

# SOLVING THE MIXTURE DESIGN PROBLEM ON A SHARED MEMORY PARALLEL MACHINE

J.A. Martínez, L.G. Casado and I. García  
*Dept. of Computer Architecture and Electronics*  
*University of Almería, 04120, Spain*

Eligius M.T. Hendrix  
*Group Operations Research and Logistics, Wageningen University*  
*Hollandseweg 1, 6706 KN, Wageningen, The Netherlands*

## ABSTRACT

A computational model which explores the power of the distributed computing paradigm is the well known Branch and Bound scheme. One of the characteristics of such a scheme is the unpredictable behavior of the model in relation to the set of sub-problems dynamically generated, because this is application dependent. Therefore, specific load balance and task partition strategies have to be applied. This paper deals with a Branch and Bound approach for the semi-continuous quadratic mixture design (SCQMD) problem from a parallel computing perspective. SCQMD is a multi-objective global optimization problem with linear, quadratic and semi-continuity constraints. The Branch and Bound approach for the SCQMD problem is extremely complex from a computational point of view. Practical cases coming from the industry show a high computational complexity that make the problem hard to be solved on a uniprocessor platform. In this work, two parallel approaches are proposed and evaluated on a shared memory distributed computer.

## KEYWORDS

Parallel Branch-and-Bound, shared memory, threads.

## 1. INTRODUCTION

The mixture design problem consists of identifying mixture products, each represented by a vector  $x \in R^n$ , which meet certain requirements. The set of possible mixtures is mathematically defined by the unit simplex  $S = \{x \in R^n \mid \sum_j x_j = 1.0; x_j \geq 0\}$ , where the variables  $x_j$  represent the fraction of the components in a product  $x$ . In mixture design (blending) problems, the cost of the material has to be minimised. In practical situations, such problems are solved on a daily base in fodder and petrochemical industry where often requirements are modelled by linear inequalities, see e.g.(Williams 1993). Three aspects get additional attention in the SCQMD Problem compared to the traditional blending problem: (i) the variables have a semi-continuous character; (ii) the number of used raw materials is minimised in a separate objective function; (iii) the requirements for the product are modelled by quadratic constraints (Hendrix et al. 2006).

The semi-continuity of the variables models a minimum acceptable dose  $md$  that the practical problems reveal, i.e. either  $x_j = 0$  or  $x_j \geq md$ . The number of sub-regions (simplices), due to the semi-continuity can be shown to be:

$$\sum_{t=1}^n \binom{n}{t} = 2^n - 1,$$

where  $t$  denotes the number of raw materials in each sub-simplex (facet).

So far, having linear constraints, a linear objective function and semi-continuous variables, a general MILP (Mixed Integer Linear Programming) solution method can be applied. In practical problems, the linear

constraints ( $h_i(x) \leq 0; i=1,\dots,l$ ) have the interpretation of bounding the design space of production. An additional feature is the appearance of quadratic inequalities that represent the requirements (production specifications) given as:

$$g_i(x) = x^T A_i x + b_i^T x + d_i \leq 0; i=1,\dots,m$$

where  $A_i$  is a symmetric  $n$  by  $n$  matrix,  $b_i$  is an  $n$ -vector and  $d_i$  is a scalar.

The quadratic mixture design problem has been studied in (Hendrix et al. 2006, Hendrix & Pintér 1991), where specific Branch-and-Bound approaches were constructed. Moreover, we are interested in finding solutions  $x$  that have a certain robustness with respect to the quadratic constraints;  $\forall i, g_i(x+r) < 0, \forall r, \|r\| \leq \varepsilon$ . This is done by the Branch-and-Bound algorithm proposed in (Hendrix et al. 2006), which is schematically described in Section 2 and analyzed from a parallel point of view. This parallel approach is described in Section 3 and evaluated in Section 4. This paper ends with conclusions in Section 5.

