UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Parallelization of Constraint Programming using Embarrassingly Parallel Search

Dissertation presented by
Guillaume DERVAL

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor
Pierre SCHAUS

Readers
Yves DEVILLE, Steven GAY, Sébastien MOUTHUY

Academic year 2015-2016

Abstract

Constraint programming solvers have a serial architecture, and do not take advantage of the parallel computing power available nowadays in our multi-core computers and in the cloud. Multiple solutions to remedy at this situation have emerged, such as Work-Stealing or Multi-threaded Portfolio Search, which allow to run multiple solvers and make them communicate in order to find solutions of the initial problem. However, these techniques scale badly with a high number of workers.

Embarrassingly Parallel Search is a recent method that aims at parallelizing solving of constraint satisfaction problems, by dividing them in numerous subproblems, and sharing them between multiple solvers. The high number of subproblems ensures proper balancing between the multiple workers. Previous works have shown nearly linear speedups and superiority of EPS over other methods on some problems.

This master's thesis proposes a new implementation of EPS over the OscaR-CP solver, with an innovative architecture based on a symbolic layer, called OscaR-Modeling, that is inspired by Objective-CP, a state-of-the-art modeling language. An analysis of the influence of the decomposition method over the speedup is made, and a new decomposition, called Cartesian Product Iterative Refinement, is proposed.

EPS is originally made for solving constraint satisfaction problems; this master's thesis introduces an adaptation for constraint optimization problems. A study of various methods to share the objective function bound is conducted.

Finally, an overview of the performances of EPS on OscaR-Modeling is given, and shows linear and super-linear speedups, even on more than one thousand threads.

Acknoledgments

I would like to thank my supervisor, Pr. Pierre Schaus, for the numerous exchange of ideas around my master's thesis, and Pr. Jean-Charles Régin, from the University of Nice-Sophia-Antipolis, for the insights he had given to me during the seminar at the start of the academic year.

I would also like to thank Anthony Gégo, Mathieu Jadin, Gautier Tihon, François Michel and Maxime Piraux who listened to me an incredible number of hours while I was explaining a large quantity of uninteresting facts about my work on this master's thesis.

The experimental part of this work has been made possible by the usage of computational resources that have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11.

Contents

Glossary

AST Abstract Syntax Tree.

BACP Balanced Academic Curriculum Problem.

COP Constraint Optimization Problem.
CP Constraint Programming.
CPIR Cartesian Product Iterative Refinement.
CSP Constraint Satisfaction Problem.

DI Detected Inconsistent (subproblems).
DSL Domain-Specific Language.

EBNF Extended Backus-Naur Form.
EPS Embarrasingly Parallel Search.
η Symbol for efficency, the ratio between the speedup and the power ratio between two architectures.

FC Forward Consistency.

GAC Generalized Arc Consistency.
GCC Global Cardinality Constraint.

IDDFS Iterative Deepening Depth-First Search.

MIP Mixed-Integer Programming.

NDI Not Detected Inconsistent (subproblems).

QAP Quadratic Assignment Problem.

S Symbol for speedup, the ratio between the latency of an architecture and the latency of another one.
SAT Boolean SATisfiability problem.

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

The computer science field is currently facing a shift of paradigms: today, parallel computing power is cheap, abundant, and continuously improving, while serial computing power is stalling [2]. The cloud is no longer a *buzzword*; multiple providers now offers virtually unlimited distributed computing power.

Constraint programming solvers are nonetheless mostly serial programs. Researchers have proposed multiple solutions to adapt them for parallelism and distribution, such as Work-Stealing, which allows multiple CP solvers to work on a single problem by stealing work from others, or Portfolio Search, for optimization problems, that uses numerous different tree search heuristics to find solutions and share the optimization bound between solvers.

These methods work well in architecture with a limited degree of parallelism, for example on fewer than 30 threads (or workers), but rapidly show their limitation: work-stealing relies too much on communication, and portfolio search has difficulties to produce sufficiently different searches to scale properly.

Embarrassingly Parallel Search[25] (EPS) is a recent technique that aims at both parallelizing and distributing constraint satisfaction and optimization problem resolution. Its main idea is simple: it involves the division of the initial problem into numerous subproblems, and the distribution of these among the workers. The very high number of subproblems leads the solving time to be balanced between the workers, which itself allows to reach a linear speedup.

Despite its apparent simplicity, implementing EPS offers interesting technical challenges, and this master's thesis will show the importance of the decomposition method, the representation of the subproblems, and the communication of the optimization bound.

This master's thesis proposes a new implementation of EPS, on OscaR-CP, a state-of-the-art constraint programming solver, in a new symbolic layer called OscaR-Modeling.

This manuscript is organized as follows. First, an overview of the core concepts of constraint programming, parallel and distributed computing is presented, along with a short analysis of the state-of-the-art for CP parallelization. An overview of modeling languages is also made, to put in perspective the architectural decisions behind OscaR-Modeling.

Chapter 3 gives an in-depth description of EPS, and describes the importance of the decomposition method and the way the optimization bound is shared. A new decomposition, called Cartesian Product Iterative Refinement, is introduced.

Chapter 4 describes OscaR-Modeling. Its domain specific language is shortly introduced, along with its semantic. OscaR-Modeling offers new possibilities thanks to its symbolicity, such as algebraic pre-solving, which is also introduced in this chapter. The internals are then presented, along

with the implementation of EPS itself. The chapter ends with an analysis of the performances, which shows linear and super-linear speedups even at a thousand threads.

Contributions

The following contributions are part of this master's thesis:

- OscaR-Modeling, a symbolic layer for OscaR, with new multithreading and pre-processing features. This is the main part of this master's thesis.
- Description of new model decomposition methods for EPS (notably the Cartesian-Product Iterative Refinement).
- An analysis of performance of EPS for solving COPs notably bound transmission influence.
- A comparison of EPS with Work-Stealing, another state-of-the-art method for parallelization.

Chapter 2

Core Concepts, State of the Art

This chapter aims at describing the basics of constraint programming (CP), parallelization, and presents methods that have been created to parallelize the solving of constraint satisfaction and optimization problems. Moreover, a short analysis of modeling languages is done, in order to put into perspective chapter 4 that will describe OscaR-Modeling.

2.1 Constraint Programming

Constraint Programming (abridged CP from now on) is a paradigm used to solve Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP). Such problems being NP-Hard, CP uses exhaustive search to find (optimal) solutions for these problems. The main concept behind CP is to use polynomial functions, enforcing constraints, to reduce the search space[3][28].

Definition 1. *(Constraint) (from [28]) A constraint c is a relation defined on a sequence of variables* $X(c) = (xi_1, ..., xi_{|X(c)|})$. *c is the subset of* $\mathbb{Z}^{|X(c)|}$ *that contains the combinations of values that satisfy c.* $|X(c)|$ *is called the arity of c.*

Definition 2. *(Constraint satisfaction problem) A constraint satisfaction problem(CSP) is defined as:*

- *A set of variables* $X = x_1, x_2, ...x_n$, *each having a finite set of acceptable values* $D_i \subset \mathbb{Z}$ *(that is called the domain of the variable)*
- *A set of constraints on these variables, that all solutions must respect. These constraints then reduce the number of possible values for each variable.*

Definition 3. *(Constraint optimization problem) A constraint optimization problem is a CSP with an additional goal of finding the solution(s) that minimize, maximize, ..., a particular objective function.*

CP solvers use exhaustive search to solve CSPs and COPs, and are thus being able to provide all the solutions and even prove optimality in the case of optimization problems. In order to reduce the exponential search space, CP solvers make an extensive use of polynomial functions enforcing constraints (at different levels of consistency). These functions are sometimes abusively called *constraints* themselves.

Enforcing the constraints is done by means of a fixed-point algorithm that *propagates* information given by the constraints to other constraints. The fact of using the fixed-point algorithm is called *constraint propagation*.

The exploration of the *search tree* (tree produced by the combination of the exhaustive search, the search method, and the fixed-point algorithm) is led by the *search method*, also called simply *search*. Each time the search method is called, we say we make a *branching* in the search tree. The *search* is a very important part of the solving, mostly with COPs, as:

- The constraint propagation is sometimes dependent on which variables are assigned
- The search "direction" can help to find better solutions faster during optimization, giving new information about the optimization bound that can help to *cut* further the search tree.

A very common mantra in the CP community is:

$$\text{CP} = \text{Model} + \text{Search}$$

This mantra shows the importance given to both a good model design, but also to a well-made search method. The design is fundamental in constraint programming, as trade-offs have to be made. Propagation time versus propagation power is one of the most important: choosing between powerful filters that takes time to propagate but cut more the search, or weaker ones that are faster but cut less is the difference between having a solving that runs in seconds or in hours. CSPs and COPs in general are NP-hard: large CP problems can take a very long time to solve, particularly if the initial problem is wrongly modeled.

This short overview of constraint programming is sufficient for understanding the next chapters of this master's thesis. More details, insights, and explanations can be found in [28] or [22] for example.

2.2 Parallel and Distributed Computing

Modern processors handle more and more parallel instructions, while the sequential power tends to increase less due to technological limitations. Moreover, a nearly unlimited distributed computing power is available for very low prices by the means of the Cloud.

Parallel and distributed computing, which involves simultaneous computations on one or multiple machines, are then an ever-growing field of research.

Software is mostly written as a single sequence of instructions that are executed one after another; this sequence is called a *thread*. Parallel computing makes the use of multiple threads that are located on a same machine, in order to use all the computing power available, while distributed computing extends this principle to multiple, possibly very distant, machines.

The degree of parallelism of a given algorithm can be measured using the speedup[1] and the efficiency:

Definition 4. *(speedup in latency) The speedup (noted S) between two architectures is the ratio between the time taken to run an algorithm on the new architecture and the time taken on the old one.*

Given that the algorithm run in T_1 seconds on the original architecture, and T_2 on the new one:

$$S = \frac{T_2}{T_1}$$

Definition 5. *(efficiency) The efficiency (noted η) is the speedup divided by the number of parallel units n used to obtain this speedup. It can be used to measure the ability of a system to use more*

[1] Two definitions of speedup exist: the speedup in latency and in throughput. In this master's thesis, only the speedup in latency is used, and thus most of the time abbreviated as simply "speedup"

units.

$$\eta = \frac{S}{n} = \frac{T_2}{nT_1}$$

In a perfect world, making parallel software would be simple and all the speedups would be linear (the efficiency would always be one). This is not the case in practice, and most parallel software behave well with a low to middle degree of parallelism, but fail to have a high efficiency in highly parallels environment.

Two factors decrease the efficiency of parallel programs. The first one is the increasing importance of the sequential part of the program. This behavior is formalized in Amdahl's law:

Theorem 1. *(Amdahl's law, derived from [1]) Given s, the speedup of the parallel part of the algorithm, and p, the percentage of the algorithm that can be parallelized, an upper-bound for the speedup of the complete algorithm is*

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

The second factor decreasing the efficiency is the *communication*. Most parallel algorithms need to communicate between different execution threads, in order to share information about the work, to update a value, etc. This communication, which can take many forms (message passing, shared memory using locks, synchronization, ...) can take a lot of time:

- In distributed environments, sending messages via the network can take milliseconds.
- In shared memory environments, locking a particular part of the memory in order to update a value requires to stop other threads that may want to access this value.

Efficient parallel and distributed algorithms then need very little communication and have a very small or non-existent sequential part that cannot be parallelized. Algorithm that has no communication and sequential part (or approximately no) are called *embarrassingly parallel*.

2.3 State-of-the-Art Methods for Parallelizing Constraint Programming

2.3.1 Static Decomposition

Static decomposition[4] of the search tree is the simplest way to parallelize or distribute a CP solving problem. Given m workers, a static decomposition algorithm divide an original CSP/COP P into m *subproblems* $p_1, p_2, ..., p_m$, such that $\bigcup_{i=1}^{m} p_i = P$. Each sub-solver is then given a subproblem and resolves it.

The performances of this method are poor, mostly because it is important to balance the solving time among the workers: the subproblems must have roughly the same size. This is, of course, very difficult to estimate and requires an in-depth knowing of the form of the tree, which is most of the time unavailable.

2.3.2 Work-Stealing

As cutting a search tree in m parts of similar size is difficult or impossible in a reasonable time, leading to unbalanced solving time between the worker, a more viable approach is then to be able to "redistribute" the work.

The work-stealing[17, 5, 12] method is based on adding to the workers the capacity to "steal" works from the others. The method itself is described in algorithm 1.

Algorithm 1 Work-stealing

Require: P, the initial CSP/COP
  m workers.
  Initially, all workers are idle
  Start solving of P on worker 1
  while not all workers are idle **do**
    wait for an idle worker
    $w_1 \leftarrow$ an idle worker
    $w_2 \leftarrow$ a non-idle worker
    Stop w_2
    Divide the problem w_2 is solving in two parts, p_1 and p_2
    Restart w_1 with problem p_1
    Restart w_2 with problem p_2
  end while

  Gather results from all the workers

Multiple variants exist, notably:

- On how to implement the division efficiently
- On where to divide to maximize the performances [5]
- On the architecture used to allow stealing [12]
- On the way COPs are handled (bound transmission)
- On when to stop allowing stealing (in order to reduce the communication at the end of computations)

Work-stealing is considered as the state-of-the-art for parallel solving of CSP/COP, and provide linear speedups in lowly parallel environments. However, the method involves a lot of communication between the workers, mainly at the start and the end of the computation, that leads to a brutal loss of performance in highly parallel environments or distributed ones, as shown in section 4.4.1. Moreover, when solving COPs, reasoning about the search becomes difficult using work-stealing.

Gecode [7] is the implementation of reference for Work-Stealing.

