Truck Scheduling on Multicore

Scheduling von Transportern auf Mehrkernrechnern

Victor Pankratius, Walter F. Tichy, Karlsruhe Institute of Technology (KIT)

Summary Transportation businesses reduce costs by optimizing the routes of their trucks. However, determining optimal truck schedules is computationally intensive, running for hours on sequential computers. This article describes experience with parallelizing SAP's Vehicle Scheduling and Routing Optimizer on shared-memory multicore computers. Re-engineering this complex application for a 24-core machine reduced typical computation time on real data from 1.5 hours to 5 minutes. The article discusses successful and unsuccessful parallelization approaches and concludes with lessons learnt. ►►► **Zusammenfassung** Speditionsunternehmen reduzieren Kosten, indem sie die Routenpläne für ihre Fahrzeuge optimieren. Die Ermittlung optimaler Transportpläne ist jedoch berechnungsintensiv und kann auf sequenziellen Rechnern mehrere Stunden dauern. Dieser Artikel beschreibt Erfahrungen mit der Parallelisierung des "Vehicle Scheduling and Routing Optimizer" der Firma SAP auf Mehrkernrechnern mit gemeinsamen Speicher. Die Parallelisierung dieser komplexen Anwendung hat auf einem Mehrkernrechner mit 24 Kernen die typische Optimierungszeit für reale Daten von 1,5 Stunden auf 5 Minuten reduziert. Der Artikel beschreibt erfolgreiche und nicht erfolgreiche Parallelisierungsansätze und fasst abschließend wichtige Erkenntnisse zusammen.

Keywords D.1.3 [Software: Programming Techniques: Concurrent Programming] Parallel programming; D.2.0 [Software: Software Engineering: General]; multicore, parallelization, transportation problem, logistics ►►► **Schlagwörter** Mehrkernrechner, Parallelisierung, Transportproblem, Logistik

1 Introduction

Transportation businesses reduce costs by optimizing the routes of their trucks. However, determining optimal schedules is computationally intensive, running for hours on sequential computers. We present the results of a parallelization project in which computation time has been reduced to minutes on shared-memory multicore computers. Multicore computers are attractive because they cost less than clusters and are easier to operate.

The article describes experience with parallelizing SAP's Vehicle Scheduling and Routing Optimizer. The original optimizer employs an evolutionary algorithm. It was important to preserve the properties of this algorithm, because a lot of effort had been spent on refining and tuning it on complex customer data. Moreover, customers would expect the parallel version to produce results close to those of the sequential version, only faster. Thus, developing an entirely new, parallel algorithm was out of the question; instead, the existing optimizer had to be re-engineered for parallelism.

The article is based on the M.S. thesis of Markus Hossner [3] and is organized as follows. Section 2 introduces the optimization problem. Section 3 describes SAP's sequential optimizer that served as a starting point. Section 4 discusses two parallelization strategies that were considered most promising. Section 5 shows parallel performance results. Section 6 summarizes lessons learnt. Section 7 contrasts this project with other multicore case studies.

2 The Problem

The vehicle scheduling and routing problem is an extension of the well-known traveling salesman problem [5]: Given a graph $G = (V, E)$ with a set of vertices V, a set of edges E and edge weights $w(e_i)$ that represent transportation costs, find a closed, cost-minimal tour in which every vertex occurs exactly once. The salesman uses this tour to visit all nodes. In the vehicle routing problem [11], the single salesman is replaced by a fleet of trucks delivering orders. Each truck covers a portion of the graph.

In addition to determining the best routes for the trucks, a solver must provide an assignment of orders to trucks, plan stops at warehouses for transshipment, and solve constraints such as that certain loads can be transported by certain truck models only.

In our real transportation scenarios, the number of orders and destinations ranges from tens to thousands. Hundreds of trucks are typically required for delivery. There can be hundreds of additional constraints for a single transport planning scenario. The constraints complicate the search and its parallelization. Since the problem is NP-complete, only approximations to the true optimum can be found.

