Parallel Simulated Annealing Library (parSA)
User Manual

Georg Kliewer, Karsten Klohs

Version 2.2

Contents

1 Introduction

1.1 Structure

This manual consists of three basic chapters and one reference chapter. Chapter 3 introduces the basic concepts and the theoretical background on which the parSA library is based. The following chapter 4 regards the parSA library as a software tool and provides instructions for installation and implementation of the user application. Chapter 5 describes the structure of the parSA library and the way it works.

1.2 How to use this Manual

We suppose that the parSA library is requested by users with a number of different aims. In this chapter we therefore suggest different ways to use this manual.

Pure Users – If you only want to solve an optimization problem with the parSA library, or if you want an easy way to parallelize the SA algorithm you should concentrate on chapter 4. However without any knowledge about the functionality of the library and the SA algorithm it may be difficult to find the parameters, which provide the best solution to your very special problem. If the initial installation of the library is successful it will be useful to read chapters 3 and 5 also.

Advanced Users – If you already know the SA algorithm and if you are familiar with its modifications chapter 5 will describe the possibilities that are provided by the parSA library and how they are used. If the parSA library suits your needs you may read chapter 4 for further information about the installation of the library.

Scientist And Developers – Everyone who is interested in a comprehensive overview is recommended to beginning in the first and ending in the last chapter. The explanations of chapter 3 concentrate on the theoretical results, that are important for the functionality of the library and do not claim to be complete. Nevertheless, this chapter is suited for a first understanding of the SA algorithm and its modifications. Chapters 4 and 5 will ensure an efficient use of the library.

2 Changes Since the Previous Versions

2.1 Library Version 2.2

SA_TimeScheduler This new scheduler was implemented to examine the behaviour of the geometric schedule under strong time restrictions.

2.2 Library Version 2.1

Full LINUX Support From this version on we provide an additional LINUX version of our library.

Restructured Scheduler Parameters The structure of the scheduler parameters have sleighly changed. Some parameters have been moved from the derivated classes to the SA_Scheduler base class and one important new parameter is introduced. The parameter *timelimit* causes all of the current schedulers to freeze if the specified timelimit has elapsed. This enables you to terminate every annealing process after a defined time. However this time represents pure annealing time.

Output The output of the parSA has become very flexible but all you have to keep in mind is that from now on a file named SA_Output.rsc has to be situated in the starting directory of the annealing program and that you may specify a so called verbose level for every class that controls the amount of the output produced.

3 Theoretical Aspects

In this chapter the Simulated Annealing Meta Heuristic (SA) is introduced. The first section describes the basic method. Different cooling schedules, that are supported by the parSA library, are introduced in the second section. The last section discusses the possibilities of parallelizing the basic method.

3.1 Simulated Annealing Meta Heuristic

The Simulated Annealing Meta Heuristic (SA) can be regarded as a variant of the traditional technique of local neighborhood search. Suppose we have a minimization problem over a set of feasible solutions S and a cost function $f : S \longrightarrow R$, which can be calculated for all $s \in S$. An optimal solution can be obtained be calculating $f(s)$ for all $s \in S$ and selecting the minimum.

Usually the set S will be far too big and therefore the technique of local optimization defines a neighborhood structure N on the set S and searches only a small subset of the solution space by confining the search for an improvement of the cost function to the neighborhood of the current solution. If no better neighbor is found the current solution is regarded as an approximation of the optimum. This technique often results in convergence to a local rather than a global minimum.

The main idea of SA is to provide a possibility of escaping a local minimum by accepting even an increase in the cost function. This acceptance depends on a control parameter (temperature) and the magnitude of the increase. The algorithm can be stated as follows:

$L := GetInitialSolution()$
$T := WarmingUp()$
do
    do
        $L_1 := Neighbor(L)$
        $\Delta C := Cost(L_1) - Cost(L)$
        if $\Delta C < 0$ or $Accept(\Delta C, T)$
            $L := L_1$
    until $Equilibrium()$
    $T := DecrementT()$
until $Frozen()$

Figure 1: The SA algorithm in pseudo code

The algorithm given above is very general and for the solution of a particular problem two categories of decisions has to be made. Primarily there are generic decisions which are concerned with parameters of the SA algorithm itself. These include factors such as the choice of the initial temperature ($WarmingUp()$), the cooling schedule (governed by the functions $Equilibrium()$ and $DecrementT()$)

and the stopping condition ($Frozen()$). The second class of decisions is problem specific and involves the space of feasible solutions (representation of L), the cost function ($Cost()$) and the neighborhood function ($Neighbor()$).

3.1.1 Generic Decisions

$WarmingUp()$ – The choice of the initial temperature should guarantee that almost every change of the solution is accepted at the beginning of the annealing process.This ensures that the process does not depend on the initial solution.

$Accept()$ – This is the criterion of accepting a worse solution than the actual one. Normally the Boltzmann-distribution is chosen:

$$Accept(\Delta C, T) \iff r < e^{-\frac{\Delta C}{T}} \quad \text{for a randomly chosen } r \in [0,1].$$

$Equilibrium()$ – The $Equilibrium()$ function determines the number of iterations that are made before the temperature is reduced.

$DecrementT()$ – The rate at which the temperature is reduced. It can be determined either in a geometric ($T_n := \alpha T_{n-1}$) or adaptive way. The adaptive cooling schedules use a feedback from the annealing process to find a convenient cooling rate.

$Frozen()$ – This is the stopping criterion of the algorithm. One can choose a certain number of steps or a certain acceptance ratio for example.

These functions characterize a cooling schedule and their parameter settings can be combined in many ways. However, not every combination provides a good solution.

3.1.2 Problem Specific Decisions

The problem specific decisions are concerned with the solution space, neighborhood structure and the cost function. Generally, it is not possible to define best choices for a given problem. Nevertheless, there are three main goals that have to be achieved. The validity of the algorithm has to be maintained, the computation time has to be used in the most efficient way and the solution should be close to the global optimum. It has been shown that every solution has to be reachable from every other, which is usually easy to verify.
In order to use the computation time most efficiently the frequently-used functions like creating a random neighbor and determining the cost of a solution should be as fast as possible. For example it is often not necessary to recalculate the complete cost function. As well it is often not necessary to communicate by sending whole solutions rather than solution changes when you are working in parallel. In order to support this more efficient communication strategy the parSA-library provides the class move which can be additionally implemented by the user.
It is also suggested that it is prudent to avoid neighborhoods which represent a

spiky topography or deep troughs in the solution space. It is also obvious that the size of the solution space and neighborhoods should be kept reasonably small.

