40.425	0.11
cubes(10)	2.592	0.05	2.862	0.06	3.503	0.12	5.762	0.11	12.032	0.16
ham(40)	4.639	0.03	4.799	0.04	6.032	0.07	17.570	0.13	16.153	0.11
knight_move(13)	13.152	0.02	13.656	0.02	14.754	0.03	19.486	0.08	29.539	0.09
magic_cube	2.165	0.03	2.311	0.08	2.941	0.11	4.680	0.15	10.258	0.22
map_colouring(46)	6.163	0.04	6.508	0.03	7.685	0.06	12.144	0.05	22.682	0.07
nsort(12)	14.617	0.01	15.330	0.02	16.883	0.05	23.140	0.13	35.180	0.11
puzzle(4)	879	0.09	1.150	0.16	1.274	0.07	2.258	0.12	5.160	0.09
queens(14)	19.264	0.02	20.046	0.02	23.023	0.07	35.420	0.10	63.219	0.13
send_more	2.797	0.04	2.921	0.05	3.497	0.09	5.444	0.13	12.783	0.13

Table A.4: Execution time in seconds and coefficient of variation for the horizontal splitting results presented in Table 6.4

16 workers	Teams(2,8)		Teams(4,4)		Teams(8,2)		Teams(16,1)	
arithmetic(10)	59.493	0.01	44.951	0.01	35.444	0.06	46.403	0.12
cubes(10)	5.070	0.03	5.338	0.02	6.469	0.08	10.473	0.14
ham(40)	8.582	0.02	8.841	0.03	10.473	0.06	18.998	0.10
knight_move(13)	26.479	0.01	26.861	0.01	29.743	0.04	38.632	0.07
magic_cube	3.360	0.05	3.681	0.02	3.981	0.03	6.404	0.05
map_colouring(46)	12.144	0.01	12.713	0.01	14.708	0.03	24.934	0.06
nsort(12)	26.251	0.01	27.524	0.02	29.538	0.03	40.981	0.15
puzzle(4)	1.396	0.05	1.764	0.05	2.302	0.09	3.843	0.10
queens(14)	35.723	0.00	36.752	0.01	39.177	0.01	49.190	0.04
send_more	4.988	0.03	5.129	0.03	5.530	0.04	8.335	0.06

24 workers	Teams(2,12)		Teams(4,6)		Teams(6,4)		Teams(12,2)		Teams(24,1)	
arithmetic(10)	55.072	0.02	34.715	0.02	29.868	0.03	30.830	0.07	43.235	0.15
cubes(10)	3.536	0.05	3.834	0.05	4.068	0.04	5.624	0.06	9.881	0.16
ham(40)	5.927	0.03	6.053	0.02	6.655	0.05	8.476	0.07	18.784	0.06
knight_move(13)	17.934	0.02	18.363	0.02	19.306	0.02	23.237	0.08	34.399	0.11
magic_cube	2.285	0.03	2.591	0.03	2.750	0.04	3.461	0.06	6.390	0.04
map_colouring(46)	8.548	0.04	8.809	0.02	9.619	0.03	12.879	0.05	24.523	0.09
nsort(12)	18.122	0.02	18.824	0.01	19.516	0.02	22.674	0.08	41.140	0.14
puzzle(4)	10.41	0.12	1.273	0.05	1.482	0.07	2.145	0.14	36.81	0.19
queens(14)	23.916	0.01	24.567	0.01	25.299	0.01	27.535	0.03	43.349	0.04
send_more	3.699	0.07	3.673	0.04	3.861	0.03	4.695	0.06	8.179	0.05

32 workers	Teams(2,16)		Teams(4,8)		Teams(8,4)		Teams(16,2)		Teams(32,1)	
arithmetic(10)	54.519	0.01	34.065	0.03	23.874	0.04	28.485	0.08	41.096	0.11
cubes(10)	2.936	0.09	3.173	0.03	3.436	0.05	5.345	0.10	9.291	0.13
ham(40)	4.557	0.02	4.621	0.02	5.642	0.08	7.906	0.08	16.366	0.16
knight_move(13)	14.078	0.02	14.049	0.01	15.122	0.02	21.870	0.06	32.237	0.17
magic_cube	1.801	0.07	2.045	0.03	2.252	0.06	3.300	0.05	5.970	0.06
map_colouring(46)	6.941	0.05	6.838	0.03	8.140	0.06	12.510	0.08	24.956	0.05
nsort(12)	14.315	0.04	14.923	0.01	16.222	0.06	22.539	0.17	36.153	0.16
puzzle(4)	827	0.12	989	0.07	1.328	0.09	2.054	0.15	3.472	0.13
queens(14)	18.034	0.01	18.769	0.01	19.729	0.01	23.357	0.04	41.566	0.05
send_more	3.031	0.04	2.935	0.04	3.090	0.03	4.360	0.02	8.466	0.08

