Abstract

To solve real-world discrete optimization problems approximately metaheuristics such as simulated annealing and other local search methods are commonly used. For large instances of these problems or those with a lot of hard constraints even fast heuristics require a considerable amount of computational time. At the same time, especially for sensitivity analyses, fast response times are necessary in real-world applications. Therefore, to speed up the computation a parallelization of metaheuristics is very desirable.

We present parSA, an object-oriented simulated annealing library based on C++ and using the MPI message passing interface. It provides an automatic, transparent way of parallelizing simulated annealing. The efficient communication in parSA is the main reason for its success in several real-world applications.

To demonstrate performance of parSA we address the weekly fleet assignment problem (FAP) as a real-world application. It is one of the optimization problems which occur in the process of operating an airline. Given a flight schedule and aircraft of different types (subfleets), to each flight leg a subfleet has to be assigned. Large real-world instances have been provided by internationally operating airlines. We show that our heuristic approach using our library parSA is very competitive to the commonly used integer program (IP) approach.

Once implemented, complex algorithms and parallel software components are supposed to be reused in several applications. Thus the effort to set up an optimization system is reduced. In this report, we present a general applicable parallel simulated annealing library. It contains a lot of algorithmic and parallel software elements and can easily be extended.

The fleet assignment problem is one of a series of optimization problems that occur in airline industry operations[7]. The range starts off with market modeling and flight scheduling followed by fleet assignment, crew pairing and crew rostering. The fleet assignment problem is defined as follows. Given a flight schedule and aircraft of different types (subfleets), to each flight leg a subfleet has to be assigned.

Large real-world instances have been provided by internationally operating airlines. We show that our heuristic approach is very competitive to the commonly used linear program (LP) or integer program (IP) approaches. IP-based tools usually try to solve the FAP optimally, but have difficulties to handle additional constraints, such as homogeneity and time windows. The short computation time of our approach of less than half hour on large instances offers the possibility for sensitivity analysis. We parallelize our algorithm with the parallel simulated annealing library (parSA) and are able to reduce the computation time significantly.

This paper is organized as follows: In section 2 we present the main ideas of simulated annealing and its parallelization. Next we introduce in 3 the parallel simulated annealing library parSA. We describe the user-interface, the internal structure and communication protocols used in parSA. Furthermore, we show how to solve a complex real-world problem - the airline fleet assignment problem (FAP). We define in 4 an integer programming (IP) model for the FAP and show how local search can be applied to that problem. We illustrate in section 5 the efficiency of the library by applying it to the FAP.
2 Simulated Annealing

Simulated Annealing (SA) is a heuristic algorithm for solving combinatorial optimization problems. Simulated annealing is based on a local search procedure, and can be viewed as a control strategy for the underlying heuristic search. SA has been shown to be a powerful stochastic search method applicable to a wide range of problems. We refer to Osman and Laporte for a comprehensive review of application reports of simulated annealing [13].

The basic idea in SA is to track a path in the feasible solution space of the given optimization problem. Starting with a valid solution, SA repeatedly generates succeeding solutions using the local search procedure. Some of them are accepted and some will be rejected, according to a pre-defined acceptance rule. The acceptance rule is motivated by an analogy with annealing processes in metallurgy. For a description of this analogy see for example [2]. In the beginning of the optimization process the main control parameter - the temperature - is high and decreases until no improvement of the current solution is attainable. Starting with an arbitrary solution, every improvement is accepted. Deteriorations of the objective function are accepted according to the Boltzmann probability $e^{-\Delta C/T}$. An outline of the basic SA algorithm is given in Figure 1.

After some iterations of the local search procedure, the temperature is decreased and the optimization continues on a new temperature level. The best solution found during the optimization is the output of the algorithm after the system is frozen, i.e. no improvements can be found.

