Rui Edgar da Silva Vieira

Or-Parallel Prolog Execution on Multicores based on Stack Splitting

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Novembro de 2011

Rui Edgar da Silva Vieira

Or-Parallel Prolog Execution on Multicores based on Stack Splitting

Dissertação submetida à Faculdade de Ciências da Universidade do Porto como parte dos requisitos para a obtenção do grau de Mestre em Engenharia de Redes e Sistemas Informáticos

Orientador: Ricardo Jorge Gomes Lopes da Rocha
Co-orientador: Fernando Manuel Augusto da Silva

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Novembro de 2011

To my family.

Dedicado à minha família.

Acknowledgments

First and foremost, I would like to give my utmost gratitude to my supervisor, Prof. Ricardo Rocha, for the encouragement and support during the research and preparation of this thesis. A professor whose inspiring and discerning personality is always there for you. When faced with certain challenges, he advises and motivates a student with his innovative suggestions. For giving me the chance to experience the world of scientific research for the first time and for offering me the possibility to be included in his research team, I am extremely thankful with nothing to regret.

Secondly, I would like to give my special thanks to my co-supervisor, Prof. Fernando Silva, for his cheerful guidance and full-hearted support.

I am also grateful to the LEAP - Logic Environment with Advanced Parallelism project (PTDC/EIA-CCO/112158/2009), for supporting me with a research grant during my research work and, for the possibility to participate in the scientific community.

To my work colleagues Miguel Areias, João Raimundo, João Santos and Tiago Gomes for the great fellowship and availability support during the research of this thesis.

Finally, I would like to dedicate my final acknowledgment to my family for their particular way of fondness and support. To my parents, Fernando and Carminda, for all the love you always indulged me. To my grandmother, Florinda, for being the best grandmother in the world. To my dog Boris. Thank you all for making me the person who firmly stands in front of you.

Rui Vieira

September 2011

Abstract

Prolog is a popular logic programming language that provides a declarative approach to programming, being thus highly amenable for implicit parallelism. There are many efficient sequential implementations of Prolog, mostly based on the Warren Abstract Machine (WAM). Prolog is currently much used by machine learning and natural language practitioners, but its applicability is much wider in scope.

Implicit parallel implementations of Prolog have been proposed in the past. The Muse and YapOr systems are arguably two of the most efficient systems for shared memory architectures, both based on the environment copying model. Stack splitting emerged as an alternative model specially geared to distributed shared memory architectures as it basically splits the computation in such a way that no further, or just minimal, synchronization is required.

With the new multicore architectures, it just makes sense to recover the body of knowledge there is in this area and either devise newer computational models that fit best recent parallel architectures, or to reengineer prior computational models to evaluate their performance on newer architectures. Here, we take the second path.

In this thesis, we focus on the design and implementation of the stack splitting strategy in the YapOr system. Our aim is to take advantage of its robustness to efficiently implement stack splitting support using shared memory, and then be able to directly compare the YapOr based on environment copying with the YapOr based on stack splitting. We devised two splitting schemes, the *vertical splitting* and the *half splitting*, and have adapted data structures, scheduling and incremental copying procedures in YapOr to cope with the new schemes. Finally, we evaluate their performance on a set of known benchmarks on a multicore machine with up to 24 cores. Our initial results confirm that YapOr with the stack splitting schemes is, in general, comparable to YapOr with environment copying, obtaining in some cases better performance than with environment copying.

Resumo

O Prolog é uma popular linguagem de programação lógica, na sua essência declarativa e, por isso, muito adequada à exploração de paralelismo implícito. Actualmente, existem já várias implementações sequenciais de Prolog bastante eficientes, na sua grande maioria baseadas na Warren Abstract Machine (WAM). O Prolog é presentemente muito usado pela comunidade nas áreas de aprendizagem automática e processamento de língua natural, mas o seu escopo de aplicação é maior.

Num passado recente, foram várias as implementações que exploraram paralelismo implícito em Prolog. Em particular, os sistemas Muse e YapOr são dois dos mais eficientes sistemas paralelos para arquitecturas de memória partilhada, ambos baseados no modelo de cópia de ambientes. O modelo de divisão da pilha de pontos de escolha (*stack splitting*) surgiu como um modelo alternativo especialmente adequado para arquitecturas de memória distribuída dado que, na sua essência, determina a divisão da computação em duas partes independentes que praticamente não requerem qualquer sincronização adicional.

Com o desenvolvimento das novas arquitecturas *multicore*, faz todo o sentido recuperar o conhecimento e experiência existente nesta área e, ou desenvolver novos modelos computacionais melhor adequados a estas novas arquitecturas, ou refazer modelos computacionais existentes, devidamente adaptados, e avaliar o seu desempenho nas novas arquitecturas. Aqui, tomamos o segundo caminho.

Esta dissertação foca o desenho e implementação da estratégia de *stack splitting* no sistema YapOr. O nosso objectivo é implementar o *stack splitting* em memória partilhada, tirando partido da implementação robusta e consistente existente, de modo a podermos comparar o desempenho das duas estratégias, *stack splitting* e cópia de ambientes, usando o YapOr como sistema base. Desenvolvemos e implementamos dois esquemas de divisão de trabalho, *vertical splitting* e *half splitting*, adaptamos estruturas de dados, estratégia de scheduling e de cópia incremental no YapOr para

podermos lidar com estes novos esquemas. Finalmente, avaliamos o seu desempenho num conjunto de programas de teste numa máquina com 24 *cores*. Os resultados iniciais confirmam que o YapOr com as estratégias de *stack splitting* é comparável ao YapOr com cópia de ambientes, obtendo em alguns casos melhor desempenho do que com a cópia de ambientes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Declarative languages, as is the case with logic programming languages, have the advantage in which programmers do not have to explicitly specify the control of program execution. Logic programming programs are characterized to be easily described and, one of the most popular and high-level logic programming languages is Prolog.

Noticeable, most of the Prolog's supremacy in logic programming languages was due to its implementation efficiency. In fact, part of that efficiency was made possible first by the implementation of a very efficient sequential machine called *Warren's Abstract Machine* (WAM) [1]. It enabled many specialized optimizations that made Prolog to rise in its application range of real world problems in areas such as Natural Language Processing, Machine Learning, Database Management, Automated Theorem Proving, Expert Systems, Automated Answering Systems, Games and Ontologies.

Prolog programs, whose semantics is based on First Order Logic, naturally exhibit implicit parallelism. One important source of parallelism arises from the simultaneous evaluation of a Prolog goal against all the predicate clauses that match that goal. This form of parallelism is called Or-parallelism. The advantage of implicit parallelism is that one can develop specialized runtime systems to transparently explore the available parallelism in Prolog programs, thus freeing the programmers from having to do it explicitly. This idea is implemented in the YapOr system [16], a Prolog system that exploit or-parallelism.

One main difficulty in the implementation of any parallel system is to devise an efficient strategy to assign computing tasks to idle workers awaiting for work. A parallel Prolog system is no exception as the parallelism it exhibits is highly irregular. Achieving the necessary cooperation, synchronization and concurrent access to shared

data structures among several workers during their execution is a difficult task. The stack splitting strategy [8] provides a simple, clean and efficient method to accomplish work sharing among workers. It successfully splits the computation work of one worker in two fully independent tasks, and thus was first introduced aiming at distributed memory parallel architectures where synchronization and communication have higher costs.

Recent advances in parallel architectures have made our personal computers parallel with multiple cores sharing the main memory. The multicores and clusters of multicores are now the norm, and exploiting implicit parallelism in a transparent way is a quite relevant research direction to take. Although many parallel Prolog systems have been developed in the past, namely Aurora [12], Muse [3], YapOr [16], evaluating their performance or even the implementation of newer computational models specialized for the multicores is still open to further research. In this thesis, we aim to design and implement in YapOr the stack splitting strategy and make it efficient on multicore architectures.

1.1 Thesis Purpose

In this thesis, we aim to study, design and implement efficient work sharing models for or-parallel execution of Prolog programs. Our approach is to benefit from prior research in our group on the development of the YapOr system and extend it to support stack splitting using a shared memory programming model. YapOr is based on the environment copying model, and, thus, changes or extensions to the current data structures, scheduling procedures and the incremental copy mechanism are necessary. The work developed allowed us to make some contributions, not only on the implementation of stack splitting, but also on the creation of new mechanisms, or simply the update of others, particularly in connection to the incremental copy technique. A list of the contributions made are as follows:

- The design and implementation of two stack splitting strategies within the YapOr system, the vertical and half stack splitting both running with or without support for the incremental copy technique. In particular, this included the following new functionalities:
  - The calculation of the first choice point to be assigned as the top choice point for the requesting worker, in the work sharing procedure between two

workers.

- The update of YapOr's bitmap membership mechanism during the work sharing procedure.
- The update of YapOr's copy synchronization mechanism.
- A dereference phase that denotes a worker's consistency when intended to be in a certain computational state with an assigned choice point. This occurs after the copy phase.
- The mechanism that ensures the consistency of the choice point value of the requesting worker's execution time, during the work sharing procedure.

- Performance comparison, with 1, 2, 4, 8, 16 and 24 cores, between YapOr with environment copying and YapOr with vertical and half stack splitting strategies.

1.2 Thesis Outline

The following list briefly describes each chapter in this thesis.

Chapter 1: Introduction. The current chapter.

Chapter 2: Logic Programming and Parallelism. Introduces the basic concepts on the foundations of Logic Programming and the Prolog language. Then, we introduce the opportunities for parallelism in Prolog and overview strategies for work sharing, namely the stack splitting strategies.

Chapter 3: The YapOr Engine. Presents the basic concepts and strategies that support the design of the YapOr parallel system. We also detail some execution models, such as the Environment Copying, and some optimization techniques, such as the Incremental Copy. Emphasis is given to the work sharing procedure and to its behavior.

Chapter 4: Supporting Stack Splitting in YapOr. Introduces the stack splitting schemes that are being implemented in YapOr, namely the vertical splitting and the half splitting. These schemes originate a different behavior than the original YapOr and might lead to different execution results in a certain set of Prolog programs.

Chapter 5: Supporting Stack Splitting with Incremental Copy. Details the most important implementation aspects of the incremental copy technique. Such technique is only acquired when certain conditions are satisfied. We also detail the necessary mechanisms that are crucial for achieving such conditions, and, thus, to support the stack splitting approach.

Chapter 6: Performance Analysis. In this chapter we analyze the advantages and weaknesses of each stack splitting scheme when compared to the environment copying scheme. Several benchmark programs with different characteristics are used in order to assess performance. A comparison is made between the original YapOr with environment copying and the YapOr with the stack splitting schemes implementation.

Chapter 7: Conclusions and Further Work. This chapter summarizes the discussion of the topics presented in this thesis and provides some conclusions on the accomplishments achieved. The conclusions not only refer to the performance comparison between the different versions of YapOr, but also advances implementation improvements that can be made to the implemented schemes. Future work is highlighted and discussed, along with the thesis final remark.

Chapter 2

Logic Programming and Parallelism

This chapter gives an overview of the most important aspects related to the work presented in this thesis. In a self contained manner, first it introduces the basic concepts of Logic Programming and the Prolog language and then it focus on the main topic of this thesis: Parallelism in Logic Programming.

2.1 Logic Programming

Logic Programming languages are supported by a strong mathematical basis and are considered to be very *high-level* and *declarative* languages, properties that induce a simpler programming model. Declarative languages allow the programmer to concentrate on describing *what* the problem is, rather than describing *how* it is computed, leaving resolution and control mostly to the computer.

Following the seminal work by Robinson in 1965 [13], Logic Programming started to emerge as a computing language. Kowalski [10] gave relevance to its procedural aspects, stating that an efficient inference procedural could be automated by applying Robinson's *Resolution Principle*. Logic Programming is based on a subset of the first-order predicate logic - the Horn Clause logic. The Horn Clause logic provides a basis for more powerful programming methods and is ideally suited to non-numerical applications, as it simplifies the programming level difficulty when processing natural language [11].

Karlsson claims that Logic Programming languages, such as Prolog, have the following features [9]:

- Variables are *logical* variables that can be instantiated *only once*.
- Variables are *untyped* until instantiated.
- Variables are instantiated via *unification*, a comparative operation that finds the most general common instance of two data objects.
- At unification failure, the execution *backtracks* and tries an alternative in the last choices set in order to satisfy the original query.

In [4], Carlsson mentions that some of the advantages of Logic Programming are:

- **Simple declarative semantics**. A logic program is simply a collection of predicate logic clauses.
- **Simple procedural semantics**. A logic program can be read as a collection of recursive procedures. In Prolog, for instance, clauses are tried in the order they are written and goals within a clause are executed from left to right.
- **High expressive power**. Logic programs can be seen as executable specifications that despite their simple procedural semantics allow for designing complex and efficient algorithms.
- **Inherent non-determinism**. Since in general several clauses can match a goal, problems involving search are easily programmed in this kind of languages.

These advantages establish an easy way to program, manipulate and understand a predominant compact code, which facilitates program transformations and allows more transparency in parallelism.

2.1.1 Logic Programs

A logic program consists of a collection of clauses. Using Prolog's nomenclature, a clause can be a *fact*, i.e., an assertion with no conditions. A fact is the simplest clause, it is represented as

$$A.$$

and, it can be read as "A is true". A clause can also be defined as a *rule* of the form

$$A \ :- \ B_1, \ ..., \ B_n.$$

where A is the head of the rule, $B_1, \ ..., \ B_n$ are the body *literals* ($n > 0$), and there is a logical implication by the conditional symbol ":-" denoted by:

$$A \ \leftarrow \ B_1, \ ..., \ B_n$$
$$or$$
$$\forall \ (\ B_1 \ \wedge \ ... \ \wedge \ B_n \ \rightarrow \ A)$$
$$or$$
$$\forall \ (\ \neg B_1 \ \vee \ ... \ \vee \ \neg B_n \ \vee \ A)$$

meaning that "If $B_1, \ ..., \ B_n$ are true, then A is true as well" as the *Resolution Principle* asserts.

Positive literals refer to the head of the clause and negative literals correspond to the body of the clause. In logic programs, a clause is allowed to have only one positive literal. The negative literals in the clause's body are also called *subgoals*.

A different kind of clauses is when the head of the clause are none existent. In this case, the clauses are denominated as *queries*. A query is a goal or a conjunction of goals of the form

$$:- \ G_1, \ ..., \ G_n.$$

A *goal* is a term that may be defined as an *atom*, a *variable* or a *functor/compound term*. Variables are undefined terms and atoms are symbolic constants with zero arity. Consider a functor term of the form

$$f \ (\ t_1, \ ..., \ t_n \ )$$

f/n is the functor name with *arity n*. The t_i's are the n argument *terms* and if *f/n* has zero arity ($n = 0$) it is denoted as a simple *atom*.

The computation process is based on the submission of *query clauses*. Using preloaded *program clauses*, which is the knowledge of the interpreter, the process of *resolution* consists on the *unification* of the submitted arguments. Introduced by Kowalsky [11], the process of *resolution* is simply a derivation of an inference system method denominated by the name of *Selective Linear Definite* (SLD) inference rule. In this process and in a procedural perspective, the clauses are selected in the order they appear in the program and the subgoals in a clause are selected from left to right. Solving a query thus involves the selection of a clause whose head matches the query goal. If their arguments unification succeeds, the goals of the matching clause are then executed from left-to-right. When an unification in these goals fails, the system backtracks to the last selection undoing all bindings made to that point, and another alternative is selected. If all goals succeed, a query solution is found.

An *unification* (or *substitution*) consists of finding the *most general unifcation* of variables that makes two expressions identical. This process is composed by a finite set of *bindings* among different variables or among variables to non-variable terms, where each binding must be unique. Having defined a program p and submitting a query $?-q$, the resolution either fails with p and q not unifying, or succeeds accordingly to the bindings of the variables of q with p. When a variable Y is bound to an atom a and to an atom b at the same time ($Y = a \wedge Y = b$) then there is no unification. See, next, a succeeded and a failed unification example:

$$p(a, f(a), Y).$$

$?-p(X, f(Y), a).$	$?-p(X, f(Y), b).$
$X = Y = a$	no $(X = a \wedge Y = a \wedge Y = b)$

The *backtracking* mechanism is a search algorithm for finding all possible solutions for a query that dynamically visits all alternative branches in the search tree. The act of *backtracking* simply consists in restoring the computation up to the previous node and trigger the next alternative from the unexplored alternative's set. General logic programs follow an execution model consisting on traversing search trees in a *depth-first left-to-right* form, as shown in Figure 2.1.

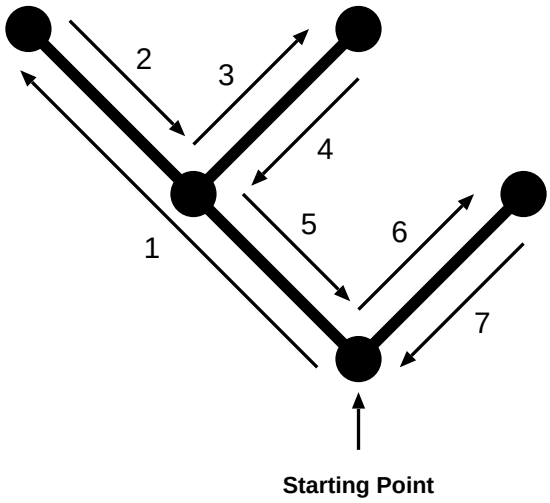

Figure 2.1: Depth-first search tree with backtracking.

When a leaf of the tree is reached, backtracking takes place. The process terminates when no more backtracking is possible, that is, when all sub-trees were traversed [17]. A leaf node can represent a solution or a failure. The *inner* nodes represent the choice points for a matching predicate, while the alternative branches illustrate the clauses being explored.

2.1.2 The Prolog Language

Starting from Robinson's work [13] that described the well known inference rule, *Resolution with Unification*, Prolog was initially intended as the programing language developed by Alain Colmerauer and Phillipe Roussel in 1972, and later, as the Logic Programming in general perspective. These were the times that Kowalski, Russel, Colmerauer and its team discovered that "*computation could be subsumed by deduction*" [11]. Curiously, the name Prolog was chosen by Phillipe Roussel's wife Jacqueline as derives from the abbreviation of the words *PROgramation en LOGique*.

In 1977, David H. Warren made Prolog a viable language by developing the first compiler [18]. In 1983, Warren proposed a new abstract machine to execute Prolog code, with the name *Warren Abstract Machine* (WAM) [19], which became the standard for efficient implementation of most Prolog systems.

To make Prolog a suitable programming language it was necessary to introduce some features that were not present in First Order Logic. These features include:

- **Cut predicate.** This predicate adds a limited form of control to the execution by pruning unexploited alternatives from the computation.

- **Meta-logical predicates.** These predicates inquire the state of the computation and manipulate terms.

- **Extra-logical predicates.** These are predicates which have no logical meaning at all. They perform input/output operations and manipulate the Prolog database, by adding or removing clauses from the program being executed.

- **Other predicates.** These include arithmetic predicates, term comparison predicates, extra control predicates, and a set of predicates that outputs the complete set of answers for a given query.

2.1.3 The Warren Abstract Machine

The Warren's legacy is an abstract machine consisting of a memory architecture and instruction set tailored to Prolog [1]. The WAM is the basis of most of the current Prolog compilers and the source for further sequential and parallel Prolog implementations. We next discuss the WAM's memory architecture, registers, instructions and code translation.

2.1.3.1 Memory Architecture and Registers

As shown in Figure 2.2, the WAM is composed by a set of stacks and a set of registers. The WAM's memory organization is divided into seven logical data structures: one stack for the code area (Code); two stacks for data objects (Heap and Stack); one stack for trailing the modifications along execution in order to allow backtracking (Trail); one stack for the unification process (PDL); one stack for the tangle of symbols; and one array used for temporary argument and registers allocation. In more detail:

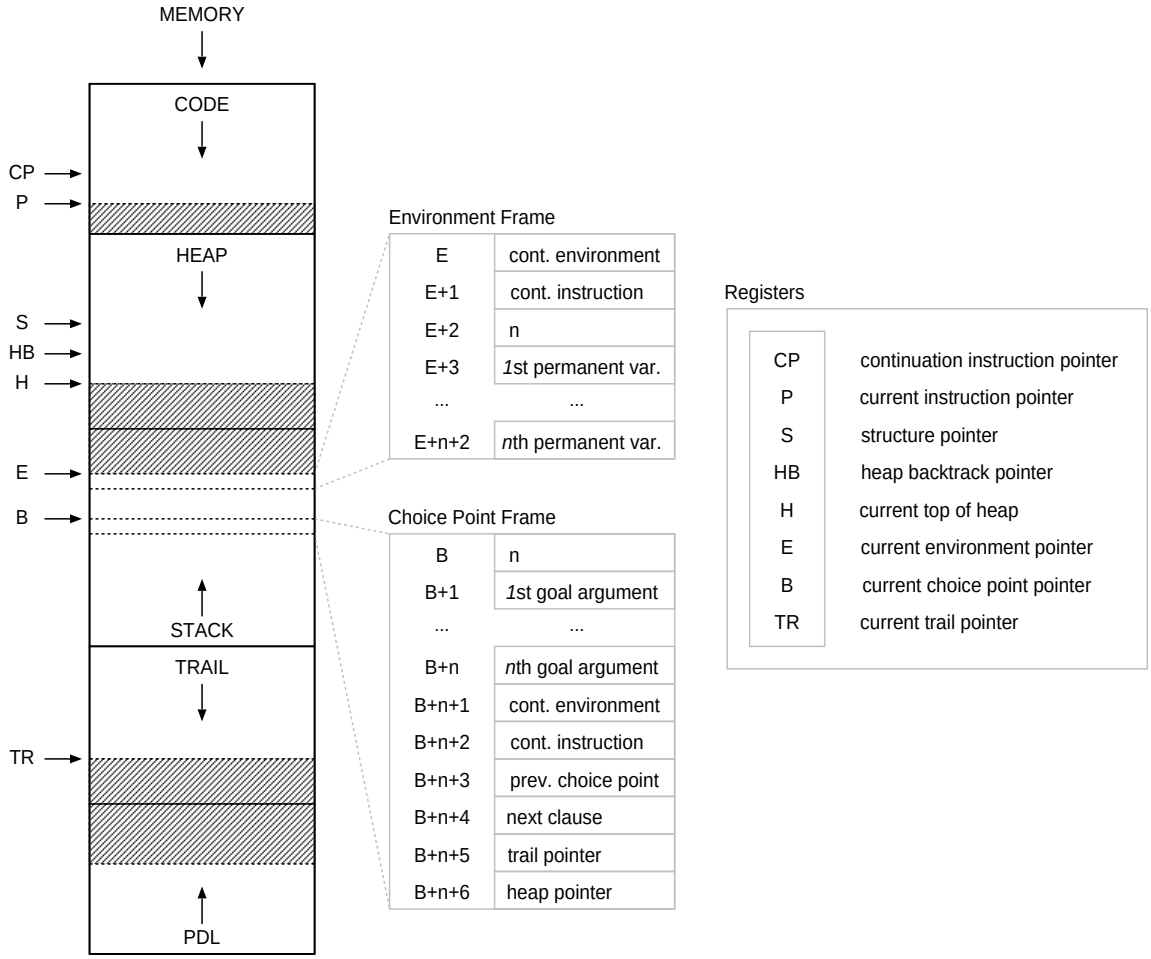

Figure 2.2: WAM memory layout and registers.

Code area is the memory section that contains the compiled WAM instructions of the loaded programs and two instruction pointers, one pointing to the current instruction being executed (P) and another pointing to the continuation instruction upon successful execution of the subgoal at hand (CP).

Heap (or **Global stack**) is an array of data cells and its main function is to store the data terms. The H register points to the current top of the heap.

Stack (or **Local stack**) is used to store the execution *environments* and the *choice points* appearing in the search tree:

- Environments are used to store previous execution states and they are important to allow the computation to safely return to the previous environment in the tree, after evaluating a subgoal. Each *environment frame* is composed by the previous environment frame address (`E`), accessed when the current environment

is deallocated; the address of the instruction to execute upon successful return in a clause invocation; and the number n of permanent variables followed by the sequence of the n permanent variables.

- Choice points are used for stacking up the unexplored alternatives corresponding to predicates with more than an alternative (or clause), by saving the corresponding computational state in such a way that, after backtracking from an alternative execution, it is possible to recover the state at the time the choice point was created. A choice point is stacked up when a predicate is invoked, and freed at the time of the last choice point alternative executes. This structure stores all the key registers needed for the job. A choice point has information about the number of arguments and the arguments themselves for the current predicate; the environment of the previous choice point; the continuation instruction; the previous choice point; the next alternative clause, the current Trail pointer and the current Heap pointer.

Trail stores the addresses of all the variables which were bound during the current execution branch. In order to recover the computational state, when backtracking occurs, the variables on the trail are restored to its previous state and marked as free variables. The TR register points to the current top of the Trail.

PDL (or **Push Down List**) is used as an unification stack with the operations *empty*, *pop* and *push*.

Finally, the registers are used to control the WAM's execution flow. Besides the already mentioned registers, there are some other useful WAM registers. The `HB` register saves the value of `H` of the most recent choice point. The `S` register is used in the unification process to point to the argument being unified and is incremented when processing the next argument.

2.1.3.2 Instructions and Code Translation

Prolog's compilation process is considered fairly simple. For a predicate p/n, each clause follows certain specifications to generate the corresponding WAM code. In WAM, there are four main instruction groups:

- **Choice Point Instructions**

  try_me_else L
  retry_me_else L
  trust_me

  This kind of instructions is intended for handling the execution order between the clauses defining a predicate. The number of clauses will be the number of choice point instructions and, since Prolog follows an SLD resolution scheme, the writing order of the choice point instructions must look like this:

  – For a two clause definition, the pattern is:

  $$\begin{array}{rcl} p/n & : & \text{try\_me\_else } L_1 \\ & & \textit{code for 1st clause} \\ L_1 & : & \text{trust\_me} \\ & & \textit{code for 2nd clause} \end{array}$$

  – For more than two clauses, the pattern is:

  $$\begin{array}{rcl} p/n & : & \text{try\_me\_else } L_1 \\ & & \textit{code for 1st clause} \\ L_1 & : & \text{retry\_me\_else } L_2 \\ & & \textit{code for 2nd clause} \\ & & . \\ L_{n-1} & : & \text{trust\_me} \\ & & \textit{code for last clause} \end{array}$$

- **Control Instructions**

  call p/n
  proceed
  allocate N
  deallocate

  When calling a clause, it's necessary to save the environment in order to return to it when the calling clause succeeds. The `call` and `proceed` instructions serve as a procedure broker and return and the `allocate` and `deallocate` instructions are defined to store and remove environments, respectively.

