Universidade de Évora

Constraint Programming on Hierarchical Multiprocessor Systems

Vasco Fernando de Figueiredo Tavares Pedro

Dissertação submetida à Universidade de Évora para obtenção do grau de Doutor em Informática.

Orientador: *Salvador Pinto de Abreu*

Évora, Maio de 2012

À Margarida

Acknowledgements

Em primeiro lugar, tenho de agradecer ao Salvador, o meu orientador, pela oportunidade que me deu, pelos empurrões pelo caminho, e por nunca se ter arrependido. Obrigado, meu.

Margarida, obrigado por me acompanhares.

O meu reconhecimento vai também para o Miguel Avillez, por ter oferecido o uso do seu *cluster*, o que me permitiu realizar grande parte da experimentação necessária para este trabalho.

Agradeço também ao João Lourenço e ao Paulo Lopes o terem-me dado acesso às suas máquinas, para mais experiências, e ao Rui Machado por fazer as minhas experiências lá no Frauhoffer ITWM.

Je remercie Olivier Aumage, Jérôme Clet-Ortega, et Cédric Augonnet pour leur suggestions et leur bon accueil à Bordeaux. Je remercie aussi Daniel Diaz, à Paris, d'avoir amené à ma connaissance les Costas Arrays et le QAP, ainsi que pour l'usage de sa machine.

À minha família e aos meus colegas do DI, agradeço o serem, em geral, boas pessoas. Uma palavra de agradecimento é devida, também, às pessoas que têm passado pela secretaria do DI, pela sua disponibilidade.

O trabalho que deu origem a esta tese foi parcialmente financiado através do projecto PHC-PESSOA CONTEMP — "CONTraintes Exécutées en MultiProcesseurs".

Abstract

The work reported in this thesis is about constraint processing in the context of hierarchical multiprocessor systems, including distributed systems. More specifically, it develops techniques and a system to help bringing the power available in today's multiprocessing networked systems into the constraint processing field.

Solving constraint specified problems is a process which lends itself naturally to parallelisation, as it usually implies going through very large search spaces, looking for a solution. Parallel constraint solving draws on the idea of dividing the search space among several workers, so the search may proceed faster, and thanks to the declarative nature of constraint programming, the parallelisation happens transparently as far as the user is concerned. However, to fully take advantage of the parallel computing power available, techniques must be developed to help ensure that the workers executing the search are kept busy at all times, which is an issue tackled by this work.

Programação por Restrições em Sistemas Multiprocessador Hierárquicos

Sumário

Esta tese debruça-se sobre a programação por restrições no contexto dos sistemas multiprocessador hierárquicos, incluindo os sistemas distribuídos. Mais especificamente, o trabalho elaborado desenvolve as técnicas de resolução de problemas de satisfação de restrições recorrendo ao paralelismo.

A actualidade do tema prende-se com a cada vez maior divulgação de que são objecto os sistemas multiprocessador que, juntamente com a omnipresença das redes de computadores, põe à nossa disposição uma capacidade de cálculo que necessita de ser posta a uso, o que tarda em acontecer. Nesta tese desenvolve-se um sistema que permite tirar partido desses recursos através do processamento de restrições.

A programação por restrições é um paradigma declarativo, em que o utilizador não tem de se preocupar com o controlo da computação, e a introdução de paralelismo nesta área pode realizar-se transparentemente. Por outro lado, o processo de pesquisa de soluções para problemas especificados por restrições adapta-se particularmente bem a ser paralelizado.

Este tese apresenta uma abordagem à resolução paralela de restrições, que junta paralelismo local, sob a forma de *trabalhadores*, com paralelismo distribuído, em que os actores são as *equipas*. O sistema construído, destinado a sistemas distribuídos de larga escala, que é descrito e os seus resultados apresentados, inclui distribuição de trabalho, através de *roubo de trabalho*. Este funciona, localmente, sem a colaboração do roubado e, remotamente, com colaboração, num ambiente em que todas as equipas cooperam na procura da solução.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraints have long been used to help solve hard problems. The process usually involves sifting through huge search spaces until a solution is found. As this may be computationally very expensive, the more computational power committed to the task, the better the chances are of coming to a conclusion in a reasonable amount of time.

In the last few years, as the quest for pure speed appeared to meet with some roadblocks [57], we have witnessed multiprocessors getting commonplace, having showed up in laptops, tablet computers, and mobile phones. At the same time, computer networks became ubiquitous, from local area networks in the workplace to wireless access points in many homes, all interconnected. Computational power is now readily available from multiple sources simultaneously, be they local or remote. So, the time seems right to bring together the needs arising from solving constraint specified problems and the power of multiprocessing systems, and this is where the focus of the work described in this text lies.

This first chapter sets the ground for the issues addressed, introducing constraints and constraint solving techniques, and their articulation with parallel and distributed processing. The aims and accomplishments of the work will also be reviewed.

1.1 Constraint Satisfaction

Many practical problems, such as planning, scheduling, and network monitoring, may be described through a set of *variables*, the *values* they may take, and the *relations* those values must obey. This, in a nutshell, is what constitutes a Constraint Satisfaction Problem (CSP), with the relations being the *constraints* on the variables values. Solving the original problem then reduces to finding a value for every variable, such that all the relations are satisfied and this is called *constraint satisfaction* or *constraint solving.*

A tuple consisting of the values assigned to the variables is a *global assignment*, or *assignment* for short, and a *solution* for the problem is an assignment that satisfies all the constraints. The set of all possible assignments is the *search space* of the problem.

One of the most widespread techniques used for constraint solving is *backtracking search* [60], where variables are assigned values in turn. As each variable is *instantiated* with an element from its *domain* — the set of all values it may possibly take —, its value is checked against those of already assigned variables, to verify that it satisfies the constraints they share. If it does not, another value is tried. If no value from the domain of a variable is compatible with the values assigned to the other variables, the system must backtrack and try to reassign the last assigned variable. On failure to find a value for a variable, *backjumping* [13] improves the efficiency of the search by identifying the earliest assigned variable whose value conflicts with the possible values for the variable being assigned, and undoing all variable assignments performed after that variable's.

With *forward checking* [1], every time a variable is assigned, which corresponds to its domain becoming a singleton, that change is *propagated*, causing the deletion of the elements in the other variables domains that are incompatible with the value just assigned. When using *(full) look-ahead* [1], these deletions will also, in turn, be propagated until they provoke no further changes. If the domain of some variable becomes empty, all changes must be undone and a new value for the variable will be tried.

The above techniques are all *complete*. If a solution for the problem exists, they are guaranteed to find it. *Local search* [22], on the other hand, is an *incomplete* search procedure. It starts by assigning each variable with a randomly chosen value. If the assignment obtained is not a solution for the problem, one or more variables are reassigned in such a way that the new assignment is closer to a solution, according to some metric. These steps are repeated until either a solution is found or a predetermined number of steps has been completed.

1.2 Parallelism and Constraint Satisfaction

Solving a CSP involves making choices and, once an alternative has been completely explored without leading to the desired outcome, a choice must be remade. Usually, this requires keeping a record of the untried alternatives at each choice point, one of which will be picked the next time it is reached. However, it is possible to imagine that instead of pursuing only one of the alternatives, at some point several could be chosen and followed in parallel by a number of *workers*.

During backtracking search, choices include selecting a variable to branch on, determining the branches to create and which to take next. Variable selection heuristics control how a variable is selected and play a major role in shaping the search strategy and its effectiveness. Having workers with distinct heuristics would mean that different approaches were being used simultaneously to solve the same problem. On the other hand, once a variable has been selected and the branches created, several of these could be taken by different workers, which would be tantamount to splitting the problem into *subproblems*, whose solutions are also solutions to the original problem, and solving the subproblems in parallel.

Another possible approach is to divide the responsibility of assigning the variables among a number of parties, here called *agents*, each of which with only partial knowledge of the problem [66, 14]. *Local* constraints affect variables from only one agent and are known by that agent, and *non-local* constraints are shared by variables belonging to different agents and are known by at least one of them. Each agent instantiates its variables according to its local constraints. As for the non-local constraints, agents must negotiate values for the foreign variables that are compatible with some assignment of the local variables. This setting is appropriate when there are privacy concerns regarding some part of the problem.

There have also been attempts to parallelise the algorithms of a single search process, such as the propagation step, but it is not possible to escape its inherent sequentiality [4, 11, 27, 61].

1.3 A Parallel Constraint Solver

As seen above, parallelism may enter the constraint satisfaction process in different ways, most of which feature a number of workers simultaneously involved in solving the problem. The work developed here looks into the concurrent exploration of the problem search space by several workers, which may reside on the same machine, be distributed across machines, or both. This approach matches the hardware for massively parallel computation currently most widespread, which are interconnected multiprocessing computers, such as the portuguese Milipeia [35], the french Grid'5000 [19], and the japanese HA8000 [58] systems. Co-located workers are able to benefit from faster communications and may operate in a tightly coupled way. Communication over the network being slower, workers on different machines communicate as little as possible, exhibiting a looser coupling.

The initial goals included developing an easily portable constraint solving system, or *constraint solver*, able to take advantage of such systems. The system built and described here has been designed from scratch to exploit these architectural features, instead of the more common approach of fitting an extant constraint solver with parallelising capabilities. Load balancing between workers, also a concern in parallel and distributed frameworks, is achieved through both local and remote *work stealing* [44, 45]. The solver displays good speedups and scalability characteristics.

Distinguishing features of the solver include the combination of multiprocessing (*e.g.* multithreading) and distributed parallelism in constraint solving, along with local and global dynamic load balancing, and its aptitude to function as a constraint solving service, receiving problems and delivering solutions over the network.

1.4 Thesis Plan

The thesis is organised as follows. In the next chapter, constraint solving and parallel constraint solving are treated in more depth, including a survey of the state of the art. Chapter 3 presents the constraint solver in detail, and Chapter 4 analyses the experimental results obtained with it and compares them to those of other systems. Finally, the outcomes of this work are discussed in Chapter 5.

Appendix A presents the Application Programming Interface of the solver.

Chapter 2

Parallel Constraint Solving

The fundamental notions and ideas from the constraint satisfaction field underlying this work are presented in the first two sections of this chapter, which also aims at fixing some terminology, as some terms have varying meanings in the literature.

Section 2.3 introduces the issues concerning parallelism in constraint processing, Section 2.4 concentrates on parallel search for constraint solving, including a review of the state of the art, and Section 2.5 situates the work done within the field.

Bibliographic Note

The following two sections are mainly based on Rina Dechter's *Constraint Processing* [13], on Krzysztof Apt's *Principles of Constraint Programming* [1], and on Chapters *Constraint Propagation*, by Christian Bessière [2], and *Backtracking Search Algorithms*, by Peter van Beek [60], from the *Handbook of Constraint Programming* [50], and further references to these works are omitted therein.

2.1 Basics of Constraint Satisfaction

A constraint satisfaction problem can be briefly defined as a set of variables whose values, to be drawn from their domains, must satisfy a set of relations. Furthermore, the domains considered here are restricted to being finite sets of values.

Definition 2.1 (CSP) *A* Constraint Satisfaction Problem (CSP) *over finite domains is a triple $P = (X, D, C)$, where*

- $X = \{x_1, x_2, \ldots, x_n\}$ *is an indexed set of* variables*;*
- $D = \{D_1, D_2, \ldots, D_n\}$ *is an indexed set of finite sets of values, with D_i being the* domain *of variable x_i, for every $i = 1, 2, \ldots, n$; and*
- $C = \{c_1, c_2, \ldots, c_m\}$ *is a set of relations between variables, called the* constraints*.*

The scope *of a constraint c_i is the non-empty set of variables $scope(c_i) = S_{c_i} \subseteq X$ over which it is defined. The* arity *of a constraint is the number of variables in its scope. Constraints with arity one and two are called, respectively,* unary *and* binary *constraints.*

Constraints are commonly stated intensionally or symbolically, *e.g.* $x_1 \leq x_2$ and *all-different*$(\{x_1, x_3, x_5\})$.

A tuple of values over $S \supseteq S_{c_i}$ *satisfies* constraint c_i when its projection onto S_{c_i} belongs to c_i. (The *projection* of a tuple t over $S \subseteq X$ *onto* S' is the tuple formed by the elements of t corresponding to the variables common to S and S'.) On the other hand, a tuple over $S \subseteq X$ *violates* c_i when no tuple in c_i matches its projection onto $S \cap S_{c_i}$.

The *search space* of a CSP consists of all the tuples from the cross product of the domains, where each variable is *assigned* a value from its domain, or *instantiated.* Solving a CSP amounts to finding some or all of those tuples that satisfy all constraints of the problem.

Definition 2.2 (Assignment, Solution, Inconsistency) *Let $P = (X, D, C)$ be a CSP. A* (global) assignment *is an n-tuple*

$$(v_1, v_2, \ldots, v_n) \in D_1 \times D_2 \times \ldots \times D_n.$$

A solution *to P is an assignment such that all constraints are satisfied, and $Sol(P)$ denotes the set of all solutions to P. When $Sol(P)$ is the empty set, the problem is said to be* inconsistent*. In a* partial assignment*, only a subset of the variables is assigned a value.*

In a *constraint optimisation problem*, the goal is not only to find a solution, but to find a *best* one, according to some *criterion.* This criterion is embodied in an *objective function*, which maps solutions to values, and whose value is to be minimised or maximised.

Definition 2.3 (COP) *A* Constraint Optimisation Problem (COP) *is a quadruple $P = (X, D, C, f)$, where*

- *(X, D, C) is a CSP; and*
- *$f : Sol((X, D, C)) \to \mathbb{N}$ is the* objective function*.*

Often, a variable constrained to have the value of the objective function is assumed to exist, and the goal becomes minimising or maximising its value [30]. In this case, which corresponds to the view taken here, an optimisation problem may be handled resorting mainly to CSP solving techniques.

As in a typical divide-and-conquer approach, a problem may be divided into subproblems, whose solutions are also solutions to the original problem.[1]

Definition 2.4 (Subproblem) *Let $P = (X, D, C)$ be a CSP. A CSP $P' = (X, D', C)$ such that $D' = \{D'_1, D'_2, \ldots, D'_n\}$ and $D'_i \subseteq D_i$, for every $i = 1, 2, \ldots, n$, is a* subproblem *of P.*[2]

To guarantee completeness of the search when working with subproblems, their search spaces must cover the search space of the original problem. In order to avoid redundant work, they must also be pairwise disjoint.

Definition 2.5 (Partition) *A set $\{P'_1, P'_2, \ldots, P'_k\}$ of subproblems of a CSP P, with $P'_i = (X, \{D'_{i1}, D'_{i2}, \ldots, D'_{in}\}, C)$, is a* partition *of P if*

$$\bigcup_{1 \le i \le k} D'_{i1} \times D'_{i2} \times \cdots \times D'_{in} = D_1 \times D_2 \times \cdots \times D_n$$

and $(\forall\, i \neq j)\, (D'_{i1} \times D'_{i2} \times \cdots \times D'_{in}) \cap (D'_{j1} \times D'_{j2} \times \cdots \times D'_{jn}) = \emptyset$.

A partition of a CSP may be dually regarded as a partition of its search space, the search spaces of the subproblems being *sub-search spaces* of the original problem. In this work we will only deal with search space partitions that correspond to some partition of a problem.

2.2 Basics of Constraint Solving

Methods for constraint solving, *i.e.* for finding the solutions of a CSP, may be *complete* or *incomplete.*

Incomplete methods, such as *(stochastic) local search* [22], try to find a solution using limited resources, and are neither guaranteed to find one nor able to detect inconsistency. They are nevertheless useful as in some cases they are able to find a solution much faster than the known complete methods. They are also appropriate when time constraints prevent performing a complete search and an approximate solution, or a good enough one in case of an optimisation problem, suffices.

Complete methods, which always find a (best) solution if one exists, make use of two main techniques: *search* and *inference.*

Search-based methods build a solution incrementally, by trying at each step to extend an initially empty partial assignment to another variable, in such a way that no constraint

[1] When dealing with an optimisation problem, the best solution to a subproblem may not be a best solution to the full problem.

[2] To be completely correct, tuples containing values no longer belonging to the domain of a variable should be removed from the constraints.

is violated. If for some variable no value from its domain allows extending the current partial assignment, *backtracking* takes place and a previous assignment is redone. This is the essence of *depth-first backtracking search.*

Extending a partial assignment requires selecting the next variable to instantiate and a value from its domain to instantiate it with, and the performance of the search depends greatly on the criteria applied in these selections. *Selection heuristics* guide the two steps and while *variable selection heuristics* try to maximise the pruning of the search space, *value selection heuristics* are geared towards choosing the value with the greatest probability of belonging in a solution.

Search traverses the (virtual) *search tree*, whose nodes correspond to partial assignments. The tree root is the empty partial assignment and each level below corresponds to one more variable being instantiated. Global assignments, the elements of the search space, are located at the leaves. Every node of the tree is the root of a subtree and can be equated with the sub-search space consisting of the leaves of that subtree.

Figure 2.1 depicts a search tree for a problem with variables x_1 and x_2, both with domain $\{a, b, c\}$. Nodes with depth 1 have only x_1 instantiated, and nodes at depth 2 have both variables instantiated. The second leaf from the left corresponds to assignment (a, b).

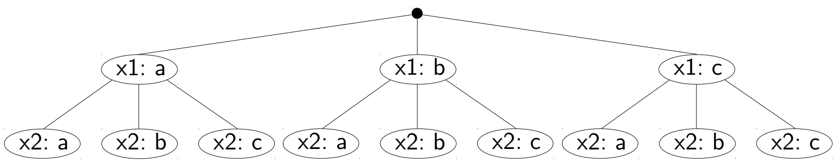

Figure 2.1: Search tree example

Inference- or *propagation-based* methods, on the other hand, work by inferring new constraints and/or *filtering* values out from, or *tightening*, the variables domains until either inconsistency is detected or it is possible to build solutions by instantiating the variables, like above, but without there being a need to backtrack. When this happens, the problem is said to be *globally consistent.*

Both techniques are in general NP-complete and hybrid approaches, combining both search and propagation, have proved effective. Either a partial assignment is built and propagation is performed on the remaining non instantiated variables, or there may be an initial limited inference phase, achieving a weaker level of consistency, followed by search on the reduced domains obtained, or search and inference are interleaved throughout the solving process.

The weaker forms of consistency comprise the *local consistencies*, so called because they apply locally to parts of the problem. If considering only the variables, a CSP is *node consistent* if every value in the domain of every variable satisfies the unary constraints over that variable. When looking at each constraint individually, *(generalised) arc-consistency*[3] comes into play, and a CSP being arc consistent means that the projections of every constraint onto each of its variables coincide with their domains. Arc consistency is the strongest level of consistency that can be obtained when propagation is performed based on single constraints.

Domain tightening does not have to be carried out on the value level. To obtain *bounds consistency* only the least and greatest values of the domains are taken into account and domains are shrunk by, respectively, raising and lowering them.

Stronger levels of local consistency are harder to enforce than weaker levels. Constraints implemented through *propagators* [53] allow weighing the degree of local consistency obtained against the effort required to obtain it for each type of constraint. A constraint propagator reflects the changes to the domain of one of the constraint variables on the domains of the other variables in the constraint scope, removing values which can no longer appear in an assignment satisfying the constraint, according to the level of consistency it enforces. Propagators thus provide a flexible framework for introducing propagation in the solving process.

Bringing all together, Figure 2.2 depicts the kernel of a backtracking constraint solver with propagation, which iterates over the possible branches from a variable (line 10), propagates the modifications effected on its domain to the other variables, and calls itself recursively to try to solve problem P.

The revise function, in Figure 2.3, calls the constraints propagators $\mathsf{propagator}_c$ to revise the domains of the variables against the one that changed, until it stabilises with no more changes taking place. When all the variables become instantiated, the problem has been solved and the solver returns the corresponding assignment, which constitutes a solution.

The branching shown in Figure 2.4 assigns a variable a value from its domain. Usually, the same variable is selected repeatedly, trying a new value every time there is a failure and backtracking occurs, until the variable domain is exhausted. This is called *variable labelling*. The functions select-variable and select-value (lines 3 and 5), not detailed, will each implement one of the variable and value selection heuristics. (The expression $\mathsf{domain}_P(\mathsf{variable})$ appearing in the pseudo-code represents the domain of variable in problem P.)

