Parallel Optimisation in the SCOOP Library

Per Kristian Nilsen[1] and Nicolas Prcovic[2]

[1] SINTEF Applied Mathematics, Box 124 Blindern, 0314 Oslo, NORWAY
E-mail: pkn@math.sintef.no
[2] CERMICS-INRIA Sophia Antipolis, FRANCE
E-mail: Nicolas.Prcovic@sophia.inria.fr

Abstract. This paper shows how parallelism has been integrated into SCOOP, a C++ class library for solving optimisation problems. After a description of the modeling and the optimisation parts of SCOOP, two new classes that permit parallel optimisation are presented: a class whose only purpose is to handle messages and a class for managing optimiser and message handler objects. Two of the most interesting aspects of SCOOP, modularity and generality, are preserved by clearly separating problem representation, solution techniques and parallelisation scheme. This allows the user to easily model a problem and construct a parallel optimiser for solving it by combining existing SCOOP classes.

1 Introduction

SCOOP (SINTEF Constrained Optimisation Package) [2], is a generic, object-oriented C++ class library for modeling and solving optimisation problems. The library has been used for solving large–scale real–life problems in the fields of vehicle routing and forestry management [7]. Originally SCOOP contained a set of sequential optimisation algorithms, but recently it has been extended to accomodate for parallel optimisation as well. There already exist several parallel optimisation libraries (e.g, [3, 4]) but they are dedicated to one specific optimisation technique. Bringing parallelism to SCOOP meant to elaborate a high level parallelism model and optimiser-independent classes.

SCOOP consists of two main parts, a problem specification or modeling part and an optimisation part.

The modeling part of the library provides means for defining optimisation problems. In SCOOP a problem is represented by an encoding which is an aggregation of variables, constraints and objective function. This problem representation will remain static during optimisation.

The optimisation part consists of a set of optimisers together with classes that represent auxiliary entities needed by the optimisers.

Section 2 gives a brief overview of encodings, section 3 describes the main features of the optimisation part of SCOOP, while section 4 describes the parallelism model and the corresponding classes and how parallel versions of the optimisers can be instantiated.

2 Encodings

Given a real world problem there are several ways to represent it mathematically. An encoding is a concrete mathematical description of the problem, which an optimiser understands. A single problem may be described by several equivalent encoding instances from different encoding classes, which require different methods for efficient optimisation. An encoding consists of

- a variable set
- a constraint set
- an objective

A variable is a mathematical entity which has a domain and may take a value from that domain. Variables are represented by the *ScoVariable* base class. There are different types of variables in SCOOP, and they are all defined as subclasses of ScoVariable.

Constraints are defined on a set of variables. Given a solution, a constraint may examine the variables' values and tell whether it is satisfied or not.

Objective functions are able to evaluate solutions in two ways. They assign to each solution a real number which is the solution's objective value. However, for some objectives, such a ranking scheme is not always meaningful. Thus objectives may also rank the solutions by telling which of two solutions is considered to be the better or whether the objective has no preference. Different kinds of objectives are defined as subclasses of the *ScoObjective* base class.

Some encodings can be very general, representing large problem classes. An example of this can be an LP-encoding (Linear Programming). The LP-encoding would require real variables, linear constraints and a linear objective function.

Other encodings can be very specific, thus allowing for the exploitation of problem-specific features to enhance efficiency. An example of this can be an encoding for TSPs (Travelling Salesman Problem).

Currently SCOOP provides a small set of encodings. If the problem at hand is not supported by any of those, or if it would be beneficial to define a more problem-specific encoding, the programmer can define a new encoding type by subclassing the existing SCOOP classes.

Consider that we want to make a TSP-encoding. One way to do it is to define a variable type which represents a TSP tour and an objective that minimizes the length of such a tour. This is done by subclassing the SCOOP variable and objective classes. By representing the cities in a TSP by the integers 1 to n a tour can be seen as a permutation of those numbers. Permutations are represented by the class *ScoPermVar*. A TSP-encoding will contain one such variable. Another attribute of the TSP-encoding will be a cost matrix that represents the cost of traveling between any cities i and j. We represent the objective by the class *ScoTSPCostSum* which has a member function *objectiveValue* that calculates the sum of the cost of traveling a full circle between the cities given by a route.