2.1 Strategies of Parallelizing the SA Algorithm

Several approaches have been proposed to implement the simulated annealing algorithm on parallel machines. Recent classification schemes in [3, 5] distinguish between single and multiple-walks (see figure 2), i.e. the number of paths which are evaluated in the search space of the optimization problem. In a single-walk algorithm only a single path in the search space is carried out, whereas in a multiple-walk approach several different paths are evaluated simultaneously. In single-walk algorithms after evaluating a part of the neighborhood of the current solution either only one step is carried out (single-step parallelism) or a sequence starting at the current solution (multiple-step parallelism). In multiple-walk algorithms the parallel walks can be independent or may interact according to a communication pattern.

The parSA library offers two different parallelization approaches: the independent walks parallelization is represented in the Multiple Independent Runs (MIR) parallelization. The clustering parallelization can be classified as both single- and multiple-walk parallelization due to the dynamic structure of the clusters.

3 parSA: A Parallel Library for Simulated Annealing

Our intention was to design a comfortable and efficient parallel simulated annealing framework, which can be applied to many different optimization problems. The use of MPI message passing standard ensures a wide portability to different parallel platforms without redesigning code.

The library is designed in an object-oriented way and is implemented in C++. It can easily be extended by implementing new SA features such as new cooling schedules or different acceptance criteria. Because of a general user-interface many different parallelization schemes of simulated annealing can be implemented. Another important point is the possibility of reusing existing module functionalities.

The parallelization of the simulated annealing algorithm is transparent to the user. The parSA library adapts itself...
to platforms and problems. This behavior ensures high efficiency on different parallel platforms. Applications on real-world problem instances have shown the implemented parallel software library to be highly efficient (see Section 5).

3.1 User-interface of the parSA Library

For the purpose of simulated annealing, the optimization problem should be described in the following way:

- search space $S$: set of all feasible solutions,
- objective function $Obj: S \rightarrow \mathbb{R}^+$,
- local search procedure $Neighbor: S \rightarrow S$

After providing this functionality by the user it is possible to implement a local search based metaheuristic (like SA). Different metaheuristic strategies are implemented in parSA, e.g. sequential or parallel approaches.

The user has to implement three classes with the following functionality:

- **SA_Problem**: input and output of problem data, solution creation and evaluation, searching for a neighbor,
- **SA_Solution**: represents one solution instance,
- **SA_Move**: represents one neighborhood operation.

Each time the parSA has to generate a new neighbor of the current solution, it calls the `GetNeighbor()` function of the user-defined class `SA_Problem` and evaluates this new solution via the function `GetCost()`. After the acceptance decision, either the function `ResetSolution()` or `UpdateSolution()` is called.

Solution exchange functions must be implemented if the the clustering parallelization is used. The user can use an `ostream` to send an solution instance by the library to another processor. But usually, an entire `solution` is too large to be sent when the processors want to communicate. An incremental exchange of `moves` provides a much faster possibility than the above mentioned method. A `move` represents the difference between two consecutive solutions, which are computed by a processor. If possible, `move` exchange functions should be preferred due to performance reasons.

With the basic functionality implemented the user can apply the sequential version of the simulated annealing algorithm, and the Multiple Independent Runs (MIR) parallelization. Solution or move exchange functions are necessary for the clustering parallelization. Details of the Multiple Independent Runs (MIR) parallelization were first presented by Lee in [12]. Basic ideas for the clustering parallelization were introduced by Aarts et.al. in [1] and by Diekmann et.al. in [6]. For the detailed description of the methods implemented in parSA we refer to [11].

3.2 The Structure of the parSA Library

**Searching for a neighbor solution.** The basic task in simulated annealing is the search for a new acceptable solution for the given problem in the neighborhood of the current solution. In the sequential and in the parallel case, this function is implemented in classes which are derived from the basic class `SA_Solver`. The class `SA_Solver` provides functionality which is common for all solvers. In the sequential case, one path in the search space is evaluated, in the parallel case the calculation of several paths is more complex and must be coordinated very carefully. This task belongs to the class `SA_ClusteringSolver` or `SA_MIRSolver`. The solver uses an instance of `SA_Scheduler` to coordinate the acceptance decision during the optimization. The solver also stores the best solution found, which is the final result of the optimization.