2.2.1 Solving Optimisation Problems

Staying within the realm of constraint satisfaction, a constraint optimisation problem could be solved by finding all solutions to the corresponding constraint satisfaction problem and selecting one for which the objective function gives the best value.

[3]These names arise from regarding a CSP as a *constraint network*, where nodes correspond to variables and (hyper-)arcs connect the variables according to the constraints they share.

```
SOLVE($P$)
  if revise($P$, $X$) = FAIL then
    return FAIL
  else
    return solve($P$)

solve($P$)
  if solved($P$) then
    return SOLUTION($P$)
  $P' \leftarrow P$
  while (($P''$, variable) $\leftarrow$ select-branch($P'$)) $\neq$ FAIL do
    if revise($P''$, {variable}) $\neq$ FAIL then
      solution $\leftarrow$ solve($P''$)
      if solution $\neq$ FAIL then
        return solution
  return FAIL
```

Figure 2.2: Generic constraint solver

```
revise($P$, variables)
  $Q \leftarrow$ variables
  while $Q \neq \emptyset$ do
    remove some $v$ from $Q$
    foreach $c \in C$ such that $v \in scope(c)$ do
      if propagate$_c$($P$, $v$) = FAIL then
        return FAIL
      add variables whose domains were modified to $Q$
  return SUCCESS
```

Figure 2.3: Full look-ahead propagation

```
select-branch($P$)
  $P' \leftarrow P$
  if (variable $\leftarrow$ select-variable($P'$)) = FAIL then
    return FAIL
  if (value $\leftarrow$ select-value($P'$, variable)) = FAIL then
    return FAIL
  domain$_{P'}$(variable) $\leftarrow$ {value}
  remove value from domain$_P$(variable)
  return ($P'$, variable)
```

Figure 2.4: Variable branching

Another method which also only requires constraint processing techniques is to solve a constraint optimisation problem as a series of constraint satisfaction problems. If the aim is to find a minimum valued solution, the domain of a variable constrained to have the objective function value is first set to the least possible value, and then increased gradually, each time trying to solve the resulting CSP. When a solution is found, it will correspond to a best solution to the optimisation problem. The same strategy may be employed to solve a maximisation problem, starting from the greatest possible value for the variable and decreasing it instead of increasing it. A binary search strategy may be used, as well.

The *(depth-first) branch and bound* algorithm provides a better suited way to deal with the issue, avoiding much of the redundant work incurred by the above strategies. During processing, it keeps a record of the best solution found so far and restricts further search to better solutions, using a *bounding evaluation function* to prune the search space of the problem. This function estimates the value of the objective function of a solution extending a given partial assignment in such a way that when this estimate is worse than the current optimum, the corresponding sub-search space may be safely left unexplored. In case there is a variable constrained to have the value of the objective function, the estimate is obtained by propagation.

Such is the case of the algorithm presented in Figure 2.5, shown in its minimising only version for ease of presentation. A straightforward adaptation of the former generic solver, the main difference is that, instead of stopping when a solution is found, the algorithm now keeps going until no better solution can possibly be obtained. Solutions improve gradually in the sense that each solution found is better than the previous one, each leading to further restricting the domain of the `bound` variable (the one constrained to hold the value of the objective function) to those values which correspond to better solutions (line 16). (The objective function f is only included for the sake of clarity, as its value for the solution just found is the value of the `bound` variable.)

Propagation, on lines 12 and 17, plays the part of the bounding evaluation function for the ‘bound’ phase of the branch and bound algorithm. When the domains of the variables have been sufficiently tightened and lead to a lower bound of the objective function for any solution extending the current partial assignment which is worse than the best value found so far, the domain of `bound` will become empty, propagation will fail, and the search space will be pruned.

2.3 Parallelism in Constraint Solving

The main issue when introducing parallelism into constraint satisfaction is deciding what to parallelise, where will it be most useful? Since parallelisation aims at increased performance, it is important to study the available choices from that point of view.

Looking at the sequential solvers of the previous section, several places where paral-

```
SOLVE-MIN($P$, $f$)
  if revise($P$, $X$) = FAIL then
    return FAIL
  else
    return solve-min($P$, $f$)

solve-min($P$, $f$)
  if solved($P$) then
    return SOLUTION($P$)
  $P' \leftarrow P$
  best-solution $\leftarrow$ FAIL
  while (($P''$, variable) $\leftarrow$ select-branch($P'$)) $\neq$ FAIL do
    if revise($P''$, { variable }) $\neq$ FAIL then
      solution $\leftarrow$ solve-min($P''$, $f$)
      if solution $\neq$ FAIL then
        best-solution $\leftarrow$ solution
        remove values $\geq f$(best-solution) from domain$_{P'}$(bound)
        if domain$_{P'}$(bound) $= \emptyset \vee$ revise($P'$, { bound }) = FAIL then
          return best-solution
  return best-solution
```

Figure 2.5: Minimising branch and bound constraint solver

lelism could be introduced can be identified. One is the branch selection (in Figure 2.4) where, once a variable has been chosen, instead of going over the branches one by one, several could be selected and taken in parallel, resulting in more than one search process being active simultaneously. This constitutes the most natural and the most common approach found in the literature, *e.g.* [61, 52, 15, 69, 33, 8], as it usually involves deploying essentially independent activities, possibly just requiring coordination to handle terminating the search. A more in-depth analysis of this approach is deferred to Section 2.4.

Variables could also be assigned in parallel. In this case, however, the work being done would be complementary, as all variables must eventually be instantiated. To avoid redundant and conflicting efforts, the search processes must reconcile the values of the variables among them, obtaining the values for the variables they are not responsible for instantiating from the processes who are. This constitutes the framework underlying *distributed constraint solving* [65, 66] which handles Distributed Constraint Satisfaction Problems (DisCSPs) through search processes called *agents*.

A DisCSP is a CSP whose variables and constraints are distributed among agents.

Each agent bears the responsibility of instantiating its own variables, in such a way that the *local* constraints, which involve only variables local to the agent, are satisfied, and the *non-local* constraints it knows are not violated. Non-local constraints are those whose scopes contain variables belonging to different agents, and are known by at least one of them. Upon having built a solution to its part of the problem, the agent sends a partial assignment to those agents who need it to check their non-local constraints. An agent that receives a conflicting assignment tries to reassign its variables with compatible values. Failing that, it signals failure to the originating agent, initiating distributed backtracking. This is the core of the *Asynchronous Backtracking (ABT) algorithm*, the main algorithm for solving DisCSPs, which first appeared in [65] and of which many variations exist. This algorithm relies heavily on message passing; however, the main concern behind the distribution of the solving process is not performance but the ability to handle incomplete information and to tackle privacy concerns [14, 4].

A different route is taken in the *portfolio approach* [4], in the context of the Boolean satisfiability problem (SAT) solving. Drawing on the unpredictability of the behaviour of backtracking search [29, 4], multiple strategies are employed in parallel on the *same* search space, trusting that one will outperform the others. As the number of parallel searches grows, so grows the probability of a better strategy being put in action. Different strategies may mean strategies differing in the variable and value selection heuristics or in the tuning of some parameter.

Another candidate for parallelisation is the propagation phase (Figure 2.3), where we can envision the propagators of the constraints affecting a variable being applied in parallel. The fine grained parallelism involved, however, entails such heavy synchronisation, since more than one propagator may try to update the domain of the same variable, that the potential associated benefits have, so far, been outweighed, as reflected by [49]. Moreover, if only part of the computation is parallelised, the speedups will be limited by the part which is not, and establishing arc consistency has been shown to be inherently sequential [61, 4, 27, 11]. Nonetheless, parallel algorithms for establishing (binary) arc-consistency have been developed, *e.g.* [67, 68].

Regarding branch and bound search, yet another possibility arises. Parallel searches could be run with different bounds, in the hope that the pruning thus obtained might lead to the faster discovery of an optimal solution. And even if some of the used bounds are too tight, the effort spent with them might not be totally wasted seeing that it must be proved that they do not correspond to any solution of the problem.

2.4 Parallel Search

Parallel search, or *search space splitting* search, is the most popular approach in the parallelisation of constraint solving. The following are some reasons for its popularity.

- *It is conceptually clear.*

  Parallel search for CSPs consists in having more than one search process active simultaneously, performing search on disjoint parts of the problem search space. The approach is clearly sound as, in the context of constraint satisfaction, it is not important what solution is found, and any solution coming from anywhere in the problem search space is equally valid. This is true whether looking for one solution, all solutions, or a best solution.

  Note that the parallel searches may conceivably be carried out within a single sequential process [55, 20, 69], by multitasking between different search spaces. However, this work is concerned with real parallelism and each search corresponds to an independent flow of execution, which will be called a *worker*.

- *It fits the task.*

  Real-life problems often have huge search spaces, matched by the amount of work involved in solving them. Furthermore, the difficulty in identifying the regions of the search space where the search is most likely to be successful and where efforts should be concentrated, together with clear-cut boundaries for dividing the search space, makes them choice candidates to being split and attacked by many concerted parallel participants.

- *It promises scalable performance gains.*

  First, the whole search is parallelised, not just a part or parts of it. Secondly, the work done on one sub-search space is independent from that done on the others. So, if we are able to achieve a well balanced division of work between the participants, there is reason to expect linear or near linear performance improvement.

  Achieving a balanced work distribution, however, is not easy, due to it being very hard to assess the amount of work associated with any search space, hence load balancing techniques must be employed.

- *It is easy to implement.*

  When a CSP search space is split, the result may be regarded as a series of new CSPs, sharing the variables and the constraints. In a seemingly simplistic approach, implementing parallel search could be as easy as running in parallel as many instances of a solver as the problems thus created, and collecting the answers from each one.

  In practice, this is what is done, with a solver executing within each one of the workers and their coordination being the job of another flow of execution.

Analysing the potential performance further, we see that the speedups which can be expected depend on the task at hand. When a problem is inconsistent or we wish to compute all solutions, the search space must be fully explored and performance gains can be linear on the number of processors employed, provided each process is handed the same amount of work. If any solution of a non inconsistent problem is what is sought, speedups can vary from no speedup, or even a slowdown, to superlinear speedups. These speedup anomalies [46] derive from how the distribution of solutions across the search space affects parallel search.

Consider a problem with only one solution, which is found by sequential search only after exploring more than half of the search space. With two processes performing the search, each having half of the search space to go through, the second process may just take the time the single process needed after finishing exploring the first half of the search space, thus displaying a speedup which can be much greater than 2. On the other hand, if the single solution lies in the first half of the search space, no speedup will be observed. In general, if there are many, evenly distributed solutions, linear speedups may be expected [46].

In what respects optimisation problems, the branch and bound algorithm can be regarded as having two phases: the first, until a best solution is found, is akin to the search for one solution and may exhibit the same speedup anomalies; after that, the unexplored part of the search space must be fully covered, to prove there is no better solution.

The amount of work each worker has to manage depends first on the initial search space partitioning. Unless a perfectly even work distribution is accomplished, one of the workers will eventually finish his while the others are still actively searching. Since static perfectly even work distribution is not possible in general, as good as the initial division of work may be, some way of dynamically sharing the load between the workers must be incorporated into the solver, in order to avoid having idle workers and to achieve higher performance gains.

When load balancing is attempted, the most frequent situation is for a worker to start looking for work as soon as it becomes idle, trying to find another worker able to share some of its work. This technique is what Wilkinson and Allen call receiver-initiated decentralised dynamic load balancing [63], in the context of distributed systems, and it is also known as *work stealing*, in opposition to *work sharing* where work is supplied without being requested [3]. (Note that in the remainder of this text, the expression 'work sharing' will mean simply that work is shared.) One way to generate work to share is to expand one of the worker search tree nodes, by picking one variable and splitting its domain, creating two or more new nodes, some of which will be parted with.

The traversal of the search space in sequential depth-first search is controlled by the selection heuristics, which determine the sub-search space the search will move to next. Even if good heuristics have been identified for some problems, which help steer the search more efficiently towards an answer, the introduction of some perturbation in the process may prove beneficial. Evidence for this lies in the fact that in parallel search it is not always the process which mimics the sequential search the first to be successful in finding a solution. If that was the case, the only place where parallelism would be useful in constraint solving would be the propagation phase.

So, work distribution, search control, and, consequently, performance are closely linked together.

The issue is mentioned in an early work by Van Hentenryck on parallel search for constraint solving, which drew on the work being done on or-parallelism within the field of Logic Programming. CHIP [61], a parallel constraint logic programming language, was implemented on top of the logic programming system PEPSys, whose or-parallel resolution infrastructure was adapted to handle the domain operations needed for constraint solving. The programmer was in control of work division, which could be performed by labelling, domain splitting, or through the addition of new constraints, relying on the scheduler of the underlying execution system to hand out the resulting tasks to the available workers. Reported speedups ranged from 6.14 to 7.24 with 8 workers, and between 3.83 and 5.67 on 16 instances of the graph-colouring problem with 6 workers, with 4 instances showing superlinear speedups of up to 20.41.

Later, and still in the context of logic programming, Mudambi and Schimpf [38] extended the approach to networks of heterogeneous computers. Besides the workers, all running the same program on distinct machines, there was a scheduler running which, upon being asked for work, would request it from the active workers and pass it on to the idle worker. Search space splitting was the consequence of explicit *parallel choice points* and the work sent would correspond to the first one still on the stack, *i.e.* the one closest to the root of the worker search tree. Work was shared in the form of an oracle, allowing the receiver to recreate the state of the sending worker just before the choice point was reached. Workers could then be searching in any region of the search space. On a 12 computer network, the authors observed speedups between 5.37 and 10.79 for a set of benchmark programs, though some were plain logic programs, not using constraints.

Schulte [52] describes a high-level implementation of a parallel constraint solver (in Oz [56, 62, 37], a language supporting constraint, concurrent, and distributed programming, whose runtime system handles the communication and distribution issues), along with the algorithms used to coordinate search and optimisation. The solver consists of a manager and of the workers, which may be distributed across machines, and all communication goes through the manager. Workers are asked to share work and, if they agree, give away one of the search space nodes in their pools (the topmost one, in the benchmarks ran). Work sharing is based on recomputation: search spaces are transmitted as paths from the root of the search tree and must be recomputed by the receiver. Branch and bound search is implemented by having the manager broadcast the current best solution to all the workers. Speedups obtained for 6 workers were between 3.17 and

3.81 in three cases and 5.21 in a fourth.

In [43], Perron, expanding on previous work on new/user defined search strategies [42], conducts a series of experiments on small-scale parallelism in ILOG [23], a commercial product incorporating a parallel constraint solver, later acquired by IBM. One of the remarkable aspects of the data presented is the variability of the times obtained when using parallelism with 2 to 4 workers. (Another one is a solution for a previously unsolved problem in the latter reference.) The two problems studied were both optimisation problems, with the smaller instances not showing any gains from the parallelisation, contrary to the bigger ones, for which both the total running time and the time to find an optimal solution generally improved, as did the rate at which better answers were found. The only detail given on the parallel implementation is that, besides the workers, there is a communication layer responsible for load balancing and termination detection.

Increasing the scale of the approach, experiments on a network of 64 uniprocessors are reported by Jaffar *et al.* in [26]. The architecture of the solver consists of one master and a possibly dynamic number of workers, structured as a perfectly balanced (binary) tree, with the master at the root. Workers, which comprise a communication thread and a search engine, keep a pool of nodes corresponding to subtrees not being explored, and when starting on a new node decide whether to explore it or to split it and add one branch to the pool. When a worker becomes idle, a request for work is forwarded to the master. The master then sends it down the tree and receives an indication of the worker which has most work to share, measured as the number of uninstantiated variables in its pool nodes. The information is sent to the idle worker, so it can request the work directly. All communication flows along the tree edges, except for the latter exchange. Using 61 workers, speedups of about 40 to 61 are reported on all solutions search, and between none and above 40 for branch and bound search.

A coordination based approach is taken by Zoeteweij and Arbab [70] to implement parallel constraint solving. Independent solvers are run on distinct machines for a certain amount of time. When that time elapses, every solution found and the nodes still unexplored are sent to the coordination layer, which collects the solutions and distributes the work by all idle solvers, following some specified criterion, which may depend on the number of nodes available, favouring either a depth- or a breadth-first step. This contrasts with all the previous approaches, where work sharing was initiated by the idle workers. Speedups between 11 and 15 were obtained on a 16 node cluster.

Nielsen [39] describes a multithreaded implementation of parallel search for Gecode [18], a well known public domain object-oriented constraint solver, with a controller plus workers architecture. All control and work sharing goes through the controller. Sharing through copying and recomputation are both considered. In the actual implementation, which first appeared in Gecode version 3.1.0, released in 2009, there is no controller and workers poll each other for work.

In a higher-level proposal, Michel *et al.* [32, 33] introduce parallel constraint solving into Comet [10] by adding a parallel controller which manages the workers. The workers explore their search space and may request further work from an active central pool when finished. Work requests are forwarded to a further unspecified worker which generates subproblems and sends them to the pool. Each worker uses an exploration controller, which guides the exploration of its search tree, and a generation controller, which generates new subproblems from the root of the unexplored part of the local search tree and puts them in the pool. The splitting strategy may be the same or different from the local search strategy, as it depends on the generation controller. Parallelisation is not completely transparent as the parallel controller must be explicitly invoked. The implementation runs on a multiprocessor, and several benchmark programs were tried with 4 workers, resulting in speedups of between 2.21 and 3.68, with four instances of optimisation problems giving rise to superlinear speedups of up to 4.76. [34] sketches an extension to a distributed setting, but details are scant. Apparently, a master process holds the work pool, queries the workers when it becomes empty, and serves work on request. Experiments with 16 distributed workers give speedups between 9.25 and 14.86 when the full search space is explored, and superlinear speedups of up to 42 090 occur when finding the best answer to an optimisation problem.

Rolf and Kuchcinski compared copying and recomputation based work sharing approaches [48], as well as heuristics for deciding which worker will share work [47], in a setting where workers communicate without intermediation. These heuristics require some kind of distributed agreement on which worker will be selected, hence curtailing on scalability. The maximum observed speedup on 32 computers was of about 23, slightly above that corresponding to random polling workers for work. The solver seems to consist of either shared memory workers or distributed single worker processes.

To help guide work division between the workers, Chu *et al.* research a *confidence* based scheme for work stealing in parallel constraint solving [8]. They associate with each node of the search tree the confidence with which the branching heuristic for the problem should be regarded at that node, reflecting the estimated ratio between the solution densities of both its subtrees. When the confidence is high, workers will tend to follow the heuristic and work on the left subtree; if the confidence is low, workers will be distributed across the tree. The work finding algorithm adapts as the search proceeds: as workers finish working on a subtree (without finding a solution), the confidence of the ancestor nodes is updated. Finding which subtree to work on next requires descending the tree from the root, guided by the nodes confidence and by where in the tree the other workers are working. A timed restart mechanism allows workers to stop their current search and to try to find a more promising search space to explore. Experiments on computers with 8 cores revealed good speedups but, interestingly, for most examples, the less the confidence in the heuristic was, the better were the results obtained. In what concerns scalability, potential drawbacks of this approach are that it requires a global view of the state of the solver, including the knowledge of where in the search tree each worker is searching, and that confidence updates must be propagated along one branch of the tree.

Experiments on as much as 128 processors, using both search space splitting and portfolios, were performed by Bordeaux *et al.* [4]. As mentioned before, portfolios try to explore the unpredictability of the behaviour of backtracking depth-first search. In the paper, the different strategies executing in parallel differed in the first three variables to be branched on. Search space splitting was achieved through hashing constraints, which divide the search space on the basis of the sum of the values of the variables in their scope. Counting all solutions of one problem exhibited linear speedups up to 30 processors, but no improvements beyond that, up to 64 processors. It should be pointed out that, although the workers run on different machines, load balancing is only attempted in the initial splitting. After that, there is no further communication involved, apart from termination signalling.

