Parallel and Random Solving of a Network Design Problem

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

Abstract

Industrial optimization applications must be "robust," *i.e.*, must provide good solutions to problem instances of different size and numerical characteristics, and must continue to work well when side constraints are added. In practice, this translates into problems where no preferred solving techniques exist and where the search tree is so huge that no method is expected to find (and prove) optimal solutions. A network design problem recently made public by France Telecom is a perfect example of this type of problem. It is also a perfect candidate for using two techniques: large neighborhood search and portfolios of algorithms. After a successful first application of these techniques to the problem, attention was focused on improving performance. First, a specializing *metaheuristic* to reduce speculative work was implemented. Second, the previous implementation was parallelized on a shared memory multiprocessor machine. New issues arose as all methods were difficult to parallelize. Multiple parallelization implementations were tried in order to improve the efficiency of parallel solving with these methods. The result was then shown to outperform all known CP-based methods.

Introduction

In the design and development of industrial optimization applications, one major concern is that the optimization algorithm must be robust. By "robust," we mean not only that the algorithm must provide "good" solutions to problem instances of different size and numerical characteristics, but also that the algorithm must continue to work well when constraints are added or removed. This expectation is heightened in constraint programming as the inherent flexibility of constraint programming is often put forward as its main advantage over other optimization techniques. Yet this requirement for robustness is rarely recognized as the top priority when the application is designed. Similarly, the benchmark problem suites that are used by the academic community generally do not reflect this requirement. In practice, it has important effects on the reinforcement of problem formulation, search management, the advantages of parallel search, the applicability of different optimization techniques including hybrid combinations, *etc.*

This paper presents a specific case study in which such questions have been addressed.

An extensive benchmark suite, presented in two joint papers (Le Pape *et al.* ; Bernhard *et al.* 2002), has been built on the basis of real network design data provided by France Telecom R&D. The suite includes three series of problem instances corresponding to different characteristics of the numerical data. In each series, seven instances of different sizes are provided. In addition, six potential side constraints are defined, leading to 64 versions of each instance. The goal is to design an algorithm which provides the best results on average when launched on each of the $3 * 7 * 64 = 1344$ instances with a CPU time limit of 10 minutes. In practice, the differences between the 1344 instances make it hard to design an algorithm that performs well on all instances. Notice that in the context of the current application, both the introduction of new technologies and the evolution of network usage can have an impact on problem size, numerical characteristics, and side constraints. It is believed that an optimization technique which applies well to all of the 1344 problem instances is more likely to remain applicable in the future than an optimization technique which performs particularly well on some instances, but fails to provide reasonable solutions on some others.

The contribution of this article is the application of new search techniques to the solving of this network design problem. CP modeling, basic search, first experiments were the result of team work and are described in (Le Pape *et al.* ; Bernhard *et al.* 2002).

Robust solving of this problem specifically requires dealing with three kind of difficulties:

Size The size of the problem instances varies substantially. This is illustrated by the depth of the first solution found by the CP program. In the cases, the depth is around 30; in hard cases, it is between 2,000 and 3,000. While exploring the complete search tree is feasible in the first case because propagation helps to cut branches of the tree, it is completely unthinkable in the second case. Moreover, even making a few mistakes early in the search tree is fatal as no search procedure can attempt to correct them in 10 minutes.

Selection: Deciding in which order demands are to be routed is error prone because of the cumulative nature of the cost. This decision also has major consequences on the minimization process afterward.

Topology: The side constraints and the numerical data make each problem instance very different. This makes it difficult

to design an algorithm efficient on each aspect of the problem. And in practice, many efforts to improve the algorithm on one aspect of the problem reduce its robustness as results are deteriorated on instances that do not contain this aspect.

We decided to attack each difficulty in sequence. The first technique used was to implement large neighborhood search (LNS) (Shaw 1998). Furthermore, randomness was introduced at different stages, first in the selection of the fragment of the whole solution to re-optimize and then in the order the demands are routed. This aspect improved the robustness of the methods, especially with regard to the selection of the demands. Finally, the robustness with regard to the topology of the problem was addressed by the implementation of the portfolio of algorithms paradigm (Gomes & Selman 1997).

The last part of this article is devoted to the improvement of the performance of the portfolio of algorithms. The first part describes improvements of the sequential implementation of the portfolio of algorithms. The second part describes the parallelization of these methods and the associated issues.

