Embarassingly Parallel Search Reengineered

Guillaume Derval[1], Pierre Schaus[1], and Jean-Charles Régin[2].

[1] INGI, Université Catholique de Louvain
Place de l'Université, 1, 1348 Louvain-la-Neuve, Belgium
`guillaume.derval@student.uclouvain.be`
`pierre.schaus@uclouvain.be`
[2] Universite Nice-Sophia Antipolis, I3S UMR 6070, CNRS, France
`jcregin@gmail.com`

Abstract. With the current cloud trend, parallelizing applications has become more and more important, and the constraint solvers will have to take profit from this massively parallel computing power now available. Embarrassingly Parallel Search is one of the most promising methods, but some of its aspects have to be explored further. This paper presents improvements on the architecture and methods used by EPS, such as decomposition strategies, bound information sharing, and subproblem representation. An open-source implementation of EPS, built on top of OscaR is also presented.

Keywords: Parallel Search, Constraint Programming

1 Introduction & background

Parallelism in Constraint Programming is not a new topic. Work Stealing [8,2,6] is one of the most used techniques for CP. The main idea behind Work Stealing is to use multiple CP solvers; each time a CP solver becomes idle, it asks to another solver a part of its current work. This method performs well for a moderate size of workers, but with a very high number of cores, the number of "steals" become too high inducing a significant communication overhead. Another method, parallel portfolio search [1], involves solving the same model with multiple searches in parallel. The results are very problem dependent, and it is difficult to create sufficiently different searches to scale on thousands of CPUs.

Embarrassingly Parallel Search[12] (EPS) is a promising technique that aims at solving CP problems on massively parallel or distributed environments. Its principle is simple: EPS decomposes an initial CSP/COP into many subproblems, that are then solved independently by multiple workers. The solving part itself can be implemented in two different ways:

- Static sharing: subproblems to solve are shared a-priori between the workers.
- Dynamic sharing: subproblems are 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 of each worker would differ by only a tiny fraction.

Theorem 1. *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. □

Theorem 1 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[3]. Of course, the number of subproblems to generate is a tradeoff since generating subproblem and sharing them among the worker also takes time.

Apart from these statistical considerations, decomposing a CSP/COP is not a trivial task: the algorithm used to decompose, and the way the subproblems are stored can have a very important impact on the solving speed-up, as shown in next theorem:

Theorem 2. *(from [12]) 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 [12] the authors assumes that when the number of subproblems increases, as a consequence, the value of $t_{\max}$ should also decrease. While this is a reasonable assumption, the theorems 1 and 2 also tell us that the results of EPS can be improved using decomposition strategies that minimize $t_{\max}$.

In [12] the authors propose a decomposition strategy that only generates subproblems consistent with propagation, that are problems not proved inconsistent by the fixed-point algorithm. This allows to avoid considering numerous inconsistent subproblems: the authors show that the ratio between consistent and inconsistent subproblems can be very high on some problems.

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 a 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 [12] 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:

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

– The decomposition: The objective is to design a decomposition such that all the subproblems are roughly equally difficult to solve. This not only allows to minimize $t_{\max}$, but 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 treated 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. Problem ordering has no impact for CSPs (Régin et al. even suggested to randomize it), but it has a crucial impact on the efficiency for solving COPs.

Our contribution is twofold. First we introduce new decomposition strategies that improve the results of EPS, both when solving CSPs than COPs. Second we introduce our architecture in a CP Solver to efficiently store and distribute subproblems.

2 Iterative Refinement for Decomposing CSPs/COPs

As stated in the introduction, [12] 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 [14].

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 [7]. However the cut-off at a given depth has the drawback of offering very little control in order to generate equally difficult sub-problems. Instead we propose to use 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 1.