Schubert *et al.* [51] combine multithreading and distribution for SAT solving, with a multithreaded solver running on each multiprocessor of a network. Workers communicate locally through a shared (C++) object, where they check whether any worker is idle, in which case they split their search space, place it there, and wake the idle worker up. The distributed solvers are coordinated by one master process, which waits for work requests from them, checks the load of each solver (locally, as it handles all work distribution), chooses the most loaded to share work, asks it for work and forwards it to the idle solver. Speedups on a 3 node network, with 8 workers in total, reached a maximum of 5.62, and the single master architecture does not seem appropriate for the use on larger networks.

Real large-scale parallel constraint solving is investigated by Xie and Davenport [64] on a particular architecture, the IBM Blue Gene /L and /P, which support several thousand processors. One of the particularities is that inter-processor communication is MPI based and, from that point of view, the system constitutes a huge cluster. Their solver architecture consists in at least one master process and several worker processes, all located on distinct processors. Each worker runs a constraint solving engine and the masters coordinate work sharing, keeping a tree-structured job pool from where work is sent when a worker asks for it. The job tree reflects the structure of the problem search tree, allowing the master to follow some search strategy when selecting the next search space to be explored, and work is communicated as a tree path, from the root to the corresponding node. Feeding the pool is the responsibility of the workers, which relinquish part of their workload once they have performed a set amount of work. Speedups grow linearly on one scheduling problem as processors increase from 64 to 256, when there is only one master, but decrease from then on; with more than two masters the solver exhibits improving speedups up to 1024 processors, but at the expense of noticeable overheads. In this case, the search space is statically divided initially among the masters, as evenly as possible in a domain specific way, and there is no further work exchange between them. Good speedups, many superlinear, are obtained with other problems as well.

Some Notes

The works described above define the state of the art in parallel constraint solving. Several trends may be observed. One is the growing tendency to move from worker initiated work stealing [38, 52, 32, 33, 51] to a model where workers give away part of their search space voluntarily [70, 64]. This strategy is usually associated with master/slave architectures [38, 26, 4, 51, 64] and, while this obviates the need to search for work to supply an idle worker and simplifies the implementation task, it may also cause workers to become needlessly idle.

Another, and somewhat related, trend is the increased focus on distribution [38, 52, 38, 70, 47]. The fact that computers with more than 16 cores are still relatively uncommon, unlike networked computer clusters, may serve as an explanation. On the other hand, many of these distributed solvers are built out of sequential solvers, executing independently on the networked processors [38, 38, 70, 47, 4, 64], and in some cases no dynamic load balancing takes place [4] or only in a limited form [64]. When present, load balancing usually favours sharing the largest available search space, except in a few cases where a more elaborated scheme is employed [32, 33, 47, 8, 64].

2.5 A Distributed Parallel Constraint Solver

The object of this thesis is PaCCS, the *Parallel Complete Constraint Solver*, a constraint solver based on search space splitting search, comprising distributed components. While its detailed analysis is the subject of the next chapter, this section highlights the features that, as a whole, distinguish it from the other solvers.

PaCCS is a distributed parallel solver PaCCS was designed to perform parallel search on networked multiprocessors. This means that co-located workers, executing on the same node of the network, work tightly coupled to each other, exploiting the sharing of physical resources and the fast communication channels available. Coupling between remote agents is looser and there is no direct communication betwixt them.

There is no master process All processes making up the solver share the same tasks: solving the problem and sharing work. During the solving procedure, no process has the role of coordinating the other processes. Nevertheless, there is a distinguished process which initiates the search, collects solutions, detects termination, and returns the answers. Still, other processes may participate in some of these tasks.

Workers steal work directly There is no intermediation in work sharing among co-located workers. When a worker exhausts its search space, it steals another from one of its co-workers, which continues its search process unperturbed.

All work is shareable As they do not need to be interrupted when work is stolen from them, workers do not need to interrupt the search to create shareable work. Every time the search engine branches, the branch not taken could be immediately shared. The creation of shareable work goes hand in hand with the search process, instead of being an extraneous task. One consequence of this design decision is that there is no backtracking; when the search fails, the worker simply steals work from itself.

Multiple search space splitting points Search may be very much influenced by where workers start searching initially. The most common approach is to hand the full search space to one worker, as the others start in work stealing mode. PaCCS, however, splits the search space in two places before beginning the search, the first to divide it among the processes and the second to give each worker its share. These two splits may apply different strategies and may be taken advantage of to help the behaviour of the search, *e.g.* by dividing the work more equally between the processes, trying to reduce inter-process communication, or by starting a worker deeper in the search tree and potentially closer to finding a solution.

Chapter 3

The Parallel Complete Constraint Solver

This chapter presents PaCCS, the Parallel Complete Constraint Solver, in detail, along with its goals, guiding principles, algorithms, and design choices. Some implementation related issues are also discussed.

Section 3.1 recalls the motivation behind this work and the following section contains an overview of the main features of PaCCS, which is expanded upon in Sections 3.5 through 3.7, while the solver behaviour and structure are described in Sections 3.3 and 3.4, respectively. Finally, the last section takes a closer look at lower-level implementation related issues.

The solver programming interface, through which constraint satisfaction problems are specified, is presented in Appendix A.

3.1 Motivation

When this work started, although parallel search was already a focus of research, it was apparent that there was still room for improvement. As it is still the case today.

Two main issues drove the endeavour that lead to PaCCS. The first was the wish to push the envelope of the pertinence of parallelisation in constraint solving, fully exploiting the current hardware trends, including tackling large-scale systems, with hundreds or thousands of processing units. The second was to develop a scalable lightweight high performance constraint solver, allowing users to transparently take advantage of the power of parallelism and distribution, hidden under the hood of the declarative nature of constraint programming.

How these goals are accomplished by the work presented in this thesis is discussed in the remainder of the chapter.

3.2 PaCCS in Short

PaCCS realises parallel constraint satisfaction problem solving on distributed memory systems, where parallel solving means the simultaneous exploration of distinct parts of a CSP search space in parallel.

The following is an overview of the main features of the solver.

- The main entity in PaCCS is the *worker*. A worker is a possibly sequential procedure solely devoted to processing search spaces, *i.e.* a search engine, playing its part in a wider effort to find an answer to a problem. As such, once a worker is given work, it will carry on processing until it runs out of work or an answer is found.

- The search engine implements a complete constraint solving method, interleaving rule-based propagation [2, 53] and search [60].

- Variable domains reside in a *domain store* [16], a compact data structure designed with work sharing and locality of memory accesses in mind. The latter aspect is particularly relevant as *all* the computation the search engine performs involves these data.

- The CSP, *i.e.* its variables and constraints, has a similarly compact internal representation, again to improve locality of memory references.

  Since a CSP is completely defined by its variables, their domains, and the constraints, these last two points allow for readily turning PaCCS into a server providing a constraint solving service.

- PaCCS is targeted at multiprocessing systems, be they multiprocessors, networked computers, or a combination thereof. To effectively handle the differences between both kinds of systems, the solver has a two-level architecture.

  On the lower level, meant to explore the sharing of resources, workers are tightly coupled, functioning as a *team*. The higher level provides a medium for inter-team coordination of the search effort.

- One important part of parallel solving is the assignment of work to the workers, which translates to the splitting of search spaces. The two-level architecture exhibits three places where it may be carried out, providing more degrees of freedom in the definition of the search strategy. First, the full search space must be divided among the teams. Then, all workers in a team must be assigned their share. Finally, each worker may create work that can be shared with other workers, within or outside the team.

- Dynamic load balancing takes place throughout the solving process. As long as there is work to do within a team, it may be shared by all its workers. Once it runs out, the other teams are probed for some search space to share, so no team stays idle while the search proceeds.
- Intra-team work sharing is accomplished directly by the workers, without any intermediation.
- In order to have a finer control over the solver internals, namely over memory usage, PaCCS is implemented in C.

What makes the approach followed here unique is partly the combination of multiple levels of parallelism, full dynamic load balancing, and seamless cooperation between co-located workers. While some of the earlier works, reviewed in Section 2.4, display some of these features, none has so far, to my knowledge, managed to bring them all together. Furthermore, default generation of shareable work and the nature of the interaction among teams are other distinguishing attributes of PaCCS.

3.3 Solver Dynamics

The solver operation has two phases: the specification of the problem and the solving. Specifying the problem consists in defining the variables and associated domains, and posting the constraints. Once this is done, the solving process may start.

At the beginning of the solving process, the search space is split into as many disjoint parts as there are teams, and the parts are delivered to the teams. Within each team, the received search space is again split so that each worker has a search space to explore.

A worker that has been alloted work executes it until either an answer has been found or it runs out of work.

The solver may pursue one of three goals:

- *Finding one solution to the problem.*

  Since one solution is all that is wanted, as soon as one is found all workers are stopped and the solution is returned. Failure is returned for inconsistent CSPs.

- *Finding and counting all solutions.*

  In this case, all the solutions are generated, but only their count is returned.

- *Finding a best solution.*

  This task follows along the lines of the algorithm shown in Figure 2.5. When a better solution is found, its value under the objective function is communicated to all workers in all teams, so they will know to only look for better solutions. In the end, either a best solution or failure will be returned.

For the latter two tasks, and for the former when the problem is inconsistent, the search process will only stop when the full search space has been explored and all workers have become idle. A worker that finishes exploring its assigned search space while the search is still in progress tries to obtain additional work from its teammates. If this is not possible, it means that the team is out of work and the other teams are tried.

3.4 Solver Architecture

A global view of the solver architecture is presented in Figure 3.1.

PaCCS consists of *workers*, grouped together as *teams*. Each active worker keeps a *pool* of *idle* search spaces and a *current* search space, the one it is currently exploring. In each team there is a *controller*, which does not participate in the search, and one of the controllers, the *main controller*, also coordinates the teams.

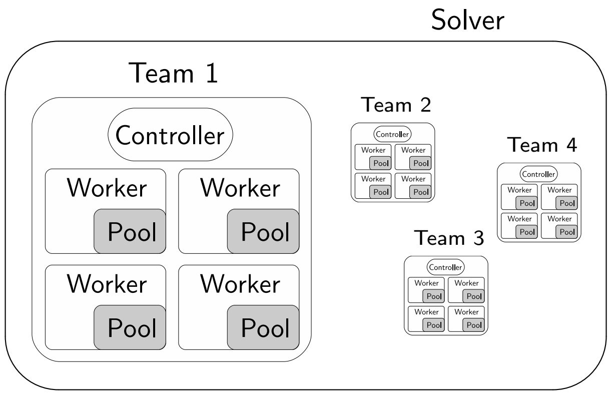

Figure 3.1: Solver architecture

PaCCS displays a two-level architecture. The bottom layer of the system comprises the search engines, embodied by the workers. The main role of the layer above is managing the communication between the teams and making sure the solver progresses towards an answer.

Structuring the workers this way serves two purposes. The first is that a worker's sole task becomes searching, as all communication with the environment required by the dynamic sharing of work among teams is handled by the controller. The second objective is the sharing of resources enabled by binding the workers in a team close together. If all workers were on the same level, either they would have to divide their attention between search and communication, or there would have to be one controller per worker, thereby increasing resource usage. On the other hand, this structure matches naturally a two-level partitioning of the search space.

3.5 Workers and Search

Even if the worker is the most fundamental building block of PaCCS, its interaction with the remainder of the system is restricted to being given a search space to scan, returning results, and waiting for more work.

Workers implement a constraint solving engine which interleaves propagator driven domain-based consistency enforcement [2, 53] with variable instantiation.

The search unfolds as the worker splits the search space it is working on, keeping one part as its current search space and adding the other to the pool, and propagates the change in the domain of the branching variable onto the new current search space.

There is no concept of backtracking. If the current search space is found to contain no solution, the worker draws a new search space from the pool and starts exploring it. As implemented, however, the pool acts like a stack and the search space retrieved is, whenever possible, the one most recently put there by the worker, leading to a behaviour that mimics chronological backtracking search. Together with splitting the search space by dividing the domain of one variable into a singleton in the new current search space and the remaining elements in the sub-search space sent to the pool, this gives variable labelling [1]. Nevertheless, there is the potential to use other strategies for choosing the next search space to fetch from the pool.

Figure 3.2 depicts the main driver algorithm for workers without optimisation. Initially, every variable domain is revised against every other variable domain (line 2). At each further step of the search process, a worker starts by looking within its current search space for a variable whose domain is not a singleton (line 3). If none is found, then the search space consists of a single tuple which constitutes a solution to the problem, and which is returned by the worker (line 10). Otherwise, and depending on the variable selection heuristic in use, one of the variables with a non-singleton domain is selected and the current search space is split into two subspaces (line 4) such that:

- In the first, which will become the worker current search space, the selected variable is set to an individual value picked from its domain, according to the chosen value selection heuristic.
- In the other, to be added to the pool of idle search spaces (line 5), that value is removed from the domain of the variable.

The domains of the other variables remain unchanged in both search spaces.

Following the split, the new current search space goes through a propagation phase (line 6). If it succeeds, another search step is performed. If the propagation fails, the worker tries to fetch an idle search space from the pool to become its current search space (line 7). If this is not possible the worker fails (line 9), otherwise the search resumes with the retrieved search space undergoing a propagation phase, as the domain of one of its variables shrank just prior to its being stored in the idle pool.

```
1: WORKER$_0$(search-space)
2:    current $\leftarrow$ filter-domains(search-space)
3:    **while** (var $\leftarrow$ select-variable(current)) $\neq$ FAIL **do**
4:       (current, other) $\leftarrow$ split-search-space(var, current)
5:       pool-put(other, var)
6:       **while** (current $\leftarrow$ revise(current, $\{$var$\}$)) = FAIL **do**
7:          (current, var) $\leftarrow$ pool-get()
8:          **if** current = FAIL **then**
9:             **return** FAIL
10:   **return** SOLUTION(current)
```

Figure 3.2: Worker main driver algorithm (without optimisation)

If all solutions to a problem are to be generated, the worker is enclosed in a wrapper which counts the solutions as they are found. When the worker eventually fails to find one, the wrapper returns the number of solutions found.

The state of a worker with two search spaces currently in the pool is shown in Figure 3.3, where solid edges mean that the children search spaces constitute a partition of the parent. Notice that the subtree to the left of the current search space (corresponding to the tuples where both x_1 and x_2 take value 1) is not displayed, as it has either already been explored and discarded, or pruned by propagation (which would be the case if x_1 and x_2 were constrained to be different from each other, for example).

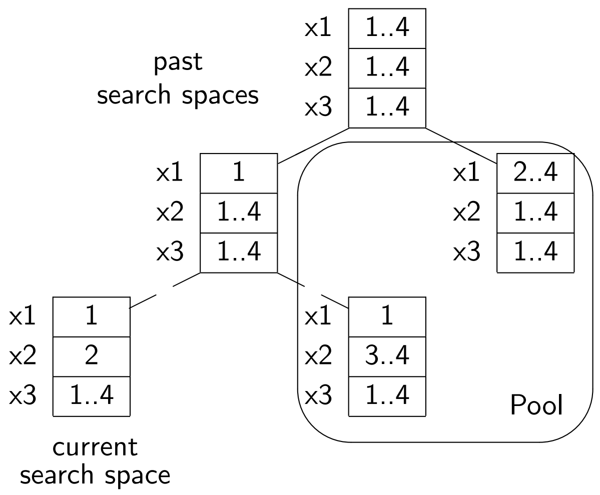

Figure 3.3: Search spaces from a worker

Handling optimisation problems requires adding the lines *6 and *9 in Figure 3.4.

The function propagate-bound updates the domain of the bound variable[1] in the search space that was just brought back from the pool, as it may have been generated before the current best solution was found. The update consists in removing from the domain of the variable all values which are not better than the value of the current best solution. If any value is actually removed from the domain, the change is propagated to the remaining variables. The value for the current best solution is assumed to be global knowledge of the process executing the worker code. (The optimising flag is a device to unify the constraint satisfaction problem and the constraint optimisation problem solving procedures.)

```
1:  WORKER(search-space)
2:     current $\leftarrow$ filter-domains(search-space)
3:     **while** (var $\leftarrow$ select-variable(current)) $\neq$ FAIL **do**
4:        (current, other) $\leftarrow$ split-search-space(var, current)
5:        pool-put(other, var)
6:        **while** (current $\leftarrow$ revise(current, $\{$var$\}$)) = FAIL **do**
*6:          **do**
7:              (current, var) $\leftarrow$ pool-get()
8:              **if** current = FAIL **then**
9:                 **return** FAIL
*9:          **while** optimising $\wedge$ propagate-bound(current) = FAIL
10:    **return** SOLUTION(current)
```

Figure 3.4: Worker main driver algorithm (with optimisation)

3.5.1 Propagation

Domain filtering, or *propagation*, in PaCCS, is rule-based [53] and each constraint has two *propagators*. One, used when exploration of a search tree branch is starting, establishes some local consistency by filtering out values from the domains of possibly all the variables in the scope of the constraint. At the same time, it may also initialise data structures which will support performing propagation incrementally as the branch continues to be explored.

The second propagator will be called afterwards, every time the domain of a variable in the scope of the constraint changes, to propagate that change onto the domains of the other variables involved in the constraint. Incremental propagators will make use of the data available from earlier applications of the constraint's propagators to perform

[1]Recall, from Section 2.2.1, that bound is the name of the optimisation problem variable constrained to hold the value of the objective function.

domain filtering more efficiently.

The level of local consistency enforced by a propagator depends on the constraint. While some will seek to attain arc consistency, others may settle for bounds consistency or other weaker form of local consistency.

The sole propagation event [53] currently considered in PaCCS is the (unspecified) change of a domain.

3.6 Search Space Splitting

The team controllers are the link between the workers in different teams. They handle incoming and outgoing work, optimisation bounds, and work requests, buffer the workers answers, and start and stop workers. Besides these tasks, the main controller also receives the CSP to solve and delivers the final answer.

Additionally, controllers play a part in the search control. Since work distribution passes through them, it is there that the problem is partitioned before being delivered to the workers. The first splitting is the responsibility of the main controller, and the obtained subproblems are passed on to the teams.

The strategy used to partition the search space may have a decisive impact on the number of steps needed to arrive at a solution, hence on performance. This is especially true when looking for one solution or solving an optimisation problem, up until the moment where a best solution is found.

Partitioning strategies may be designed either to lead to a balanced distribution of the search work, like the *even* strategy below and the prime and greedy strategies from [55], or to produce some subproblems where the search is expected to be quick (while in others it may be slow), such as *eager* partitioning. In principle, the former strategies will be more suited to situations where all solutions are requested and the whole search space must be visited, and the latter will lend themselves better to when one solution is all we want. In any case, the splitting of the problem will introduce a breadth-first component into the usual depth-first exploration of the search tree, which sometimes gives rise to superlinear speedups.

Even partitioning

In *even partitioning*, domains are split so as to obtain sub-search spaces of similar magnitude. Suppose we want to split a problem into k subproblems. If the next variable domain to be divided has at least that many values, it is split as evenly as possible among the subproblems: as it has $d \geq k$ values, it will have $\lfloor d/k \rfloor$ values in the first $k - d \bmod k$ subproblems and $\lfloor d/k \rfloor + 1$ values in the remaining $d \bmod k$ subproblems. Otherwise, the problem is split into $d < k$ subproblems and, recursively

- the first $k \bmod d$ subproblems will be each split into $\lfloor k/d \rfloor + 1$ parts, and

- the remaining $d - k \,\mathbf{mod}\, d$ subproblems will be each split into $\lfloor k/d \rfloor$ parts, along the same lines.

