Mathematical Modelling and Analysis
Volume 10 Number 3, 2005, pages 217–236
© 2005 Technika ISSN 1392-6292

TEMPLATE REALIZATION OF GENERALIZED BRANCH AND BOUND ALGORITHM

M. BARAVYKAITĖ[1], R. ČIEGIS[1] and J. ŽILINSKAS[2]

[1] *Vilnius Gediminas Technical University*
Saulėtekio al. 11, LT-10223 Vilnius, Lithuania
[2] *Institute of Mathematics and Informatics*
Akademijos 4, LT-08663 Vilnius, Lithuania
E-mail: `{mmb, rc}@fm.vtu.lt; julius.zilinskas@mii.lt`

Received June 30 2005; revised September 23 2005

Abstract. In this work we consider a template for implementation of parallel branch and bound algorithms. The main aim of this package to ease implementation of covering and combinatorial optimization methods for global optimization. Standard parts of global optimization algorithms are implemented in the package and only method specific rules should be implemented by the user. The parallelization part of the tool is described in details. Results of computational experiments are presented and discussed.

Key words: Templates, parallel algorithms, branch and bound algorithms, combinatorial optimization, global optimization

1. Introduction

This paper presents the template implementation of generalized branch and bound (BB) algorithm analyzes the performance of parallel algorithms implemented in the template.

Branch and bound algorithm is a popular method that can be applied for a variety of optimization problems. It has a general logical structure and using this feature a template programming can be applied.

The idea of the template programming is to implement general structure of the algorithm that could be later used to solve different problems. All general features of the algorithm and its interaction with the particular problem must be implemented in the template. The particular features related to the problem must be given by the template user. The user only has to identify

the needed algorithm, choose the right template and implement problem dependent parts. Templates eases programming, clears algorithm logic, allows easy re-use of the implementation.

Template based programming can be very useful in parallel programming [11, 20, 23]. Parallel algorithm template must fully or partially specify the main parallel algorithm features: partitioning, communication, agglomeration and mapping. The goal is to provide a technique for quick and reliable development of parallel applications that employ frequently occurring parallel structures [23]. From the user's point of view, all or nearly all the coding should be sequential; all the parallel aspects (or almost all) should be provided by the tool. Often parallel programs are created by parallelizing the existing sequential programs. Then parallel algorithm template can use features implemented by the sequential algorithm template. If a sequential template was used to create the sequential program, then there is no need to rewrite existing code to obtain parallel one. In this way, templates save time and efforts of the users. On the other hand generalization of main parallel aspects of algorithms may result in lower efficiency of the implementation. Some examples of parallel templates of different algorithms are MST [21], Mallba [1], ARNIA [20], CODE [23].

Unlike the data parallel applications optimization problems characterized by an unpredictably varying unstructured search space [25]. It produces additional difficulties for creation of parallel BB algorithms:

- the change of space search order with respect to sequential one,
- processor load disbalance,
- costs of additional communications.

Developing a single parallel BB application these difficulties can be controlled, but it is a real challenge to solve these problems in a template. Some examples of BB parallelization tools are BOB [20] , PICO [7], PPBB [24], PUBB [22].

The rest of the paper is organized as follows. In Section 2 sequential version of the general branch and bound algorithm is described. In Section 3 parallel branch and bound algorithms are discussed, some notes on their implementation are given and a general structure of the BB template is described. The analysis of performance of different experiments are given in Section 4 Here our main goal is to show that template programming allows us to test various variants of parallel BB algorithms without additional programming efforts, when the parallel template of BB algorithm is implemented. Section 5 gives some conclusions.

2. Parallel Branch and Bound for Combinatorial and Global Optimization

2.1. Optimization

Many problems in engineering, physics, economic and other subjects may be formulated as optimization problems, where the minimum value of an objective function should be found. Mathematically the problem is formulated as follows

$$f^* = \min_{X \in D} f(X). \tag{2.1}$$

where $f(X)$ is an objective function, X are decision variables, and D is a search space. Besides of the minimum f^*, one or all minimizers $X^* : f(X^*) = f^*$ should be found.

Type of optimization is defined by the search space and the objective function of the problem. In *combinatorial optimization* search space is discrete. In *local optimization* and *global optimization* the objective function is a nonlinear function of continuous variables $f : \Re^n \to \Re$, where n is number of variables, and a continuous search space $D \subseteq \Re^n$ is called feasible region. Differently from global optimization, local optimization is based on assumption that the objective function is unimodal, in other words it has a single local minimum.

2.2. Generalized Branch and Bound Algorithm

Branch and bound is a technique used in combinatorial optimization and covering global optimization algorithms. Covering global optimization methods can solve global optimization problems of some classes with guaranteed accuracy. They detect the sub-regions not containing the global minimum and discard them from further search. The partitioning of the sub-regions stops when global minimizers are bracketed in small sub-regions guaranteeing the prescribed accuracy. A lower bound (LB) for the objective function over the sub-region may be used to indicate the sub-regions which can be discarded. Some methods are based on lower bound constructed as convex envelope of an objective function [8]. Lipschitz optimization is based on assumption that the slope of an objective function is bounded [15, 16]. Interval methods estimate the range of an objective function over a sub-region defined by a multidimensional interval using interval arithmetic [14]. Branch and bound technique is used for managing the list of sub-regions and the process of discarding and partitioning. The general branch and bound algorithm is shown in Figure 1, where L denote candidate set, S the solution, $UB(D_i)$ and $LB(D_i)$ denote upper and lower bounds for minimum value of the objective function over sub-space D_i.