## 2. B&B ALGORITHM FOR THE SCQMDP

The scheme outlined for solving the optimization problem falls into the general framework of Branch-and-Bound (B&B) algorithms. The basic idea in B&B methods consists of a recursive decomposition of the original problem into smaller disjoint sub-problems until the solution is found. The method avoids visiting those sub-problems which are known not to contain a solution. B&B methods can be characterized by four rules: Branching, Selection, Bounding, and Elimination (Ibaraki 1976, Mitten 1970). For problems where the solution is determined with a desired accuracy, a Termination rule has to be incorporated.

Algorithm 1 describes the multi-objective global optimization strategy to solve the SCQMD Problem. In our case, a sub-problem consists of a simplex denoted by  $C_k$  where  $k$  determines the order in which the simplex was generated and  $t_k$  specifies the number of raw materials of  $C_k$ . In the algorithm (line 6), the evaluation of  $C_k$  is based on the values in all the  $t_k$  vertices denoted by  $v_{k,j}, j=1,\dots, t_k$ , that initially are evaluated or during the splitting process. It concerns the values of the quadratic constraints  $g_i(v_{k,j}) i=1,\dots,m$  the linear constraints  $h_i(v_{k,j}) \leq 0; i=1,\dots,l$  and the cost value  $f(v_{k,j})$ . The cost value is used to update the global upper bound value  $f_i^U$  for  $t=t_k$ . If  $v_{k,j}$  happens to be feasible, it is stored as a Pareto optimal design if  $f(v_{k,j}) \leq f_{t_k}^U$  and the upper bounds are updated accordingly.

### Algorithm 1: Branch-and-Bound algorithm for finding robust solutions of SCQMDP.

	<b>Funct</b> B&B( $n, f, g_1, \dots, g_m, h_1, \dots, h_l, \varepsilon, \alpha$ )	
1.	Set $ns = 2^n - 1$	No. simplices
2.	Set the working list $\Lambda := \{C_1, \dots, C_{ns}\}$	
3.	Set the final list $Q := \{\}$	
4.	<b>While</b> $\Lambda \neq \{\}$	
5.	Select a simplex $C = C_k$ from $\Lambda$	Selection
6.	Evaluate $C$	
7.	Compute a lower bound $f^L(C)$ of $f$ on $C$	Bounding
8.	<b>If</b> $C$ cannot be eliminated	Elimination
9.	<b>if</b> $C$ satisfies the termination criterion	Termination
10.	Store $C$ in $Q$	
11.	<b>else</b>	
12.	Divide $C$ into $C_{ns+1}, C_{ns+2}$	Division
13.	$C = \operatorname{argmin} \{f^L(C_{ns+1}), f^L(C_{ns+2})\}$	
14.	Store $\{C_{ns+1}, C_{ns+2}\} \setminus C$ in $\Lambda$	
15.	$ns = ns + 2$	
16.	Goto 6	
17.	<b>Return</b> $Q$	

Algorithm 1 starts by generating the initial set of  $2^n-1$  sub-problems which defines the search space. This initial set of simplices is stored in the working list  $\Lambda$  (line 2). While the working list is not empty, a simplex  $C_k$  from  $\Lambda$  is selected (Best-First search) and evaluated (lines 5 and 6). If  $C_k$  can not be eliminated (line 8) and neither satisfies the termination criterion (line 9), it is bisected. From the two generated simplices the most expensive simplex is stored in the work list  $\Lambda$  while the algorithm proceeds with the cheapest one (Depth-First search). Those simplices which satisfy the termination criterion are stored in the final list  $Q$  that determines the set of all simplices where a solution of the problem can be located (if any). The algorithm also stores the best solution found for each number of raw materials  $t=1,\dots,n$ . In the following, descriptions of the rules of Algorithm 1 are provided.

**Bounding Rule:** Due to the linearity of  $f$ , the lower bound of the linear cost function  $f$  on  $C_k$  can be taken as:  $f_k^L = \min_{j=1,\dots,t_k} \{f(v_{k,j})\}$ .