The TSP-encoding is summarized in the following figure:

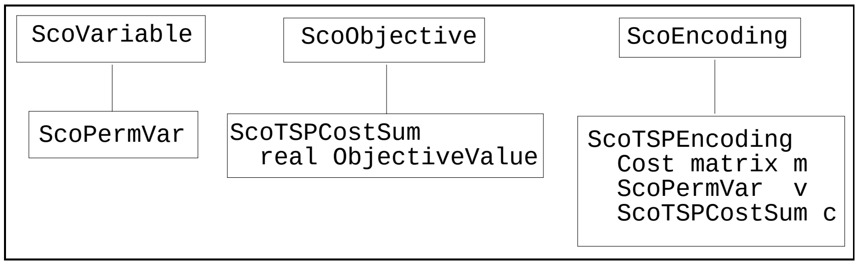

Fig. 1. ScoTSPEncoding

3 Optimisers

An optimiser is a class that represents some optimisation algorithm. The optimiser is given an encoding to optimise, and produces trial solutions to that encoding during optimisation. The best solution found is externally available.

Optimisers may be created on two levels of generality. The least general optimisers are dedicated to solving problems of one particular encoding class. E.g. an LP solver is only able to solve a linear program. Other more general optimisers are in principle able to solve any problem encoding. These optimisers represent the abstract aspects of optimisation methods, while the concrete details are filled in by auxiliary classes. In the current version of SCOOP we have focussed on a particular class of such general optimisers known as iterative improvement techniques (IIT), which are meta-heuristics for discrete optimisation. Examples are simulated annealing, genetic algorithms and tabu search.

Every optimiser is implemented as a subclass of the *ScoOptimiser* class which is an abstract class defining the minimum interface of optimisers. It contains a virtual function *optimise()* which is where the optimisation is performed. The different optimisers have their own versions of the optimise() function as well as additional data structures and functionality not found in the ScoOptimiser class.

Optimisers have a set of attributes in common. These are:

- solution
- initial solution generator
- solution manipulators

– stop criterion

In the following we will explain what these entities are and how they interact with the optimisers. The emphasis will be on IIT optimisers.

Solutions are represented by the *ScoSolution* class. The IIT optimisers operate on complete solutions, i.e. solutions where each variable is assigned a value. Therefore, SCOOP also provides initial solution generators, represented by the *ScoInitialSolGen* class, which generate complete solutions that are taken as starting points for the optimisers. An optimiser has an initial solution generator attached to it so that it can itself generate a starting solution if it is not provided one.

The IIT optimisers decide which solutions should be manipulated and how, but never perform any actual manipulation or evaluation which requires knowledge of the encoding and solution structure. This manipulation is left to attached solution manipulator classes which provide the detailed knowledge. The manipulators are the means by which new solutions are created and thus they represent the problem-specific aspects of an optimisation problem. They come in various types which take different numbers of input solutions and produce different numbers of new solutions. By applying a suitable sequence of manipulators, the optimiser attempts to produce an optimal solution.

There are two main groups of solution manipulator classes: neighbourhood operators and solution operators, represented by the base classes *ScoNeighbourhoodOp* and *ScoSolutionOp*, respectively. Examples of solution operators are crossover operators for genetic algorithms, while two-opt and relocate are examples of neighbourhood operators which can be used e.g. by a simulated annealing optimiser for solving TSPs and VRPs (vehicle routing problems), respectively.

In addition to solution manipulators the optimiser needs a stop criterion. A stop criterion is a criterion that the optimiser uses in order to determine when to stop optimising. It is represented by the *ScoStopCriterion* base class. Currently SCOOP provides three different stop criteria, represented by the following subclasses of ScoStopCriterion:

– *ScoIterStopCriterion.* Makes the optimiser stop after a specified number of iterations.
– *ScoUserInterrupt.* Becomes fulfilled when the user presses "Ctrl-C".
– *ScoStopWhenEither.* Consists of a number of sub-criteria. The criterion is satisfied when one or more of the sub-criteria is satisfied.

