Proceedings CPAIOR'02

Practical Parallelism in Constraint Programming

Laurent Perron
ILOG SA
9 rue de Verdun
94253 Gentilly Cedex, France
email: lperron@ilog.fr

Abstract

This article presents the qualitative and quantitative results of experiments applying parallelism to constraint programming. We do not set out to prove that parallel constraint programming is the perfect problem-solving tool, but rather to characterize what can be expected from this combination and to discover when using parallelism is productive. In these experiments, we do not deal with scalability involving large numbers of processors; nor do we try to solve extremely difficult or large problems by using brute force algorithms running for days, weeks or even longer. Instead, we concentrate on parallelism involving one to four processors and we restrict our solving time to small durations (around 15 minutes in general). This protocol is applied to different series of problems. The results are analyzed in order to gain a better understanding of the practical benefits of parallel constraint programming.

1 Introduction

In the related fields of constraint programming, combinatorial optimization and operational research, parallelism is often seen as something exceptional. While the operations research[8, 11, 3] and Parallel Prolog[5, 7, 6, 4, 1] communities have long used parallelism as a problem-solving tool, parallelism has only been recently introduced in the constraint programming community[12, 10][1]. The difficulty involved in the implementation of parallelism envelops it in an air of mystery. Few experiments are made on parallel constraint programming systems. Also, parallelism can lead to inflated expectations of a "magic" parallel solution that will overcome all flaws in the modeling and heuristic search parts of a problem. The mere notion of a brute force search involving large numbers of processors (32, 64, or 128 on shared memory computers, and thousands on clusters), creates expectations of a world where difficult problems are solved and open problems are closed. This notion is reinforced by the existence of controversial super-linear speedups, often found when solving optimization problems.[2] The concept of super-linear speedups rein-

[1] Some early experiments with CHIP and PEPSys are exceptions to this statement.[9]

[2] Optimization is not mandatory for observing these so-called super-linear speedups.

forces these elevated expectations of the problem-solving capacities of parallelism, leading one to expect gains of orders of magnitude in the solving time.

Of course, all the mystery and expectations surrounding parallelism are, at the very least, naive. Thankfully, they are not shared by a majority of people in the constraint programming community. A NP problem remains a NP problem, even in parallel. Additionally, the high cost of a parallel computer involving a large number of processors is such that experiments on these computers are rare. On the other side of the parallelism story, computers with two and four processors, and, soon, eight processors are becoming cheaper everyday since they function well as small network, print, mail, and file servers. Thus, even in the shared memory world, parallelism can be characterized as a diverse subject, ranging from inexpensive two-processor and four-processor boxes to highly expensive 64-processor boxes. It should be noted that this range of price corresponds to a range of implementation problems as dead-lock and synchronization issues become more acute with a large number of processors.

With the availability of industrial parallel constraint programming solvers[13], we can finally lift the veil of mystery surrounding the application of parallelism on problem solving and produce some hard results. In this article, we will try to achieve this objective. We will take two different problems: an academic problem, the Golomb Ruler problem; and a real-world network planning problem. Both of these problems have different instances. Some are trivial to solve, while others are very difficult to solve to optimality. One instance is still open. With these experiments, we will analyze two aspects of parallel problem solving. First, we will analyze the effective speedup in solving time, or the effective improvement when solving in a limited time frame. Secondly, we will analyze the variation of these results. In fact, we would like to experiment with the random results we obtain when using parallelism. We will try to come up with both hard results and an intuitive understanding of the differences in performance improvements gained by using parallelism. These differences will then be examined in more detail by changing the nature of the problem.

2 Study of the Golomb Ruler Problem

We begin our study with a well-known pure problem in constraint programming, the Golomb Ruler problem. The description is quite simple. Given an array of size *n* of integer variables, we want to find an assignment where (1) the integer variables are ordered, (2) the value of the last is minimal, and (3) all the differences between the variables are different from each other. Thus it is an optimization problem where the objective is the value of the last variable in the array.

While this problem has no practical application[3], it is a good place to start our investigations into parallel constraint programming. The Golomb Ruler problem has the following features: (a) it is scalable, (b) it can easily become very difficult, (c) it has no practical "killer" solutions, and (d) it has a quite substantial proving part. This allows us to try to solve the problem to optimality or to find the best solution in a limited time on a series of problems ranging from very easy to very difficult.

In practice, we will study Golomb 10, 11, and 12 (Golomb n means Golomb with an array of n variables). We should note that Golomb 10 is solved in 30s with our sequential implementation, Golomb 11 is solver in 650s while Golomb 12 is solver in 6000s. We will use two different models, one better than the other since it propagates more effectively. We will also examine the best solutions found in a short amount of time, or in an amount of time equivalent to the time

[3]However, this problem does comes from a real-world situation involving interference among radar signals.

needed to find the best solution. Finally, we will examine the time needed to prove these optimal solutions. Our implementation just states the above constraints and uses a range propagation on the AllDiff constraint.ls

2.1 Simple Scalability

We begin our experiments with Golomb 10, 11 and 12 and a time limit which allows the best algorithms to find the optimal solution. For the record, all experiments are conducted on a quad-processor, pentium-pro computer running Linux. All use ILOG Parallel Solver 5.2[13].

We run the following experiments: Golomb 10 with a 20 second time frame (figure 1), Golomb 11 with a 30 second time frame (figure 2), and Golomb 12 with a 4500 second time frame (figure 3).