3.2 Supported Cooling Schedules

The parSA library provides a number of different cooling schedules which had shown a good performance on large sized real world problems. Some of these cooling schedules are adaptive. This means that the reduction of the temperature depends on the current SA run which leads usually to a better performance while the determination of the behavior of the algorithm becomes more complex. The MIRScheduler and the AartsScheduler are the adaptive schedules that have currently be implemented in the parSA library.

3.2.1 Geometric Scheduler

The geometric scheduler is a rather simple and frequently used cooling schedule. It is implemented in the class SA_EasyScheduler which also provides an improved a more flexible warming up strategy.

WarmingUp(): The initial temperature is set to an user defined value and the length of a subchain is fixed.

Equilibrium() and DecrementT(): After the required number of iterations has been made the temperature is reduced by a constant factor α according to:

$$T_n := \alpha T_{n-1} \text{ with } 0 < \alpha < 1$$

Frozen(): The algorithm terminates when the average acceptance ratio is lower than a fixed acceptance ratio χ_{min} for a fixed number k of temperature steps.

Since every parameter has to be set by the user some test runs are necessary to find suitable parameter settings for a given problem.

3.2.2 Aarts Scheduler[1]

WarmingUp(): An initial acceptance ratio χ_0 is set and an initial temperature T_0 is chosen that approximately provides this acceptance ratio. In order to achieve this, the temperature is set to zero at the beginning and m_0 iterations are made where m_0 is the average number of neighbors of a solution. After each iteration step the temperature is updated according to the following rule:

$$T = \overline{\Delta C^{(+)}} \left(\ln \frac{m_2}{m_2 \chi_0 - (1 - \chi_0) m_1} \right)^{-1}$$

with m_1 and m_2 being the better or respectively worse neighbors. $\overline{\Delta C^{(+)}}$ is the mean value of the differences between the cost function of all worse solutions. After m_0 steps the initial temperature T_0 is set to T.

Equilibrium() and DecrementT(): The length of a subchain with constant temperature is set to the number of the local neighborhood. After this number of iterations the temperature is reduced to

$$T_n = T_{n-1}\left(1 + \frac{\ln(1+\delta)T_{n-1}}{3\sigma T_{n-1}}\right)^{-1}$$

where $\sigma(T_{n-1})$ is the standard deviation of the values of the cost function at the current temperature and δ is the so called *distanceparameter*. The size of δ determines the speed of the reduction of the temperature. Aarts [1] suggests the value $\delta = 0.1$.
Because of the constant length of the subchains this cooling schedule is a suitable choice for the clustering parallelization.

Frozen(): The algorithm terminates when the mean value of the cost function shows only very small changes or respectively the derivation of the smoothed mean value is smaller than the set value ϵ.

If the neighborhood size is polynomial in the size of variables of the optimization problem it has been shown that the SA algorithm has a polynomial time complexity with this cooling schedule.

3.2.3 MIRScheduler

The SA_MIRScheduler represents a cooling schedule which is used by the MIR strategies. Therefore, it has to be used in combination with the SA_MIRSolver. Fu-Hsieng Allisen Lee [2] introduced the following strategy: The whole SA run has a fixed length. This leads to an important property: a time limit can be set after that the algorithm must have calculated a solution.

WarmingUp(): The choice of the initial temperature is similar to Aarts strategy which provides an initial acceptance ratio χ_0 close to 1. Thus, a sequence of solutions is generated and differences in their cost values are recorded. The maximum ΔC_{max} and the minimum ΔC_{min} are used to determine the start and end temperature.

$$\begin{aligned} T_{start} &= -\frac{\Delta C_{max}}{\ln \chi_0} \\ T_{end} &= -\frac{\Delta C_{min}}{\ln \chi_0} \end{aligned}$$

DecrementT(): The temperature is reduced by a constant factor α similar to the reduction used by the geometric schedule.

Equilibrium() and Frozen(): After the calculation of T_{start}, T_{end} and the temperature reduction factor it is simple to determine the length of a subchain from the overall number of available iterations. The parSA library supports slightly different calculations but the algorithm always ends after the calculable number of iterations.

3.3 Parallelizing Simulated Annealing

The parallelization strategy of the parSA library is controlled by the class SA_Solver. The SA_ClusteringSolver and the SA_MIRSolver are currently implemented in the library. The SA_ClusteringSolver can be used in combination with the SA_AartsScheduler and the SA_EasyScheduler while the SA_MIRSolver requires the SA_MIRScheduler.

3.3.1 Clustering Parallelization

The clustering parallelization is based on the following fact: the number of neighbors that has to be visited before a move is accepted increases when the temperature sinks. Therefore, the current solution remains unchanged for many iterations when the temperature is low. The time that is spend with the same current solution can then be reduced by using more than one processor. Such a processor group is called a cluster. Each cluster works on one single subchain. Every processor particularly has the same actual solution. If a move is accepted by the cluster the new actual solution has to be broadcasted in the cluster.
It is obvious that the use of several processors is only reasonable if the reduction of the latency is bigger than the communication overhead. That is why the clustering moment is essential for this kind of parallelization. The general strategy for N processors is the following:

1. Every single processor forms a cluster with size 1.
2. Every cluster calculates its own subchain. The length of such a chain depends on the number of clusters.
3. When the calculation at one temperature step is finished, a new solution is selected from the actual solutions. This solution becomes the first solution for the next subchain in every cluster.
4. If necessary the cluster size is increased.
5. The temperature reduction and the termination is similar to the sequential algorithm.

The reduction of the subchain length in step 2 is fairly important because otherwise the calculation of a single subchain is independent of the number of processors. This would slow down the algorithm dramatically. This reduction is possible because the reaching of the equilibrium at high temperatures is not disturbed by the fact that the required number of iterations are not made in one but several subchains.

3.3.2 Multiple Independent Runs

There are two facts that lead to the idea of the multiple independent runs parallelization.

- The best solution of several SA runs usually provides a better quality of the solution than the solution of a sequential run with the same length.

- Is is possible to estimate the achieveable solution quality for a given length of a run.

Our aim is to provide an algorithm that achieves a given solution quality in the shortest possible time. Therefore, we have to determine the length and the number of runs. An SA run in which the generated chain has the length n is called a run of length n. This terminology is used when speaking of a short run, multiple independent runs or runs with duplicate length.
It has been show that the convergence speed can be calculated according to this formula:

$$P(X_n \notin Cost_{min}) \sim \left(\frac{K}{n}\right)^{\alpha}$$

K and α are constants specific to the problem, X_n is the solution of a run with length n and $Cost_{min}$ is the set of solutions with an appropriate quality. Empirically a rather similar formula has been found for the relation between the solution quality and the run length. Only the problem specific constants differ slightly from the ones above. Therefore, theoretical results concerning the convergence speed can be used to provide an improved solution quality with the same run length choice.
There are three different strategies for improving the performance of the algorithm by the choice of independent runs on parallel systems:

1. Improving the convergence speed with the same number of iterations.
2. Maintaining the same convergence speed with a smaller number of iterations.
3. Maintaining the same convergence speed as the sequential algorithm.