Table A.5: Execution times in seconds and coefficient of variation for the vertical splitting results presented in Table 6.7 and Table 6.8

	1 machine		2 machines		4 machines		8 machines	
4 cores	Team(1,4)		Team(2,4)		Team(4,4)		Team(8,4)	
arithmetic(10)	169.015	0.01	88.735	0.01	44.802	0.02	23.363	0.05
cubes(10)	16.182	0.00	9.087	0.01	5.218	0.04	3.543	0.05
ham(40)	31.791	0.00	17.002	0.01	9.004	0.03	6.315	0.11
knight_move(13)	95.698	0.00	51.969	0.00	26.619	0.01	14.975	0.03
magic_cube	13.374	0.00	7.298	0.02	4.162	0.05	2.874	0.08
map_colouring(46)	44.189	0.00	23.102	0.00	12.291	0.02	7.822	0.05
nsort(12)	102.433	0.00	56.291	0.00	29.218	0.01	16.692	0.06
puzzle(4)	4.688	0.01	2.735	0.03	1.750	0.07	1.360	0.08
queens(14)	137.521	0.00	73.595	0.00	38.550	0.02	21.745	0.04
send_more	17.982	0.00	9.546	0.02	5.353	0.04	3.660	0.12
8 cores								
arithmetic(10)	111.728	0.00	59.293	0.01	32.656	0.02		
cubes(10)	8.130	0.00	4.724	0.02	2.862	0.06		
ham(40)	16.050	0.00	8.568	0.02	4.799	0.04		
knight_move(13)	48.124	0.00	25.759	0.01	13.656	0.02		
magic_cube	6.688	0.00	3.895	0.01	2.311	0.08		
map_colouring(46)	22.267	0.00	11.784	0.01	6.508	0.03		
nsort(12)	51.213	0.00	28.525	0.00	15.330	0.02		
puzzle(4)	2.358	0.01	1.510	0.05	1.150	0.16		
queens(14)	68.690	0.00	36.829	0.01	20.046	0.02		
send_more	8.961	0.00	4.972	0.03	2.921	0.05		
16 cores								
arithmetic(10)	105.305	0.00	53.960	0.01				
cubes(10)	4.085	0.00	2.592	0.05				
ham(40)	8.102	0.00	4.639	0.03				
knight_move(13)	24.174	0.00	13.152	0.02				
magic_cube	3.378	0.01	2.165	0.03				
map_colouring(46)	11.256	0.01	6.163	0.04				
nsort(12)	25.975	0.00	14.617	0.01				
puzzle(4)	1.182	0.01	879	0.09				
queens(14)	34.778	0.00	19.264	0.02				
send_more	4.474	0.00	2.797	0.04				

Table A.6: Execution times in seconds and coefficient of variation for the horizontal splitting results presented in Table 6.7 and Table 6.8