- **Unification Instructions**

put_structure $f/n, Xi$	get_structure $f/n, Xi$
set_variable Xi	unify_variable Xi
set_value Xi	unify_value Xi
put_structure $f/n, Ai$	get_structure $f/n, Ai$
put_variable Xn, Ai	get_variable Xn, Ai
put_value Xn, Ai	get_value Xn, Ai

There are two different kinds of unification instructions. On the left side are the *query instructions* and on the right side are the *program instructions*. While the query instructions are used on the body's clause of a rule translation, the program instructions translates pretty much everything else of the clause's head. The `put_` and `set_` query instructions are used to associate the argument registers to the subgoals of the clause. The `get_` and `unify_` program instructions are used for head unification with registers and structure arguments, respectively.

- **Indexing Instructions**

switch_on_term V, C, L, S
switch_on_constant N, T
switch_on_structure N, T
try L
retry L
trust L

The indexing instructions are not mandatory, they are simply used to avoid the predicate clause's not unifying with a given subgoal call, ie., the idea is to determine beforehand which clauses unifies with a given predicate by selecting those that are compatible with certain filtering rules. On first argument indexing [1], the filtering method is guided by the first argument type of the invoked predicate. The `switch_on_term` instruction serves as a conditional jumper accordingly to the type of the term, which can be, respectively, a variable, a constant, a non-empty list or a structure. The `switch_on_constant` and `switch_on_structure` second level indexing instructions are used to matching clauses terms, respectively a constant term and a structure term. Identical to the choice point sequence instructions referred before, the `try/retry/trust` third level indexing instruction jump to a label L and, in the `try/retry` cases, they save the next instruction in sequence as the next alternative for the choice point.

2.2 Parallelism in Logic Programming

Considering the popularity and efficiency of the WAM, traditional Prolog implementations have started to grasp the interest on the idea of extending the existent implementation models based on the WAM to a parallel model.

Exploring parallelism in a given paradigm, leads to solving two fundamental problems:

Parallelism recognition in a given program. This problem can be solved by the compilers capacity of exploiting concurrent execution code, distinguish it from sequential code (Implicit Parallelism). The other way is by direct intervention of the programmer, which allows the use of explicit instructions on the program code, and therefore in a certain way, switches the execution state from sequential to parallel or vice versa (Explicit Parallelism).

Efficient work distribution among a set of process units. It is important to define the best way of serving the several *workers* in a multiprocessor system. The idea is to provide a work distribution scheme, or scheduling strategy, that successfully balances the work load among all cooperating workers and minimizes worker idle time.

Prolog programs naturally exhibit several forms of parallelism, namely:

Or-parallelism allows the clauses of a predicate to be explored in parallel. This happens when a predicate is defined by several clauses and when a subgoal call unifies with the head of more than one clause.

$$\begin{aligned} \mathbf{a(X,Y)} &:- b(X),\ c(Y). \\ \mathbf{a(X,Y)} &:- d(X,Y),\ e(Y). \\ \mathbf{a(X,Y)} &:- f(X,Z),\ g(Z,Y). \end{aligned}$$

And-parallelism allows the goals of a clause body to be executed in parallel.

$$a(X,Y) :- \mathbf{b(X)},\ \mathbf{c(Y)}.$$

In this case, it is obvious the convenience of having more than one goal in a clause's body. The parallel execution of these goals raises two kinds of And-parallelism [6]:

- **Independent and-parallelism** arises when the goals do not share variable bindings, ie., the intersection among sets of the accessible variables in each goal is empty meaning that, each goal has no influence on the outcome of the other goals and do not share any unbound variable.

$$a(X,Y) \ :- \ b(\mathbf{X}),\ c(\mathbf{Y}).$$

- **Dependent and-parallelism** arises when the goals have common variables. One goal will create the bind between the common variables, which in a parallel approach might cause *racing conditions* or trigger an incompatible binding. In this type of And-parallelism, there are two ways of dealing with this: **(i)** the goals can be executed simultaneously till the end and then check for compatibility by comparing the variables assignments of the produced bindings. **(ii)** the goals can be executed simultaneously until one of them instantiates a common variable. The worker that claims the binding is called the *producer* and, the others are considered to be *consumers* of that common variable, by simply reading the binding as an input argument.

$$\begin{aligned} a(X,Y) \ &:- \ d(X,\mathbf{Y}),\ e(\mathbf{Y}). \\ a(X,Y) \ &:- \ f(X,\mathbf{Z}),\ g(\mathbf{Z},Y),\ h(\mathbf{Z}). \end{aligned}$$

Unification parallelism happens during the process of unification of the arguments of one's body subgoal with one's clause head. The various arguments can be unified in parallel, as well as the different subterms in a term. This form of parallelism is considered to be of low granularity and has not been one of the most relevant research topics on parallelism in Prolog.

2.2.1 Or-Parallelism

At a first step, Or-parallelism seems much more simpler to be implemented efficiently than And-parallelism. The major advantages motivating Or-parallelism's use are [12]:

Generality. This kind of parallelism is relatively easy to exploit and does not restricts any of the logic programming assets. An important advantage it brings is the fact of being able to retrieve all solutions to a query.

Simplicity. There is no need for any extra programmer annotations, neither any complex real-time analysis by the Prolog compiler.

Closeness to Prolog. It is possible to easily exploit Or-parallelism by extending the sequential execution of Prolog. The language semantics are preserved and one can take advantage of the existing implementations of Prolog in order to overpower max performance.

Granularity. This refers to the amount of work that can be run concurrently, ie., without the intervention of other work pieces that are being processed simultaneous. Or-parallelism has potential for *high granularity* Prolog programs with *high grain size* of concurrent computations.

Applications. Or-parallelism encompasses a wide variety of applications, especially in the area of Artificial Intelligence, in natural language processing, proving theorems or answering queries in a database.

Despite the theoretical simplicity, the implementation of Or-parallelism is difficult due to practical complications that emerge from the sharing of nodes in the Or-parallel search tree. An important difference from a sequential implementation is that an Or-parallel implementation must support *multiple bindings* for the same variable [7]. Given two nodes in two different branches of an Or-tree, all nodes above the least common ancestor of these nodes (including it) are shared between the two branches. A variable created in one of the ancestors nodes might thus be bounded differently in these two branches. This invites for a new environment organization that guarantees the correctness of the bindings in both branches [5].

A major problem in Or-parallelism implementation is thus the efficient representation of the multiple environments coexisting simultaneous during program execution. The act of accessing or binding a variable has a management cost. Since it has been proven impossible to avoid both costs simultaneously [7], they can be minimized. A way of doing this is by designing a scheduler, which equally distributes the unexplored alternatives, of the search tree, among the set of running workers.

2.2.2 Or-Parallel Execution Models

Many or-parallel execution models have been proposed [6], but two have emerged as references for future implementations. One is the *Binding Arrays*, introduced in the Aurora system [4], and the other is the *Environment Copying*, introduced in the Muse system [9]. Both systems are implemented on a SMM (*Shared Memory Multiprocessors*) architecture where all the resources can be accessed by all processing

units in a shared memory environment, ie., if there is a change in a memory location, this change is visible on all processing units.

2.2.2.1 Binding Arrays

In the *Binding Arrays* model, each *worker* carries an auxiliary data structure denominated by *binding array* containing a logical set of conditional variables. These variables are all numerated upon creation of a choice point. When a conditional variable is instantiated, its value is stored in the *binding array*, and the address and the conditional assignment are stored in the Trail. When accessing a variable assignment, the variable label is used in order to index the *binding array* and retrieve the assignment. Nevertheless, the model adds some computation overhead since every variable is cached in the *binding array*.

Every worker has an entry in the *binding array* for the same variable. The operation of work sharing takes place by updating the *binding array*. When a worker changes its branch, for consistency reasons, it first updates its *binding array* by uninstalling the stored assignments in the Trail of the old branch of the worker and then installs the new information concerning the new worker's position. This process of uninstallation/installation entails an overhead which is difficult to minimize.

2.2.2.2 Environment Copying

In the *Environment Copying* model, each worker keeps a separated copy of its own environment, in which it can freely store assignments without making conflicts. Unlike in the *Binding Arrays* model, the unconditional assignments are not shared. Every time a worker shares work with another worker, it copies all the execution stacks from the sharing worker in order to be in the same computational state as the sharing worker. This stack copying is realized through a mechanism called *Incremental Copy*. This kind of copy takes advantage of the fact that the requesting worker has already traversed part of the path between the root node and the youngest common node of both workers. Therefore, it does not need to copy the stacks referring to the whole path from root, but only the stacks starting from the youngest common node of both agents.

2.2.2.3 Stack Splitting

Both Binding Arrays and Environment Copying models need mechanisms that employ mutual exclusion and ensure synchronization when accessing shared branches of the search tree. An alternative strategy to this problem boils down by previously *dividing the available unexplored alternatives* (undone work) by the sharing workers. The term used for this strategy is *Stack Splitting*.

Stack Splitting has two main types of distributing undone work: *Vertical Splitting* and *Horizontal Splitting* [8]. *Vertical Splitting* is based on the alternately division of choice points between the two involved agents P an Q, in the work sharing process.

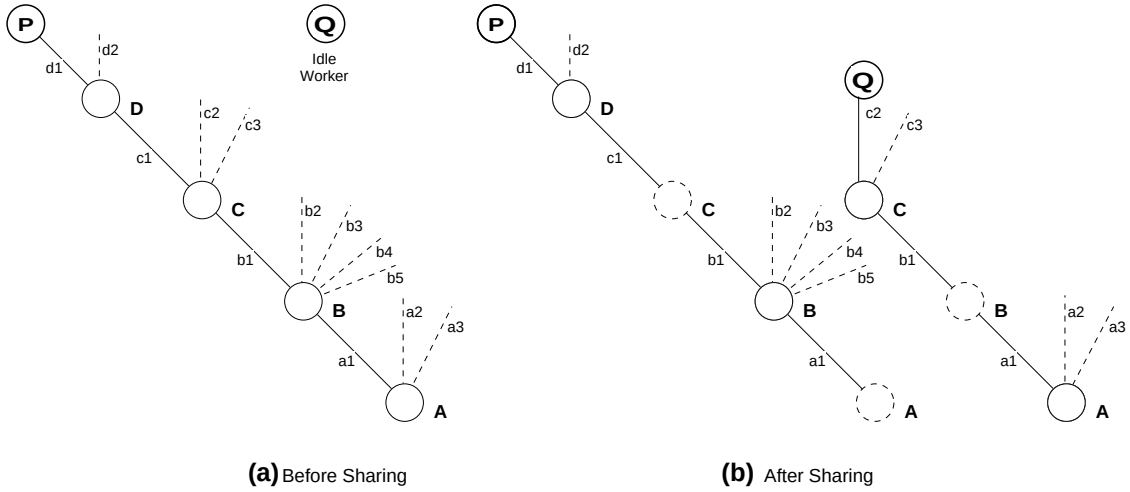

Figure 2.3: *Vertical Splitting*.

As illustrated on Figure 2.3, the process begins with an idle worker Q asking for work to a busy worker P. Accordingly to the available work in P's branch, P's choice points are alternately split between P and Q. After applying the division process, the stacks are copied to Q and Q is set to begin its own execution.

In the *Horizontal Splitting* strategy, what is alternately divided is the unexplored alternatives in each choice point. This strategy has the same divide-copy procedure as *Vertical Splitting*. The split is shown as follows in Figure 2.4.

Diagonal Splitting is an alternative strategy that was idealized as a variation of both *Vertical Splitting* and *Horizontal Splitting*, with the objective of extracting the advantages of both and minimize the disadvantages of each [15]. This strategy adds a better overall balance distribution (of tree's distribution) because it is based on the alternated division of *all unexplored alternatives*, regardless of the choice points

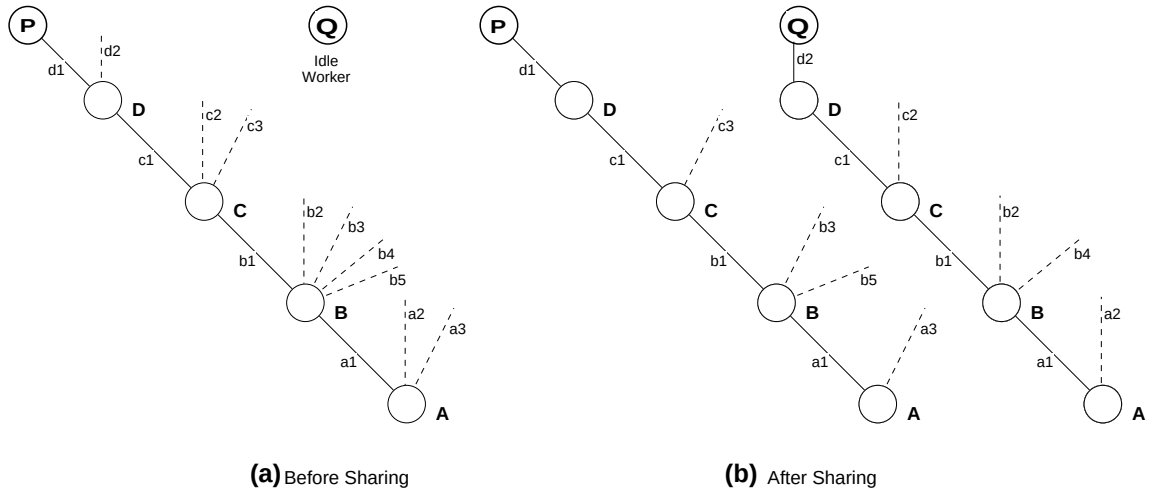

Figure 2.4: *Horizontal Splitting.*

positioning. This is shown in Figure 2.5.

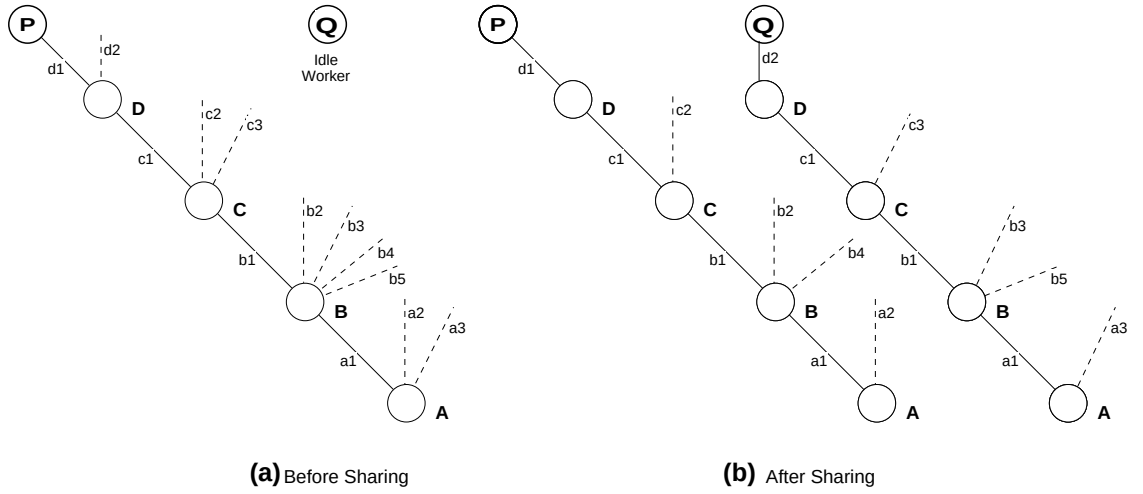

Figure 2.5: *Diagonal Splitting.*

Figure 2.6 illustrates a novel strategy introduced and proposed in this thesis, named *Half Splitting*. Basically, this strategy is similar to *Vertical Splitting* and can be resumed as a bi-partition of the available choice points in worker P. The closest choice points to root are assigned to the requesting worker Q, and the remaining stay in P computation.

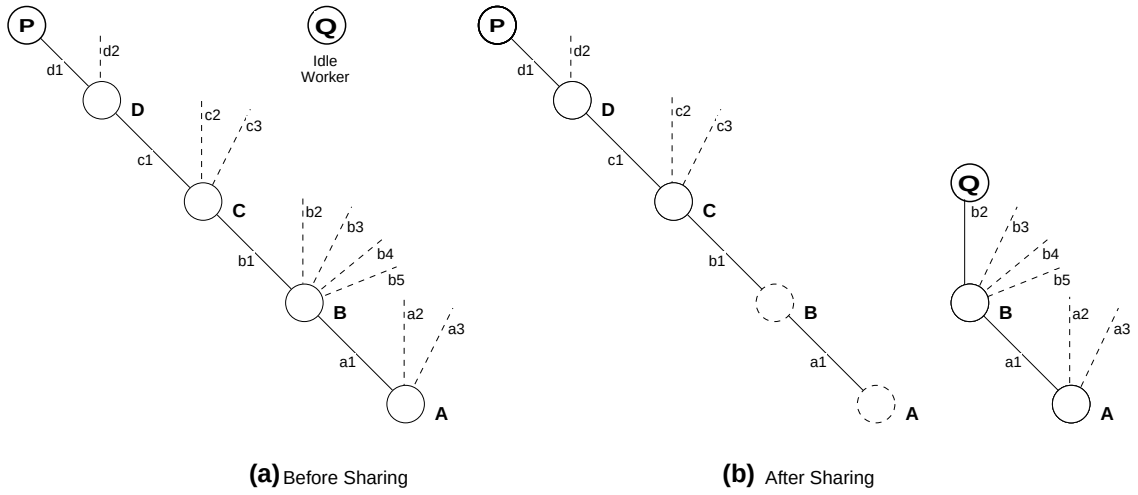

Figure 2.6: *Half Splitting.*

In this thesis, special emphasis will be given specially to the *Vertical Splitting* and *Half Splitting* strategies.

2.3 Chapter Summary

We introduced briefly Logic Programming and the Prolog language. In particular, we discussed in detail the Warren Abstract Machine, as the pillar for most Prolog implementation basis, and we highlighted the main advantages of exploring Or-parallelism and the solutions encountered for environment representation and work distribution in parallel Prolog.

The following chapter will introduce the YapOr system, a parallel implementation of Prolog based on the Environment Copying model with the Incremental Copy scheme.

Chapter 3

The YapOr Engine

This chapter introduces *YapOr*, an or-parallel engine that extends the Yap Prolog system to exploit implicit or-parallelism in Prolog programs [16]. YapOr is based on the Environment Copying model and follows most of the Muse implementation concepts [3].

3.1 Execution Model

In YapOr's shared memory execution model, each worker has a reserved local memory space with the same layout, and shares a global memory space. As a WAM based model, there are certain requirements that must be carried out for the efficient parallelization of WAM's sequential model. The idea is to have each worker behaving, most of the time, as a pure Prolog sequential engine and, at some points of the execution, it switches to execute scheduler tasks in order to synchronize or distribute work among the available workers.

In YapOr, a *worker* is represented by a single processor or system process sharing two common characteristics:

- **Sequential execution** Each *worker* is denoted as a sequential execution entity with the usual WAM execution stacks.
- **Logical address space** Each *worker* manipulates its private execution stacks by using an identical logical address space and sharing a common global address space.

YapOr's addressing space is thus divided into two spaces: the global space and the set of local spaces.

- The **global space** provides the basic support for sharing the main structures for parallel execution. The space is divided into four parts: *code area*, where the code for the loaded programs is stored; *global information area*, where synchronization data is stored; *shared structures creation area*, where shared structures are allocated; *solutions area*, where the solutions for the current executing goal are stored.

- The **local space** is divided into a subset of local spaces, each belonging to a system worker. The *local stack*, *heap stack* and *trail* are stored in each worker's local space, representing the worker's execution state. Besides this, there is a *local information area*, where local data related to parallelism is stored.

In the WAM, the nodes in the search tree are represented by the choice points containing the unexplored alternatives of a certain predicate. These alternatives are independent and, thus, can be executed in parallel.

A node is considered *private* when it is only accessible by the owner. On the other hand, a node is considered *shared* when it is accessible by more than one worker. A node is defined as a *live node* when it owns at least one unexplored alternative. It is defined as a *defunct node* if there are no more alternatives left to try. When a node is stated as shared, there is an identical choice point entry in the *local stacks* of the workers who possess it. For each worker, this divides the search tree in two regions: a *private region* and a *shared region*.

The execution procedure for the environment copying model can be summarized as follows:

1. Just before a program execution begins, all workers are *idle*. When the execution starts, one of the workers, namely P, is set to trigger the first unexplored alternative for the given *query*. If P then executes a predicate with more than one alternative, in practice, this corresponds to create a choice point in P's local stack. This said, if P has more than one alternative in a choice point, this choice point is represented as a *live node* and can then be shared among other workers.

2. After that, one idle worker, say Q, calls for work to P in order to cooperate with it in the pending unexplored alternatives.

3. Thereafter, if P accepts the request it should share its private nodes with Q which includes:

   - For each private node, P creates a *shared structure* in the global space, in order to dictate the correspondent nodes as shared nodes (see Figure 3.1). This structure includes information about the unexplored alternatives left in the node and about the workers sharing it (see Figure 3.2). Each private choice point entry, instead of pointing to the alternatives set, now points to the correspondent created entry in the shared structure entry chain (see field `OrFr_alt`) and includes information about the workers membership owning the choice point at hand (see field `OrFr_members`). Besides that, and in order to avoid simultaneous access to these structures, they use a *lock* mechanism that guarantees mutual exclusion in the deliver of an available alternative to the workers sharing a node (see field `OrFr_lock`). Also, it is implemented a new field, named `OrFr_nearest_livenode`, which may contain a reference to the nearest live node or **null** if no more live nodes exist. In YapOr, these shared structures are called *or-frames*.
   - P copies his current state to Q, which corresponds to copy the local stack, heap stack and trail respectively.

4. After the sharing process, P proceeds with its computation, while Q simulates a failure. This failure calls the backtracking mechanism in order to get an unexplored alternative from the shared region, ie., from the corresponding youngest shared structure, meaning that there is an access interface to scatter unexplored alternatives among the workers. To prevent racing conditions on accessing the shared structures it is used a lock based mutual exclusion mechanism.

5. Whenever a worker has no more alternatives to explore, it returns to the *idle* state and comes back to search for a *busy* worker in order to inherit some of its unexplored work.

6. When all alternatives of the search tree have been explored, the computation terminates and all workers return to the idle state.

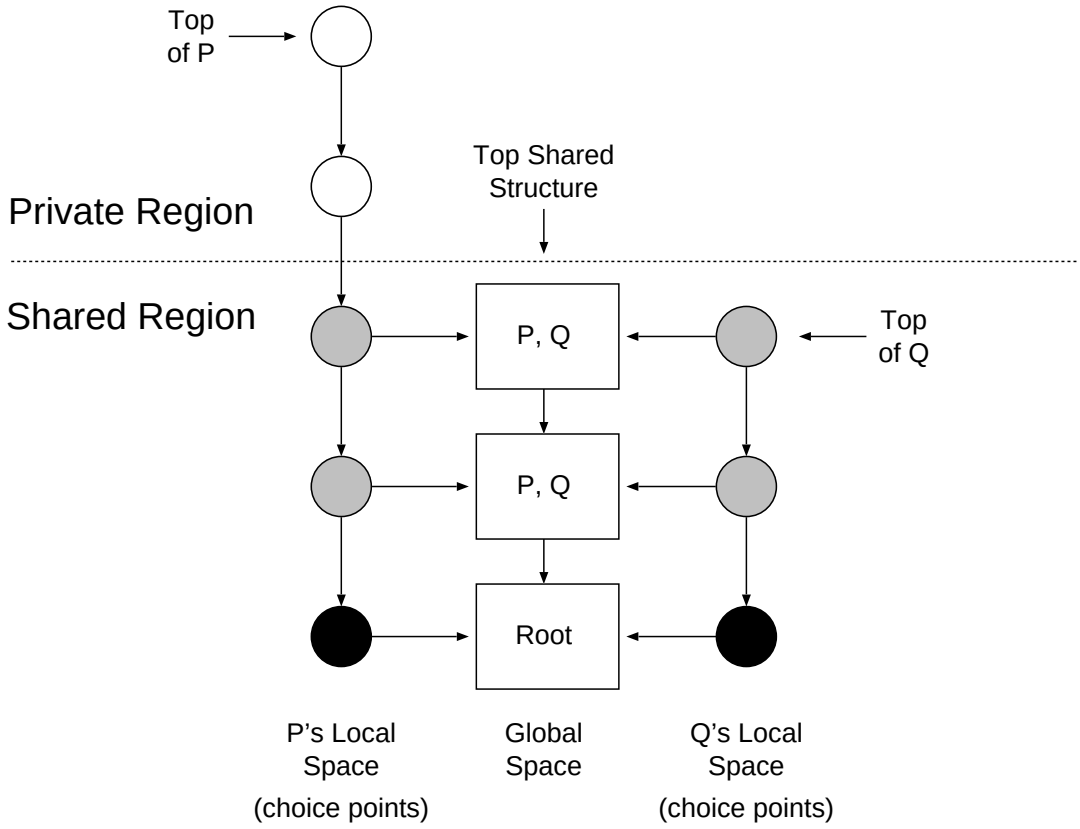

Figure 3.1: Relation between choice points and shared structures.