2.3.3 Portfolio Multithreading

Portfolio multithreading[4] is a completely orthogonal approach to parallelizing CSP/COP solving compared to Work-Stealing, Static Decomposition and EPS.

Portfolio multithreading focuses on COPs. Its principle consists of using different search methods (different tree forms and ordering of the nodes) on the same problem, on distinct solvers, that are running in their own threads (or workers). The bound information is then shared between the multiple workers in order to cut the search trees.

The key point is the generation of enough search strategies. While using hand-made search strategies is sufficient and feasible when solving using two or three threads or workers, creating thousands is impossible by hand. Several articles, notably [11] for SAT solvers, have proposed ways to generate random searches.

The results[4] are problem dependent and tend to offer a sub-linear speedup.

2.3.4 Embarrassingly Parallel Search

Embarrassingly Parallel Search[25] (EPS) is a new technique that aims a solving CP problems on massively parallel or distributed environments. Its principle is simple: EPS involves the decomposition of an initial CSP/COP into a huge number of subproblems, that are then solved independently by a less high number of workers. EPS is then a direct improvement of the Static Decomposition method, with a key difference: the number of subproblems. While in Static Decomposition, each worker has a single subproblem to solve, in EPS they have multiple subproblems.

The solving part itself can be implemented in two different ways:

- Static sharing: subproblems to solve can be shared a-priori between the workers.
- Dynamic sharing: subproblems can be put into a queue that is pulled each time a worker becomes idle.

The central idea behind EPS is that, if the initial CSP/COP is divided in a sufficiently high number of subproblems, the running time on each worker would differ by only a tiny fraction. Chapter 3 will describe more completely the maths behind EPS.

Dividing a CSP/COP in a very high number of subproblems is sightly more difficult than with a low one. In [25] the authors propose a decomposition strategy that only generates *Not Detected Inconsistent subproblems* (NDI), which are problems not proved inconsistent by the fixed-point algorithm. This allows to avoid considering numerous *Detected Inconsistent* (DI) subproblems. The authors show that the ratio between DI and NDI subproblems can be very high on some problems.

EPS is a generic method, and depends on the following points:

- The decomposition is, as said earlier, not a trivial task. Chapter 3 will show that while the balancing of the solving time increases with the number of subproblems, it can be reached faster using a carefully designed decomposition.
- The way subproblems are stored in memory and communicated to the workers is important to reach good speedups
- Handling the bound when solving COPs is also a critical point.

This master's thesis takes its basis on EPS, and aims at improving the performance of EPS, particularly when solving COPs.

2.4 Overview of modeling languages

In this section, we will review some CP solvers and modeling languages. The goal is to introduce the state-of-the-art of the modeling languages and of the solvers themselves, and to present the ideas that have been reused for OscaR-Modeling (see chapter 4).

The reader should keep in mind the final objective of this work: to provide a simple, easy-to-use and efficient implementation of a modeling tool, that will have a support for EPS, on OscaR.

2.4.1 Low-level libraries

Most of the open-source solvers offer an interface in their "host language", and this interface is most of the time a very trivial DSL.

Gecode [7] is a toolkit, written in C++, that allows to model and run constraints solving problems (including optimization ones). It is considered in the community as one of the best solvers available and implements most of the state-of-the-art techniques available. In the context of parallel solving, Gecode uses the work-stealing method.

The modeling is made through an interface to Gecode written in C++. As C++ is a low-level language, it is somewhat expected that it is more difficult to write a model using Gecode than other modeling languages. Listing 2.1 provides an excerpt of the implementation of the Golomb-ruler model using Gecode.

```
IntVarArray m(*this,opt.size(),0, 1 << (opt.size()-1))-1);
const int n = m.size();       // Number of marks
const int n_d = (n*n-n)/2;    // and differences
IntVarArgs d(n_d);            // Array of differences

rel(*this, m[0], IRT_EQ, 0); // Assume first mark to be zero
rel(*this, m, IRT_LE);       // Order marks


for (int k=0, i=0; i<n-1; i++)
  for (int j=i+1; j<n; j++, k++)
    rel(*this, d[k] = expr(*this, m[j]-m[i]), IRT_GQ, (j-i)*(j-i+1)/2);

distinct(*this, d, opt.icl()); //all different constraint on differences

rel(*this, d[0], IRT_LE, d[n_d-1]); //symmetry breaking

branch(*this, m, INT_VAR_NONE(), INT_VAL_MIN()); //search
```

Listing 2.1: Golomb-ruler model in Gecode

Gecode makes an extensive usage of constants(`IRT_EQ`, `IRT_LE`, ...) and class methods (`distinct`, ...) to model the problem, and do not make use of operator overriding that the host language offers; without comments or access to the documentation the initial comprehension of the model is objectively difficult compared to other languages.

This lack of specific and efficient DSL is balanced by the speed due to the quality of the implementation and the nearly non-existent overhead of C++.

Other solvers, such as Choco[23] (written in Java)[2] or the CP part of Or-Tools[9] (written in C++)[3] offer such a simple DSL.

2.4.2 Solver-agnostic modeling languages

As having multiple library- or application-specific languages is rather impractical to compare them using given models, multiple solver-agnostic languages have emerged. One of the well-known one is MiniZinc.

```
include "globals.mzn";
```

[2]Please note that the developers of Choco recently developed DSL in Scala that interface to Choco. It is somewhat similar, although a bit less developed, than the one of OscaR, presented later.

[3]Or-Tools also offer interfaces in Python and C#, that are more elegant but do not allow to use all the power of Or-Tools directly, some functionalities being hidden by the interface.

```
2
3  int: m;
4  int: n = m*m;
5
6  array[1..m] of var 0..n: mark;
7  array[1..(m*(m-1)) div 2] of var 0..n: differences =
8  [ mark[j] - mark[i] | i in 1..m, j in i+1..m];
9
10 constraint mark[1] = 0;
11 constraint forall ( i in 1..m-1 ) ( mark[i] < mark[i+1] );
12 constraint alldifferent(differences);
13
14 constraint differences[1] < differences[(m*(m-1)) div 2];
15
16 solve :: int_search(mark, input_order, indomain, complete)
17
18 minimize mark[m];
19 output [show(mark)];
```

Listing 2.2: Golomb-ruler model in MiniZinc

MiniZinc[20] and other agnostic languages are a language on their own; they have the advantage of not having to deal with a host language and to focus on modeling while leaving all the other details on the solver.

On the contrary, since MiniZinc models are not expressed in the solver host language, it needs to be parsed and its content converted to the way the solver stores its models, which can be very different.

Listing 2.2 shows an exemple[4] of a golomb ruler model in MiniZinc.

2.4.3 OscaR

OscaR[21] is a toolkit for Operations Research, written in Scala, that notably hosts a state-of-the-art CP solver (adequately named Oscar-CP). The development team of OscaR-CP put a particular emphasis on its modeling DSL, which is, with Objective-CP, one of the best example of a CP modeling DSL. Written as a library, OscaR allows to write model directly as CP Program, just as Gecode and other libraries presented earlier do.

Oscar-CP uses most of the DSL features of Scala, its host language, to propose an efficient modeling:

- Implicits
- Implicit conversions
- Postfix notation
- Traits
- Closures
- By-name parameters

The DSL used to make creating custom searches, or solution handlers simpler than the languages previously described. Listing 2.3 shows an example of a Golomb ruler in Oscar-CP[5]. Listing

[4] Source: `https://github.com/MiniZinc/minizinc-benchmarks/blob/master/golomb/golomb.mzn`

[5] Please note that this model is not exactly equivalent to the other models presented earlier. The version presented here is an excerpt from the sources of OscaR-CP

2.4 shows an alternative custom search on this model, on which the usage of closure and by-name parameters is explicit.

```
val mark = Array.fill(m)(CPIntVar(0 to n))
val differences = for { i <- 0 until m; j <- i + 1 until m } yield mark(j) -
    mark(i)
minimize(mark(m - 1))
add(allDifferent(mark), Strong)
increasing(mark)
add(mark(0) == 0)
add(mark(1) - mark(0) < mark(m - 1) - mark(m - 2))
differences.foreach(d => add(d > 0))
search { binaryStatic(mark)  }
```

Listing 2.3: Golomb-ruler model with OscaR-CP

```
//define a "closure" (given as a by-name parameters) that will be used a
    search
search {
  val nextMark = mark find(!_.isBound)
  if(nextMark isDefined) {
    val value = nextMark.min
    //branch on two "closures" (by-name parameters again) that will be
    evaluated later during search
    branch(add(nextMark == value))(add(nextMark != value))
  }
  else
    oscar.cp.noAlternative
}
```

Listing 2.4: Custom search using OscaR-CP

OscaR-CP is the code basis of this master's thesis, and its DSL will be reused and improved in Oscar-Modeling.

While being an innovator in the way models are described, the DSL of OscaR-CP is focused on being an interface to the CP solver, like what we saw in Gecode and other solvers. Modeling and solving are not uncoupled as in MiniZinc.

2.4.4 Objective-CP

Objective-CP[31], that is written in Objective-C, works differently than solvers presented earlier: it makes a distinction between the model, and the program (combination of model plus solver). While OscaR and others directly instantiate variables in memory and run propagation algorithms on them when constraints are pushed, Objective-CP keep the variables and constraints "as-is", until the model is *instantiated*("concretized").

More precisely, the modeling language of Objective-CP intends to be "solver-type"-agnostic: the models can be solved using both the integrated CP solver, or a solver for linear programming, for example. Moreover, Objective-CP introduces the concept of *model operators*.

A model operator is a function that takes a model as input, and returns another one as output, with a single restriction: the variables of the initial model must be present in the output model.

This, combined with the Small-talk-like message mechanism of Objective-C, allows to define searches and other methods in order of *model* variables instead of *instance*(or "concrete") variables; leading to re-usability among multiple types of solvers.

Model operators and instantiation operators can be a very powerful abstraction for multiple tasks, such as model hybridization and parallelizing.

Chapter 3

Embarrassingly Parallel Search

As stated in section 2.3.4, Embarrassingly Parallel Search (EPS) is a direct improvement of the static decomposition technique, which, like its predecessor, aims at enabling to solve constrained problems in a parallel or distributed way.

EPS uses multiple independent CP solvers, called workers, that solves parts of the original problem. By decomposing the initial problem to be solved in numerous subproblems, EPS ensures that the solving time on each worker is (statically) balanced.

In order to transmit the subproblems to the workers, two methods can be used:

- Static sharing: subproblems to solve can be shared a-priori between the workers.
- Dynamic sharing: subproblems can be put into a queue that is pulled each time a worker becomes idle.

The law of large numbers can be used to show that the solving time balancing is improving when dividing into more subproblems:

Theorem 2. *Let m be the number of workers, s the number of subproblems per worker and P a constraint satisfaction or optimization problem, divided in $n = m \cdot s$ subproblems $P_1, P_2, ...P_n$ and such that $P = \bigcup_{i=1}^{n} P_i$ and that subproblem P_i takes t_i seconds to be solved. Moreover, $\sum_{i=1}^{n} t_i = T$, the time taken to solve P. Let us divide the work among all workers such that worker j receive subproblems W_j. Then, for each worker j, we have that $\lim_{s\to\infty} \sum_{i\in W_j} t_i = \frac{T}{m}$.*

Proof. This is a direct consequence of the law of large numbers in probability theory. □

Theorem 2 is true in a statistical way, but in practice can require a very high number of subproblems to allow reaching this optimal case, depending on how the problem is decomposed a-priori[1]. Of course, the number of subproblems to generate is a tradeoff: generating subproblem and sharing them among the workers takes time.

A bound on the solving time can be obtained by considering the largest subproblem (that takes the most time to be solved):

Theorem 3. *(from [25]) Using dynamic sharing, let $t_{max} = \max_{i=1}^{n} t_i$. Then (i) The minimum running time for solving all the subproblems is t_{max}. (ii) The maximum inactivity time for a worker is t_{max}.*

In [25] the authors assumes that when the number of subproblem increases, as a consequence, the value of $t_{\max}$ should also decrease. While this is a reasonable assumption, the theorems 2 and 3 also tell us that the results of EPS can be improved using decomposition strategies that minimize

[1] Static sharing is used in theorem 2, but it remains true with dynamic sharing

$t_{\max}$.

Decomposing a CSP/COP is then not a trivial task, as it should minimize the size of the largest subproblem. Moreover, the time taken by the decomposition itself is important (as it is the "sequential part" that participate in the Amdahl's law), and the way subproblems are stored in memory and transferred is equally important to allow to attain good speedups.

In [25] the authors propose a decomposition strategy that only generates *Not Detected Inconsistent subproblems* (NDI), those being problems not proved inconsistent by the fixed-point algorithm. *Detected Inconsistent* (DI) subproblems have by definition an empty search space, and thus do not need to be considered.

The method proposed is an Iterative Deepening Depth First Search (IDDFS), made on top of the CP solver: the algorithm incrementally runs the CP solver using an n-ary search, with a given cut-off based on the height of the tree. If the number of subproblems (leaves in the search tree) is not high enough, a table constraint containing all the current solutions is added to the model, and the cut-off height is increased. As [25] focuses mainly on enumerating all the solutions to CSPs, this cut-off is not an issue. However some particularities must be taken into account while solving COPs with EPS:

- The decomposition: The objective is to design a decomposition such that all the subproblems are equally difficult to solve. This not only allows to minimize $t_{\max}$, but it also ensures that the interesting parts of the search tree are shared more evenly between the workers, leading to better chances to find the optimum faster. This is, of course, difficult to estimate, especially for optimization problems where the difficulty also depends on the upper-bound of the objective function.
- The ordering of the sub-problems: The subproblems that are more likely to contain optimal solutions should be solved first (this is intuitively similar as the impact of a good value heuristic) such that the best-bound can quickly be used by all the workers to prune most of the subsequent problems that are explored only to proof the optimality. While being important, problems ordering has no impact for CSPs (Régin et al. even suggested to randomize it) it has a crucial impact on the efficiency for solving COPs.