3 The Sequential Vehicle Scheduling Optimizer

The sequential vehicle scheduling and routing optimizer is part of the SAP Business Suite. It is written in C++ and consists of about 150 000 lines of code, of which 20 000 lines are relevant for parallelization.

The optimizer uses an evolutionary algorithm to compute a transport plan consisting of vehicle schedules and routes. Evolutionary algorithms are well matched to customer requirements in the trucking business:

1. It is not necessary to find the global optimum; customers just need a solution that is "good enough";
2. orders can be added or removed even if the computation has already started, without rerunning the optimization from scratch.

The evolutionary algorithm works with candidates each of which represents a solution to the transport problem, including information on vehicles, schedules, and routes. Each candidate's cost is computed and the best solutions are mutated to generate a new population of candidates. This iterative process continues until the cost of a solution is below a pre-defined threshold or computation time runs out. The details are beyond the scope of this paper, so we sketch the key working principles.

The sequential optimizer starts with five candidates, each of which is generated with a different greedy algorithm, for example based on tours ordered by earliest/latest due date, shortest/longest driving time, or some user-defined order. Identical candidates are eliminated.

Then the algorithm applies a sequence of random mutations, depth-first search, and iterated local search as outlined in Fig. 1. Random mutations are used to diversify solutions. Depth-first search explores a particular part of the search space. Iterated local search perturbs a candidate and then generates new neighboring candidates from it by applying a mutation operator; for example, the 2-opt operator [5] eliminates two randomly chosen edges and reconnects the new graph "cross-wise", which yields a new tour. A similar operator is applied on other tour data, such as vehicle schedules or loads. In total, the optimizer uses more than 20 different mutation operators.

A general problem with this approach is that the optimizer might get stuck in a local minimum. This is the reason why the optimizer employs several candidate sets and meta-heuristics. The general idea is to switch to candidates that may be worse than the best solutions found so far, but lie far away from current candidates and initiate search in other subspaces.

4 Parallelization Approaches

In principle, evolutionary algorithms are not difficult to parallelize. The challenge was to preserve the evolutionary strategies of the sequential optimizer. We outline the implementation constraints prior to parallelization and present two parallelization approaches that were considered most promising; we refer to [3] for others. The first approach splits up the search space into disjoint parts that are traversed in parallel. The second approach applies several heuristics simultaneously on the entire search space.

4.1 Implementation Constraints

The practical context imposed several constraints on the parallel implementation. For example, it was not possible to employ existing libraries or compiler extensions such as Pthreads [4], Threading Building Blocks [10], or OpenMP [2]. SAP has its own threading library built on top of Pthreads [4] which uses wrappers and facade patterns. Applications in the SAP business suite are required to use the proprietary library to make code easier to maintain. In addition, parallelizing the existing code required task parallelism and fine-granular synchronization rather than loop parallelization as supported by OpenMP.

4.2 Parallel Search Space Traversal

About 80% of the sequential optimizer's runtime is spent on candidate generation and evaluation. It seemed straight-forward to parallelize generation and evaluation of candidates within a search step. As shown in Fig. 2a, each thread receives a copy of a candidate, performs mutations to generate neighbors, and computes their cost. This strategy is intuitive, but unfortunately it does not increase performance! The problem is due to overhead: Measurements revealed that creating a copy of the start

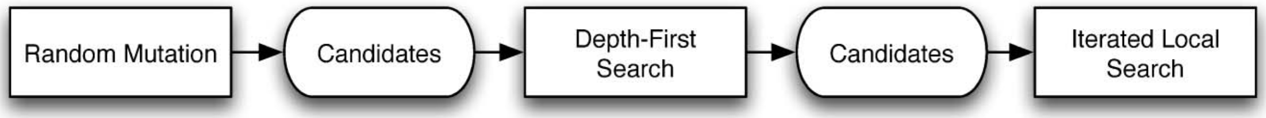

Figure 1 Steps applied by the sequential optimizer on candidate solutions.