An iteration of the classical branch and bound algorithm processes a node in the search tree representing a not yet explored sub-space of the search space [4]. Iteration has three main components:

Cover solution space D by $L = \{L_j | D \subseteq \bigcup L_j, j = \overline{1,m}\}$ using **covering rule**.
$S = \emptyset, \quad UB(D) = \infty$.
while sub-space list is not empty $L \neq \emptyset$ **do**
  Choose $I \in L$ using **selection rule**, exclude I from L.
  if $LB(I) < UB(D) + \epsilon$ **then**
    Branch I into p subsets I_j using **branching rule**.
    for all $I_j, j = \overline{1,p}$ **do**
      Find $UB(I_j \bigcap D)$ and $LB(I_j)$ using **bounding rules**.
      $UB(D) = \min(UB(D), UB(I_j \bigcap D))$.
      if $LB(I_j) < UB(D) + \epsilon$ **then**
        if I_j is a leaf **then** $S = I_j$.
        else $L = \{L, I_j\}$.
        end if
      end if
    end for
  end if
end while

Figure 1. General branch and bound algorithm.

1. Selection of the node to process.
2. Branching of the search tree by dividing the selected sub-space.
3. Bounding of the branches by discarding not promising sub-spaces.

Before the cycle of iterations the list of candidate sub-spaces should be initialized by covering search space by one or more sub-spaces. In combinatorial optimization '*leaf*' of search tree means that the node is a solution, in global optimization it means that it is a small sub-region bracketing a solution with predefined accuracy. The rules of covering, selection, branching and bounding differ from algorithm to algorithm.

Covering and branching rules

The rules of covering and branching depend on type of partitions used. For example, in global optimization partitions may be hyper-rectangular, simplicial, hyper-conic or hyper-spherical. Partitions obtained with branch and bound algorithms for global optimization differ from those used in combinatorial optimization in that the classes of partitions may overlap and the number of possible partitions is infinite [15]. Usually feasible regions of general global optimization problems are hyper-rectangles. All interval and most of Lipschitz global optimization branch and bound algorithms use hyper-rectangular partitions. In this case initial covering is very simple: $L = \{D\}$. Covering by hyper-spheres causes overcovering of feasible region as well as overlapping of spheres themselves. Use of regular simplexes causes overcovering of feasible region and a non-overlapping branching is not known in more than two dimensions. The use of irregular simplexes enables non-overcovering of feasible region as well as non-overlapping branching.

Selection strategies

Main strategies of selection are the following:

- **Best first.** Select a candidate with minimal lower bound. Candidate list L can be implemented using *heap* or *priority queue*.
- **Depth first.** Select the youngest candidate. A node with the largest level in the search tree is chosen for exploration. First-In-Last-Out structure is used for candidate list which can be implemented using *stack*.
- **Breadth first.** Select the oldest candidate. First-In-First-Out structure is used for candidate list which can be implemented using *queue*.
- **Improved selection.** It is based on heuristic [18, 5] or probabilistic [6] criteria. Candidate list can be implemented using *heap* or *priority queue*.

Node selection strategies influence the efficiency of branch and bound algorithm and the number of nodes kept in candidate list. For particular problems some strategies can considerably improve the performance of the algorithm.

Bounding rules

The bounding rule describes how the bounds for minimum of the objective function are found. For the upper bound for minimum over the search space $UB(D)$ the best currently found value of the objective function might be accepted. In global optimization the lower bound for minimum of the objective function over sub-region $LB(I)$ is the lower bound for values of objective function over considered sub-region which can be estimated using convex envelopes, Lipschitz condition or interval arithmetic.

3. Parallel Branch and Bound Algorithms

Any parallel algorithm for a given problem attempts to divide it into subproblems which can be solved concurrently on different processors. Four main steps are performed during development of a parallel algorithm [9]: partitioning, communication, agglomeration, mapping.

The aim of *partitioning* is to decompose the computations into sub-tasks. Attention is focused on recognizing opportunities for parallel execution. During this step we should take into account that a larger number of sub-tasks gives more possibilities to improve a load balancing among processors, but at the same time it increases data communication costs.

Then *communication* required to coordinate task execution is determined.

In *agglomeration* step, if necessary, tasks are combined into larger tasks to improve performance and to reduce development costs. This step is not necessary for parallel BB algorithms described further.

Then each sub-task is *mapped* or assigned to a processor. During this step we try to minimize the computation time T_p or, equivalently, to preserve a good load balance among processors.

There are three main approaches to design parallel branch and bound algorithms [10]:

1. *Parallelism of type 1* introduces parallelism performing the operations on each separate sub-problem. It consists, for example, of executing the bounding operation in parallel for each sub-problem to accelerate the execution or the value of the objective function may be computed in parallel. Thus, this type of parallelism has no impact on the general structure of branch and bound algorithm and depends essentially on the given particular problem.
2. *Parallelism of type 2* consists of building branch and bound search tree in parallel by performing operations on several sub-problems simultaneously. Hence, this type of parallelism may affect the execution and subspace search order of the algorithm. Algorithms may be synchronous or asynchronous, to have single or multiple (distributed) lists of candidates.
3. *Parallelism of type 3* implies that several branch and bound search trees are built in parallel. The trees are characterized by different operations (branching, bounding, testing for elimination, or selection), and the information generated when building one tree can be used for the construction of another tree.

3.1. Implementation of parallel branch and bound

Application of algorithms to solve practical problems crucially depends on efficiency and reliability of algorithms implementing optimization methods. Development of such algorithms is not trivial. However, branch and bound algorithms have general scheme and differ only by rules of covering, selection, branching and bounding. Parallel branch and bound algorithms for combinatorial and global optimization possess many similarities and few differences as discussed in [4]. We propose a template for development of sequential and parallel branch and bound algorithms, where only particular rules should be implemented.