A Network Design Problem

The benchmark problem consists in dimensioning the arcs of a telecommunications network, so that a number of commodities can be simultaneously routed over the network without exceeding the chosen arc capacities. The capacity to be installed on an arc must be chosen in a discrete set and the cost of each arc depends on the chosen capacity. The objective is to minimize the total cost of the network.

Given are a set of n nodes and a set of m arcs (i, j) between these nodes. A set of d demands (commodities) is also defined. Each demand associates to a pair of nodes (p, q) an integer quantity Dem_{pq} of flow to be routed along a unique path from p to q. In principle, there could be several demands for the same pair (p, q), in which case each demand can be routed along a different path. Yet, to condense notation and keep the problem description easy to read, we will use a triple (p, q, Dem_{pq}) to represent such a demand.

For each arc (i, j), K_{ij} possible capacities $Capa^k_{ij}$, $1 \leq k \leq K_{ij}$, are given, to which we add the null capacity $Capa^0_{ij} = 0$. One and only one of these $K_{ij} + 1$ capacities must be chosen. However, it is permitted to multiply this capacity by an integer between a given minimal value $Wmin^k_{ij}$ and a given maximal value $Wmax^k_{ij}$. Hence, the problem consists in selecting for each arc (i, j) a capacity $Capa^k_{ij}$ and an integer coefficient w^k_{ij} in $[Wmin^k_{ij}, Wmax^k_{ij}]$. The choices made for the arcs (i, j) and (j, i) are linked. If capacity $Capa^k_{ij}$ is retained for arc (i, j) with a non-null coefficient w^k_{ij}, then capacity $Capa^k_{ji}$ must be retained for arc (j, i) with the same coefficient $w^k_{ji} = w^k_{ij}$, and the overall cost for both (i, j) and (j, i) is $w^k_{ij} * Cost^k_{ij}$.

Six classes of side constraints are defined. Each of them is optional, leading to 64 variants of each problem instance, identified by a six-bits vector. For example, "011000" indicates that only the second constraint $nomult$ and the third constraint $symdem$, as defined below, are active.

- The security (*sec*) constraint states that some demands must be secured. Secured demands must be routed through secured arcs.
- The no capacity multiplier (*nomult*) constraint forbids the use of capacity multipliers.
- The symmetric routing of symmetric demands (*symdem*) constraint states that for each demand from p to q, if there exists a demand from q to p, then the paths used to route these demands must be symmetric.
- The maximal number of bounds (*bmax*) constraint associates to each demand (p, q, Dem_{pq}) a limit $Bmax_{pq}$ on the number of bounds (also called "hops") used to route the demand, *i.e.*, on the number of arcs in the path followed by the demand.
- The maximal number of ports (*pmax*) constraint associates to each node i a maximal number of incoming ports Pin_i and a maximal number of outgoing ports $Pout_i$.
- The maximal traffic (*tmax*) constraint associates to each node i a limit $Tmax_i$ on the total traffic managed by i.

Twenty-one data files, organized in three series, are available. Each data file is identified by its series (A, B, or C) and an integer which indicates the number of nodes of the considered network. Series A includes the smallest instances, from 4 to 10 nodes. Series B and C include larger instances with 10, 11, 12, 15, 16, 20, and 25 nodes. The instances of series B have more choices of capacities than the instances of series A, which have more choices of capacities than the instances of series C. So, in practice, instances of series B tend to be harder because the search space is larger, while instances of series C tend to be harder because each mistake has a higher relative cost.

Finally, as described in (Le Pape *et al.*), we tried to solve it with three approaches: Constraint Programming using Ilog Solver, MIP using Ilog CPLEX and Colum Generation using both. In this article, we focus on the CP approach.

Using a Subset of the Full Benchmark

As the whole benchmark takes 1344 * 10 minutes to run, which is more than nine days, we first concentrated our efforts on a small subset of the whole set of parameters. We chose to focus on five problem instances:

B10 000000 An average problem with a best CP-found solution of 20510 and the best known solution of 19395[1].

B10 100111 A problem where the first solution may become very difficult to solve if the heuristic is not robust enough. The best CP-found solution is 28083, the best known solution is 25534, found by CPLEX and a MIP formulation.

C10 100011 A problem where the first solution may become very difficult to solve if the heuristic is not robust enough. The best CP-found solution is 18925, the same as the best known solution.

[1]This solution is not actually found, but inferred from a found solution of a more constrained variation of the problem. In this case, when the problem is more constrained, the search space is much smaller, thus allowing the solvers to find a better solution.