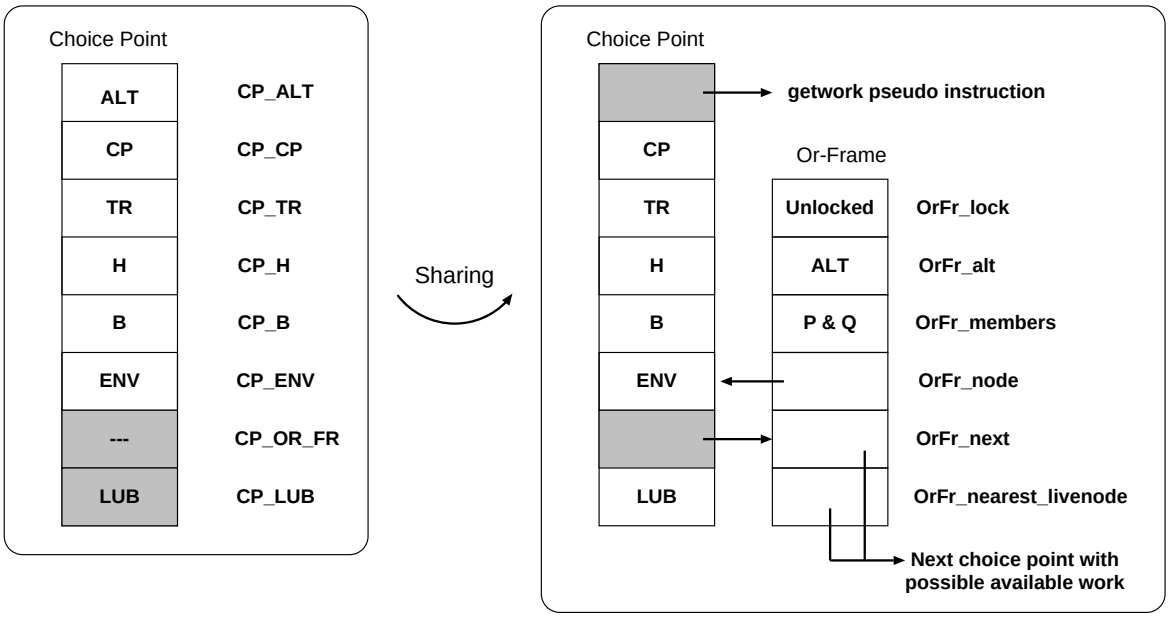

Figure 3.2: Sharing a choice point.

3.2 Incremental Copy

One of the goals when sharing work is to put the two workers in the same computational state. This includes copying the stacks of a worker to another worker, which can be considered an important overhead. Thus, a mechanism that minimizes the amount of data transferred between the sharing worker P and the requesting worker Q, can play an important role. One such mechanism is called *Incremental Copy* of the environment assigned to a worker. Instead of copying the full stacks, the incremental copy technique copies only the difference between the states of the two workers. To do that, worker Q must first travel to the *youngest common node* with P, to allow state consistency before asking for work. The process then determines the difference stack limits and copies the missing stacks from P to Q.

This mechanism is efficiently implemented in YapOr. When a worker Q has no more work in its computation sub-tree, ie., all choice point nodes are defunct, and there is a worker P with live nodes, Q asks P to send sharable work. In order to be in the same computational state, worker Q backtracks to the youngest common node with P (see Figure 3.3). Worker P is now available for creating the shared structures in the global space starting from the youngest common node. Then, for worker Q, it just copies the parts from the local stack, heap stack and trail which reflect the difference between P and Q states. This difference is measured by using the information on the choice point corresponding to the youngest common node of Q and P and, by consulting the top stack segments of P.

In case of existing references in P's trail, to variables in the common stack segments the respective assignments must also be updated in Q in order to maintain the intended state consistency. The extra copy of these carriable assignments is named the *installation phase*.

3.3 Scheduler

This section describes the work distribution solution implemented in YapOr [14, 16].

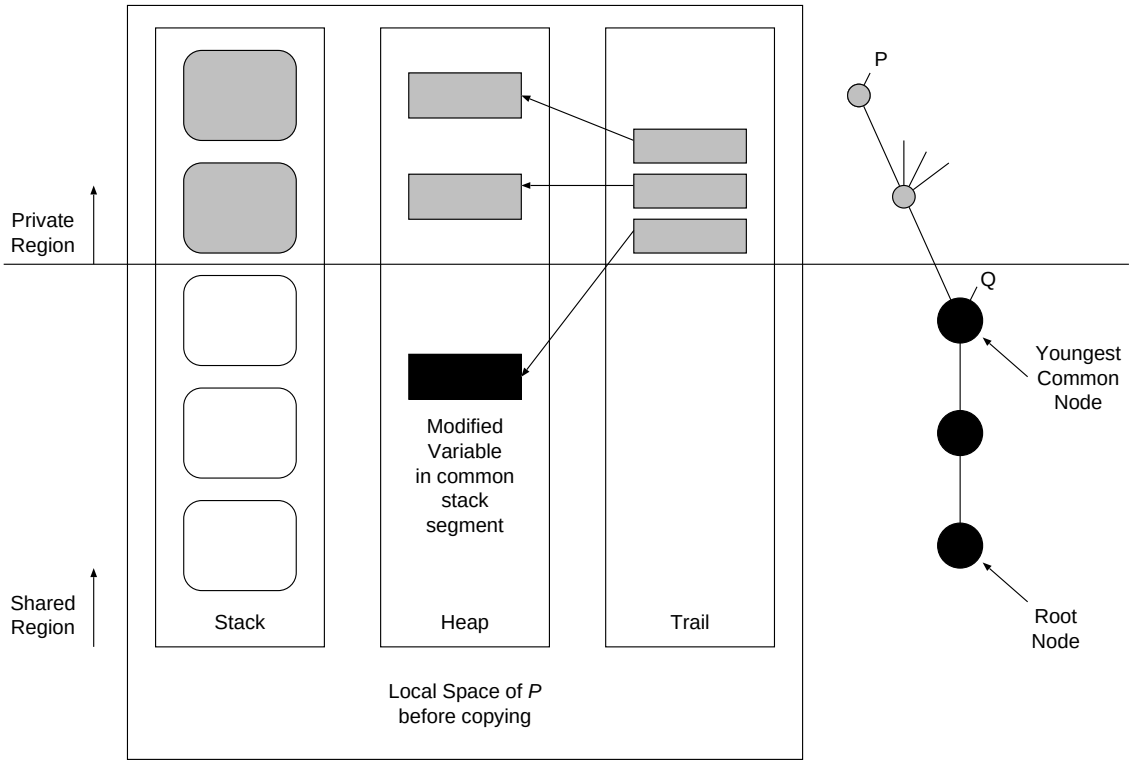

Figure 3.3: Incremental copy's important conditions.

3.3.1 General Ideas

YapOr implements two execution modes: *scheduling mode* and *engine mode*. A worker enters in scheduling mode whenever it runs out of work and starts searching for available work. As soon as new work is assigned to it, the worker enters in engine mode. In engine mode, a worker acts similarly as in pure sequential Prolog except that it has the ability to communicate with other workers.

One of the problems to maintain a good scalability is that available work is irregularly sized and thus some workers may terminate at different times, originating an unbalanced work distribution. Thus, the two main functions of a work distributor are: keeping intact the correctness of Prolog sequential semantics and, efficiently, assign new *tasks*, ie. continuous portion of work, among the available idle workers. Some of the main overheads that affects the overall system performance are: **(i)** the sharing process of the private nodes and the copy of the state parts of a worker and, **(ii)** synchronization to get new tasks from the shared region.

3.3.2 Strategies

Proposed by Ali and Karlsson [2], YapOr takes advantage of the following scheduler strategies used up to enhance the original Muse implementation:

- In the work sharing process, the sharing worker with sharable work must share all of its private nodes. This maximizes the shared work quantity, which allows the requesting worker to find new tasks in the region already shared without entering again in the idle state.
- The scheduler must select the worker with the *largest* amount of work and *closest* in the search tree to the idle worker. The amount of work corresponds to the number of unexplored alternatives owned by a worker in its sub-tree. Again, this is intended to maximize the shared quantity of work. Being the closest worker to the idle worker regards the relative position and proximity of both workers in the search tree, this is intended to minimize the amount of stacks to copy between workers.
- When the scheduler is not able to find available work in the system, it must try to move in advance the idle worker into a better position in the search tree, in order to avoid unnecessary costs to the system in future sharing operations.

The scheduler basic procedure can be resumed as follows. Every time a worker backtracks to a shared choice point, it tries to solve the next unexplored alternative. If there are no unexplored alternatives, the worker becomes idle and tries to select a busy worker, with a sharable amount of work, searching first for workers below its current position and only if no such worker exists it searches for workers above its position. When no such worker exist, the scheduler moves the idle worker to the best available position in the search tree, in order to minimize the future sharing operations.

Regarding the selection of busy workers, a good strategy might be the one that selects the busy worker according to the relative distance between them and, the one with the most available work. Another implemented strategy is: if Q did not find a busy worker P starting from the sub-tree of its current node, it opts by finding a busy worker outside the sub-tree of its current node. Among them, Q chooses the worker P with the most available work and backtracks in the tree until reaching the youngest common node with P. Then the worker Q will try the work sharing process in order to claim work from P.

3.4 Or-parallelism Support

This section discusses in more detail YapOr's main support mechanisms for extending the sequential model to or-parallelism, which includes the environment copying model, the scheduling policy and scheduling support.

3.4.1 Implemented Mechanisms

The process of searching for available work in the shared region, can be made more efficient by quickly finding the youngest live node in the current branch of the tree. For this, it is necessary a decision mechanism to find such live node in an efficient way.

Remember that shared structures created during the operations of work sharing have information about unexplored alternatives and about the workers sharing the node. Also, to prevent execution racing conditions a *locking* mechanism and *membership* mechanism are used. At last, for decide whether there is or not a living node in the current branch of tree, it is used the *nearest live node* decision mechanism (see Figure 3.2).

When a worker ends a shared task, it verifies if the current youngest node is alive. If so, it locks the or-frame corresponding to the node at hand and takes the next available alternative. After that, it updates the or-frame with the next alternative of the taken task (decreasing the number of available alternatives), releases the lock and starts the taken task. If the node is defunct, as the node with depth 3 in Figure 3.4, the worker accesses the `OrFr_nearest_livenode` field in the or-frame structure, which will lead to the nearest live node with available work or, if pointing to the root or-frame, will mean that there is no more available work in the current branch of the tree.

3.4.2 Work Sharing Process

This process happens when an idle worker Q sends a work request to a busy worker P and P accepts such request. A sharing request can be refused: **(i)** when Q is not in the search branch of P; **(ii)** when P has a lower value than the stipulated minimum load threshold value; **(iii)** when P's and Q's top or-frame are equal, that is when P has no more unexplorable work besides the executing alternative. Though, when P sends a negative answer, Q returns to scheduler mode and starts searching for another busy worker.

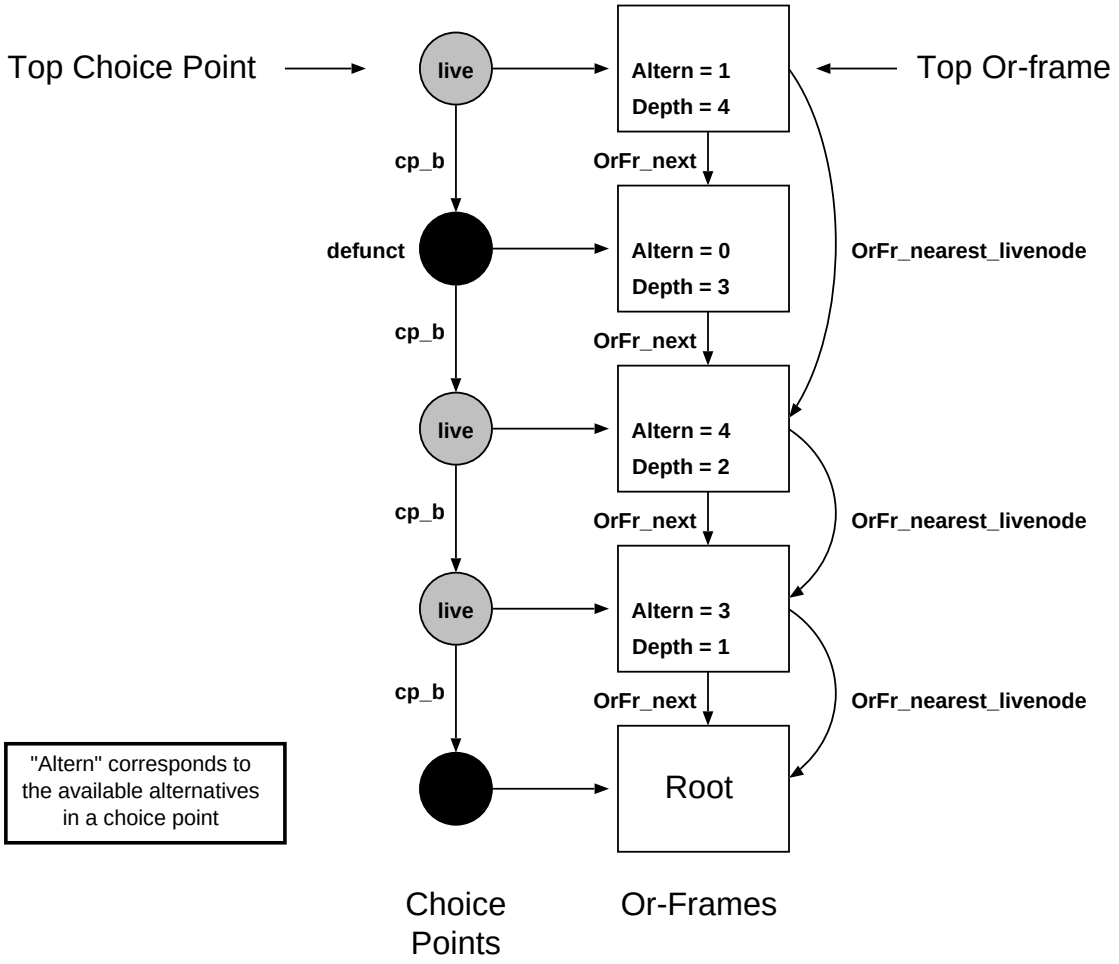

Figure 3.4: Nearest live node.

Accordingly to [16], the process of sharing can be divided in four main steps: **(i)** the *Initial step* is where some auxiliary variables are initialized and the stack segments to be copied are computed; **(ii)** the *Sharing step* is where the private choice points are turned into public ones; **(iii)** the *Copy step* is where the computed segments are copied from the busy worker stacks to the idle worker ones and, at last, **(iv)** the *Installation step* is where the variables in the maintained stacks of the idle worker are updated with the bindings present in the busy worker stacks.

The computation of the stack segments to copy, when using incremental copy, is done by worker P. These stack segments are shown in detail in Figure 3.5.

The limits `Q[B->cp_h]` and `Q[B->cp_tr]` are both representative of Q's youngest choice point fields. `Q[B]` defines the youngest choice point of Q in the search tree, as `P[B]` defines for P, and `Q[top_node]` defines the youngest shared choice point of Q as

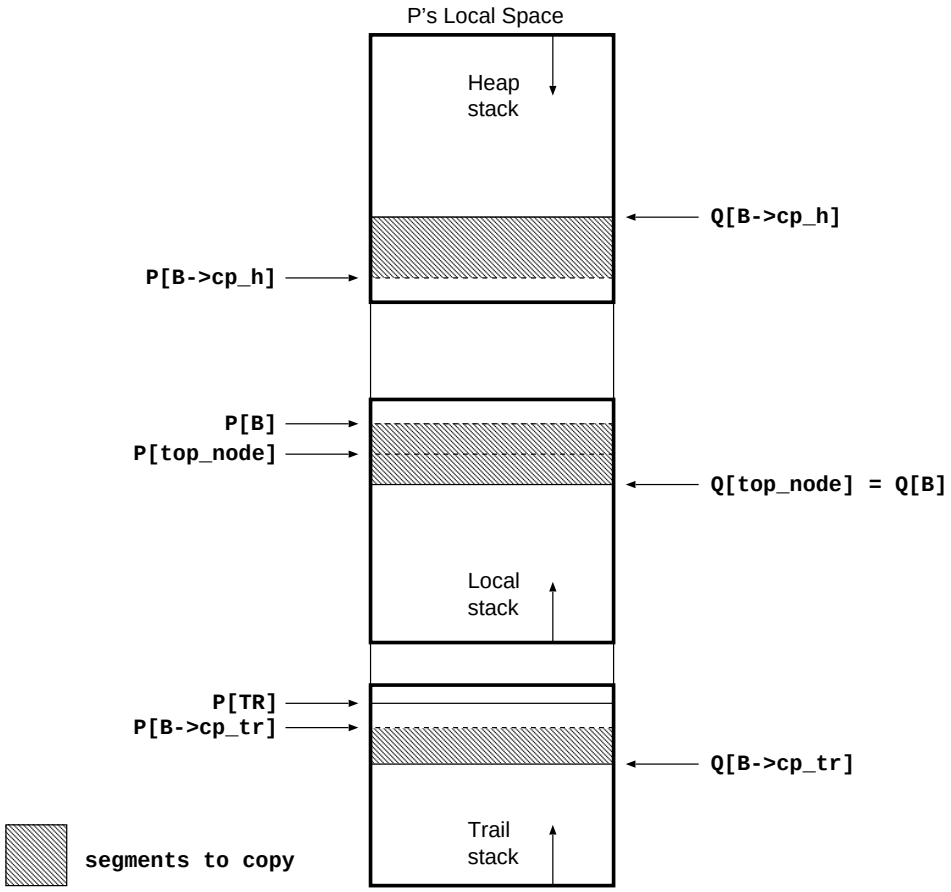

Figure 3.5: Memory areas involved in the incremental copy process.

`P[top_node]` defines for P. The `Q[top_node]` always coincide with `Q[B]` which does not happen with P. `P[top_node]` does not coincide with `P[B]` when P has private work.

In YapOr, the four steps mentioned above are implemented in cooperation by the two workers involved in the sharing process. The sharing execution of P is set by the `p_share_work()` function, and for worker Q, it is set by the `q_share_work()` function. The cooperation exercised by both workers tries to minimize the cost of the sharing process. The P and Q sharing process is detailed in Figure 3.6.

While Q is initially waiting to receive a *sharing* signal from P, P accepts the share request and computes the stacks to copy. When the request is accepted, Q starts the stack's copying in advance as P shares its private nodes pairing them with new shared structures. After that, P enters in copy mode along with Q in a synchronized way and the stacks are copied in the following order: trail, heap and local stack for Q and local stack, heap and trail for P.

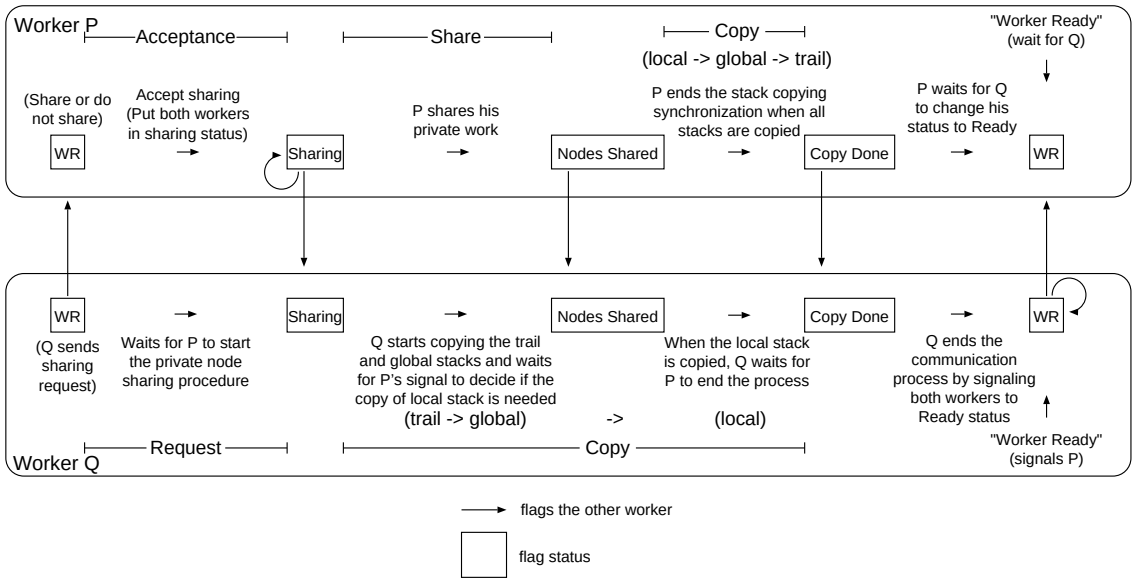

Figure 3.6: The work sharing synchronous process.

After the copy is done, P continues execution and goes back to Prolog execution while Q performs the incremental copy *installation phase*. The installation phase, as shown in Figure 3.7, is the process of installing the bindings in P referring to variables in the maintained stack segments of Q. The installation procedure traverses all the references to variables in the copied segment of the trail. Note that, the bindings assigned by P's current alternative are not copied, because it is considered unnecessary work, since the backtracking of Q to restart execution would simply dispose those bindings. In Figure 3.5, this area corresponds to the interval between `P[TR]` and `P[B->cp_tr]`.

While Q's installation step is not complete, P cannot dispose the shared nodes from its stacks. This is necessary in order to avoid possible undoing of bindings and consequently, having different values assigned to the same variable of P and Q.

In Figure 3.7, the gray area represents the stack segments to be copied from P to Q. Notice also how variables are created and/or instantiated between choice points. For example, there are variables whose values are assigned at the moment they are created. A good example would be variable **A** which is assigned with the value zero when created. Other variables are only instantiated afterwards. For instance, variable **C** was created before the choice point **CP3** and then it was assigned with the value two only after the choice point **CP3**. When this happens, such instantiations are considered conditional bindings and must be marked with a reference in the trail stack, as shown in the local space of P in Figure 3.7.

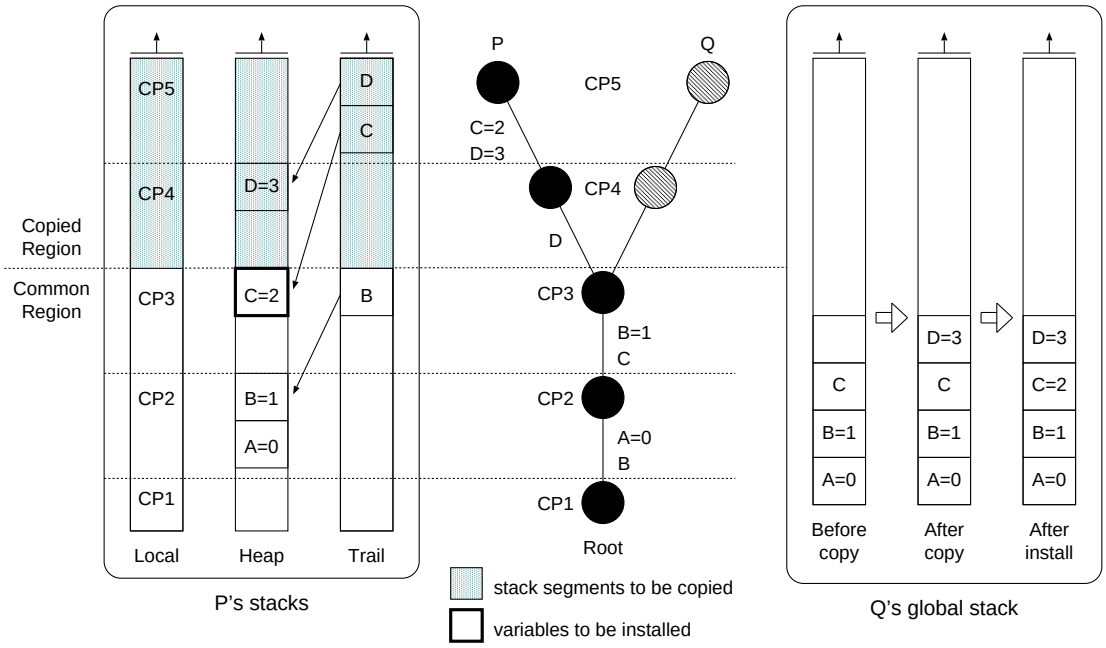

Figure 3.7: Q's installation phase.

After the stack copying is finished, the worker Q may have wrong values in the variables belonging to the common stack segments that were assigned by a certain value in P's private region and thus, a direct copy of such variable's assignments must be made. As shown in Figure 3.7 on the right, the variable **C** fits in these conditions and must be installed with the value two. This allows the worker Q to retrieve its state consistency by installing the correct value of every variable referenced in the copied trail segments and instantiated in the common region.

3.5 Chapter Summary

This chapter introduced the YapOr or-parallel engine, an extension of the Yap Prolog system that supports implicit or-parallelism in Prolog programs. Based on the environment copying model of Muse, emphasis was given to the main features of the YapOr system, such as incremental copy advantages, the scheduler behavior and the work sharing process in a distributed stacking system.

In the next chapter, we explain the implementation of stack splitting in the YapOr system.

Chapter 4

Supporting Stack Splitting in YapOr

This chapter describes our approach to extend the YapOr system to efficiently support stack splitting. We start by presenting the main concepts behind the stack splitting technique and then we discuss in more detail two stack splitting strategies to accomplish work sharing, namely vertical splitting and half splitting.

4.1 General Ideas

Stack splitting is a work sharing technique whose main goal is to split the available work of a worker, that is all of its unexplored alternative branches, in approximately two halves, one that will be kept and another to be shared with a requesting worker. The split is done in such a way that no further synchronization will be necessary between the workers involved, the one sharing and the one requesting work, when they search for work in the shared region of the search tree. Here, we detail the implementation of the vertical and half splitting schemes. Having two stack splitting schemes allows extra flexibility to maximize parallel performance as one can pick the strategy that fits best a particular program.

4.1.1 Getting Work in the Shared Region

The main goal of a scheduling strategy is to maximize the performance of a parallel system by transferring available tasks from busy workers to less loaded ones, or to idle workers waiting for work.

In YapOr, the shared region starts with a reference to the youngest shared choice point and since the execution of alternatives is guided through the chaining of the or-frame's `OrFr_nearest_livenode` field, these fields are used to support the implementation of the stack splitting model. Although in YapOr, the `OrFr_nearest_livenode` field was used as an optimization for searching for available work in the shared region, for the stack splitting implementation this field becomes mandatory.

As Figures 4.1 and 4.2 show, our proposal for implementing stack splitting is by reassigning the top point of execution, one assigned to each sharing and requesting worker, and then use the `OrFr_nearest_livenode` field to redirect the work continuation points in such a way that the available work is split in two fully independent computations.

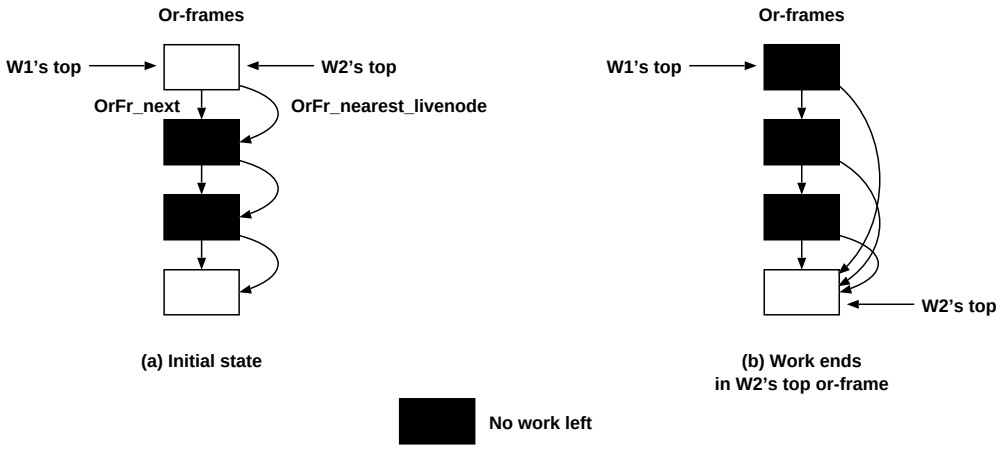

Figure 4.1: Getting work in the shared region with YapOr.