To sum up, the basic attributes of optimisers described in the current section are represented by the following SCOOP classes:

– ScoSolution
– ScoInitialSolGen
– ScoStopCriterion

– ScoSolutionOp and ScoNeighbourhoodOp

Each of these classes represent the base of a class hierarchy and are fairly generic. At the lowest levels in the hierarchies are classes representing more specific entities. An example of this is the stop criterion hierarchy with the base class ScoStopCriterion and the subclasses described above. Similarly to the encoding part of SCOOP, the optimiser part hierarchies will in many cases be extended by the SCOOP user by subclassing in order to solve specific optimisation problems efficiently. To illustrate how this is done we will continue the TSP example from the previous section.

3.1 Optimiser and Solution Operators for the TSP

We decide to use a genetic algorithm [5] to solve our TSP. For our purposes we can use the *ScoGAOptimiser* provided by SCOOP. However, we have to provide it with a suitable crossover operator and mutator. Since these solution operators represent the specific aspects of our problem we will have to define them ourselves.

We define a crossover operator which takes two permutations $p1$ and $p2$ and an integer k as arguments. The new permutation will be identical to $p1$ up to and including index k, wherefrom the rest of the numbers will follow in the same order as in $p2$, viewed cyclically from the position i where $p2(i) = p1(k)$.

In SCOOP there is a solution operator defined by the class *ScoOptimiser-RunOp* that applies an optimiser to a given solution to produce a new solution. This solution operator can be used as a mutator. We will use the *ScoDescent* optimiser as operator for the ScoOptimiserRunOp. ScoDescent is a simple steepest descent algorithm. One of its attributes is a neighbourhood operator. We will use a common neighborhood operator for TSPs: two-opt, which we define by the *ScoTSPTwoOptNH* class.

For both optimisers we define an initial solution generator by the class *ScoT-SPRandomSolution*. It generates a solution which is a random path.

Finally we choose the ScoIterStopCriterion for both the genetic algorithm optimiser and the mutator. Our choices and definitions can be summed up in figures 2–4.