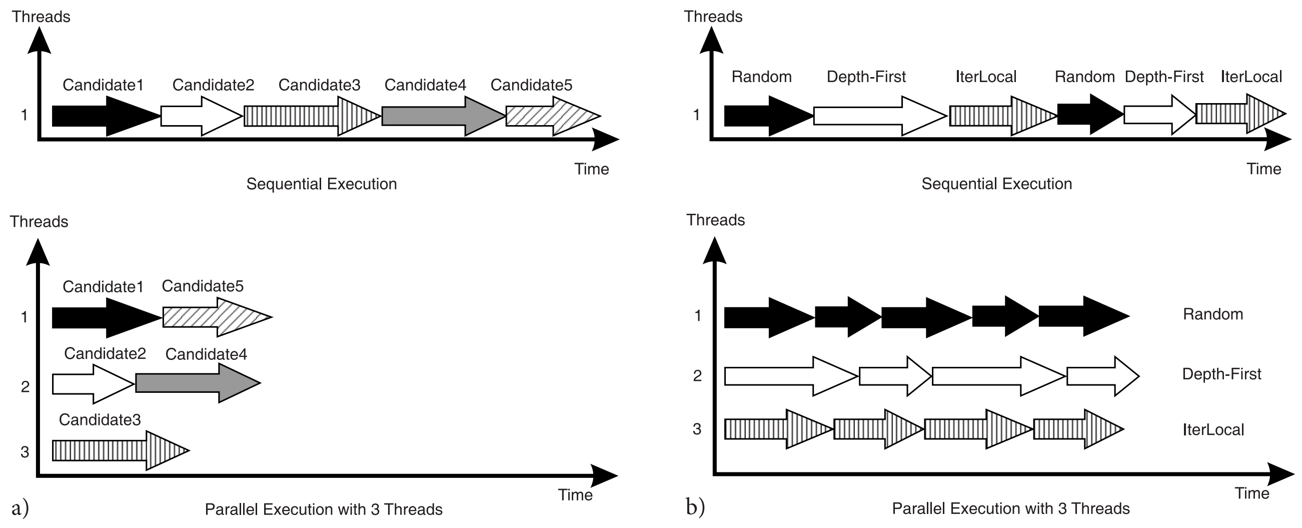

Figure 2 Comparison of parallelization approaches: (**a**) Parallel search space traversal; (**b**) Parallel application of heuristics.

candidate takes about 1 ms per thread, while the cost of mutation and evaluation is about 0.01 ms. So at least one hundred evaluations would have to be performed to amortize the cost per candidate. However, the typical neighborhood explored per thread is too small to amortize the cost of copying.

This experience demonstrates that it is not enough to reason about a parallelization strategy in general, but that it is important to estimate overhead and compare it to computation time.

4.3 Parallel Application of Heuristics

This approach uses different threads to apply different classes of heuristics in parallel, as shown in Fig. 2b. Each of the heuristics random mutation, depth-first search, and iterated local search is executed by one or more threads. The number of threads executing a particular heuristic is balanced in such a way that the candidate sets are always non-empty, avoiding waiting times.

If candidates are processed in parallel, there is a particular implementation problem that the developer needs to handle: Sets may contain candidates from different evolutionary generations at the same time. This wasn't a problem in the sequential case, because mutations were applied serially. However, failing to synchronize populations in the parallel implementation makes evolutionary search ineffective.

Our first implementation of parallel heuristics was inefficient and analyzed fewer candidates than the sequential version. Intel's VTune performance profiling tools pointed to the source of the problem, which was synchronization overhead. SAP's threading library used global locks that serialized many calls, e. g., for each dynamic memory allocation and for each usage of container data structures such as *list*, *map*, and *vector*.