A further analysis of decomposition methods and proposals for a new one is presented in section 3.2.

Moreover, solving COPs involves minimizing or maximizing a given objective function. In most CP solvers, this is done using a branch-and-bound algorithm, which uses the best solution found so far to reduce the solution space. These tree cuts are primordial in most optimization problems that would be too large to be solved without them. As EPS involves spreading the original search tree on multiple workers, transmitting the current optimization bound between the workers become an important point in order to maintain a good efficiency. Section 3.1 will describe the effect of multiple ways of sharing this bound.

Section 3.3 finishes this chapter by giving an insight at possibilities provided by EPS, and how it can be combined with other ideas in the future.

3.1 Analysis of Solving COP Using EPS

The way bound information is transmitted among workers is then important in order to reach a reasonable efficiency. Multiple methods are possible:

- **No communication** Not communicating the optimization bound between the workers is a

possibility. This is the way EPS is originally designed [25]. Having no communication is, of course, an advantage considering that sending messages (or share information between workers, in a more general way) takes time and can slow down the whole process; however, without bound information, the search space explored will be far greater than with a sequential search.

- **No bound update during subproblem resolution** Another possibility is to make workers retrieve the bound each time a new subproblem is pulled from the main queue. This is equivalent to say that no bound updates will be received by a worker during the resolution of a subproblem. This is a middle-ground between instant bound update and no communication at all. It comes at a cheap price: when using dynamic sharing, subproblems already have to be communicated, and adding the current bound to the message is a very small modification (most of the time, the optimization bound is a single 32-bit integer).

  In order to share the bound with the worker when subproblems are pulled, the workers must send back the new bounds found in local solutions to the master. Two methods are possible:

  - **Instant update** A message is sent to the master directly when a solution is found.
  - **At the end of a subproblem resolution** A message, containing the best solution/bound found in the current subproblem, is sent when the solving of the subproblem ends.

- **With bound update during subproblem resolution** This final option forces workers to interact directly with the solver[2] to update the bound during resolution. This method is, of course, the one that gives the most information and enables to reduce the search space the most. However, it can involve a lot of communication, and may be difficult to implement in an efficient way[3].

  The way the bound on the objective function is shared can be implemented using the master queue, or not. Three options are possible:

  - **Instant update, worker-to-worker** bound update is sent immediately by the worker that finds the new solution, to all the other workers. In distributed architecture, such a method can lead the network to be overused. Since no central agent does bookkeeping, a worker can send a message to all workers while another message, with a better bound, has not yet been received but already sent.
  - **Instant update, worker-to-master** bound update is sent immediately by the worker that finds the new solution, to the master, which then sends the bound update to the other workers if the new optimization bound is the best bound found yet. This reduces the communication problem of the previous method, but can lead to a higher latency (since two messages need to be exchanged to update a given worker).
  - **At the end of a subproblem resolution** A message is sent to the master at the end of the subproblem solving. This solution makes little sense compared to the other two, as there is already communication involved.

Using no communication is not efficient in practice, as the subproblems have to be retrieved from the master (the bound can then be pulled in the same time). Moreover, using a worker-to-worker architecture reveals to be complicated and tend to send too many messages for a gain of merely some milliseconds, compared to the worker-to-master architecture. It is possible to compare the (sound) methods by communication and search space reduction:

No communication

[2] that may not be possible with some solvers

[3] The implementation in OscaR-Modeling is based on a Java `volatile` variable that is checked each time the search branching function is called. Since this partly bypass the cache of the CPUs, a little overhead exists.

$\leq$

No update during resolution, bound sent back at the end of resolution

$\leq$

Update during resolution, bound sent back immediately

The following sections will analyze the influence of the last two methods on the speedup and efficiency.

3.1.1 Without external bound update during subproblem resolution

Using no external bound updates provide an upper bound on the speedup and the efficiency:

Theorem 4. *In a minimization problem, given subproblems are run in the same order as a sequential search would do, the upper bound available at each subproblem start in a parallel environment is less than ($\leq$) the upper bound in a sequential search. This effect is increasing with the number of threads.*

Proof. Given that you are at subproblem number n, and that you have p threads solving subproblems simultaneously. Since the subproblems are started sequentially, the worst case is that the subproblems numbered $\{0, 1, ..., n-p\}$ are already solved, and that problems $\{n-p+1, n-p+2, ..., n-1\}$ are currently running in the other threads, i.e. we don't have the bound information for these problems yet, as would have had a completely sequential resolution. □

Theorem 5. *The number of nodes explored by a branch-and-bound algorithm, given a specific bound, is higher than the number of nodes that is visited given a better bound.*

Proof. Straightforward considering how branch-and-bound works: a smaller upper bound will prune at least as many nodes at the higher one. □

This leads us to this somewhat disappointing conclusion:

Theorem 6. *A multithreaded resolution of a COP using a static problem decomposition (without external bound update) will explore as least as many nodes as its sequential equivalent.*

Proof. Direct from Propositions 1 and 2. □

Corollary 1. *Static problem decomposition (and then EPS), given that no external bound update are made during a subproblem resolution, can only lead to $\eta < 1$.*

While this is expected for any parallel application, it is still worth noting that this result theoretically limits us.

As shown in the proof of theorem 1, a subproblem with id n in p-thread environment has the bound information about all the subproblems with id $n' \leq n - p$. This simple fact can be used to roughly estimate the number of nodes that will be unnecessarily explored compared to a sequential ($p = 1$) resolution.

Proposition 1. *When the number of subproblems $\to \infty$, the loss of information about the bound on the objective function becomes negligible.*

There is then (as usual) a balance to be done between the loss of bound information and the subproblem generation overhead (which also involve the communication between the threads): too many subproblems would take too much time to generate, while having a very small number of subproblems would lead to a very high amount of supplementary visited nodes.

3.1.2 With external bound updates

While it is more difficult to make strong theorem and propositions about the number of nodes that are visited in a parallel resolution with external bound update, the following theorem shows external bound updates can lead to super-linear speedups:

Theorem 7. *There exists COPs where parallel resolution using external bound updates can lead to* $\eta > 1$

Proof. Let us construct an example. Let be a COP divided in two subproblems; the first one which is huge and can be heavily pruned with a better upper bound, and the other which contains the optimal solution and is small.

Solving this problem with $p = 2$ will lead to a $\eta > 1$ since the second problem will be solved rapidly, leading to a huge pruning in the first one. In a sequential ($p = 1$) environment, such a pruning would not have been possible. □

From a qualitative point of view, this result will only compensate in part the loss of information showed in theorem 4. This new bound information can only be obtained from the subproblems that are running in the other threads and that started "after" (in the $p = 1$ environment) the problem being currently updated.

The proof constructs a very specific search tree, that may seem infrequent in practice. However, real examples exist: some instances of the BACP presented in [19] [4] takes more than 600 seconds to be solved on a single thread, but fewer than one second when divided in four parts and ran on four threads, giving an efficiency of, at least, 150. This is not an isolated case; models on which the filtering with the bound heavily cuts the tree tend to offer similar behaviors. The final chapter, about performances, will show multiple examples of more reasonable super-linear speedups on others COPs.

3.1.3 Choosing a Bound Communication Method

Both of these methods can be used in practice, and some problems will probably require to use a particular one. However, and from a more empirical point of view, most COP solvings find only a little amount of new solutions (from tens to hundreds): having immediate bound update is then the preferable choice, given the low amount of communication involved and the great speedup that can be offered. There is still an overhead caused by the update in the solver, however, it is most of the time small compared to the gain offered by direct updates.

Figure 3.1[5] shows the speedup obtained by OscaR-Modeling using different communication methods on different models. The three methods tested are:

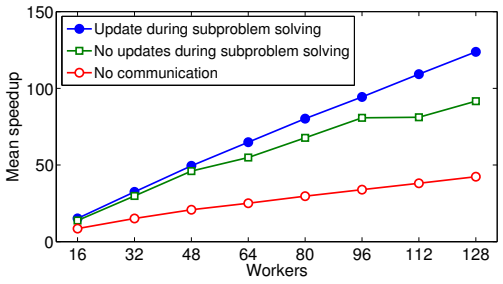
- No communication (bound is kept locally)
- No updates during subproblem solving (no updates are made once the solver is started, and the master is updated at the end of each subproblem solving)
- Update during subproblem solving

These figures show that, in various proportions, communication is beneficial for solving these problems.

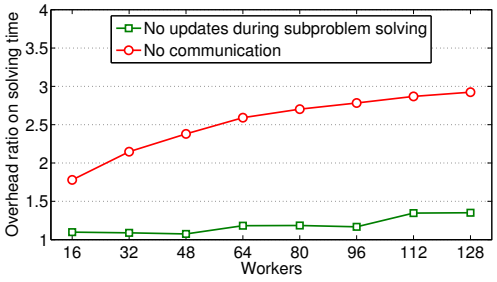
Solving times (on 40 workers) of a Golomb-ruler of size 13, with the three different communication methods are presented in figure 3.2. As can be observed, the difference between having,

[4] for example `instances12/inst88.txt` that is available in the OscaR repository

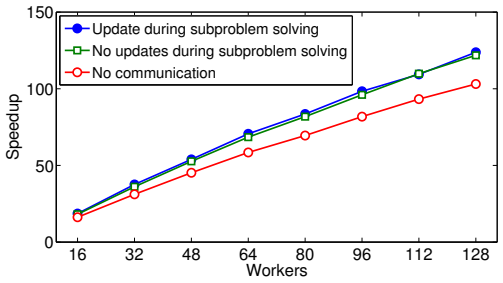
[5] See [10] for a description of QAP; the instance chosen is derived from this source.



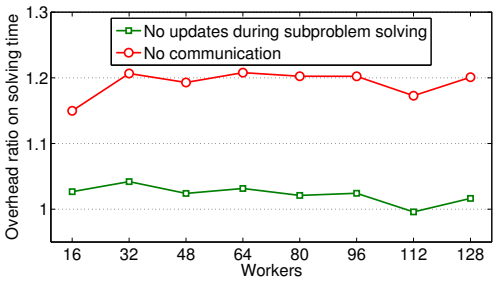
(a) Mean speedup (three runs), Golomb-ruler of size 13



(b) Overhead in solving time, compared to immediate update, Golomb-ruler



(c) Mean speedup (two runs), QAP, instance for $n = 11$ derived from [10]



(d) Overhead in solving time, compared to immediate update, QAP

Figure 3.1: Performance of different communication methods

or not, bound updates during solving tends to reduce with the number of subproblems, as said in proposition 1. Using immediate communications of the bound provides better results for most number of subproblems per worker, until the difference between the two methods becomes negligible compared to the additional decomposition time needed to provide more subproblems. As a side note, the best number of subproblems for this model is 50, which was already the value provided in [25] as the best one.

3.2 Decompositions

3.2.1 Iterative refinement for decomposing CSPs/COPs

As stated in the introduction of this chapter, [25] uses an IDDFS with a static variable ordering(with n-ary branching) and a cut-off based on the height of the tree to decompose a CSP/COP into subproblems. This method can be improved to use custom searches, as shown in [27].

IDDFS is a standard choice for CP as it offers the time complexity of Breadth-First-Search while requiring the same amount of memory as a Depth-First-Search [15]. However the cut-off at a given depth has the drawback of offering very little control in order to generate equally difficult subproblems.

This master's thesis proposes the use of an *iterative refinement* approach: starting with an initial subproblem in a priority queue, pop the first element in the queue, split it (heuristically with the branching heuristic), and push the new subproblems into the queue again. This method is presented in Algorithm 2.

Algorithm 2 describes a generic algorithm for generating subproblems: the way the priority queue orders the problem can be specified as well as the branching heuristic. These two parameters should be carefully chosen to favor the generation of equally difficult problems in the queue.