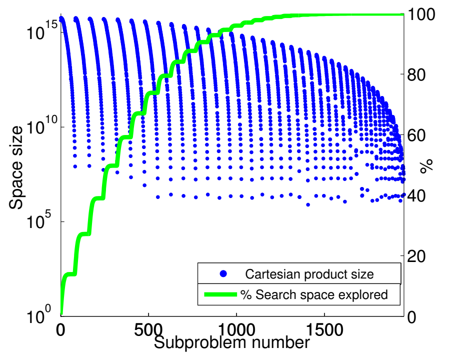
Algorithm 1 Iterative refinement for decomposing CSP/COP

```
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 $P$ to $q$
  while size of $q < c$ do
      $s \leftarrow$ pop $q$
      for all $c \in$ generateChildNodes($s$) do
          push $c$ to $q$
      end for
  end while
  return content of $q$, reordered by priority
```

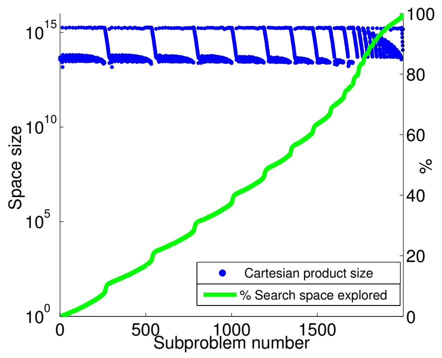
Algorithm 1 describes a generic algorithm for generating subproblems: the way the priority queue orders the problem can be specified as well as the branching heuristic. These two parameters should be carefully chosen to favor the generation of equally difficult problems in the queue.

The performance of IDDFS and algorithm 1 are asymptotically equivalent under the assumption that running the fixed-point algorithm after the addition of one additional constraint takes a constant time w.r.t. the number of constraints added so far. 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 but the refinement approach offers more flexibility.

3 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

Fig. 1: Comparison of decomposition with different tree forms and methods

An optimal decomposition should generate equally difficult subproblems. Unfortunately evaluating how much time a CSP/COP would require to be solved is difficult. A widely used estimator of the size of the search tree is the Cartesian product of the domains [11].

Figure 1a shows the Cartesian product associated with each subproblem created by the IDDFS [12]. As can be observed, it is very unbalanced among problems. Furthermore undesirable "size patterns" appears. We observed the same behavior with different searches strategy on the Golomb-Ruler. For instance, we observed that a binary search with a static ordering of the variables resulted in 90% of the estimated search space being assigned to the first 50% subproblems.

We propose to use instead a Cartesian Product Iterative Refinement (CPIR). This is algorithm 1 instantiated with a priority queue that always returns the subproblem with the greatest cardinality. As shown in figure 1 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. Another advantage of CPIR over IDDFS is that it behaves better when combined with a binary branching. The problem is that binary search trees are generally unbalanced so cutting at a given depth also results in unbalanced subproblems.

4 Architecture of an EPS implementation

An implementation of EPS needs to be carefully designed in order to take full advantage of it. The original implementation [13] is based on FlatZinc [9] and is using, underneath, compatible solvers such as Gecode[4] or Or-tools [5]. The architecture is pretty simple:

1. A *master* uses the IDDFS decomposition presented earlier to decompose an initial MiniZinc/FlatZinc model into multiple new FlatZinc models, to which constraints have been added.
2. These FlatZinc models are then sent to the multiple workers, either a-priori or using a queue, and then solved independently. When solving COPs, the queue is preferably used, and the current bound on the main model is sent with the subproblem.
3. Once all these subproblems have been solved, the solutions are fetched from the workers.

Although this architecture allows to easily change the used solver thanks to FlatZinc, it presents two weaknesses:

– Using FlatZinc to represent subproblems leads to overhead,
– The new bound information is not sent to "running" workers, thus missing potential filtering.

We describe the architecture of the implementation of EPS in OscaR-Modeling, a symbolic layer for OscaR-CP[10]. OscaR-Modeling reuses many of the ideas present in Objective-CP [15], such as model concretization and model operators that makes it a perfect building block to rely on for implementing EPS. Moreover, OscaR-Modeling is very similar to the existing DSL (Domain-Specific Language) currently used to model with OscaR-CP.

The important ingredients of a successful EPS architecture are 1) the way subproblems are represented, 2) how they are transferred to the workers, and 3) how the optimization bound is updated when solving COPs.