To tackle the problems, most work went into restructuring code to make parallelization more efficient. Memory allocators were rewritten to allocate from thread-private memory. Other restructuring was necessary to aggregate and process data privately within threads and to reduce the size of critical sections. The parallelization of the optimizer was successful only after completing the code restructuring work. The next section outlines the performance results.

5 Performance Results

We measured the performance of the parallel heuristics approach (Sect. 4.3), which was the best one. All results were obtained on a 24 core machine (4x Intel Dunnington at 2.4 GHz) with 32 GB RAM, running Windows 2003 Server Edition.

Table 1 shows 15 different benchmark scenarios for vehicle scheduling, out of which 12 are based on real customer data. Using 24 threads, Fig. 3 shows the speedups for each scenario. The bars show the minimum, average, and maximum speedup from three executions of each scenario. The average speedup for all scenarios is 17, which is a respectable result, but it shows that it is difficult to achieve a uniform speedup in all situations.

Performance results vary for each scenario due to the specific problem size, constraints, and parallelization and synchronization overhead resulting from the respective constraints. It is interesting to observe that some runs achieve superlinear speedups, up to 211 on 24 cores! Superlinear speedups of this sort are due to randomization effects: One of the search threads may find the best solution right away, so there is no need to continue. In contrast, the sequential version might exhaustively explore a subspace until it reaches a search space that contains the satisfactory solution. This case study provides a demonstration that these extreme and rare cases discussed in textbooks [12] do occur in practice.

Scenarios 2, 3, and 4 are not based on real customer data. Their performance results differ systematically from the other scenarios. This is excellent evidence that parallel applications need to be evaluated on real-world data. If

Table 1 Details outlining the complexity of each transportation scenario.

scenario	real customer	orders	departure stations	desti-nations	transfer stations	requires trailers	requires other resources	has time constraints	problem description file size [KB]
1	yes	140	1	75	–	–	–	–	1211
2	no	1101	4	200	–	–	×	×	2024
3	no	1101	4	200	–	–	–	×	2025
4	no	75	4	25	1	–	–	–	96
5	yes	1019	3	250	–	–	–	×	1175
6	yes	1960	1	142	–	–	–	×	2533
7	yes	50	1	8	–	×	×	–	256
8	yes	804	1	31	–	–	×	–	626
9	yes	662	11	46	–	–	×	–	732
10	yes	90	14	62	–	–	×	×	1839
11	yes	227	2	30	1	–	–	–	230
12	yes	1176	1	559	–	–	–	×	7709
13	yes	255	5	194	–	–	–	×	24 746
14	yes	255	5	194	–	–	–	×	20 988
15	yes	217	2	160	–	–	–	×	68 016

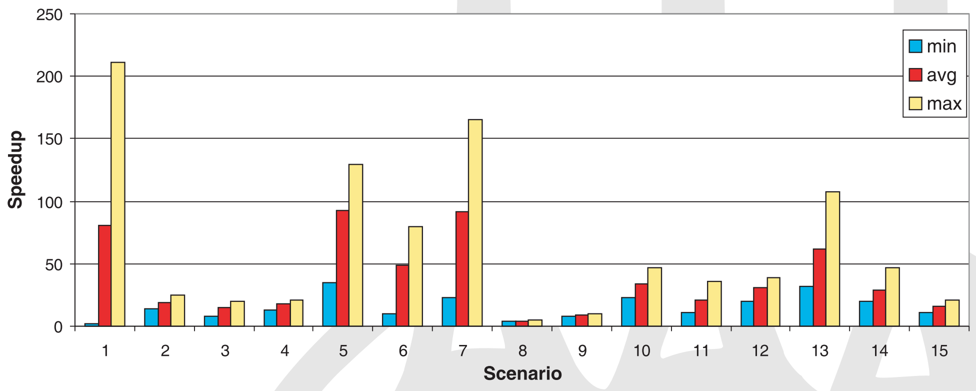

Figure 3 Speedups for 15 scenarios at 24 threads. Results for minimum, average, and maximum of three runs.

this study had been conducted on the artificial benchmarks only, the results would not have shown the real picture. Compared to the artificial scenarios, the average speedup of almost all real-world scenarios is higher.