	1 machine		2 machines		4 machines		8 machines	
4 cores	Team(1,4)		Team(2,4)		Team(4,4)		Team(8,4)	
arithmetic(10)	164.563	0.00	86.898	0.01	37.588	0.02	23.601	0.03
cubes(10)	18.096	0.00	9.826	0.01	49.33	0.01	3.372	0.07
ham(40)	30.789	0.00	16.509	0.00	8.284	0.01	5.494	0.07
knight_move(13)	97.945	0.00	52.490	0.00	26.475	0.01	14.982	0.04
magic_cube	12.333	0.01	6.501	0.01	3.222	0.01	2.236	0.06
map_colouring(46)	45.600	0.00	23.747	0.00	11.981	0.02	8.144	0.06
nsort(12)	96.799	0.00	51.072	0.01	26.324	0.01	16.183	0.06
puzzle(4)	4.734	0.01	2.610	0.03	1.323	0.02	1.302	0.10
queens(14)	135.580	0.00	71.590	0.00	36.116	0.00	19.570	0.02
send_more	17.564	0.00	9.164	0.01	4.658	0.01	3.038	0.05
8 cores								
arithmetic(10)	111.740	0.00	59.493	0.01	34.065	0.03		
cubes(10)	9.098	0.00	5.070	0.03	3.173	0.03		
ham(40)	15.639	0.00	8.582	0.02	4.621	0.02		
knight_move(13)	49.349	0.00	26.479	0.01	14.049	0.01		
magic_cube	6.227	0.00	3.360	0.05	2.045	0.03		
map_colouring(46)	22.995	0.00	12.144	0.01	6.838	0.03		
nsort(12)	49.062	0.00	26.251	0.01	14.923	0.01		
puzzle(4)	2.415	0.00	1.396	0.06	989	0.07		
queens(14)	68.155	0.00	35.723	0.00	18.769	0.01		
send_more	8.828	0.00	4.988	0.03	2.935	0.04		
16 cores								
arithmetic(10)	101.033	0.00	54.519	0.01				
cubes(10)	4.629	0.01	2.936	0.09				
ham(40)	7.969	0.00	4.557	0.02				
knight_move(13)	24.957	0.01	14.078	0.02				
magic_cube	3.176	0.01	1.801	0.07				
map_colouring(46)	11.604	0.01	6.941	0.05				
nsort(12)	24.946	0.00	14.315	0.04				
puzzle(4)	1.242	0.01	827	0.13				
queens(14)	34.177	0.00	18.034	0.01				
send_more	4.520	0.00	3.031	0.04				

Table A.7: Execution times in seconds and coefficient of variation for the vertical splitting results presented in Table 6.8 using standard stack sppliting

	4 workers		8 workers		16 workers		32 workers	
with simulation	SS(4)		SS(8)		SS(16)		SS(32)	
arithmetic(10)	99.390	0.01	57.424	0.04	43.471	0.11	40.425	0.11
cubes(10)	21.739	0.03	13.680	0.09	12.720	0.15	12.032	0.16
ham(40)	36.611	0.03	23.148	0.04	17.541	0.09	16.153	0.11
knight_move(13)	117.789	0.00	62.633	0.02	37.631	0.07	29.539	0.09
magic_cube	15.755	0.04	10.557	0.14	9.857	0.13	10.258	0.22
map_colouring(46)	49.112	0.01	29.547	0.05	25.138	0.08	22.682	0.07
nsort(12)	113.310	0.01	60.990	0.03	39.713	0.11	35.180	0.11
puzzle(4)	6.013	0.07	5.399	0.12	5.218	0.12	5.160	0.09
queens(14)	168.330	0.01	95.185	0.05	68.270	0.12	63.219	0.13
send_more	20.497	0.03	14.083	0.07	11.963	0.15	12.783	0.13
without simulation								
arithmetic(10)	98.082	0.03	58.600	0.07	40.068	0.08	38.981	0.09
cubes(10)	20.971	0.11	13.867	0.12	11.212	0.21	11.182	0.14
ham(40)	35.599	0.04	20.177	0.04	13.251	0.11	12.214	0.16
knight_move(13)	117.381	0.03	60.770	0.01	36.600	0.08	27.161	0.11
magic_cube	15.240	0.05	9.994	0.04	8.663	0.09	9.291	0.14
map_colouring(46)	48.903	0.03	28.813	0.05	21.856	0.06	20.723	0.05
nsort(12)	113.100	0.02	60.046	0.03	36.947	0.05	32.583	0.11
puzzle(4)	5.778	0.09	4.417	0.09	3.929	0.13	3.906	0.16
queens(14)	168.869	0.03	89.985	0.03	56.135	0.10	53.687	0.13
send_more	20.728	0.09	12.987	0.10	11.906	0.16	10.376	0.20

Table A.8: Execution times in seconds and coefficient of variation for the horizontal splitting results presented in Table 6.8 using standard stack sppliting