Parallel aspects of algorithms of type 1 are strongly problem dependent. Although the sequential part of branch and bound algorithm from the template can be used to create the program, the user has to take care of all parallel aspects at lower level operations like computing the objective function. Algorithms of type 3 are parallel by their definition, but we will not consider this type of parallelization in this paper.

Our aim is to develop parallel branch and bound algorithm template oriented to type 2 parallel algorithms. They are based on domain decomposition method and perform the same branch and bound steps over different subspaces of the solution space. Due to such structure of the algorithm it can be generalized for different problems and BB algorithms are suitable for template programming. The domain decomposition is achieved by distribution of candidate list.

We note that a full list of feasible search subspaces has unstructured and non-deterministic structure, thus it is very difficult to distribute local subproblems among processors preserving the good load balance. In addition we

should guarantee that the total costs of parallel BB algorithm are similar to sequential costs.

Master – slave algorithm

Parallel BB algorithms of type 2 with a single list of candidates can be implemented using Master – slave algorithm. At least two processes should execute Master – slave algorithm, one process is called Master, others – Slaves. Master process forms and controls the job pool, takes a job form the pool and sends it to the idle Slave process. The Slave processes gets the job from the Master process, calculates it and sends the result back to the Master. Then Master process collects calculation results and adds them to the totals. End of calculations is determined when there is no jobs in the job pool and all slaves had finished their calculations. This algorithm has a strict structure and a parallel template was created and used for parallelization of *Master–slave* logic programs [2]. After writing a sequential Master – slave program following some instructions given by the template, parallel program can be obtained automatically.

For the BB algorithm the list of subspaces is considered as a job pool. The Master process selects the subspace and sends it to the idle Slave process to perform bounding and branching. After that Slave sends newly generated subspaces to the master process together with the new upper bound $UB(D)$, if obtained. Master executes algorithm presented in Figure 2 and Slave's algorithm is presented in Figure 3.

Automatic parallelization of such algorithm can be obtained using Master – slave algorithm template MST [2]. PUBB [22] , PICO [7] BB algorithm templates have implementations of Master – slave algorithm.

3.2. Domain decomposition with distributed list of candidates

In this section we consider some static distribution methods, which are implemented in our template tool.

A priori distribution of feasible search space

Considering a possible parallel execution of BB algorithm we note that any subspace of the feasible search space can be searched independently and in any order. Initial space is divided into several large subspaces that are mapped to the processes and algorithm presented in Figure 1 is performed. The number of subspaces coincides with the number of processors. Fine-grain subspaces are generated from these initial subspaces dynamically during the calculation process performing branching step. There is no need to re-map subspaces generated later. Communications are required for initial subspace distribution and final gathering of solution found. In this way a simple asynchronous parallel BB algorithm with distributed job pool is obtained. We will call it a parallel branch and bound algorithm with a static distribution of job pool (SJP).

Cover solution space D by $L = \{L_j | D \subseteq \bigcup L_j, j = \overline{1,m}\}$ using **covering rule**.
$S = \emptyset, \quad UB(D) = \infty$.
Create list of idle slave processes $SL = \{S_k, k = \overline{1,n}\}$
for all $SL_k, k = \overline{1,n}$ **do**
  Choose $I \in L$ using **selection rule**, exclude I from L.
  Select idle process S_k, exclude S_k from SL
  Send I, $UB(D)$ to the idle Slave process S_k
end for
while sub-space list is not empty $L \neq \emptyset$ and not all Slave processes are idle
  Receive $\{ I_j, LB(I_j), UB(I_j) \}$, $j = 1, \ldots, J$ from Slave process S_k
  Insert slave S_k into the list of idle slave processes SL
  for all $I_j, j = \overline{1,J}$ **do**
    $UB(D) = \min(UB(D), UB(I_j \bigcap D))$.
    if $LB(I_j) < UB(D) + \epsilon$ **then**
      if I_j is a leaf **then** $S = I_j$.
      else $L = \{L, I_j\}$.
    end if
  end for
  for all slaves S_k in SL **do**
    if sub-space list is not empty $L \neq \emptyset$
      Choose $I \in L$ using **selection rule**, exclude I from L.
      if $LB(I) < UB(D) + \epsilon$ **then**
        Exclude S_k from SL
        Send $\{I, UB(D)\}$ to the slave process S_k
      end if
    end if
  end for
end while
Send *Finish* signal to all slave processes

Figure 2. Master's algorithm.

while signal *Finish* is not received **do**
  Receive the $\{ I, UB(D) \}$ from Master
    Branch I into p subsets I_j using **branching rule**.
    for all $I_j, j = \overline{1,p}$ **do**
      Find $UB(I_j \bigcap D)$ and $LB(I_j)$ using **bounding rules**.
      if $LB(I_j) < UB(D) + \epsilon$ **then**
        insert $\{I_j, LB(I_j), UB(I_j \bigcap D)\}$ to the message M
    end for
  Send message M to the Master process
end while

Figure 3. Slave's algorithm.

Random distribution of the feasible search space

In order to achieve a better load distribution among processors the space of feasible solutions is divided into M subspaces, where M is much larger than

the number of processors. Then subspaces are distributed a priori among processors in random order. We expect that each processor will get approximately the same number of hard and easy sub-tasks. This algorithm will be called RJP parallel branch and bound algorithm with a random distribution of job pool.

A similar modification was used for parallel numerical integration algorithms [3].

Exchange of the new local minimum objective function value

A subspace is eliminated from the further search by comparing the lower bound of the subset with the upper bound. The best currently found value of objective function can be used for the upper bound. In previously described parallel algorithms processors know only values of objective function found in the subspaces mapped to the particular process. In some situations this can result in slower subspace elimination. Processors can share the best known value. When such new value is found, the process broadcasts it to the other processes. In order not to stop calculations, this exchange is performed asynchronous. These modifications of the BB algorithm will be called SJP SE and RJP SE, depending on the rule to distribute the initial job pool.