We will present all our results for both the Golomb Ruler and network design problems in the same way. A given problem with a given time frame will be displayed in four boxes, each representing solving this problem with one to four processors. The top left stands for 1 thread, top right for 2, bottom left for 3 and bottom right for 4. We will display 10 different runs for the Golomb Ruler problem and three different runs for the network design problem. All the results of the Golomb Ruler problem are displayed such that the curve hitting the x-axis is equivalent to finding the optimal solution[4]. All four boxes are comparable, meaning the x-range and the y-range are equal. A curve stands for one run. And finally, the x-axis represents time in seconds, while the y-axis represents the objective value.

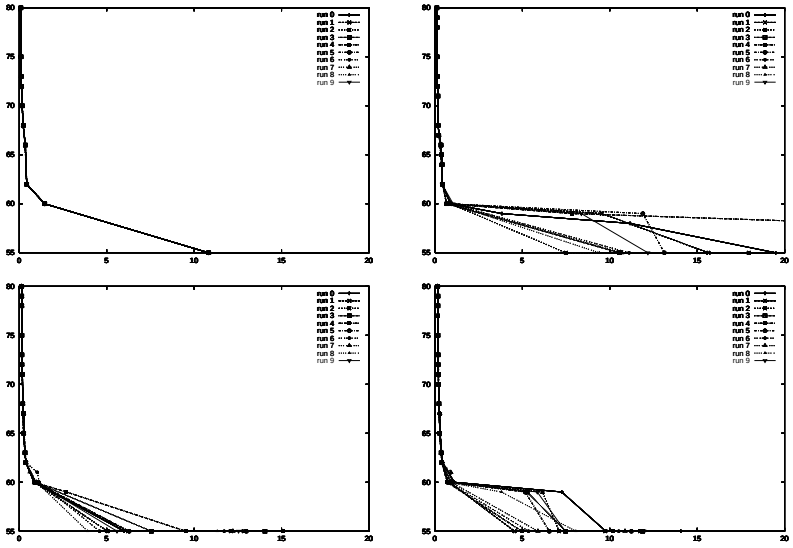

Figure 1: Golomb 10, x max = 20s

This first series of experiments already gives us almost enough material to support the three major points we want to illustrate in this article.

[4]when the optimal value is known

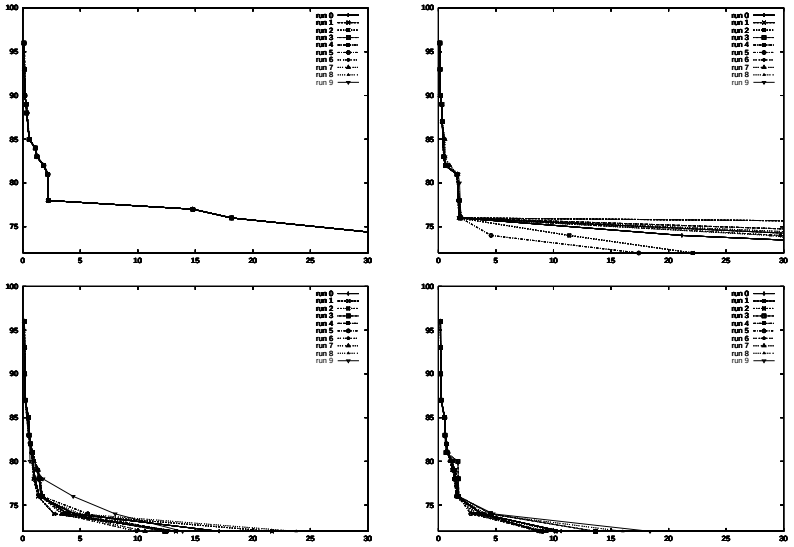

Figure 2: Golomb 11, x max = 30s

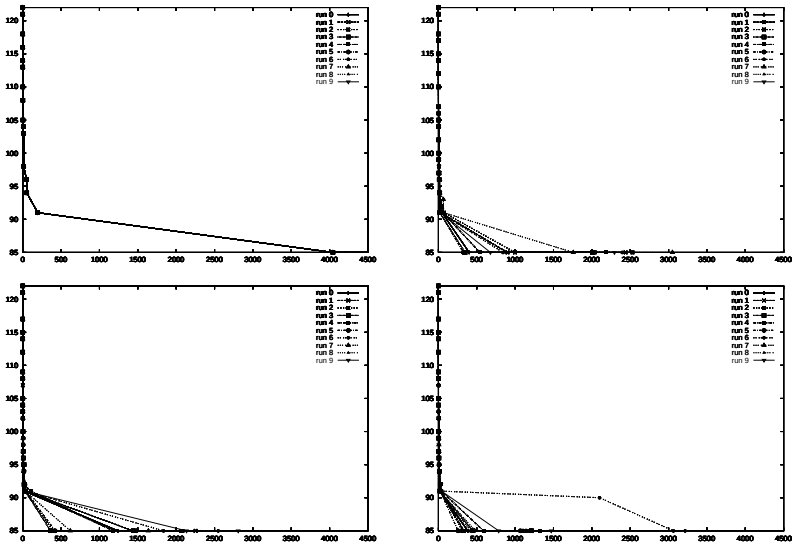

Figure 3: Golomb 12, x max = 4500s

The first point we want to emphasize is **speedups**. On the average, it is clear from these experiments that adding processors to solve a problem is worth it. This aspect is reinforced as the problem becomes more difficult. For instance, speedups are more visible with Golomb 11

and Golomb 12. This indicates that parallelism should be used for difficult problems [5].