Example 3.1 *Even partitioning a problem where the domain of all variables is* $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ *into six subproblems would lead to the result depicted in Figure 3.5. In the first two subproblems, the domain of variable* x_1 *would have only value* a*, whereas the domain of* x_2 *would be* $\{\mathsf{a}\}$ *in one and* $\{\mathsf{b}, \mathsf{c}\}$ *in the other. The remaining subproblems would differ from these in the domain of* x_1*. (Only the variables whose domains are affected by the splitting are shown.)*

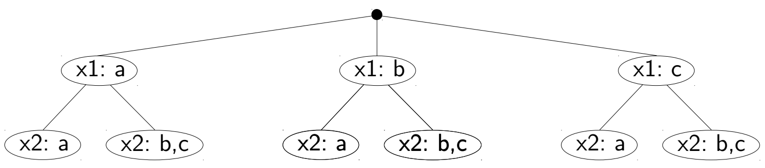

Figure 3.5: Even partition into 6 sub-search spaces

Eager partitioning

Eager partitioning corresponds roughly to a partial breadth-first expansion of the search tree and it will mostly produce subproblems where at least one of the variables has had its domain reduced to a single value. The splitting is performed according to the algorithm depicted in Figure 3.6, whose inputs are the number of subproblems to create and a sequence of problems from which to create them. Initially, this sequence only contains the original problem.

Example 3.2 *Figure 3.7 shows the result of applying eager partitioning to split into six parts the problem from the previous example, where all variables have domain* $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$*. (Again, only the variables whose domains are affected by the splitting are shown.)*

The partitioning of the CSP may affect the behaviour of the search, even to the point of defeating the variable and value selection heuristics which are usually appropriate to a given problem, as has been noted in [32, Section 6]. This suggests that the partitioning strategy, introducing another degree of freedom in the search strategy, needs to be adapted to the problem being solved and matched with the search heuristics used, and that no overall 'best' partitioning strategy exists.

As problem partitioning takes place at two points in the process — to distribute work to all the teams and, initially within every team, to assign work to each worker —, different splitting strategies can be used while solving a problem: a more balanced one to allot (expected) similar amounts of work to the individual teams, and another

Notation *If P is a CSP and $V \subseteq D_i$ is a finite set, PV_i stands for the CSP which is identical to P except that the domain of the i^{th} variable is V.*

```
eager-split($k, (P_1\ P_2 \cdots P_q)$)
    $(X, D, C) \leftarrow P_1$
    $i \leftarrow \min \{j \mid |D_j| > 1\}$
    $d \leftarrow |D_i|$
    $\{v_1, v_2, \ldots, v_d\} \leftarrow D_i$
    if $k \leq d$ then
        return $(P_1\{v_1\}_i\ P_1\{v_2\}_i \cdots P_1\{v_k, \ldots, v_d\}_i\ P_2 \cdots P_q)$
    else
        return eager-split($k - d + 1, (P_2 \cdots P_q\ P_1\{v_1\}_i\ P_1\{v_2\}_i \cdots P_1\{v_d\}_i)$)
```

Figure 3.6: Eager partitioning algorithm

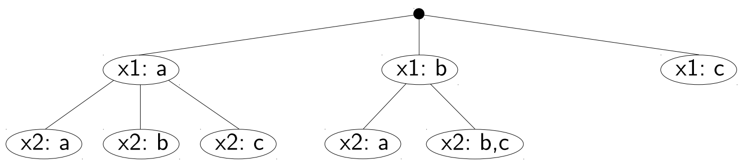

Figure 3.7: Eager partition into 6 sub-search spaces

to focus the efforts of the workers. The latter strategy could be finer grained than the former, the cost of compensating for the imbalance introduced being much lower within a team than between remote teams, where it requires interaction over the network.

Additionally, in parallel search, different teams might split their problems differently, allowing us to take advantage of one not yet identified strategy being more effective than the others for the problem at hand, in the spirit of the portfolio approach [4].

3.7 Work Stealing

Realising the potential of parallel constraint solving requires that the workers be kept as busy as possible throughout the entire computation. As knowing in advance the amount of work a given search space represents is very hard, there is the need to enable dynamic load balancing, in case a worker becomes prematurely idle. The technique exploited here is *work stealing* [3], or receiver-initiated decentralised dynamic load balancing [63], where an idle worker will search for a busy one willing to share some of its work.

When a worker tries to fetch a new search space from its pool and finds it empty, it means it has exhausted exploring its assigned search space, and it will then attempt to obtain a subproblem to solve from one of its teammates. Failing that, work must be obtained from a different team.

As a consequence, there are two kinds of work stealing in PaCCS, *local* and *remote work stealing.* The former takes place among close by workers, and the latter between distant workers, being more expensive to accomplish as it happens over slower communication channels. Regarding the pools of the workers belonging to the same team as a global *team pool* of work, contacting remote teams should not be engaged in unless that pool is empty.

3.7.1 Local Work Stealing

Local work stealing, *i.e.* work stealing between workers from the same team, should have the least possible impact on the operation of the busy workers, namely on the worker from which it is stolen, the *victim.*

Work stealing is carried out through the pool, transparently to the workers affected, which corresponds to viewing the individual pools together as a team pool. From the perspective of the worker, its operation consists solely of solving the problem it has been charged with, while putting and retrieving subproblems from the pool. It is up to the implementation to make sure that:

- until the moment the team pool goes empty, the worker always succeeds in fetching a search space from the pool, be it one it put there or not; and
- when the worker pool is empty, work is obtained from another worker pool, without the latter being disturbed by the action.

What follows is an example shared-memory implementation of a work stealing scheme abiding by the above guidelines.

Implementing Local Work Stealing

To ensure that local work stealing has the least detrimental effect on the performance of the solver, it must be achieved with as little cooperation from the holder of the retrieved search space (the victim) as possible. In fact, in this implementation, the idle worker will effectively steal work from a teammate while the latter continues its task, oblivious to what is being done to its work queue.

Here, the intended discipline of a worker pool is that of a *deque* (double-ended queue), as depicted in Figures 3.8 and 3.9. While the owner works on one end of its pool (lines 2, 8, and 12), a worker whose pool is empty will remove an entry from the other end (line 25). This way, the only penalty a worker incurs during normal processing is the cost of an extra check on the size of its pool (line 6). The protocol used to avoid interference during pool accesses is similar to the ones in [44, 17]. Only when the number of entries in the pool is small and there is the risk of the worker trying to use a search space another one is stealing, will it be necessary to enforce mutual exclusion in the accesses to the pool, and even then only when removing a search space.

```
1:  pool-put(search-space, variable)
2:      pool.append((search-space, variable))

3:  pool-get()
4:      if pool.size = 0 then
5:          return steal-work()
6:      else if pool.size < SAFE-SIZE then
7:          lock(pool)
8:          entry ← pool.remove-last()
9:          unlock(pool)
10:         return entry
11:     else
12:         return pool.remove-last()
```

Figure 3.8: Pool insertion and retrieval

```
13: steal-work()
14:     worker-is-idle(YES)
15:     work ← FAIL
16:     lock(stealing)
17:     do
18:         if workers-all-idle() then
19:             unlock(stealing)
20:             return FAIL
21:         v ← random-worker()
22:         if v.pool.size ≥ THRESHOLD then
23:             lock(v.pool)
24:             if v.pool.size ≥ THRESHOLD then        ▷ Confirm victim has enough stores left
25:                 work ← v.pool.remove-first()
26:                 worker-is-idle(NO)
27:             unlock(v.pool)
28:     while work = FAIL
29:     unlock(stealing)
30:     return work
```

Figure 3.9: Work stealing algorithm

To reduce contention, work stealing is only allowed from a pool when the number of entries in it reaches a given threshold (lines 22 and 24).

Refining the Implementation

The above implementation ignores any characteristics of the search spaces when deciding which one to steal. This increases the likelihood of stealing a search space corresponding to little work, leading to both its recipient and the victim becoming idle shortly after. While this may have a low impact when only local workers are involved, as the cost of stealing and of retrieving a search space is of the same order of magnitude, if work is to be sent to another team it may pay to try to choose one more carefully, to compensate for the added cost of sending it to a remote location.

Even if far from accurate, the size of a search space, *i.e.* the product of the sizes of the domains of the variables, provides a rough estimate of the amount of work required to explore it. (Another possible estimate is the number of uninstantiated variables [26].) This could be used to try to select the most loaded worker and steal from it, if possible, as shown in Figure 3.10.

```
steal-work'()
    **if** workers-all-idle() **then**
        **return** FAIL
    lock(stealing)
    v $\leftarrow$ worker-with-most-work()
    lock(v.pool)
    **if** v.pool.size $<$ THRESHOLD **then**
        work $\leftarrow$ FAIL
    **else**
        work $\leftarrow$ v.pool.remove-first()
    unlock(v.pool)
    unlock(stealing)
    **return** work
```

Figure 3.10: Size-based work stealing algorithm

Such an approach makes it more expensive to steal a search space, since their sizes have to be computed, but the cost will be offset if it results in less small search spaces being stolen.

Another estimate of the load of a worker is the number of search spaces its pool contains.

Some Comments

As the stolen search spaces are the oldest ones in a pool, they correspond to locations nearer the root of a worker search tree, introducing the breadth-first component into depth-first search.

The search within the worker search space proceeds according to the heuristics deemed adequate to the problem until it either finds a solution or the work is exhausted. Upon stealing work from a peer, a worker picks up the search at a point that the worker it was stolen from would not reach until having finished exploring its current branch, thus subverting the problem search strategy and introducing in it a measure of randomness. This may be either beneficial or detrimental, depending on the specific problem.

3.7.2 Remote Work Stealing

In the event of an idle worker failing to obtain work within its team, it notifies the team controller and waits, either to be later restarted or to be terminated. When all the agents in a team have become idle, the team controller will initiate trying to obtain work from another team.

One of the roles of the team controllers is serving requests for work from other teams. A controller trying to supply another team with work will use the same protocol as the workers to steal a search space from the local pool (see the previous section).

Several models of inter-team communication are possible. The two experimented with are discussed next.

Peering Teams

In this first model, all teams communicate with each other. Inter-team work stealing requests may be broadcast to all teams or directed at a specific team.

Initially, one of the team controllers is assigned the role of fulfilling broadcast requests for work; it is the designated *(work) supplier*. The supplier listens for requests broadcast by the other teams and, on receiving one, it will try to steal a search space from its team pool. Then, it will either send the work seeker the stolen search space or a notification that no work is available. The supplier is the only controller that will respond to broadcasts, all the others will ignore them.

A team controller that detects that all its workers are idle, first broadcasts a request for more work to all the other teams. When work is sent in response by the supplier, it is split between the workers, which resume searching. Additionally, this controller will become the new work supplier. If no answer arrives within a set time period or a negative one does, it will start polling the other teams [52]. When a team is polled, its controller will answer by sending work or a refusal to share work.

If, after sending a request for work to an individual team, a timeout period elapses, the next team will be polled. There is an exception to this rule, however, when polling the team where the main controller resides. Since it will be the last team polled, there will be no time limit to wait for an answer.

Once all teams have unsuccessfully polled the main controller, all workers must be idle and all controllers are notified that the solving process has ended. After receiving confirmation that all teams really are idle, the main controller proceeds to collect the answers from all teams and the final answer is returned.

The protocol described above is able to handle delayed messages, such as a team receiving work from another after having being told that the search was over. This is achieved at the cost of a slightly involved termination detection protocol.

Termination detection is much simpler when looking for one answer and one is found: the controller of the finder team forwards it to the main controller, which simply tells all the teams to stop. When solving a constraint optimisation problem, every time a better solution is found, its value under the objective function is broadcast to all teams.

A brief analysis

A simple model of communication was sought when this model was designed.

The purpose of requesting further work by broadcasting and of having a supplier was to have a fast, but possibly brittle, way of obtaining work. But seeing that disruptions are possible, such as two teams becoming idle and broadcasting for work, one, not being the supplier, discarding the other's request and then being the one serviced by the supplier, a more reliable and potentially slower mechanism had to be added.

Behind the definition of this model lay equally the concern to promote the sharing of the search space corresponding to the most work, if one exists. In the worst case, as work starts dwindling down, that search space will be sent to some team. Since this team becomes the new supplier, there is a good chance that it will be further shared with the next team that asks for work, and that it keeps being shared as other teams become idle.

There is the possibility of late answers to work sharing requests, due to communication delays or to a team being too busy to handle incoming messages in a timely manner. This may lead to a team receiving search spaces from more than one team. When this happens, the receiving team will keep the spare ones until either its workers run out of work again or it receives a request for work, broadcast or not.

On the other hand, two teams independently looking for work may both receive some and become suppliers. As a consequence, the next team that broadcasts a request for work may receive it from both suppliers. While this may be the source of a slight inefficiency in the system, it is better to have multiple suppliers than none, as otherwise teams would have to wait for the timeout period to expire before starting polling the other teams.

While exhibiting good results on small clusters, this model presents two major drawbacks, which become more noticeable as the number of teams increases. One is the growth in the number of messages exchanged, especially towards the end of the solving process, when all teams start becoming idle. The other is the rise in the time it takes to unsuccessfully poll all the other teams, before termination can be detected.

One of the aims of this work being to tackle large scale distributed systems, another communication model was developed, able to address these issues.

Team Neighbourhoods

Inspired by the B$^+$-tree data structure [9] and drawing on the good performance of the peering model on small scale clusters, the previous approach was lifted to aggregate the teams into *neighbourhoods*, within which communication proceeds as before. When a team is in need of work, it broadcasts a work request to *its* neighbours and, if necessary, polls *its* neighbours.

Neighbourhoods, which will also be called the shorter *groups*, are in turn joined in further hierarchical neighbourhoods, creating *multi-level neighbourhoods.*

To help make these ideas clear, Figure 3.11 shows a *degree* 2 *multi-level neighbourhood* with 8 teams. At level 0, all the teams, represented by dots, are joined together in two element groups, where the leftmost team is the *group leader.* Level 0 group leaders are put together in level 1 groups with the same size, whose leader is again the leftmost team. Once again, these leaders will form the level 2 groups, and so on until reaching a level with only one group. Now, the controller of the group leader at the highest level of the hierarchy is the main controller and will ultimately coordinate the whole search process. (Note that straight lines in the figure connect different group memberships of the same team.)

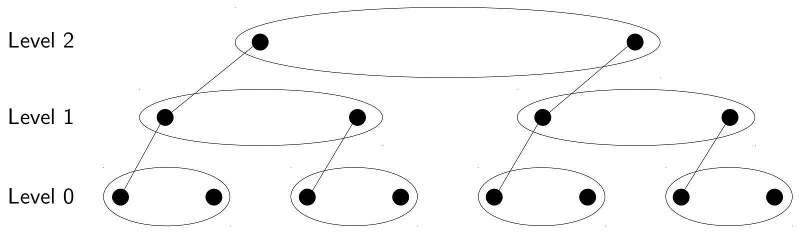

Figure 3.11: Multi-level neighbourhood

The *degree* of a multi-level neighbourhood is the maximum number of elements in a group. Since communication never crosses group boundaries, in a degree s neighbourhood $(s-1)/s$ of the teams can talk to at most $s-1$ other teams and their messages will be contained in their immediate level 0 neighbourhood. Team leaders, which belong to several groups, act as proxies for the remaining group elements. Through them, there exists a path with $O(\log_s N)$ hops between any two teams in an N team system.

Inside level 0 groups, work stealing goes exactly like above, except that the level 0 group leader replaces the main controller. (Polling by a team always starts with the rightmost team and proceeds from right to left.) But, when the group leader is polled, signifying that work in the group is about to be exhausted, and it cannot find work to share in its local pool, it then starts polling the teams from its group one level up for work. These will then either supply a search space from their pool or will try to obtain work from the teams in the level 0 group for which they are the leaders.

If no member of the level 1 group manages to obtain work to share, the level 1 leader will eventually be polled and, if needed, will go up another level in the hierarchy looking for work. There, it will poll the group members, which may poll the teams in the groups where they are the leader, starting at the level 0 group.

If work is once more not found in this level group, its leader will then try the groups above in succession until either work is obtained or all of the teams in the system have declared themselves unable to share a search space. In the first case, the work travels back to the original work requester, being divided along the way among all the teams that meanwhile have also requested work and are waiting. Otherwise, no team in the system is able to share work, and the search process may stop as soon as all workers become idle. This is detected by the main controller, which having failed to obtain work from any of its neighbours on all levels of the hierarchy, notifies all teams waiting for work. This notification is forwarded by the group leaders to all teams expecting work, and sent in response to any new request for work from a team whose workers were still busy.

Termination of the search again follows a three step protocol and must be propagated through the hierarchy. As all communication not local to a group is handled by the groups leaders, it is up to them to collect all answers from the members of the groups they lead, build their own answer, and send it on to the leader of the highest level group they belong to.

Similarly, the value under the objective function of the current best solution of an optimisation problem, found by some team, is propagated along the hierarchy by the group leaders. If it arrives at a team holding a better value, it is overridden.

Only one kind of message follows a different route. If the answer wanted is a solution to the CSP, the team where one is found sends it directly to the main controller. Both teams then start disseminating a message to make the search process stop.

A brief analysis

The main purpose of organising the teams in groups was to limit the number of teams communicating with each other, thus reducing the number of messages exchanged. In fact, experiments have shown it decreasing by two orders of magnitude, alongside with an increase in performance on medium sized clusters.

3.8 Implementation Notes

Some lower level details regarding the implementation of PaCCS are discussed in this section.

All development carried out in the context of the thesis took place on Unix systems. Two general characteristics of the implementation are:

1. Given the sharing of information that was likely to be needed within a team, especially among workers, lightweight processes, allowing for memory sharing, such as the POSIX threads, were chosen as the support for the multiprocessing parallel search implementation.

2. The distribution of the solver is driven by message passing, currently in the form of the Message Passing Interface (MPI) standard, for which several implementations are available.

Accordingly, a team corresponds to an MPI process, where each worker runs in its own POSIX thread, as does the team controller. Since the controller mostly waits for communication, either from the team workers or from the other teams, the workers' threads may map to the processing units available.

3.8.1 Domain Store

The *domain store* [16, 31], or simply *store*, is the internal representation of the set of variable domains of the constraint satisfaction problem. These constitute the dynamic part of the problem, and together with the variables and constraints completely define a search space. As such, the data contained in the store are the basis of all computation pertaining to the solving process and it is important that this is kept in mind when deciding on an implementation.

Additionally, within the framework studied in this thesis, if on one side search spaces are to be transmitted between workers and between teams, on the other, several searches will be unfolding simultaneously and the implementation should promote good neighbouring relations between team members.

All the above aspects benefit from a compact representation, which favours locality of memory references, leading to a good caching behaviour, and which is easy to assemble and disassemble, facilitating the task of remote sharing of search spaces.

Taking this all into account, stores were implemented as a contiguous region of memory, effectively an array of domains, displaying the layout depicted in Figure 3.12.

Variable domains currently have a fixed-size bitmap implementation, possibly with two additional fields containing the domain current minimum and maximum values.

variable 1	domain$_1$	min$_1$	max$_1$
variable 2	domain$_2$	min$_2$	max$_2$
	$\vdots$		
variable n	domain$_n$	min$_n$	max$_n$

Figure 3.12: Domain store

As a store contains all the dynamic information required to define a search space, work stealing is performed by moving stores around. A worker stealing work from a teammate copies a store from the latter's pool to its own. For remote work sharing, stores are sent from one team to another.

The store is thus one of the main units of communication in PaCCS, being used in work stealing between teams, as well as to send solutions. If the size of the store becomes a concern due to the communication costs, it may be reduced either by resorting to compression techniques, or by limiting the amount of information transmitted, taking a recomputation-based approach for inter-team work sharing.

3.8.2 Pool

The worker pools are where the worker stores are kept and their operation is a cornerstone of the solver. On the one hand, the normal mode of operation of a worker involves constantly storing and retrieving stores from the pool. On the other, to have unobtrusive and fast work stealing, all co-located workers will access each other's pools (but only modify their own, except as noted below).

A worker pool occupies a contiguous region of memory and behaves like a deque, implemented with an (extendable) array of stores with two indices, one pointing to the most ancient store in the pool and the other to the next free pool entry (see Figure 3.13). Along with the stores, the pool also holds information about the variable whose domain was split when each store was generated. This way, a worker retrieving a store from the pool or stealing a store knows which domain changed when it was saved, and can propagate that change.

The two indices allow a worker looking for work to know whether a co-worker has enough search spaces to be a candidate to be stolen from. Stealing a search space consists in making the first index point at the next most ancient store. Once it has been done, the thieving worker can then copy the store into 'private' memory. Figure 3.14 shows the state of the pool after a store has been stolen from it.