3.3. Complexity of parallel optimization algorithms

Parallel optimization algorithms have unpredictably varying unstructured search space [25]. It should be noted that because of the domain decomposition the order of search can differ for parallel and sequential branch and bound algorithm even using the same subset selection rule. Sub-spaces eliminated in the sequential algorithm can be explored in parallel one, and it is possible that a total number of the sub-spaces searched in the parallel algorithm can be large.

Let us define the number of nodes in the generated search tree as a unit to measure the complexity of branch and bound algorithm. Then the growth of number of sub-spaces searched using parallel algorithm can be measured by the *search overhead factor*

$$SOF = \frac{W_p}{W_0},$$

where W_p is the sum of processed sub-spaces in parallel algorithm, and W_0 is the sum of tasks processed by the sequential algorithm. This coefficient helps to estimate efficiency of the parallel algorithms [12].

3.4. Structure of branch and bound algorithm template

The parallel branch and bound algorithm template proposes C++ classes for implementation of the branch and bound algorithm. A problem to be solved and the solution of the problem should be described by the user. MPI library

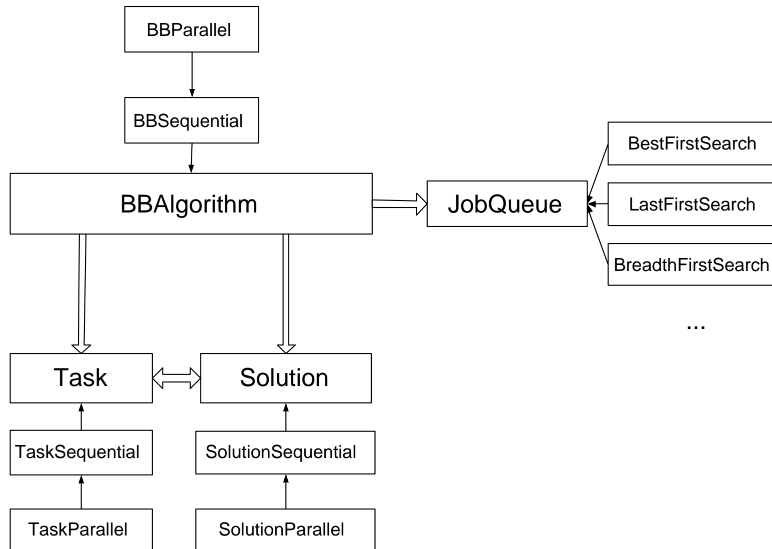


Figure 4. General structure or the template.

is used for underlying communications. General structure of the template is given in Figure 4.

`BBAlgorithm` implements various sequential and parallel BB algorithms. The algorithm is performed using `Task`, `Solution` and `JobQueue` instances. `BBAlgorithm` is implemented by the template.

`JobQueue` defines the strategies how to select next task from the list of tasks for subsequent partitioning. The most popular strategies are already implemented as methods and they are ready for application. The user can implement his/her own specific rules, in this case he/she should define methods `Insert, Delete, QueueSize, QueueEmpty`.

Class `Task` defines the problem to be solved. It should implement the basic BB algorithm methods: `Initialize`, `Branch`, `Bound`. Some often used `Branch` methods for hyper-rectangular, hyper-conic, hyper-spherical or irregular simplex could be implemented in the template. Standard `Bound` calculation methods such as for Lipschitz functions could be included in the template as well. Method `Ready` is used to check the accuracy for global optimization or the '*leaf*' condition for combinatorial optimization problems.

Class `Solution` implements the solution to be found.

Parallel methods `Task` and `Solution` additionally have methods `Pack` and `Unpack` that define how to prepare a task and solution in order to exchange information between processors.

4. Experiments

4.1. Experimental investigation of heap and priority queue

Implementation of the list of candidates is one of important factors of performance of branch and bound algorithms. Implementation of stack and queue for 'depth first' and 'breadth first' selection is trivial. Time of insertion and deletion of element to/form such type of structure does not depend on number of elements in the list. However the list of candidates for 'best first' and 'improved selection' requires selection of candidate with the smallest value of criterion. Priority queue is often used for this purpose, for example in PROFIL V 2.0 [17] implementation of the global unconstrained minimization method involving a combination of local search, branch and bound technique and interval arithmetic, and in CToolbox (C++ toolbox for verified computing) [13] which is a library for problem-solving routines covering one-dimensional and multidimensional problems: accurate evaluation of polynomials, automatic differentiation, linear and nonlinear systems of equations, linear optimization, global optimization, and zeros of complex polynomials.

Although time of selection and deletion of element from the priority queue does not depend on the number of elements in the list, the worst case insertion time linearly depends on the number of elements. Because of this priority queue implementation can be usable only for solving small example problems where the largest number of candidates in the list is small. For most of optimization problems the list of candidates grows rapidly and insertion of elements to the list of candidates can take even more time than calculation of bounds which is supposed to be the most time consuming part of branch and bound algorithms. 'Depth first' and 'breadth first' selection strategies can perform better than other selection strategies because of efficient implementation of the list of candidates, but not because the number of investigated nodes is smaller. Parallel branch and bound with priority queue implementation of the list of candidates can have better speedup because of reduced time of insertion after distribution of the list of candidates, but not because of excellent load balancing.