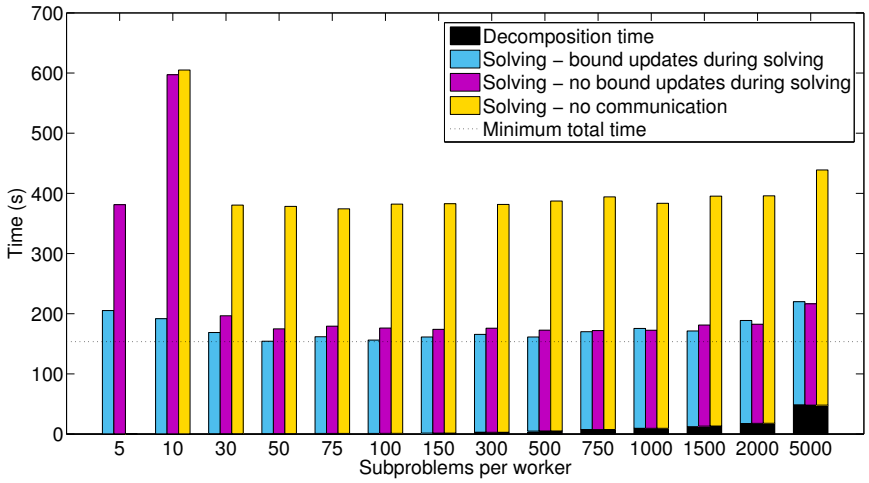


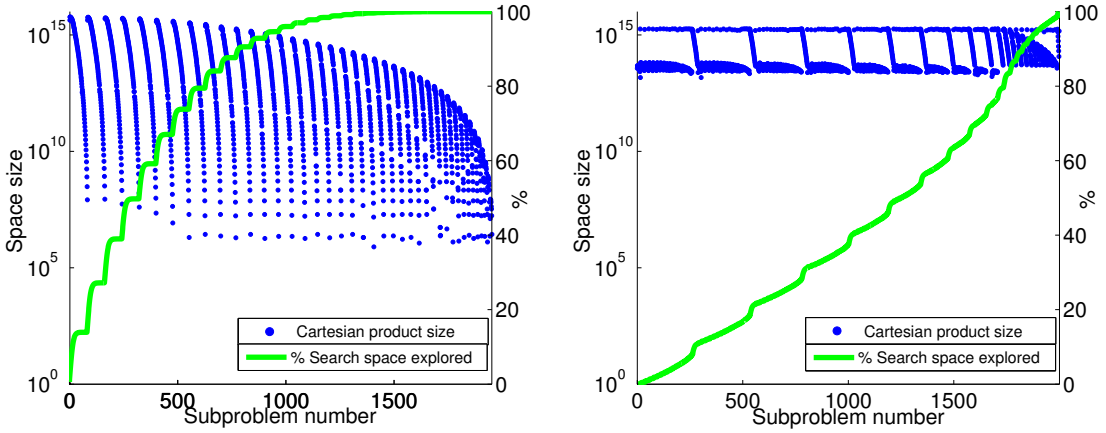
Figure 3.2: Mean solving times (on three runs) of OscaR-Modeling with EPS on a Golomb-ruler model of size 13, with different communication methods and number of subproblems per worker, using 40 workers

Algorithm 2 Iterative refinement for decomposing CSP/COP

Require: P, the initial CSP/COP
  generateChildNodes(s), a heuristic function that returns consistent children of the CSP/COP s
  c, the desired number of subproblems
  $q \leftarrow$ priority queue of subproblems, initially empty
  push P in q
  while size of $q < c$ **do**
    $s \leftarrow$ pop q
    for all $c \in$ generateChildNodes(s) **do**
      push c in q
    end for
  end while
  return content of q, reordered by priority

The performance of IDDFS and algorithm 2 are asymptotically equivalent if you make the assumption that running the fixed-point algorithm that propagates changes due to new constraints runs in constant time w.r.t. the number of constraints added[6]. Also, while IDDFS may do multiple passes on each node of the decomposition tree, iterative refinement only does one pass on each node. In practice, we observed that both IDDFS and iterative refinement obtain the same performances; the refinement approach offers more flexibility, notably because it allows to select precisely the number of subproblems wanted, while IDDFS only offers to set a lower bound.

3.2.2 Domain size oriented decomposition



(a) **Depth-bounded** decomposition with a static **n-ary** search tree

(b) **Cartesian-product** iterative refinement with a static **binary** search tree

Figure 3.3: Comparison of decomposition with different tree forms and methods

As described earlier, an optimal decomposition would generate subproblems requiring the same amount of solving time. Evaluating how much time a CSP/COP would take to be solved is, of course, a very complicated task. An estimator widely used in such cases is the cardinality of the Cartesian product associated to the domain of the problem [24].

The Cartesian product is an estimation of the total search space a given set of variables and domains have. In the context of constraint programming, it is in fact an upper-bound on the effectively visited search space that is only reached when there is no constraint propagation. Its precision then depends on the filtering.

Figure 3.3a shows the Cartesian product size associated with each subproblem created by the IDDFS method from [25]. As can be observed, it is very unbalanced among problems. Furthermore "size patterns" are being formed.

The same behavior was observed on many different searches strategy that were tested on the Golomb-Ruler. The worst case being the binary search with static ordering of the variables: 90% of the estimated search space being assigned to the last 10% of the subproblems. The same behavior is observed on very different CSPs/COPs.

Intuitively, it is expected that a cut-off based on the height of the tree would give such poor balancing. The implicit assumption of this method is that each node on the same level in the search tree has approximately the same importance or size. While this is true on some problems such as the n-Queens, it is not true at all for other problems such as the Golomb.

[6] The priority queue being negligible compared to the fixed point in any practical case

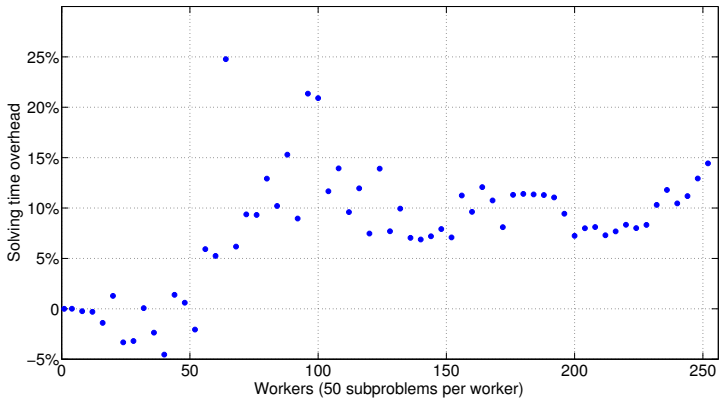


Figure 3.4: Overhead on the solving time of IDDFS with a cut-off based on the height of the tree versus CPIR, on a 16-queens model, with 50 subproblems per worker.

Cartesian Product Iterative Refinement (CPIR), a new method proposed in this master's thesis, is a specialization of algorithm 2, with a priority queue that always returns the subproblem with the greatest Cartesian product cardinality. The goal is intuitive: dividing the biggest subproblem at each iteration should decrease $t_{\max}$, and provide an optimal decomposition w.r.t. the Cartesian product cardinality.

As shown in figure 3.3b decomposing using Cartesian product refinement reduces the importance of the initial search tree shape, and also the magnitude of the patterns (while not removing them completely). As can be observed, the overall search space (evaluated by the Cartesian product) is now more evenly spread among the subproblems. CPIR has a direct influence on the solving time, visible in figure 3.4, which shows the overhead on the solving time of using IDDFS with a cut-off based on tree height, versus CPIR, on a 16-queens model. While the two methods gives mostly equivalent results with a low number of workers/subproblems (due to the fact the first levels of the search tree of an n-queens model are pretty well balanced), the difference between the two methods increases rapidly with a high number of workers/subproblems.

While the usage of n-ary branchings with IDDFS is important (the implicit assumption about balancing of the tree is completely false with binary branchings), CPIR (and algorithm 2 in general) works without any problem with binary branchings or searches with unbalanced branching strategy.

3.3 An Overview of Possibilities Offered by EPS

EPS is relatively new method (less than three years old), and thus some aspects of the possibilities it offers are yet to be explored. In this section, some of them are described.

3.3.1 Running Time Estimation

Most of the time, a computer scientist that writes a CP model and runs it would not be able to know beforehand how much time the solving of the model would take. The idea behind EPS is that increasing the number of subproblems will lead to a greater balance on the solving time taken by each worker. This very same argument can be used to estimate a-priori the total running time.

Some experiments were made in the context of this master's thesis, but were not conclusive or polished enough to be detailed in depth; this subsection only contains an overview.

The law of the law large numbers says that the sample average of i.i.d. random variables tends to the expectation when the number of samples tends to infinity. When running a solving with EPS, multiple subproblems are being solved; it is then possible to gather the running time of the firsts subproblems and to estimate a total running time from them, using the law of large numbers.

More precisely, given n subproblems that have a (unknown) mean resolution time μ, and given the running time of the first m subproblems $T_1, T_2, ..., T_m$, from the law of large numbers we have that

$$\lim_{m,n\to\infty} \bar{T} = \mu$$

And the total estimated running time can then be estimated as $n\bar{T}$ at any moment of the computation.

This approach gives good results on CSPs when there is a very large number of subproblems available, and if the subproblems are balanced enough. A solution to ensure this is to randomize the order of the subproblems, but this is unthinkable with COPs. A refinement of this method is to use tree size evaluation (for example, using the Cartesian product cardinality) to weight the running times. This approach showed very good results in the little amount of tests made, and it is definitely a point that needs further research.

A basic implementation of these heuristics is available in OscaR-Modeling. A screenshot of the tool is available in figure 3.5. A video showing a different (older) version of the tool during solving is available at `https://www.youtube.com/watch?v=OTN4-sjHRhs`.

COPs are a particularly difficult case, as the solving time of a given subproblem is correlated to the location it has in the execution order, due to the shared information on the bound. The simple method using the average then tends to overestimate the final running time. Other methods that take into account this correlation can be used, such as the ones used to analyze time series (ARMA, ARIMA, ...).

3.3.2 Portfolio Search

Parallel portfolio search, that is presented in chapter 2, is a completely orthogonal way to parallelize constraint programming solvers: it can thus be combined with work-stealing or EPS.

Moreover, the numerous subproblems used by EPS can allow to develop new techniques allowing to estimate the effectiveness and compare the searches. Creating a good search heuristic is a difficult task, and can make the difference between a run time that equals to your lifetime or that is measured in seconds. EPS then would enable to select, from a given or generated portfolio of searches which ones are the bests, by using the statistical power given by the number of subproblems.

3.3.3 Best Constraint-Strength Estimation

In CP, most constraints have multiple implementations, depending on the consistency wanted (GAC, FC, ...). Choosing which implementation to use in the model is an important part of the modeling process: stronger consistencies take time to be computed but filter more, while weaker ones are faster to compute and filter less, thus leading to the need to select which constraint implementation would optimize this ratio.

Most of the time, this choice is made by hand, empirically, on smaller models. This selection problem is complicated as there is interaction between the constraints.

Figure 3.5: A screenshot of the tool developed in OscaR-Modeling

Running the subproblems of EPS with random constraint implementations could allow to select in an automated fashion which is the best selection of constraint implementations.

Chapter 4

OscaR-Modeling, a symbolic modeling layer for OscaR

OscaR-Modeling is a new symbolic modeling layer for OscaR, that serves as a basis for implementing EPS in an efficient and elegant way.

This chapter describes the reasons that pushed for creating this new layer, and the solution it provides. The semantic aspects of the new DSL are explored, and the architecture of the source code explained. The implementation of EPS itself is then described. This chapter ends with an overview of the performances.

4.1 Needs & Goals

As most CP solvers, OscaR provides a direct programming interface to its solver; models can be written directly in the host language, Scala. Listing 4.1 shows an example of a model in OscaR-CP.

This modeling language, as shown in the listing, is a full-featured DSL, which uses most of the advanced features of Scala, and helps to write concise and clear models.

However, as it is a direct interface to the solver, the variables are directly *concretized*. For example, when creating `CPIntVar.sparse(0, nQueens - 1)`, the solver allocates an array of size `nQueens` in memory. This memory value is then directly modified even before the solving starts by the concretization of the constraints, leading to "holes" in the domain. This example with domains of size 10 is, of course, a simplification. In practice, domains can contain millions of entries.

In the context of EPS, this *concretization* of the variable make difficult the creation of efficient representation of the subproblems. While storing a domain containing $\{1, 100\}$ can take as low as two integers, storing the same domain but with holes created by the domain requires, most of the time, to write the complete list of possible value.

Moreover, in the current architecture of OscaR-CP, references to the constraints themselves are lost once created; only a callback to given methods they implement is stored by the solver, making difficult to retrieve the original constraint.

An additional problem with the current architecture of OscaR is the way searches and solution managers are handled, via closures. Listing 4.1 shows an example of such a closure, given to the method `onSolution`. For multiple reasons inherent to the way Scala works, it is impossible to send this closure in a serialized way without sending the variables it contains. Since, in OscaR, variables contains a reference to the solver, by default, sending the closure will send the complete solver

```scala
object Queens extends CPModel with App {
  val nQueens = 10
  val Queens = 0 until nQueens

  // Variables
  val queens = Array.fill(nQueens)(CPIntVar.sparse(0, nQueens - 1))

  // Constraints
  add(allDifferent(queens))
  add(allDifferent(Queens.map(i => queens(i) + i)))
  add(allDifferent(Queens.map(i => queens(i) - i)))

  // Search heuristic
  search(binaryFirstFail(queens))

  // Solution manager
  onSolution {
    println("Found a solution")
    println(queens.mkString(" "))
  }

  // Execution
  println(start())
}
```

Listing 4.1: 10-Queens in OscaR

over the network.

For these different reasons, creating a new layer was deemed necessary. The following list shows the goals that OscaR-Modeling should (and do) fulfill:

- Being able to send the model, as well as the search and the solution manager over the network in a simple and concise way.
- Provide at least a DSL as good as the original one in OscaR.
- Can be extended easily, and to access the underlying OscaR solver.
- Provide an implementation of EPS that is usable in parallel or in a distributed environment.

The final implementation goes further, by adding these new features:

- OscaR-Modeling is "solving-type" agnostic. It can be used with OscaR-CP, of course, but is also compatible with other types of solvers such as Integer programming one. The implementation of these other solvers is not present in the final version of the work, but OscaR-Modeling can be easily extended in such a way.
- The new layer is declarative and symbolic.
- It uses the concept of model operators, taken from Objective-CP [31].
- It allows to run pre-solving on the model itself, to simplify it before its concretization.

All these features are described in following sections.

```scala
object NQueens extends DistributedCPApp[Unit] {
  val nQueens = 10
  val Queens = 0 until nQueens

  // Variables
  val queens = Array.fill(nQueens)(IntVar(0, nQueens - 1))

  // Constraints
  post(AllDifferent(queens))
  post(AllDifferent(Queens.map(i => queens(i) + i).toArray))
  post(AllDifferent(Queens.map(i => queens(i) - i).toArray))

  // Search heuristic
  setSearch(Branching.binaryFirstFail(queens))
  setDecompositionStrategy(new CartProdRefinement(queens, Branching.
    binaryFirstFail(queens)))

  // Solution manager
  onSolution {
    println("Found a solution")
    println(queens.mkString(" "))
  }

  // Execution
  println(solve())
}
```

Listing 4.2: 10-Queens in OscaR-Modeling

4.2 Semantic aspects

4.2.1 Models are first-class objects

OscaR-Modeling provides a language that is visually almost equivalent as the one OscaR-CP, but that is *semantically* very different. Listing 4.2 shows an example of model with OscaR-Modeling. The differences with the original code in listing 4.1 are imperceptible.