The impact on the victim is minimal, due only to the update of one of its indices, if

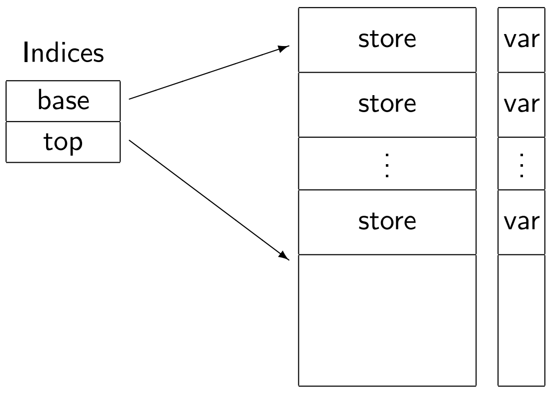

Figure 3.13: Pool

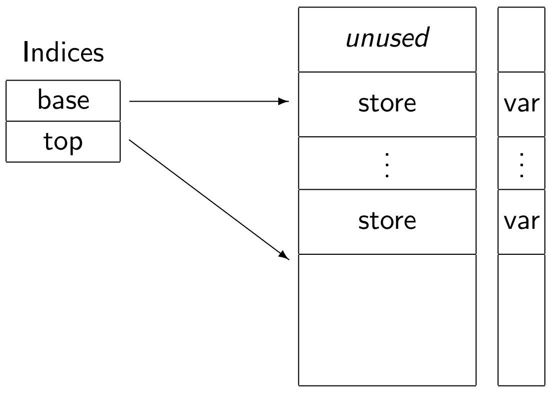

Figure 3.14: Pool after work stealing

the other end of the pool is sufficiently removed. Otherwise, it may find its pool locked and will have to wait for the other worker to finish stealing the store. However, in the experiments performed the latter occurred very infrequently.

3.8.3 Synchronisation Issues

When dealing with cooperating parallel threads of execution on a multiprocessor machine, one crucial aspect that has to be handled is the synchronisation of operations. This usually involves either ensuring exclusive access to resources or common data, or supporting some amount of interaction between the threads.

Both situations potentially entail a performance penalty, as the threads wait to enter a critical program section or for the interaction to take place. Besides, synchronisation enabling primitives also have an impact on performance and their use should be as sparing as possible.

Team Internal Interactions

In PaCCS, and within a team, communication happens only when a worker delivers its final or intermediate search results to the controller. This constitutes a typical producer/consumer scenario and is handled through a pair of semaphores, which serve to both synchronise the workers and controller operations, and to notify the controller that an answer is available.

The result is final when looking for the first solution and one has been found. In this case, the solver will halt, returning the solution. The result is also final, from the worker point of view, when counting solutions and when failing to find a solution. Here, the team assigned work has been exhausted, and workers will all eventually wait on a condition variable [21] while more work is sought from another team.

Intermediate results are the solutions to optimisation problems which, as far as the worker is aware, are the current best solution. After notifying the controller that a new solution is available, the worker waits on a semaphore for the solution to be copied out of its store and for the new bound to be installed team-wide. The need for this last step arises from the fact that the only way for a worker to know the current bound for the objective function is through a team-global variable, which is only consulted after a store is retrieved from the pool (see Figure 3.4). The waiting by the worker ensures that the updated value is available when it restarts searching and, at the same time, prevents it from competing with the controller for the processor while the latter checks whether the new solution really improves the current bound and updates the variable.

Critical Regions

The only other moment where attempts to modify data used by another thread may occur is during the retrieval of a store from the pool, as work stealing involves accessing a data structure owned by a different thread. It is not only the workers that may try to steal work, but the controller may do it as well, when asked for work by another team. The difference in the procedure is small, however, and will be covered below.

As seen in Section 3.8.2, the only foreign data a thread needs to modify to steal a store is the base index of the pool belonging to the worker from which it is being stolen. Other than that, all threads share a count of the number of workers currently trying to steal work. Accesses to the counter are protected by a spinlock, as the lock will only be held while the counter is either being incremented, decremented, or read. A worker increments the counter before starting trying to steal a store, and decrements it on succeeding. If the counter value ever reaches the number of workers in the team, then all workers are out of work and work stealing fails. The main difference between the controller and the workers behaviours lies in that the former does not update the count.

In order to reduce inter-cache traffic, only one thread at a time may seek to steal work, as Figure 3.9 indicates. As locking operations are expensive, looking for a worker with a sufficient number of stores in its pool is carried out without any further access restrictions.

Once a prospective victim has been identified, a lock controlling access to its pool is acquired and, before stealing a search space, its size is once again checked, to confirm that enough stores remain. If that is the case, the stealing thread holds the lock while copying the stolen search space to its working memory. The fact that another thread is removing an entry from its pool will go unnoticed by the victim, unless the number of stores kept there becomes very small. When a pool becomes full, the same lock is used by its owner while it is being extended.

3.8.4 Problem Representation

The same considerations made with regard to stores, with the exception (currently) of the remote sharing, apply to the internal representation of the static part of the constraint problem, namely its variables and constraints.

The data associated with a variable comprise its index number, linking it to the location of its domain within a store, an estimate of the number of variables with which it shares a constraint, used by one of the variable selection heuristics, and the constraints it appears in.

The representation of a constraint includes its index number, information identifying the kind of constraint in question, and the variables it concerns. The constants involved in the specification of some of the constraints will also be included in their representation.

Figure 3.15 depicts the layout chosen to represent the static part of the problem, which assembles the above data into a compact format. Its design attempts, once again, to promote good locality of memory accesses.

Note that, within a team, only one instance of the problem exists, being shared by all its workers. The additional temporary memory required by some of the constraints for performing incremental propagation has dynamic contents, depending on the local state of the search within each worker, and is neither part of the problem representation nor shared between workers.

The problem representation shown in Figure 3.15 may be completely reconstructed from the information contained in the data structure on the right-hand side of the figure. In consequence, the problem representation is easily relocatable, which makes it also appropriate for implementing the solver as a CSP solving service.

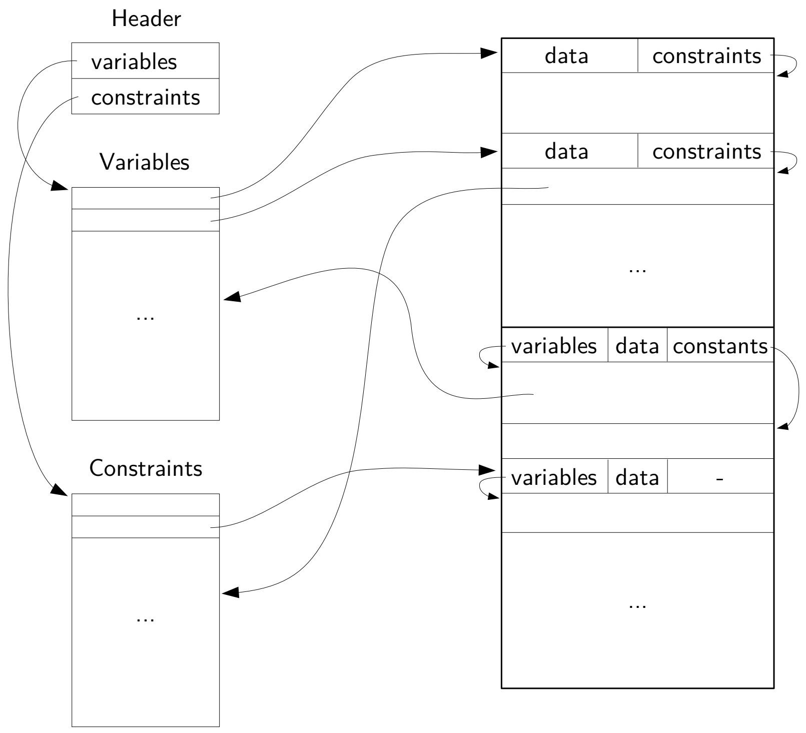

Figure 3.15: Internal problem representation

3.8.5 Memory Model

The memory shared by the team members comprises the CSP representation, the pools, the locks used for synchronisation and for enforcing mutual exclusion, and the last best value of an optimisation problem.

Private worker memory includes the current search space, constraints' dynamic data, and the queue of pending revisions.

Chapter 4

Experimental Results

This chapter reports on the results of the experiments performed with PaCCS on several systems, and compares them with other solvers.

One of the difficulties encountered is the general unavailability of distributed parallel solvers. One known instance is COMET [10], although it is unclear whether the current version already features distributed solving. Besides, the parallel tests made with it on a multiprocessor gave inconsistent results, and often crashed. So, comparisons must be made based on the results reported in the literature.

Another difficulty of running benchmarks on modern processors is due to their intelligence. As a protection measure, processors throttle down their operating frequency, in some cases leading to wildly varying execution times. In other cases, multi-core processors run faster when only one core is active than when two or more are in use.

In the next sections, first the problems used for testing and the testing environments are presented, followed by the results obtained and by some conclusions.

4.1 Problems

The problems used to test the performance of the solver were chosen to constitute a mix of constraint satisfaction and constraint optimisation problems exhibiting different properties with respect to characteristics such as the number of solutions, the distribution of work throughout the search space, and the sensitivity to the search strategy.

4.1.1 n-Queens

The n-queens problem is one of the most frequently cited in constraint satisfaction related literature. It is a constraint satisfaction problem consisting in finding a placement for n queens on an $n \times n$ chess board, such that no queen attacks another one.

A solution to this problem is a permutation of the set $\{1, \dots, n\}$, with the value in its i^{th} position representing the row where queen in column i is placed, subject to the restriction that no two queens may share a board diagonal. The problem may be modelled with the aid of n variables, where the i^{th} variable corresponds to queen in column i, an *all-different* global constraint, and $n(n-1)$ constraints to handle the diagonals.

Its main characteristics are that it has many solutions, a number that grows very fast with the number of queens, and that the solutions are well spread out throughout the search space, which leads to sub-search spaces of similar sizes requiring a comparable amount of work to explore.

4.1.2 Langford's Number Problem

Langford's number problem [12, problem 024] consists in arranging k sets of the integers from 1 to n in such a way that between two consecutive occurrences of the integer i lie exactly i other numbers. For example, if k is 2 and n is 4, the only arrangement satisfying the constraints (modulo symmetry) is:

$$2\ 3\ 4\ 2\ 1\ 3\ 1\ 4.$$

This instance of the problem is also known as $L(2,4)$. More generally, $L(k,n)$ denotes a problem instance, for some value of k and n.

When modelling this problem, solutions which are symmetrical to another one are avoided by constraining the first element of the sequence to be smaller than the last one.

Many instances of the problem have no solution. For example, if k is 2, the problem is inconsistent if n is not of the form $4m$ or $4m-1$, for some m [36]. Other instances have many solutions, but unevenly distributed, which is compounded by discarding half of them.

4.1.3 Golomb Ruler

The Golomb ruler problem [12, problem 006] also appears often in the literature. Contrary to the n-queens problems, it is a constraint optimisation problem whose goal is to find the minimal length ruler with n marks such that the distance between any two marks is different from the distance between any other two marks. The first mark corresponds to the 0 position, and the last mark determines the ruler length.

The problem may also be stated as finding a set of n natural numbers such that their pairwise differences are all distinct, and its greatest element is minimal.

Also in contrast to the n-queens problem, the number of rulers of minimal length is very small and they tend to lie on the 'left' side of the search tree, where the value of the second mark is small (to break symmetries, the second ruler mark, whose value coincides with the difference between the second and first marks, is constrained to be less than the difference between the last and the last but one marks). As the number of marks grows, it soon becomes a very hard problem [54].

4.1.4 Quadratic Assignment Problem

The Quadratic Assignment Problem (QAP) [6, 7] is an optimisation problem that can be very hard to solve, even for small dimension instances. The formulation used is the following:

> Given an $n \times n$ distance matrix d and an $n \times n$ flow matrix f, find a permutation p of $\{1, \ldots, n\}$ which minimises
>
> $$\sum_{r=1}^{n} \sum_{c=1}^{n} d_{p(r),p(c)} f_{r,c},$$
>
> where $m_{i,j}$ stands for the value at row i and column j of matrix m, and $p(i)$ is the value at position i of the permutation p.

QAP may be used, for instance, to determine the location of a set of facilities such that the cost of the required movement of goods among them is minimal. In this case, the distance matrix contains the distances between the possible locations for the facilities, and the flow matrix represents the costs of transporting the goods between any two facilities per distance unit. Other applications include campus planning, the travelling salesman problem, and circuit layout design [5, 6].

Instances where n is 16 may be very hard to solve, and for some known size 30 instances an optimal solution has not yet been found [5].

4.2 Testing Environments

The operating system of all the computing systems used for testing was Linux, and several versions of the GNU C compiler GCC were used.

Systems

A brief presentation of the computer systems used in the experiments follows. Not having direct access to all of them, namely the fhg-1, fhg-2, and ha8000 systems, has limited the amount of testing performed there.

ism A 12 node network from the Grupo de Astrofísica Computacional, lead by Miguel Avillez at Universidade de Évora. Each symmetric multiprocessing (SMP) node has an Intel Core 2 Quad Q6600 CPU, with 2–4 GB RAM, running at 2.4 GHz. Nodes are connected by a Gigabit Ethernet network.

Experiments with up to 8 nodes were performed, depending on the availability.

sunfire A Non-Uniform Memory Access (NUMA) machine with 8 Dual-Core AMD Opteron 8220 processors, at 2.8 GHz, with 4 GB RAM per processor. Access was given by João Lourenço, at Universidade Nova de Lisboa.

This was one of the most frustrating systems to use, as two consecutive executions of a sequential program could take 20 s or 60 s to complete, for example, on an otherwise unoccupied system.

cri-lima Another NUMA computer, belonging to the Centre de Recherche en Informatique, at Université Paris 1. It features 2 Intel Xeon W5580 CPUs (with 4 hyper-threaded cores), running at 3.2 GHz, with 12 GB per processor. Access granted by Daniel Diaz.

fhg-1 A 64 node cluster, each node having two dual-core Intel Xeon 5148LV processors. Both this and the following system were used in the context of a collaboration with Rui Machado, at the Fraunhofer ITWM.

fhg-2 32 nodes with one Intel Xeon X5670 CPU per node (6 hyper-threaded cores), running at 2.93 GHz.

ha8000 The HA8000 Hitachi at University of Tokyo. Experiments are still ongoing on this system. So far, is has only been possible to run PaCCS on one node with four quad-core AMD Opteron 8350 processors, running at 2.3 GHz, with 8 GB RAM per processor. Used with the help of Florian Richoux, at the JFLI.

4.3 Multithreaded Performance

This section presents and analyses the results obtained with PaCCS on shared-memory and NUMA multiprocessor systems with the problems used for testing the solver.

In the tables displaying the speedups observed that follow, the first number on each row, in italics, represents the time taken to solve the problem with a single worker (or thread), in seconds. It represents the solver sequential performance. The remaining numbers on a row are the speedups obtained as more workers were employed.

4.3.1 n-Queens

The results discussed in this section correspond to counting the number of valid placements of the queens on the 16×16 and 17×17 boards. The 16-queens case was initially chosen as the time to solve it sequentially ranges between 5 and 10 minutes, while the size 17 problem takes around 1 hour, which makes it suitable for sporadic use in testing only.

16-Queens

The speedups obtained, shown in Table 4.1 and in Figure 4.1, are generally good up to 4 workers, except in the case of the cri-lima machine. However, experiments where the

	Workers					
System	1	2	3	4	8	16
ism	*531.5*	2.00	3.00	4.00		
fhg-1	*562.0*			4.00		
sunfire	*685.1*	2.00		3.99	7.76	15.32
cri-lima	*316.4*	1.92		3.82	7.46	8.71
	Time (s)			Speedups		

Table 4.1: Multithreaded performance for the 16-queens problem

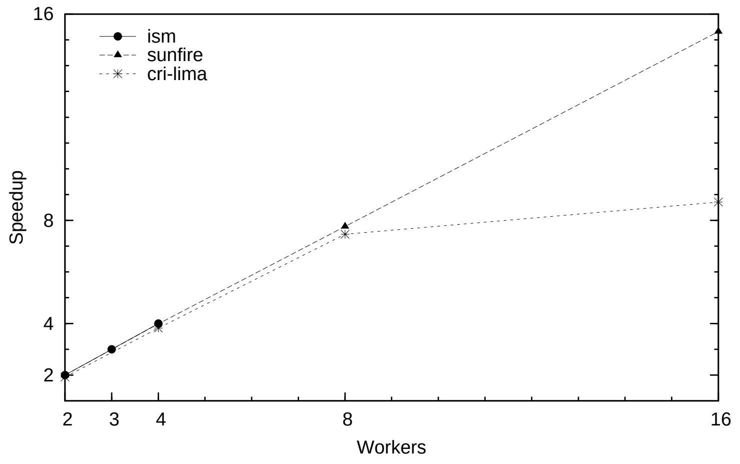

Figure 4.1: Evolution of the speedups for the 16-queens problem

search space is halved, by restricting the placement of the first queen to either the first 8 or the last 8 rows, exhibited times which are similar to those obtained when two workers explore the full search space, as shown in Table 4.2.

Due to the symmetrical nature of the problem, the amount of work performed on each half of the search space is exactly the same, and corresponds to the amount of work each of the workers would ideally perform when two workers explore the full search space in parallel. Consequently, is it not expectable that the latter may run faster than the former single workers. This suggests that the lower than ideal speedups obtained on cri-lima may not be a fault of the parallelisation, but that they reflect some advantage that a single process exploring the full search space gains on that system, such as a cacheing effect.

Possible rows for the 1st queen	1–16	1–8	9–16
Search space %	100	50	50
Workers	2	1	1
ism	265.5	264.4	264.7
sunfire	342.5	342.1	343.0
cri-lima	164.8	162.3	163.1
	Time (s)		

Table 4.2: Time taken by 2 workers to search the full search space *vs* time taken by 1 worker to search half the search space for the 16-queens problem

One instance where the performance observed was far behind the theoretically possible is when using 16 workers on cri-lima. This system has 8 cores, employing simultaneous multithreading (SMT) to achieve 16 threads of execution, and this is a setting PaCCS is not currently able to fully exploit.

PaCCS's implementation was designed to maximise locality of memory accesses, in order to derive the most benefit from data cacheing by the processor. Since all processing units in a multithreaded processor share its lowest level cache, similar access patterns by the threads being executed may result in increased competition for its use and prevent profiting more completely from the increased parallelism.

On the other hand, the processors in this system feature Intel's "Turbo Boost Technology" [25], whereby when only a few cores are in use, their operating frequency may be raised. When just one processor core is active, it may run between 4 and 8% faster than when all four are engaged in executing a program [24][1].This means that as the number of workers increases they may operate slower, which will contribute to lower speedups.

One aspect which could detrimentally influence PaCCS's performance is the time the workers spend trying to steal a store from a teammate. Table 4.3 shows the number of stores stolen by all workers and the total time spent attempting to steal one. This time covers both ultimately successful and unsuccessful attempts, which will occur when the search space has been completely explored. As a consequence, it includes the time the workers took to detect termination of the search.

Unsurprisingly, the number of stores stolen during the solving process increases with the number of workers, which is due both to the disparity between the initial workload of the workers becoming more pronounced and to the differences in how they progress. The time spent in store stealing attempts increases as well, although it stays under 0.5 ms per worker.

[1]Experiments with a program where each thread executed a tight loop, without accessing the memory, revealed a 4% increase in CPU time per thread when more than one thread ran, as well as a 4% increase in total execution time.

	Workers									
System	2		3		4		8		16	
ism	22	0.2	72	0.4	137	0.5				
sunfire	11	0.2			89	0.8	440	2.5	978	6.4
cri-lima	23	0.2			96	0.8	466	3.4	1130	6.0

Number of stolen stores / Time spent stealing (ms)

Table 4.3: Number of stores stolen and the time spent stealing for the 16-queens problem

Where available, the sequential and multithreaded performances of Gecode were also measured, and the results are displayed in Table 4.4. While Gecode's sequential

	Workers				
System	1	2	4	8	16
sunfire	*413.54*	0.87	1.05	1.00	0.95
cri-lima	*296.34*	1.21	2.39	1.61	1.54

Time (s) Speedups

Table 4.4: Gecode performance for the 16-queens problem

performance is better than that of PaCCS for this problem, its multithreaded behaviour is much worse.