The second point is **variability**. Solving the Golomb 10 problem with two processors clearly demonstrates this aspect of parallel problem solving as the results differ greatly among the 10 runs. Some of the runs do not finish and others do not find the optimal solution in 20 seconds, while the single processor run always finds the best solution in a bit more than 10 seconds. It is interesting to note that, of course, variability is null with a single processor and that variability is reduced when switching from two to three or four processors. In fact, this variability is the essence of super-linear speedups as it indicates the possible deviation from the predefined route one can expect from parallelism. This deviation can then lead to better solutions and thus to super-linear speedups. Thus the more variability we have, the greater the potential for super-linear speedups.

The last point we want to emphasize is linked to the **proof of optimality** of these problems. We can see in the tests of Golomb 10 with three and four processors that the constraint solver is able to prove the optimality[6] of Golomb 10. We can conjecture that parallelism induces nearly linear speedups when proving the optimality of a problem.

2.2 Solving Golomb in a Limited Time Frame

In this part of the study, we experiment with Golomb 10 with a 5 second time frame (figure 4), Golomb 11 with a 10 second time frame (figure 5), and Golomb 12 with a 250 second time frame (figure 6).

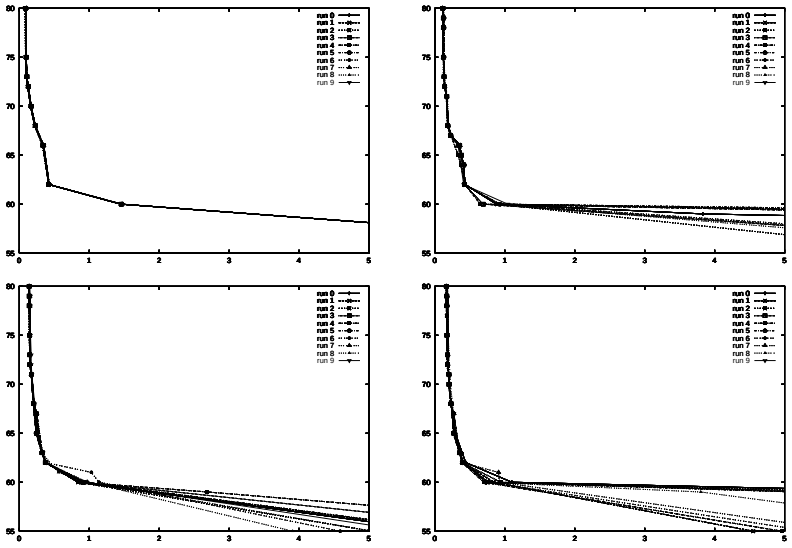

Figure 4: Golomb 10, x max = 5s

Restricting the time frame demonstrates another promising feature of parallel constraint programming as it transforms what we can expect from an incomplete search.

[5] This is even illustrated by the fact that we do not display results for Golomb problems with a size less than 10. These problems are solved in less than a second and parallel solver does not exhibit any meaningful speedups.

[6] This proof is indicated by a lobe sign on the x-axis right of the curve that touches the same x-axis