	4 workers		8 workers		16 workers		32 workers	
with simulation	SS(4)		SS(8)		SS(16)		SS(32)	
arithmetic(10)	98.885	0.01	56.511	0.04	46.403	0.12	41.096	0.11
cubes(10)	21.467	0.02	13.075	0.06	10.473	0.14	9.291	0.13
ham(40)	35.232	0.02	22.009	0.08	18.998	0.10	16.366	0.16
knight_move(13)	114.568	0.00	61.814	0.04	38.632	0.07	32.237	0.17
magic_cube	13.565	0.02	7.590	0.05	64.04	0.05	5.970	0.06
map_colouring(46)	50.316	0.01	29.707	0.05	24.934	0.06	24.956	0.05
nsort(12)	107.131	0.01	59.610	0.04	40.981	0.15	36.153	0.16
puzzle(4)	5.827	0.07	4.150	0.11	3.843	0.10	3.472	0.13
queens(14)	161.024	0.00	83.973	0.01	49.190	0.04	41.566	0.05
send_more	19.146	0.01	10.765	0.05	8.335	0.06	8.466	0.08
without simulation								
arithmetic(10)	97.957	0.01	56.880	0.05	44.640	0.11	41.581	0.17
cubes(10)	21.262	0.01	12.383	0.03	9.281	0.08	8.476	0.11
ham(40)	34.429	0.01	19.448	0.05	13.646	0.10	14.164	0.11
knight_move(13)	115.053	0.00	60.223	0.02	37.431	0.05	31.687	0.14
magic_cube	13.362	0.02	7.306	0.04	6.020	0.06	5.881	0.05
map_colouring(46)	49.518	0.01	28.309	0.04	23.101	0.06	22.160	0.08
nsort(12)	107.702	0.00	55.916	0.01	35.593	0.10	30.620	0.14
puzzle(4)	5.530	0.06	3.681	0.13	3.060	0.17	3.026	0.17
queens(14)	161.222	0.00	83.590	0.01	48.396	0.03	38.304	0.03
send_more	18.994	0.01	10.502	0.03	7.918	0.05	7.711	0.06

References

[1] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org.

[2] H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. The MIT Press, 1991.

[3] K. Ali and R. Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990.

[4] K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[5] K. Ali, R. Karlsson, and S. Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. *New Generation Computing*, 11(1 & 4):81–103, 1992.

[6] L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming*, 33(1):49–79, 1997.

[7] J. Barklund. *Parallel Unification*. PhD thesis, Uppsala University, 1990.

[8] A. Beaumont, S. Raman, P. Szeredi, and D. H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *Conference on Parallel Architectures and Languages Europe*, number 506 in LNCS, pages 403–420. Springer-Verlag, 1991.

[9] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, pages 45–64. Wiley & Sons, 1992.

[10] A. Calderwood and P. Szeredi. Scheduling Or-parallelism in Aurora – the Manchester Scheduler. In *International Conference on Logic Programming*, pages 419–435. The MIT Press, 1989.

[11] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un Système de Communication Homme–Machine en Francais. Technical report cri 72-18, Groupe Intelligence Artificielle, Université Aix-Marseille II, 1973.

[12] J. S. Conery and D. F. Kibler. Parallel Interpretation of Logic Programs. In *Conference on Functional Programming Languages and Computer Architecture*, pages 163–170. ACM, 1981.

[13] M. Correia, F. Silva, and V. Santos Costa. The SBA: Exploiting Orthogonality in And-Or Parallel Systems. In *International Logic Programming Symposium*, pages 117–131. The MIT Press, 1997.

[14] I. C. Dutra. A Flexible Scheduler for the Andorra-I System. In *ICLP Workshop on Parallel Execution of Logic Programs*, pages 70–82, 1991.

[15] G. Gupta. *Parallel Execution of Logic Programs on Multiprocessor Architectures*. PhD thesis, Department of Computer Science, University of North Carolina, 1991.

[16] G. Gupta, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. Research report, Laboratory for Logic, Databases and Advanced Programming, New Mexico State University, 1997.

[17] G. Gupta, V. Santos Costa, and E. Pontelli. Shared paged binding array: A universal datastructure for parallel logic programming. In *NSF/ICOT Workshop on Parallel Logic Programming*, pages 94–4, 1994.

[18] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions on Programming Languages*, 15(4):659–680, 1993.

[19] G. Gupta and E. Pontelli. Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *International Conference on Logic Programming*, pages 290–304. The MIT Press, 1999.