Some of the mentioned problems can be at least partly avoided by using heap structure, which is a complete binary tree where each node has larger value of criterion than its parent. Heap and priority queue are implemented in the proposed template. To compare performance of heap and priority queue, lists of different sizes have been constructed inserting elements with random keys. The sum times of construction (insertion of elements) and use (selection/deletion of elements) for different sizes of lists have been measured and are shown in Figure 5. The figure shows that priority queue and 100 times larger heap have similar construction and deletion time. This strongly suggests use of heap structure for implementation of the list of candidates for 'best first' and 'improved selection'. Heap was used for *b*est first in presented experiment results.

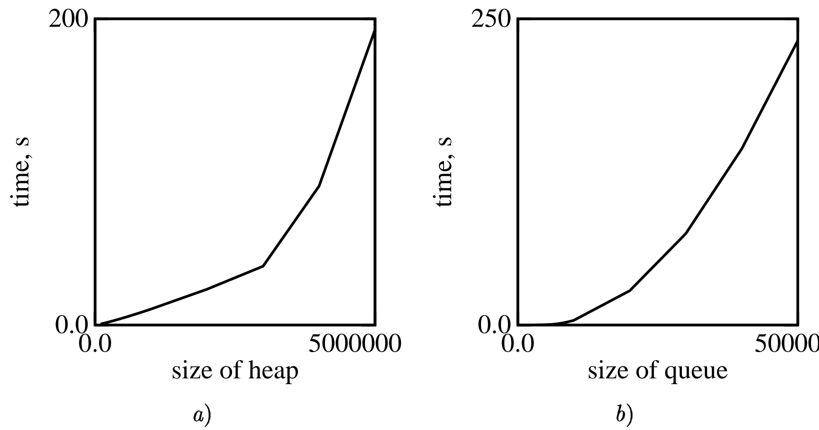


Figure 5. Comparison of performance of heap and priority queue when inserting elements with random keys.

4.2. Experiment results

In this section we present results of computational experiments which demonstrate the performance of our tool. We solve Lipschitz function minimization and symmetric traveling salesman problems. Our main goal was to test the ability of the tool to generate automatically parallel BB algorithms subject to different selection rules and task distribution strategies.

Calculation experiments were performed on up to 15 nodes of Vilnius Gediminas Technical university computer cluster *Vilkas* (*www.vilkas.vtu.lt) and up to 256 nodes of* IDRIS *supercomputer (*www.idris.fr). Results are analyzed calculating speedup, efficiency of parallel execution [9] and SOF. The aim of the experiments was to test the usage of the template solving different problems and the performance and scalability of implemented algorithms.

Lipschitz function minimization

Let f be a real valued Lipschitz function on a compact set $D \subset \mathbb{R}^2$, i.e. it satisfies the inequality

$$|f(x_1) - f(x_2)| \leq L\|x_1 - x_2\| \quad \text{for all} \;\; x_1, x_2 \in D\,,$$

where L is called a Lipschitz constant. We solve the minimization problem 2.1. Let D be a rectangle $(a_1, b_1) \times (a_2, b_2)$. The bisection of a rectangle is implemented at the midpoint of the largest edge. A simple lower bound μ for minimum value of $f(x),\ x \in P$ is defined in the following way

$$\mu(P) = f(m) - \frac{1}{2} L\delta(P),$$

where $\delta(P)$ is the diameter of the rectangle P:

$$\delta(P) = \Big(\sum_{j=1}^{2} (b_j - a_j)^2 \Big)^{1/2}.$$

The algorithms for computing more accurate (but also more expansive) lower bounds are given in [16].

Problem 1. Consider minimization problem (2.1) in $S = [0,10] \times [0,10]$, when

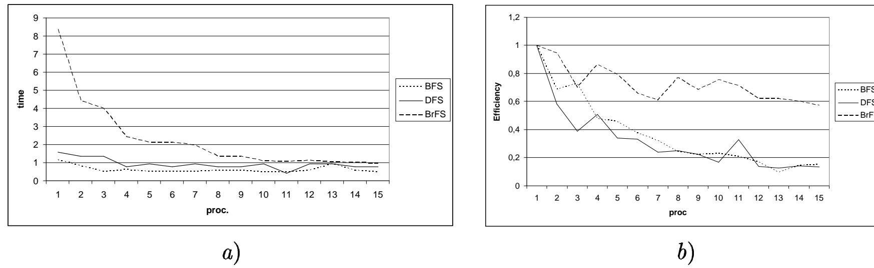
$$f_1(x) = \frac{1}{2}x_1^2 - 9x_1 + 20 + \frac{1}{2}x_2^2 - 9x_2 + 20\,.$$

Problem 2. Consider minimization problem (2.1) in $S = [0,10] \times [0,10]$, when

$$f_2(x) = -\frac{1}{(x_1-4)^2 + (x_2-4)^2 + 0,7} - \frac{1}{(x_1-2,5)^2 + (x_2-3,8)^2 + 0,73}.$$

In both examples we use Lipschitz constant calculation algorithm given in [16].

First we present results for minimization of function $f_1(x)$. In Figure 6 and Figure 7 calculation time, efficiency and SOF of SJP algorithm using different search orders are given. Although *breadth first* search is not efficient for sequential usage, using it for parallel applications we obtain situation similar to data parallelism. Therefore even SJP algorithm performs quite efficiently.



a) *b)*

Figure 6. Results of experiments for $f_1(x)$ and different search orders and SJP algorithm on *Vilkas*: a) computation time, b) efficiency of the algorithm.

For the sequential BB algorithm the *best first* search rule is the most efficient, but parallel SJP algorithm with this search rule does not produce a good performance. Two factors influence efficiency of this parallel algorithm. First search overhead factor *SOF* is increased and therefore the efficiency is lower. The second drawback deals with non-uniform load balance distribution among processors. Best minimum value exchange modification reduces the search overhead factor and thus improves the efficiency of a parallel algorithm.