As in YapOr implementation, each worker gets the available work trough the or-frame of the current top choice point - the `top_or_frame` global variable. This allows us to reuse YapOr's general execution model, however, there are some issues that need to be considered. In YapOr's scheduler, there is one procedure, called `put_no_work_in_upper_nodes()`, that nullifies the `OrFr_nearest_livenode` connection of one live node when there is no more work left in all youngest nodes. This is done by traversing the chain of nodes, using the `OrFr_next` fields, and by updating

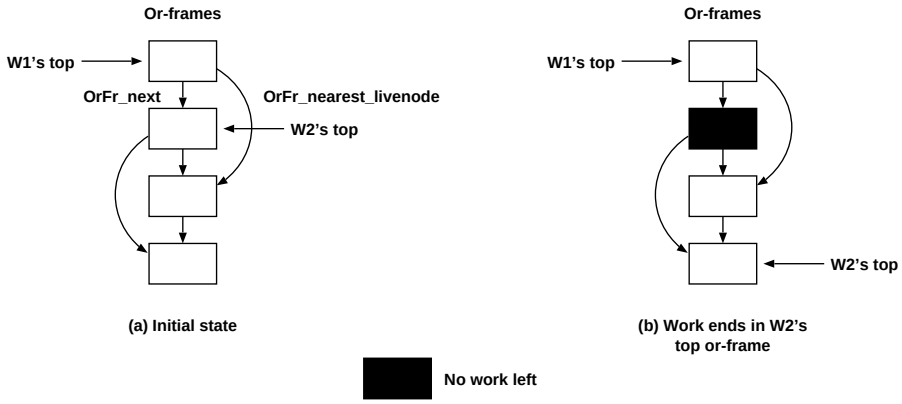

Figure 4.2: Getting work in the shared region with stack splitting.

the `OrFr_nearest_livenode` field to NULL. This happens because in YapOr, any or-frame can become with no work left, independently of its order. For stack splitting this is a dangerous situation since the `OrFr_nearest_livenode` chaining sequence can be incorrectly destroyed. However, for stack splitting, this procedure is irrelevant and can be ignored since there is an independent `OrFr_nearest_livenode` work chaining sequence for each worker and, the work in the older or-frames can never be initiated if there is still work in the younger or-frames.

4.1.2 Copying the Execution Stacks

In a parallel system based on environment copying, the copy of the execution stacks can have a big impact on the system performance if the frequency of such activity or the size of the stacks are very high.

The cost for placing a requesting worker Q in the same computational state as a sharing worker P, is defined by having the stacks of Q in a node common to P, and then by copying to Q the current state of P older than the common node. This operation includes copying the global stack, local stack and trail.

In order to minimize the cost of copying the execution stacks, YapOr implements the mechanism of *incremental copy*. This mechanism defines the segments of the stacks to be copied and instead of copying the whole stacks starting from the root node, it copies starting from the youngest common node, between P and Q. The ending point is assigned to be the last node in P's current state, as denoted by the **B** register that points to the current choice point. By reducing the difference between the starting and ending points, incremental copy minimizes the copying overheads.

With stack splitting, the idea of incremental copy is similar, however, the ending copy point may differ. Due to the pre-determined work distribution strategy in stack splitting, which assigns the evaluation of each shared choice point either to P or Q, the ending copy point is different from P's current choice point, which is not intended to be on Q's execution stacks after sharing. As we will see, this results on having less stack segments to copy.

4.1.3 Membership and Locking

The membership mechanism is one of the main YapOr's function and corresponds to a *bitmap* field in every or-frame data structure that defines the current set of workers that owns or acts upon the respective choice point. Since the or-frames can be accessed by the various acting workers, they are susceptible to racing conditions among the worker's set. For example, to retrieve the next available alternative in a shared choice point or to manage the membership information, a *locking* mechanism is thus used.

With stack splitting this locking mechanism is maintained as well and the bitmap membership is still used to define the set of workers sharing the choice point. *Locking* is needed in order to provide mutual exclusion on managing the membership information on such structures.

For instance, consider that a worker backtracks from a shared choice point: the worker must access the or-frame's bitmap of the corresponding choice point and zeroing the position referring to it. Then, if there are no more workers marked in the bitmap, meaning that the or-frame is only owned by the current worker, the or-frame is removed.

One of the advantages of the stack splitting model is that each choice point stays with only one worker responsible for the execution of its alternatives and, thus, the simultaneous access by more than a worker to the same choice point to retrieve alternatives never happens. Note that this differs from the case where the choice point is shared, thus owned, by multiple workers in which case the respective or-frame's bitmap and locking mechanism are still required.

4.1.4 Sharing Work

In this subsection, we introduce in more detail the implemented procedure for sharing work in YapOr. Its goal is to share all private nodes of a busy worker, and in the following sections we discuss its extension to support stack splitting. In general, the method of interaction between workers is maintained for both models. The selection, requesting and negotiation parts are a common feature between YapOr's implementation and stack splitting. The major difference has to do with how to share the private choice points. In YapOr, this is done by the procedure called `share_private_nodes()` and one of its functions is to define the available work chain sequence. When a worker P provides another worker Q with some of its available work, P shares with Q all private nodes in its branch of the tree. A worker P accepts sharing its private work with a worker Q only when P has an acceptable private work load, above a given *threshold* value.

For YapOr, the process of sharing P's private nodes is divided into five main stages:

1. **Sharing loop.** This stage corresponds to the sharing of the private nodes of P. For each private node, a new or-frame is allocated and the access to the unexplored alternatives previously done through the `cp_ap` pointer in the private choice points, is now done through the new or-frame which acts as an interface for work retrieval. The alternative's access is moved to the or-frame field `OrFr_alternative`, and the private `cp_ap` pointers are updated to a `getwork` pseudo instruction. All the private nodes have now a corresponding or-frame, that are sequentially connected through the fields `OrFr_next` and `OrFr_nearest_livenode`. These frames are also assigned to P and Q in the membership field (see Figure 4.3).

2. **Connecting old shared frames.** After sharing the private nodes, what is left undone is the connection between the last new created or-frame and the current `top_or_frame` of the worker. With such connection, we complete the chaining between the newer and the older or-frames (see Figure 4.4).

3. **Updating depth.** In this stage, the `depth` field for the newly created or-frames is updated to the value corresponding to the or-frames deepness in the tree's branch (see Figure 4.5).

4. **Updating old shared frames.** Next, the old shared or-frames on P's branch are updated. Since the depth value is already correct on such frames, it only

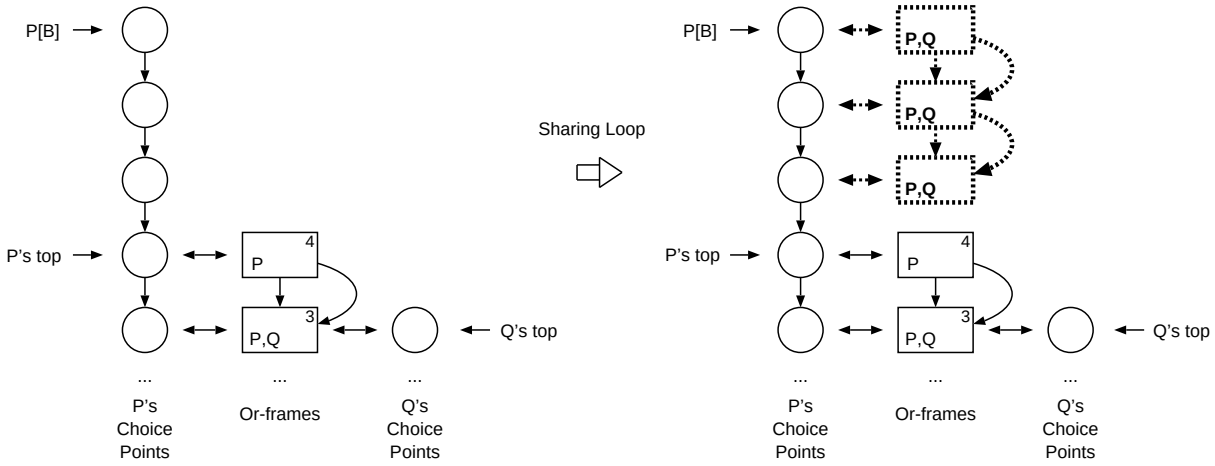

Figure 4.3: Stage 1: Sharing loop.

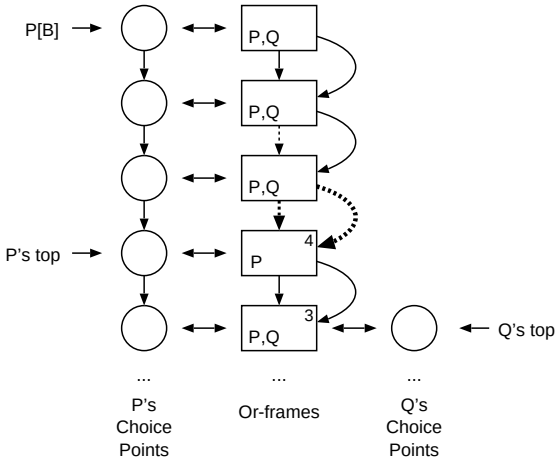

Figure 4.4: Stage 2: Connecting old shared frames.

remains to include the requesting worker Q in the membership field in the frames starting from P's current `top_or_frame` til Q's `top_or_frame` (see Figure 4.6).

5. **Updating top shared frames.** Finally, the new top or-frames in each worker are set. In YapOr, since all shared work is available to both workers, both workers get the same `top_or_frame` (see Figure 4.7). This is not the case for stack splitting as there are no workers with the same assigned nodes, and thus the `top_or_frame` pointers are not the same. As we will see next, they are set accordingly to the splitting scheme at hand.

In a certain way, we can say that the stack splitting model is going to be embedded into these stages. After this procedure, the stacks are copied as usual.

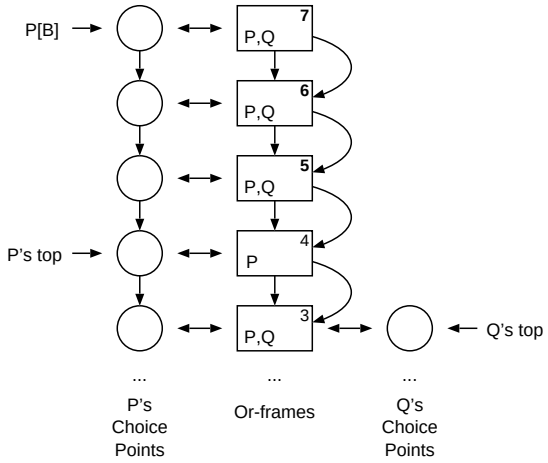

Figure 4.5: Stage 3: Updating depth.

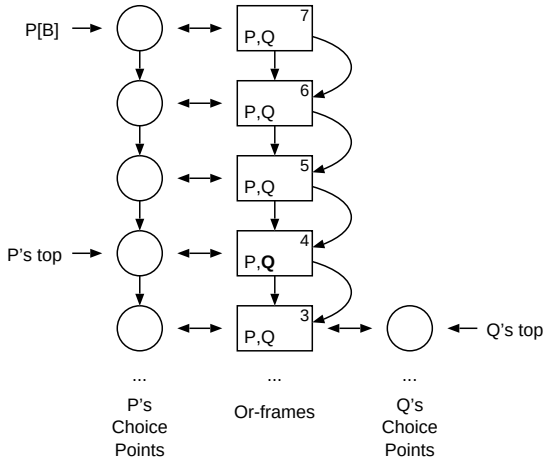

Figure 4.6: Stage 4: Updating old shared frames.

4.2 Vertical Splitting

The vertical splitting strategy follows a pre-determined work splitting scheme in which the chain of shared choice points defining the worker's execution path is divided in two, by alternating the choice points assigned to the sharing workers. At the implementation level, the idea of this scheme is to use the `OrFr_nearest_livenode` field in order to generate two alternated chain sequences in the or-frames, and thus divide the initial sequence of work in two independent execution paths. Workers can share the same or-frames but they have their own independent path without caring for the or-frames not assigned to them.

The acceptance of a sharing request may not always succeed. When P has no private

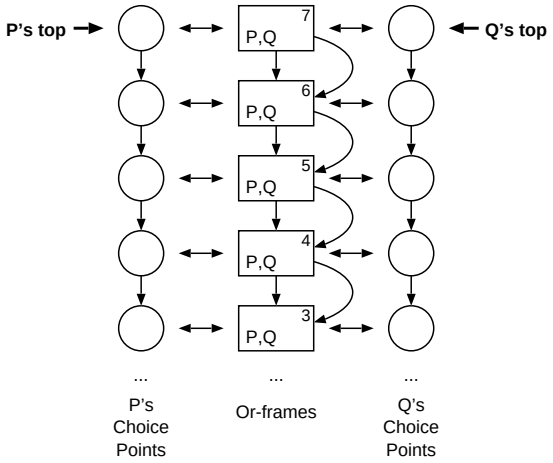

Figure 4.7: Stage 5: Updating top shared frames.

choice points and the available work is restricted to the current shared choice point, the sharing request is not accepted. This happens when the worker is executing in a shared choice point with the `OrFr_nearest_livenode` field pointing to a dead-end. In such situation, it is impossible to share the available alternatives in the shared choice point since the procedure must respect the vertical splitting scheme where the division process is made throughout choice points and not alternatives.

We next describe the major changes made to the sharing stages described previously in order to implement the vertical splitting scheme.

As illustrated in Figure 4.8, the first noticeable difference from the previous description is how the `OrFr_nearest_livenode` field is now connected. Instead of sequentially connected, the `OrFr_nearest_livenode` field is now double spaced connected during the or-frame creation process.

Starting from P's youngest choice point, the sharing loop procedure (see Figure 4.9) traverses all P's private choice points and creates a corresponding or-frame by calling the `alloc_or_frame()` procedure. In the pseudo-code in Figure 4.9, the `current_frame`, `aux_next_frame` and `aux_nearest_frame` variables represent, respectively, the or-frame allocated in the current step, the or-frame allocated in the previous step, which is used to link to the current or-frame by the `OrFr_next` field, and the or-frame allocated before the `aux_next_frame`, which is used as a double spaced frame marker in order to initiate the `OrFr_nearest_livenode` fields. To continue the loop, the `aux_nearest_frame` is updated to the `aux_next_frame`, and the `aux_next_frame` is updated to the `current_frame`.

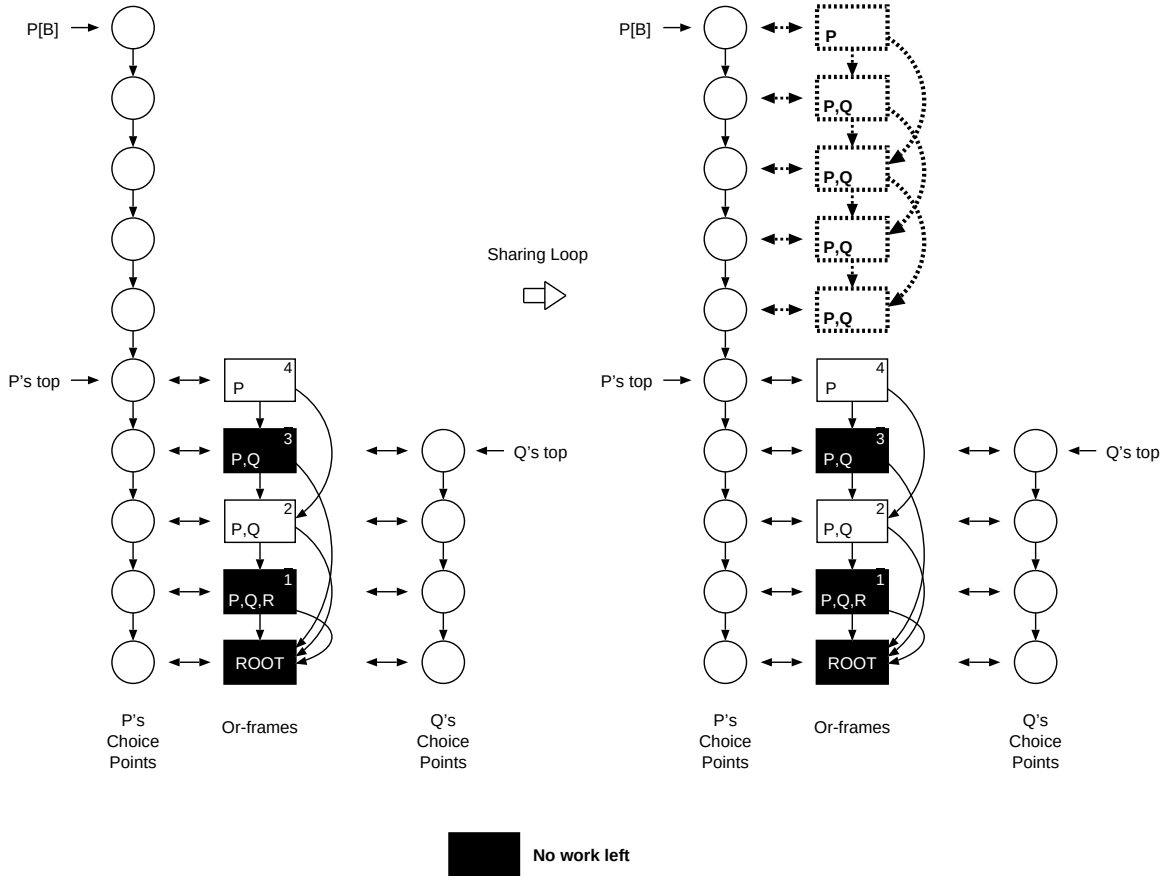

Figure 4.8: Double spaced connection in or-frame creation.

The sequentially created or-frames are connected by the `OrFr_next` fields. So, if there is a defined `aux_next_frame`, its `OrFr_next` field is made to point to the `current_frame` variable. Moreover, If `aux_nearest_frame` is defined, then its `OrFr_nearest_livenode` is also assigned to the `current_frame` variable. For the top choice point, the or-frame is initialized with just the owning worker P in the membership bitmap. The other or-frames are initialized with both workers.

Next, follows the connection of the last newly allocated or-frames with the older and already stored or-frame structure (see Figure 4.10(a)).

In this stage, consideration must be given to the condition of the `current_frame` being the root or-frame or just an ordinary or-frame (see Figure 4.11). If it is a root or-frame, the `OrFr_nearest_livenode` fields of the auxiliary or-frames are assigned with the value `DEAD_END`. If not, they are assigned to the `current_frame` variable, which points to P's current top or-frame.

The `DEAD_END` assignment marks the ending point for unexplored work. Initially, the

```
aux_next_frame = NULL
aux_nearest_frame = NULL
current_cp = B  // the B register points to the youngest choice point
while (current_cp != top_cp)
  // top_cp points to the youngest shared choice point
  current_frame = alloc_or_frame(current_cp)
  init_lock_field(OrFr_lock(current_frame))
  if (aux_next_frame)
    OrFr_next(aux_next_frame) = current_frame
    add_to_bitmap(P & Q, OrFr_member(current_frame))
  else
    add_to_bitmap(P, OrFr_member(current_frame))
  if (aux_nearest_frame)
    OrFr_nearest_livenode(aux_nearest_frame) = current_frame
  aux_nearest_frame = aux_next_frame
  aux_next_frame = current_frame
  current_cp = cp_b(current_cp)  // move to the next choice point on stack
...  // follows the pseudo-code on Figure 4.11
```

Figure 4.9: Vertical splitting sharing loop pseudo-code.

ending point was defined as the root or-frame, but in order to take advantage of the incremental copy technique for stack splitting, the ending point was later defined as the NULL value. This is discussed in detail next in Chapter 5.

The stage that follows is the depth field initialization for the newly created or-frames. For vertical splitting, this is done as in YapOr implementation (see Figure 4.10(b)).

For the next stage, we need to decide where to continue the application of the vertical splitting algorithm for the old shared nodes. If no private work was shared, which means that we are only sharing work from the old shared nodes, the starting or-frame is P's current top or-frame. Otherwise, if some new or-frame was created, the starting or-frame is the last created frame in the sharing loop stage, which was connected to P's current top or-frame in stage two. Either way, this serves the decision to elect the or-frame where the continuation of vertical splitting, guided through the `OrFr_nearest_livenode` field, should start. Figure 4.12(a) shows this update for the old shared nodes and Figure 4.13 shows the pseudo-code for this procedure.

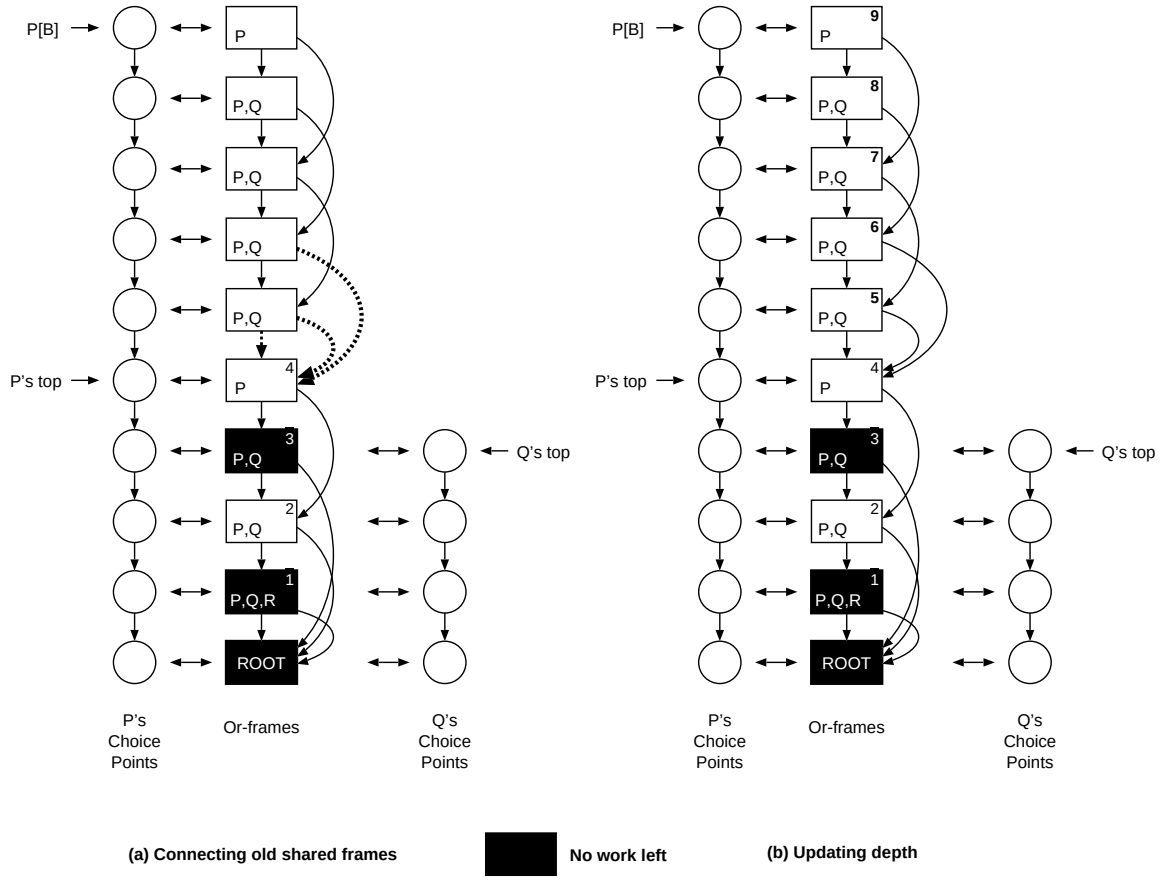

Figure 4.10: Connecting old shared frames and updating depth.

The procedure traverses the old shared frames until a dead-end frame is reached. At each frame lies a reconnection process of the `OrFr_nearest_livenode` field. The `OrFr_nearest_livenode` of the `current_frame` is first saved to the `aux_nearest_frame` variable. Then, if the `aux_nearest_frame` is not a `root_frame`, the `aux_nearest_frame`'s `OrFr_nearest_livenode` is assigned to the `current_frame`'s `OrFr_nearest_livenode`, and the process continues by moving the `current_frame` to the `aux_nearest_frame`. Otherwise, if the `aux_nearest_frame` is the `root_frame`, the process ends.

Finally, Figure 4.12(b), illustrates the next and final stage of the sharing process, the **updating top shared frames** stage. In the stack splitting implementation without incremental copy, where the stacks are fully copied, this stage is similar to YapOr's implementation, except that Q's top or-frame is made to point to the `OrFr_nearest_livenode` of P's top or-frame. As we will see next in Chapter 5, at this stage for stack splitting with incremental copy, we also need to check if Q's top or-frame includes some older P's shared frames, which originates a bitmap inclusion of Q in such frames, or if Q's top or-frame was moved to an older Q's frame, originating

```
...  // pseudo-code on Figure 4.9
if (aux_next_frame)
  if (current_frame == root_frame)
    OrFr_nearest_livenode(aux_next_frame) = DEAD_END
  else
    OrFr_nearest_livenode(aux_next_frame) = current_frame
  OrFr_next(aux_next_frame) = current_frame
if (aux_nearest_frame)
  if (current_frame == root_frame)
    OrFr_nearest_livenode(aux_nearest_frame) = DEAD_END
  else
    OrFr_nearest_livenode(aux_nearest_frame) = current_frame
...  // follows the pseudo-code on Figure 4.13
```

Figure 4.11: Last or-frame connection pseudo-code.

a bitmap exclusion till such frame.

Upon completion of the sharing process, it follows the stack copying phase. In some situations of stack splitting, there is no need for any copy at all, and a backtracking action is enough to place the requesting worker ready for execution. Next, when the stacks are synchronized, in the stack splitting implementation without incremental copy, P returns to its computation while Q processes the *installation phase* before starting execution.