Figure 4a and b summarize performance results. Figure 4a shows that increasing the number of threads leads to an almost linear increase in the number of candidate solutions analyzed, reaching a factor of 23.6 at 24 threads. The parallel algorithm was set to a 10 minute threshold, and all 15 scenarios were executed and averaged. Note that the average speedup is 17 because some candidates are examined by multiple threads. This result

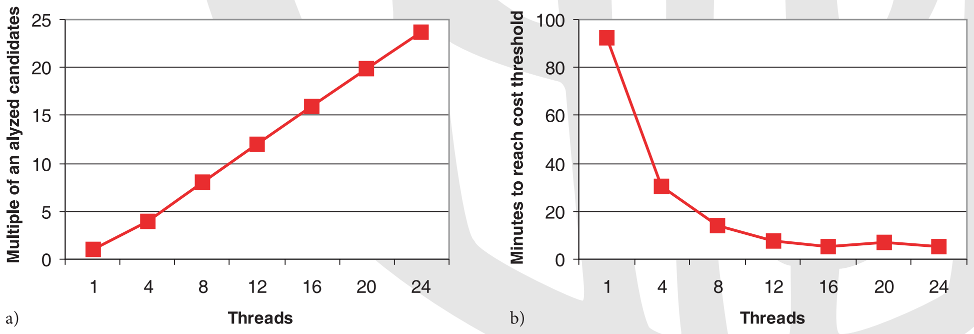

Figure 4 (**a**) Multiple of analyzed candidate solutions for a 10 minutes threshold, averaged over 15 scenarios. (**b**) Time until a pre-defined cost threshold is reached.

is influenced by the time limit and several threads examining the same candidates by accident. Figure 4b shows runtime in minutes until a pre-defined cost threshold is reached, dependent on the number of threads. The chart shows that more threads shorten the time required, but the curve levels off and there appears to be more variance compared to Fig. 4a due to randomization.

6 Lessons Learnt

The lessons from this project are as follows:

- Restructuring the existing code base was enough; it was not necessary to rewrite everything or to develop an entirely new algorithm.
- The initial, plausible parallelization approach did not work. The overhead outweighed the benefit of parallelization.
- Estimating overhead is important in practice. Developers need to consciously think about the minimum number of tasks, work package sizes, and execution times per thread necessary to amortize parallelization overhead. We recommend measurement and back-of-the-envelope calculations to estimate the overhead for a particular parallelization strategy before the strategy is fully implemented.
- Multithreaded memory allocators are critical for good parallel performance.
- Tools are indispensable for coping with the complexity of real-world applications. Performance profiling tools provide insight into what causes performance problems. In our case, profiling tools helped trace performance problems to global locks. However, tools have limitations: Intel's VTune Performance Analyzer and Thread Profiler could not handle large memory footprints in several routing scenarios.

Overall, parallelizing a sequential complex application worked, but not in the straightforward manner that some parallel programming textbooks [1; 2] suggest.

7 Other Multicore Case Studies

We conducted other parallelization case studies in addition to this project. References [6–9] report on applications in areas such as indexing, compression, biological data analysis, simulation, and graph computations that were parallelized for multicore. In particular, reference [8] discusses parallelization strategies for BZip2, an open-source compression program, and confirms some of the observations made in this article. First of all, the sequential compression strategy had to be preserved to obtain binary compatibility on compressed files. Second, parallelization approaches that were thought to be obvious and straight-forward did not work. Third, most work went into restructuring the sequential version to enable parallelization. All studies illustrate that developing parallel algorithms is not a major issue; the greatest difficulty lies in re-engineering the sequential codes.