17-Queens

The queens problem on a 17×17 is an order of magnitude harder than with only 16 queens. The results obtained with it, which can be seen in Table 4.5 and Figure 4.2, present a similar profile to those obtained in the previous case.

	Workers					
System	1	2	3	4	8	16
ism	*3638.2*	2.00	3.00	3.96		
sunfire	*4735.7*	2.00		3.86	7.72	15.41
cri-lima	*2244.1*	1.99		3.78	7.49	8.95

Time (s) Speedups

Table 4.5: Multithreaded performance for the 17-queens problem

Table 4.6 shows store stealing information for this problem size.

While the numbers of stores successfully stolen increase, they remain of the same magnitude as before. The fact that the time spent trying to steal a store hardly changes means that most of it comes from the final attempt, when the search space is becoming

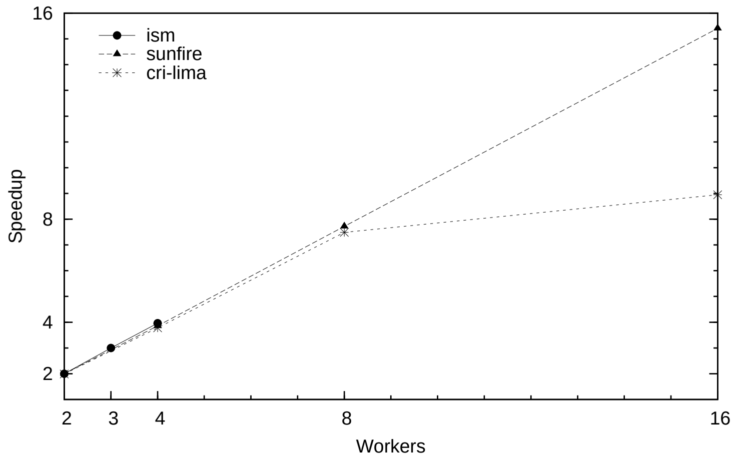

Figure 4.2: Evolution of the speedups for the 17-queens problem

	Workers									
System	2		3		4		8		16	
ism	29	0.2	74	0.4	145	0.7				
sunfire	27	0.3			150	0.8	630	3.2	1596	9.1
cri-lima	26	0.2			152	1.1	556	2.9	1424	5.6

Number of stolen stores / Time spent stealing (ms)

Table 4.6: Number of stores stolen and the time spent stealing for the 17-queens problem

exhausted, and just before termination is detected. In this situation, all workers but one will end up waiting for the last one to finish exploring its search space, before they all decide that there is no more work to do.

As is the case for PaCCS, Gecode's performance on the bigger instance of the problem, summed up in Table 4.7, also replicates its previously observed behaviour.

	Workers				
System	1	2	4	8	16
sunfire	*2746.8*	0.84	1.05	1.00	1.01
cri-lima	*1991.2*	1.41	2.10	2.65	1.62

Time (s) Speedups

Table 4.7: Gecode performance for the 17-queens problem

4.3.2 Langford's Number Problem

The solving of Langford's number problem exhibits a more irregular behaviour than is the case with the n-queens problem. If for the latter the number of search tree nodes explored remains basically constant, regardless of the number of workers involved, for Langford's problem it may grow by almost 20% when employing 16 workers.

The reason for the above is that the search process is more sensitive to the order by which the variables are selected, which deviates from the one determined by the heuristics used because of problem partitioning. This leads to a less effective pruning of the search tree, which results in the solver performing more work, as it has to explore a greater number of nodes. This is the effect mentioned in Section 3.6, on page 30.

Two problem instances were used during testing, $L(2, 15)$ and $L(2, 16)$. While the first has 39 809 640 solutions and is solved by one worker in around 1 hour, the second requires about 10 times that time to find its 326 721 800 solutions.

$L(2, 15)$

The results obtained for counting all solutions of Langford's problem with $k = 2$ and $n = 15$ are presented in Table 4.8 and plotted in Figure 4.3.

	Workers					
System	1	2	3	4	8	16
ism	*3520.0*	1.99	2.89	3.69		
sunfire	*4908.3*	1.99		3.70	6.80	12.88
cri-lima	*2574.1*	1.97		3.63	6.65	8.91
	Time (s)		Speedups			

Table 4.8: Multithreaded performance for $L(2, 15)$

While the growth of the speedups shows a good pace, as long as simultaneous multithreading is not employed, their values are significantly below those seen in the n-queens problem.

This, however, may be partly ascribed to the aforementioned increase in the number of nodes visited, which follows from the modification of the search strategy ensuing from the parallel exploration of the search tree. The importance of this effect may be gauged in Table 4.9, which reflects the change in the number of nodes visited as more workers are employed in the search process. Note that, for a solver configured as a single multithreaded team, the number of nodes explored only varies with the number of active workers, remaining otherwise constant across executions and computing platforms. Hence, for this problem, the partitioning strategy applied is a determining factor in how the search will unfold.

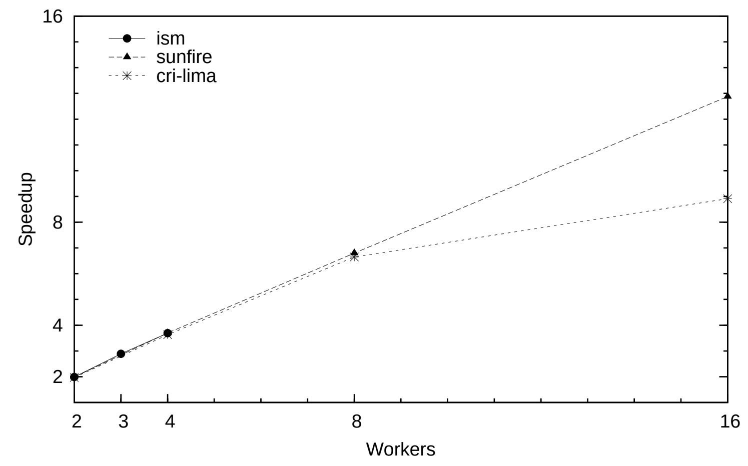

Figure 4.3: Evolution of the speedups for $L(2, 15)$

	Workers					
	1	2	3	4	8	16
Nodes	1 003 026 551	+0.0005%	+3.51%	+6.78%	+14.00%	+18.86%
	Nodes	Variation				

Table 4.9: Number of nodes visited to solve $L(2, 15)$ and increment with respect to the 1 worker case

Comparing the number of nodes processed per time unit by the several configurations of the solver leads to Table 4.10. There, it is shown that the node processing throughput, plotted in Figure 4.4, increases almost linearly with the number of workers, resembling the speedups observed for the n-queens problem. Applying a partitioning strategy that makes an initial work distribution similar to the one that results from the search heuristics used and which does not lead to such a large increase in the number of nodes explored, would allow that node processing power to be reflected on the speedups obtained.

	Workers				
System	2	3	4	8	16
ism	1.99	2.99	3.94		
sunfire	1.99		3.97	7.86	15.63
cri-lima	1.97		3.88	7.58	10.59

Table 4.10: Comparison of the number of nodes processed per time unit with the number of nodes processed per time unit by the sequential solver for $L(2, 15)$

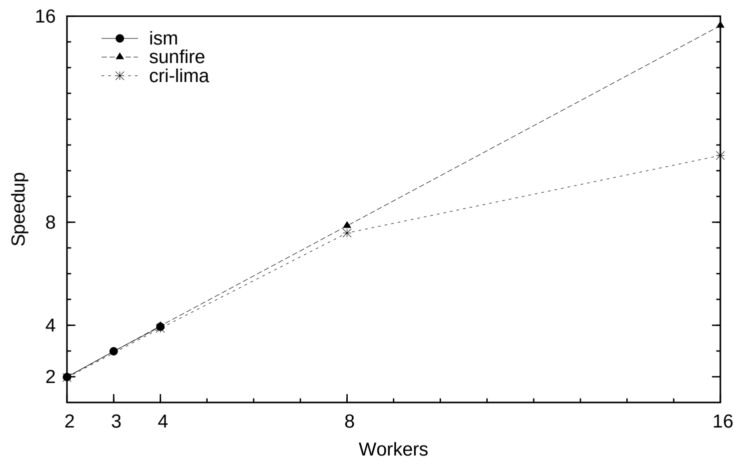

Figure 4.4: Evolution of the number of nodes processed per time unit for $L(2, 15)$

The statistics respecting the stealing of work, in Table 4.11, are comparable to those observed in the 17-queens case, which is a problem of similar magnitude.

	Workers									
System	2		3		4		8		16	
ism	34	0.2	78	0.3	126	0.6				
sunfire	30	0.4			150	0.9	627	3.4	1955	10.2
cri-lima	33	0.3			129	1.1	416	2.5	2013	8.2

Number of stolen stores / Time spent stealing (ms)

Table 4.11: Number of stores stolen and the time spent stealing for $L(2, 15)$

PaCCS outperforms Gecode in this instance, as Table 4.12 witnesses, both in the sequential search and in the speedups obtained, except with simultaneous multithreading. Gecode, however, does not suffer from the increase in the number of nodes visited, as it reports the same number in all configurations.

	Workers				
System	1	2	4	8	16
sunfire	*8253.65*	1.82	3.42	5.89	7.52
cri-lima	*5600.82*	1.82	3.58	6.32	8.93

Time (s) Speedups

Table 4.12: Gecode performance for $L(2, 15)$

$L(2,16)$

Langford's number problem for $k = 2$ and $n = 16$ is one order of magnitude larger than the previous instance. Solving $L(2,16)$ explores around 10 times the nodes, taking 10 times the time it took before. Nevertheless, the results obtained follow closely those above, whether they respect the multithreaded speedups, shown in Table 4.13 and in Figure 4.5, the evolution of the number of nodes explored and of their processing rate, in Tables 4.14 and 4.15, or the store stealing statistics of Table 4.16.

	Workers					
System	1	2	3	4	8	16
ism	*36116.4*	2.00	2.91	3.79		
sunfire	*46626.8*	2.02		3.66	7.00	13.00
cri-lima	*24406.9*	1.98		3.63	6.86	8.87
	Time (s)	Speedups				

Table 4.13: Multithreaded performance for $L(2,16)$

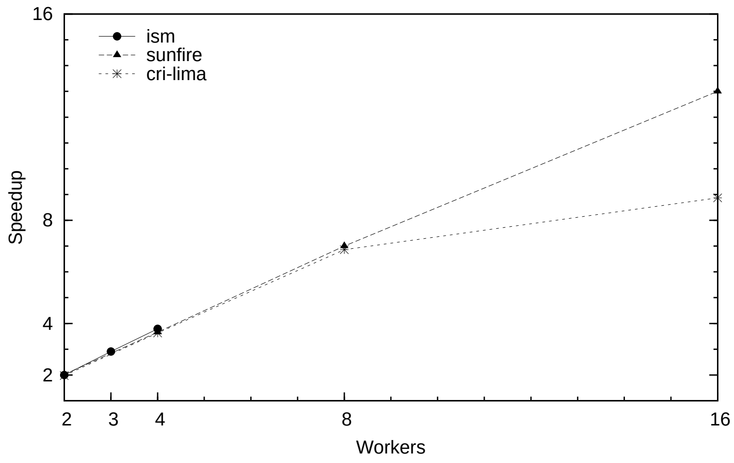

Figure 4.5: Evolution of the speedups for $L(2,16)$

	Workers					
	1	2	3	4	8	16
Nodes	9 201 034 367	+0.0002%	+3.17%	+6.16%	+12.85%	+17.55%
	Nodes	Variation				

Table 4.14: Number of nodes visited to solve $L(2,16)$ and increment with respect to the 1 worker case

	Workers				
System	2	3	4	8	16
ism	2.00	3.00	4.02		
sunfire	2.02		3.88	7.90	15.28
cri-lima	1.81		3.83	7.29	10.65

Table 4.15: Comparison of the number of nodes processed per time unit with the number of nodes processed per time unit by the sequential solver for $L(2, 16)$

	Workers									
System	2		3		4		8		16	
ism	37	0.2	103	0.4	146	0.5				
sunfire	39	0.4			150	0.8	564	3.0	2438	13.3
cri-lima	29	0.2			113	0.7	482	2.1	2650	10.6

Number of stolen stores / Time spent stealing (ms)

Table 4.16: Number of stores stolen and the time spent stealing for $L(2, 16)$

As is the case with PaCCS, Gecode behaviour maintains the profile displayed above, as may be checked in Table 4.17.

	Workers				
System	1	2	4	8	16
sunfire	*74396.8*	1.87	3.50	6.20	8.19
cri-lima	*50260.7*	1.81	3.61	6.46	9.06
	Time (s)	Speedups			

Table 4.17: Gecode performance for $L(2, 16)$

4.3.3 Golomb Ruler

The problems studied so far have all been pure constraint satisfaction problems. The Golomb ruler problem is the first constraint optimisation problem used to test the performance of PaCCS. The speedups observed for the 13-mark ruler, described in Table 4.18 and in Figure 4.6, comprise some superlinear speedups.

The solving of a constraint optimisation problem can be divided into two parts. The first one consists in finding a best solution. Then, is must proven that there is no solution better than the one found, and this is the second phase of the solving process. This division is only know *a posteriori*, after having failed to improve on the last solution found.

The first phase of the search may be regarded as a race through the search tree

	Workers					
System	1	2	3	4	8	16
ism	*6940.4*	2.24	3.30	4.33		
sunfire	*8345.6*	2.25		4.36	8.38	14.78
cri-lima	*4224.4*	2.20		4.16	7.82	8.31
	Time (s)	Speedups				

Table 4.18: Multithreaded performance for the 13-mark Golomb ruler

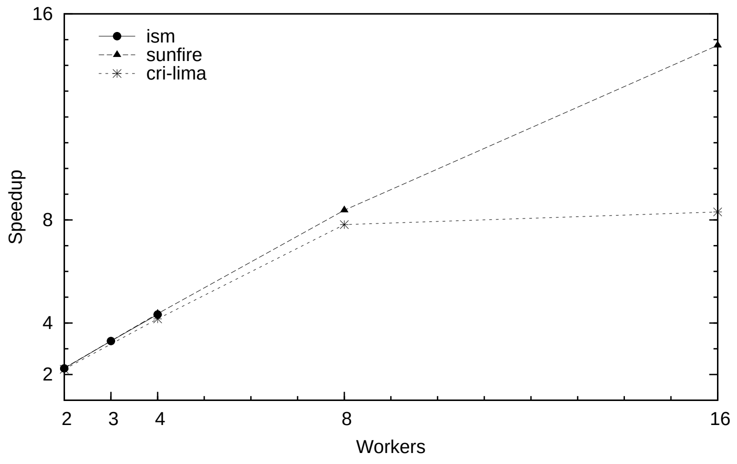

Figure 4.6: Evolution of the speedups for the 13-mark Golomb ruler

between all the workers. As one of them arrives at a solution better than the previous one, it broadcasts its value to all the others, that then reset their goal according to the new bound received. This phase is liable to the superlinear speedups discussed in Section 2.4 for two reasons. One is that the problem partitioning may create a sub-search space where a solution lies closer to its root for a worker adhering to the normal problem search strategy. The other is that as a worker finds a solution closer to the optimum, all the other workers benefit by being able to further prune their search spaces.

Table 4.19 presents the times, in seconds, to find a shortest 13-mark ruler, *i.e.* not including the optimality proof. The speedup observed when going from one to two workers, is a superlinear speedup of around 11, but it remains mostly constant as more workers are employed.

A superlinear reduction in the time taken in the first phase of the solving process means fewer nodes will be explored in total, as Table 4.20 confirms. Once a best solution has been found, the remaining unexplored search space must be processed by all the workers. A linear speedup is expected in this phase of the search and, combined with a superlinear speedup obtained in reaching it, it gives rise to the results reported in

	Workers					
System	1	2	3	4	8	16
ism	2196.6	194.9	195.0	195.3		
sunfire	2579.4	233.5		235.9	229.0	233.7
cri-lima	1316.3	121.0		123.1	123.0	209.0

Table 4.19: 13-mark Golomb ruler: time to find a best solution (s)

	Workers					
System	1	2	3	4	8	16
ism	609 781 742	-11.82%	-10.32%	-8.90%		
sunfire	609 781 742	-11.78%		-8.79%	-3.24%	7.98%
cri-lima	609 781 742	-11.80%		-8.92%	-3.20%	7.74%
	Nodes			Variation		

Table 4.20: Number of nodes visited to solve the 13-mark Golomb ruler problem and variation with respect to the 1 worker case

Table 4.18. The sublinear speedups obtained for a 16 worker team are partly explained by the speedup in finding the best solution being below 16 and the consequent increase in the number of nodes explored.

Table 4.21 documents the way the number of nodes processed per time unit evolves as the number of workers grows. There, it may be seen that the superlinear speedups are achieved in spite of only a linear evolution of the processing throughput.

	Workers				
System	2	3	4	8	16
ism	1.97	2.96	3.95		
sunfire	1.98		3.97	8.11	15.96
cri-lima	1.94		3.79	7.57	8.95

Table 4.21: Comparison of the number of nodes processed per time unit with the number of nodes processed per time unit by the sequential solver for the 13-mark Golomb ruler

Data related to work stealing, detailed in Table 4.22, follow those observed in the other similarly sized problems.

Gecode clearly outperforms PaCCS in this problem, as Table 4.23 attests. Its much more effective propagation prunes the search tree so that it visits between 10.5% and 12.6% as many nodes as PaCCS, which is reflected by the execution times. For this problem, Gecode's parallelisation speedups are comparable to those of PaCCS.

	Workers									
System	2		3		4		8		16	
ism	26	0.4	57	0.4	48	0.8				
sunfire	23	0.9			52	1.0	364	3.5	1631	15.9
cri-lima	23	0.3			47	1.3	353	3.3	1541	9.9

Number of stolen stores / Time spent stealing (ms)

Table 4.22: Number of stores stolen and the time spent stealing for the 13-mark Golomb ruler

	Workers				
System	1	2	4	8	16
sunfire	*1931.0*	2.30	3.72	8.69	13.16
cri-lima	*1396.8*	2.00	4.36	7.58	10.17
	Time (s)	Speedups			

Table 4.23: Gecode performance for the 13-mark Golomb ruler

4.3.4 QAP

The esc16e instance [5] of the Quadratic Assignment Problem, another constraint optimisation problem, was used in these experiments. The esc16g problem, which is significantly easier, was also used for comparing with Gecode results, as the Gecode version, besides being slower, does not explore the symmetries of the problems. Note that, in this instance, PaCCS was run with the exact same model as Gecode.

QAP esc16e

Table 4.24 presents the time (in seconds) needed by PaCCS to solve the esc16e QAP with one worker on several of the systems, as well as the speedups obtained when using more workers. These values are plotted in Figure 4.7

	Workers					
System	1	2	3	4	8	16
ism	*7003.47*	1.99	3.01	4.00		
fhg-1	*11652.38*			4.01		
sunfire	*9230.31*	2.00		4.04	8.15	16.23
cri-lima	*3882.35*	1.94		3.81	7.64	8.45
	Time (s)	Speedups				

Table 4.24: Times and speedups for QAP esc16e

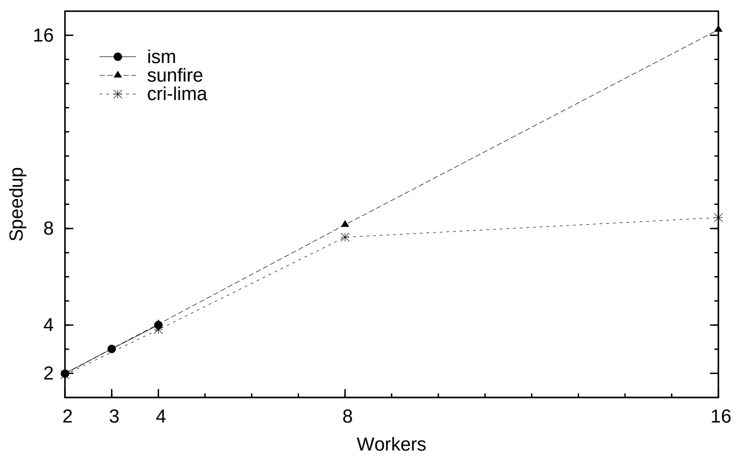

Figure 4.7: Evolution of the speedups for QAP esc16e

The speedups obtained are mostly linear, or superlinear, which is as good as could be expected, given the time to find a solution with the best value, discussed below. As can be verified, PaCCS is not yet able to fully explore the SMT cores of cri-lima. The remaining suboptimal speedups displayed on that system, even when simultaneous multithreading is not used, are consistent with the operation of the Turbo Boost mechanism.