Figure 4.14 illustrates a more complex situation showing three consecutive vertical splitting sharing operations. Following the illustration order, `W1` first creates eight or-frames and assigns four nodes to `W2`. Next, `W1` shares his current pile of available or-frames with a different worker `W3`, which leaves two nodes for `W3` and the remaining two for `W1`. Finally, `W2` shares work with worker `W4`, giving two of its nodes to `W4`.

4.3 Half Splitting

The half splitting strategy also follows a pre-determined work splitting scheme in which the chain of shared choice points in the worker's execution path is partitioned in two consecutive and equally sized parts. Each partition is a set of consecutive choice points that are chained by the connections through the `OrFr_nearest_livenode` field

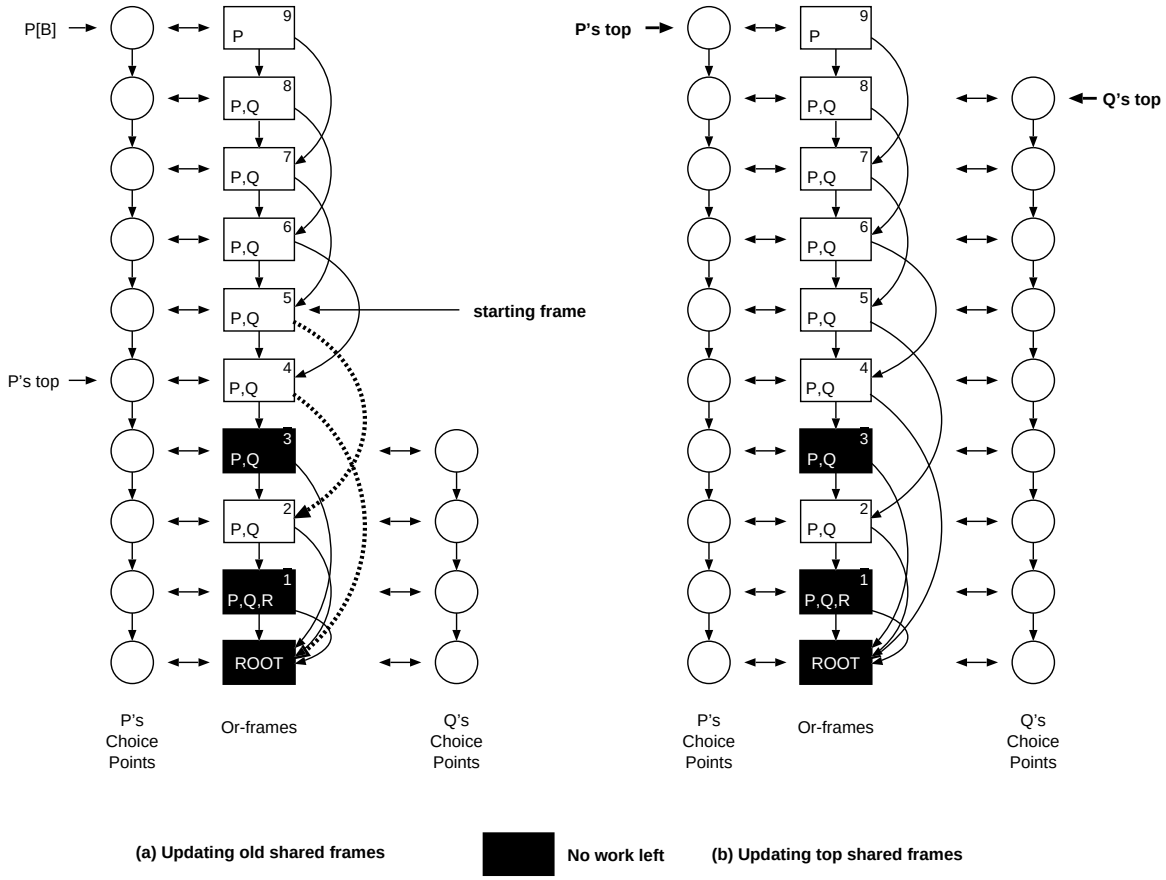

Figure 4.12: Updating the `OrFr_nearest_livenode` in the old shared frames and the top or-frame for both workers.

```
...  // pseudo-code on Figure 4.11
while (OrFr_nearest_livenode(current_frame) != DEAD_END)
  aux_nearest_frame = OrFr_nearest_livenode(current_frame)
  OrFr_nearest_livenode(current_frame) =
      OrFr_nearest_livenode(aux_nearest_frame)
  current_frame = aux_nearest_frame
```

Figure 4.13: Pseudo-code for updating the `OrFr_nearest_livenode` fields in the old shared frames.

of the corresponding or-frames.

In the half splitting strategy, the nodes assigned to a worker are numbered sequentially and independently from the other workers. By doing this, the condition to accept or refuse a sharing request is quite simple. If the current choice point, pointed by register

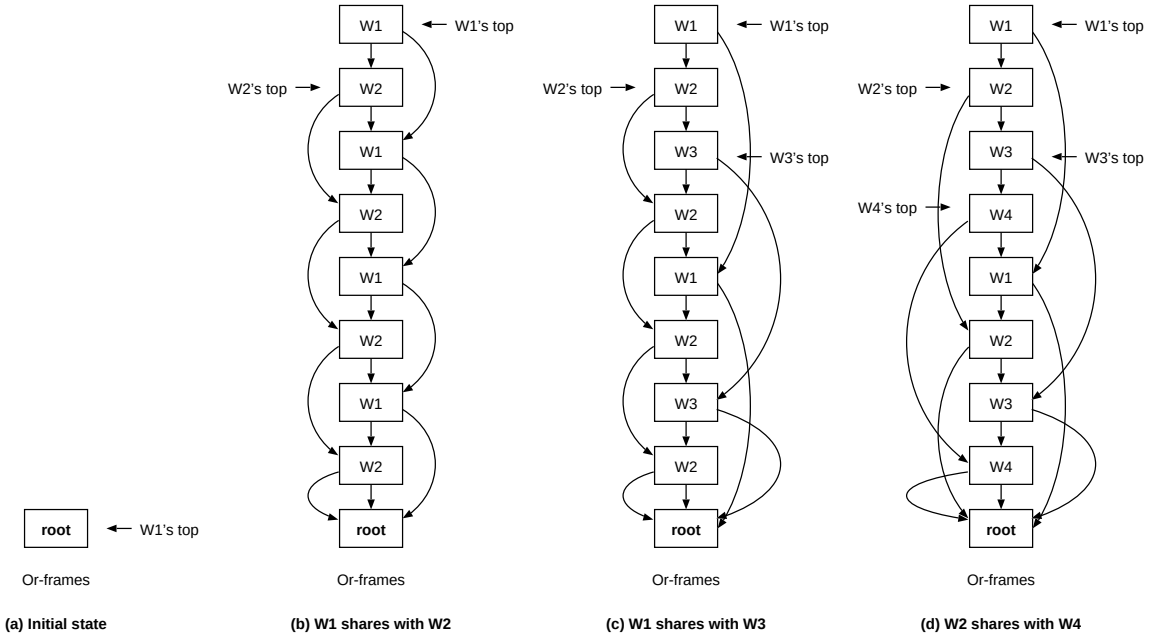

Figure 4.14: Work chaining sequence of or-frames in vertical splitting.

B, is numbered with a value higher than one, then it is considered that there is sharable work. Otherwise, if the value is one or zero, the sharing request will be refused.

4.3.1 Split Counter

In order to support this numbering of nodes, a new field, named *split counter*, was introduced in the choice point structure to provide information about the current choice point's depth in the corresponding partition for the worker at hand.

Note that a global depth information is already stored in the or-frame structures, but this new field is necessary to allow the calculation of the *relative depth* of the worker's assigned choice points. By relative depth we mean the depth starting from the oldest assigned choice point on the worker's current branch.

After an half splitting sharing operation, there will be two consecutive sequences of relative depths, both starting with the value one. One sequence is the actual sequence of the sharing worker current branch. The other sequence starts in the choice point where the split is done. Figure 4.15 illustrates a situation showing how the choice points are numbered accordingly to the split counter mechanism described above.

Every time a split occurs, the sharing worker creates/shares the or-frames needed

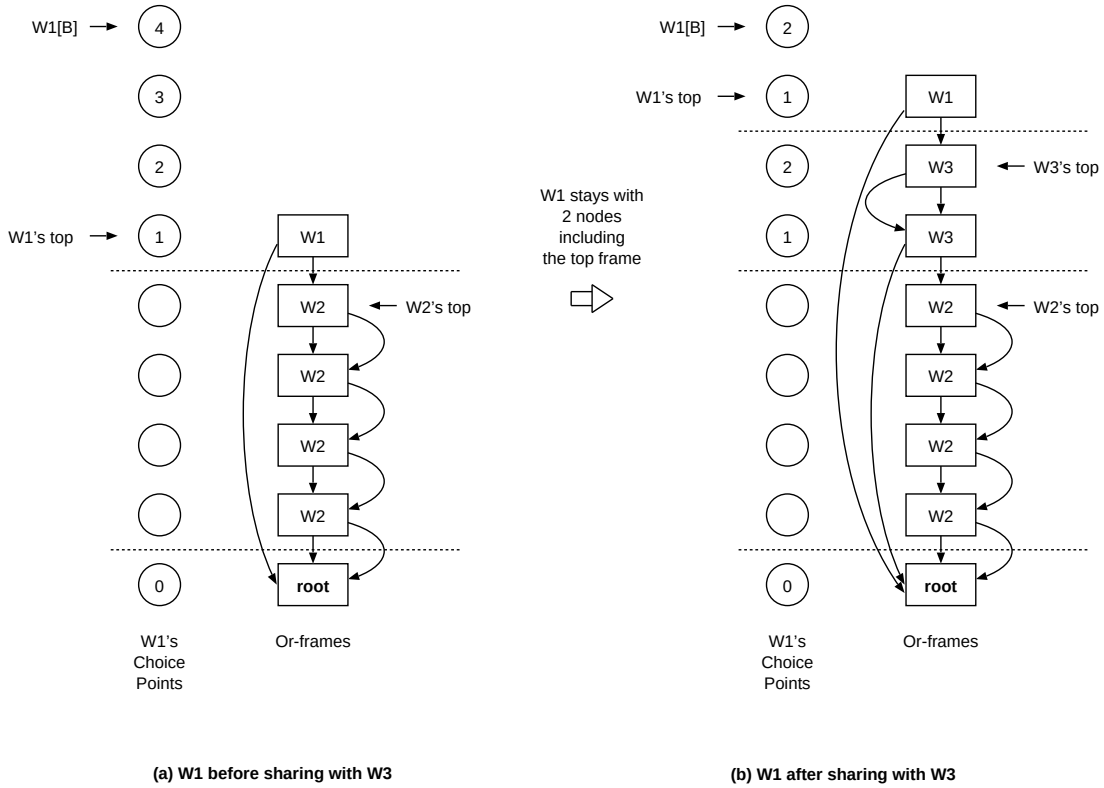

Figure 4.15: Split counter sequences.

according to the half splitting algorithm. In the example, worker `W1` starts the sharing process with four nodes assigned to it, as the counter on the top choice point shows. The choice points numbered 1 and 2 are then moved to worker `W3`, which includes the associated or-frames creation/initialization. The choice points numbered 3 and 4 continue as belonging to worker `W1` and are thus renumbered as 1 and 2, respectively.

Note also that the nodes numbered with the value 1 mark the last node of execution for each worker and may thus protect the worker from backtracking to nodes belonging to other workers. To guarantee such behavior, a practical solution was introduced by creating an or-frame with the `OrFr_nearest_livenode` field pointing to a dead-end, in this case the root or-frame. As explained next in Chapter 5, the dead-end mark may differ when supporting incremental copy.

4.3.2 Sharing Model

Next, we describe in detail the extensions made to YapOr's sharing work procedure, in order to implement the half splitting scheme.

The basic rule of how half splitting model is driven is by sharing only half of the assigned choice points belonging to the sharing worker. Figure 4.16 shows a situation where the sharing worker P has six nodes assigned to it, which means that three of them are going to be moved to the requesting worker Q. To update the split counter after a new split, we need to update the split counter numbers for the choice points starting from P's top choice point (register `B`) until the middle choice point in P's initial partition. Figure 4.17 shows the pseudo-code for this procedure.

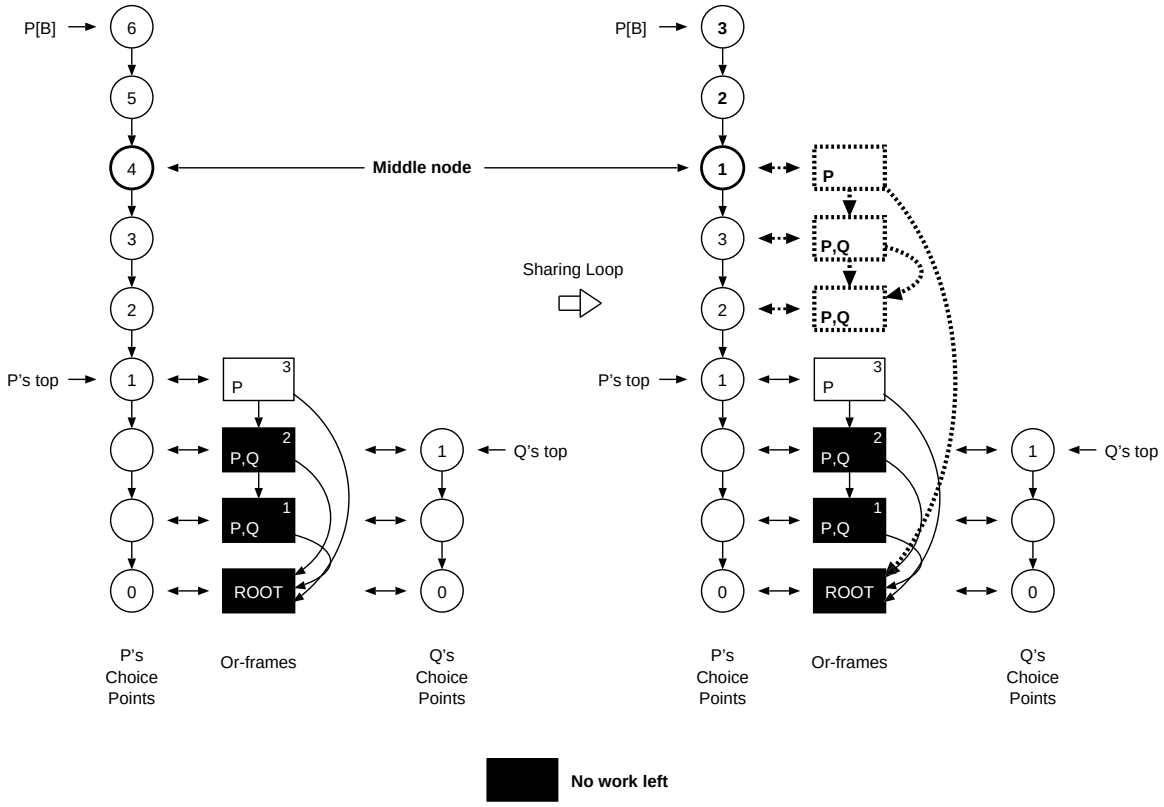

Figure 4.16: Sharing loop stage with half splitting.

```
current_cp = B  // the B register points to the youngest choice point
split_number = cp_sc(current_cp) / 2  // cp_sc is the split counter field
while (cp_sc(current_cp) != split_number + 1)
  cp_sc(current_cp) = cp_sc(current_cp) - split_number
  current_cp = cp_b(current_cp)  // move to the next choice point on stack
cp_sc(current_cp) = 1
...  // follows the pseudo-code on Figure 4.18
```

Figure 4.17: Updating the split counter.

The oldest assigned choice point to the sharing worker is denominated the *middle node*. Starting from P's top choice point (initially `B`), the split counter field, named `cp_sc` is updated by assigning the new counting values in a decreasing order until reaching the middle node, which gets a split counter value of 1.

At the end of updating the split counter, the `current_cp` variable refers to the middle node (see figure 4.17). Starting from the middle node, the remaining nodes are updated to belong to Q, which includes allocating and initializing the corresponding or-frames. For the middle node an or-frame pointing to the dead-end `root_frame` is created. As already mentioned, the goal for this or-frame is simply to mark the end of P's assigned work. This is shown in Figure 4.16 on the right side.

Here, we can distinguish two different situations for the sharing loop stage. In the cases where there are more old shared nodes than private nodes in P's branch, the middle node is already assigned with an or-frame. Thus, there is no need for a procedure for the sharing loop described above. Therefore, we start by checking if the middle node has an associated or-frame and if it is the case, the middle frame is assigned to a dead-end and the requesting worker is excluded from all or-frames from the top frame til the middle frame (see Figure 4.18).

```
...  // pseudo-code on Figure 4.17
middle_frame = cp_or_fr(current_cp)  // current_cp is the middle node
if (middle_frame)
  OrFr_nearest_livenode(middle_frame) = DEAD_END
  current_frame = top_or_frame  // top_or_frame points to the youngest
  while (current_frame != middle_frame)                    // or-frame
    remove_from_bitmap(Q, OrFr_member(current_frame))
else
  // sharing loop stage
```

Figure 4.18: Checking if the middle node is already shared.

After the **sharing loop** stage, it follows the **connecting old shared frames** stage (see Figure 4.19(a)). Unlike vertical splitting, half splitting is not endured to carry the decision of choosing which or-frame will link to the current top or-frame. In half splitting, the process is resumed by linking the last new or-frame to the youngest P's old shared or-frame.

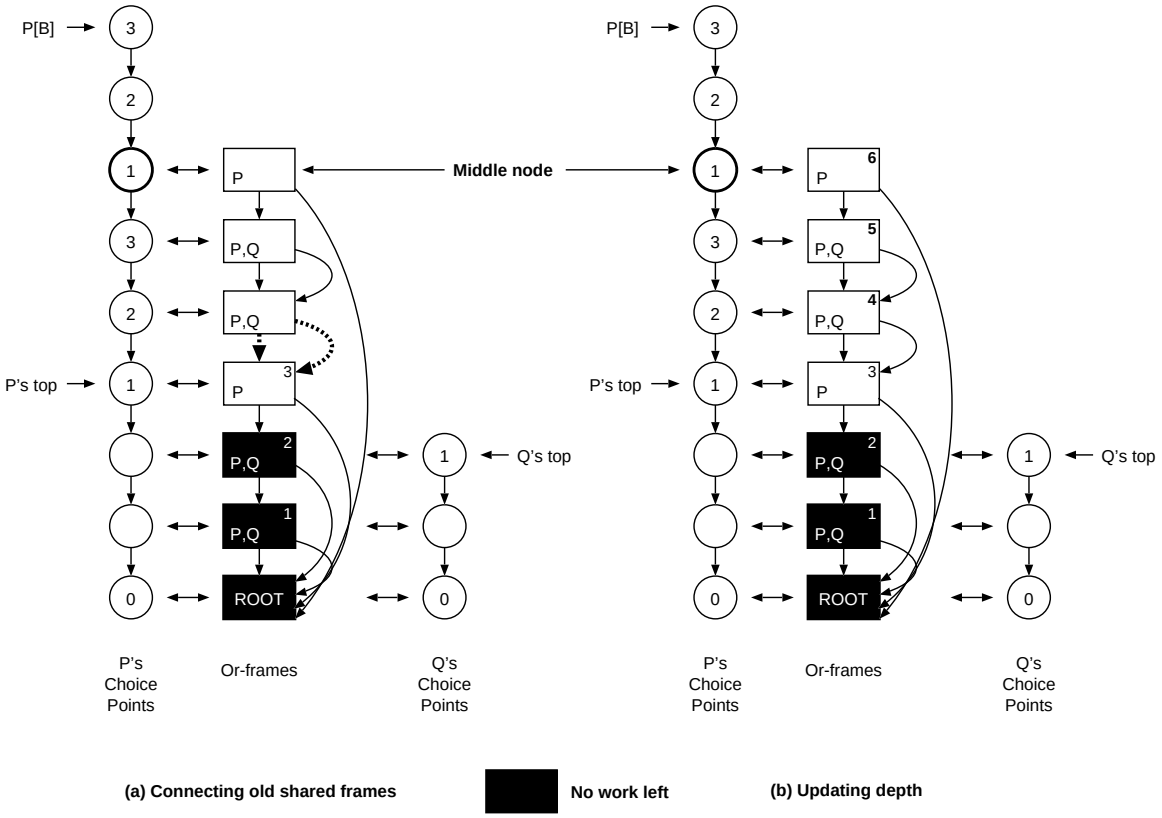

Figure 4.19: Half splitting stages 2 and 3.

The **updating depth** stage then starts from the middle frame until P's top frame and it is the same as for the vertical splitting scheme (see Figure 4.19(b)). Next, after the **updating depth** stage, we may have to decide if stack copying is needed or not. The copy is needed when P's and Q's top frames differ. Note that this equality is possible since Q's backtracking mechanism updates its top or-frame when Q is looking for work. The same goes for the **updating old shared frames** stage. If the equality is false, then the update starts from the frame pointed by the `OrFr_next` middle frame field, by adding Q to the old shared nodes (see Figure 4.20(a)).

The final and last stage is the **updating top shared frames**. As Figure 4.20(b) shows, the top frame of the sharing worker P is assigned to be the middle frame and, the top frame of the requesting worker Q is assigned to be the frame pointed by the

middle frame's `OrFr_next` field.

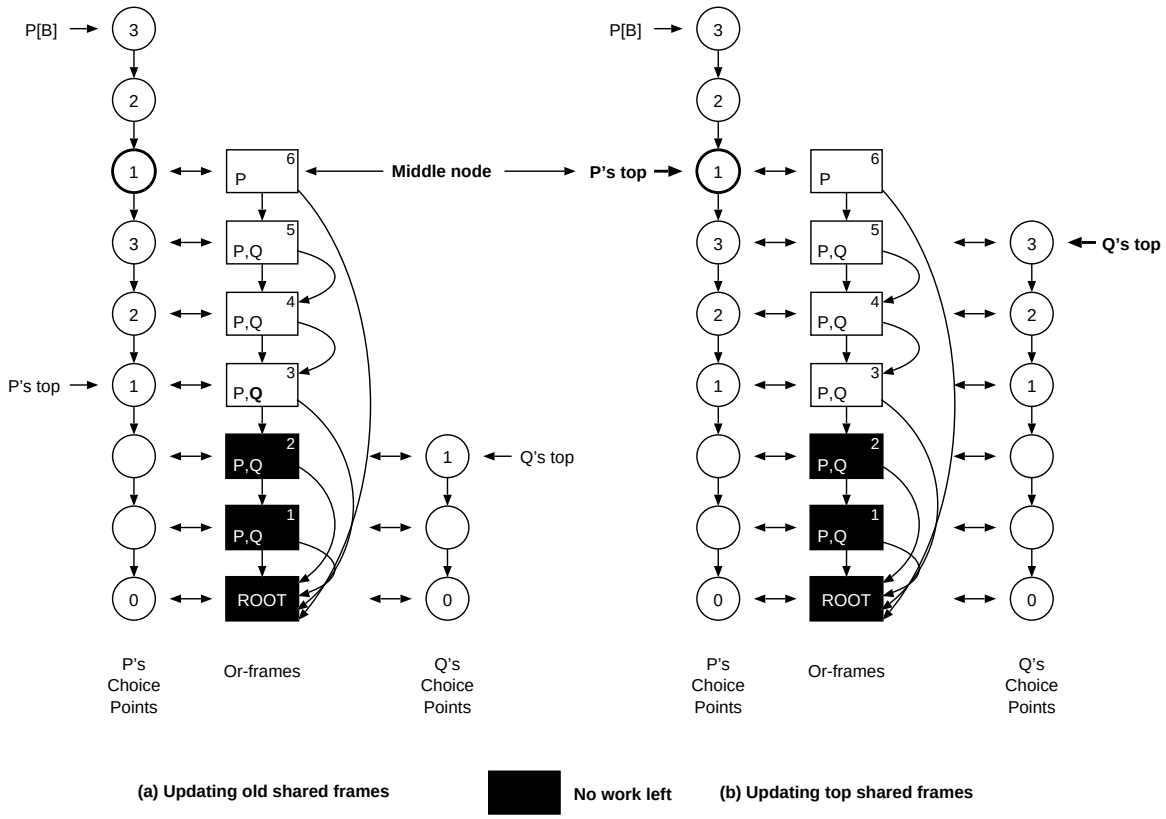

Figure 4.20: Half splitting stages 4 and 5.

Both workers are now setup to continue execution according to the half splitting scheme. Q continues execution with the work available on the assigned shared or-frames with global depth values 5, 4 and 3. P continues execution with its private nodes and with the shared frame with global depth 6.

A final example of a sharing sequence with three consecutive half splitting share procedures is shown in Figure 4.21. In the same order as illustrated, `W1` first shares four private nodes with `W2`, claiming four or five of its private nodes. Next, `W1` shares two private nodes with `W3`, claiming the remaining two or three nodes. Afterwards, `W2` shares three nodes with `W4` and claims the remaining three or four nodes.

4.4 Chapter Summary

This chapter has provided a detailed description of the implementation of the vertical and half stack splitting strategies. We first introduced initial concepts and sharing

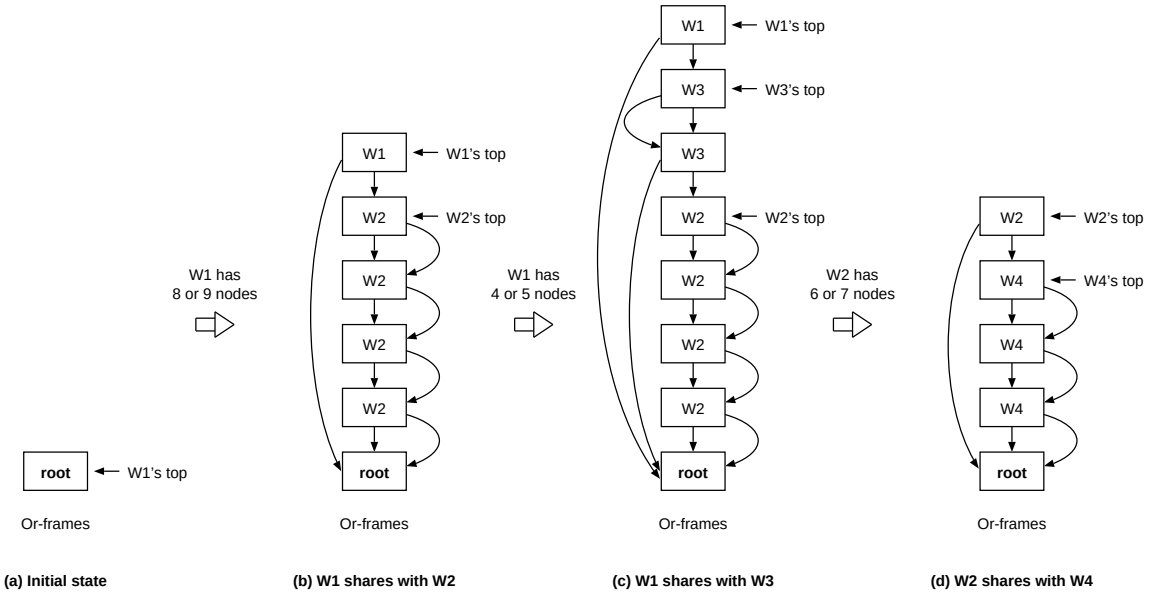

Figure 4.21: Work chaining sequence of or-frames in half splitting.

stages, and then described the implementation details necessary to extend these stages to correctly and efficiently implement the vertical and half splitting schemes.