C12 000000 A problem where column generation gives much better solutions. CP finds 40099 and column generation finds 37385.

C25 100000 A huge problem where CP is currently the only technique finding feasible solutions[2]. The CP solution is 149500 and the best known solution is 109293.

We believe this selection, while totally arbitrary, will allow us to conduct fruitful experiments. In the end, we will use the best of the breed method on the complete benchmark to evaluate how it performs.

We will display each set of results[3] in this order in an array:

20510	28083	18925	40099	145900

In order to take into account the randomness involved in these results, we will display one standalone run for each experiment that we will choose as representative of the multiple runs.

And as a goal, we will always compare to the best known results:

19395	25534	18925	37385	109223

Breaking Barriers with Randomization and Large Neighborhood Search

Using Randomization and Large Neighborhood Search

Our first addition to the standard CP framework was a large neighborhood search schema (Shaw 1998). Starting from an instantiated solution stating routes for each demand, we chose to freeze a large portion of this solution and to reoptimize the unfrozen part.

The unfrozen part was chosen using the following algorithm: Given a number n of visits and $size$, the total number of visits, we compute the ratio $\rho = n/size$. Then we iterate on all demands and freeze them with a probability $(1 - \rho)$. Throughout the rest of the paper, the number n will be fixed to 30.

Then we loop over the optimization part, each time with a new neighborhood.

Using a Different Implementation of Fast Restart

We chose to follow the guidelines in (Gomes & Selman 1997). We decided to implement a fast restart strategy, which was implemented in two different ways.

- The first implementation relied on the Discrepancy-Bounded Depth First Search (DBDFS) procedure (Beck & Perron 2000) with a maximum discrepancy usually set to 1.
- The second implementation used the search techniques implemented by O. Lhomme in (Lhomme 2002). The idea is to kill the right branch at each binary choice point with a probability p (set to 93% throughout this paper).

[2]This will change when the MIP of the column generation approach is improved.

[3]Each sequential run was conducted on a Pentium III 1.13 GHz running Windows XP, using Microsoft Visual Studio.NET C++ Compiler.

We believe all these fast restart implementations are better than those based on a restart after a given number of backtracks (or failures) as they will more uniformly search the search tree while the previous schema based on Depth First Search will concentrate on the bottom of the search tree.

The first method (DBDFS) gave the results:

20760	27414	18925	39738	140738

The second method (amortized) gave the results:

20188	27592	18925	40099	138929

As we can see, no method strictly dominates the other and they both improve previous results by a small margin.

Using a Change in the Instantiation Order

As described in (Le Pape *et al.*), the instantiation order between demands is fixed using heuristic weights associated with each demand. We tried to give a random order over demands, hoping that this would correct mistakes in the fixed instantiation order. We believe that the fixed instantiation order, while quite efficient, makes some big mistakes in first routing big unconstrained demands. This means that small critical demands will be routed afterward on a network already full of traffic.

The DBDFS method was judged best using this improvement and gave the results:

20778	27417	18925	36603	140922

As we can see, this method gives a significant improvement in the fourth test (C12 000000) which is significantly better than the column generation approach. There is a deterioration in results on the fifth test. The amortized search does not benefit much from this change.

Using a Portfolio of Algorithms

While working on the routing of each demand, it appeared that each modification we made that dramatically improved the solution of one or two problem instances would deteriorate in a significant way the solution of one or two other instances.

This was the perfect case to apply portfolios of algorithms (Gomes & Selman 1997).

Simple Implementation of Portfolio of Algorithms

Our first implementation of the portfolio of algorithms technique was made using a round-robin schema. Each large neighborhood search loop would be made using one algorithm, with each algorithm chosen in a round-robin way. To implement different algorithms, we examined the routing of each demand. As described in (Le Pape *et al.*), for each demand, we compute a shortest path from the source to the sink (of the unrouted part of the demand). The last arc of this shortest path is then chosen and a choice point is created. The left branch of the choice point states that the route must use this arc and the right branch states that this arc is forbidden for this route. This selection is applied until the demand is completely routed.

This method was changed by computing different kinds of penalties on the cost of each arc and by choosing different

combinations of standard cost and penalties. This allowed us to create different *algorithms* by applying different coefficients to each component of the cost of one arc.

One again, the DBDFS method was better and gave the results:

20524	27498	18925	38805	131323

This method is most useful for the fifth test, where only the additional algorithms (which are not efficient on the other problems) are able to go below the 135000 barrier.