In Figure 8 the efficiency of different parallel algorithms using the same Best first search rule is shown. In SJP and SJP SE algorithms the SOF factor is poorly controlled (see, *b*) part of Figure 8).

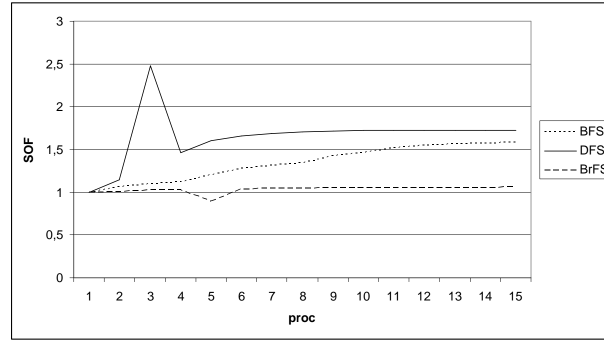
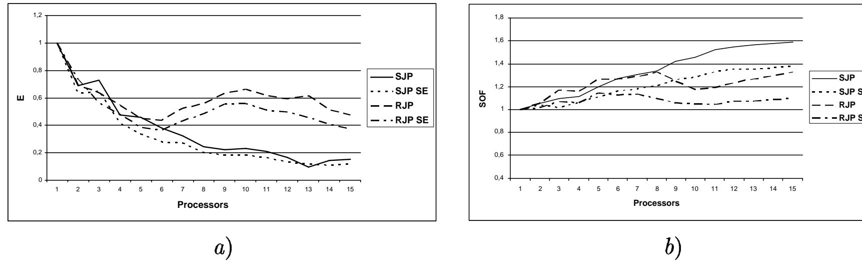


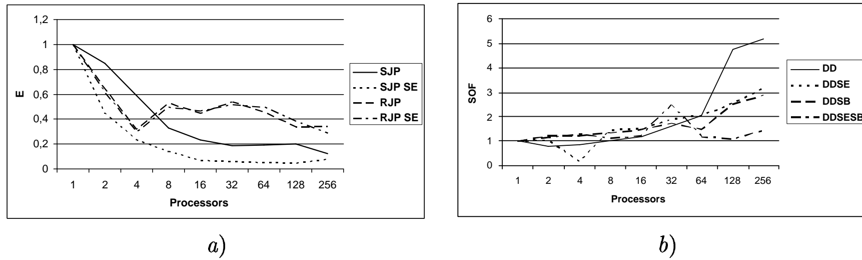
Figure 7. SOF of $f_1(x)$ using different search orders and SJP algorithm on *Vilkas*.



a) *b)*

Figure 8. Efficiency and SOF of $f_1(x)$ using different parallel BB algorithms and the Best first search rule on *Vilkas*.

To test the scalability of the algorithms experiments on up to 256 processors of on IDRIS supercomputer were performed. In Figure 9 the efficiency and the SOF of different parallel algorithms using *best first* search rule calculations is shown. The best speedups up to 90 was obtained for RJP algorithm.

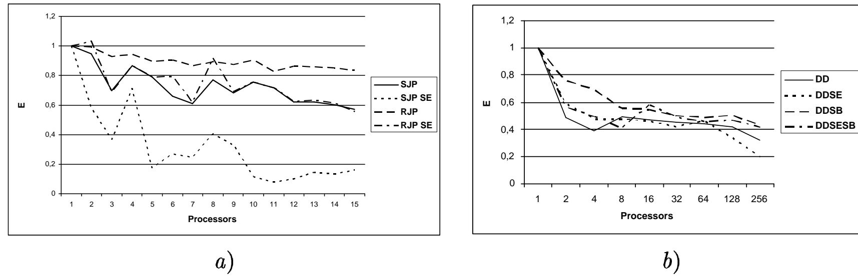


a) *b)*

Figure 9. Efficiency and SOF of $f_1(x)$ best first search using different algorithms on *IDRIS*.

Using the *breadth first* search for the same function (Figure 10) the best efficiency is obtained using simple DD and DD SB methods. Efficiency using

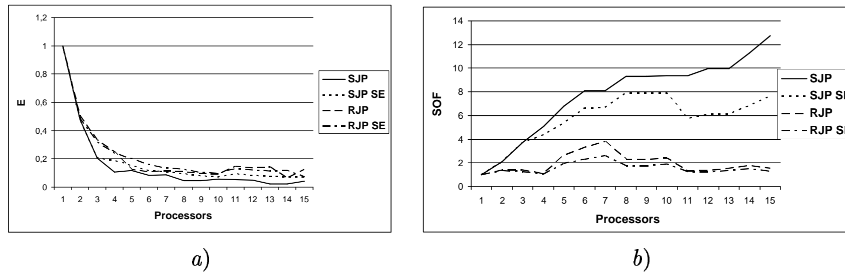
more processors are shown in *b* part of Figure 10. Speedups up to 110 were obtained for RJP algorithm on 256 processors.



a) *b)*

Figure 10. Efficiency of different parallel BB algorithms using the Breadth first search on *Vilkas* and *IDRIS*.

For function $f_2(x)$, Figure 11 show the efficiency of different parallel algorithms using the *best first* search rule. As for function $f_1(x)$, in SJP and SJP SE algorithms the SOF factor is poorly controlled (see, *b* part of Figure 11), but there's a great processor load disbalance for RJP SE algorithm shown in Figure 12. Here *'max', 'min' and* 'avg' shows maximum, minimum and average number of tasks executed among the processors.



a) *b)*

Figure 11. Efficiency and SOF of $f_2(x)$ using different parallel BB algorithms and the Best first search rule on *Vilkas*.