**Branching Rule:** Given a simplex  $C_k$ , with number of raw materials  $t_k \geq 2$ , its longest edge is bisected generating two new simplices. In this way, simplices never get a needle shape (Horst 1997). Notice that this does not always generate a new point, as a generated vertex can be shared by different simplices. If all edges are of equal length, the edge in between the cheapest and most expensive vertex is bisected. Therefore, the branching rule generates an on average more expensive and on average cheaper simplex. This helps the selection rule which gives higher priority to *cheaper* simplices.

**Termination Rule:** A simplex is not divided further when its size ( $Size(C_k) = \max_{v,w \in C_k} \|v-w\|$ ) is smaller than the value of the accuracy  $\alpha$ . If it has not been rejected by the elimination rules, it is saved in a final list  $Q$ . Algorithm 1 ends when the working list  $\Lambda$  is empty. List  $Q$  provides the set of simplices where the optimal solution can be found (if any).

**Selection Rule:** The selection rule has been designed to reach two goals: to facilitate discarding simplices with a large number of raw materials and to reduce the memory requirements. The first goal is met by giving priority to simplices with less raw materials and low cost (Best-First search). The second goal is met by applying a Depth-First selection rule. Once a simplex is selected and divided, its cheaper child will be the new selected simplex until no further subdivision is allowed. This reduces the memory requirement of the algorithm.

**Rejection Rule:** The rejection rule consists of a set of tests which takes into account feasibility conditions associated to the linear and quadratic constraints as well as the minimization of the number of raw materials and the cost function. A set of rejection tests has been designed to be applied to the selected simplex:

- *Linear infeasibility test:* If for one of the linear constraints all vertices are infeasible  $h_i(v_{k,j}) > 0$ ;  $j=1,\dots,t_k$ , then  $C_k$  is discarded because  $C_k$  does not fulfill  $h_i(x) \leq 0$ .
- *Quadratic infeasibility test:* Given a simplex  $C_k$  and the set of quadratic constraints  $g_i(v_{k,j})$   $i=1,\dots,m$ ,  $C_k$  can be rejected if  $\forall x \in C_k \exists i | g_i(x) > 0$ , see (Hendrix et al. 2006).
- *Pareto bounding test:* If a feasible solution has been found with  $t_k$  raw materials and a lower cost than  $f_k^L$ , then  $C_k$  can be eliminated, i.e. discard  $C_k$  if  $f_{t_k}^U < f_k^L$  (Multi-objective optimization).

For a wide and theoretical description of these rejection tests consult (Hendrix et al. 2006).

### 3. PARALLEL STRATEGY

Current parallel architectures can be categorized as distributed memory, shared memory or grid, with systems that comprise features for several of these classes. Parallel codes will perform best if the programming model and the hardware match. An overview of the programming models appropriate for efficient algorithms on these architectures can be found in (Bader et al. 2005). Parallelizing B&B methods mainly consists of distributing among processors the set of sub-problems which are dynamically created (Dorta et al. 2006). In order to achieve an efficient parallel method one should be concerned about the following issues:

1. All processors should always be busy handling sub-problems and doing useful work;
2. The total cost of handling all the sub-problems should not be greater than the cost of the sequential method;
3. The overhead due to communications among processors (memory contention) should be small.

It is necessary to point out that B&B algorithms executed in parallel, do not process the sub-problems in the same order as a sequential program does. So, the number of created and/or eliminated sub-problems will depend on the number of processors used in a particular execution. The resulting effect is that a specific parallel execution may create more (sometimes less) sub-problems depending on the number of processors.

Several frameworks for parallel B&B algorithms have been developed. They allow the user to provide custom implementations of certain aspects of the algorithm, simplifying the difficulty of achieving previously mentioned goals (Dorta et al. 2006, Crainic et al. 2006). As will be shown later, our data structures do not facilitate the use of these frameworks or skeletons. On the other hand, some of these skeletons use linked or double linked lists to store the active nodes of the search tree. When the size of the list is large, the complexity of an ordered insertion can become an important percentage of the total time of the sequential algorithm execution. Parallel versions of this code, with local lists per processor, usually obtain super-speedups because the size of the parallel local lists decreases as the number of processors increases.