The next chapter discusses the extension of these two schemes to support the incremental copy technique.

Chapter 5

Supporting Stack Splitting with Incremental Copy

This chapter describes the concepts and mechanisms necessary to extend the vertical and half splitting schemes to support the incremental copy technique.

5.1 General Ideas

The *incremental copy* technique is one of the most important attributes in YapOr's current implementation. The efficiency of an or-parallel Prolog system based on stack copying is highly dependent on the amount of the execution stacks that are needed to be copied from a sharing worker to a requesting worker. With incremental copy, the requesting worker is placed in a common consistent state with the sharing worker before copying and the difference between them corresponds to the stack portions to be copied from the sharing worker. In order to support this technique with the stack splitting schemes, there are several implementation issues that need to be reviewed. We next discuss in detail such modifications.

In functional terms, both stack splitting models, with or without incremental copy, work by copying a limited amount of the execution stacks. However, with incremental copy only the difference between workers is claimed as part of the copy. This behavior is caused by the requesting worker's placement when establishing a work request.

With incremental copy, whenever there is no more available work in a worker's frame set, instead of being placed in the root or-frame position as before, the worker is

placed at the choice point corresponding to the or-frame that links through the `OrFr_nearest_livenode` field to the root or-frame. Due to this placement, the requesting worker can be maintained closer to the sharing worker in the search tree and thus minimize the total amount of stack copying needed to put the two workers in the same computational state. These structural differences are illustrated in Figures 5.1 and 5.2 for the vertical splitting and half splitting schemes, respectively.

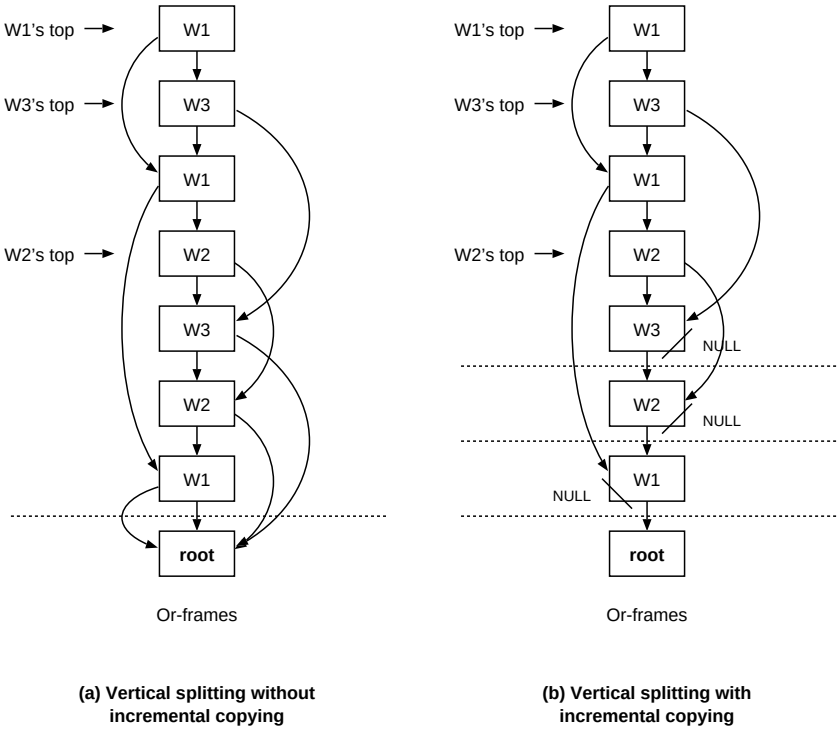

Figure 5.1: Structural differences for vertical splitting (a) without and (b) with incremental copy.

The basic idea is that the final `OrFr_nearest_livenode` connection, in a worker's frame set, is replaced by the dead-end `NULL` value. For example, consider the worker `W3` in Figure 5.2, which is assigned with two or-frames with available work. After executing all alternatives in the first or-frame, the worker moves to the choice point assigned to the second or-frame. Then, when the available work in the second frame is fully explored, `W3` stops execution, as it reached a dead-ending or-frame, and its current state of execution is maintained in the stacks.

Every time a worker reaches the last frame in its computation path and there are no more unexplored alternatives, the worker tries to find new work from busy workers. Consider again worker `W3` in Figure 5.2 after exploring all available work. On one hand, if it asks for work from worker `W1`, it can keep the stacks corresponding to the

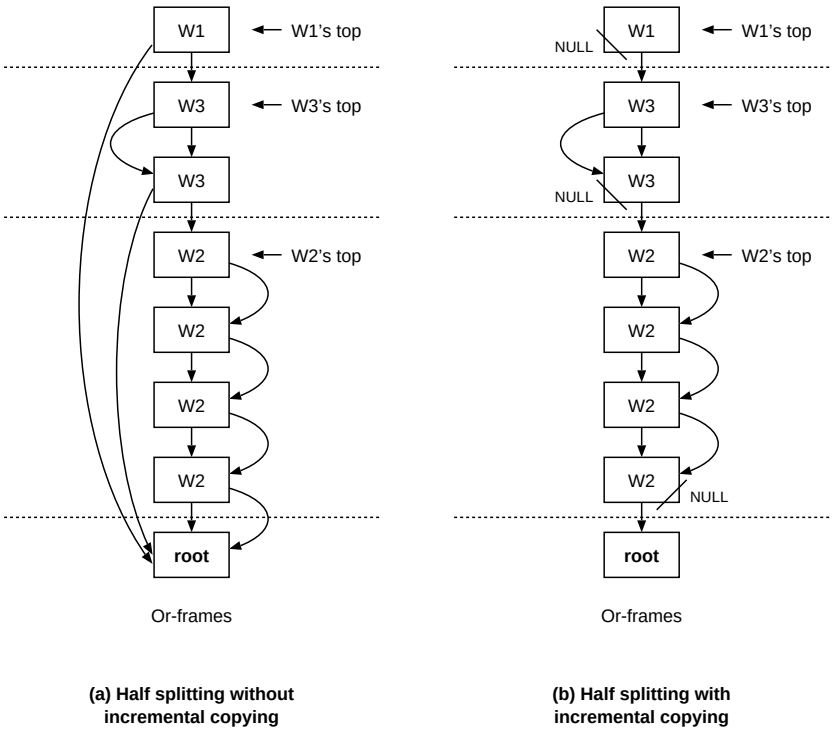

Figure 5.2: Structural differences for half splitting (a) without and (b) with incremental copy.

first five or-frames and only copy the differences to worker `W1`. On the other hand, if it moves up in the tree and backtracks to `W2`'s frame set, it can ask `W2` for available work by requesting work situated in the same execution path as `W2` and thus, no stack copying may be needed.

In stack splitting, two different workers can never be executing in the same choice points since there is a full separation of the work execution paths of each worker. This division of choice points is a great advantage, as it reduces need for synchronization and also allows for efficient copying. This is a crucial factor for system performance.

5.2 Supporting Incremental Copy

We now introduce the practical aspects for implementing stack splitting with support for the incremental copy technique.

5.2.1 Sharing Without Copying the Stacks

First, we discuss the situations where a requesting worker Q does not need to copy any stack segments of the sharing worker P in order to get new work. This may happen when the top (youngest) or-frame of Q is equal or older than the current top or-frame of P. Remember that, for Q to perform a sharing request to P, Q must first backtrack to the youngest choice point common with P, which, in particular, can be the current top choice point of P. Figures 5.3 and 5.4 show two different situations, for vertical and half splitting respectively, where sharing is done without copying the stacks.

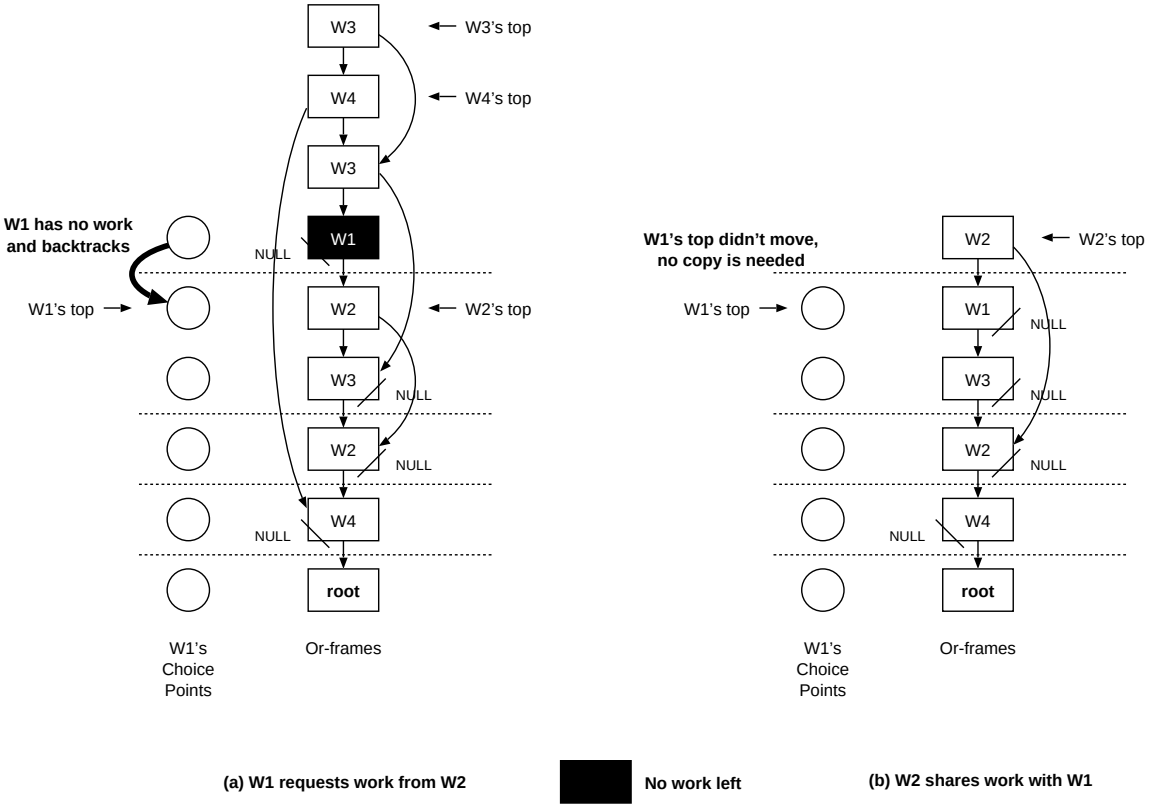

Figure 5.3: Vertical splitting without copying the stacks.

Figure 5.3 shows a worker `W1`, with no work left in its assigned path, looking for a busy worker `W2` above its current position. As worker `W2` is above worker `W1`, `W1` has to backtrack in such a way that the top or-frames of both workers coincide. After `W2` has shared its work with `W1`, `W1`'s top or-frame after the sharing procedure remains the same as the top or-frame before the sharing procedure and thus, in such case, no stack copying is needed.

Figure 5.4 illustrates a similar situation for the half splitting scheme, but here the top

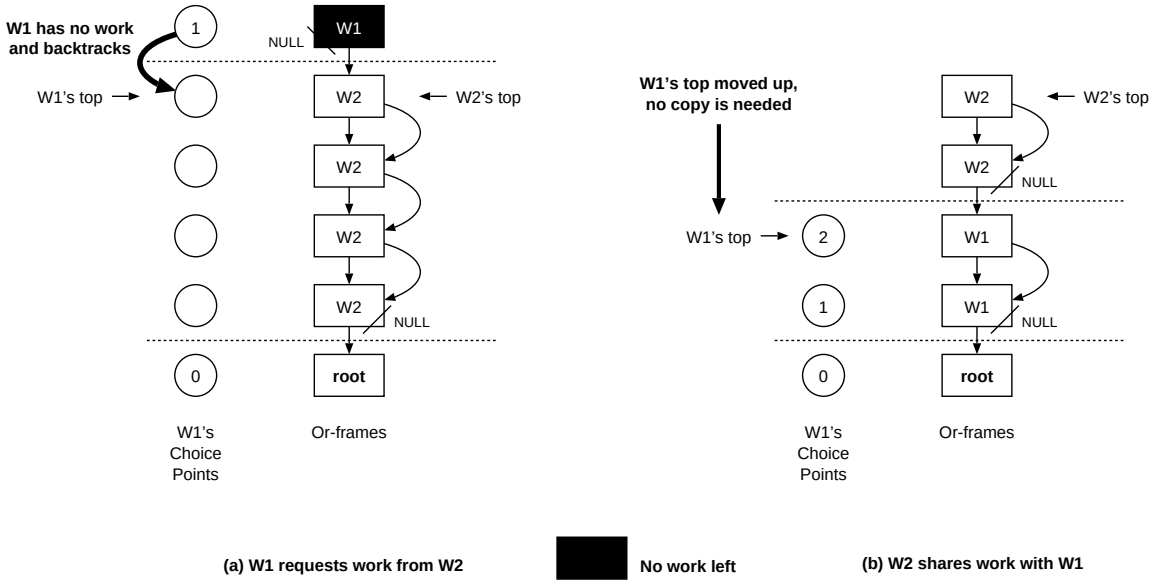

Figure 5.4: Half splitting without copying the stacks.

or-frame assigned after the sharing procedure corresponds to a choice point younger than the one for the top or-frame before sharing. In such situation, the requesting worker `W1` only needs to move up in the search tree in order to be consistent with the new top choice point assigned by the sharing procedure. By doing that, no stack copying is needed and `W1` only has to synchronize its stacks with `W2` by passing through the *dereference phase* and *installation phase*.

As we will see next, for the vertical splitting scheme an *unbitmapping* of the backtracked frames is still needed, and for the half splitting scheme a *split counter checking* may also be needed for the backtracked frames.

5.2.2 Unbitmapping

As mentioned above, when a requesting worker Q obtains work from a sharing worker P without copying the stacks, it still has to move up in the tree in order to reach the new work assigned to it. In this movement, we may have to update the membership information for the or-frames corresponding to the backtracked path. We named this procedure as *unbitmapping*.

The unbitmapping procedure traverses the or-frames starting from the requesting worker's top or-frame before the sharing procedure until reaching the requesting worker's top or-frame after the sharing procedure, and removes the requesting worker

from the bitmap field for such or-frames. This procedure is only applied with the vertical splitting scheme and was embedded in the **updating top or-frames** stage.

5.2.3 Copy Ranges Definition

In order to correctly copy the stacks from the sharing worker P to the requesting worker Q, we need to define the copy range, i.e., the *starting point* and *ending point*, for the stack segments to be copied. As Figure 5.5 shows, there are three important stack pointers that are crucial for determining the copy ranges: **(i)** P's top after sharing; **(ii)** Q's top before sharing; and **(iii)** Q's top after sharing.

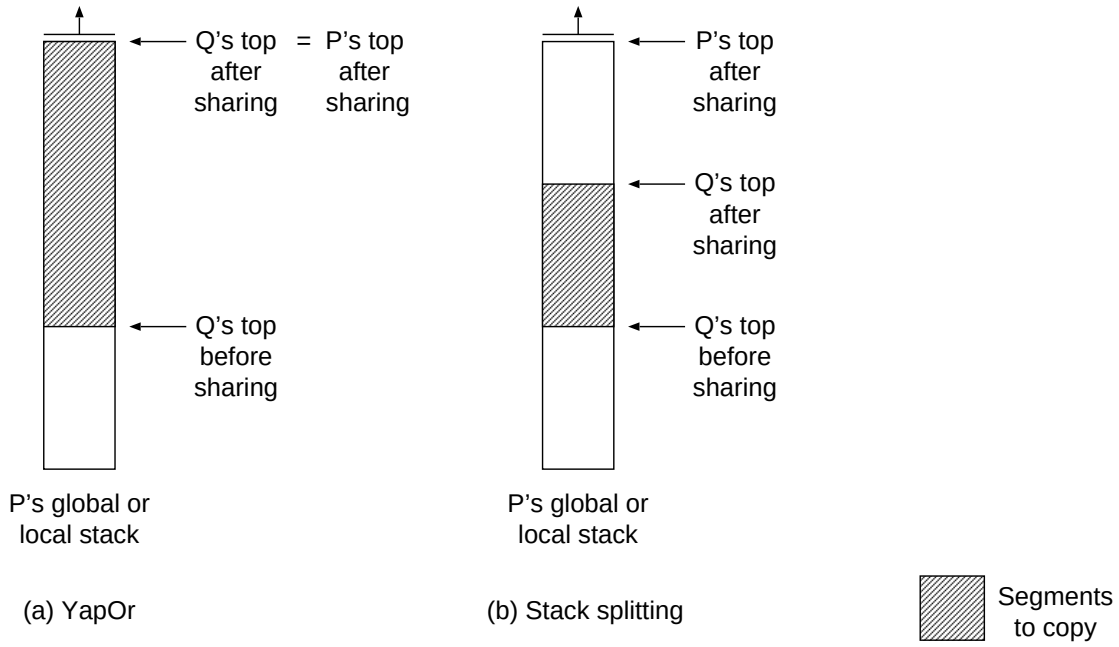

Figure 5.5: Copy ranges in YapOr and in stack splitting.

In YapOr, the incremental copy process includes copying everything in P's stack segments that Q doesn't have. With stack splitting, we only need to copy the interval between Q's top before and after sharing for the global and local stacks. For the trail stack, the process is similar to YapOr's implementation and the same interval of the trail stack is copied.

In YapOr, the copy ranges can be defined before starting the work sharing procedure since P's current state will be fully shared with Q. For stack splitting, these ranges can only be determined after traversing some of P's choice points.

Figure 5.6 shows the stack segments to be copied for our stack splitting implementation with the incremental copy technique. Note that some of the copy ranges can be determined before starting the work sharing procedure, such as:

```
start_global = Q[old_top_node->cp_h]
end_local = Q[old_top_node]
start_trail = P[TR]
end_trail = Q[old_top_node->cp_tr]
```

The other two ranges:

```
end_global = Q[new_top_node->cp_h]
start_local = Q[new_top_node]
```

can only be determined after the `new_top_node` is known. In the vertical splitting scheme, the top node can be known in the sharing loop stage, if there are choice points in P's private region local stack, assigning the `new_top_node` with the second choice point in P's choice point set (`P[B->cp_b]`). If there are no private work to share, then the `new_top_node` is assigned with the choice point that corresponds to the or-frame pointed by `OrFr_nearest_livenode(P[old_top_node->cp_or_fr])`.

For the half splitting implementation, the `new_top_node` is always assigned with the choice point denoted by `P[middle_node->cp_b]`. If there is private work to share, then the `new_top_node` in known in the **sharing loop** stage, otherwise, occurs before the **updating old shared frames** stage and assigns the `new_top_node` with `P[old_top_node->cp_b]`. Note that for the trail, it is mandatory to copy the interval between `P[TR]` and `P[B->cp_tr]` in order to implement a new phase, named the *dereference phase*, necessary to correctly support stack splitting with incremental copy, as it will be explained in the next subsection.

5.2.4 Dereference Phase

According to YapOr's implementation, after copying the stack segments between the worker P and the worker Q, P continues its execution while Q starts the *installation phase*. Since the stack splitting work sharing process does not fully copies the stack segments of P, the installation phase of the variables in the trail may be not enough to

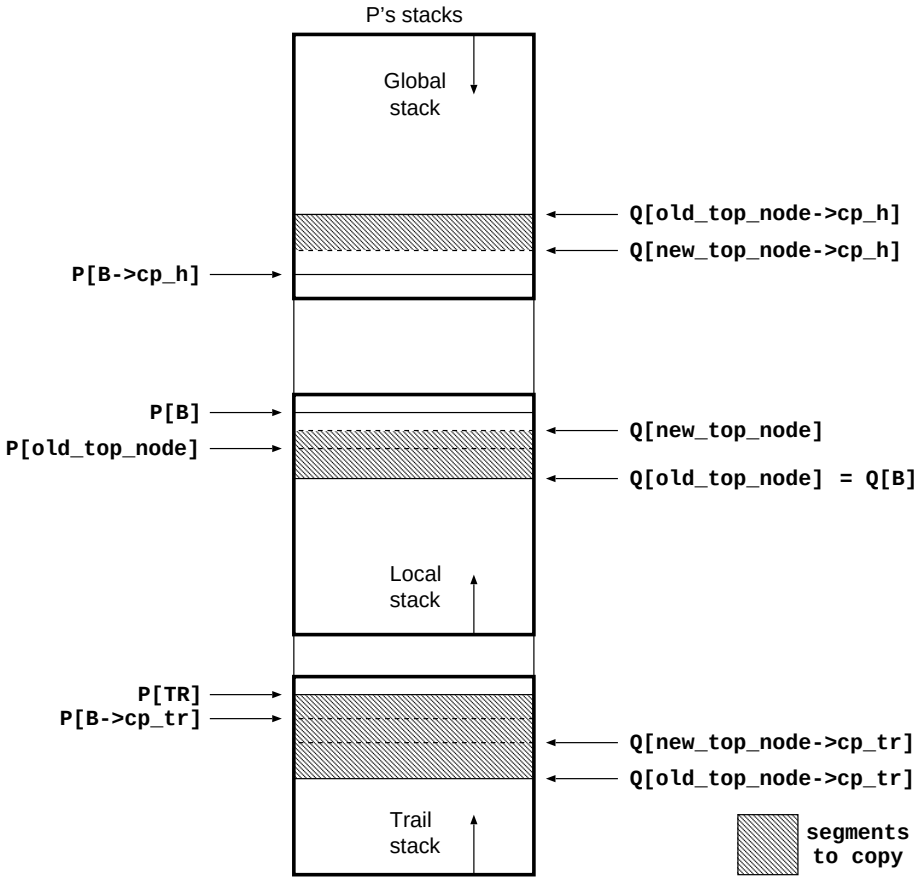

Figure 5.6: Stack segments to copy for stack splitting with incremental copy.

correctly setup Q's stacks and a new phase, called *dereference phase*, must come first than the installation phase. This is necessary in order to avoid the possibility of Q have incorrectly bounded variables in the copied segments. This may happen when P has instantiated variables belonging to the copied segments, i.e., in the execution path between `Q[new_top_node]` and `Q[old_top_node]`, that where bound in the execution path not copied to Q, i.e., between `P[B]` and `Q[new_top_node]`.

The dereference procedure traverses the trail from `P[TR]` to `Q[new_top_node->cp_tr]` looking for references to variables in the copied segments of the global and local stacks. If such a variable is found then the variable is dereferenced and becomes a free variable with no value assigned. Figure 5.7 illustrates a situation that shows why the dereference phase is necessary to correctly setup Q's stacks.

Starting from Q's assigned top choice point, **CP4**, notice how some variables in Q's global stack are not consistent with the computational state corresponding to the **CP4** choice point. One of them is variable **D** which was a free variable before **CP4** creation

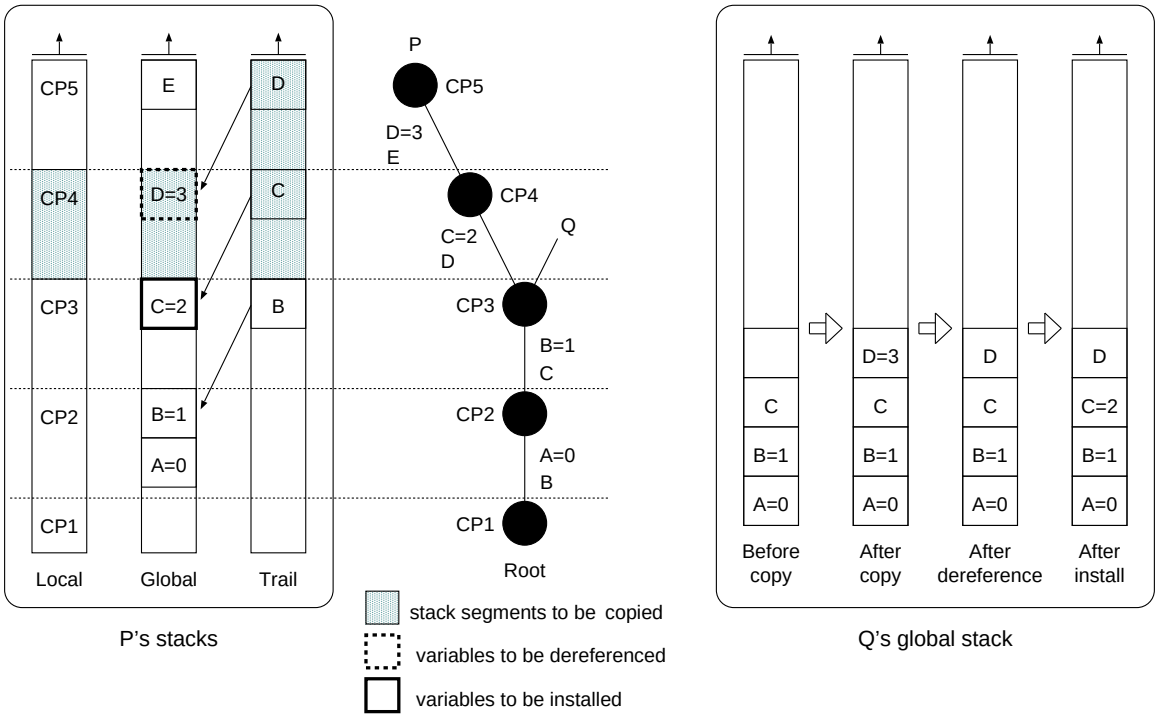

Figure 5.7: Dereference phase.

and is bound with the value three in Q's global stack after copying. This happens because **D** was instantiated by P only after **CP4** creation. The reference to **D** in the trail after **CP4** creation confirms such behavior. Thus, after the copying phase, the dereferencing procedure operates in order to reset such incorrectly bound variables in the copied stack segments.