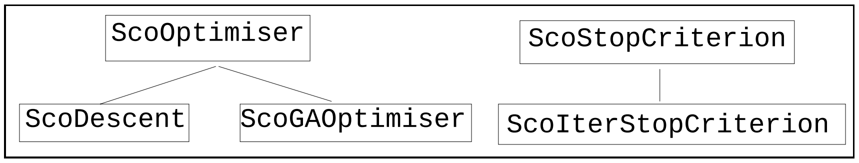

Fig. 2. Optimisers and stop criterion

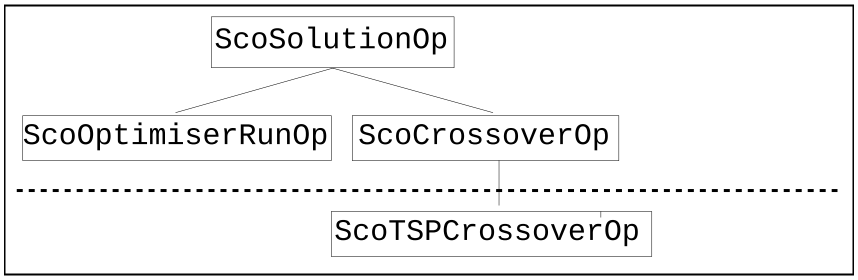

Fig. 3. TSP crossover operator

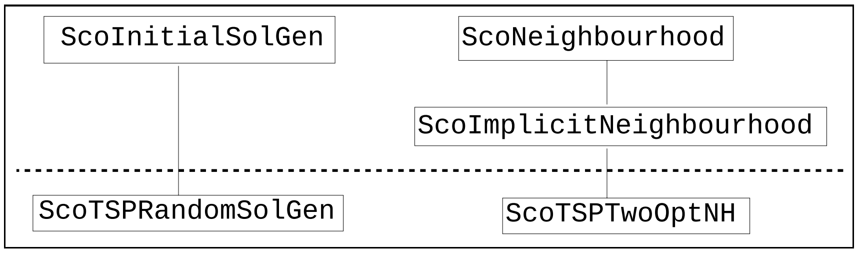

Fig. 4. Initial solution generator and two–opt operator

The figures show the parts of the SCOOP class hierarchies we used. Figure 2 contains only SCOOP classes. In figures 3 and 4 the classes above the dotted line are SCOOP classes while the classes below are the user-defined classes. The user-defined classes will henceforth be a part of SCOOP. In this way SCOOP is a constantly evolving library.

We can now set up a main program for solving TSPs by using the above classes:

```
//Include necessary files
//Initialize cost matrix c

ScoTSPEncoding encoding;
encoding.setCosts(c);

ScoTSPTwoOptNH nh;
```

```
ScoDescent descent;
descent.setInitialSolGen(* new ScoTSPRandomSolution);
descent.setNeighbourhoodOp(nh);
descent.setStopCriterion(* new ScoIterStopCriterion(20));

ScoGAOptimiser optimiser;
optimiser.setCrossoverOp(* new ScoTSPCrossoverOp);
optimiser.setMutator(* new ScoOptimiserRunOp(descent));
optimiser.setPopGenerator(* new ScoTSPRandomSolution);
optimiser.setPopulationSize(200);
optimiser.setStopCriterion(* new ScoIterStopCriterion(500));
optimiser.setEncoding(encoding);

optimiser.optimise();
optimiser.getBestSolution().print(cout);
```

4 Parallelism in SCOOP

In order to keep the generality and modularity aspects of the SCOOP library, two main goals were to be reached:

- We wanted to define new classes that would be general enough to permit parallel (i.e. each process uses the same optimiser on a different part of the search space of the problem), concurrent (i.e. each process uses a different optimiser to solve the whole problem) or cooperative (i.e. each process uses a different optimiser on a different part of the problem) optimisation.
- We wanted to re-use all the Sco[...]Optimiser::optimise() functions already defined, not to re-write a special parallel optimisation code corresponding to each way of optimising. Actually, we intended to be able to generate a ScoParOptimiser object by coupling a Sco[...]Optimiser object and a Sco-HandleMsg object (which would contain all that relates to parallelism).

In the following, we explain how we achieved this.

4.1 The ScoParOptimiser Class

The ScoParOptimiser class had to inherit from the ScoOptimiser class for several reasons. It is conceptually still an optimiser that will be used in the same way as a sequential one by using the same methods (optimise(), etc.). It contains two new variables: *optimiser*, a ScoOptimiser object and *message_handler*, a ScoHandleMsg object. As we wanted to be as general as possible, we had to be able to combine parallel, concurrent and cooperative processing. For example, if eight processors are available, it should be possible to run two concurrent optimisers, each using four processors to run a different parallel optimisation. Thus, a ScoParOptimiser also had to be able to handle a ScoParOptimiser object stored in *optimiser* and that was another reason to make it inherit from the ScoOptimiser class. Here is what the definition of ScoParOptimiser looks like:

```
class ScoParOptimiser : public ScoOptimiser
{
   ScoOptimiser*   optimiser;
   ScoHandleMsg    message_handler;
};
```

and the optimise() method:

```
void ScoParOptimiser :: optimise()
{
  [initialisations]
  optimiser->optimise();
  [terminations]
}
```

The ScoHandleMsg class will be described in section 4.3.

4.2 From sequential to parallel optimisation

Whatever kind of optimiser is used (systematic search like tree search algorithms, or local search like simulated annealing, genetic algorithms, tabu search, etc.), the algorithm always consists of a loop containing the following steps:

- select a state from the previously generated ones.
- generate one or several new states from the selected one.
- if none of the newly generated states is the best (or good enough) then loop.