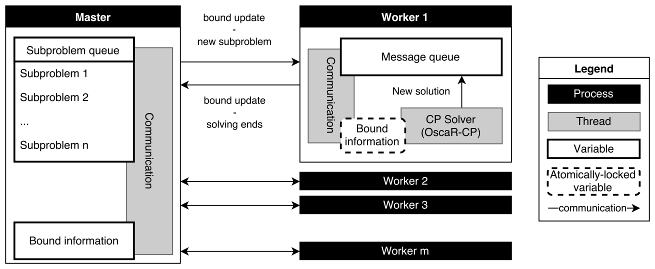


Fig. 2: Overview of the architecture of OscaR-Modeling with EPS

4.1 Global architecture

The architecture of OscaR-Modeling when using EPS is presented in figure 2. Each worker is composed of two threads. The first thread is a running instance of a CP solver (namely Oscar-CP), while the second is in charge of the communication between the solver and the master agent. The current bound on the objective is shared between them. This state is atomically locked to ensure thread-safety. The search/branching strategy given to OscaR-CP checks, each time it is invoked (at each node of the tree), if the value of the bound has changed. Each time the CP solver finds a new solution while solving a subproblem, depending on the type of CP model, the solution, will be either stored locally (for CSPs) or directly sent, with the new objective bound, to the master agent (for COPs). On the reception of this new bound, the master agent will bounce the message to all the workers. This direct communication of the bound ensures each worker (and by extension, each CP solver) has, at any time, the best possible bound.

The shared state and the real-time communication of the bound add a slight but limited overhead to EPS that is largely compensated by the additional filtering obtained with the refreshing of the bound, as shown in figure 3.

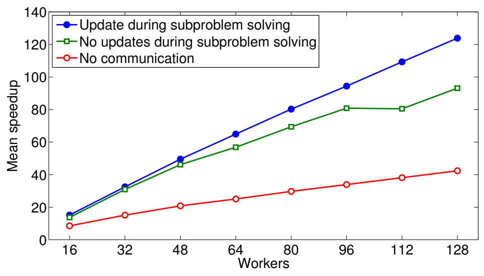


Fig. 3: Mean speedup (on three runs) of OscaR-Modeling with EPS on a Golomb-ruler model of size 13, with different communication methods

4.2 Representing subproblems

OscaR-Modeling is a *symbolic* layer on top of OscaR-CP. The representation of the models simply consists of a list of variable, with domains (range or sets), and a list of constraints with scopes defined on these variables. This consumes very little memory and can easily and lightly be serialized to be transferred to the workers.

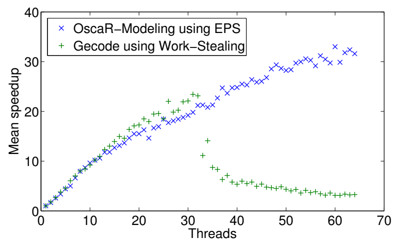
Subproblems are represented as a list of branching constraints leading to the corresponding subproblem node. This representation requires to re-compute the fix-point of the node when a worker starts to process a subproblem.

An alternative representation would be to store the difference of the domains between the subproblem and the initial mode. However this representation is often more costly (memory-wise) as DFS search trees are generally not very deep as compared to the potential difference of the domains.

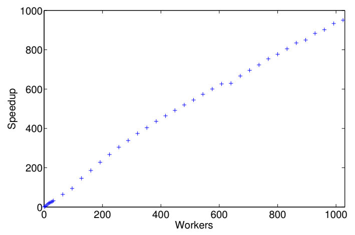
4.3 Example and Performance

Code listings 1.1 and 1.2 compare the difference between a standard single-threaded n-Queens model on OscaR-CP and the same model whose solving can be parallelized and distributed with EPS (on OscaR-Modeling). As can be observed they are mostly cosmetic making it almost transparent for any user to use EPS. The main difference is that one branching is given for both the decomposition and for the subproblem searches. Also the decomposition strategy is parametric.

Figure 4a compares the speed-up between OscaR-Modeling with EPS and Gecode using Work-Stealing. These figures show a nearly linear speedup on these two problems with EPS, while work-stealing fails to continue to improve above 30 threads due to the communications involved. Figure 4b shows the speed-up of OscaR-Modeling with EPS on a Golomb ruler model when distributed over more than one thousand threads, which is clearly linear.



