Parallel Crew Scheduling in PAROS[*]

Panayiotis Alefragis[1], Christos Goumopoulos[1], Efthymios Housos[1], Peter Sanders[2], Tuomo Takkula[3], Dag Wedelin[3]

[1] University of Patras, Patras, Greece
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[3] Chalmers University of Technology, Göteborg, Sweden

Abstract. We give an overview of the parallelization work done in PAROS. The specific parallelization objective has been to improve the speed of airline crew scheduling, on a network of workstations. The work is based on the Carmen System, which is used by most European airlines for this task. We give a brief background to the problem. The two most time critical parts of this system are the pairing generator and the optimizer. We present a pairing generator which distributes the enumeration of pairings over the processors. This works efficiently on a large number of loosely coupled workstations. The optimizer can be described as an iterative Lagrangian heuristic, and allows only for rather fine-grained parallelization. On low-latency machines, parallelizing the two innermost loops at once works well. A new "active-set" strategy makes more coarse-grained communication possible and even improves the sequential algorithm.

1 The PAROS Project

The PAROS (Parallel large scale automatic scheduling) project is a joint effort between Carmen Systems AB of Göteborg, Lufthansa, the University of Patras and Chalmers University of Technology, and is supported under the European Union ESPRIT HPCN (High Performance Computing and Networking) research program. The aim of the project is to generally improve and extend the use and performance of automatic scheduling methods, stretching the limits of present technology.

The starting point of PAROS is the already existing Carmen (Computer Aided Resource Management) System, used for crew scheduling by Lufthansa as well as by most other major European airlines. The crew costs are one of the main operating costs for any large airline, and the Carmen System has already significantly improved the economic efficiency of the schedules for all its users. For a detailed description of the Carmen system and airline crew scheduling in general, see [1]. See also [3, 6, 7] for other approaches to this problem.

At present, a typical problem involving the scheduling of a medium size fleet at Lufthansa requires 10-15 hours of computing, for large problems as much as 150 hours. The closer to the actual day of operation the scheduling can take

[*] This work has been supported by the ESPRIT HPCN program.

place, the more efficient it will be with respect to market needs and late changes. Increased speed can also be used to solve larger problems and to increase the solution quality. The speed and quality of the crew scheduling algorithms are therefore important for the overall efficiency of the airline.

An important objective of PAROS is to improve the performance of the scheduling process through parallel processing. There exist attempts to parallelize similar systems on high end parallel hardware (see [7]), but our focus is primarily to better utilize a network of workstations and/or multiprocessor workstations, allowing airlines such as Lufthansa to use their already existing hardware more efficiently.

In section 2 we give a general description of the crew scheduling process in the Carmen system. The following sections report the progress in parallelizing the main components of the scheduling process: the pairing generator in section 3, and the optimizer in section 4. Section 5 gives some conclusions and pointers for future work.

2 Solution Methodology and the Carmen System

For a given timetable of flight *legs* (non stop flights), the task of crew scheduling is to determine the sequence of legs that every crew should follow during one or several workdays, beginning and ending at a home base. Such routes are known as *pairings*. The complexity of the problem is due to the fact the crews cannot simply follow the individual aircraft, since the crews on duty have to rest in accordance with very complicated regulations, and there must always be new crews at appropriate locations which are prepared to take over. At Lufthansa, problems are usually solved as weekly problems with up to the order of 10^4 legs.

For problems of this kind there is usually some top level heuristic which breaks down the problem into smaller subproblems, often to different kinds of daily problems. Due to the complicated rules, the subproblems are still difficult to handle directly, and are solved using some strategy involving the components of pairing generation and optimization.

The main loop of the Carmen solution process is shown in Figure 1. Based on an existing initial or current best solution a subproblem is created by opening up a part of the problem. The subproblem is defined by a *connection matrix* containing a number of legal leg connections, to be used for finding a better solution. The connection matrix is input to the *pairing generator* which enumeratively generates a huge number of pairings using these connections. Each pairing must be legal according to the contractual rules and airline regulations, and for each pairing a cost is calculated. The generated pairings are sent to the optimizer, which selects a small subset of the pairings in order to minimize the total crew costs, under the constraints that every flight leg must receive a crew. In both steps the main algorithmic difficulty is the combinatorial explosion of possibilities, typical for problems of this kind. Rules and costs are handled by a special rule language, and is translated into runnable code by a rule compiler, which is then called by the pairing generator.