The following values can be proven for the mentioned different strategies:

strategy	total iterations	number of runs	run length	convergence
sequential	N	1	N	$(\frac{K}{N})^{\alpha}$
1	N	$\frac{N}{eK}$	eK	$e^{-N\frac{\alpha}{eK}}$
2	$eK \ln \frac{N}{K}$	$\ln \frac{N}{K}$	eK	$(\frac{K}{N})^{\alpha}$
3	$\frac{N}{e}$	$\frac{N}{eK}$	K	$(\frac{K}{N})^{\alpha}$

4 Installation Guide

4.1 Instructions for Installation

4.1.1 Conventions

The parSA library can be installed on a large number of different systems and architectures. Therefore, we make the following conventions to facilitate the language of this section:

shortcut	description
PARSADIR	directory which contains the **parSA package**
MPIINCDIR	directory which contains the mpi headerfile **mpi.h**
MPILIBDIR	directory which contains the libraries **libmpi.a lib**

4.1.2 Requesting the Packages

Since the parSA library is primarily designed for the use on parallel architectures an installation of the MPI library is needed. If it is not installed on your system you may visit the following webpage for further information:

http://www.mcs.anl.gov/Projects/mpi/index.html

The latest version of the parSA library can be requested on our homepage at

http://www.uni-paderborn.de/˜parsa

or contact us via email :

parsa@uni.paderborn.de

You have received a zip-file called parsa.zip. Place this file in a new directory.

4.1.3 Installation of the Library

Now unzip the file using the command

unzip parSA.zip

and check if all files specified in section 7.3 have been properly installed. The structure of the package is very simple:

- The subdirectories *lib* and *include* contain the necessary header files and version of the library itself. The naming convention of the library is libparSA$(OPERATINGSYSTEM)$(OS_VERSION).a. If you have received the developer version an additional subdirectory named *src* will exiist where the source files of the parSA reside.

- The subdirectory *example* contains the example program which is detailly described later.
- Some useful configuration file template are situated in the subdirectory *cfg* and and *doc* is the place of this document.

4.2 Compiling the Example Program

Compiling and running the example program should assure you that the parSA package is fully functional. Altough a simple makefile is included we will describe the compilation process more detailled because of the amount of different operating systems and environments on which the parSA meight be used.
First of all change the working directory to:
cd PARSA/example

Now compile and link the sourcefile example.cc to produce an executable called example. Please take the following instructions into account:

compiler: use a C++-compiler.

include directories: make sure that the directories MPIINCDIR, PARSADIR/example and PARSADIR/include are searched for headerfiles.

sources: compile both example.cc and SA_Problem.cc (the source file SA_Problem.cc contains the implementation of the interface between example and the parSA library).

object files: you will find the object files example.o and SA_Problem.o in the directory PARSA/example assumed that the compilation was successful.

library directories: make sure that the directories MPILIBDIR, PARSADIR/libs and the directory of your standard C++ libraries are searched by your linker.

libraries: apart from the standard C++ Libraries you will have to link the libraries libmpi.a, libsocket.a, libnsl.a (supported by the MPI package) and the library libparSAxxx.a (which can be found in the directory PARSADIR/libs)

Suppose you are working with a Sparc Sun Solaris Microcomputer, then the following command works properly:

g++ -I. -I../include -IMPIINCDIR SA_Problem.cc example.cc -LPARSADIR/libs -LMPILIBDIR -L/local/gnu/lib -lparSA_solaris_2_5 -lmpi -lsocket -lnsl -o example

You should now be able to run the created executable with the following command:

mpirun -np 1 example

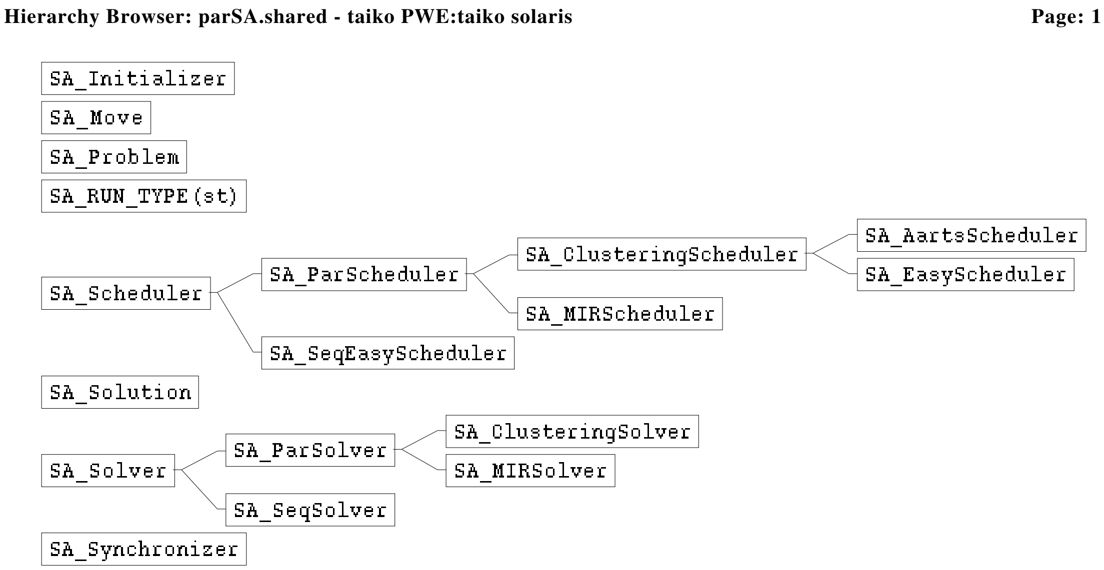
If the program has terminated without any error messages the installation of the library files should have been successful. If you want to have a closer look on the example you have just produced see section 6. Please notice that the parSA prints its source version and revision number before the SA_Initializer starts configuring the solver. Please indicate this number when you contact us.