In the original OscaR-CP DSL, the object you create in your code in not really a model; it is an interface to a solver. Semantically, you add variables to this solver, and add constraints that will restrict the possible values these variables can take. More precisely, you add constraint *implementations*.

OscaR-Modeling uses a different approach, and make a complete dichotomy between the concepts of models, model declarations, and solvers. A *model*, in the sense of OscaR-Modeling, is a pure list of variables, domains and constraints that ties these domains. The domain contained in the models are accessible through the *model declarator*. An example of a model declarator is the `NQueens` object of listing 4.2. The purpose of the model declarator is to put a name on the variable, in order to create searches and solution manager easily. A model declarator is then an interface to all the models it creates.

While the model declarator is a static object, the model themselves can be modified (for example, more constraints can be added to create a subproblem). This concept is linked to the *model tree*, related to *model operators*, which are two concepts that are taken from Objective-CP[31].

A *model operator* is an operator that takes as input a model, and returns a new model. In order to make sure the model declarator continue to represent the new model, a restriction applies

to *model operators*: all the variables that are present in the original model also have to be present in the new one. All the other parameters of the model can change: constraints can be added or withdrawn, new variables added, domains changed.

Successive usage of model operators then leads to the creation of a *model tree*, whose models are all inheriting from the first model, and represented by the same model declarator.

A particular kind of model operator is *concretization operators*. These operators create leaves in the model tree: they concretize models by instantiating a solver. Concretization operators are a powerful abstraction, that can be used to use given models with solvers of any kind. Moreover, it offers new possibilities for hybridization of solvers.

An unconcretized variable (a variable that has not been concretized for a specific solver) is immutable in OscaR-Modeling: all the modifications on the domain must either use a constraint or a model operator. This property can be used to compute *model differences*: the difference between two models can be represented as the set of constraints added, if there is no rewriting of domains by a model operator.

Figure 4.1 shows an overview of these semantic aspects.

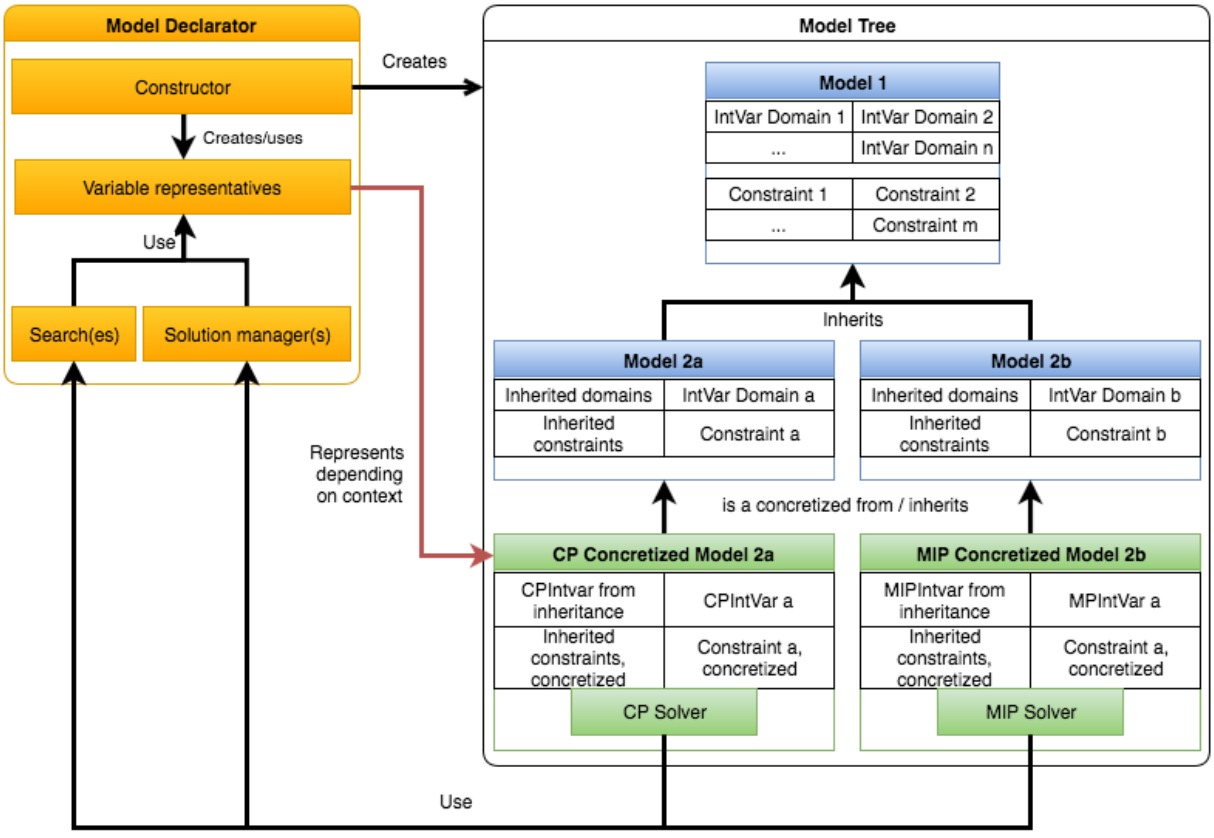


Figure 4.1: Overview of the semantic of OscaR-Modeling, with an example of a model tree.

4.2.2 Expressions

Moreover, and again taking an opposite direction than most CP solvers, OscaR-Modeling has a concept of *expressions*. An expression, as its name indicates, is a combination of variables and operators, such as sum, product or union. This choice makes particularly sense in a "solver-type-agnostic" modeling layer, as some types of solvers, such as MIP ones, mostly works directly with expressions, but this approach also offers new possibilities for CP solvers.

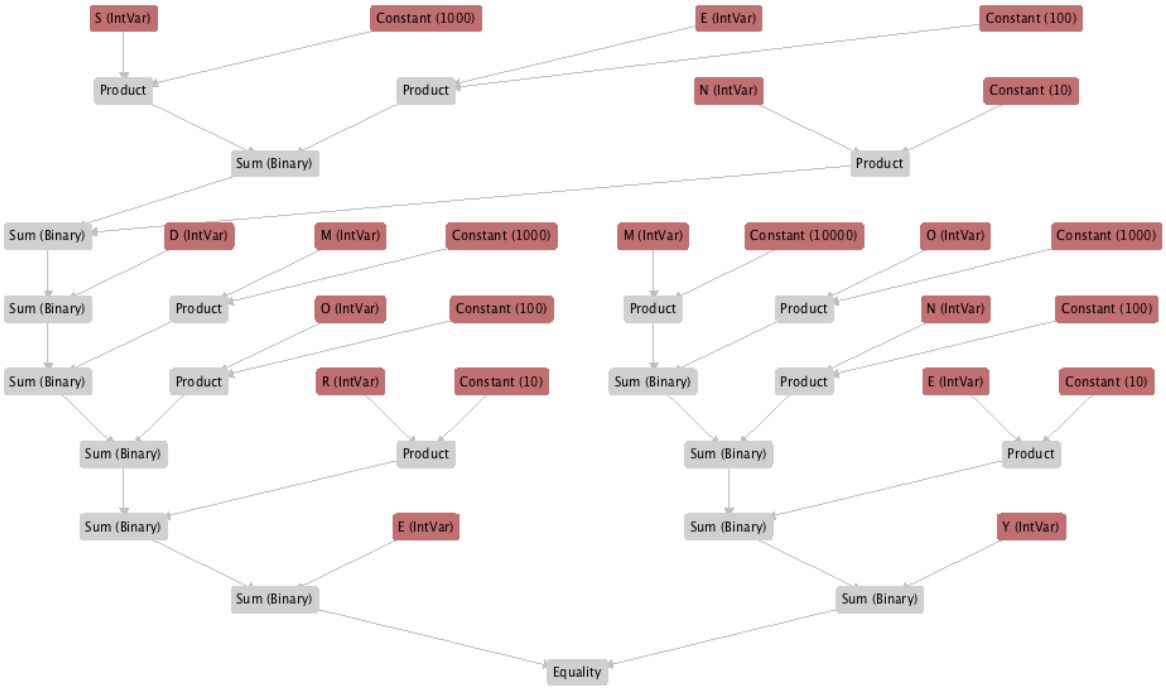


Figure 4.2: AST for the SEND+MORE=MONEY constraint shown in listing 4.3

```
val S = IntVar(0,9)
val E = IntVar(0,9)
val N = IntVar(0,9)
val D = IntVar(0,9)
val M = IntVar(0,9)
val O = IntVar(0,9)
val R = IntVar(0,9)
val Y = IntVar(0,9)
add(S*1000 + E*100 + N*10 + D +
M*1000 + O*100 + R*10 + E ==
M*10000 + O*1000 + N*100 + E*10 + Y)
```

Listing 4.3: SEND+MORE+MONEY main constraint

In OscaR-Modeling, everything is an expression, from variables to constraints, and everything in between. Constraints are *terminal expressions*, meaning that no other expression can use them (they are always at the root of the ASTs): semantically, they are *tautological expressions*. Variables are the operands of the expressions and are then always the leaves of the trees.

Figure 4.2 shows the expressions of a basic SEND+MORE=MONEY constraint (itself shown in listing 4.3) in the form of an AST. Listing 4.4 shows a **simplified** version of the expression grammar of OscaR-Modeling; it only gives a hint about the organization of the expressions, and is not representative of the real grammar (that is managed by the Scala language), which, for example, manages the order of operations and is, of course, unambiguous.

There is then five types of expressions:

- Constraints, a.k.a. terminal expressions
- Boolean expressions (that can have as return value only 0 or 1)
- Integer expressions (that can have as return value any integer)

```
constraint      = boolean_expr | alldifferent | gcc | ...
boolean_expr    = int_expr | bool | eq | geq | gr | leq | lr | ...
bool            = "0" | "1"
eq              = int_expr, "==", int_expr
geq             = int_expr, ">=", int_expr
gr              = int_expr, ">" , int_expr
leq             = int_expr, "<=", int_expr
lr              = int_expr, "<" , int_expr
int_expr        = int | intvar | bool_expr | binary_sum | minus | sum | ...
int             = bool | "2" | "3" | ... | "9"
intvar          = "IntVar", "(", int, int, ")" | "IntVar", "(", int_array, ")"
binary_sum      = int_expr, "+", int_expr
minus           = int_expr, "-", int_expr
sum             = "Sum", "(", int_expr_array, ")"
int_array       = "Array", "(", {int} ,")"
int_expr_array  = "Array", "(", {int_expr}, ")"
alldifferent    = "AllDifferent", "(", int_expr_array, ")"
gcc             = "GCC", "(", int_expr_array, int_array, int_array ")"
...
```

Listing 4.4: Simplified EBNF-like grammar for expressions in OscaR-Modeling

- Variables, which are a particular kind of integer expressions. Variables whose domains contain only 0 or/and 1 are also boolean expressions.
- Integers, which are also a particular kind of integer expressions. 0 and 1 are boolean expressions too.

Some common conversions apply between type of expressions. For example, integers and integer expressions can be converted to boolean expression: the new boolean expression will be true (1) if and only if the integer expression it represents is different from zero. Boolean expressions can themselves be converted to constraint, enforcing that the expression is always true (always resolve to one).

Most of the operators available on constraints (*n*-ary sum, weighted sum, equality, greater than, ...) are, in OscaR-CP, and in most other CP solvers, implemented as a constraint, while in OscaR-Modeling they are implemented as functions in an algebra. The DSL of OscaR-CP provides syntactic sugars to allow using the variable in an algebra-like manner, while creating behind the scenes new variables (and domains associated) and constraints to enforce the relation. OscaR-Modeling proposes a true "algebra": each expression must then be *flattened* into a list of constraints when the model is *concretized* for a CP solver.

Concretizing an expression AST can be made in a straightforward way, by simply reapplying the rules that the original DSL of OscaR-CP uses; however, as OscaR-Modeling has the full information on the expressions used in all the constraints, it is possible to pre-solve part of the model, by making its representation more optimal, as inspired by [16]. Two examples are shown below:

- A trivial example is the SEND+MORE=MONEY problem. Most implementations of the model use the simple constraint

```
S*1000 + E*100 + N*10 + D + M*1000 + O*100 + R*10 + E == M*10000 + O*1000
    + N*100 + E*10 + Y
```

  which is, of course, not optimal:

  1. It uses multiple binary constraints with IntVars multiplied by constant. The usage of the constraint/operator WeightedSum would give a better pruning, and reduce the number of constraints to three.

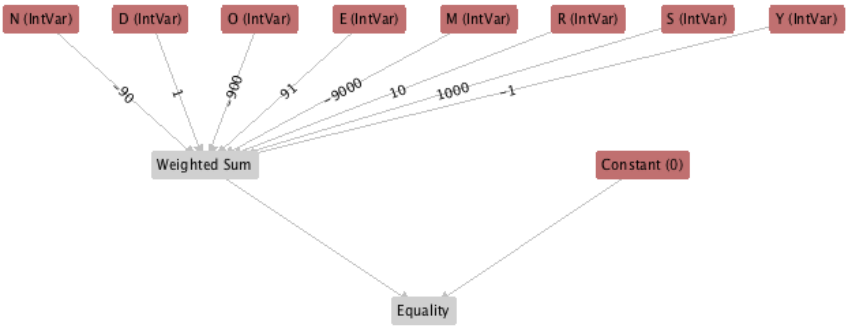


Figure 4.3: AST for the SEND+MORE=MONEY constraint after simplification

```
WeigthedSum(Seq(S,E,N,D,M,O,R), Seq(1000, 101, 10, 1, 1000, 100, 10))
    == WeightedSum(Seq(M, O, N, E, Y), Seq(10000, 1000, 100, 10, 1))
```

2. It is redundant, with M, O, N and E being present in both sides of the equality. Removing these redundancies should improve the efficiency of the filtering.

```
WeigthedSum(Seq(S,E,R,D), Seq(1000, 91, 10, 1)) == WeigthedSum(Seq(M,
    O,N,Y), Seq(9000, 900, 90, 1))
```

3. Further pushing all the operands at the left of the equality reduce the number of constraints to two[1]:

```
WeigthedSum(Seq(S,E,R,D,M,O,N,Y), Seq(1000, 91, 10, 1, -9000, -900,
    -90, -1)) == 0
```

   The corresponding AST is shown in figure 4.3

While this process may seem straightforward to a reader who is used to CP modeling, it can become quite cumbersome on more complicated models, reducing simplicity for efficiency. This kind of pre-solving can be automated: as a proof-of-concept, a model operator called `SimplifySum` is provided in OscaR-Modeling.