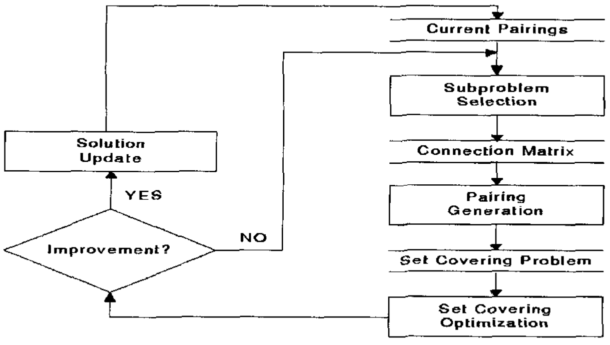

Fig. 1. Main loop in Carmen solution process.

A typical run of the Carmen system consists of 50-100 daily iterations in which 10^4 to 10^6 pairings are generated in each iteration. Most of the time is spent in the pairing generator and the optimizer. For some typical Lufthansa problems, profiling reveals that 5-10% of the time is consumed in the overall analysis and the subproblem selection, about 70-85% is spent in the pairing generator, and 10-20% in the optimizer. These figures can however vary considerably for different kinds of problems. The optimizer can sometimes take a much larger proportion of the time, primarily depending on the size and other characteristics of the problem, the complexity of the rules that are called in the inner loop of the pairing generator, and various parameter settings. As a general conclusion however, it is clear that the pairing generator and the optimizer are the main bottlenecks of the sequential system, and they have therefore been selected as the primary targets for parallelization.

3 Generator Parallelization

The pairing generation algorithm is a quite straightforward depth first enumeration that starts from a crew base, and builds a chain of legs by following possible connections as defined by the matrix of possible leg connections. The search is heuristically pruned by limiting the number of branches in each node, typically between 5-8. The search is also interrupted if the chain created so far is not legal.

The parallelization of the pairing generator is based on a manager/worker scheme, where each worker is given a start leg from which it enumerates a part of the tree. The manager dynamically distributes the start legs and the necessary additional problem information to the workers in a demand driven manner. Each worker generates all legal pairings of the subtree and returns them to the manager.

The implementation has been made using PVM, and several efforts have been made to minimize the idle and communication times. Large messages are sent whenever possible. Load balancing is achieved by implementing a dynamic workload distribution scheme that implicitly takes into account the speed and the current load of each machine. The number of start legs that are sent to each worker is also changing dynamically with a fading algorithm. In the beginning, a large number of start legs is given and in the end only one start leg per worker is assigned. This scheme reconciles the two goals of minimizing the number of messages and of good load balancing. Efficiency is also improved by pre-fetching. A worker requests the next set of start legs before they are needed. It can then perform computation while its request is being serviced by the manager.

The parallel machine can also be extended dynamically. It is possible to add a new host at any time to the virtual parallel machine and this will cause a new worker to be started automatically.

Another feature of the implementation is the suspend/resume feature which makes sure that only idle workstations are used for generation. Periodically the manager requests the CPU load of the worker, suspends its operation if the CPU load for other tasks is over a specified limit, and resumes its operation if the CPU load is below a specified limit. The performance overhead depends on the rate of the load checking operation which in any case is small enough ($\leq 1\%$).

3.1 Scalability

For the problems tested, the overhead for worker initialization depends on the size of the problem and consists mainly of the initialization of the legality system and the distribution of the connection matrix. This overhead is very small compared to the total running time.

The data rate for sending generated pairings from a worker to the manager is typically about 6 KB/s, assuming a typical workstation and Lufthansa rules. This is to be compared with the standard Ethernet speed of 1 MB/s, so after its start-up phase the generator is CPU bound for as many as 200 workstations. Given the high granularity and the asynchronous execution pattern we therefore expect the generator to scale well up to the order of 100 workstations connected with standard Ethernet, of course provided that the network is not too loaded with other communication.

3.2 Fault Tolerance

Several fault tolerance features have been implemented. To be reliable when many workstations are involved, the parallel generator can recover from task and host failures. The notification mechanism of PVM [4] is used to provide application level fault tolerance to the generator.

A worker failure is detected by the manager which also keeps the current computing state of the worker. In case of a failure the state is used for reassigning the unfinished part of the work to another worker. The failure of the manager can be detected by the coordinating process, and a new manager will be started. The recovery is achieved since the manager periodically saves its state to the filesystem (usually NFS) and thus can restart from the last checkpoint. The responsibility of the new manager is to reset the workers and to request only the generation of the pairings that have not been generated, or have been generated partially. This behavior can save a lot of computing time, especially if the fault appears near the end of the generation work.