Here we shall investigate the parallelization of a B&B multi-objective global optimization algorithm (SCQMDP) (Algorithm 1) on a shared memory multi-computer with NUMA (Non Uniform Memory Access). The goal of our research is to analyze the aspects of the algorithm which affect the performance of its parallel implementation. Three main topics are investigated here: the dynamic partition of the problem, the load balancing of the computational burden, and the interaction among sub-problems which are processed in different processors. As a result of the analysis of these topics, two algorithms (LV and SV) have been elaborated.

From a parallel perspective, one should select the way the total workload is subdivided into smaller pieces of work i.e. the size of the problem should be defined. Our algorithm is based on threads. The decision consists of creating many small threads (a thread consists of processing a single simplex), or a few number of heavy threads (a thread consists of processing a part of the search tree). In our algorithm, the simplices are pieces of work that are not fully independent (discussed later). Therefore, following the asynchronous load balancing scheme presented in (Zabatta & Ying 1998), the maximum number of active threads has been set equal to the number of processors ( $NProc$ ); i.e. we choose the option of a heavy thread per processor. A shared variable  $NThreads$  determines the current number of active threads.  $NThreads$  is increased when a thread is created and  $NThreads$  is decreased when a thread finishes its work. At every iteration, working threads check the value of  $NThreads$ . When a thread notices that  $NThreads$  is smaller than the number of processors ( $NProc$ ), a new thread containing half of its own pending sub-problems is generated. Only threads with at least two pending sub-problems can create a new thread. In this way, we are ensuring that as soon as a processor is idle, a new thread is created to feed it. The time an idle processor is waiting for a new thread is given by the time (i) to check that  $NThreads$  is less than  $NProc$ , (ii) to divide the work structure, (iii) to create a new thread and (iv) to migrate the new thread to the idle processor. The migration of threads is done by the operating system and is out of the scope of this work.

The set of sub-problems associated to the new thread are chosen in a way that ensures a balance of the computational work of both threads (the new and the old). Taken into account that the set of sub-problems in a thread are ordered by its potential computational burden (Best-First ordering), the new thread will contain the second, fourth, and so on, sub-problems of the original thread. This strategy helps to create threads with a good balance of the computational burden.

To avoid visiting non promising sub-problems, the best upper bound should be accessible to all processors as soon as it is updated (Pareto bounding tests). Due to this reason, the best upper bound is shared by all processors and its update is done in an exclusive way.

Additionally, the B&B algorithm we are dealing with exhibits a characteristic which makes the problem difficult to solve. The way the set of sub-problems is stored and their interrelation plays an important role in the development of the parallel version of the algorithm. To be more precise, a stored sub-problem contains all the information about one simplex and pointers to its vertices. Simplices and vertices are stored in two separated data structures because different simplices can share several vertices. This is done to save memory consumption and vertex evaluations. Notice that there are many cases where a new simplex is generated but the evaluation of its vertices is not needed because it has previously been done. Balanced binary trees have been used to save access time to the simplex and vertex data structures.

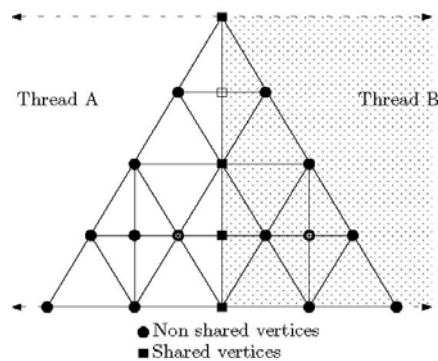


Figure 1. Vertices shared between threads.

In a parallel environment, each thread consists of a subset of sub-problems (simplices)  $C_k$ , with  $t_k$  vertices and many times some of these vertices also belong to simplices located in different threads. Figure 1 shows graphically an instance (two processors and  $n = 3$ ) where a subset of vertices belonging to the set of simplices associated to the thread “A” also belong to the thread “B”. This kind of data structure can be handled in two different ways: (i) to consider that all the sub-problems are fully independent at the cost of evaluating every vertex in all the threads it belongs to (redundant computation); or (ii) to deal with memory contention problems; i.e. every vertex is evaluated just by one processor and the remaining processors take the information from the shared memory.