Specialization Schema on a Portfolio of Algorithms

While attractive, this implementation of a portfolio of algorithms tends to waste a lot of resources. Let us imagine that we have n algorithms and that only one algorithm can improve our routing problem, then we spend $(n-1)/n$ of our time in a speculative and unproductive way.

We then decided to implement a specialization mechanism. Given n algorithms A_i, we use an array of integer weights w_i. Initially, each weight is set to 3. We then choose one algorithm against its weight probability $(w_i)/(\sum_j w_j)$. In the event of the success of a LNS loop, the weight of the successful algorithm is increased by 1 (with an upper bound arbitrarily set to 12). In case of repeated failure of an algorithm (in our implementation, 20 consecutive failures), the weight w_i is decreased by 1 (with another arbitrarily chosen lower bound of 2).

The result is a specializing schema which concentrates on the successful algorithms for a given problem.

The first method (DBDFS) gave the results:

19627	27249	18925	37495	125327

The second method (amortized) gave the results:

19605	28219	18925	36429	125327

As we can see, this specialization schema is very effective. Every test benefits from this approach. And in this approach, there is no clear winner between the DBDFS approach and the amortized one.

Parallelizing the Portfolio of Algorithms

Given the success of the previous sequential methods, we decided to use our Pentium III 700MHz quad processor.

A Brief Introduction to Parallel Solver

ILOG Parallel Solver is a parallel extension of ILOG Solver (Solver 2001). It was first described in (Perron 1999). It implements *or*-parallelism on shared memory multiprocessor computers. ILOG Parallel Solver provides services to share a single search tree among workers, ensuring that no worker starves when there are still parts of the search tree to explore and that each worker is synchronized at the end of the search.

First experiments with ILOG Parallel Solver are described in (Perron 2002) and (Bernhard *et al.* 2002). Switching from the sequential version to the parallel version required a minimal code change of a few lines, and so we were immediately able to experiment with parallel methods.

Simple Parallelism

The first parallelization of our LNS + random search was very simple. We simply used the usual API of ILOG Parallel Solver and with minor changes to the sequential code (less than five lines of code), we were able to implement parallel LNS + random + portfolio of algorithms.

The first method (DBDFS) gave the results:

19960	27357	18598	36963	126530

The second method (amortized) gave the results:

20057	27627	18598	37583	125327

These results are a bit disappointing. If we do simple math, 4 * 700 MHz = 2.8 GHz (compared to 1.13 GHz). We expected better results. In fact, we have a degradation in the first and second test results, a breakthrough in the third where we improve the best known solution, a small improvement in the fourth test, and a small degradation in the fifth one.

Deeper investigation showed that parallelization was inefficient as, on average, only 2 out of 4 processors were used at a time. For the first method, this is a consequence of the degenerated nature of the search tree as a DBDFS search procedure with a maximum number of discrepancies of 1 builds a very special tree where load balancing is not very efficient. For the second method, the search tree is different but produces only a few active choice points, thus leading to poor parallelization results.

Concurrent Use of a Portfolio of Algorithms

As shown in the previous section, parallelizing the specialization schema gives poor results. We then decided to investigate another kind of parallelization. We implemented the original portfolio of algorithms design where each method was run in parallel. Furthermore, in order to hide latency and idle workers, we decided to use more algorithms than processors (6 algorithms on a 4-processor box).

The first method (DBDFS) gave the results:

20326	27573	18598	36507	126530

The second method (amortized) gave the results:

20173	24940	18925	36582	126530

The results, compared to the previous test, are equivalent in the DBDFS approach and improve the amortized approach. Study of the computer workload revealed that approximately 60% of the total computing power is used at any given time. This is a little better than the previous naive implementation, but this implementation has a serious flaw - it is not scalable. In fact, it will not be efficient if, for example, there are more processors than different algorithms in use.

Multipoint Large Neighborhood Search

Therefore, we decided to implement a last schema, which combined ideas from the two previous implementations. We implemented multipoint LNS with specialization. This means that at each LNS loop, the algorithm is chosen using the specialization schema. The instantiation order of the

demands is chosen randomly, but in the same way for each worker. However, each worker works on a different fragment of the whole problem to re-optimize, using different randomly chosen parts. Furthermore, in order to hide latency, we use a few more workers than processors (7 workers and 4 processors).

The first method (DBDFS) gave the results:

20021	25382	18448	36191	123628

The second method (amortized) gave the results:

19855	25412	18592	35931	122190

This last implementation is the best so far. It combines excellent results, scalability, and robustness. The load was constant between 90% and 100% during the whole search process.