3.3 Experimental Results

We have measured the performance of the parallel generator experimentally. The experiments have been performed on a network of HP 715/100 workstations of roughly equivalent performance, connected by Ethernet at the computing center of the University of Patras, during times when almost exclusive usage of the network and the workstations was possible. The results are shown in Table 3.3. What we see is the elapsed time for a single iteration of the generator, and the running time in seconds as a function of the number of CPUs. The speedup in all cases is almost linear to the number of CPUs. The generation time decreases in all cases almost linearly to the number of the CPUs used.

Table 1. Parallel generator times for typical pairing problems.

problem name	legs	pairings	1 CPU	2 CPUs	4 CPUs	6 CPUs	8 CPUs	10 CPUs
lh_dl_gg	946	396908	26460	13771	7061	4536	3466	2818
lh_dl_splimp	946	318938	20760	10797	5448	3686	2797	2181
lh_wk_gg	6196	594560	31380	16834	8436	5338	4312	3288
sj_dl_kopt	1087	159073	10860	5563	2804	1892	1385	1112

4 Optimizer Parallelization

The problem solved by the optimizer is known as the set covering problem with additional base capacity constraints, and can be expressed as

$$\min\{cx : Ax \geq 1, Cx \leq d, x \text{ binary}\}$$

Here A is a 0-1 matrix and C is arbitrary non-negative. In this formulation every variable corresponds to a generated pairing and every constraint corresponds to a leg. The size of these problems usually range from 50×10^4 to $10^4 \times 10^6$ (constraints $\times$ variables), usually with 5-10 non-zero A-elements per variable. In most cases only a few capacity ($\leq$) constraints are present.

This problem in its general form is NP-hard, and in the Carmen System it is solved by an algorithm [9], that can be interpreted as a Lagrangian based heuristic. Very briefly, the algorithm attempts to perturbate the cost function as little as possible to give an integer solution to the LP-relaxation. From the point of view of parallel processing it is worth pointing out that the character of the algorithm is very different, and for the problems we consider also much faster, compared to the common branch and bound approaches to integer programming, and is rather more similar to an iterative equation solver. This also means that the approach to parallelization is completely different, and it is a challenge to achieve this on a network of workstations.

The sequential algorithm can be described in the following simplified way that highlights some overall aspects which are relevant for parallelization. On the top level, the sequential algorithm iteratively modifies a Lagrangian cost vector $\bar{c}$ which is first initialized to c. The goal of the iteration is to modify the costs so that the sign pattern of the elements of $\bar{c}$ corresponds to a feasible solution, where $\bar{c}_j < 0$ means that $x_j = 1$, and where $\bar{c}_j > 0$ means that $x_j = 0$. In the sequential algorihm the iteration proceeds iteratively by considering *one constraint at a time*, where the computation for this constraint modifies the $\bar{c}$ values for the variables of that constraint. Note that this implies that values corresponding to variables not in the constraint are not changed by this update. The overall structure is summarized as

```
$\bar{c} = c$
reset all $s^i$ to 0
$\kappa = 0$
repeat
        for every constraint $i$
            $r^i = \bar{c}^i - s^i$
            $s^i$ = function of $r^i$, $b_i$ and some parameters
            $\bar{c}^i = r^i + s^i$
        increase $\kappa$ according to some rule
until no sign changes in $\bar{c}$
```

In this pseudocode, $\bar{c}^i$ is the sub-vector of $\bar{c}$ corresponding to non-zero elements of constraint i. The vector s^i represents the contribution of constraint i to $\bar{c}$. This contribution is cancelled out before every iteration of constraint i where a new contribution is computed. The vectors r^i are temporaries.

A property relevant to parallelization is that when an iteration of a constraint has updated the reduced costs of its variables, then the following constraints have to use these new updated values. Another property is that the vector s^i is usually a function of the current $\bar{c}$-values for at most two variables in the constraint, and these variables are referred to as the *critical variables* of the constraint.

In the following subsections we summarize the different approaches to parallelization that have been investigated. In addition to this work, an aggressive optimization of the inner loops was performed, leading to an additional speedup of roughly a factor of 3.