5 Design Philosophy of the parSA Library

The parSA library was designed to provide a comfortable and efficient parallel framework for a simulated annealing optimization system, which can be applied to many different optimization problems. The MPI message passing interface and the use of C++ ensures that the library is portable to different parallel platforms without redesigning the code.
The object oriented design keeps the library expandable. It is fairly easy to create new solver and scheduler classes to increase the number of parallelization and cooling strategies of the library.
This section introduces the basic concepts of the parSA library. It gives a brief overview of the structure of the library and describes the main ideas of the application interface and the SA configuration file. This configuration file is used to adapt the parameters of a SA run to the problem.

5.1 Structure

The following figure shows the most important classes, that are currently implemented in the library :



There are two aspects that influence the annealing process. First of all the problem that must be solved with the library has to be modelled. This modelling has to be done by the parSA user by the implementation of the so called interface classes *SA_Problem, SA_Solution* and *SA_Move* which are all defined in the files *SA_Problem.h* and *SA_Problem.cc.* These classes are more precisely described in section 5.4. The other aspect is the configuration of the annealing process itself. A very flexible configuration is the main advantage of the parSA library, because you can not only change the characteristic parameters of one single cooling scheme but also choose between many different cooling startegies. Moreover you can simply use simualated annealing on many processors just by selecting another solver and so speed up the calculation significantly. This configuration is done by the settings in the configuration file which will be introduced in section 5.6 The following section will concentrate on the description of the base classes SA_Solver and SA_Scheduler and their derivated classes. The classes SA_ParScheduler and SA_ParSolver were used as base classes for all combinations of solver and scheduler that can work in parallel. In contrast to this classes the classes SA_SeqScheduler and SA_SeqSolver do not need an implementation of the MPI. The pure sequential version of the parSA library does only provide the derivatives of these classes.

5.2 Solver Classes

The solver classes control the organization of the SA algorithm. Currently, the parSA-library provides the following solver:

SA_Solver: This is an abstract base class used to derivate the base classes of both the sequnetial and the parallel branch of the solver hierarchy. Moreover this class is capable of setting up the data interface of the parSA library by opening streams for reading the datafile and writing outputfiles which contain information about a single simulated annealing run and about the best solution found during the optimization.

SA_SeqSolver: This solver is designed to serve as the basic solver from which any other sequential solver should be derivated. It provides the minimal functionality for controling sequential SA runs and is able to produce some statistical information. This Solver can be used in combination with the SA_SeqEasyScheduler.

SA_ParSolver: This solver is the parallel analogon of the SA_Solver. It extends the capabilities of the sequential version by providing statistical methods for a SA run on more than one processor.

SA_MIRSolver: This solver is used for the MIR parallelization of the SA algorithm. Because of the particularities of the parallelization strategy it is necessary to use the **MIRScheduler** in combination with this solver.

SA_ClusteringSolver: This solver uses the clustering parallelization and can be either used with the SA_EasyScheduler or the adaptive SA_AartsScheduler.

5.3 Scheduler Classes

The class SA_Scheduler is the abstract base class for all cooling schedules, that are used during the SA process. The different cooling strategies have an important influence on the efficiency of the SA algorithm. Currently, the following schedulers are implemented:

SA_EasyScheduler: This scheduler reduces the temperature according to a constant factor. It had shown a rather good performance on many different problem instances.

SA_AartsScheduler: This scheduler tries to adapt the temperature reduction to a certain problem instance. It also calculates automatically a suitable start temperature. Its strategy of finding this start temperature is also used by the GeometricScheduler if a start temperature is not specified.

SA_TimeScheduler: This scheduler was designed to take a given timelimit into consideration. There are two different startegies implemented in this scheduler. The first one tries to achieve a given end temperature within the timelimit. This is done by adapting the subchainlength of the cooling schedule of the SA_EasyScheduler.

The second strategy which is implemented in this scheduler tries to achieve a given solution quality in a given time. If the timelimit expires and the solution quality is not reached than the scheduler freezes otherwise it freezes at the given solution quality. Up to now this strategy has not been analysed.

SA_MIRScheduler: This is a specialized scheduler which has to be used when using the MIR parallelization of the SA algorithm.

5.4 The Application Interface

The class SA_Problem must be used by the application to set up a representation of the solution space. Every kind of solver uses an instance of SA_Problem to find an at least locally optimal solution with its own SA strategy. Therefore this class must provide elementary methods like finding a neighbor of an actual solution or determining the cost of a solution. The class SA_Solution represents one single element of the solution space and is mainly used by the class SA_Problem. Finally the class SA_Move represents just the change between an solution and its neigbour. In many solution spaces such a move can be represented more efficiently than a whole solution. When working in parallel the sending of moves rather than solutions may reduce the communication costs significantly.

This section describes, how an interface to the parSA library is usually implemented. It is divided into subsections dealing with the implementation of the problem specific classes SA_Problem and SA_Solution and also the class SA_Initializer, which is used to invoke the annealing process.

5.4.1 The Class SA_Solution

The class SA_Solution represents one single point of the solution space. In most implementations this class contains information about the solutions itself about the change that was made last and perhaps about the neighborhood of the solution. Nevertheless the funcionality working on this information especially the search of a new neighbor and the updating or resetting the solution was integrated in the class SA_Problem to keep the problem description as flexible as possible. Therefore only the method SA_Solution.Copy(S) and the iostram functions are called by the library and must be implemented by the user.

5.4.2 The Class SA_Move

During the annealing process the cost of the actual solution is compared very often to the cost of its neighbors. In many cases neighboring solutions only slightly differ, and so when working in parallel the communication costs may be reduced very much by sending only this difference to another node. In the parSA library the description of such a difference between neighbors is called a move. As one can even imagine problems where the size of a move remains constant the advantages of the communication by moves becomes rather obvious. If you want to use this parSA feature you will not only have to implement the class *SA_Solution* but also *SA_Move*. However *SA_Move* is only a container and the implementation of some additional methods of *SA_Problem* is also required.

5.4.3 The Class SA_Problem

The class SA_Problem is a representation of the solution space of the optimization problem. The class has to provide three basic methods which are essential for the annealing process. They are:

SA_Problem.GetInitialSolution(S): This method creates a primary solution S, which is the starting point of the optimization process. This method is supposed to be deterministic. Its non deterministic counterpart is the method SA_Problem.GetRandomSolution(S).

SA_Problem.GetCost(S): This method calculates the cost of a given solution S. The cost value is used to compare the quality of a new solution to an older one.

SA_Problem.GetNeighbor(S): This method determines (small) changes of a given solution S. This change should be only temporarily, because the method SA_Problem.ResetSolution() is used to discard the last change.