The time needed to find a best solution to the problem also exhibits some superlinear speedups, shown in Table 4.25, which help explain those of the full solving process, as it accounts for 1% of the time the single worker solver needs to complete the search. As a consequence, the number of visited nodes decreases, albeit only slightly, when there are more than two workers, never by more than 0.25%. The different partitioning strategy

	Workers					
System	1	2	3	4	8	16
ism	70.4	38.6	1.1	8.1		
sunfire	92.7	48.3		10.6	4.4	0.7
cri-lima	41.4	21.4		0.3	0.4	0.5

Table 4.25: Time to find an optimal solution for QAP esc16e (s)

applied on cri-lima justifies the times differences for 4 and 8 workers.

The time taken to steal a store increases a little on this problem, as Table 4.26 testifies. The model for this QAP instance uses 261 variables, whose domains occupy around 9K bytes. This is the size of the store for this problem, larger than in the previous problems, which must be copied every time one is stolen. The sunfire is the system most sensitive to the store size, as it is a NUMA computer with only two cores per node, and stores must sometimes be copied between node memories. Nevertheless, it stays under

1.5 ms per worker in a 569 s execution.

	Workers									
System	2		3		4		8		16	
ism	28	0.5	92	0.9	112	1.3				
sunfire	31	0.6			140	2.7	371	8.9	991	22.7
cri-lima	46	0.6			96	1.2	349	3.7	1045	16.0

Number of stolen stores / Time spent stealing (ms)

Table 4.26: Number of stores stolen and the time spent stealing for QAP esc16e

QAP esc16g

The results obtained for solving the esc16g QAP with both PaCCS and Gecode are contained in Table 4.27. In this case, both PaCCS and Gecode achieve near ideal performance, with the latter managing to benefit more from simultaneous multithreading, although still remaining more than 5 times slower than PaCCS.

	Workers				
	1	2	4	8	16
sunfire					
PaCCS	*591.4*	1.98	3.98	7.89	15.65
Gecode	*3278.1*	1.99	3.95	7.88	15.54
cri-lima					
PaCCS	*278.9*	2.07	4.04	8.00	8.96
Gecode	*1804.5*	2.07	3.94	7.78	10.66
	Time (s)	Speedups			

Table 4.27: Comparing PaCCS and Gecode on QAP esc16g

4.4 Distributed Performance

Similarly to the previous section, this section presents the results obtained with PaCCS on distributed systems with the same problems as above.

In the following tables, the speedups shown are with respect to the time taken by a single *team* to solve the problem. When running on the fhg-2 system, teams consist of 12 workers, as each node has 6 cores with simultaneous multithreading. On every other system, a team comprises 4 workers.

Like before, the first number on each row of the tables featuring speedups represents the time, in seconds, taken by one team to solve the problem. This number is written in italics. The remaining numbers on a row are the speedups obtained as more teams were employed.

Work stealing is much more expensive in a distributed setting. The time spent with it changes from the so far few milliseconds to a few seconds, and it becomes a more important factor in determining the solver ultimate performance.

Another factor is the overhead of starting and stopping the solver. The setup time of PaCCS was measured on the ism cluster, as the time needed to launch all teams, create the workers, and terminate. The values obtained are presented in Table 4.28, where the time for the 1 team solver corresponds to deploying it on a remote host.

Teams	1	2	3	4	5	6	7	8
Setup time (s)	0.160	0.165	0.242	0.248	0.253	0.256	0.259	0.258

Table 4.28: PaCCS distributed setup time on ism

It only was possible to run the solver on just a few problems on several systems. Many tests ran only on the ism cluster, which is a small scale system, due to access constraints.

4.4.1 n-Queens

As was the case with the multithreaded parallel solver, both the 16- and 17-queens problems were used to test PaCCS in distributed contexts. Solving this problem consists in counting all possible solutions.

PaCCS demonstrates good scalability for this problem, particularly on the larger instance.

16-Queens

With 16 queens, the problem is more manageable, but soon the communication overheads prevent obtaining more gains from the increased parallelism. Even so, speedups of more than 41 were observed with 64 teams, when PaCCS only needs 3.4 s to find all solutions to the problem. This can be checked in Table 4.29 and Figure 4.8.

	Teams										
System	1	2	3	4	5	6	7	8	16	32	64
ism	*134.5*	2.00	2.99	3.96	4.88	5.84	6.78	7.71			
fhg-1	*140.5*	2.00		3.93				7.56	14.76	26.22	41.32
	Time (s)	Speedups									

Table 4.29: Distributed speedups for the 16-queens problem

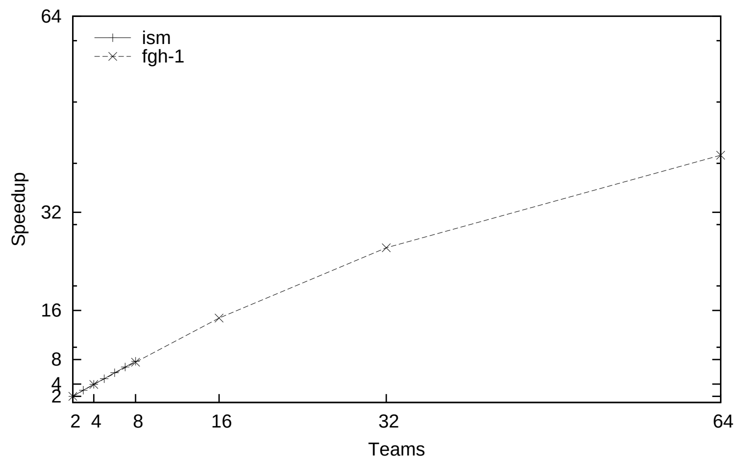

Figure 4.8: Evolution of the distributed speedups for the 16-queens problem

The measured speedups evolve almost linearly up to 16 teams, when the time to solve the problem reaches 9.5 s.

Taking the setup time into account, by subtracting it from the total execution time, gives what may be considered the *net* solving time, *i.e.*, the time truly dedicated by the solver to solving the problem. Since this time includes distributing work initially to all teams and collecting the answers, it would be the time a network available constraint solving service would take to respond to a client.

Using the *net* solving time to compute the speedups on ism, leads to the results in Table 4.30.

	Teams						
System	2	3	4	5	6	7	8
ism	2.00	3.00	3.98	4.91	5.90	6.87	7.82

Table 4.30: Net distributed speedups for the 16-queens problem

Table 4.31 shows the work stealing statistics per team for the configurations of the solver considered. The data consists of the number of stores stolen, both within a team and from other teams, the time spent trying to steal a store internally, in milliseconds, and the idle time, also in milliseconds. This last time is the time each team waited, on average, for some other team to send work, and includes the detection of the end of the search process.

In the worst case, corresponding to the 8 team solver configuration, both times combined represent 0.88% of the total execution time of 17.5 s, and the average time needed to steal one store was 0.17 ms.

More distributed work stealing leads to less effective pruning of the search tree. The

	Teams							
	1	2	3	4	5	6	7	8
Stolen stores	137	178	351	734	460	811	692	903
Time stealing (ms)	0.5	1.6	3.7	9.7	6.6	10.3	10.2	14.0
Idle time (ms)		16.1	29.9	72.9	66.2	88.3	99.3	140.0

Table 4.31: Number of stores stolen and the time spent stealing per team for the 16-queens problem

number of nodes visited during the search rises more significantly for the four solver configurations with the more teams, but only by around 0.05%, which has almost no impact on the solver performance.

17-Queens

Being a larger instance, the 17-queens problem is less affected by the overheads associated with running PaCCS on a distributed system. In fact, Table 4.32 and Figure 4.9 show linear scalability for up to 8 teams. Due to the regularity of the problem, the number of nodes visited suffers little variation for the various solver configurations, allowing the added computing power to translate fully into the increased performance.

	Teams							
System	1	2	3	4	5	6	7	8
ism	*917.8*	2.05	3.07	4.06	5.10	6.07	7.09	7.99
	Time (s)	Speedups						

Table 4.32: Distributed speedups for the 17-queens problem

Work stealing statistics, in Table 4.33, show increased activity. However, as the 17-queens instance takes about 7 times the time required to solve the 16-queens case, but the time involved in load balancing only doubles, the solver spends a greater percentage of time carrying out the search, leading to the better measured performance.

	Teams							
	1	2	3	4	5	6	7	8
Stolen stores	145	304	829	604	1151	1831	1755	1266
Time stealing (ms)	0.7	2.4	7.1	7.1	12.8	22.4	23.6	19.7
Idle time (ms)		19.5	51.0	59.2	100.8	179.2	205.6	180.1

Table 4.33: Number of stores stolen and the time spent stealing per team for the 17-queens problem

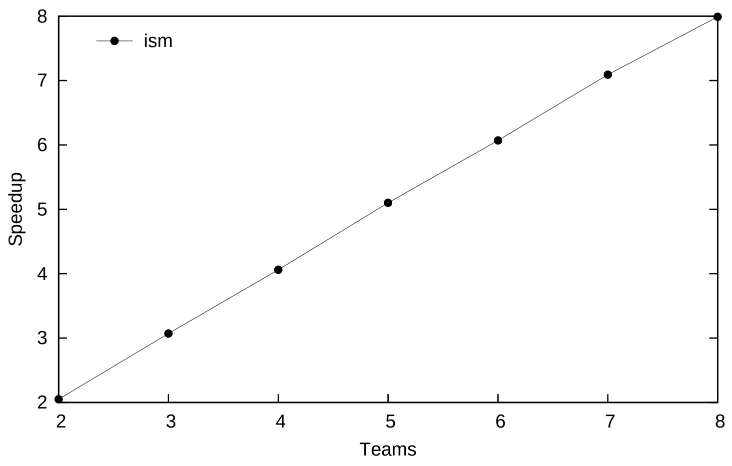

Figure 4.9: Evolution of the distributed speedups for the 17-queens problem

4.4.2 Langford's Number Problem

The same two instances of the problem as before were solved in a distributed context. PaCCS reveals the potential to attain linear speedups on both, but the increase in the number of nodes explored prevents getting closer to that goal.

$L(2, 15)$

As in the multithreaded case, the increase in parallelisation and work sharing causes the number of nodes visited for $L(2, 15)$ to grow as well. This increase may represent a significant part of the search space, as seen in Table 4.34.

	Teams							
	1	2	3	4	5	6	7	8
Nodes	1 070 990 062	9.60	10.47	11.48	12.18	13.23	13.37	15.31
	Nodes	Increment (%)						

Table 4.34: Number of nodes visited to solve $L(2, 15)$ and increment with respect to the 1 team case

Table 4.35 and Figure 4.10 show the distributed speedups obtained for $L(2, 15)$ on ism, and the evolution in the number of nodes processed per time unit. Once again, a different problem partition strategy might enable obtaining linear speedups.

Despite being of similar dimension to the 17-queens problem, the imbalance of the distribution of work within the $L(2, 15)$ search space leads to increased work stealing activity, reported in Table 4.36. Yet, the total time spent with it is still under 0.6 s and the influence it has on the measured speedups is very small.

	Teams							
	1	2	3	4	5	6	7	8
Time / Speedups	*954.4*	1.83	2.72	3.60	4.43	5.25	6.11	6.88
Node throughput		2.00	2.99	4.01	4.96	5.90	6.95	7.93
	Time (s)	Speedups						

Table 4.35: Distributed speedups for $L(2,15)$ on ism and evolution of the number of nodes processed per time unit

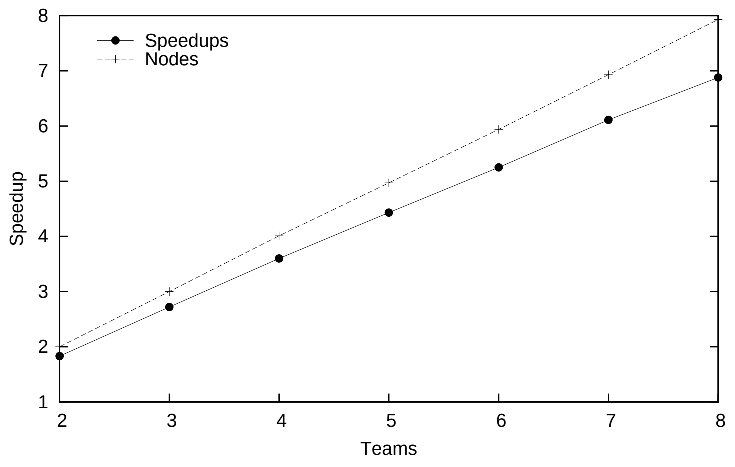

Figure 4.10: Evolution of the distributed speedups and of the number of nodes processed per time unit for $L(2,15)$

	Teams							
	1	2	3	4	5	6	7	8
Stolen stores	126	879	1823	2092	3036	2487	3703	3524
Time stealing (ms)	0.6	6.7	18.1	22.3	39.2	31.3	54.5	219.7
Idle time (ms)		40.0	94.3	129.1	235.3	199.4	361.1	349.9

Table 4.36: Number of stores stolen and the time spent stealing per team for $L(2,15)$

$L(2,16)$

The behaviour profile manifested by the $L(2,16)$ instance of Langford's Number Problem, portrayed in Figure 4.11, Tables 4.37, 4.38, and 4.39, is the same as that of $L(2,15)$, albeit on a larger scale. Although spending 3 s, on average, waiting for answers to work requests may seem a long time to be idle, it just accounts for near 0.2% of the total running time of the solver.

	Teams							
	1	2	3	4	5	6	7	8
Nodes	9 767 919 974	9.27	9.53	10.55	13.15	12.86	14.70	13.83
	Nodes	Increment (%)						

Table 4.37: Number of nodes visited to solve $L(2, 16)$ and increment with respect to the 1 team case

	Teams							
	1	2	3	4	5	6	7	8
Time / Speedups	*9533.3*	1.88	2.79	3.70	4.57	5.39	6.24	7.07
Node throughput		2.05	3.06	4.09	5.17	6.08	7.16	8.05
	Time (s)	Speedups						

Table 4.38: Distributed speedups for $L(2, 16)$ on ism and evolution of the number of nodes processed per time unit

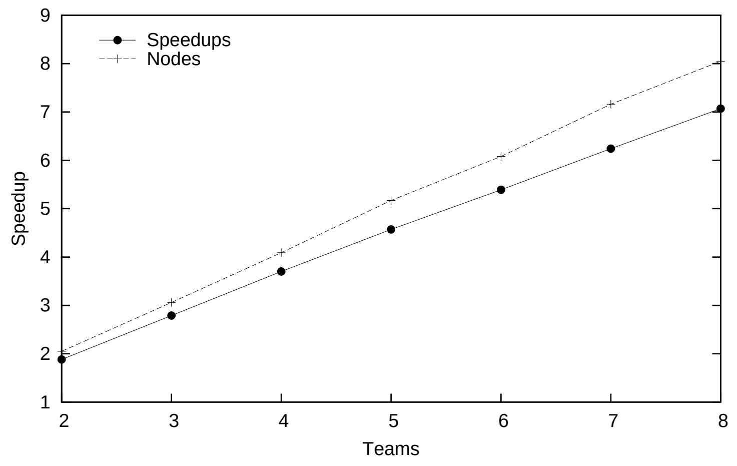

Figure 4.11: Evolution of the distributed speedups and of the number of nodes processed per time unit for $L(2, 16)$

	Teams							
	1	2	3	4	5	6	7	8
Stolen stores	146	1444	1760	3063	6566	13193	30631	20400
Time stealing (ms)	0.5	10.6	15.9	29.9	86.9	181.5	218.5	319.3
Idle time (ms)		50.5	76.1	167.4	717.3	975.1	2936.5	2014.6

Table 4.39: Number of stores stolen and the time spent stealing per team for $L(2, 16)$

4.4.3 Golomb Ruler

Solving the 13-mark Golomb ruler problem resulted in the speedups presented in Table 4.40. Considering the number of nodes explored, in Table 4.41, gives the number of nodes processed per time unit in Table 4.42, plotted in Figure 4.12.

	Teams							
System	1	2	3	4	5	6	7	8
ism	*1601.1*	1.91	2.89	3.79	4.53	5.38	5.97	6.64
	Time (s)	Speedups						

Table 4.40: Distributed speedups for the 13-mark Golomb ruler

	Teams							
	1	2	3	4	5	6	7	8
Nodes	555 523 660	5.12	4.31	7.48	9.02	12.04	16.25	20.53
	Nodes	Increment (%)						

Table 4.41: Number of nodes visited to solve 13-mark Golomb ruler and increment with respect to the 1 team case

	Teams						
	2	3	4	5	6	7	8
Node throughput	2.05	3.06	4.09	5.17	6.08	7.16	8.05

Table 4.42: Evolution of the number of nodes processed per time unit for the 13-mark Golomb ruler

The time to arrive at a best solution progresses as shown in Table 4.43. It hardly changes from a solver with 3 teams on, and for 8 teams represents almost half of the solving time. As more workers participate in the search, the time spent in this phase of the solving process will cause the growth experienced in the number of visited nodes.

	Teams							
System	1	2	3	4	5	6	7	8
ism	195.28	186.4	113.71	104.83	100.24	100.05	102.79	106.20

Table 4.43: Distributed 13-mark Golomb ruler: time to find a best solution (s)

Higher speedups in constraint optimisation problems may only be achieved if it is possible to continue to improve on the time to arrive at a best solution, but this is highly dependent on the properties of the problem and cannot be controlled. The selection

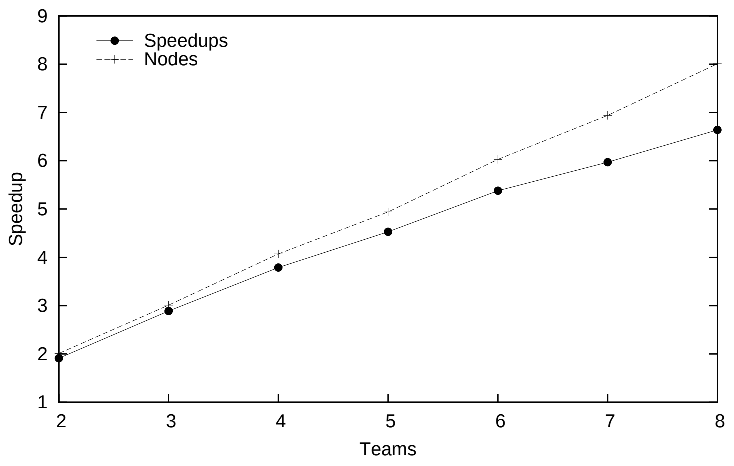

Figure 4.12: Evolution of the distributed speedups and of the number of nodes processed per time unit for the 13-mark Golomb ruler

heuristics which are usually deemed the best for a given problem, help steer the search towards a solution, but their success is not guaranteed, and parallelising the solving process adds another dimension which may further hinder their effectiveness.

Table 4.44 shows that work sharing for the 13-mark Golomb ruler follows a similar pattern to that of the similarly sized $L(2, 15)$.