4.1 Constraint Parallel Approach

A first observation is that if constraints have no common variables they may be computed independently of each other, and a graph coloring of the constraints would reveal the constraint groups that could be independent. Unfortunately, the structure of typical set covering problems (with many variables and few long constraints) is such that this approach is not possible to apply in a straightforward way. The approach can however be modifed to let every processor maintain a local copy of $\bar{c}$, and a subset of the constraints. Each processor then iterates its own constraints and updates its local copy of $\bar{c}$. The different copies of $\bar{c}$ must then be kept consistent through communication. Although the nature of the algorithm is such that the $\bar{c}$ communication can be considerably relaxed, the result for our type of problems did not allow a significant speedup on networks of workstations.

4.2 Variable Parallel Approach

Another way of parallelization is to parallelize over the variables and the inner loops of the constraint calculation. The main parts of this calculation that concern the parallelization are

1) collect the $\bar{c}$ for the variables in the constraint.
2) find the two least values of r^i.
3) copy the result back to $\bar{c}$.

When the variables are distributed over the processors, each processor is then responsible for only one piece of $\bar{c}$ and the corresponding part of the A-matrix. Some of the operations needed for the constraint update can be conveniently done locally, but the double minimum computation requires communication. This operation is associative, and it is therefore possible to get the global minimal elements through a reduction operation on the local critical minimum elements. To minimize communication it is also possible to group the constraints and perform the reduction operation for several independent constraints at a time.

This strategy has been successfully implemented on an SGI Origin2000, with a speedup of 7 on 8 processors. However, it cannot be used directly on a network of workstations due to the high latency of the network. We have therefore investigated a relaxation to this algorithm, that does a lazy update of $\bar{c}$. The approach is based on updating $\bar{c}$ only for the variables required by the constraint group that will be iterated next. The reduced cost vector is then fully updated based on the contributions of the previous constraint group, during the reduction operation of the current constraint group, and so overlaps computation with both communication and idle time. The computational part of the algorithm is slightly enlarged and the relaxation requires that constraint groups with a reasonable number of common variables can be created. The graph colouring heuristics that have been used are based on [8], which gives 10-20% fewer groups than a greedy first fit algorithm, although the computation times are slightly higher.

For numerical convergence reasons random noise is added in the solution process and even with minor changes in the parameters the running times can vary considerably, and also the solution quality can be slightly different. Tests are shown in Table 4.2 for pure set covering problems from the Swedish Railways, and the settings have been adjusted to ensure that a similar solution quality is preserved on the average (usually well within 1% of the optimum), and the problems shown here are considered as reasonably representative for the average case. Running times are given in seconds for the lazy variable parallel implementation for 1 to 4 processors. The hardware in this test has been 4 × HP 715/100 connected with 10 Mbps switched Ethernet. It can be seen that reasonably good speedup results are obtained even for medium sized problem instances. For networks of workstations a possibility could here also be to use faster networks such as SCI [5] or Myrinet [2] and streamlined software interfaces.

Table 2. Results for the lazy variable parallel code.

problem name	rows	columns	1 CPU	2 CPUs	3 CPUs	4 CPUs
sj_daily_17sc	58	1915	0.60	2.95	3.53	4.31
sj_daily_14sc	94	7388	17.30	9.45	11.78	11.82
sj_daily_04sc	429	38148	288.73	124.76	82.94	72.00
sj_daily_34sc	419	156197	951.53	634.54	365.90	259.32

4.3 Parallel Active Set Approach

As previously mentioned, the constraint update is usually a function of at most two variables in the constraint. If the set of these critical variables was known in advance, it would in principle be possible to ignore all other variables and receive the same result much faster. This is not possible, but it gives intuitive support for a variation of the sequential algorithm, where a smaller number of variables that are likely to be critical are selected in an *active set*. The idea is to make the original algorithm work only on the active set, and then sometimes do a scan over all variables to see if more variables should become active. For our problems this is especially appealing considering the close relation to the so called column generation approach to the crew scheduling problem, see [3]. The approach was first implemented sequentially and turns out to work well on real problems. An additional and important performance benefit is that the scan can be implemented by a columnwise traversal of the constraint matrix which is much more cache friendly and gives an additional sequential speedup of about 3 for large instances.

This modified algorithm also gives a new possiblity for parallelization. If the active set is small enough, then the scan will dominate the total execution time, even if the active set is iterated many times between every scan. In constrast to the original algorithm, all communication necessary for a global scan of all variables can be concentrated into a single vector valued broadcast and reduction operation. Therefore, the parallel active set approach is successful even on a network of workstations connected over Ethernet. Some results for pure set covering problems from Lufthansa are shown in Table 4.3. The settings have been adjusted to ensure that a similar solution quality is preserved on the average.