Moreover the class SA_Problem has to provide some other methods which also depend on the representation of the solution space:

SA_Problem.ResetSolution(S): retrieves the former solution by discarding the change made by GetNeighbor().

SA_Problem.UpdateSolution(S): makes the change made by GetNeighbor() permanent.

SA_Problem.CreateSolution(): creates a new instance of the representation of the solution.

SA_Problem.GetLocalN(): determines an estimation of the average size of the neighborhood of a point in the solution space. Even if your optimization problem is not discrete a value is required because the ratio between GlobalN and LocalN is used by some cooling schedules.

SA_Problem.GetGlobalN(): determines the approximated number of possible solutions.

SA_Problem.ReadProblemData(): should be used to read the information from a data file which is necessary to build up a problem representation.

SA_Problem.Copy(S,S): is used by the library to copy one solution into another.

If you want to use the parallelization capabilities of the library you will at least have to implement the following methods of the class SA_Problem, too:

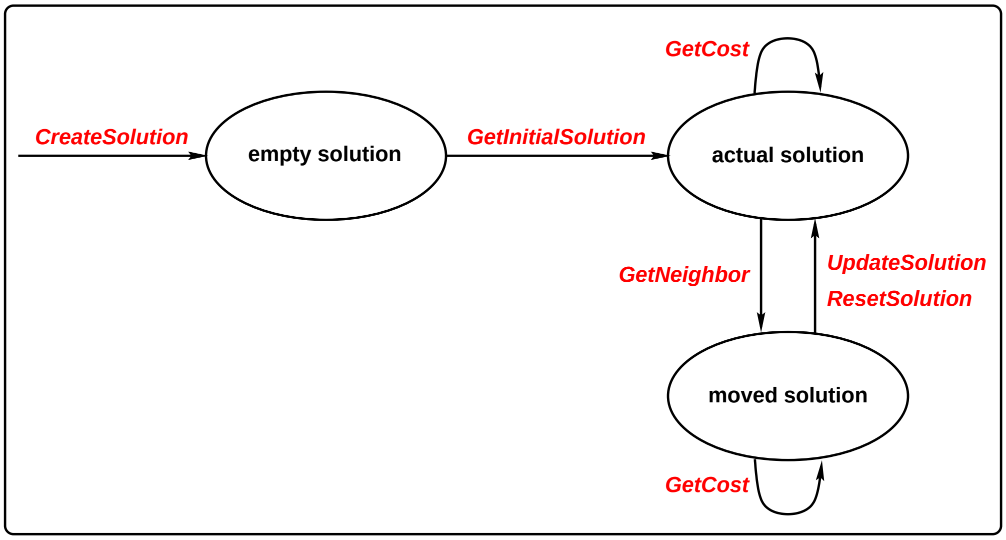
OutputSolution(ostream,S): writes the solution S to an ostream

InputSolution(istream,S): reads a solution S from an istream

All of the described methods must be implemented to provide a minimal functionality. The additional methods can be divided into three sections. The most important are the ones which are needed to create moves, extract moves from solutions and to realise the move communication. This communication methods are rather similar to the solution communication methods. The second section of additional methods are special communication methods which may be used to implement an own MPI based communication. The last section are special methods which where introduced to support special applications.

5.4.4 The Solution Life Cycle

The following graphic explains how all these methods are used within the annealing process:



With *CreateSolution* a new empty solution is generated. *GetInitialSolution* produces a feasible solution (in a deterministic manner). For each solution, the value of the cost function may be evaluated and *GetNeighbor* returns a neighbor of the actual solution. The cost of this so called moved solution is calculated. According to the actual annealing situation this new solution is either accepted or rejected, which is done by the methods *UpdateSolution* or *ResetSolution*.

5.5 The Class SA_Initializer

The class SA_Initializer is used to intialize an special solver and to launch the annealing process. In order to understand the use of the class SA_Initializer, have a look at the following simple main program:

```
#include <SA_Problem.h>
#include <SA_Initializer.h>

main(int argc,char **argv)
{
        SA_Problem p;

        SA_Initializer start;

        SA_Solver *sp = start.ReadConfigFile(argc,argv,p);
        if ( sp != NULL )
        {
                sp->RunAnnealing();
                delete sp;
        }
}
```

After you have declared an instance of the class SA_Initializer you will only have to invoke the method ReadConfigFile. This methods receives the parameters of the command line and an instance of the class SA_Problem. The instance of the class SA_Problem will be initialized after the initializer has read the SA.cfg file by invoking the method SA_Problem.ReadProblemData, with the filename specified by the keyword datafilename in the configuration file. The method SA_Initializer.ReadConfigFile() will also automatically initialize the chosen solver depending on the entries of the SA.cfg file.
After your program has received the initialized solver the annealing process is simply started by calling its method RunAnnealing() from the recieved SA_Solver instance.

5.6 The Configuration File

The configuration file, which is usually named SA.cfg, is the main tool to control the library and to adapt the parameters to special problems. The following sections describe the concept of the configuration file and explain a rather simple example. The default name of the configuration file is SA.cfg, but you may alter the name by using the command line option -cfg "filename" when starting a program that uses the parSA library.

5.6.1 The General Structure

The following figure presents the general structure of such a configuration file:

```
{
    SA_Solver NAME_OF_SOLVER {
        SA_Solver {
            SETTINGS_OF_SOLVER_BASE_CLASS
        }
        SA_Scheduler NAME_OF_SCHEDULER {
            SA_Scheduler {
                SETTINGS_OF_SCHEDULER_BASE_CLASS
            }
            SETTINGS_OF_DERIVED_SCHEDULER_CLASSES
        }
        SETTINGS_OF_DERIVED_SOLVER_CLASSES
    }
}
```

As you can see each block contains all the information needed to create an instance of the specified kind. The solver selected by NAME_OF_SOLVER contains information about the settings of its base class, the kind of scheduler which is to be used and finally its own settings which influence the parameters of this solver. The scheduler which is used by the solver is selected by the value of NAME_OF_SCHEDULER and also contains the information about its own settings and the settings of its base class. The next section describes how such a configuration file may look like.