	Teams							
	1	2	3	4	5	6	7	8
Stolen stores	48	498	686	1013	1392	1808	2923	3822
Time stealing (ms)	0.8	7.4	12.0	21.0	33.3	49.8	79.8	106.9
Idle time (ms)		32.8	49.4	84.1	129.9	251.9	447.1	696.7

Table 4.44: Number of stores stolen and the time spent stealing per team for the 13-mark Golomb ruler

4.4.4 QAP

The same problem as before, esc16e, was used while experimenting with PaCCS on distributed systems. The results appear on Table 4.45 where, once again, numbers in italic represent times in seconds and the remaining numbers are the speedups with respect to those times.

The speedups obtained are good on all systems. (Note that the base time on the fhg-2 system is for a two teams solver. Also note that on the ha8000 system, all the teams ran on the same node, and the results are similar to those obtained on multithreaded machines.)

The tests on ism, and on fhg-1 and fhg-2, were executed using the hierarchical model of communication from Section 3.7.2. The line marked with an asterisk corresponds to a case where the peering model was used, and its performance proves to be a little worse. Figure 4.13 plots the upper part of Table 4.45.

	Teams										
System	1	2	3	4	5	6	7	8	16	32	64
ha8000	*2330.2*	1.99		3.92							
ism	*1784.0*	2.00	3.00	3.99	4.99	5.98	6.96	7.97			
fhg-1	*2908.8*	2.00		4.01				8.00	15.97	31.27	60.80
fhg-1*	*2908.8*										55.53
fhg-2	—	*264.4*		2.04				4.08	7.93	14.30	
	Time (s)				Speedups						

Table 4.45: Distributed speedups for QAP esc16e

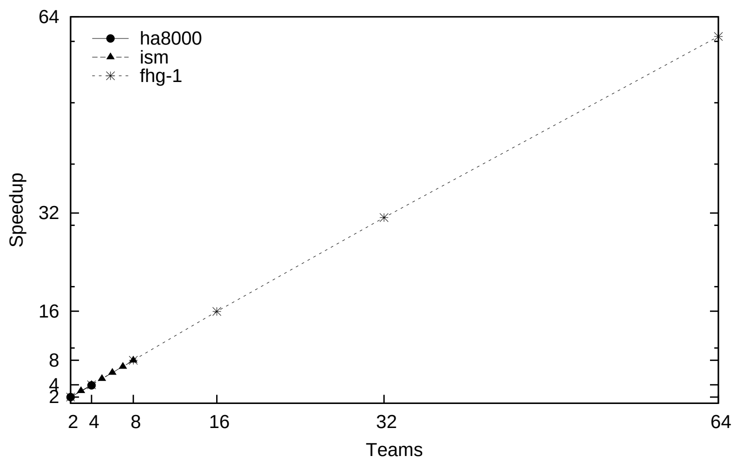

Figure 4.13: Evolution of the distributed speedups for QAP esc16e

The speedups obtained are mostly linear for all systems. The main development system for PaCCS was the ism cluster, which meant it was often tested there. Even without this help, the solver adapted itself to the other environments, which served mainly to gain insight on PaCCS behaviour and performance.

Further analysis of these results will be carried out in relation with the ism cluster only, starting with the time to arrive at a best solution, in Table 4.46. As as consequence of these times, the number of nodes explored diminishes very slightly for the 2 and 3 teams solver, then starts rising again, keeping below the number visited by the 1 team solver.

	Teams							
System	1	2	3	4	5	6	7	8
ism	8.14	3.27	0.48	0.48	0.49	0.43	0.46	0.46

Table 4.46: Time to find an optimal solution for QAP esc16e (s)

The work stealing behaviour exhibited in this instance is similar to that obtained with the 16-queens problem, a simpler problem. According to Table 4.47, workers spend, on average, less than 0.05% of their time either stealing or waiting for work.

	Teams							
	1	2	3	4	5	6	7	8
Stolen stores	112	192	756	417	703	1565	1835	537
Time stealing (ms)	1.3	4.4	19.8	15.1	23.5	56.3	179.6	25.5
Idle time (ms)		22.7	75.5	72.7	105.7	256.4	362.3	152.8

Table 4.47: Number of stores stolen and the time spent stealing per team for QAP esc16e

4.5 Conclusions

This chapter presented the results obtained using PaCCS to solve constraint satisfaction problems and constraint optimisation problems, both on multiprocessor and distributed systems.

Emphasis was given to scenarios where a large part of the search space has to be explored, such as finding all the solutions of a CSP or solving a COP. First solution results are usually similar to those obtained arriving at at best solution for a constraint optimisation problem [40], not allowing performance gains after some number of workers or teams are used.

The results presented support the claim that PaCCS is a scalable parallel constraint solver, able to achieve state-of-the-art speedups on several systems and with many solver configurations. This is confirmed by the values contained in Table 4.48, which summarises the total efficiency of the parallelisation of the solver for the esc16e QAP, comparing the speedup obtained with respect to sequential solving with the maximum, *i.e.* linear, speedup that could be expected, and the efficiency of distributed solving with respect to using only one team (except in the fhg-2 case, where a minimum of two teams were used).

The performance accomplished by PaCCS stems from the underlying architecture, designed with parallelism in mind, which is supported in part by viewing the store as basic building block and by the seamless integration of work stealing in the search process.

	Speedup	Maximum	Efficiency	Efficiency
System	w.r.t. 1 Worker			w.r.t. 1 Team
ha8000	n/a			0.98
ism	31.90	32	1.00	1.00
fhg-2	n/a			0.89
fhg-1	243.57	256	0.95	0.95
fhg-1*	222.46	256	0.87	0.87

Table 4.48: Parallelisation efficiency for QAP esc16e

Work stealing extends to teams, creating a two-level communication structure which is then further structured to achieve a good balance between accessibility among teams and scalability of the solver. This sets this work apart from earlier approaches which only considered one dimension of parallelism and often just glued extant solvers.

The results presented confirm that the approach followed in this work, which delivered a parallel solver, featuring a competitive sequential component, is valid, even though there remain areas to explore and improve.

Chapter 5

Closing Remarks

At the beginning of the work presented in this thesis, the plan was to explore hierarchical multiprocessor systems, namely networked distributed memory multiprocessor systems, for constraint solving and to develop a system capable of doing so. That system is the Parallel Complete Constraint Solver described here, a parallel constraint solver built from scratch with that goal in mind. While still susceptible to improvement, the results obtained so far are better or compare favourably with those obtained with the systems discussed in Section 2.4.

The system created allows the transparent use of the multiprocessing hardware increasingly available, sparing the users the burden to invest in learning the low-level issues involved in parallel and distributed programming.

The local work stealing scheme developed is a low contention, highly parallelisable, load balancing technique, which enables the full use of the power of multiprocessor computers. This is helped by the compact representation of the domain store used, and by the default creation of shareable work during the search process, which also leads to a setting where backtracking on failure is just one strategy available among others.

Distributed load balancing flows along a scalable architecture, which favours local communication patterns and reduces communication among more distant systems, especially if the underlying distribution system is able to provide that kind of information, and which does not display a performance bottleneck, which the master present in most of the current proposals constitutes. This is a novel communication model for work stealing, as is the reunion of dynamic load balancing with multiprocessor based parallel constraint solving.

Other feature that makes this work a step forward towards a more effective utilisation of constraint programming and networked resources is the internal problem representation, which readily enables the creation of a network constraint solving service.

Nevertheless, there are many ways by which this work can be extended, besides applying it to larger scale systems.

One is the inclusion of and-parallelism in the propagation process, which may help fully exploiting simultaneous multithreading in today's processors. Another is exploring other approaches to parallel search, where workers do not all perform the same task, but some may be dedicated to complementary work, which may help speed up the whole search, such as nogood synthesis from failures encountered during search, propagation, or helping define the better search strategy for the problem.

Other possible extensions relate to the system underlying the solver. New kinds of hardware, such as the graphical processing units, are being disseminated and could also be explored. A collaboration is currently under way to explore new paradigms, such as the Global Address Space Programming Interface (GPI) [28, 41]. Also of interest is the PM2 [59] system from the INRIA Runtime team, at Bordeaux, namely in what regards low-level process management.

Constraint programming is an area which has not yet realised its full potential, as the size of the problems has so far prevented its wider application under the current approaches. Parallel constraint solving continues to be studied and developed, and can contribute to a more widespread use of the constraint paradigm.

Appendix A

PaCCS Application Programming Interface

This appendix contains a brief presentation of the programming interface of PaCCS.

The current version is geared towards direct programming with the PaCCS primitives, in C, but the intended model is that of providing a back end to a higher level language featuring constraint modelling constructs.

A.1 Variables

The variables in PaCCS are finite domain integer variables. Their C type is `fd_int`, which is an opaque type.

Variable creation and domain initialisation is achieved through the primitive:

```
fd_int fd_new(int min, int max)
```

A.2 Constraints

Constraints in PaCCS are represented by the opaque `fd_constraint` type.

There are several built-in constraints. Their constructor is a function returning the newly created constraint.

The constraints available are the following.

A.2.1 Arithmetic Constraints

`fd_eq(fd_int x, fd_int y)`

$x = y$

`fd_ne(fd_int x, fd_int y)`

$x \neq y$

`fd_lt(fd_int x, fd_int y)`

$x < y$

`fd_le(fd_int x, fd_int y)`

$x \leq y$

`fd_gt(fd_int x, fd_int y)`

$x > y$

`fd_ge(fd_int x, fd_int y)`

$x \geq y$

`fd_minus_eq(fd_int x, fd_int y, int k)`

$x - y = k$

`fd_minus_ne(fd_int x, fd_int y, int k)`

$x - y \neq k$

`fd_var_eq_minus(fd_int x, fd_int y, fd_int z)`

$x = y - z$

`fd_var_eq_times(fd_int x, fd_int y, fd_int z)`

$x = y \times z$

A.2.2 Global Constraints

`fd_all_different(fd_int X[], int n)`

$\forall 0 \leq i, j < n, i \neq j \rightarrow X[i] \neq X[j]$

`fd_element(fd_int X[], int n, fd_int y, int k)`

$0 \leq y < n \wedge X[y] = k$

`fd_element_var(fd_int X[], int n, fd_int y, fd_int z)`

$0 \leq y < n \wedge X[y] = z$

`fd_sum(fd_int X[], int n, int k)`

$\sum_{i=0}^{n-1} X[i] = k$

`fd_sum2(fd_int X[], int n, fd_int y)`

$\sum_{i=0}^{n-1} X[i] = y$

`fd_sum_prod(fd_int X[], fd_int Y[], int n, int k)`

$X \cdot Y = k$

`fd_poly_eq(int C[], fd_int Y[], int n, fd_int z)`

$z \leq C \cdot Y \leq z$

`fd_knapsack(fd_int X[], fd_int Y[], int n, fd_int z)`

$z \leq X \cdot Y \leq z$

`fd_exactly(fd_int X[], int n, int c, int k)`

$\#\{i \mid X[i] = k\} = c$

`fd_exactly_var(fd_int X[], int n, fd_int y, int k)`

$\#\{i \mid X[i] = k\} = y$

A.2.3 Optimisation

`fd_min(fd_int x)`

Minimise the value of x.

`fd_max(fd_int x)`

Maximise the value of x.

Bibliography

[1] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[2] Christian Bessière. Constraint propagation. In Rossi et al. [50], chapter 3, pages 29–83.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[4] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In Craig Boutilier, editor, *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence IJCAI-09*, pages 443–448, Pasadena, California, July 2009. AAAI Press.

[5] Rainer E. Burkard, Eranda Çela, Stefan E. Karisch, and Franz Rendl. QAPLIB — a quadratic assignment problem library. `http://www.seas.upenn.edu/qaplib/`. Maintained by Peter Hahn and Miguel Anjos.

[6] Rainer E. Burkard, Eranda Çela, Panos M. Pardalos, and Leonidas S. Pitsoulis. The quadratic assignment problem. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 241–338. Kluwer Academic Publishers, 1998.

[7] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. QAPLIB — a quadratic assignment problem library. *Journal of Global Optimization*, 10(4):391–403, June 1997.

[8] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In Ian P. Gent, editor, *15th International Conference on Principles and Practice of Constraint Programming — CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241, Lisboa, Portugal, September 2009. Springer.

[9] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[10] The Comet programming language and system. `http://www.comet-online.org/`.

[11] Paul R. Cooper and Michael J. Swain. Arc consistency: parallelism and domain dependence. *Artificial Intelligence*, 58(1-3):207–235, December 1992.

[12] CSPLib: A problem library for constraints. URL `http://www.csplib.org/`.

[13] Rina Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, 2003.

[14] Boi Faltings. Distributed constraint programming. In Rossi et al. [50], chapter 20, pages 699–729.

[15] Lígia Ferreira. *Programação por Restrições Distribuídas em Java.* PhD thesis, Universidade de Évora, Portugal, 2004.

[16] Lígia Ferreira and Salvador Abreu. Design for AJACS, yet another Java constraint programming framework. *Electronic Notes in Theoretical Computer Science*, 48:167–178, June 2001. Declarative Programming – Selected Papers from AGP 2000.

[17] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI'98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. ACM.

[18] Gecode: Generic constraint development environment. `http://www.gecode.org/`.

[19] Grid'5000. URL `http://www.grid5000.fr/`.

[20] Youssef Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11(2):167–188, June 2002.

[21] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[22] Holger H. Hoos and Edward Tsang. Local search methods. In Rossi et al. [50], chapter 5, pages 135–167.

[23] IBM ILOG. URL `http://ilog.com/`.

[24] Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) based processors. `http://download.intel.com/design/processor/applnots/320354.pdf`, November 2008. White Paper.

[25] Intel Corporation. Intel Xeon processor 5500 series: An intelligent approach to IT challenges. URL `http://download.intel.com/pressroom/kits/xeon/5500series/pdf/ProductBrief_Xeon5500.pdf`, 2009.

[26] Joxan Jaffar, Andrew E. Santosa, Roland H. C. Yap, and Kenny Q. Zhu. Scalable distributed depth-first search with greedy work stealing. In *ICTAI'04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 98–103, Washington, DC, USA, November 2004. IEEE Computer Society.

[27] Simon Kasif and Arthur L. Delcher. Local consistency in parallel constraint-satisfaction networks. In *First Workshop on Principles and Practice of Constraint Programming PPCP93*, pages 139–145, Newport, Rhode Island, April 1993.

[28] Rui Machado, Carsten Lojewski, Salvador Abreu, and Franz-Josef Pfreundt. Unbalanced tree search on a manycore system using the GPI programming model. *Computer Science - Research and Development*, 26:229–236, 2011.

[29] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[30] Pedro Meseguer, Francesca Rossi, and Thomas Schiex. Soft constraints. In Rossi et al. [50], chapter 9, pages 281–328.

[31] Laurent Michel and Pascal Van Hentenryck. Comet in context. In *Principles of Computing and Knowledge: Proceedings of the Paris C. Kanellakis Memorial Workshop*, pages 95–107, San Diego, CA, June 2003.

[32] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Parallelizing constraint programs transparently. In Christian Bessière, editor, *13th International Conference on Principles and Practice of Constraint Programming — CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528, Providence, RI, USA, September 2007. Springer.

[33] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3):363–382, December 2008.

[34] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. COMET technical paper, Dynadec, 2009. URL `http://dynadec.com/resources/comet-technical-papers/`.

[35] Milipeia. URL `http://www.lca.uc.pt/`.

[36] John E. Miller. Langford's problem. URL `http://legacy.lclark.edu/~miller/langford.html`.

[37] The Mozart programming system. URL `http://www.mozart-oz.org/`.

[38] Shyam Mudambi and Joachim Schimpf. Parallel CLP on heterogeneous networks. Technical Report ECRC-94-17, European Computer-Industry Research Centre, 1994.

[39] Morten Nielsen. Parallel search in Gecode. Master's thesis, KTH - Royal Institute of Technology, Stockholm, Sweden, 2006. Available as Technical Report ICT/ECS-2006-45.

[40] Vasco Pedro and Salvador Abreu. Distributed work stealing for constraint solving. In German Vidal and Neng-Fa Zhou, editors, *Joint Workshop on Implementation of Constraint Logic Programming Systems and Logic-based Methods in Programming Environments (CICLOPS-WLPE 2010)*, Edinburgh, Scotland, U.K., July 2010.

[41] Vasco Pedro, Rui Machado, and Salvador Abreu. A parallel and distributed framework for constraint solving. In Philippe Codognet, editor, *Proceedings of the Workshop on Parallel Methods for Constraint Solving (PMCS'11)*, pages 41–44, Perugia, Italia, September 2011. Universita' di Perugia.

[42] Laurent Perron. Search procedures and parallelism in constraint programming. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360, Alexandria, VA, USA, October 1999. Springer.

[43] Laurent Perron. Practical parallelism in constraint programming. In N. Jussien and F. Laburthe, editors, *Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 261–275, March 2002.

[44] V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, 16(6):479–499, December 1987.

[45] V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. Part II. Analysis. *International Journal of Parallel Programming*, 16(6):501–519, December 1987.

[46] V. Nageshwara Rao and Vipin Kumar. Superlinear speedup in parallel state-space search. In *Eighth Conference on Foundations of Software Technology and Theoretical Computer Science FSTTCS'88*, volume 338 of *Lecture Notes in Computer Science*, pages 161–174, Pune, India, December 1988. Springer.

[47] Carl Christian Rolf and Krzysztof Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *2008 IEEE International Conference on Cluster Computing*, pages 304–309, 2008.

[48] Carl Christian Rolf and Krzysztof Kuchcinski. State-copying and recomputation in parallel constraint programming with global constraints. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2008*, pages 311–317, Toulouse, France, February 2008.

[49] Carl Christian Rolf and Krzysztof Kuchcinski. Parallel consistency in constraint programming. In Hamid R. Arabnia, editor, *PDPTA'09: The 2009 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 638–644, July 2009.

[50] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming.* Foundations of Artificial Intelligence. Elsevier, 2006.

[51] Tobias Schubert, Matthew Lewis, and Bernd Becker. PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.

[52] Christian Schulte. Parallel search made simple. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, pages 41–57, Singapore, September 2000.

[53] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Rossi et al. [50], chapter 14, pages 495–526.

[54] James B. Shearer. Golomb rulers page. `http://researchweb.watson.ibm.com/people/s/shearer/grule.html`.

[55] Marius-Călin Silaghi and Boi Faltings. Parallel proposals in asynchronous search. Technical Report TR-01/371, Swiss Federal Institute of Technology (EPFL), Lausanne, August 2001.

[56] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 1995.

[57] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[58] T2K Open Supercomputer Alliance. `http://www.open-supercomputer.org/`.

[59] Team Runtime. High performance runtime systems for parallel architectures. URL `http://runtime.bordeaux.inria.fr/Runtime/`.

[60] Peter van Beek. Backtracking search algorithms. In Rossi et al. [50], chapter 4, pages 85–134.

[61] Pascal Van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of CHIP within PEPSys. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 165–180, Lisboa, Portugal, June 1989. The MIT Press.

[62] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. In *Theory and Practice of Logic Programming*, pages 715–763. Cambridge University Press, November 2003.

[63] Barry Wilkinson and Michael Allen. *Parallel Programming*. Pearson, 2nd edition, 2005.

[64] Feng Xie and Andrew Davenport. Solving scheduling problems using parallel message-passing based constraint programming. In Miguel A. Salido and Roman Barták, editors, *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems COPLAS 2009*, pages 53–58, Thessaloniki, Greece, September 2009.

[65] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.

[66] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, June 2000.

[67] Ying Zhang and Alan K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, Dallas, TX, December 1991. IEEE.

[68] Ying Zhang and Alan K. Mackworth. Parallel and distributed finite constraint satisfaction: Complexity, algorithms and experiments. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence 1*, Machine Intelligence and Pattern Recognition, pages 305–334. Elsevier, 1994.

[69] Roie Zivan and Amnon Meisels. Concurrent search for distributed CSPs. *Artificial Intelligence*, 170(4–5):440–461, April 2006.

[70] Peter Zoeteweij and Farhad Arbab. A component-based parallel constraint solver. In Rocco De Nicola, Gian Luigi Ferrari, and Greg Meredith, editors, *Proceedings of the 6th International Conference on Coordination Languages and Models COORDINATION 2004*, volume 2949 of *Lecture Notes in Computer Science*, pages 307–322, Pisa, Italia, February 2004. Springer.

Index