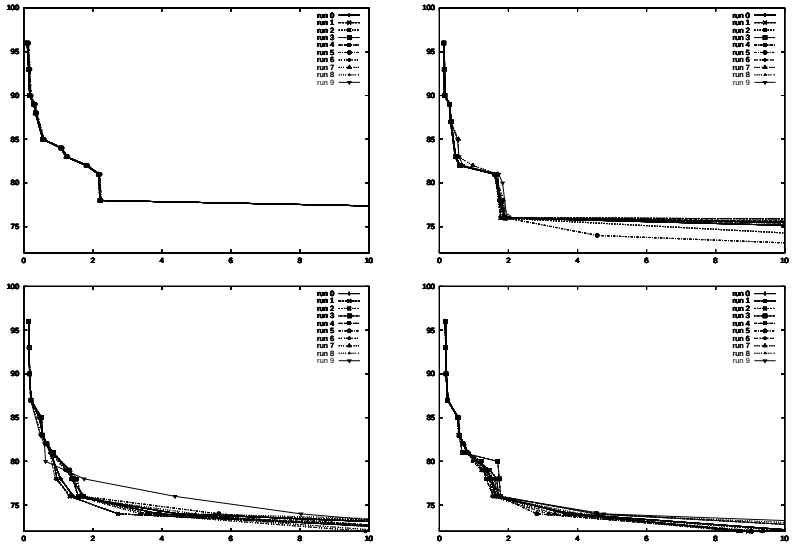

Figure 5: Golomb 11, x max = 10s

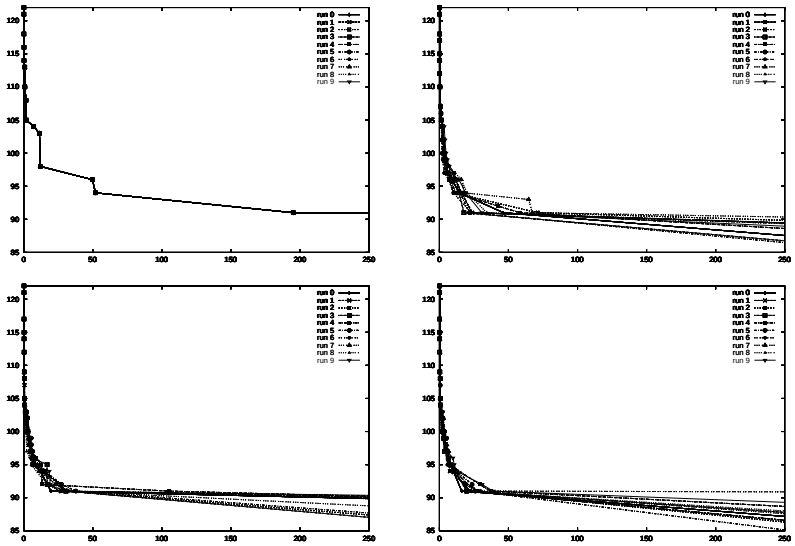

Figure 6: Golomb 12, x max = 250s

In fact, this translates into a profound change in perspective. We go from asking the question *What is the time needed to find the best solution or to prove it?* to *How can my solution, found in x seconds, be improved if I use more than a single processor to search?*

The results of these Golomb Ruler experiments clearly demonstrate the benefits of parallel constraint programming; in each of these experiments, the best solution found in a limited time

frame is always better with two to four processors than with a single processor.

Once again, this effect becomes more evident as the problem becomes more difficult. With the Golomb 10 problem, some runs of Parallel Solver with four processors find worse solutions after 5 seconds than the sequential version. This variation appears less frequently with Golomb 11 and Golomb 12.

2.3 Modeling Effects

In this section, we enrich the previous experiments with a second modeling of the problem. This modeling includes a symmetry-breaking constraint that improves propagation. This constraint states that the difference between the second and the first variable is greater than the difference between the last variable and the previous one. This has two main effects: (1) the search goal performs better and (2) there is more back-propagation of the cost function, leading to more pruning and reduced proving time of the optimal solution.

2.3.1 Solving with a Long Time Frame

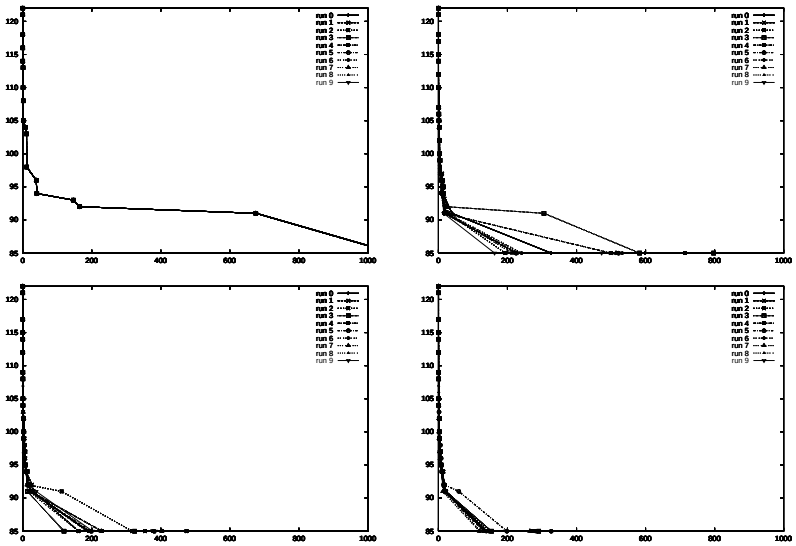

Figure 7: Golomb 12, x max = 1000s, second version

The improved modeling of the problem has a direct impact on variability as more pruning implies a smaller search tree and tighter follow-ups of intermediate solutions. This helps concentrate the search on the more promising parts of the search tree. As a direct consequence, variability is reduced on the Golomb 12 problem.

2.3.2 Solving with a Limited Time Frame

The improved modeling also has an impact on the results found when solving with a limited time frame (figure 8).

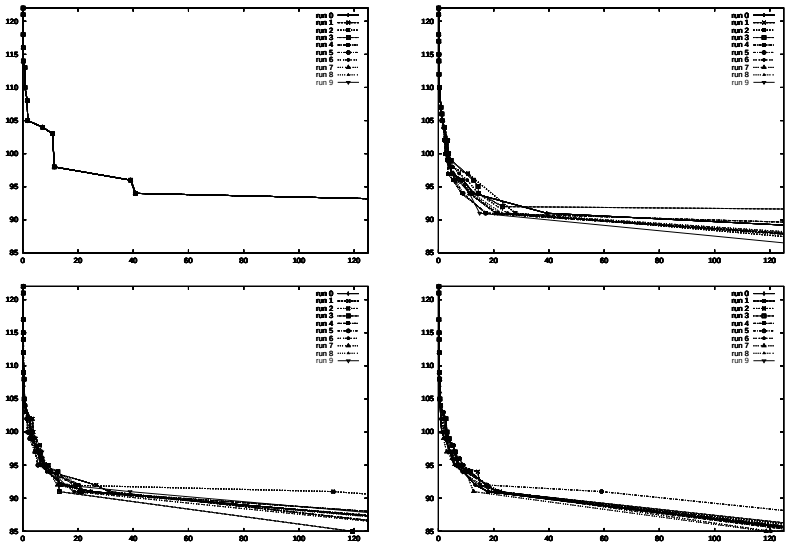

Figure 8: Golomb 12, x max = 125s, second version

With this improved modeling, it is clearly worthwhile to use four processors instead of a single processor when solving the Golomb 12 problem in a 125 second time frame.