5.2.5 Split Counter Checking Phase

Here, we introduce the *split counter checking phase* that is used to install the correct split counter values in the older choice points belonging to a requesting worker Q. This mechanism is done by the requesting worker Q after the work sharing and copying procedures. This checking phase is necessary in order to avoid incoherent values in the split counter `cp_sc` fields for the choice points, in the requesting worker Q, not copied from P. We can say that such incoherence is caused by the independent work sharing operations with different workers that make the common (not copied) stack segments of P and Q, namely the local stack's choice points `cp_sc` fields, to be inconsistent in Q. Notice also that when Q's new top choice point is younger than the old top choice point, no copy is done, which can also lead to inconsistencies in the `cp_sc` values of

Q's choice points.

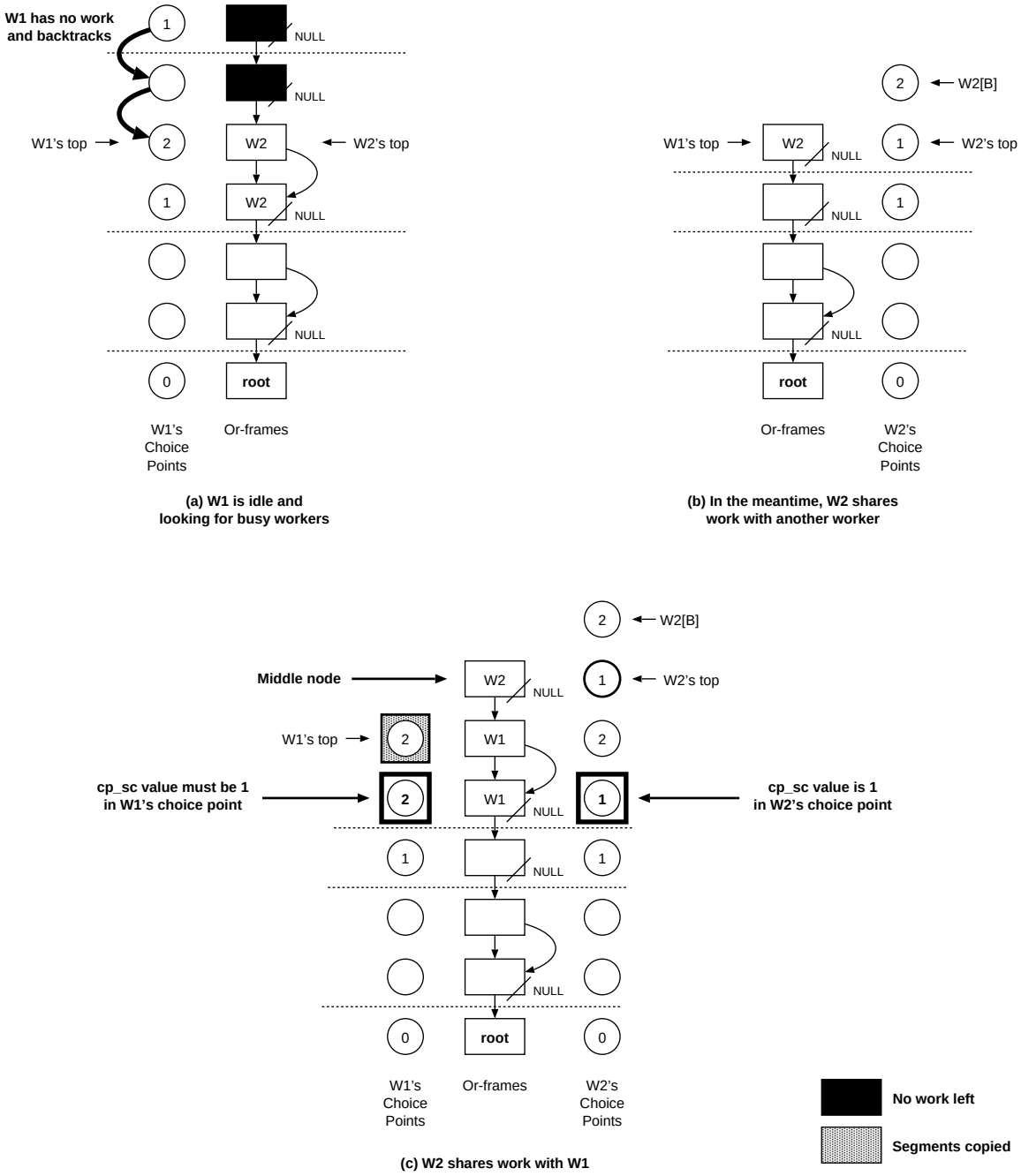

Figure 5.8: Split counter checking phase.

Figure 5.8 shows an example where the split counter checking phase is necessary. Such situation starts with an idle worker `W1` moving up in the tree until reaching a busy worker `W2`. By doing that, `W1` places itself in the same top choice point as `W2`. However, in the meantime, `W2` shared part of its work with another worker, which

lead to differences in the split counter values in the choice points of W1 and W2 (see Figure 5.8(b)).

Next, W1 requests and gets work from W2 (see Figure 5.8(c)), but W1's choice points not copied from W2 have wrong split counter values. W1 accesses W2's local stack and retrieves one value, from the same choice point that corresponds to W1's new top choice point, that is correct. After this, it is followed the update of the split count values of the older choice points in W1's local stack. The split counter checking phase is thus used to install the correct split counter values in the older choice points belonging to the requesting worker W1.

5.3 Chapter Summary

This chapter presented the most important implementation details for supporting stack splitting with incremental copy. We discussed the copy range definition of the stack segments to be copied, the mandatory dereference phase necessary to unbound the incorrectly bound variables in the copied segments of the requesting worker and, finally, the situations where the split counter mechanism may need a checking phase.

In the next chapter, we present experimental results for both stack splitting schemes and for that we use a set of benchmark programs widely used to assess the performance of or-parallel Prolog systems.

Chapter 6

Performance Analysis

In this chapter, we evaluate the performance of our two stack splitting schemes when using a set of well-known benchmarks widely used to evaluate or-parallel Prolog systems, and we make a comparison between the vertical splitting, half splitting and YapOr's implementation based on environment copying.

As parallel platform for our experiments, we used a machine with four AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores in total) and 64 (4x16) GB of DDR-2 667MHz RAM, running GNU/Linux (kernel 2.6.31.5-127 64 bits). The machine was running in multi-user mode, but no other users were using the machine. Each benchmark was executed twenty times and the results presented next are the average of those twenty executions.

6.1 Benchmark Programs

The details for the set of benchmark programs used in our experiments are presented in appendix B. Here, we just give a briefly description:

- **cubes7**. A program that consists of stacking 7 colored cubes in a column in such a way that no color appears twice in the same column for each given side.
- **ham**. A program for finding all the Hamiltonian cycles in a graph with 26 nodes, with each node connected to 3 other nodes.
- **magic**. A program to solve the Rubik's magic cube problem.

- **map**. A program for solving the problem of coloring a map of 10 countries with five colors in such a way that all two adjacent countries have different colors.

- **nsortN**. A program for ordering a list of N elements using a naive algorithm and starting with the list inverted.

- **puzzle**. A program that solves a version of the sudoku problem where the diagonals must add up to the same amount.

- **puzzle4x4**. A program that solves a maze problem in a 4x4 grid by moving an empty square.

- **queens13**. A program to solve the 13-queens problem that analyzes the board state at every step.

All the benchmark programs find all the solutions for the given problem by simulating an automatic failure whenever a new solution is found. Some of programs have search trees with depths that, in some cases, are over the value 60. Others have depths that not exceed the value 10, but can be quite extensive in their branching numbers, generating a very wide search tree.

6.2 Performance Results

Next, we show the performance results for the stack splitting implementation in the YapOr system. We start by measuring the cost of the parallel model over the sequential system. Then, we evaluate the behavior of the vertical splitting and the half splitting implementations with and without the incremental copy technique and compare with YapOr's environment copying model. For shaping a fair comparison among all implementations, we considered not only the base execution times with 1 worker for each strategy, but also the base execution times of the sequential implementation.

All systems were compiled with the same configuration parameters and using the same compiler's back-end architecture. To measure the execution time, we took advantage of YapOr's timing support and we used it in all models in the same way.

6.2.1 Cost of the Parallel Model

Table 6.1 presents the execution times, in seconds, for the set of benchmark programs, when using the sequential version of the Yap system and when using the several YapOr parallel versions with one worker. Table 6.2 presents the corresponding ratios that show the cost of supporting the parallel models when not taking advantage of them, since we are executing with a single worker. The last row in Table 6.2 presents the average of all execution times for each version.

Table 6.1: Execution times, in seconds, for Yap's sequential model and for YapOr's implementation based on environment copying (EC), on vertical splitting not using (VS) and using incremental copy (VS+IC), and on half splitting not using (HS) and using incremental copy (HS+IC), all running with a single worker.

Programs	**Yap**	**YapOr with 1 worker**				
		EC	**VS**	**VS+IC**	**HS**	**HS+IC**
cubes7	0.202	0.211	0.211	0.210	0.213	0.214
ham	0.321	0.385	0.386	0.384	0.252	0.352
magic	45.990	45.322	45.435	45.358	41.149	41.431
map	22.434	25.352	25.359	25.358	25.803	25.591
nsort10	2.567	2.928	2.885	2.948	2.669	2.670
nsort11	28.239	32.063	31.336	32.008	29.432	29.035
nsort12	339.406	382.037	385.050	383.204	341.111	340.400
puzzle	0.154	0.177	0.177	0.177	0.173	0.170
puzzle4x4	9.875	10.187	10.168	10.173	9.434	9.462
queens13	48.220	51.162	51.099	51.272	48.180	48.277
Σ	497.408	549.824	552.105	551.093	498.516	497.601

By observing the results on Tables 6.1 and 6.2, we can say that, for these set of benchmark programs, YapOr's vertical splitting scheme has on average an overhead of 9,7% without incremental copy and 10.1% with incremental copy over Yap's sequential implementation. Also, YapOr's half splitting scheme has on average an overhead of 3.7% without incremental copy and 3.3% with incremental copy over Yap's sequential implementation, which is the best result among the five models. Notice that YapOr's implementation based on environment copying has on average an overhead of 10.0%. This cost is identical to the cost observed previously for YapOr's implementation based

Table 6.2: Ratios showing the cost of YapOr's parallel models, running with a single worker, in comparison with Yap's sequential model.

Programs	EC / Yap	VS / Yap	VS+IC / Yap	HS / Yap	HS+IC / Yap
cubes7	1.044	1.044	1.038	1.056	1.059
ham	1.198	1.203	1.197	1.101	1.098
magic	0.985	0.988	0.986	0.895	0.901
map	1.130	1.130	1.130	1.150	1.141
nsort10	1.140	1.124	1.149	1.040	1.040
nsort11	1.135	1.110	1.133	1.042	1.028
nsort12	1.126	1.134	1.129	1.005	1.003
puzzle	1.152	1.149	1.151	1.124	1.106
puzzle4x4	1.032	1.030	1.030	0.955	0.958
queens13	1.061	1.060	1.063	0.999	1.001
Average	1.100	1.097	1.101	1.037	1.033

on environment copying [16]. In general, for all models, YapOr overheads result from handling the work load register and from testing operations that (i) verify whether the youngest node is shared or private, (ii) check for sharing requests, and (iii) check for backtracking messages due to cut operations.

6.2.2 Parallel Execution

To assess the performance of the or-parallel models, we ran YapOr with a varying member of workers for our set of benchmark programs, and we show the obtained *speedups*. For the speedups we used the obtained execution times and compare them (i) against the corresponding execution times with one worker, which reflects the improvement in execution time for each parallel implementation independently, and (ii) against the execution times for the sequential implementation, reflecting the general improvement in execution time starting from the sequential implementation. These second results give a more fair comparison between the parallel implementations since, when an obtained value is considered to be the best speedup value among all implementations, it shows that it has the fastest execution time values.

Tables 6.3 to 6.7 show the obtained speedups for each of the five YapOr's models. The execution times are in appendix A. For each entry in these tables we show the speedup against the case with 1 worker and, in parenthesis, against the sequential execution time. Also, in all cases, the best results among all implementations are marked with a gray background color. For each number of workers, at the end, we also show the *average* and *efficiency* calculation of all speedups.

Table 6.3: Speedups against the 1 worker case and against the sequential execution (in parenthesis) for YapOr's implementation based on environment copying.

Programs	**Workers**					
	1	**2**	**4**	**8**	**16**	**24**
cubes7	(0.958)	1.917(1.836)	3.411(3.267)	5.908 (5.659)	7.960 (7.624)	7.760 (7.432)
ham	(0.834)	1.968(1.642)	3.712(3.098)	6.405(5.344)	8.768 (7.316)	7.775 (6.487)
magic	(1.015)	2.009(2.039)	3.994(4.053)	7.959 (8.077)	15.850 (16.083)	23.598 (23.945)
map	(0.885)	2.036(1.802)	4.047(3.581)	8.029(7.105)	15.731(13.920)	22.965(20.322)
nsort10	(0.877)	2.104(1.845)	4.116 (3.609)	8.070 (7.076)	15.333 (13.444)	20.488 (17.965)
nsort11	(0.881)	2.125 (1.871)	4.216 (3.714)	8.369 (7.371)	16.611 (14.630)	24.560 (21.631)
nsort12	(0.888)	2.054(1.825)	4.145(3.683)	8.318(7.390)	16.763(14.893)	24.973(22.186)
puzzle	(0.868)	1.971(1.711)	3.415 (2.963)	5.395 (4.682)	6.846 (5.942)	5.793 (5.027)
puzzle4x4	(0.969)	2.010(1.948)	4.027 (3.903)	8.016 (7.770)	15.803 (15.319)	23.148 (22.438)
queens13	(0.943)	1.999(1.884)	3.991 (3.761)	7.953 (7.496)	15.841 (14.930)	23.582 (22.226)
Average	(0.912)	2.019(1.840)	3.907 (3.563)	7.442 (6.797)	13.551 (12.410)	18.464 (16.966)
Efficiency	(0.912)	1.010(0.920)	0.977 (0.891)	0.930 (0.850)	0.847 (0.776)	0.769 (0.707)

From table 6.3 we can see how the YapOr's implementation based on environment copying compares with the new stack splitting approaches. Each gray background entry illustrates the cases where environment copying is not surpassed by any stack splitting approach, while the remaining entries correspond to cases where the results obtained with one of the stack splitting approach is better.

In general, we can observed that stack splitting obtains better results for the cases with a smaller number of workers and that environment copying seems to perform better, on average, for the cases with 16 and 24 workers. In any case, if considering the speedups against the sequential execution times, for the 10 programs in analysis, environment copying only obtains better results in 4 and 5 programs for 16 and 24 workers, respectively.

Tables 6.4 and 6.5 illustrate the speedup results for the vertical splitting implementation without and with the incremental copy technique, respectively.

Table 6.4: Speedups against the 1 worker case and against the sequential execution (in parenthesis) for YapOr's vertical splitting implementation without incremental copy.

Programs	Workers					
	1	**2**	**4**	**8**	**16**	**24**
cubes7	(0.958)	1.758(1.684)	2.742(2.626)	3.488(3.341)	3.131(2.999)	2.519(2.413)
ham	(0.831)	1.867(1.551)	2.875(2.389)	3.862(3.210)	3.959(3.290)	3.532(2.935)
magic	(1.012)	2.011(2.036)	3.988(4.037)	7.907(8.004)	15.606(15.797)	22.828(23.107)
map	(0.885)	2.037(1.802)	4.047(3.581)	7.963(7.045)	15.362(13.590)	22.067(19.522)
nsort10	(0.890)	2.067(1.839)	3.953(3.518)	7.329(6.521)	11.025(9.811)	11.702(10.413)
nsort11	(0.901)	2.070(1.865)	4.121(3.714)	8.121(7.318)	15.639(14.094)	22.116(19.931)
nsort12	(0.881)	2.075 (1.829)	4.181 (3.686)	8.382 (7.388)	16.851 (14.853)	25.018 (22.052)
puzzle	(0.870)	1.770(1.540)	2.256(1.963)	2.158(1.878)	1.814(1.578)	1.396(1.214)
puzzle4x4	(0.971)	2.002(1.944)	3.990(3.875)	7.851(7.625)	15.051(14.617)	21.024(20.418)
queens13	(0.944)	1.991(1.879)	3.949(3.726)	7.803(7.364)	15.076(14.227)	21.767(20.540)
Average	(0.914)	1.965(1.797)	3.610(3.311)	6.487(5.969)	11.351(10.486)	15.397(14.255)
Efficiency	(0.914)	0.982(0.898)	0.903(0.828)	0.811(0.746)	0.709(0.655)	0.642(0.594)

Table 6.5: Speedups against the 1 worker case and against the sequential execution (in parenthesis) for YapOr's vertical splitting implementation with incremental copy.

Programs	Workers					
	1	**2**	**4**	**8**	**16**	**24**
cubes7	(0.963)	1.924 (1.853)	3.454 (3.327)	5.733(5.521)	7.252(6.984)	6.284(6.052)
ham	(0.835)	2.002 (1.673)	3.717 (3.106)	6.415 (5.359)	8.375(6.996)	5.980(4.996)
magic	(1.014)	2.007(2.035)	3.989(4.044)	7.954(8.065)	15.817(16.037)	23.465(23.792)
map	(0.885)	2.041(1.805)	4.058 (3.590)	8.055 (7.126)	15.777 (13.958)	23.017 (20.363)
nsort10	(0.871)	2.110 (1.837)	4.107(3.575)	8.039(6.999)	15.121(13.165)	20.165(17.557)
nsort11	(0.882)	2.106(1.858)	4.152(3.663)	8.223(7.255)	16.243(14.331)	23.987(21.162)
nsort12	(0.886)	2.043(1.809)	4.095(3.627)	8.207(7.269)	16.480(14.597)	24.576(21.767)
puzzle	(0.869)	1.983 (1.723)	3.376(2.933)	5.204(4.522)	6.015(5.226)	4.912(4.268)
puzzle4x4	(0.971)	2.011 (1.952)	4.016(3.899)	7.990(7.756)	15.786(15.324)	23.141(22.463)
queens13	(0.940)	2.002 (1.882)	3.985(3.748)	7.932(7.460)	15.699(14.765)	23.323(21.934)
Average	(0.912)	2.023 (1.843)	3.895(3.551)	7.375(6.733)	13.256(12.138)	17.885(16.435)
Efficiency	(0.912)	1.011 (0.921)	0.974(0.888)	0.922(0.842)	0.829(0.759)	0.745(0.685)

In general, the overall performance of vertical splitting with incremental copy is quite close to the performance of the original YapOr with environment copying. By analyzing the speedups in both tables, it is clear the improvement obtained with the

incremental copy technique in the vertical splitting implementation. On terms of average, the difference is noticeable in all worker cases. For example, for 8, 16 and 24 workers, the speedup gain is 0.764 (from 5.969 without incremental copy to 6.733 with incremental copy), 1.652 (from 10.486 to 12.138) and 2.180 (from 14.255 to 16.435), respectively, which shows a clear positive tendency as the number of workers increases.

The only exception seems to be the **nsort12** program. Note that, for the **nsort11** program, the speedup gain already shows a huge reduction (from 19.931 without incremental copy to 21.162 with incremental copy for 24 workers), when compared with **nsort10**, where the speedup gain is clear (from 10.413 to 17.557 for 24 workers). We believe that this behavior is related to the balance between the overhead of copying unneeded stack segments, as happens without incremental copy, against the overhead of executing the dereference and installation phases, as necessary with incremental copy. In this particular case, it seems that the percentage of saving for using incremental copy and thus not copy the full set of stacks, starts to be considerable for the **nsort10** program, but then as we increment the size of the program, this percentage becomes less significant for the **nsort11** program and for the **nsort12** it seems to be irrelevant, making the overhead of executing the dereference and installation phases a potential cost.

Finally, we present in Tables 6.6 and 6.7, the obtained speedup results for the half splitting implementation without and with the incremental copy technique, respectively.

By analyzing the speedups in both tables, the improvements obtained with the incremental copy technique in the half splitting implementation are clear. On terms of average, the difference is overwhelming in all worker cases. For example, for 8, 16 and 24 workers, the speedup gain is 2.138 (from 4.542 without incremental copy to 6.680 with incremental copy), 4.132 (from 7.193 to 11.325) and 6.116 (from 8.827 to 14.943), respectively, which shows again a clear positive tendency as the number of workers increases. We believe that these good results with incremental copy are also related to the percentage of saving achieved for not copy the full set of stacks. This advantage is more clear in the case of halt splitting since, by splitting the search tree in two halves and by sharing the older half, it reduces the stacks segments to be shared and thus to be copied, which augments the potential percentage of common stack segments that do not need to be copied.

Comparing to vertical splitting, on average, the overall performance of half splitting with incremental copy is not so close to the performance of the original YapOr with

Table 6.6: Speedups against the 1 worker case and against the sequential execution (in parenthesis) for YapOr's half splitting implementation without incremental copy.

Programs	Workers					
	1	**2**	**4**	**8**	**16**	**24**
cubes7	(0.947)	0.598(0.566)	0.745(0.705)	0.811(0.769)	0.619(0.586)	0.429(0.406)
ham	(0.908)	1.324(1.202)	1.672(1.519)	2.036(1.849)	2.090(1.898)	1.790(1.626)
magic	(1.118)	1.970(2.201)	3.715(4.153)	6.840(7.644)	11.938(13.342)	14.719(16.451)
map	(0.869)	1.097(0.954)	1.973(1.716)	2.860(2.487)	2.963(2.576)	2.628(2.285)
nsort10	(0.962)	1.827(1.757)	3.401(3.270)	5.985(5.756)	8.785(8.448)	9.307(8.950)
nsort11	(0.959)	1.962(1.883)	3.848(3.692)	7.493(7.189)	13.722(13.166)	19.322(18.539)
nsort12	(0.995)	1.896(1.886)	3.782(3.763)	7.505(7.468)	14.751(14.678)	21.290(21.183)
puzzle	(0.890)	1.522(1.355)	1.821(1.621)	1.983(1.765)	1.778(1.582)	1.428(1.270)
puzzle4x4	(1.047)	1.924(2.014)	3.659(3.830)	6.516(6.821)	10.908(11.418)	12.985(13.592)
queens13	(1.001)	1.674(1.675)	2.657(2.660)	3.673(3.676)	4.233(4.236)	3.962(3.965)
Average	(0.970)	1.579(1.549)	2.727(2.693)	4.570(4.542)	7.179(7.193)	8.786(8.827)
Efficiency	(0.970)	0.790(0.775)	0.682(0.673)	0.571(0.568)	0.449(0.450)	0.366(0.368)

Table 6.7: Speedups against the 1 worker case and against the sequential execution (in parenthesis) for YapOr's half splitting implementation with incremental copy.

Programs	Workers					
	1	**2**	**4**	**8**	**16**	**24**
cubes7	(0.944)	1.886(1.781)	3.211(3.033)	4.948(4.672)	5.485(5.179)	3.866(3.651)
ham	(0.911)	2.000(1.822)	3.539(3.224)	5.735(5.224)	6.099(5.557)	4.447(4.051)
magic	(1.110)	2.036 (2.260)	4.001 (4.442)	7.923(8.795)	15.713(17.442)	23.299(25.863)
map	(0.877)	2.167 (1.900)	3.821(3.350)	6.117(5.363)	6.724(5.894)	5.541(4.857)
nsort10	(0.961)	1.968(1.892)	3.827(3.680)	7.629(7.335)	14.030(13.490)	18.631(17.913)
nsort11	(0.973)	1.948(1.895)	3.882(3.776)	7.791(7.577)	15.321(14.901)	22.677(22.056)
nsort12	(0.997)	1.894(1.889)	3.799(3.788)	7.606(7.583)	15.402(15.357)	22.827(22.761)
puzzle	(0.904)	1.974(1.785)	3.268(2.955)	4.954(4.480)	5.543(5.013)	4.933(4.461)
puzzle4x4	(1.044)	1.998(2.086)	3.961(4.134)	7.743(8.081)	14.950(15.602)	21.966(22.925)
queens13	(0.999)	1.986(1.983)	3.913(3.909)	7.697(7.688)	14.834(14.817)	20.921(20.897)
Average	(0.972)	1.986(1.929)	3.722(3.629)	6.814(6.680)	11.410(11.325)	14.911(14.943)
Efficiency	(0.972)	0.993(0.965)	0.931(0.907)	0.852(0.835)	0.713(0.708)	0.621(0.623)

environment copying. For example, the average speedups for environment copying, vertical and half splitting are, respectively, 12.410, 12.138 and 11.325 for 16 workers and 16.966, 16.435 and 14.943 for 24 workers.

On the other hand, for the 10 programs in analysis, we can observe that half splitting with incremental copying obtains the best speedup results in 9, 7, 6, 5 and 4 programs for 2, 4, 8, 16 and 24 workers, respectively. Considering all combinations of programs and workers, half splitting obtains the higher number of best results among all implementations and it owns the best average/efficiency for the cases of 2 and 4 workers.

6.3 Chapter Summary

In this chapter, we analyzed and compared the performance of the vertical and half splitting schemes in the YapOr system and for that we used a set of benchmark Prolog programs widely used to assess the performance of or-parallel Prolog systems.

Although stack splitting was initially proposed for distributed memory architectures, our results show that YapOr with the stack splitting schemes is, in general, comparable to YapOr with environment copying, obtaining in some cases better performance than with environment copying. According to the set of results obtained, we can also conclude that both stack splitting implementations, vertical and half splitting, clearly benefit from incremental copy. Globally, the results are quite encouraging as well given that in many benchmarks we achieved performances that are above a speedup of 20 on 24 cores.

Chapter 7

Conclusions and Further Work

In this chapter we give a summary of the obtained conclusions on the implementation of stack splitting model and its performance evaluation on multicore shared memory architectures. We also describe topics of future work that builds on the work presented in this thesis.

7.1 Conclusions

In this thesis, we aimed for the design and implementation of work sharing stack splitting schemes, such as vertical splitting and half splitting, in the YapOr system. YapOr is a matured Or-Parallel Prolog system based on the environment copying model. The implementation of the stack splitting model required modifications and extensions to existing data structures, re-implementation of the incremental copy technique, and prepare the code to accommodate two different splitting strategies.