**Cooling control.** The main task of the cooling control is the handling of the temperature, which determines the acceptance probability for worther solutions during the optimization. In the beginning of the optimization, a start temperature must be chosen. This calculation is fully adaptive and can be carried out in different ways. After the start temperature is determined, the cooling strategy is applied after each temperature level. A temperature level is completed if an equilibrium in the SA-chain is reached. The computation is aborted if the changes of the objective function are small and larger improvements can not be achieved any more.

Each of the classes in the parSA library can easily be extended if any new features have to be incorporated into the algorithms or parallelization strategies.

3.3 Communication in parSA

In the beginning of the optimization, the temperature is high and therefore the acceptance ratio is high, too (about 90%). Near to the end of the computation the acceptance ratio on processors are low. The processors generate many new solutions which cannot be accepted due to the low temperature. Only 2-5% of generated solutions are accepted in this stage of the algorithm. This causes long `idle times` which can be avoided when generating the solutions in parallel. Clustering parallelization starts with every processor evaluating its individual path in the search space. As the acceptance ratio decreases parallel processors are grouped into clusters. One cluster works on a single subchain in the simulated annealing algorithm. The whole system consists of many clusters which communicate after each temperature level is finished. In the last phase all processors can be grouped together in one large cluster. This motivates the following definitions:
Definition 1 (Cluster) A cluster is a processor group, which has one common actual solution in each SA-step.

Definition 2 (SA-subchain) A SA-subchain is a sequence of actual solutions, evaluated during one temperature level.

Whereas in a sequential SA algorithm a SA-subchain is evaluated by one processor, a cluster evaluates a SA-subchain in the following way:

\begin{align*}
\text{Setup Cluster } (p_1, p_2, \ldots, p_n) \\
L := \text{Actual Solution}() \\
do
\text{Neighbor}(L) := \\
\{\text{Neighbor}(p_1, L), \ldots, \text{Neighbor}(p_n, L)\} \\
\text{Decide about acceptance of } \\
\text{Neighbor}(p_1, L), \ldots, \text{Neighbor}(p_n, L) \\
\text{Choose } L_{\text{new}} \in \text{Neighbor}(L) \text{ as new actual solution} \\
L := L_{\text{new}} \text{ on processors } p_1, p_2, \ldots, p_n \\
\text{until} \text{Equilibrium()} \\
\end{align*}

Figure 3. A simulated annealing subchain, computed by a cluster

\begin{align*}
\text{WarmingUp()} \\
\text{Setup clusters } (C_1, C_2, \ldots, C_m) \\
L := \text{GetInitial Solution}() \\
do
\text{in parallel} \\
\text{Compute SA-subchains} \\
in clusters (C_1, C_2, \ldots, C_m) \\
\text{Synchronize all clusters} \\
\text{Choose one actual solution for all clusters} \\
\text{Make clustering decision} \\
\text{Setup new clusters } (C_1, C_2, \ldots, C_k) \\
\text{DecrementT()} \\
\text{until} \text{Frozen()} \\
\end{align*}

Figure 4. Outline of the clustering algorithm

Communication within a cluster. All processors of a cluster search for an acceptable solution in parallel. If one or more acceptable solutions are found, this event must be communicated between the processors and this new solution must be broadcasted to all processors in the cluster. This can be done using group communication functionality in MPI. In current MPI versions, asynchronous group communication functionality is not available. Therefore asynchronous communication protocols were implemented using basic MPI functions. The performance of the implemented communication protocols is presented in Section 5. The efficiency of the parallel simulated annealing will clearly increase if asynchronous communication is chosen.