3 Study of a Network Design Problem

In this section, we experiment with Parallel Solver on a network design problem. The problem can be described as follows. Given the following: (1) a graph where nodes are cities and arcs are possible routes for Internet traffic, (2) a quadratic set of demand from each city to all the others, and (3) a price per arc and per max capacity table, we must find a route for each demand that minimizes the total cost. This cost is computed as the sum of the cost for each arc, this latter cost being the cost for this arc of the capacity needed to route all traffic going through. As we can see, if *n* is the number of cities and n^2 is the size of the graph, then n^4 is the size of the solution. Therefore, this is a very big problem where the size of the closed instances does not exceed 10. Finally, describing this benchmark is a big work and should be the bulk of a future article.

Furthermore, this problem has two variations: (1) we force the traffic to be balanced on each arc (the total traffic going from A to B must be equal to the total traffic going from B to A) and (2) the routes must be symmetric (the route from city A to city B must be the symmetric of the route from city B to city A).

We can note that the solutions of the second variation of the problem are solutions of the first variation. And solutions of both variations are solutions to the original problem. We will use three numbers to denote a particular instance of the problem. The first is the number of nodes. The second number is one when we force the traffic to be balanced. The third number is one when we force the routes to be symmetric.

3.1 Poorly Behaved Experiments

We begin this discussion with two experiments where parallelism produces no performance gains.

3.1.1 The Lucky Goal

The first experiment represents an instance of size 7 (figure 9). It is solved with the Discrepancy-Bounded Depth First Search (DBDFS) procedure[2] with a width of two. We cannot achieve any kind of speedup with these experiments.

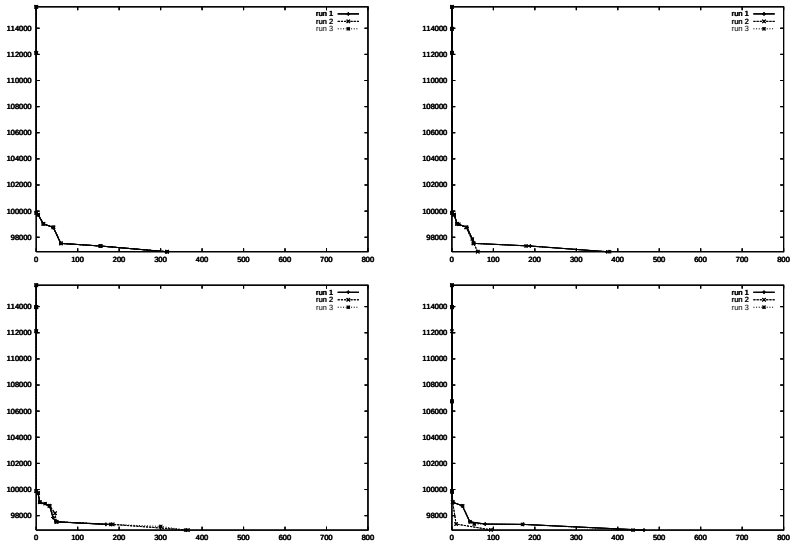

Figure 9: Network 7, 0 0 dbdfs(2) max = 800s

The explanation for the lack of performance gains due to parallelism in this experiment is quite enlightening. In fact, the search goal is quite lucky and is able to find a good solution quickly. This is a consequence of the fact that some of the costs involved in the data are nearly equivalent, leading to nearly equivalent choices[7]. Thus at some point, the search engine has to decide among a few alternatives that seem the same. However, while one will lead to a good solution and eventually to the optimal one, the others will not. This could also be seen as a lack of back-propagation from the objective function. The end result is that these choices are very sensitive and any perturbation, like another worker finding a somewhat better solution and perturbing the search tree, may lead the search engine out of the promising part of the search tree.

This behavior can be linked to the following simple rule. In a large search space, if the first search attempt finds the optimal solution, there can be no speedups at all regarding the objective function. This rule is naive, but illustrates the previous example well.

[7]Nearly equivalent choices with regard to an evaluation function based on cost.

3.1.2 The Easy Search

Another unsatisfactory result is observed with an instance of size 8. This best solution in the symmetric variation is obtained very quickly. We cannot expect any impressive speedups while solving time to optimality is less than a few seconds (figure 10).

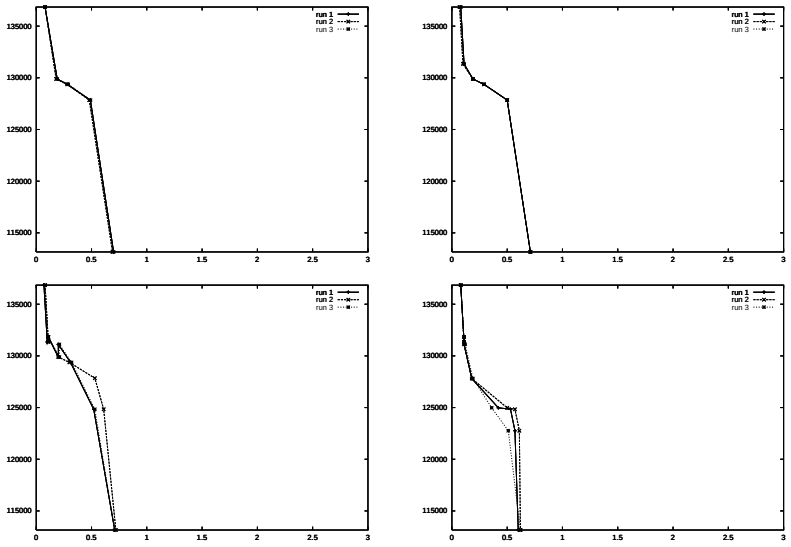

Figure 10: Network 8, 1 1 dbdfs(1) max = 3s

The interesting part is that while parallelism does not produce any performance gains in these two experiments, neither do these experiments exhibit any slowdown related to parallelism. All searches converge to the optimal solution in about 0.7 seconds.