5.6.2 A Simple Configuration File

To give a short introduction to the SA.cfg file we will now describe a file that selects the sequential solver and the geometric scheduler:

```
{
    SA_Solver SA_SeqSolver {
        SA_Solver {
            datafilename example.data
            solutionfilename example.solution
        }
        SA_Scheduler SA_SeqEasyScheduler {
            SA_Scheduler {
                OptType MAX
            }
        }
    }
}
```

In this file the solver SA_SeqSolver is selected. In the parameter block of the SA_Solver the name of the data file and the solution file is specified. The name of the data file will be passed to the method SA_Problem.ReadProblemData() by the SA_Initializer. The solutionfilename will be used to place the solution of the SA algorithm. As a scheduler the SA_SeqEasyScheduler is choosen and

in its parameter block the SA algorithm is advised to treat the problem as a maximization problem. Thus, it will search for the solution with the highest value of the cost function.
The subdirectory *config* contains some useful templates of configuration files. These templates must be renamed to SA.cfg and must be placed in the directory where the executable is situated. The parameter settings of this files are rather general and you may probably speed up the algorithm for a given problem. If you want to know more about the possibilities of the SA.cfg file consult section 7.2.

5.7 The SA_Output.rsc File

With the version 2.1 a new output concept was introduced to the parSA library. There are some points in every kind of annealing where a output is suitable, for example after the initialization, when the Equilibrium is reached or when the scheduler freezes. It may also be useful to be able to specify certain levels of output. All the output during the annealing process is now set up by the use of the SA_Output.rsc file in which bocks are specified. These blocks correspond to certain schedulers and solvers. By leading integer values lines of output are marked. Only lines with values lower or equal to the level defined by the *verboselevel* parameter in the configuration file are used. If you take a look at this file you will recognize other integers with the special literal # in front of them. Each variable of a scheduler and a solver has a corresponding number and these structures are substituted by the variable values.
Therefore the new concept does not only make the output more flexible but it is also possible to adapt the output of the parSA library to a special purpose or even a different language.
However from the users point of view it is only necessary to ensure that the SA_Output.rsc file resides in the parSA directory. The output defined in this file is currently rather similar to the output of previous versions apart from being better to understand.

6 Solving the QAP with the parSA

From version 2.1 on the parSA is delivered with an example program that solves the quadratic assignment problem (QAP). The QAP was choosen because it is a well known optimization problem. As the mathematical description of the QAP is rather simple it is also very suitable to explain how the parSA has to be adapted so that it can solve an optimization problem.

6.1 The Quadratic Assignment Problem

The origin of the QAP is the problem how to distribute n fatories on n locations and minimizing the transporting costs between them. Mathematically

this problem can be described as follows:
given two n-dimensional quadratic matrices $A = (a_{ij})$ and $B = (b_{ij})$ minimize

$$f(p) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} \dot{b}_{p(i)p(j)}$$

p being a permutation. For the general QAP A and B are not expected to be symmetric. There are as many possible solutions to this problem as there are different permutations so $n!$. As simulated annealing is a local search algorithm it is necessary to define a neighborhood of a certain feasable solution. We simply regard every permutation that differs only on two postions from a given one as one of its neighbors. There are only as many possibilities to choose a neighbor as there are subsets of two element in a set of n element so:

$$\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$$

which is much smaller than the whole problem space. The other criterion which has to be fullfilled is that every solution has to be reachable from another by using neighborhood operations. Obviously every permution can be constructed in that way. Given a permutation p_{old} and a solution p_{new} just choose index j with $p_{old}(j) = p_{new}(1)$ and apply the neighborhood operation on the indices 1 and n. The result is a permutiation that differs at most $n - 1$ positions from the required one and so the construction can be done by recursion.
Therefore the QAP seems to be a rather good canditate for the simulated annealing approach. The following section will show how easyily the interface of the parSA library can be adapted to a special problem.

6.2 Transforming the QAP to an user application of the parSA

Only two classes have to be implemented. As we have mentioned the terms of the QAP can be translated to terms of an optimization problem as follows:

solution space: our solution space is set of all possible solutions. This is equivalent to all permutations of n numbers

solution : a solution is one single permutation $p(x)$

cost function: the cost function is defined by the function $f(x)$ according to its definition in the previous section.

neighborhood of $p(x)$: another permutation that differs only on two postions from the actual one.

The two files *SA_QAPProblem.h* and *SA_QAPProblem.cc* in the subdirectory *example* contain all changes that has to be made to solve all the QAP instances with the parSA library. We added all instances of the QAP that are accessible at the QAPLIB home page [3]. For a detailled description of the pecularities of these instances refer to this page.
The most important problem specific changes to the class SA_Problem are the following:

SA_Problem: the class SA_Problem contains the information about the dimension n of the two quadratic matrices A and B and the matrices itself.

SA_Problem.GetInitialSolution(): we simply choose the identity function $Id(x)$ as an initial solution.

SA_Problem.GetRandomSolution(): choose randomly a permutation.

SA_Problem.GetCost(): Determine the cost function with a given permutation. If the cost of the old permutation and the two changed indices are known then the calculation can be done in $O(n)$ instead of $O(n^2)$ because only $O(n)$ summands are effected by the change. There are also many summands like $ac-ad-bc+bd$ when you calculate the difference between two neighbors. These are calculated more efficiently by $(a-b)(c-d)$.

SA_Problem.GetNeighbor(): Choose two random indices ij and switch the values $p(i)$ and $p(j)$. These indices are part of the solution because then the calculation of the cost function becomes very easy.

SA_Problem.ResetSolution(): reset the indices and the permutation.

SA_Problem.UpdateSolution(): make the change to the permutation permanent by deleting the indices.

Beside these changes some other more technical changes have to be done like implementing input and output methods. These few methods are all an user must think about. Nevertheless they could remarkably speed up your program if they are implemented efficiently. As the cost function is calculated thousands of times saving only a few multipications may result in seconds or even minutes of saved time.

7 Reference

7.1 Contact

If you have any questions, remarks or suggestions concerning the parSA library or this manual please feel free to contact us:

Email: parsa@uni-paderborn.de

or visit our homepage at

http://www.uni-paderborn.de/~parsa

7.2 The Parameters of the SA-Configuration File

When examining the following sections always keep the gerneral structure of a configuration file in mind:

```
{
    SA_Solver NAME_OF_SOLVER {
        SA_Solver {
            SETTINGS_OF_SOLVER_BASE_CLASS
        }
        SA_Scheduler NAME_OF_SCHEDULER {
            SA_Scheduler {
                SETTINGS_OF_SCHEDULER_BASE_CLASS
            }
            SETTINGS_OF_DERIVED_SCHEDULER_CLASSES
        }
        SETTINGS_OF_DERIVED_SOLVER_CLASSES
    }
}
```

The terminology of the description of the parameters is the following:

required parameter < possible value 1 | possible value 2 ...> :
required parameters are in normal prints and the possible choices are bracketed by <> and divided by |.

required parameter type :
if the required parameter is a number, the type is specified.

optional parameter <<u>default value</u> | possible value 2 ... > :
optional parameters are printed with slanted literals. The default value which is automatically used by the library is underlined and other possible values are not.