- Equivalence between two variables or expressions can be expressed in two ways: the direct usage of the same variable/expression everywhere, or by adding an equality constraint. A misconception often met is that these two methods give the same performances: implementing a GAC constraint for equality requires most of the time an $O(n)$ performance. It is the case in OscaR-CP, which uses trailing, and as a direct consequence unsort the domains in memory, forbidding to use a binary-search for the update (which would still give $O(\log n)$, which is still not equal to the $O(1)$ of using the same variable). The default for equality constraint is then to use bound consistency, reducing filtering power.

  In traditional modeling language, the creator of the model is responsible for taking care of this issue. In some context, using an equality constraint adds to readability, and withdrawing the constraint would reduce maintainability and simplicity of the model.

  In expression-aware modeling languages such as OscaR-Modeling, it is possible to check and merge variables and expressions before concretization in an automated way.

Model operators are then a natural way to implement these pre-solving methods. It is debatable to know if such pre-solving can be seen as an improvement of modeling languages. Some would argue that it is the responsibility of the model writer, and that not knowing what is made "under the hood" by the solver is damageable, while others would say that it simplifies the sometimes

[1] In fact, it reduces it to one, as the pair "WeightedSum ==" is converted to a single WeightSum constraint in OscaR

complicated writing work. These techniques can be also be used only as hints, to guide the model writer to a better model. This debate is out of the context of this master's thesis, and OscaR-Modeling provides simplifications as an opt-in functionality.

The implementation of pre-solving in OscaR-Modeling is proof-of-concept at very early stages, only present to demonstrate the ability of the new modeling language. Further work is then needed to implement and improve techniques presented in [16].

4.3 Architecture

Semantic aspects of OscaR-Modeling have mostly been modeled by design choices for the architecture. As it was thought from the beginning for parallel and distributed usage, OscaR-Modeling often has a radically different structure.

This section describes some of the key points of the implementation.

4.3.1 Model Declarator

The *model declarator* presented in the last section is a construction that allows to reuse the *variable references* in different contexts. In most architectures, a synonym for context is thread[2]; the variable redirection is then implemented as *thread-local* variable, a variable which contents depends on the current thread.

```
class ModelDeclaration extends Serializable {
  private val current_model: DynamicModelVariable = new DynamicModelVariable()
  current_model.value = BaseModel(this)

  /**
   * Get the current model
   */
  def getCurrentModel = current_model.value

  /**
   * Apply the function func, which uses Var declared in this ModelDeclaration,
   * on the model, temporarily changing the current model in the current context (thread).
   *
   * @param model: model on which to apply the function
   * @param func: function to apply
   */
  def apply[RetVal](model: Model)(func: => RetVal): RetVal = ...

  /* ... */
}

class IntVar(model_decl: ModelDeclaration, id: Int) extends Var(model_decl, id) with IntVarLike with IntExpression {
  protected def getRepresentative: IntVarImplem = model_decl.getCurrentModel.getRepresentative(this).asInstanceOf[IntVarImplem]

  /* Proxy to real variables implementations */
  override def isBound: Boolean = getRepresentative.isBound
  override def max: Int = getRepresentative.max
  override def min: Int = getRepresentative.min
```

[2] A significant counter-example being software architectures with "lightweight threads", such as Gevent and Akka

```
29    override def hasValue(value: Int): Boolean = getRepresentative.hasValue(
        value)
30    /* ... */
31  }
```

The redirection is, most of the time, invisible for the model writer, as done when the solvers are concretized in the `solve` functions.

4.3.2 The Model Tree

While *semantically* the usage of model operators creates a *model tree*, the architecture of OscaR-Modeling pushes this concept further, by making it slightly different using constraints. As said earlier, unconcretized variables have immutable domains. A consequence is that, if no model operator rewrites the domains, all models can share the same variable store: the only difference between them is the constraint list.

Models are then immutable objects in OscaR-Modeling, and are composed of a linked list of constraints and of a reference to the variable store, which are also both immutable: each time a new constraint or variable is added, a new model has to be created in the model tree.

- When a new constraint is added, the new model is an exact copy of the old one, but with an additional constraint in the constraint list. In memory, there is still a single copy of the constraints: because constraint lists are immutable, the new item is prepended to the list and points to the original list. Adding new constraint is then nearly free: $O(1)$ in both memory and time complexity. The reference to the variable store remains the same.
- When a new variable is added, the new model is an exact copy of the old one, but with an additional element in the variable list with its domain. The implementation is immutable and uses a Scala `Vector`, which is a Radix-Balanced Tree[30][3], providing $O(\log_m n)$ access time and $O(m \log_m n)$ to append at the end of the tree, with m being a branching factor, generally 32: this structure provides an "*effectively constant access time*". The constraint list remains the same.

This model tree allows to directly compute the difference between a model and one of its descendants. Model operators are allowed to *break* the constraint tree or the variable store, by modifying domains or removing constraints. However, using such model operators have implications on the ability of OscaR-Modeling to compute differences between models.

4.3.3 Network architecture for EPS

As shown in section 3.1, the way the optimization bound is shared is a critical point; the best method is to share it as soon as possible between the workers, to cut the search tree as most as possible.

The network architecture is based on this fact, and is described in figure 4.4. OscaR-Modeling uses Akka, a well-known toolkit for building distributed applications using the *actor model*.

The agent model is a well-known design pattern, involving the usage of multiple *actors* that have each their own task, with no shared state between actors: communication between the agents is made through message-passing. This allows to reduce traditional problems that occur while using shared state, namely race conditions.

[3] [30] proposes a new kind of tree, Relaxed-Radix-Balanced Trees, that further improves RB-Trees.

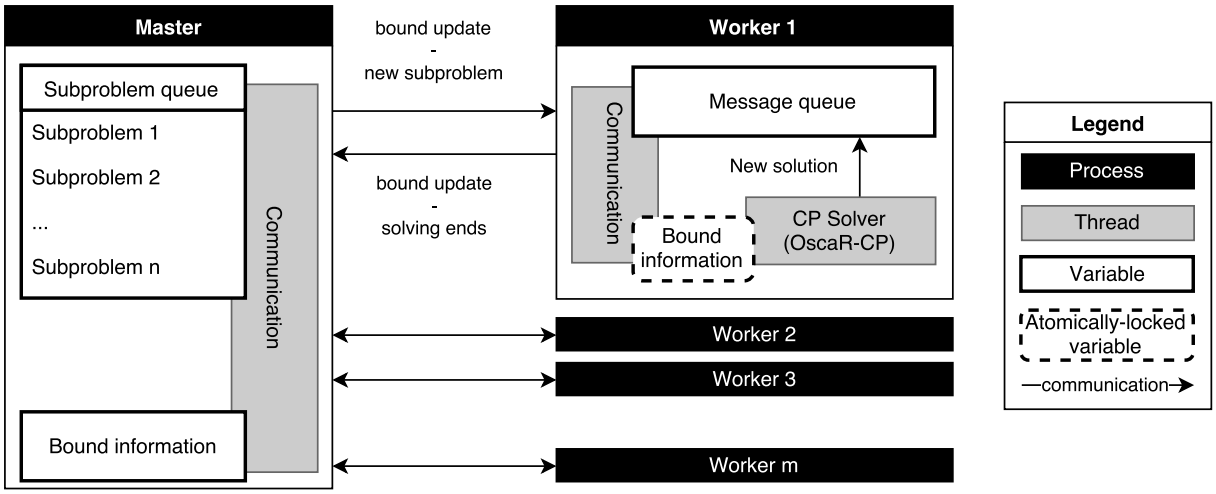


Figure 4.4: Overview of the architecture of OscaR-Modeling with EPS

OscaR-Modeling uses two kinds of actors: the *master* actor, and the *workers*[4]. The master creates and maintains the subproblem queue and the status of the optimization bound. Its behavior is then simple:

- The master has the responsibility to send new subproblems to workers that asks for one.
- The master receives bound updates from workers, and has to verify that each update needs to be dispatched to all the workers or not. Due to the fact that communications can be made over the network, and that computation is made in a parallel fashion, bound may be received in an incoherent order; only the effective improvements of the optimization bound have to be memorized and shared to all the workers.
- At the end of the computation (when all the workers are idle and the subproblem queue is empty), the master has to get the solutions back from the workers.

As an Akka agent is single-threaded, all these actions are taken sequentially, ensuring correct coordination between the workers.

Workers are sightly more complex. A worker is mainly an Akka actor: its role is to ask for subproblems, to resolve them, and to send information about new solutions (and new bounds) to the master actor. OscaR-Modeling being a layer above OscaR-CP, the solving itself is made by a separate thread running an OscaR-CP solver: this allows the worker to receive messages concurrently (not necessarily in a parallel way, as the two threads on the worker runs most of the time on the same CPU, but at least concurrently) while solving. When solving COPs, information on the bound then needs to be shared between the Akka communication thread and the solver thread:

- The bound information needs to be shared. For now, OscaR-Modeling only implements basic minimization and maximization: the optimization bound is then a single integer. As a consequence, using a single Java volatile variable (that implements a happens-before relationship) is sufficient to ensure the thread-safety[5].

  The original search method provided in the model or the program is modified such that, at each branching operation during the search, the optimization bound is checked and updated in the solver (by running the fixed point algorithm), if necessary.

[4] Also called *minions* sometimes.

[5] To be completely precise, the `volatile` keyword only forces the visibility of the modifications between the thread. Operations on `int`s are already atomic in Java and most languages

- New solution (that, when doing optimization, give an additional bound on the optimization goal) have to be shared with the master actor. As the solution are obviously found in the solver thread, this information has to be shared with the communication thread. Akka provides abstraction for doing this transfer in a thread-safe way.

4.3.4 Subproblem Representation and Serialisability

The main form of communication between the master actor and the workers is the transfer of new subproblems. The size of the subproblem representation in the messages must then be limited at the strict minimum, to ensure the message is received as soon as possible and solving starts immediately after, while being descriptive enough to avoid too much unnecessary computations.

A first method to represent subproblems would be to share the domains of the variables. However, the domain representation can be a lot larger than the initial model representation, and restrict how subproblems can be created by only allowing domains to be cut: no additional constraints can be added. This approach is then impractical.

The first implementations of EPS used FlatZinc[20] to share the subproblems[25, 26]: new constraints are added to the original subproblem in the FlatZinc file, the file is then sent to the worker, parsed, and solved. This, of course, causes an overhead, due to re-parsing, unnecessary file length due to the encoding chosen (readable text), and recomputation of the initial fixed-point for the *whole* model.

OscaR-Modeling uses an improvement of this approach. The initial problem, along with its variables, domains, and constraints, is shared with all workers when solving starts. Subproblems are represented as list of constraints, that are themselves computed from the *model tree*: the path from the initial model to the subproblem defines the subproblem completely. With a good serialization, this list of constraint takes a very little amount of memory, and when received, a small fixed-point must be run. This is then a good compromise between size and efficiency.

The serialization is the key point. Constraints in OscaR-Modeling are placeholder objects, containing only the variables and parameters for the constraint: they do not contain other state or logic. Most of them are implemented as a single line of code. Serializing them consists only at giving them a global id, and then serializing the parameters. Variables are the main kind of parameter that are given to constraints (the other being integers), and pose a particular problem.

OscaR-Modeling uses Java/Scala serialization. By default, its involves serialization of the whole context that is needed by the objects. As variables need to access their domain, domains are serialized and transferred by default. In order to alleviate this, `IntVar` are represented as a single integer. These identifiers are shared beforehand when the solving begins. Variables are then serialized as a single 32-bit integer when the subproblems are shared.

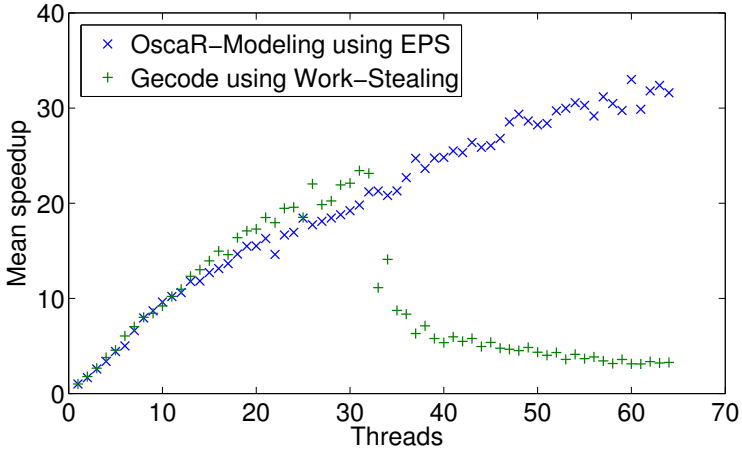
4.4 Performance

This section provides an analysis of the performances of OscaR-Modeling. It focuses on three well-known problems, the N-Queens, the Quadratic Assignment Problem, and the Golomb Ruler.