3.2 Well behaved experiments

Here we present different runs on the instances of size 10 and 12. Please note that the instance of size 12 is still an open problem.

3.2.1 Simple Scalability Results

Here we present some tests where Parallel Solver is able to find better solutions more quickly than the sequential version. This is shown in figure 11.

3.2.2 Optimal Breakthrough

Here we present some very promising results. Not only did parallel search find equivalent solutions more quickly, but it is also continues to improve the objective value when the sequential version remains *stuck*. This is shown in figures 12, 13 and 14. Figure 14 clearly demonstrates the following two effects: improvement to the time needed to find a good solution (as clearly shown in the runs with one, two and three processors for the last solution found) and breakthrough as the four-processor runs are all able to find solutions unattainable by one, two and three processors.

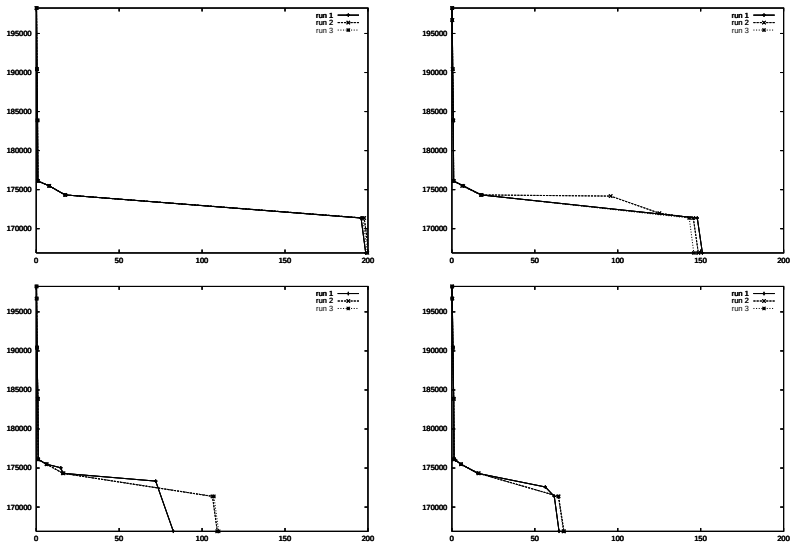

Figure 11: Network 10, 1 1 dbdfs(1) max = 200s

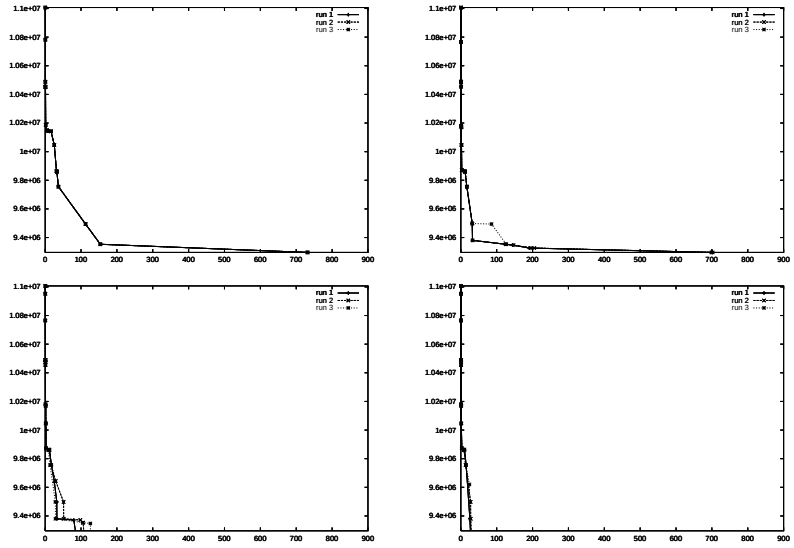

Figure 12: Network 12, 1 1 dbdfs(1) max = 900s

3.3 Effect of the Search Procedure

Finally, we illustrate the effect of the search procedure on parallel constraint programming. Here we decide to play with the width of the strips explored by the DBDFS search procedure[2]. We use the network design problem of size 10 with symmetric arcs, but not symmetric routes. We