The parallel version of an optimiser is very similar. The difference is that each process running its optimiser has also to communicate sometimes with the other processes. Here is what the loop looks like:

- select a state from the previously generated ones.
- generate one or several new states from the selected one.
- handle messages : check if messages have arrived and execute the corresponding procedures and send messages if necessary.
- if none of the newly generated states is the best (or is good enough) *and no "stop" message has arrived*, then loop.

To turn a sequential optimiser into a parallel one, a sequence of instructions managing the interprocess communication has to be inserted at the end of the optimisation loop. Nothing more has to be changed, apart from some initializations and terminations, before and after the loop.

4.3 The ScoHandleMsg Class

Now, we are going to explain how it was possible to insert message passing in an optimiser without modifying the already existing classes.

Here is what a Sco[...]Optimiser::optimise() function looks like:

```
Sco[...]Optimiser::optimise()
{
  [initialisations]
  while (! stop_criterion -> isSatisfied())
  {
   [generation of solution(s)]
  }
  [terminations]
}
```

The main idea is to store a message handler in the isSatisfied() function in order to leave the optimise() function unchanged.

Here is what the ScoHandleMsg class looks like:

```
class ScoHandleMsg : public ScoStopCriterion
{
  [...]
  StopCriterion* stop_criterion;

  Boolean handle_messages();
  [...]
}
```

and the isSatified() method:

```
Boolean ScoHandleMsg::isSatified()
{
  return handle_messages() ||
         stop_criterion->isSatisfied();
}
```

handle_messages() is a boolean function which receives new messages, calls the corresponding procedures and sends resulting messages. It returns True only when the process receives a "stop" message. When a parallel optimiser *paropt* is built from a sequential optimiser *opt*, a ScoHandleMsg object *hm* will be created. The stop criterion of *hm* will be set to the one of *opt*, then the stop criterion of *opt* will be set to *hm*. So when the isSatified() method will be called in the optimisation loop, the handle_messages() method will be called before the initial isSatified() method of the optimiser.

4.4 Building a ScoParOptimiser Object

From a user's point of view, creating a parallel version of an optimiser is very simple. He just has to define his optimiser *opt* and his message handler *hm* and then declare in his source files:

```
  ScoParOptimiser paropt(opt, hm);
```

then perform optimisation by:

```
  paropt.optimise();
```

opt and *hm* are gathered and embedded in *paropt* during the call of the ScoParOptimiser constructor. It sets the *optimiser* and *messagehandler* variables to *opt* and *hm* and then changes their contents and redirects pointers as described in section 4.3.

Now we recapitulate what happens during the call of *paropt.optimise()*. After some initializations (e.g. initial work sharing between the processes) the optimise() method of *optimiser* is called. So the loop of *opt* is run and, when calling the isSatified() method, handle_messages() is called before running the initial isSatisfied() method of *opt*. The ScoParOptimiser class provides a boolean function *hasBestSolutionGlobal()* that returns true if the optimiser's solution is better than the solutions of the other optimisers. This function should be called when all processes has finished optimising to identify which process contains the best solution.

The parallel scheme presented here maintains the high degree of modularity which is one of the most important aspect of SCOOP. By clearly separating parallel management from sequential optimisation, it allows two complementary advantages: firstly, applying the same parallel scheme to different optimisers; secondly, applying different parallel schemes to the same optimiser, without rewriting any of them.

4.5 SCOOP Message Handlers

Some basic but general message handlers (inheriting from ScoHandleMsg) are already defined in SCOOP:

- the ScoHandleMsgFirst class : every process is independently trying to find the best (or a good enough) solution, and as soon as one has found it, it tells the others to stop. The stop criterion to be used with this parallel scheme is the quality/value of the objective function.
- the ScoHandleMsgBest class : every process has to look for its own local optimum, and when every process has found it, results are gathered and the best of the local optima is returned. The stop criterion to be used is the number of steps of the optimisation loop.