Some relevant mechanisms that were implemented to ensure the stack splitting model's consistency are:

- A dereference phase to dereference the conditional bindings from variables that are not in the stack segments to be copied or should become unassigned.

- A split count checking phase that verifies the chaining of counters so that they are consistent. In the half splitting strategy, after a number of sharing operations, this counter could appear inconsistent and thus needs to be re-installed.

- An efficient unbitmapping operation and a consistent update of the top choice point are ensured during the work sharing procedure.

In chapter 6, we presented extensive results on the performance of the different strategies implemented, namely, the original YapOr with environment copying, and YapOr with vertical and half stack splitting. The results observed for the benchmarks used, allowed us to derive the following conclusions:

- Incremental copy clearly benefits performance as, in general, both stack splitting strategies show significant gains over not using incremental copy. Only in cases where the ratio between a full copy of the stacks and a partial copy, as it results from the incremental copy, does not compensate for the overheads involved, then incremental copy does not show gains.
- The performance of vertical stack splitting overall is quite close to the performance of the original YapOr.
- The half stack splitting strategy performs better in 4 of 10 benchmarks.
- The results overall on the new multicore architectures are quite encouraging given that in many benchmarks we achieve performances that are above a speedup of 20 on 24 cores.
- Although stack splitting was initially proposed envisaging a distributed memory architecture, the results show that it is equally suitable for shared memory architectures. This is a clear advantage of stack splitting over the environment copying model, since we could use it as the basis for an hybrid execution model aiming at clusters of shared memory.

7.2 Future Work

Although a major effort was made to consolidate the work presented in this thesis, there is still room for improvements. We describe next some ideas for further work that could influence positively performance results.

Improving the worker positioning for work sharing. By devising a clever strategy of worker's positioning in the search tree, possibly more at the top of the

tree, we may be able to overcome some of the inefficiencies in the work sharing observed in the vertical and half splitting strategies.

Implementation of new stack splitting strategies. Using this thesis as an introduction to the field, other schemes of stack splitting can be implemented and embedded in the work sharing procedure of YapOr. Examples of these schemes are the horizontal [8] and diagonal stack splitting [15].

Designing a strategy for clusters of multicores. Considering the success of the stack splitting model for the both types of architectures, distributed memory and shared memory, we can innovate by proposing a strategy that can exploit the mix of both architectures as it is now the normal case with clusters of multicores. This requires the design and implementation of a new system that supports two levels of work sharing/distribution using stack splitting strategies. The idea is to combine workers into teams. A team of workers should run on a shared memory machine and should behave as a normal parallel system as just described in this thesis. Different teams should be assigned to different nodes of the cluster. Work sharing between teams should follow a work stealing strategy in which a team without work has one of its workers to probe other teams for work. Whenever a team worker shares work with another team, it performs the stack splitting to divide its available work, and then sends it to the requester worker in a different team. Care must be taken in order to detect computation completion.

Appendix A

Execution Times

This appendix includes a set of tables that contains the execution times for the programs used in Chapter 6.

Environment Copying

Table A.1: Execution times, in seconds, for YapOr's implementation based on environment copying.

	Workers				
Programs	**2**	**4**	**8**	**16**	**24**
cubes7	0.110	0.062	0.036	0.026	0.027
ham	0.195	0.104	0.060	0.044	0.049
magic	22.556	11.348	5.694	2.859	1.921
map	12.451	6.264	3.158	1.612	1.104
nsort10	1.391	0.711	0.363	0.191	0.143
nsort11	15.092	7.604	3.831	1.930	1.305
nsort12	185.951	92.159	45.928	22.790	15.298
puzzle	0.090	0.052	0.033	0.026	0.031
puzzle4x4	5.068	2.530	1.271	0.645	0.440
queens13	25.594	12.820	6.433	3.230	2.169
Σ	268.499	133.653	66.806	33.353	22.488

Vertical Splitting

Table A.2: Execution times, in seconds, for YapOr's vertical splitting implementation without incremental copy.

	Workers				
Programs	**2**	**4**	**8**	**16**	**24**
cubes7	0.120	0.077	0.060	0.067	0.084
ham	0.207	0.134	0.100	0.098	0.109
magic	22.590	11.392	5.746	2.911	1.990
map	12.447	6.266	3.184	1.651	1.149
nsort10	1.396	0.730	0.394	0.262	0.247
nsort11	15.141	7.604	3.859	2.004	1.417
nsort12	185.579	92.092	45.939	22.851	15.391
puzzle	0.100	0.078	0.082	0.098	0.127
puzzle4x4	5.080	2.548	1.295	0.676	0.484
queens13	25.668	12.940	6.548	3.389	2.348
Σ	268.328	133.862	67.208	34.006	23.345

Table A.3: Execution times, in seconds, for YapOr's vertical splitting implementation with incremental copy.

	Workers				
Programs	**2**	**4**	**8**	**16**	**24**
cubes7	0.109	0.061	0.037	0.029	0.033
ham	0.192	0.103	0.060	0.046	0.064
magic	22.601	11.371	5.702	2.868	1.933
map	12.426	6.249	3.148	1.607	1.102
nsort10	1.397	0.718	0.367	0.195	0.146
nsort11	15.199	7.709	3.893	1.971	1.334
nsort12	187.604	93.582	46.692	23.252	15.593
puzzle	0.089	0.052	0.034	0.029	0.036
puzzle4x4	5.059	2.533	1.273	0.644	0.440
queens13	25.616	12.866	6.464	3.266	2.198
Σ	270.293	135.244	67.669	33.908	22.880

Half Splitting

Table A.4: Execution times, in seconds, for YapOr's half splitting implementation without incremental copy.

	Workers				
Programs	**2**	**4**	**8**	**16**	**24**
cubes7	0.357	0.286	0.263	0.345	0.498
ham	0.267	0.211	0.174	0.169	0.197
magic	20.892	11.075	6.016	3.447	2.796
map	23.516	13.076	9.022	8.709	9.817
nsort10	1.461	0.785	0.446	0.304	0.287
nsort11	15.000	7.649	3.928	2.145	1.523
nsort12	179.953	90.191	45.449	23.124	16.022
puzzle	0.114	0.095	0.087	0.097	0.121
puzzle4x4	4.904	2.578	1.448	0.865	0.727
queens13	28.790	18.130	13.117	11.383	12.161
Σ	275.253	144.077	79.949	50.587	44.150

Table A.5: Execution times, in seconds, for YapOr's half splitting implementation with incremental copy.

	Workers				
Programs	**2**	**4**	**8**	**16**	**24**
cubes7	0.113	0.067	0.043	0.039	0.055
ham	0.176	0.100	0.061	0.058	0.079
magic	20.350	10.355	5.229	2.637	1.778
map	11.808	6.698	4.183	3.806	4.619
nsort10	1.357	0.698	0.350	0.190	0.143
nsort11	14.902	7.478	3.727	1.895	1.280
nsort12	179.691	89.598	44.756	22.101	14.912
puzzle	0.086	0.052	0.034	0.031	0.035
puzzle4x4	4.735	2.389	1.222	0.633	0.431
queens13	24.311	12.337	6.272	3.254	2.308
Σ	257.529	129.770	65.879	34.644	25.640

Appendix B

Benchmark Programs

This appendix contains the set of benchmark programs used in Chapter 6.

cubes7

```
benchmark :- cubes7(X).

cubes7(Sol):-
   cubes(7,Qs),
   solve(Qs, [], Sol).

cubes(7,[q(p(5,1),p(0,5),p(3,1)),
 q(p(2,3),p(1,4),p(4,0)),
         q(p(3,6),p(0,0),p(2,4)),
 q(p(6,4),p(6,1),p(0,1)),
 q(p(1,5),p(3,2),p(5,2)),
 q(p(5,0),p(2,3),p(4,5)),
 q(p(4,2),p(2,6),p(0,3))]).

solve([],Rs,Rs).
solve([C|Cs],Ps,Rs):-
   set(C,P),
   check(Ps,P),
   solve(Cs,[P|Ps],Rs).

check([],_).
check([q(A1,B1,C1,D1)|Ps],P):-
   P = q(A2,B2,C2,D2),
   A1 =\= A2, B1 =\= B2, C1 =\= C2, D1 =\= D2,
   check(Ps,P).

set(q(P1,P2,P3),P):- rotate(P1,P2,P).
set(q(P1,P2,P3),P):- rotate(P2,P1,P).
```

```
set(q(P1,P2,P3),P):- rotate(P1,P3,P).
set(q(P1,P2,P3),P):- rotate(P3,P1,P).
set(q(P1,P2,P3),P):- rotate(P2,P3,P).
set(q(P1,P2,P3),P):- rotate(P3,P2,P).

rotate(p(C1,C2),p(C3,C4),q(C1,C2,C3,C4)).
rotate(p(C1,C2),p(C3,C4),q(C1,C2,C4,C3)).
rotate(p(C1,C2),p(C3,C4),q(C2,C1,C3,C4)).
rotate(p(C1,C2),p(C3,C4),q(C2,C1,C4,C3)).
```

ham

```
benchmark :- ham(_).

ham(H):-cycle_ham([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z],H).

cycle_ham([X|Y],[X,T|L]) :-
chain_ham([X|Y],[],[T|L]),
ham_edge(T,X).

chain_ham([X],L,[X|L]).
chain_ham([X|Y],K,L) :-
ham_del(Z,Y,T),
ham_edge(X,Z),
chain_ham([Z|T],[X|K],L).

ham_del(X,[X|Y],Y).
ham_del(X,[U|Y],[U|Z]) :- ham_del(X,Y,Z).

ham_edge(X,Y) :-
ham_connect(X,L),
ham_el(Y,L).

ham_el(X,[X|_]).
ham_el(X,[_|L]) :- ham_el(X,L).

ham_connect(a,[b,n,m]).
ham_connect(b,[c,a,u]).
ham_connect(c,[d,b,o]).
ham_connect(d,[e,c,v]).
ham_connect(e,[f,d,p]).
ham_connect(f,[g,e,w]).
ham_connect(g,[h,f,q]).
ham_connect(h,[i,g,x]).
ham_connect(i,[j,h,r]).
ham_connect(j,[k,i,y]).
ham_connect(k,[l,j,s]).
ham_connect(l,[m,k,z]).
ham_connect(m,[a,l,t]).
ham_connect(n,[o,a,t]).
ham_connect(o,[p,n,c]).
ham_connect(p,[q,o,e]).
ham_connect(q,[r,p,g]).
ham_connect(r,[s,q,i]).
ham_connect(s,[t,r,k]).
```

```
ham_connect(t,[s,m,n]).
ham_connect(u,[v,z,b]).
ham_connect(v,[w,u,d]).
ham_connect(w,[x,v,f]).
ham_connect(x,[y,w,h]).
ham_connect(y,[z,x,j]).
ham_connect(z,[y,l,u]).
```

nsort10

```
benchmark :- nsort(_).

nsort(L) :- go_nsort([10,9,8,7,6,5,4,3,2,1],L).

go_nsort(L1,L2) :-
    nsort_permutation(L1,L2),
    nsort_sorted(L2).

nsort_sorted([X,Y|Z]) :-
    X =< Y,
    nsort_sorted([Y|Z]).
nsort_sorted([_]).

nsort_permutation([],[]).
nsort_permutation(L,[H|T]):-
    nsort_delete(H,L,R),
    nsort_permutation(R,T).

nsort_delete(X,[X|T],T).
nsort_delete(X,[Y|T],[Y|T1]) :- nsort_delete(X,T,T1).
```

puzzle

```
benchmark :- puzzle_solution(_).

puzzle_solution([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S]) :-
  List=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
  puzzle_member(A,List,La), puzzle_member(B,La,Lb),
  C is 38-A-B, puzzle_member(C,Lb,Lc),
  A < C,
  puzzle_member(D,Lc,Ld), H is 38-A-D, puzzle_member(H,Ld,Lh),
  A < H, C < H,
  puzzle_member(E,Lh,Le), puzzle_member(F,Le,Lf), G is 38-D-E-F, puzzle_member(G,Lf,Lg),
  L is 38-C-G, puzzle_member(L,Lg,Ll),
  A < L,
  puzzle_member(I,Ll,Li), M is 38-B-E-I, puzzle_member(M,Li,Lm),
  Q is 38-H-M, puzzle_member(Q,Lm,Lq),
  A < Q,
  puzzle_member(J,Lq,Lj), N is 38-C-F-J-Q, puzzle_member(N,Lj,Ln),
  K is 38-H-I-J-L, puzzle_member(K,Ln,Lk),
  P is 38-B-F-K, puzzle_member(P,Lk,Lp),
  S is 38-L-P, puzzle_member(S,Lp,Ls),
  A < S,
  R is 38-Q-S, puzzle_member(R,Ls,Lr), 38 is D+I+N+R,
```

```
  puzzle_member(O,Lr,_Lo), 38 is M+N+O+P, 38 is A+E+J+O+S, 38 is G+K+O+R.

puzzle_member(X,[X|Y],Y).
puzzle_member(X,[X2|Y],[X2|Y2]) :- X \== X2, puzzle_member(X,Y,Y2).
```

map

```
benchmark :- map(_).

map(M) :-    other_map(M),
             map_colours(C),
             colour_map(M,C).

my_map([country(a,A,[B,C,D,F,G]),
        country(b,B,[A,C,E,G]),
        country(c,C,[A,B,D,E]),
        country(d,D,[A,C,E,F,H]),
        country(e,E,[B,C,D,H,I,J]),
        country(f,F,[A,D,G,H,J]),
        country(g,G,[A,B,F,J]),
        country(h,H,[D,E,F,I,J]),
        country(i,I,[E,H,J]),
        country(j,J,[E,F,G,H,I])]).

other_map([country(pa,PA,[TO,MA,AM,RR,AP]),
           country(am,AM,[AC,RR,RO,MT,RN]),
           country(ac,AC,[AM]),
           country(rn,RO,[AM]),
           country(ro,RR,[PA,AM]),
           country(ap,AP,[PA]),
   country(to,TO,[PA,MA]),
   country(ma,MA,[TO,PA,PI]),
   country(pi,PI,[MA,CE,PE]),
   country(ce,CE,[PI,RN,PB]),
   country(rn,RN,[CE,PB]),
   country(pb,PB,[RN,PE,CE]),
   country(pe,PE,[PB,CE,PI,AL]),
   country(al,AL,[PE,SE]),
   country(se,SE,[BA,AL])]).

colour_map([],_).
colour_map([Country|Map], Colourlst) :-
    colour_country(Country, Colourlst),
    colour_map(Map, Colourlst).

colour_country(country(_,C,AdjacentCs), Colourlst) :-
    map_del(C, Colourlst, CL),
    map_subset(AdjacentCs, CL).

map_subset([],_).
map_subset([C|Cs], Colourlst) :-
    map_del(C, Colourlst, _),
    map_subset(Cs, Colourlst).

map_colours([red, green, blue, white]).
```

```
map_del(X, [X|L], L).
map_del(X, [Y|L1], [Y|L2] ) :-
    map_del(X, L1, L2).
```

magic

```
benchmark :- magic_go7(S).

magic(S) :- magic_go7(S).

magic_go7(S):- magic_problem7(Y), magic_solve(7,Y,S).

magic_problem7([ [2,3,3,1,5,5,5,3,3],
                 [6,3,5,2,2,5,6,6,2],
                 [4,6,1,4,6,6,6,6,5],
                 [4,1,5,4,4,5,2,4,3],
                 [6,3,1,2,3,5,3,4,4],
                 [1,1,1,1,1,2,4,2,2] ]).

magic_movimento(1,0).
magic_movimento(2,1).
magic_movimento(3,2).
magic_movimento(4,3).
magic_movimento(5,4).
magic_movimento(6,5).
magic_movimento(7,6).

magic_solve(_,[ [1,1,1,1,1,1,1,1,1],
                [2,2,2,2,2,2,2,2,2],
                [3,3,3,3,3,3,3,3,3],
                [4,4,4,4,4,4,4,4,4],
                [5,5,5,5,5,5,5,5,5],
                [6,6,6,6,6,6,6,6,6] ],[]).

magic_solve(N,[ [L11,L12,L13,_,_,_,_,_,_],
                [L21,L22,L23,_,_,_,_,_,_],
                [L31,L32,L33,_,_,_,_,_,_],
                [L41,L42,L43,_,_,_,_,_,_],
                [L51,L52,L53,L54,_,L56,L57,L58,L59],
                [_,_,_,_,_,_,_,_,_] ], [dir1|L]):-
                magic_movimento(N,N1),
                magic_solve(N1,[ [L41,L42,L43,_,_,_,_,_,_],
                                 [L11,L12,L13,_,_,_,_,_,_],
                                 [L21,L22,L23,_,_,_,_,_,_],
                                 [L31,L32,L33,_,_,_,_,_,_],
                                 [L53,L56,L59,L52,_,L58,L51,L54,L57],
                                 [_,_,_,_,_,_,_,_,_] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [dir2|L]):-
      magic_movimento(N,N1),
```

```
      magic_solve(N1,[ [L11,L12,L13,L44,L45,L46,L17,L18,L19],
                 [L21,L22,L23,L14,L15,L16,L27,L28,L29],
                 [L31,L32,L33,L24,L25,L26,L37,L38,L39],
                 [L41,L42,L43,L34,L35,L36,L47,L48,L49],
                 [L53,L56,L59,L52,L55,L58,L51,L54,L57],
                 [L61,L62,L63,L64,L65,L66,L67,L68,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [dir3|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L12,L13,L14,L15,L16,L47,L48,L49],
        [L21,L22,L23,L24,L25,L26,L17,L18,L19],
                 [L31,L32,L33,L34,L35,L36,L27,L28,L29],
                 [L41,L42,L43,L44,L45,L46,L37,L38,L39],
                 [L51,L52,L53,L54,L55,L56,L57,L58,L59],
                 [L63,L66,L69,L62,L65,L68,L61,L64,L67] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [lat1|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L67,L12,L13,L64,L15,L16,L61,L18,L19],
                 [L21,L22,L23,L24,L25,L26,L27,L28,L29],
                 [L31,L32,L57,L34,L35,L54,L37,L38,L51],
                 [L43,L46,L49,L42,L45,L48,L41,L44,L47],
                 [L11,L52,L53,L14,L55,L56,L17,L58,L59],
                 [L33,L62,L63,L36,L65,L66,L39,L68,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [lat2|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L68,L13,L14,L65,L16,L17,L62,L19],
                 [L21,L22,L23,L24,L25,L26,L27,L28,L29],
                 [L31,L58,L33,L34,L55,L36,L37,L52,L39],
                 [L41,L42,L43,L44,L45,L46,L47,L48,L49],
                 [L51,L12,L53,L54,L15,L56,L57,L18,L59],
                 [L61,L32,L63,L64,L35,L66,L67,L38,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [lat3|L]):-
      magic_movimento(N,N1),
```

```
      magic_solve(N1,[ [L11,L12,L69,L14,L15,L66,L17,L18,L63],
                  [L27,L24,L21,L28,L25,L22,L29,L26,L23],
                  [L59,L32,L33,L56,L35,L36,L53,L38,L39],
                  [L41,L42,L43,L44,L45,L46,L47,L48,L49],
                  [L51,L52,L13,L54,L55,L16,L57,L58,L19],
                  [L61,L62,L31,L64,L65,L34,L67,L68,L37] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [frn1|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L17,L14,L11,L18,L15,L12,L19,L16,L13],
                  [L57,L22,L23,L58,L25,L26,L59,L28,L29],
                  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
                  [L41,L42,L67,L44,L45,L68,L47,L48,L69],
                  [L51,L52,L53,L54,L55,L56,L49,L46,L43],
                  [L61,L62,L63,L64,L65,L66,L27,L24,L21] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [frn2|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
                  [L21,L54,L23,L24,L55,L26,L27,L56,L29],
                  [L31,L32,L33,L34,L35,L36,L37,L38,L39],
                  [L41,L64,L43,L44,L65,L46,L47,L66,L49],
                  [L51,L52,L53,L48,L45,L42,L57,L58,L59],
                  [L61,L62,L63,L28,L25,L22,L67,L68,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [frn3|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
                  [L21,L22,L51,L24,L25,L52,L27,L28,L53],
                  [L33,L36,L39,L32,L35,L38,L31,L34,L37],
                  [L61,L42,L43,L62,L45,L46,L63,L48,L49],
                  [L47,L44,L41,L54,L55,L56,L57,L58,L59],
                  [L29,L26,L23,L64,L65,L66,L67,L68,L69] ],L).
```

nsort11

```
benchmark :- nsort(_).

nsort(L) :- go_nsort([11,10,9,8,7,6,5,4,3,2,1],L).
```

```
go_nsort(L1,L2) :-
    nsort_permutation(L1,L2),
    nsort_sorted(L2).

nsort_sorted([X,Y|Z]) :-
    X =< Y,
    nsort_sorted([Y|Z]).
nsort_sorted([_]).

nsort_permutation([],[]).
nsort_permutation(L,[H|T]):-
    nsort_delete(H,L,R),
    nsort_permutation(R,T).

nsort_delete(X,[X|T],T).
nsort_delete(X,[Y|T],[Y|T1]) :- nsort_delete(X,T,T1).
```

nsort12

```
benchmark :- nsort(_).

nsort(L) :- go_nsort([12,11,10,9,8,7,6,5,4,3,2,1],L).

go_nsort(L1,L2) :-
    nsort_permutation(L1,L2),
    nsort_sorted(L2).

nsort_sorted([X,Y|Z]) :-
    X =< Y,
    nsort_sorted([Y|Z]).
nsort_sorted([_]).

nsort_permutation([],[]).
nsort_permutation(L,[H|T]):-
    nsort_delete(H,L,R),
    nsort_permutation(R,T).

nsort_delete(X,[X|T],T).
nsort_delete(X,[Y|T],[Y|T1]) :- nsort_delete(X,T,T1).
```

puzzle4x4

```
benchmark :- puzzle4x4(X).

puzzle4x4(X) :- pz4x4_go12(X).

pz4x4_go12(S):- pz4x4_problem12(Y), pz4x4_solve(12,Y,S).

pz4x4_problem12([ 2, o, 3, 8,
            1, 6,11, 7,
            5, 9,10,16,
           13,14,12,15]).

pz4x4_movimento(1,0).
```

```
pz4x4_movimento(2,1).
pz4x4_movimento(3,2).
pz4x4_movimento(4,3).
pz4x4_movimento(5,4).
pz4x4_movimento(6,5).
pz4x4_movimento(7,6).
pz4x4_movimento(8,7).
pz4x4_movimento(9,8).
pz4x4_movimento(10,9).
pz4x4_movimento(11,10).
pz4x4_movimento(12,11).

pz4x4_solve(_,[ 1, 2, 3, o,
          5, 6, 7, 8,
          9,10,11,12,
         13,14,15,16],[]).

pz4x4_solve(N,       [P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m11|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[  o,P11,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m11|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[  o,P12,P13,P14,
                P11,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [  o,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P12,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,  o,P12,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
```

```
pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,  o,P13,P14,
                P21,P12,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P13,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,  o,P13,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,  o,P14,
                P21,P22,P13,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m14|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P14,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m14|L]):-
      pz4x4_movimento(N,N1),
```

```
      pz4x4_solve(N1,[P11,P12,P13,  o,
                P21,P22,P23,P14,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [  o,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P21,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  o,P21,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  o,P22,P23,P24,
                P21,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P22,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P22,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
```

```
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P22,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,  o,P22,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P23,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P23,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P23,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,  o,P23,
```

```
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m24|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P24,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m24|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P24,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P44],[m24|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P24,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m31|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P31,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m31|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P31,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
```

```
                o,P42,P43,P44],[m31|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P31,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m32|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P32,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m32|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P32,  o,P33,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m32|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P32,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P43,P44],[m32|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P32,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P33,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],L).
```

```
pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P33,  o,P34,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P33,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P44],[m33|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32, o,P34,
                P41,P42,P33,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m34|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P34,
                P31,P32,P33,  o,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m34|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P34,  o,
                P41,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,  o],[m34|L]):-
      pz4x4_movimento(N,N1),
```

```
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P34],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m41|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P41,P32,P33,P34,
                  o,P42,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P43,P44],[m41|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                  o,P41,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m42|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P42,P33,P34,
                P41,  o,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                  o,P42,P43,P44],[m42|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P42,  o,P43,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P44],[m42|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P42,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
```

```
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m43|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P43,P34,
                P41,P42,  o,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P43,P44],[m43|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P43,  o,P44],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,  o],[m43|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P43],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P44],[m44|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P44,
                P41,P42,P43,  o],L).

pz4x4_solve(N,       [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P44],[m44|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P44,  o],L).
```

queens13

```
benchmark :- get_solutions(13,S).

get_solutions(Board_size, Soln) :- solve(Board_size, [], Soln).
```

```
newsquare([square(I,J)|Rest],square(X,Y)) :-
       X is I+1,
       snint(Y),
       not_threatened(I,J,X,Y),
       safe(X,Y,Rest).
newsquare([],square(1,X)) :- snint(X).

a(_).
a.

safe(X,Y,[square(I,J)|L]) :-
       not_threatened(I,J,X,Y),
       safe(X,Y,L).
safe(X,Y,[]).

not_threatened(I,J,X,Y) :-
       I =\= X,
       J =\= Y,
       I-J =\= X-Y,
       I+J =\= X+Y.

solve(Board_size,Initial,Final) :-
       newsquare(Initial,Next),
       solve(Board_size,[Next|Initial],Final).
solve(Bs,[square(Bs,Y)|L],[square(Bs,Y)|L]) :- size(Bs).

size(13).
snint(1).
snint(2).
snint(3).
snint(4).
snint(5).
snint(6).
snint(7).
snint(8).
snint(9).
snint(10).
snint(11).
snint(12).
snint(13).
```

References

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

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

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

[4] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology, 1990.

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

[6] 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.

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

[8] 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.

[9] R. Karlsson. *A High Performance OR-parallel Prolog System*. PhD thesis, The Royal Institute of Technology, 1992.

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

[11] R. Kowalski. The Early Years Of Logic Programming. *Communications of the ACM*, 31:38–43, 1988.

[12] 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.

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

[14] R. Rocha. Um Sistema Baseado na Cópia de Ambientes para a Execução de Prolog em Paralelo. Master's thesis, University of Minho, 1996.

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

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

[17] Nilsson Ulf and Jan M. *Logic, Programming and Prolog.* John Wiley and Sons, 1995.

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

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