Furthermore, the choice of 7 workers is optimal as we can see in the results with 8 processors (first line) and 6 processors (second line):

20344	25449	18448	36905	127764
20177	26120	18925	36457	128177

Application to the Complete Benchmark

We then decided to compare the simple parallelism approach to the approach used in (Le Pape *et al.*). We compared the sum of all the results and the Mean Relative Error (MRE) on 6 instances of the problem using the 64 parameter values.

The following table indicates the results of sequential CP + local search (first line), parallel CP + local search (second line), and parallel CP + LNS + specialization (third line).

	B10	B11	B12	C10	C11	C12
Sum	1626006	3080608	2571936	1110966	2008833	2825499
MRE	10.57%	15.12%	10.07%	5.77%	11.30%	13.97%
Sum	1592778	2967717	2535516	1085266	2005714	2777129
MRE	9.18%	13.13%	9.35%	3.66%	11.29%	12.91%
Sum	1552054	2815979	2413371	1079480	1865869	2526719
MRE	6.46%	7.23%	3.98%	2.73%	4.00%	2.69%

The figures speak for themselves. They demonstrate a significant improvement in terms of robustness thanks to the combination of parallelism, LNS, randomness, and portfolios of algorithms. These results should be further improved by the implementation of concurrent multipoint LNS.

Conclusion and Future Work

The contributions of this article can be summarized as follows. First, the use of randomness and portfolios of algorithms is very important, especially in the context of robustness, as shown in this industrial benchmark. The use of these techniques leads to significant improvements.

The second aspect is linked to the implementation of these search methods. We believe every implementation of fast restart should benefit from both the DBDFS and the amortized approaches. We have also shown that parallel efficiency is greatly improved by the combination of the specialization mechanism and the multipoint concurrent implementation.

Finally, we think that future improvements will come from additional structures in this random approach. Future work could explore the choice of the fragment to re-optimize. This choice could, for example, benefit from additional information extracted from the previous solution. Fragments could also be constructed in a more structured way. But we are aware that any simple selection schema will not increase the robustness of our program. As many failed experiments have taught us, nothing but the most sophisticated ideas will be more robust than the simple random-based implementations.

And finally, we would like to experiment these ideas on computers with more processors: 8, 16 and even 32.

Acknowledgments

This work is only a small part of a project partially financed by the French MENRT, as part of RNRT project ROCOCO. I wish to thank all the persons that have worked directly and indirectly on it, without whom nothing would have been possible. This includes Jacques Chambon and Raphaël Bernhard from France Télécom R&D and Dominique Barth from the PRiSM laboratory. This also includes, from ILOG, Jean-Charles Régin, Claude Le Pape, Alain Chabrier, Philippe Réfalo, and Emilie Danna. Finally, I would like to thank Jamel Tayeb from Intel for his invaluable help in this parallelization effort and Stephanie Cook-McMillen for transforming this article into something readable.

References

Beck, J. C., and Perron, L. 2000. Discrepancy-Bounded Depth First Search. In *Proceedings of CP-AI-OR 00*.

Bernhard, R.; Chambon, J.; Lepape, C.; Perron, L.; and Régin, J. C. 2002. Résolution d'un problème de conception de réseau avec parallel solver. In *Proceeding of JFPLC*. (In French).

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

Gomes, C. P., and Selman, B. 1997. Algorithm Portfolio Design: Theory vs. Practice. In *Proceedings of the Thirteenth Conference On Uncertainty in Artificial Intelligence (UAI-97)*. New Providence: Morgan Kaufmann.

Le Pape, C.; Perron, L.; Régin, J.-C.; and Shaw, P. Robust and parallel solving of a network design problem. Submitted to CP 2002.

Lhomme, O. 2002. Amortized random backtracking. In *Proceedings of CP-AI-OR 2002*, 21–32.

Perron, L. 1999. Search procedures and parallelism in constraint programming. In Jaffar, J., ed., *Proceedings of CP '99*, 346–360. Springer-Verlag.

Perron, L. 2002. Practical parallelism in constraint programming. In *Proceedings of CP-AI-OR 2002*, 261–276.

Shaw, P. 1998. Using constraint programming and local search methods to solve vehicle routing problems. In Maher, M., and Puget, J.-F., eds., *Proceeding of CP '98*, 417–431. Springer-Verlag.

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