In Figure 13 speedup and SOF obtained on IDRIS are shown.

Traveling salesman problem

Given a collection of cities and the cost of travel between each pair of them, the traveling salesman problem, or TSP for short, is to find the cheapest way of visiting all of the cities and returning to the starting point. We will restrict to the case when the travel costs are symmetric in the sense that traveling from city A to city B costs just as much as traveling from B to A [19].

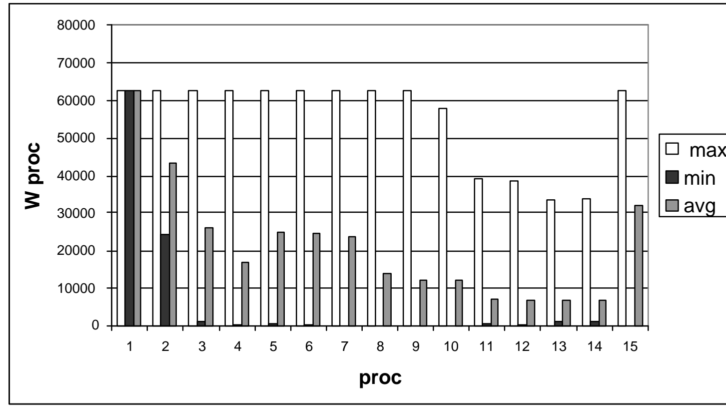
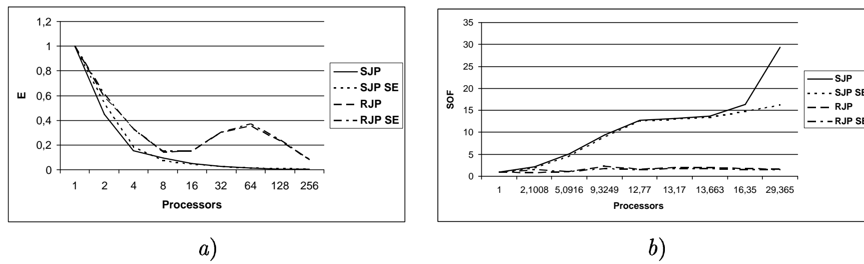


Figure 12. Disbalance of processor load of $f_2(x)$ using different parallel BB algorithms and the Best first search rule on *Vilkas*.



a) *b)*

Figure 13. Efficiency and SOF of $f_2(x)$ best first search using different algorithms on *IDRIS*.

Mathematical formulation is: Given an undirected graph $G = \{N, A\}$ consisting of n nodes and m arcs together with costs c_{ij} for each arc $(i, j) \in A$, the traveling salesman problem is to find a tour of minimum cost

$$\min \sum_{i,j} c_{ij} x_{ij},$$

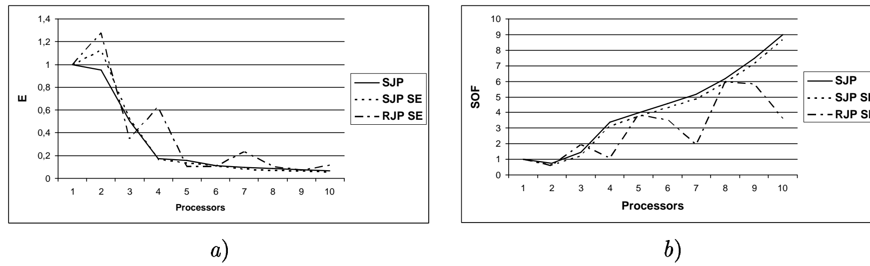
subject to

$$\sum_{j:(i,j)\in A} x_{ij} = 1, i = 0, 1, ..., n-1,$$

$$\sum_{i:(i,j)\in A} x_{ij} = 1, j = 0, 1, ..., n-1$$

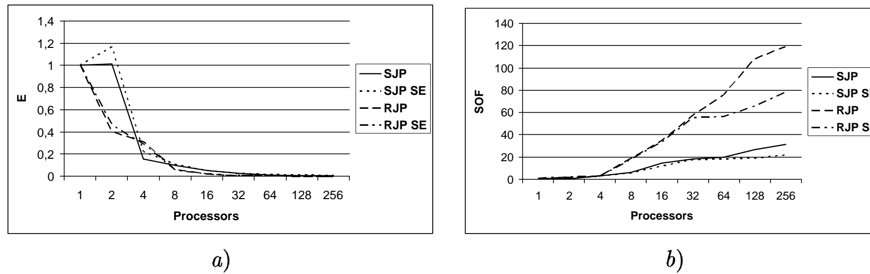
and $x_{ij} \in \{0, 1\}$

The tour over 20 cities was calculated. The efficiency and SOF of different parallel algorithms using the Best first search rule are given in Figure 14 for *Vilkas* cluster and Figure 15 for *IDRIS*.



a) *b)*

Figure 14. Efficiency and SOF of TSP best first search using different algorithms on *Vilkas*.



a) *b)*

Figure 15. Efficiency and SOF of TSP best first search using different algorithms on *IDRIS*.

5. Conclusions

In this work the template for implementation of branch and bound algorithms is presented. The main aim of this tool is to ease the implementation of sequential and parallel programs for covering and combinatorial optimization methods for global optimization. Standard parts of global optimization algorithms are implemented in the template and only method specific rules should be implemented by the user. Parallel programs can be obtained automatically if the sequential program is implemented by the same template.

Parallel algorithms using distributed search space are implemented in the template and algorithms with a single search space can be implemented using Master - slave algorithm and suitable parallelization tool.

The implementation of Best first search sub-space selection strategy is analyzed and heap data structure is shown to more efficient then priority queue for problems with larger list of candidates.