Running times are given in seconds for the original production code Probl, the new code with the active set strategy disabled, and with active set strategy for 1, 2 and 4 processors. The hardware in this test has been 4 × Sun Ultra 1/140 connected with a shared 100 Mbps Ethernet. From these results, and other more

Table 3. Results of parallel active set code.

problem name	rows	columns	Probl	no active set	1 CPU	2 CPUs	4 CPUs
lh_dl26_02	682	642613	9962	4071	843	412	294
lh_dl26_04	154	121714	1256	373	216	105	60
lh_dt1_11	5287	266966	1560	765	298	78	66
lh_dt58_02	5339	409350	2655	2305	924	406	207

extensive tests, it can be concluded that in addition to the sequential speedup, we can obtain a parallel speedup of about a factor of 3 using a network of four workstations. It does not make sense however to increase the number of workstations significantly, since it is then no longer true that the scan dominates the total running time.

5 Conclusions and Future Work

We have demonstrated a fully successful parallelization of the two most time critical components in the Carmen crew scheduling process, on a network of workstations.

For the generator, where a coarse grained parallelism is possible, a good speedup can be obtained also for a large number of workstations, when however issues of fault tolerance become very important. Further work includes the management of multiple APC jobs, a refined task scheduling scheme, preprocessing inside the workers (e.g. elimination of identical columns) and enhanced management of the virtual machine (active time windows of machines).

For the optimizer, which requires a much more fine grained parallelism, Figure 2 shows a theoretical model of the expected parallel speedup of the different parallelizations, for a typical large problem. Although the variable parallel approach scales better, the total speedup is higher with the active set since the sequential algorithm is then faster. On the other hand, the active set can be responsible for a lower solution quality for some instances. Further tuning of the parallel active set strategy and other parameters is therefore required to improve the stability of the code both with respect to quality and speedup. It may be possible to combine the two approaches but it is not clear if this would give a significant benefit. Also, the general capacity constraints are not yet handled in any of the parallel implementations.

The parallel components have been integrated in a first PAROS prototype system that runs together with existing Carmen components. Given that the

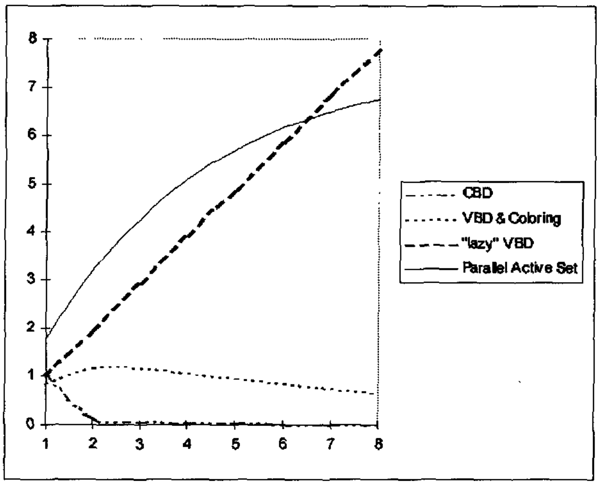

Fig. 2. Expected parallel speedup.

generator and the optimizer are now much faster, we consider parallelization also of some other components such as the generation of the connection matrix, which should be comparatively simple. Other integration issues are the communicaton between the generator and the optimizer, which is now done through files, and related issues such as optimizer preprocessing.

References

1. E. Andersson, E. Housos, N. Kohl, and D. Wedelin. *OR in the Airline Industry*, chapter Crew Pairing Optimization. Kluwer Academic Publishers, Boston, London, Dordrecht, 1997.
2. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.
3. J. Desrosiers, Y. Dumas, M. Solomon, and F. Soumis. *Handbooks in Operations Research and Management Science*, chapter Time Constrained Routing and Scheduling. North-Holland, 1995.
4. A. Geist. Advanced programming in PVM. *Proceedings of EuroPVM 96*, pages 1–6, 1996.
5. IEEE. Standard for the scalable coherent interface (sci), 1993. IEEE Std 1596-1992.
6. S. Lavoie, M. Minoux, and E. Odier. A new approach for crew pairing problems by column generation with an application to air transportation. *European Journal of Operational Research*, 35:45–58, 1988.
7. R. Marsten. RALPH: Crew Planning at Delta Air Lines. *Technical Report. Cutting Edge Optimization*, 1997.
8. A. Mehrota and M. Trick. A clique generation approach to graph coloring. *INFORMS Journal of Computing*, 8, 1996.
9. D. Wedelin. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. *Annals of Operations Research*, 57, 1995.