[20] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[21] G. Gupta, E. Pontelli, M. V. Hermenegildo, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–109. The MIT Press, 1994.

[22] M. V. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3-4):233–257, 1991.

[23] M. V. Hermenegildo and F. Rossi. Strict and Nonstrict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *The Journal of Logic Programming*, 22(1):1 – 45, 1995.

[24] L. V. Kalé. The Reduce-Or Process Model for Parallel Execution of Logic Programs. *The Journal of Logic Programming*, 11(1&2):55–84, 1991.

[25] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing*, pages 569–574. North-Holland, 1974.

[26] R. Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series. North-Holland, 1979.

[27] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[28] 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. In *International Conference on Fifth Generation Computer Systems*, pages 819–830. Institute for New Generation Computer Technology, 1988.

[29] E. Pontelli and G. Gupta. Implementation Mechanisms for Dependent And-Parallelism. In *International Conference on Logic Programming*, pages 123–137. The MIT Press, 1997.

[30] E. Pontelli, G. Gupta, and M. V. Hermenegildo. A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–571. IEEE Computer Society, 1995.

[31] E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. V. Hermenegildo. Improving the Efficiency of Nondeterministic Independent And-Parallel Systems. *Journal of Computer Languages*, 22(2/3):115–142, 1996.

[32] E. Pontelli, K. Villaverde, Hai-Feng Guo, and G. Gupta. Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. *Journal of Parallel and Distributed Computing*, 66(10):1267–1293, 2006.

[33] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[34] R. Rocha, F. Silva, and R. Martins. YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In *Portuguese Conference on Artificial Intelligence*, volume 2902 of *LNAI*, pages 136–150. Springer-Verlag, 2003.

[35] R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Portuguese Conference on Artificial Intelligence*, volume 1695 of *LNAI*, pages 178–192. Springer-Verlag, 1999.

[36] V. Santos Costa. COWL: Copy-On-Write for Logic Programs. In *International Parallel Processing Symposium, Held Jointly with the Symposium on Parallel and Distributed Processing*, pages 720–727. IEEE Computer Society, 1999.

[37] V. Santos Costa, I. Dutra, and R. Rocha. Threads and Or-Parallelism Unified. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 10(4–6):417–432, 2010.

[38] V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

[39] V. Santos Costa, R. Rocha, and F. Silva. Novel Models for Or-Parallel Logic Programs: A Performance Analysis. In *EuroPar 2000 Parallel Processing*, number 1900 in LNCS, pages 744–753. Springer-Verlag, 2000.

[40] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, 1991.

[41] K. Shen. Exploiting Dependent And-parallelism in Prolog: The Dynamic Dependent And-Parallel Scheme (DDAS). In *Joint International Conference and Symposium on Logic Programming*, pages 717–731. The MIT Press, 1992.

[42] K. Shen. *Studies of AND/OR Parallelism in Prolog*. PhD thesis, University of Cambridge, 1992.

[43] F. Silva and P. Watson. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming*, 43(2):173–186, 2000.

[44] R. Sindaha. Branch-Level Scheduling in Aurora: The Dharma Scheduler. In *International Logic Programming Symposium*, pages 403–419. The MIT Press, 1993.

[45] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *North American Conference on Logic Programming*, pages 713–732. The MIT Press, 1989.

[46] Hans Tebra. *Optimistic and-parallelism in Prolog*, volume 259 of *LNCS*, pages 420–431. Springer-Verlag, 1987.

[47] R. Vieira, R. Rocha, and F. Silva. On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores. In *Proceedings of the 12th Colloquium on Implementation of Constraint and LOgic Programming Systems, CICLOPS'2012*, pages 71–85, 2012.

[48] R. Vieira, R. Rocha, and F. Silva. Or-Parallel Prolog Execution on Multicores Based on Stack Splitting. In *International Workshop on Declarative Aspects and Applications of Multicore Programming*. ACM Digital Library, 2012.

[49] K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 27–42. Springer-Verlag, 2001.

[50] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, 1997.

[51] D. H. D. Warren. *Applied Logic – Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977.

[52] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[53] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog – Abstract Design and Implementation Issues. In *International Logic Programming Symposium*, pages 92–102. IEEE Computer Society, 1987.

[54] D. S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *International Logic Programming Symposium*, pages 198–203. IEEE Computer Society, 1984.