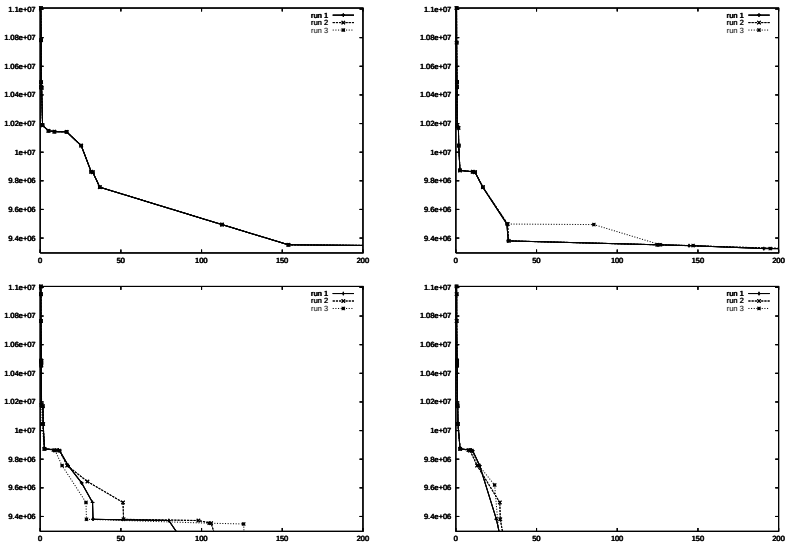

Figure 13: Network 12, 1 1 dbdfs(1) max = 200s

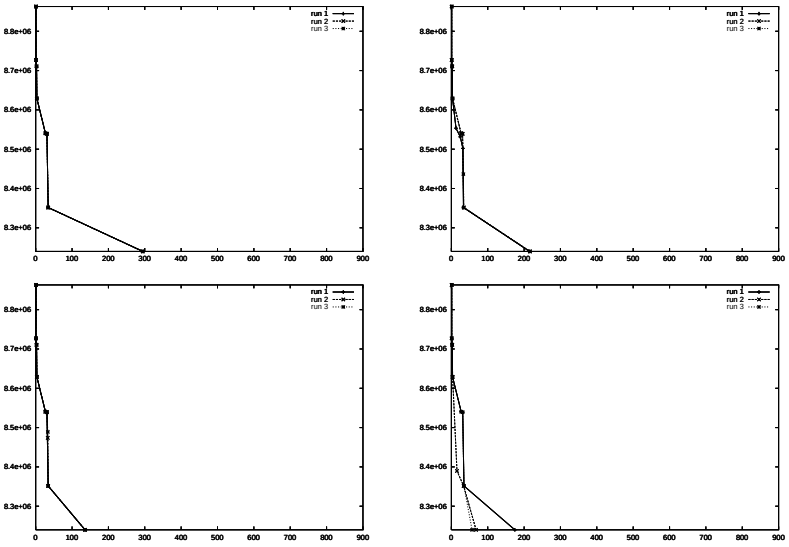

Figure 14: Network 12, 0 0 dbdfs(1) max = 900s

also use a width of one (figure 15), a width of two (figure 16), and a width of three (figure 17).

These runs exhibit many different aspects. The first one is that the solver is not able to find optimal solutions with a width of two or three while it is quite easy to find one with a width of one. This is the same effect as described in [2]. That is to say, there is a trade-off between (1) exploring larger strips, spending more time doing so but possibly finding better solutions than

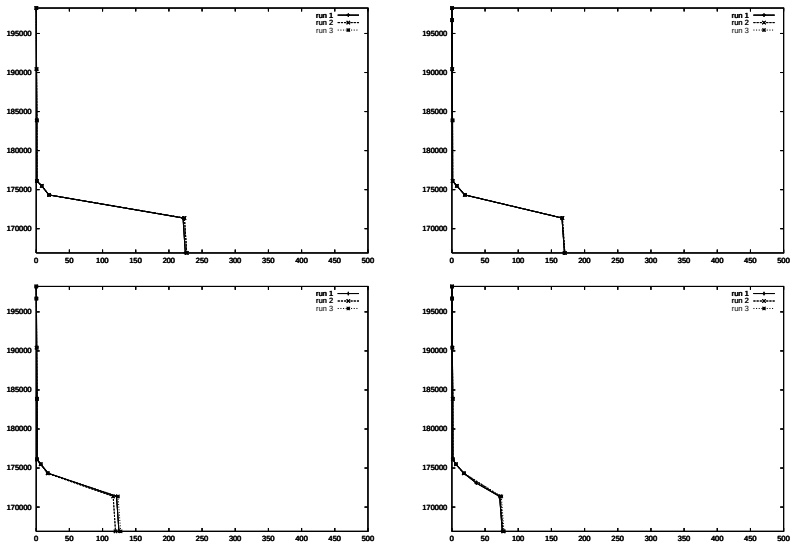

Figure 15: Network 10, 0 1 dbdfs(1) max = 500s

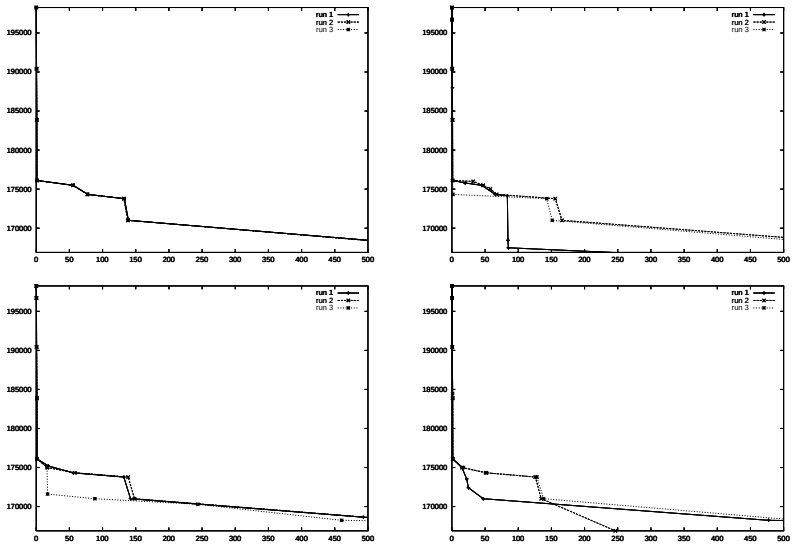

Figure 16: Network 10, 0 1 dbdfs(2) max = 500s

pure Depth First Search, and (2) exploring smaller strips, benefiting from the back-propagation of the cost and finally exploring a smaller search tree.