***optional parameter type = xxx* :**
if the optional parameter is a number, xxx specifies the default value.

Within the sections of the schedulers and solvers the parameters are listed in alphabetical order.

7.2.1 Solver and Scheduler Selection

Solver and scheduler are selected by using their names for the solver or scheduler definition. The following values are valid:

NAME_OF_SOLVER :
< SA_SeqSolver | SA_MIRSolver | SA_ClusteringSolver >

NAME_OF_SCHEDULER :
< SA_SeqEasyScheduler | SA_EasyScheduler | SA_AartsScheduler | SA_MIRScheduler | SA_TimeScheduler >

7.2.2 Settings of SA_Solver Base Class

The following keywords are accepted in the block SETTINGS OF SOLVER BASE CLASS:

datafilename PATH :
the name of the datafile has to be specified with its complete path.

outputfilename PATH :
one single line is written to this file for every SA-run.

***overwritesolution* < *never* | *<u>better</u>* | *always* >:**
decision of how the solution file is modified if more than one SA run is made.

***solutionfilename* < *PATH* | *<u>*</u>* > :**
this file contains the best solution found during one SA run. If no name is specified the datafilename with the suffix *.solution* is used.

***Startsolution* < *Random* | *<u>Init</u>*> :**
choice of the initial solution. Either the method SA_Problem::GetInitialSolution or the method SA_Problem::GetRandomSolution will be called.

***verbose* < *<u>on</u>* | *off* > :**
some information is written to the standard output while running.

verboselevel INT = 10
specifies the exent of output. 10 is standard output, greater levels produce

more detailed information. In the standard implementation of the library the maximum output level is 30. As the output can be modified by changing the SA_Output.rsc one can also define higher verboselevels.

writefunction $<$ ***stream*** $|$ ***path*** $>$:

the parameter specifies the output function. Stream selects SA_Problem::OutputSolution(ostream&,SA_Solution&) while path selects SA_Problem::OutputSolution(char *,SA_Solution&).The first of these methods is usually used with *cout* as ostream and the second method is used to write the solution to a file.

7.2.3 Settings of SA_Scheduler Base Class

The following keywords are accepted in the block SETTINGS OF SCHEDULER BASE CLASS:

initaccratio FLOAT = 0.9:

the requested initial acceptance ratio for the Aarts warming up method. If nothing is specified 0.9 is assumed.
The value has no effect if *initialtemperature* is set.

initialtemperature FLOAT = determine using Aarts method:

the initial temperature. If no value is specified the adaptive method of Aarts [1] is used.

OptType $<$ ***MIN*** $|$ ***MAX*** $>$:

select the goal of the algorithm. MIN means minimize the cost function, MAX means maximize the cost function.

picturedirectory DIRECTORY :

specifies the directory in which the input files for GNUPlot will be placed, which show the progress of the algorithm graphically. The runtime will increase rapidly if this option is used!

timelimit LONG = -1

Amount of seconds allowed for the annealing process. When set to a positiv value actual schedulers but the SA_TimeScheduler just freeze after this number of CPU-seconds.

thresholdvalue FLOAT = 0:

a new solution must be better than the value of *thresholdvalue* $\cdot$ *BestE* to be accepted. If it is set to 0 then the classical SA is used.

verboselevel INT = 10

outputlevel of the scheduler base class. See also *verboselevel* parameter of SA_Solver base class.

7.2.4 SA_SeqSolver

The following additional keywords are accepted in the block SETTINGS OF DERIVED SOLVER CLASSES when the selected solver is SA_SeqSolver:

algorithm $<$ **SeqAlg** $|$ **TestProblem** $>$ **:**
Either the sequential algorithm or a simple test of the implemented user functions is started.

7.2.5 SA_ClusteringSolver

The following keywords are accepted in the block SETTINGS OF DERIVED SOLVER CLASSES when the SA_ClusteringSolver is selected:

algorithm $<$ **SeqAlg** $|$ **ClusteredAlg** $|$ **ComputeLoopFactor** $|$ **TestProblem**$>$**:**
selection of the algorithm. *SeqAlg* starts a pure sequential simulated annealing run on each processor. *ClusteredAlg* starts the normal clustering algorithm. *ComputeLoopFactor* tries to compute the necessary subchainlength to solve a given problem. *TestProblem* runs a simple test of the user implemented functions.

ChooseMove $<$ ***best*** $|$ ***boltzmann*** $|$ ***random*** $|$ ***first*** $>$ **:**
decides which move is chosen if a cluster finds several acceptable solutions. The keyword boltzmann means that the boltzmann distribution is used to make the choice.

CommunicateInCluster $<$ **Group** $|$ **Async** $>$ **:**
selection of the communication mode in a cluster. For better efficiency in many applications Async is recommended.

CommunicationMode $<$ **Solution** $|$ **Move** $>$ **:**
In a cluster processors communicate either by sending whole solutions or only incremental by sending moves. If Move is chosen, make sure that the class SA_Move has been properly implemented.

ConstantSizeMove $<$ ***yes*** $|$ ***no*** $>$ **:**
In cases where constant size of moves can be guaranteed, the communication becomes more efficient.

DistributeSolutionAfterSubchain $<$ ***best*** $|$ ***boltzmann*** $|$ ***random*** $|$ ***no*** $>$**:**
specifies if and how an initial solution is chosen for all clusters after one

subchain.

***Equilibrium* < *Global* | *Local* > :**
decision whether the equilibrium is achieved or not. Global means that one single chief processor decides about the equilibrium. Local means that every master of a cluster decides for his own cluster.

***ExchangeFunctions* < *stream* | *MPI* >:**
selection of the user implemented exchange functions. Stream selects the stream functions while MPI selects the MPI based functions. Even the stream functions internally use MPI for communication, but as most users do allready know streams these functions should be easy to implement. Nevertheless more experienced users may choose to implement the MPI communication directly by adapting the MPI based functions.

MaxCluster INT = -1.0
maximum number of clusterlevels. The maximum number of processors in a cluster is bounded by $2^{\text{MaxCluster}}$. Therefore the value has to be in between 0 and log(numberofprocessors).

***MinEff FLOAT = 1.1*:**
selection of the clustering strategy. If the value is greater than 1.0 then product of cluster efficiency and cluster speedup is maximized. If the value is between 0.0 and 1.0 then the clusters are enlarged by cluster efficiency > MinEff. If the value is set to 0.0 then the speedup is maximized.