In this work, two parallel strategies have been analyzed and evaluated. For both approaches, each thread has its local simplex data structure. For the so called LV (Local Vertices) algorithm, every thread contains its own vertex data structure which is not shared with other threads. In contrast, the SV (Shared Vertex) algorithm provides a full access to a single global shared vertex data structure for all the threads. In the following section, both LV and SV strategies are evaluated.

#### 4. EXPERIMENTAL RESULTS

The algorithms were coded in C and were run on an SGI Altix 3000 machine with Intel Itanium2 1.6 GHz processors and Linux operating system with 2.4 smp kernel. POSIX Thread Library was used to create threads.

Some of the details of the experimentation are outlined here for a better understanding of the results presented:

- The queue attribute *max\_user\_run* of OpenPBS (Open Portable Batch System) is set to one. This allows to run only one user program in the system at a time, even when there are free processors available.
- Vertices not shared by any simplex, can be removed to free system memory. These vertices appear when a simplex is rejected. The algorithms presented here do not remove any orphan vertices, because they may belong to a new simplex as the algorithm proceeds. In this way, one avoids to evaluate them again. This generates differences between the LV and SV versions of the algorithm. The SV algorithm stores all the evaluated vertices in the shared binary tree of vertices. The LV version evaluates more vertices, because when a thread has to move half of its simplices to a new thread, only the vertices associated to these simplices are copied to the new thread data structures.
- The algorithms could return the complete set of all simplices and their corresponding vertices where the optimum solution can be located. The number of final simplices can be large and their storage can spent a time that may be considerable with respect to the total algorithm execution time. Algorithms executed here, only store the simplices that contain the vertices with the best *Pareto Cost* as final result.
- For the sake of simplicity, the general description of the B&B algorithm uses working and final lists. We are in favour of the use of binary balanced trees (BBLT), because they have a smaller ordered insertion complexity. Actually, the experiments do not only use one BBLT, but a vector of BBLTs. For instance, the number of raw materials of a simplex is used to identify the vector index that

determines the BBLT where the simplex has to be stored or extracted. This reduces the amount of data in each BBLT, compared to the data stored if just one BBLT is used. Therefore, the insertion complexity is reduced accordingly. The same applies for the vertices. Additionally, the elements (nodes) of a BBLT all have a different key index. BBLTs of vertices do not cause problems because each vertex is indexed by its coordinates and duplicated vertices do not exist. On the other hand, simplices with the same key index do occur. The nodes of simplex BBLTs store a linked list with simplices with the same key index. The list has two sorted indices according to the priority rule: simplices with smaller width first and for simplices with the same width, less vertex cost first. This contributes to maintain a Depth-First search.

Tables 1 and 2 show the computational effort of Algorithms LV and SV, respectively. The data shown in Tables 1 and 2 are the average values of five executions to solve a 7-dimensional problem taken from Unilever (Uni7Spec5b) (Hendrix et al. 2006). The problem was solved with an accuracy of  $\alpha = \sqrt{2}/100$ , a robustness parameter of  $\epsilon = \sqrt{2}/100$ , and a minimal dose of  $md = 0.03$ . The notation used in Tables 1 and 2 is the following:

- *NProc*: Number of Processors.
- *Simplices*: Number of evaluated simplices.
- *Vertices*: Number of evaluated vertices.
- *NThr*: Number of generated threads.
- *Time*: The running time in seconds.

The computational results given in Tables 1 and 2 show that parallel executions always create more simplices than the sequential one. This fact is inherent to Branch and Bound algorithms which can exhibit detrimental or acceleratory anomalies depending on the order the sub-problems are created (Correa & Ferreira 1995). In our case, the selection rule of the algorithm had been optimized for a sequential execution. So, when more processors are used, it may happen that some of the sub-problems which were rejected and not subdivided in the sequential execution, for the parallel executions are not discarded but subdivided, creating two new sub-problems.