Communication between the clusters. After a temperature level is completed, the clusters communicate and one solution is chosen to proceed in all clusters. The clusters communicate and decide whether new clusters should be build or not. In the current implementation, two clusters are always grouped into a larger one. The sizes of the clusters needs not be equal for all clusters. parSA is efficient even if the number of processors in various clusters is different. The communication between the clusters is asynchronous, as well.

Clustering decision. The decision about forming new clusters is based on real-time measurements of the communication and computation speed of the communication platform and processors. parSA adapts its behavior to the given environment. For this reason it is efficient on several tested platforms. A detailed explanation of the methods used for the clustering decision is given in [11].

3.4 Parameter which can be defined by the User of the parSA Library

Users of the parSA library can specify the behavior of the parallel simulated annealing algorithms in several ways. First of all, an appropriate scheduler can be chosen. There is a geometric scheduler with a relatively simple strategy for cooling. A fully adaptive scheduler is provided as well. A sequential optimization algorithm or two different parallelization approaches can be chosen. The most important parameters for the performance of the parallel optimization are the following:

- clustering decision strategy,
- communication mode: synchronous or asynchronous,
- communication of full solutions or only neighborhood moves.
- the rate of SA-chain shortening, which is essential for the attainable solution quality in the parallel case.

After presenting the parSA library, we show an application of parSA on one of the optimization problems from the airline industry. We also present experimental results for different parallel machines and networks on that problem.
4 Application: Fleet Assignment

For operating an airline different optimization problems have to be solved. These include, e.g., market modeling, aircraft and crew scheduling. In this paper, we address the fleet assignment problem (FAP) which turns up in the midterm planning of an airline. Briefly, in an FAP a flight schedule is given, consisting of a list of flight legs with departure and arrival airport, departure time and block/ground time for every subfleet. A type of aircraft (subfleet) has to be assigned to each flight leg while maximizing the profit. Usually in an FAP a time interval of fixed length is planned and then operated rotationally. An often suggested interval length is one day which applies to many large U.S. domestic airlines [9]. For a more irregular schedule the FAP is solved on a weekly basis like by European or international operating airlines which is true for our problems.

The FAP has attracted researchers for many years since operating aircraft is one of the largest expense factors in airlines [4, 8, 9, 14, 15]. A common approach for solving the FAP is to model the FAP as an integer program (IP).

4.1 FAP Model

The FAP is solved for a standard period of a predefined length of $T$ minutes, in our case a week ($T=10080$). An FAP can be viewed as a multi-commodity network flow problem in which the subflights are the different commodities. The network is called time-space network [9]. The vertices of the network are the flight events on the stations (arrivals and departures) dependent on the subflight. A direct flight connection between two airports with its departure time is called a leg. The arrival time of a leg for a certain subflight is defined by adding block and ground time for this subflight to the departure time of the leg. The edges of the network are comprised of the flight arcs and the ground arcs. A flight arc connects a departure and an arrival event of one leg flown by one subflight. The ground arcs connect two subsequent (modulo $T$) flight events on one airport.

The set of legs is denoted by $L$ and the set of available subflights by $F$. The profit of each leg $l$ when flown by subflight $f$ is $p_{l,f}$. The set of flight events $V$ are the vertices of the network. $v \in V$ is described by a triple $(t, s, f)$ where $t \in [0, \ldots, T-1]$ is the time of arrival (or departure), $s$ is the airport of arrival (or departure), and $f$ is a subflight that can fly the leg corresponding to this flight event. The set of edges is comprised by the flight arcs $E_F$ and the ground arcs $E_G$. A flight arc represents a leg $l$ that is flown by subflight $f$. For $v = (t, s, f) \in V$ the subsequent (preceding) modulo $T$ flight event on the same airport of the same subflight is denoted by $v^+$ ($v^-$). A ground arc $(v, v^+)$ combines the flight events $v$ and $v^+$ and $y_{v,v^+}$ counts the number of aircraft of subflight $f$ between $v$ and $v^+$ residing on airport $s$. The decision variable $x_{l,f}$ is set to 1, if subflight $f$ is assigned to leg $l$, and to 0 otherwise. The (arrival) event of leg $l$ flown by subflight $f$ is denoted by $l_{f}^{arr}$ ($l_{f}^{dep}$). $size_f$ denotes the maximal number of aircraft available for each subflight $f \in F$. The set of flight arcs that cut the count time ($t = 0$, w.l.o.g.) are denoted by $E_{F0}$ and the set of ground arcs which cut the count time are $E_{G0}$. Now, the Fleet Assignment Problem can be described as:

\[
\text{maximize} \sum_{i \in L} \sum_{f \in F} p_{l,f} \cdot x_{l,f} \\
\text{subject to} \sum_{f \in F} x_{l,f} = 1 \quad \forall l \in L \quad (1) \\
\sum_{i \in V} y_{v,v^+} \cdot x_{l,f} = 0 \quad \forall v \in V \quad (2) \\
\sum_{(i,l) \in E_F} x_{l,f} \leq size_f \quad \forall f \in F \quad (3) \\
x_{l,f} \in \{0,1\} \quad \forall l \in L; \forall f \in F \quad (4) \\
y_{v,v^+} \geq 0 \quad \forall (v,v^+) \in E_G \quad (5)
\]

The first set of constraints (1) ensure that exactly one subflight is assigned to each leg. The constraints described in (2) guarantee the flow conservation in the network and the constraints denoted under (3) restrict the number of aircraft per subflight to be used. Since the decision variables $x_{l,f}$ are in $\{0, 1\}$ (4) and because of (2), the variables $y_{v,v^+}$ are integral and therefore only constraints (5) are needed.

Since flight schedules evolve over time, solving the FAP does not have to start from scratch. Changes to a flight schedule are integrated into the old schedule by the airline experts, so that for a real-world fleet assignment instance an initial solution is given which can be used by the SA.

The local search procedure generating neighbored solutions for a given FAP solution is the problem specific part of the optimization system based on parSA. The details about the neighborhood structure are out the scope of this paper. Briefly, we call $(l_0, \ldots, l_k)$ a leg sequence $S$, if the arrival airport of $l_i$ is the departure airport of leg $l_{i+1}$ for all $i = 0, \ldots, k-1$ and all legs $l_0, \ldots, l_k$ are assigned the same subflight. Leg sequences are generated by a limited depth first search. The neighborhood transition alters the assigned subflight for one or more leg sequences.

5 Experimental Results

In this section we present the performance of the sequential version of the FAP algorithm and the performance of the parSA library applied to the fleet assignment problem. The experimental results were obtained on parallel systems of the Paderborn Center for Parallel Computing and on the workstation cluster of the Department of Computer Science of University of Paderborn.
The instances we solved are weekly problems of recent summer and winter schedules provided by two large internationally operating airlines. The problem sizes are shown in Table 1. The objective function is composed of the expected revenues, the fixed costs per flown leg, and the expected passenger-related costs.

<table>
<thead>
<tr>
<th>FAP</th>
<th>legs</th>
<th>subfleets</th>
<th>airports</th>
<th>SA average solution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3337</td>
<td>4</td>
<td>68</td>
<td>3m20s</td>
</tr>
<tr>
<td>B</td>
<td>3911</td>
<td>10</td>
<td>63</td>
<td>12m40s</td>
</tr>
<tr>
<td>C</td>
<td>4558</td>
<td>8</td>
<td>75</td>
<td>17m20s</td>
</tr>
<tr>
<td>D</td>
<td>4616</td>
<td>11</td>
<td>75</td>
<td>25m30s</td>
</tr>
<tr>
<td>E</td>
<td>4311</td>
<td>11</td>
<td>74</td>
<td>24m40s</td>
</tr>
<tr>
<td>F</td>
<td>3389</td>
<td>9</td>
<td>73</td>
<td>16m10s</td>
</tr>
</tbody>
</table>

Table 1. FA problem sizes and initial solution provided by the airlines, results of the SA algorithm, upper bound (UB) with the reduced number of aircraft used by the SA solution, upper bound (UB') with the number of aircraft used by the initial solution. Due to confidentiality only percentage results on the objective (profit) can be shown.