(a) Size 13, comparison with Gecode/Work-Stealing



(b) Size 14, up to 1024 threads

Fig. 4: Mean speedup (on five runs) on a Golomb-ruler model.

```
object Queens extends CPModel with App {
  val nQueens = 10
  val Queens = 0 until nQueens
  // Variables
  val queens = Array.fill(nQueens)(CPIntVar(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))
  // Execution
  println(start())
}
```

Listing 1.1: 10-Queens in OscaR

```
object NQueens extends DistributedCPApp[Unit] {
  val nQueens = 16
  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)))
  post(allDifferent(Queens.map(i => queens(i) - i)))
  // Search heuristic
  val ff = Branching.binaryFirstFail(queens)
  setSearch(ff)
  // Search used for decomposition and method used
  decompositionStrategy(new CartProdRefinement(queens, ff))
  // Execution
  println(solve())
}
```

Listing 1.2: 10-Queens in OscaR-Modeling

5 Conclusion

EPS is a powerful and non-intrusive technique to parallelize constraint solvers that can lead to linear or super-linear speedups with carefully designed implementations. The key points are the problem decomposition strategy, the way bound information is shared, and the representation of subproblems. Decomposition strategies should tend to minimize the solving time of the most difficult subproblem. This article presents a new method, Cartesian Product Iterative Refinement (CPIR), which aims at reducing the Cartesian product cardinality of the subproblems. While the Cartesian product seems to provide a good estimation of the search tree size, the CPIR decomposition should be tested on a larger set of benchmarks.

The importance of the decomposition for EPS is at least as important as the search itself when using linear search. To emphasize this we rephrase the well-known CP mantra for EPS:

$$\text{CP with EPS} = \text{Model} + \text{Search} + \text{Decomposition}$$

More information and details about the subjects seen in this article can be found in the associated master's thesis[3].

References

1. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence. pp. 443–448. IJCAI'09, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009), http://dl.acm.org/citation.cfm?id=1661445.1661516
2. Chu, G., Schulte, C., Stuckey, P.J.: Principles and Practice of Constraint Programming - CP 2009: 15th International Conference, CP 2009 Lisbon, Portugal, September 20-24, 2009 Proceedings, chap. Confidence-Based Work Stealing in Parallel Constraint Programming, pp. 226–241. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04244-7_20
3. Derval, G.: Parallelization of Constraint Programming using Embarrassingly Parallel Search. Master's thesis, EPL/INGI, Université Catholique de Louvain (2016)
4. Gecode Team: Gecode: Generic constraint development environment (2006), available from http://www.gecode.org
5. Google: Or-Tools (2015), https://developers.google.com/optimization/
6. Jaffar, J., Santosa, A.E., Yap, R.H.C., Zhu, K.Q.: Scalable distributed depth-first search with greedy work stealing. In: Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on. pp. 98–103 (Nov 2004)
7. Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence 27, 97–109 (1985)
8. Michel, L., See, A., Hentenryck, P.V.: Transparent parallelization of constraint programming. INFORMS Journal on Computing 21(3), 363–382 (2009), http://dx.doi.org/10.1287/ijoc.1080.0313
9. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. pp. 529–543. CP'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1771668.1771709
10. OscaR Team: OscaR: Scala in OR (2012), available from https://bitbucket.org/oscarlib/oscar
11. Refalo, P.: Principles and Practice of Constraint Programming – CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27 -October 1, 2004. Proceedings, chap. Impact-Based Search Strategies for Constraint Programming, pp. 557–571. Springer Berlin Heidelberg, Berlin, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-30201-8_41
12. Régin, J.C., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Principles and Practice of Constraint Programming. pp. 596–610. Springer (2013)
13. Régin, J.C., Rezgui, M., Malapert, A.: Improvement of the embarrassingly parallel search for data centers. In: Principles and Practice of Constraint Programming, pp. 622–635. Springer International Publishing (2014)
14. Rezgui, M.: Parallélisme en programmation par contraintes. Ph.D. thesis, Nice (2015), http://www.theses.fr/2015NICE4040/document
15. Van Hentenryck, P., Michel, L.: Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings, chap. The Objective-CP Optimization System, pp. 8–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40627-0_5