8 Conclusion

This case study demonstrates that it is possible to parallelize a complex industrial application without starting from scratch. The average computation time for SAP's parallel transport optimizer was reduced from 1.5 hours to 5 minutes. This reduction makes it possible to compute schedules much more often and enables the optimization of larger problem instances. From a theoretical standpoint, a speedup average of 17 on a 24 core machine is satisfactory. Some scenarios showed dramatic super-linear speedups due to randomization effects. Even though extreme speedups are considered rare, this study shows that they do occur in practice. SAP's future product versions will benefit from the insights and the successful parallelization strategies described in this article.

Acknowledgements

We thank Markus Hossner, who worked extremely hard to translate parallelization concepts into working programs, but unfortunately did not want to co-author this paper. We also thank Carsten Schumm of SAP for providing knowhow, the sequential implementation, data sets, computers, and a productive working environment.

References

[1] S. Akhter and J. Roberts. *Multi-Core Programming*. Intel Press, 2006.
[2] B. Chapman, G. Jost, and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press. 2007.
[3] M. Hossner. Lösen großer Transportprobleme durch Parallelisierung auf Multikern-Systemen. Master's thesis, Karlsruhe Institute of Technology, 2009.
[4] IEEE. Standard for Information Technology – Portable Operating System Interface (POSIX). Base definitions, 2004.
[5] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, 1985.
[6] D. J. Meder and W. F. Tichy. Parallelizing an Index Generator for Desktop Search. In: 1st Workshop on Applications for Multi and Many Core Processors (A4MMC), 2010.
[7] V. Pankratius, Chr. Schaefer, A. Jannesari, and W. F. Tichy. Software Engineering for Multicore Systems – An Experience Report. In: 1st Int'l Workshop on Multicore Software Engineering (IWMSE) 2008, pp. 53–60.
[8] V. Pankratius, A. Jannesari, and W. F. Tichy. Parallelizing Bzip: A Case Study in Multicore Software Engineering. In: IEEE Software 26(6):70–77, Nov/Dec 2009.
[9] V. Pankratius, A. Adl-Tabatabai, and F. Otto. Does Transactional Memory Keep Its Promises? Results from an Empirical Study. Technical Report 2009-12, Karlsruhe Institute of Technolgy, September 2009.
[10] J. Reinders. Intel Threading Building Blocks. O'Reilly Media, 2007
[11] P. Toth and V. Daniele. An Overview of Vehicle Routing Problems. In: The Vehicle Routing Problem. SIAM, 2002.
[12] B. Wilkinson and M. Allen. Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers. Second edition. Prentice Hall 2005.

Received: August 13, 2010

Dr. Victor Pankratius heads the Multicore Software Engineering young investigator group at KIT and serves as the elected chairman of the "Software Engineering for Parallel Systems (SEPARS)" international working group. His current research concentrates on how to make parallel programming easier and covers a range of research topics including auto-tuning, language design, debugging, and empirical studies. He is a member of the ACM, IEEE, and GI. Contact him at http://www.victorpankratius.com.

Address: Karlsruhe Institute of Technology (KIT), IPD, Am Fasanengarten 5, D-76131 Karlsruhe, Tel.: +49-721-608-47333, Fax: +49-721-608-47343, e-mail: victor.pankratius@kit.edu

Prof. Walter F. Tichy has been professor of Computer Science at the University Karlsruhe, Germany, since 1986, and was dean of the faculty from 2002 to 2004. Previously, he was senior scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and served six years on the Faculty of Computer Science at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. He is director of the Forschungszentrum Informatik, a technology transfer institute. He is cofounder of ParTec, a company specializing in cluster computing. Dr. Tichy is a member of ACM, GI, and the IEEE Computer Society. Contact him at tichy@kit.edu

Address: Karlsruhe Institute of Technology (KIT), IPD, Am Fasanengarten 5, D-76131 Karlsruhe, Tel.: +49-721-608-43934, Fax: +49-721-608-47343, e-mail: tichy@kit.edu