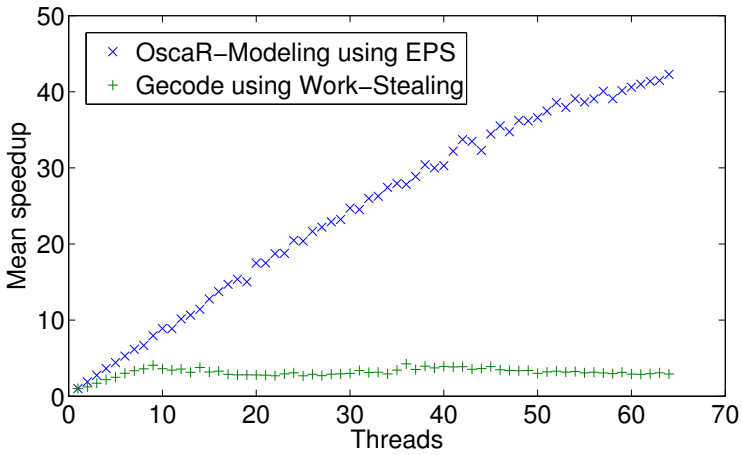
4.4.1 OscaR-Modeling versus Work-Stealing

Work-Stealing is considered as the current state of the art for parallelizing CP solvers, as it is the solution that provided the best results. The authors of [25] shows that it is no longer the case

for most CSPs. Figure 4.5a and 4.5b shows the speedup obtained by using OscaR-Modeling with EPS and CPIR compared to the speedup obtained by Gecode[7] (the reference implementation for Work-Stealing) on a precisely equivalent model and search.



(a) Golomb-ruler of size 13



(b) 16-queens

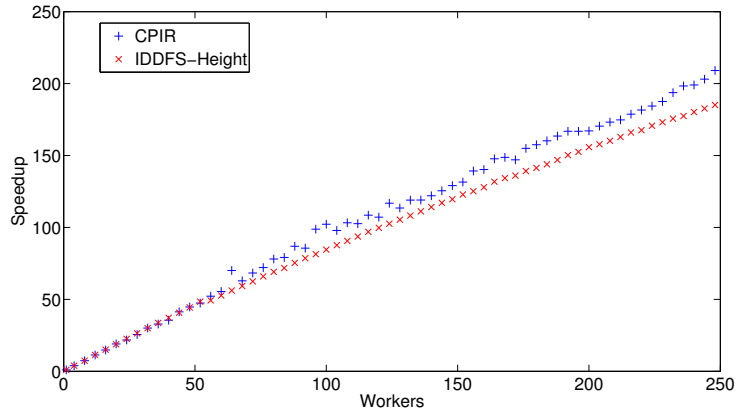
Figure 4.5: Mean speedup (on five runs) comparison between OscaR-Modeling (EPS) and Gecode (Work-stealing). Ran on a single machine with four 16-cores AMD Bulldozer 6272.

Figure 4.5a shows that the work-stealing performs very well until approximately 30 threads, where the speedup suddenly falls to five and continues to decrease. This behavior is more than probably caused by the number of steals needed to make sure all the workers are not idle; having too much workers causes starvation. On the other hand, Oscar-Modeling with EPS continues to scale sub-linearly. The sublinearity of EPS in this architecture is due to the fact this test was run on a single machine, with a very high number of cores: the efficiency of the system itself is not constant, impacting EPS and Gecode.

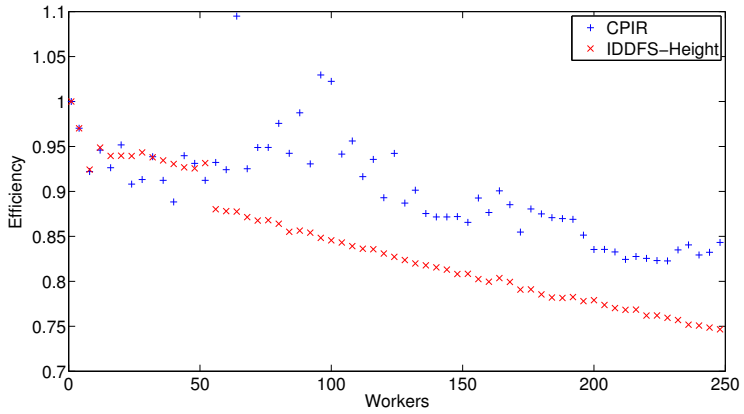
Figure 4.5b shows similar results for EPS. Work-stealing do not perform well; it is difficult to establish a precise cause for this behavior, but an hypothesis would be that the location where the tree is divided is not optimal and leads to a very unbalanced division.

4.4.2 CPIR versus IDDFS with height cut-off

In addition to figure 3.4 (page 26) that shows the overhead caused by IDDFS on the solving time, versus CPIR, figure 4.6 shows the speedup and efficiency of OscaR-Modeling with EPS when solving a 17-Queens model. As can be seen, CPIR scales better, having its efficiency decreasing at a less steady rate than IDDFS.



(a) Speedup



(b) Efficiency

Figure 4.6: Mean speedup and efficiency (on three runs) of OscaR-Modeling with EPS on a 17-Queens model, with two different decomposition methods

Both decompositions lead to sublinear speedups, as it is expected: there is no optimization involved, and the overhead is caused by the decomposition and the communication time. However, it still scales very well, with $\eta \approx 0.85$ at 248 threads (for CPIR).

4.4.3 Distributed Environments - Overhead Analysis

Figure 4.7 shows the speedup, efficiency, and number of nodes visited by OscaR-Modeling with EPS, on a Golomb-ruler of size 13. The experiment is fine-grained, going from 4 to 252 workers(threads) with increments of four. The computations were made on a grid of machines with two 8-cores Intel E5-2650 processors (2.0 GHz).

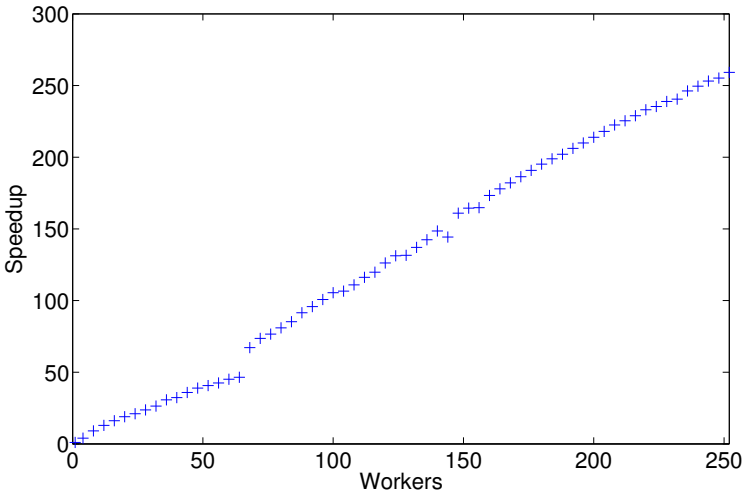
This is a particularly interesting case of super-linear speedup caused by bound communication. On fewer than 60 workers, the number of nodes visited increase gradually due to different factors, mostly because the problem is not well-balanced enough. This leads the efficiency to decrease

gradually. A sudden decrease of the number of nodes visited, increasing the efficiency, then appears at 64 workers, due to bound propagation. The efficiency then remains greater than one and begins to decrease slowly around 170 workers.

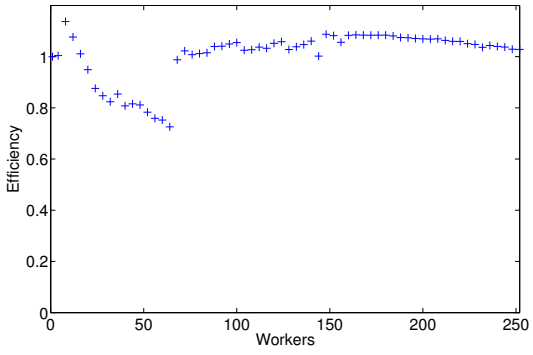
Figure 4.8 shows the results with a QAP [10] model, with a relatively small instance[6]. The results are superlinear when using fewer than 128 threads, but the efficiency decreases rapidly, as the problem is too small compared to the number of threads involved: at 252 threads, the decomposition time takes the third (7 seconds) of the total solving time (22 seconds). It can be seen this behavior can be explained only by this ratio, as the number of nodes explored does not increase.

The influence of communication is then clearly visible, as, without it, the efficiency would have always been lower than one (see corrolary 1, in section 3.1).

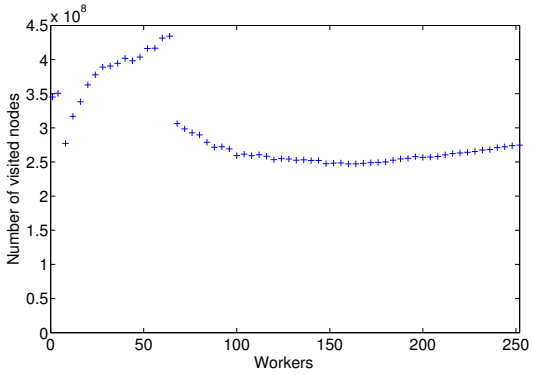
Figure 4.9 shows the results on a bigger instance[7], which has a superlinear speedup on up to, at least, 252 workers, because the problem is big enough and the decomposition/solving ratio is small.



(a) Speedup



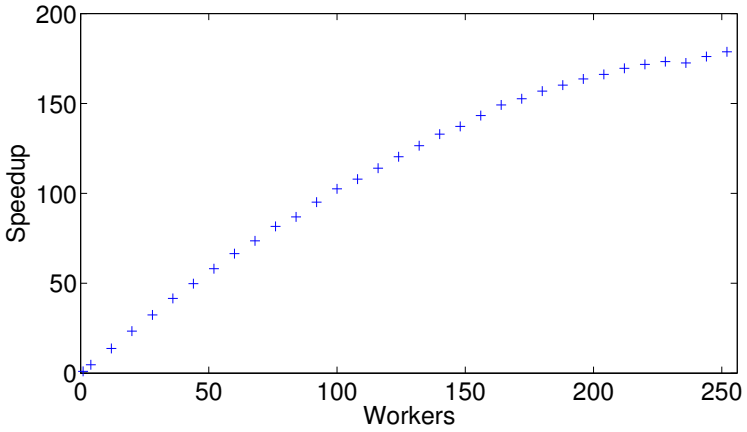
(b) Efficiency



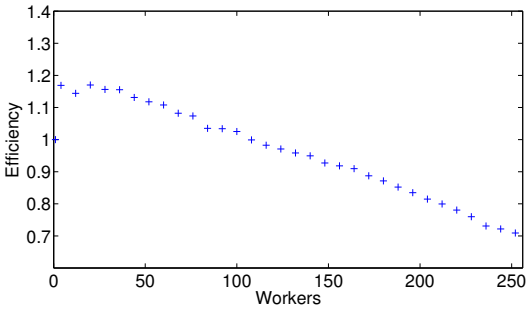
(c) Visited nodes

Figure 4.7: Mean speedup, efficiency and nodes visited (on five runs) of OscaR-Modeling with EPS on a Golomb-ruler model of size 13

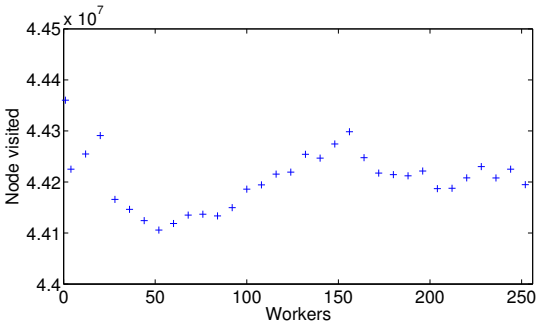
[6] Precisely, it is an instance of size 11, derived from [10]
[7] Size 12, from [10]



(a) Speedup

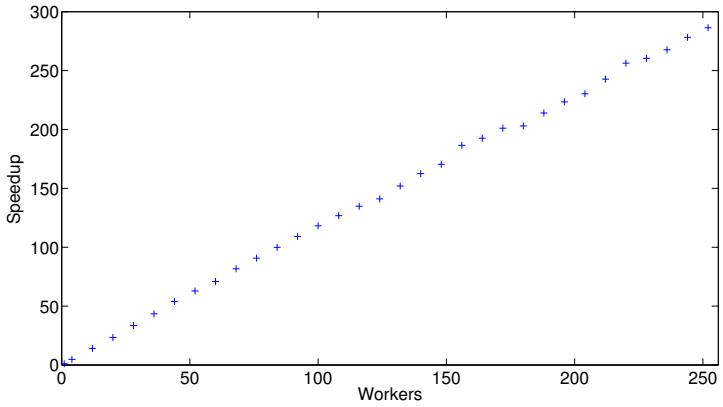


(b) Efficiency

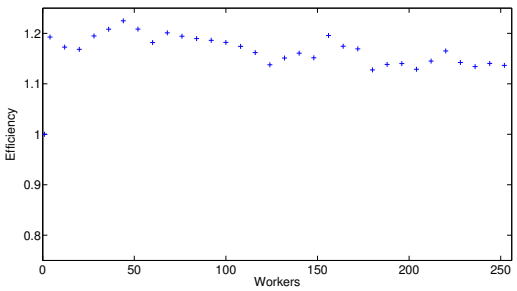


(c) Visited nodes

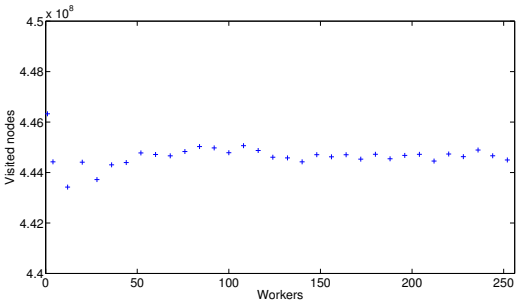
Figure 4.8: Mean speedup, efficiency and nodes visited (on three runs) of OscaR-Modeling with EPS on a QAP model (instance of size 11 derived from [10])