7.2.6 SA_MIRSolver

The following keywords are accepted in the block SETTINGS OF DERIVED SOLVER CLASSES when the SA_MIRSolver is selected:

***Betta_Runtime FLOAT = 1.1*:**
the increase factor for the run lengths must be ≥ 1.0

Ebest FLOAT:
the eastimated cost of the best known solution has to be specified.

***Epsilon FLOAT = 0.01* :**
the solution quality, which has to be achieved.

***Maximum_RunLength FLOAT = RunFactor · GetLocalN()*:**
length of the longest run.

***Minimum_RunLength FLOAT = GetLocalN()*:**
the initial length of a run.

***Samples INT = 10*:**
number of runs

***RunFactor FLOAT = 5* :**
In the second phase runs with a length out of the interval GetLocalN() ...RunFactor · LocalN(). This value is ignored when Maximum or Minimum-RunLength is defined.

7.2.7 SA_SeqEasyScheduler

The following keywords are accepted in the block SETTINGS OF DERIVED SCHEDULER CLASSES when the EasyScheduler is selected:

coolingratio FLOAT = 0.9:
the factor the temperature is reduced by after every temperature step. It has to be greater than 0 and less than 1.

frozenlimit INT = 5
the number of succsessive temperature steps with an acceptance ratio smaller than 'minaccratio' needed to reach the frozen state in the scheduler.

minaccratio FLOAT = 0.01:
decision if the SA algorithm is frozen. A value greater 0 than means cooling until 'frozenlimit' subchains have an acceptance ratio < minaccratio.The value 0 means decrement the temperature until the mean value stays almost the same.

subchainfactor FLOAT = 1.0:
chosen subchain length is increased by this factor.

subchainlength INT = GetLocalN:
the length of subchains (number of iterations) on a single temperature level.

verboselevel INT = 10
outputlevel of this scheduler class. See also *verboselevel* parameter of SA_Solver base class.

7.2.8 SA_EasyScheduler

The SA_EasyScheduler accepts the same keywords than SA_SeqEasyScheduler. The following additional keywords are accepted in the block SETTINGS OF DERIVED SCHEDULER CLASSES when the SA_EasyScheduler is selected:
subchainreduction < <u>linear</u> | sqrt | none > :

in each cluster, the subchainlength is adapted referring to NPROC. *linear* means that the subchainlength is devided by $NPROC$. *sqrt* selects the devision by $\sqrt{NPROC}$ and *none* means no subchain shortening at all.

7.2.9 SA_AartsScheduler

The following keywords are accepted in the block SETTINGS OF DERIVED SCHEDULER CLASSES when the SA_AartsScheduler is selected:

***delta FLOAT = 0.1*:**
distance parameter.

epsilon FLOAT $= 10^{-4}$**:**
stop parameter.

***omega FLOAT = 0.8*:**
smoothing parameter.

7.2.10 SA_MIRScheduler

The following keywords are accepted in the block SETTINGS OF DERIVED SCHEDULER CLASSES when the SA_MIRScheduler is selected:

Alpha FLOAT :
a kind of temperature reduction factor. It is used to calculate the number of temperature steps. It must be a value between 0 and 1.

Betta FLOAT :
chains are this times longer at the next temperature level. This value has to be ≥ 1.

***endtemperature FLOAT* :**
has to be greater than 0.

***Temperature_Reset = 0*:**
determines whether temperature is resetted or not.

7.2.11 SA_TimeScheduler

The following keywords are accepted in the block SETTINGS OF DERIVED SCHEDULER CLASSES when the SA_TimeScheduler is selected:

endtemerature FLOAT = -1.0 :
the endtemperature which should be achieved in a given timelimit. The temperature steps that are probably needed to reach the frozen state are estimated referring to the the start temperature, the end temperature and the cooling ratio. After that all the iterations, that can be made in the given timelimit, are distributed constantly on the single steps.

solutionquality FLOAT = 1.0 :
if set, then the scheduler freezes either after the given timelimit or if this solution quality is reached. It represents the value of the cost function.

7.3 List Of Filenames

This section contains a list of the files that are part of the parSA-package.

subdirectory	filename	description
.	version.txt	description of the delivery package
example	SA_QAPProblem.h	interface between example program and library
example	SA_QAPProblem.cc	interface between example program and library
example	SA_QAP.cc	the example program
example	SA.cfg	configuration file that controls the annealing algorithm
example	SA_Output.rsc	defines the output
example	ofiles.incl	a list of all generated object files
example	Makefile	LINUX/UNIX makefile
example	data	subdirectory containing QAP instances
include	SA_Scheduler.h	abstract scheduler base class
include	SA_ParScheduler.h	abstract base class MPI using schedulers
include	SA_SeqEasyScheduler.h	geometric scheduler sequential
include	SA_EasyScheduler.h	geometric scheduler parallel
include	SA_AartsScheduler.h	Aarts' adaptive scheduler
include	SA_ClusteringScheduler.h	scheduler class used by ClusteringSolver
include	SA_MIRScheduler.h	scheduler class used by MIRSolver
include	SA_TimeScheduler.h	EasyScheduler using a given timelimit
include	AllSchedulers.h	collection of scheduler header files
include	SA_Solver.h	abstract solver base class
include	SA_SeqSolver.h	simple sequential solver
include	SA_ParSolver.h	parallel solver base class
include	SA_ClusteringSolver.h	clustering parallelization
include	SA_MIRSolver.h	multipel independent runs parallelization
include	SA_Initializer.h	starts the selected solver
include	SA_Synchronizer.h	sycronizes the communication in a cluster
include	SA_Output.h	this class controls the output of the annealing process
include	SA_Output.rsc	in this file the outputlines are specified
include	TIntKeyList.h	sorted list used by SA_Output
include	parrandom.h	random number generator for parallel machines
include	rngs.h	random number generation
include	constants.h	constants definition
include	util_ salib.h	mathematical functions
libs	libparSAxxx.a	the parSA library; xxx specifies the operating system
doc	parSALibDoc.ps	User Manual (this file)
cfg	SA.cfg.xxx.yyy	templates for configuration files: xxx specifies the solver yyy the scheduler

References

[1] Aarts, E.H.L. et al
Parallel implementations of the Statistical Cooling Algorithm,
INTEGRATION, the VLSI journal **4** (1986), 209–238.

[2] Lee, S.Y., K.G. Lee
Synchronous and Asynchronous Parallel Simulated Annealing with Multiple Markov Chains,
IEEE Transactions on Parallel and Distributed Systems, Vol. **7**, Nr. **10** (1996) 993–1008.

[3] The QAP home page:
http://www.imm.dtu.dk/ sk/qaplib/