Results of computational experiments are presented for the problem of minimization of the Lipschitz functions and traveling salesman problem. The portability of the template was tested by running it on two different parallel computer systems. RJP SE algorithm performs better for test problems. The increased value of the SOF factor and disbalance of work between processors reduce the efficiency of parallel algorithms implemented in the template. The

static load balancing does not control load disbalance enough and dynamical load balancing algorithms should be used.

6. Acknowledgments

The authors wish to acknowledge the support of the HPC-Europa programme, funded under the European Commission's Research Infrastructures activity of the Structuring the European Research Area programme, contract number RII3-CT-2003-506079.

References

[1] E. Alba and F. Almeida at all. Mallba: A library of skeletons for combinatorical optimization. Technical report, 2001.
[2] M. Baravykait˙ e and R.Šablinskas. The template programming of parallel algorithms. *Matematical Modelling and Analysis*, **7**(1), 11–20, 2002.
[3] R. Čiegis, R. Šablinskas and J. Wasniewski. Hyper-rectangle distribution algorithm for parallel multidimensional numerical integration. In: *Recent advances in PVM and MPI, 6th European PVM/MPI user's group meeting, Barcelona, Spain*, volume 1697 of *Lecture Notes in Computer Science*. Springer, 275–282, 1999.
[4] J. Clausen. Parallel search-based methods in optimization. In: *Applied Parallel Computing - Industrial Computation and Optimization, Proceedings of PARA96*, volume 1184 of *Lecture Notes in Computer Science*. Springer, 176–185, 1996.
[5] T. Csendes. Generalized subinterval selection criteria for interval global optimization. *Numerical Algorithms*, **37**(1–4), 93–100, 2004.
[6] M. Dür and V. Stix. Probabilistic subproblem selection in branch-and-bound algorithms. *Journal of Computational and Applied Mathematics*, **182**(1), 67–80, 2005.
[7] J. Eckstein and W.E. Hart C.A. Phillips. Pico: An object-oriented framework for parallel branch and bound, rutcor research report. Technical Report 40-2000, Rutgers University, Piscataway, NJ, 2000.
[8] C.A. Floudas. *Deterministic Global Optimization: Theory, Methods and Applications*, volume 37 of *Nonconvex Optimization and its Applications*. Kluwer Academic Publishers, 2000.
[9] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
[10] B. Gendron and T.G. Crainic. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, **42**(6), 1042–1066, 1994.
[11] D. Goswami and B. Preiss A. Singh. From design patterns to parallel architecture skeletons. **62**(4), 669–695, April 2002.
[12] A. Grama, A. Gupta, G. Karypis and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.
[13] R. Hammer, M. Hocks, U. Kulish and D.Ratz. *C++ Toolbox for Verified Computing: Basic Numerical Problems*. Springer, Berlin, 1995.
[14] E. Hansen and G.W. Walster. *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, 2nd edition, 2003.

[15] P. Hansen and B. Jaumard. Lipschitz optimization. In: *Handbook of Global Optimization*, volume 2 of *Nonconvex Optimization and Its Applications*, Kluwer Academic Publishers, Dodrecht, 404–493, 1995.

[16] R. Horst, P.M. Pardalos and N.V. Thoai. *Introduction to Global Optimization*, volume 48 of *Nonconvex Optimization and its Applications.* Kluwer Academic Publishers, 2nd edition, 2001.

[17] O. Knüppel. PROFIL/BIAS V 2.0. Technical Report 99.1, Technische Universität Hamburg-Harburg, 1999.

[18] V. Kreinovich and T. Csendes. Theoretical justification of a heuristic subbox selection criterion for interval global optimization. *Central European Journal of Operations Research*, **9**(3), 255–265, 2001.

[19] E.W. Lawler, J.K. Lenstra, A. Rinnooy Kan and D.B. Smoys. *The Traveling Salesman Problem : A Guided Tour of Combinatorial Optimization.* Wiley Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1985.

[20] B. Le Cun and C. Roucairol. Bob: a unified platform for implementing branch-and-bound like algorithms. Technical Report 95/16 sep., Université de Versailles - Laboratoire PR*i*SM, 1995.

[21] R. Šablinskas. *Investigation of algorithms for distributed memory parallel computers.* PhD thesis, 1999.

[22] Y. Shianno and T. Fujier. Pubb (parallelization utility for branch-and-bound algorithms) user manual. Technical Report Version 1.0., 1999.

[23] A. Singh and D. Szafron J. Schaeffer. Views on template-based parallel programming. In: *CASCON 96 CDRom Proceedings,Toronto, October*, 1996.

[24] S. Tschoke and T. Polzer. Portable parallel branch-and-bound library ppbb-lib. user manual. Technical Report Version 2.0., Department of Computer Science, University of Paderborn, 1996.

[25] C. Xu and F. Lau. *Load Balancing in Parallel Computers. Theory and Practice.* Kluwer Academic Publishers, 1997.

Apibendrinto šakų ir rėžių algoritmo šablono realizacija

M. Baravykaitė, R. Čiegis, J. Žilinskas

Straipsnyje pristatyta apibendrinto šakų ir rėžių algoritmo šablono realizacija. Įrankis skirtas palengvinti nuosekliųjų ir lygiagrečiųjų optimizacijos uždavinių programų kūrimą. Nuo uždavinio nepriklausančios algoritmo dalys yra įdiegtos šablone ir vartotojui reikia sukurti tik nuo uždavinio priklausančių dalių realizaciją. Šablone įdiegti keli lygiagretieji algoritmai, paremti tyrimo srities padalinimu tarp procesorių. Pateikiami skaičiavimo eksperimentų rezultatai.