For measuring the solution quality of our algorithm we computed an upper bound on the objective function by relaxing the integrality of the decision variables $x_{i,j}$. Solving the linear program was carried out by CPLEX 6.0. Since our SA algorithm tries to reduce the number of subfleets where possible, the upper bound is also computed for the same number of subfleets used by the SA algorithm. Objective values of the SA algorithm and the upper bounds are listed in Table 1. The upper bound UB (linear program solution) cannot directly be used for generating a valid integer solution. A full integer programming approach for the instances mentioned in Table 1 will run for several hours which is significantly more than 10–30 minutes as in our approach. Moreover we can even solve larger instances in a reasonable amount of time (up to 30000 legs).

### 5.1 Parallel Results: Speedup & Solution Quality

The SCI cluster used in parallel tests with parSA is a Siemens hpcLine system. The system utilizes 96 double-processor Intel Pentium II (450 MHz) boards. The cluster uses the Scalable Coherent Interface (SCI) interconnect standard. The implementation of the MPI standard is realized through the global shared space provided by the SCI. The implementation is high performant with a latency of 6 micro seconds and a bandwidth of 70 Mbytes/sec.

The parallel results for the fleet assignment problem are presented in Figure 5. The two first plots show the efficiency of the parallelization and the third one the obtained solution quality (the 1 in the plot represents the value of the upper bound on the objective function calculated by an LP solver). The application scales well up to 16 processors. Further improvement is not easy to achieve because of very short computation times: It was reduced from 30 minutes in the sequential case to 2-5 minutes on 16 processors. The efficiency of parSA will be even higher if the computation times are considerable greater, i.e. in the case of the fully dated fleet assignment problem. This extension from a weekly model to a period of six weeks brings a substantial increase in the complexity of the problem. That is where the parSA comes into its own. Extending the FA model to the fully dated FAP is a part of the ongoing work.

In the first setting three different clustering strategies are compared. The standard strategy maximizes the product of cluster speedup and cluster efficiency. The other strategies maximize only the cluster speedup (speedup) or decide to cluster when the cluster efficiency is at least 1/2 (efficiency). One can observe that the standard clustering strategy balances the both values in a reasonable way and achieves the best results in this experiment.

The second experiment compares different communication modes in the parallel version of the simulated annealing algorithm: full solutions or only neighborhood moves (standard setting). The communication of moves is slightly more efficient because of smaller amount of data which is communicated between the nodes. The long chains setting results in a lower efficiency but significant higher solution quality in the parallel case. The reason for this effect is the better convergence of SA in this case.

Note that modeling the real-world FAP involves some inaccuracies, e.g., estimated number of passengers which is again dependent on the assigned subfleets. In practice, an optimal solution to an FAP is not necessarily of more value than a high quality approximative solution. The solution quality in the parallel case is not always as high as in the sequential case. But for the mentioned reasons the gain in the running time outweighs the loss in the solution quality.

### 6 Conclusion

We have presented a general parallel simulated annealing library parSA which is publicly available [10] for research purposes. The user-interface of the library is clear and flexible. The generality of the metaheuristic simulated annealing was kept. The interface is rather intuitive and easy to implement. The clear distinction between problem-independent and problem-specific constructions of the SA-algorithm makes the library generally applicable. Because of the exactly modeled structure of the simulated algorithm existing implementation of the algorithm can be eas-
As an application we presented a parSA-based heuristic algorithm to solve the airline fleet assignment problem. This module is now in use in a production planning system of a large internationally operating airline. In real-world applications like this the library provides the possibility to reduce the problem solution times significantly by the use of parallel system power. Moreover we can even solve larger instances in a reasonable amount of time. This leads to an interactive decision support tool for optimization problems which used to take hours to compute.

References