(a) Speedup



(b) Efficiency

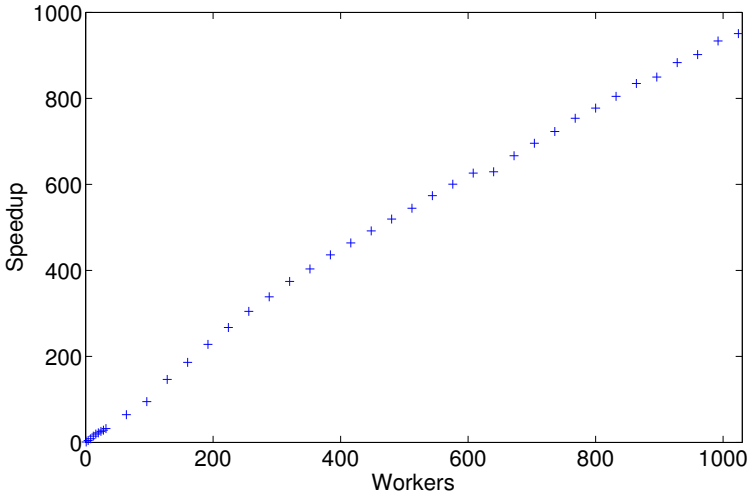


(c) Visited nodes

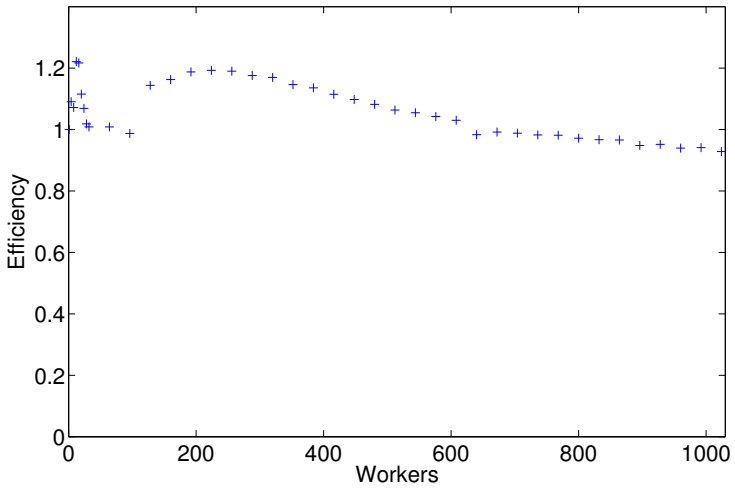
Figure 4.9: Mean speedup, efficiency and nodes visited (on three runs) of OscaR-Modeling with EPS on a QAP model (instance of size 12 from [10])

4.4.4 Heavily distributed environments

Figure 4.10 shows the speedup and efficiency of OscaR-Modeling with EPS and CPIR on up to 1024 workers. These tests were made on the Lemaitre2 cluster hosted at UCL; it is composed from 112 compute nodes with two 6-cores Intel E5649 processors at 2.53 GHz. The results show a nearly linear speedup, even at a thousand cores. The efficiency graphs for the golomb ruler (figure 4.10b) is here a good hint to estimate the quality of the solution, with mostly super-linear speedup before 512 workers; the speedup begins to decrease slowly afterwards, which is expected at such a high number of workers. Tables 4.1 shows an overview of the running times.



(a) Speedup



(b) Efficiency

Figure 4.10: Mean speedup and efficiency(on five runs) of OscaR-Modeling with EPS on a Golomb-ruler model of size 14

4.5 Further work

OscaR-Modeling is a (fully working) proof-of-concept, which aims at showing the power of EPS while being included in a full-featured solver (the other implementations only using the solvers externally). It also became, through implementation and design choice, a laboratory for implementing

Workers	Average solving time on five runs		Speedup	Efficiency
1	1 day 2 hours 57 minutes 49 seconds	(97069s)		
4	6 hours 10 minutes 56 seconds	(22256s)	4.36	1.0904
16	1 hour 23 minutes 6.3 seconds	(4986.3s)	19.46	1.2167
64	25 minutes 4 seconds	(1503.8s)	64.54	1.0086
128	11 minutes 3 seconds	(662.84s)	146.44	1.1441
256	5 minutes 19 seconds	(318.61s)	304.66	1.1901
512	2 minutes 58 seconds	(178.27s)	544.50	1.0635
1024	1 minute 42 seconds	(102.10s)	950.71	0.9284

Table 4.1: Overview of running times for a golomb-ruler of size 14

a symbolic layer and presolving in OscaR-lib. The author of this master's thesis has the conviction it is already usable for some practical usages.

However, further work is needed. A non-exhaustive list of possible improvements follows:

- The DSL of OscaR-Modeling could share most of the DSL of OscaR-CP. There is currently a list of small differences between them that could be harmonized.
- OscaR-Modeling is currently a layer over OscaR-CP, that do not modify a single line of code in OscaR-lib. This comes at the price of duplication of some part of OscaR-CP, with small modifications, mostly on the branching algorithms. The complete integration of OscaR-Modeling in OscaR-lib must get rid of these duplicates. Moreover, the new layer provides a new interface (namely `IntVar`) over real implementations (`CPIntVar`). As the original CPIntVar from OscaR-CP do not respect precisely this implementation (for various reasons), OscaR-Modeling ships it own `CPIntVar` that is another layer on the `CPIntVar` of OscaR-CP. This is an example of small modifications that are needed in OscaR-lib to include OscaR-Modeling.
- OscaR-Modeling does only support integer variables, there is no support for set and graph variables. More work is needed to include them, and a refactor of the way domains are stored will be necessary.
- There is currently no support for other optimization methods than simple minimization or maximization. Supporting multi-optimization requires a modification of the communication between the master and the workers when using EPS.
- There is currently no method to add new constraints to OscaR-Modeling without hacking in the code. Solving this problem is actually very simple, and requires a little refactor of the concretization operators, to displace logic from the operator to the constraints.
- Presolving capabilities are very simplistic in the current implementation. Other works exist[16] and should be added to the implementation.
- There is currently no support for advanced search techniques such as discrepancy search. While this is only a small abstraction to add over the sequential solver, more work is needed to use them over EPS.
- While distributing computation is currently possible with OscaR-Modeling, it is not transparent to the model designer, as the context (the environment) of the closures needs to be sent with them, and thus carefully designed. Currently, this is done using Scala `spores`. It is possible to make this process transparent using macros.

Chapter 5

Conclusion

As shown in the previous chapters, Embarrassingly Parallel Search provides very good results for constraint programming solvers parallelization, providing linear or super-linear speedup on all the scales tested in this master's thesis, even to a thousand threads.

The apparent simplicity of EPS, whose main principle is to decompose an initial constrained problem into numerous subproblems, hides important architectural design choices: the decomposition method used, the way optimization bound and subproblems are shared and the global network architecture have proven in the past chapters their importance.

This master's thesis introduced a new decomposition, CPIR, that is based on an estimation of the tree sizes using the Cartesian product. It provides some gains (approximately 10%) over the original IDDFS method. Using search trees size to decompose problems seems to be a natural choice. Search tree size estimation is a widely explored field: the seminal paper from Knuth [14] is a notable example, and other methods more appropriate for branch-and-bound exists, such as the one presented in [13].

It is arguable to see if using these methods could give a significant speedup, as they involve more computing power and complexity than the simple Cartesian Product estimation, but give more precise estimations: this is definitely a topic that needs further research.

Communication of the optimization bound when solving COPs may seem to be counter-intuitive with the inner principle of *Embarrassingly* Parallel Search. This observation must be nuanced: the communication involved is most of the time very small compared to the one involved in distributing the subproblems, and the speedups reached on some problems using communications are by far higher than without communication. Examples have shown that this even allow to reach super-linear speedups.

Embarrassingly Parallel Search offers many more possibilities. A broad spectrum of techniques has yet to be adapted to EPS and more generally to parallelized solvers. One of the most obvious examples is Large Neighborhood Search (LNS), but other techniques such as hybridization can also benefit from EPS. The statistical power given by the numerous subproblems used also allows to develop new methods for the running time estimation or the evaluation of the best constraint strength, or search heuristics. Synergies with other methods such as the ones used in Big Data (Map-reduce ...) can be explored too.

But EPS is not a magic wand. Its main weakness is the decomposition. Most problems have a very broad search tree, and visiting a node takes microseconds, but there exists models where the tree is comparatively small, but where visiting a node takes tens of milliseconds or even seconds. These problems lead to long decomposition time, as it is mainly a sequential process, which itself reduces the speedup. [27] and [26] proposes some parallel decomposition algorithms that tackle

partly this issue.

There is then a lot of topics that need to be explored. OscaR-Modeling, the new symbolic layer developed during this master's thesis, is a good basis to continue these researches. It is a fully working implementation of EPS, which was the main goal at the start of this work, and can already be used to solve large-scale problems. It provides better performance with EPS than previous implementations, thanks to its architecture. Moreover, the usage of the symbolic layer introduced in Objective-CP provides new possibilities, such as algebraic pre-solving. The integration of OscaR-Modeling directly in OscaR is planned in the near future, but requires more work, mainly to harmonize the APIs.

Further work on OscaR-Modeling will also involve the integration of other modeling languages, such as XCSP3, to be able to tackle the main limitation of this master's thesis: the lack of adapted models and benchmarks. While benchmarks for solvers exist, notably the MiniZinc Benchmark Suite, they do not provide problems big enough to need scaling on a thousand cores. Moreover, implementing MiniZinc/FlatZinc in OscaR-Modeling would have needed to much time. The focus was then put on a few models, such as QAP, the Golomb-Ruler, the N-Queens, and the BACP, to show detailed statistics on the evolution of the efficiency with the number of threads and multiple factors, which was not explored in depth in other works.

Research on EPS and development of OscaR-Modeling will be continued in the near future, thanks to the FSR grant received at the time these lines are written.

Bibliography

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[3] Roman Bartak. Constraint programming: In pursuit of the holy grail. In *in Proceedings of WDS99 (invited lecture*, 1999.

[4] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 443–448, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[5] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. *Principles and Practice of Constraint Programming - CP 2009: 15th International Conference, CP 2009 Lisbon, Portugal, September 20-24, 2009 Proceedings*, chapter Confidence-Based Work Stealing in Parallel Constraint Programming, pages 226–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[6] Guillaume Derval. Parallelization of constraint programming using embarrassingly parallel search. Master's thesis, EPL/INGI, Université Catholique de Louvain, 2016.

[7] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[8] Ian P Gent, Chris Jefferson, Ian Miguel, NC Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. A preliminary review of literature on parallel constraint solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, 2011.

[9] Google. *Or-Tools*, 2015.

[10] SW Hadley, Franz Rendl, and Henry Wolkowicz. A new lower bound via projection for the quadratic assignment problem. *Mathematics of Operations Research*, 17(3):727–739, 1992.

[11] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2008.

[12] J. Jaffar, A. E. Santosa, R. H. C. Yap, and K. Q. Zhu. Scalable distributed depth-first search with greedy work stealing. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 98–103, Nov 2004.

[13] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *proceedings of the 21st national conference on Artificial intelligence-Volume 2*, pages 1014–1019. AAAI Press, 2006.

[14] Donald E Knuth. Estimating the efficiency of backtrack programs. *Mathematics of computation*, 29(129):122–136, 1975.

[15] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[16] Kevin Leo and Guido Tack. Multi-pass high-level presolving. In *International Joint Conference on Artificial Intelligence*, 2015.

[17] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3):363–382, 2009.

[18] Laurent Michel and Pascal Van Hentenryck. A microkernel architecture for constraint programming. *arXiv preprint arXiv:1401.5334*, 2014.

[19] Jean-Noël Monette, Pierre Schaus, Stéphane Zampelli, Yves Deville, Pierre Dupont, et al. A cp approach to the balanced academic curriculum problem. In *Seventh International Workshop on Symmetry and Constraint Satisfaction Problems*, volume 7. Citeseer, 2007.

[20] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, CP'07, pages 529–543, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[22] Pierre Schaus. CP for the impatient, 2015. Available from `http://info.ucl.ac.be/ pschaus`.

[23] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2015.

[24] Philippe Refalo. *Principles and Practice of Constraint Programming – CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27 -October 1, 2004. Proceedings*, chapter Impact-Based Search Strategies for Constraint Programming, pages 557–571. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[25] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *Principles and Practice of Constraint Programming*, pages 596–610. Springer, 2013.

[26] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Improvement of the embarrassingly parallel search for data centers. In *Principles and Practice of Constraint Programming*, pages 622–635. Springer International Publishing, 2014.

[27] Mohamed Rezgui. *Parallélisme en programmation par contraintes*. PhD thesis, Nice, 2015.

[28] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[29] Pierre Schaus, Pascal Van Hentenryck, Jean-Noël Monette, Carleton Coffrin, Laurent Michel, and Yves Deville. Solving steel mill slab problems with constraint-based techniques: Cp, lns, and cbls. *Constraints*, 16(2):125–147, 2011.

[30] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. Rrb vector: a practical general purpose immutable sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 342–354. ACM, 2015.

[31] Pascal Van Hentenryck and Laurent Michel. *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, chapter The Objective-CP Optimization System, pages 8–29. Springer Berlin

Heidelberg, Berlin, Heidelberg, 2013.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**