This trade-off between speed and quality is partially mitigated by the brute force of parallel search. Parallel search, in effect, makes the larger strips smaller and enables the solver to explore an improved search space.

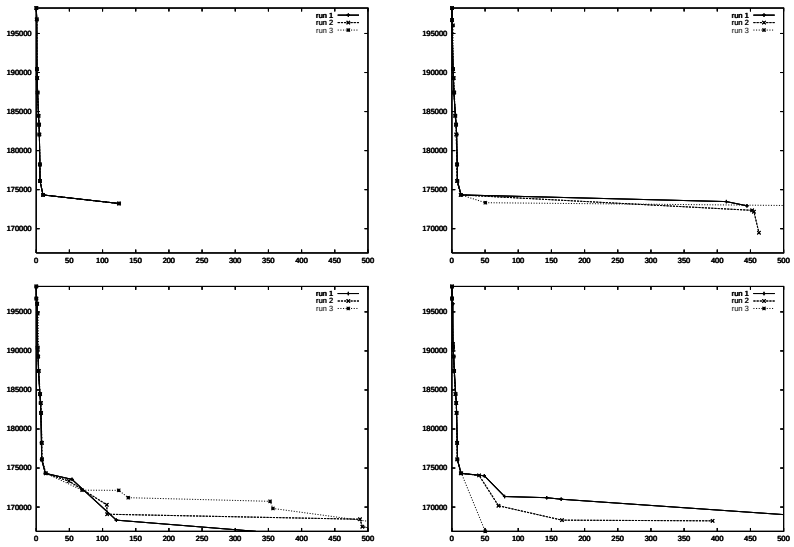

Figure 17: Network 10, 0 1 dbdfs(3) max = 500s

Searching with a width of one (figure 15) exhibits nice speedups with very little variability, which is quite interesting and indicates a robust goal.

On the contrary, searching with a width of two (figure 16) presents some variability. In fact, searching in other parts of the first strips may lead to better intermediate solutions (as seen in the runs with three processors), but eventually, everything will converge to the same asymptotic behavior, except for one run with two processors and one run with four processors. This convergence could be explained by the fact that good values are obtained in strips far away from the first one and that the search needs about the same amount of time to find them.

Finally, with a width of three, a single processor is stuck inside the first few big strips while with three and four processors, the exploration of the search tree is large enough so that one worker is able to find good solutions and eventually the optimal one.

4 Conclusion

We hope we have fulfilled our goals of developing realistic expectations and an intuitive understanding about parallel constraint programming. We try to provide basic general understanding from our limited experiments. With this guideline, our results can be summarized by the following simple statements:

Parallelism improves the volume of search space explored in a given amount of time. In a good scenario, this improves the performance of the search engine.

If the whole search space has to be explored and there are few back-propagation and superlinear speedup behaviors, we can expect a linear speedup.

On the contrary, if the search space is small, then we cannot expect meaningful improvements.

Furthermore, if the total explored search space is very small compared to the total search

space, randomness appears. This randomness can be contained by different aspects of the problem. If the search goal is good, any worker will be directed to interesting leaves of the search tree. If the pruning is good, any intermediate solution will reduce the search space and guide the workers to promising parts of the search tree.

Finally, we also hope to convince the constraint programming community of the potential associated with ILOG Parallel Solver. Our results demonstrate that parallelism improves the performance of search engines when used to solve difficult problems.

Acknowledgments

I would like to thank everybody who helped me in my work, especially Jean-François Puget for helping me getting out of design and implementation issues with Parallel Solver, Claude LePape and Jean-Charles Régin for their wonderful work on the network design problem, and Stephanie Cook-McMillen for helping me transform this article into something readable, which it was not before she worked on it.

References

[1] U. Baron, J.C. de Kergommeaux, H. Hailperin, M. Ratcliffe, P. Robert, J.C. Syre, and H. Westphal. The parallel ECRC prolog system PEPSys: an overview and evaluation results. In *International Conference on Fifth Generation Computer Systems*, pages 841–849, Tokyo, 1988. ICOT.

[2] J. Christopher Beck and Laurent Perron. Discrepancy-Bounded Depth First Search. In *Proceedings of CP-AI-OR 00*, March 2000.

[3] Bob Bixby and A. Martin. Parallelizing the dual simplex method. *INFORMS Journal on Computing*, 12:45–56, 2000.

[4] Mats Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, SICS, 1990.

[5] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Survey*, 26(3):295–336, September 1994.

[6] A. Ciepielewski, S. Haridi, and B. Hausman. Or-parallel prolog on shared memory multiprocessors. *Journal of Logic Programming*, 7:125:147, 1989.

[7] W. Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.

[8] Cplex. *ILOG CPLEX 7.5 User's Manual and Reference Manual*. ILOG, S.A., 2001.

[9] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[10] Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP99)*, pages 346–360. Springer-Verlag, 1999.

[11] T. K. Ralphs, L. Ladányi, and L.E. Jr Trotter. Branch, cut and price: Sequential and parallel. http://www.lehigh.edu/ tkr2/research/papers/LNCS.pdf.

[12] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*. The MIT Press, 1994.

[13] Solver. *ILOG Solver 5.2 User's Manual and Reference Manual*. ILOG, S.A., 2001.