These two message handler classes may be used with any optimiser but they are not likely to be efficient for a broad number of problems. They neither keep on sharing the work during the resolution in order to maintain a good balance, nor do they exchange useful information that could accelerate the search (e.g. a new upper bound in a branch and bound solver). In order to perform an efficient parallel optimisation, other more specialized ScoHandleMsg classes have to be defined in order to fit each solving technique.

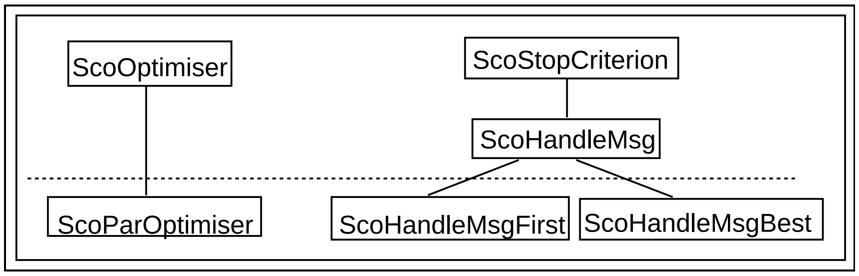

Fig. 5. Classes added in SCOOP for parallelism

4.6 An Example of Parallel Optimisation

Let us consider again the TSP example presented in the previous sections. This problem was solved using a genetic algorithm optimiser but we may suppose that other solving techniques would fit and may be even more efficient. If we cannot have any good estimate of which optimisation technique is the best, we can try them all in parallel and get the result of the one that has produced the best solution after a given elapsed time. Here is an example of a main program for concurrently solving the TSP example by using a genetic algorithm and simulated annealing [6]:

```
//Include necessary files
//Initialize cost matrix c

ScoTSPEncoding encoding;
encoding.setCosts(c);

// Define ScoGAOptimiser optimiser1 and make the
// appropriate settings as in the sequential example
// Define ScoSAOptimiser optimiser2 and make its settings

ScoOptimiser* optimiser;
if (my_id == FIRST_PROCESS)
    optimiser = &optimiser1;
else /* my_id == SECOND_PROCESS */
    optimiser = &optimiser2;

ScoParOptimiser par_opt(*optimiser, *new ScoHandleMsgBest);
par_opt.optimise();
if (par_opt.hasBestSolutionGlobal())
    par_opt.getBestSolution().print(cout);
```

Parallelism in SCOOP has been implemented with MPI (Message Passing Interface) [1] so this main program will be run in parallel on n different processes, where the number n is determined by the user.

5 Conclusion

Because of the modularity of SCOOP, we have been able to integrate parallelism by adding a few general classes, without changing anything that was already defined in the library. These new classes are independent from the optimisation techniques because they only deal with message handling. Thus, it is possible to parallelize any optimiser with them and parallelism can be extended by defining subclasses from them. However, in order to be really efficient, the already existing classes may have to be customized with the specific aspects of some optimisation technique. In the near future we intend to extend SCOOP with parallel classes with general or specific load balancing techniques.

References

1. MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8 (3/4):165–416, 1994.
2. SCOOP 2.0 Reference Manual, SINTEF report no. STF 42A98001, ISBN 82-14-00047-5, 1998.
3. M. Benaichouche, V. D. Cung, S. Dowaji, B. Le Cun, T. Mautor and C. Roucairol. Building a Parallel Branch and Bound Library. *Solving Combinatorial Optimization Problems in Parallel*, LNCS 1054, Springer, 201–231, 1996.
4. R. Finkel and U. Manber. DIB—A Distributed Implementation of Backtracking. *ACM Transaction on Programming Languages and Systems*, 9(2):235–256, 1987.
5. J.H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.
6. S. Kirkpatrick, C. Gellat and M. Vecchi. Optimisation by simulated annealing. *Science*, 220:671–680, 1983.
7. G. Misund, G. Hasle and B. S. Johansen. Solving the clear-cut scheduling problem with geographical information technology and constraint reasoning. In *ScanGIS'95 Proceedings*, pages 42–56, 1995. Also published as SINTEF report no. STF 33S95027.