Notice that the number of simplices (sub-problems) evaluated by the LV and SV parallel approaches are similar when using the same number of processors (see column *Simplices* in Tables 1 and 2). However, the number of vertices (column *Vertices* in Tables 1 and 2) increases with the number of processors for the LV strategy while stays almost constant for the SV approach. This is the result of creating fully independent data structures for every processor in the LV approach; some of the simplices located in different processors actually share many vertices which are evaluated in several processors. This is the main drawback of the LV approach compared to the SV strategy. The SV strategy does not exhibit computational redundancies with relation to the number of vertices evaluated. On the other hand, the SV strategy has to deal with problems such as memory contention when several processors try to update the tree of vertices (when inserting new vertices and reordering the global balanced binary tree of vertices in a shared memory). Additionally, SV is working on a data structure of vertices which in average is *NProc* times greater than the data structure generated by the LV strategy.

Tables 1 and 2 (column *Time*) show that LV is faster than SV for all the parallel executions and it also exhibits a better scalability. These results are also shown in Table 3 where the speedups obtained by both algorithms are provided.

Table 1. Computational effort for LV algorithm.

<i>Nproc</i>	<i>Simplices</i>	<i>Vertices</i>	<i>NThr</i>	<i>Time (sec.)</i>
1	97,183,929	2,019,349	1	2,311
2	97,288,273	2,315,523	14	1,187
4	97,212,689	2,689,439	101	633
8	97,389,425	3,272,200	311	354
15	97,506,440	3,741,056	670	226

Table 2. Computational effort for SV algorithm.

<i>Nproc</i>	<i>Simplices</i>	<i>Vertices</i>	<i>NThr</i>	<i>Time (sec.)</i>
1	97,183,929	2,019,349	1	2,339
2	97,285,808	2,030,892	13	1,250
4	97,213,242	2,023,836	88	750
8	97,364,397	2,039,874	307	596
15	97,531,861	2,055,322	786	625

Table 3. Speedup and work load imbalance for LV and SV.

<i>Nproc</i>		<b>2</b>	<b>4</b>	<b>8</b>	<b>15</b>
<i>Speedup</i>	LV	1.95	3.65	6.53	10.23
	SV	1.87	3.12	3.93	3.74
<i>Imbalance(%)</i>	LV	7.9	7.4	5.2	4.5
	SV	7.4	4.8	6.3	12.4

Values of the speedup less than linear are mainly due to the redundant computations and to the memory contention problems for LV and SV, respectively. Table 3 gives the values of the imbalance produced by both algorithms with respect to the number of evaluated simplices (sub-problems) per processor. Values of Imbalance are given as  $100 \cdot SD / Av$ , where  $Av$  and  $SD$  are the average and standard deviation values of number of evaluated simplices per processor. Notice that for the LV strategy, the imbalance associated to the number of sub-problems (simplices) is lower than 8%, which means that the tree of sub-problems is well balanced. However, the SV strategy exhibits a relatively large value of the imbalance when running 15 processors (12,4%), which can be one of the reasons for the low value of the speed-up (3,74 for 15 processors). From these results it can be concluded that the dynamic load balancing strategy proposed in this work looks satisfactory for the LV strategy.

## 5. CONCLUSION

Parallel branch and bound algorithms to solve the quadratic mixture design problem have been presented. The difficulty of these problems is the use of two dependent structures of data, one for simplices and one for vertices and the fact that simplices can share vertices. On the other hand, the number of evaluated simplices by the parallel versions is slightly greater than by the sequential version. The parallel versions run on a shared memory machine. We can conclude that algorithms should avoid every type of mutual exclusion access to memory to get reasonable speed-ups. Authors think that the presented dynamic load balancing algorithm can be applied to other branch and bound scenarios on distributed memory machines.

## REFERENCES

- Bader, D. A., Hart, W. E. & Phillips, C. A. (2005), *Parallel algorithm design for branch and bound*, in H. J. Greenberg, ed., *Tutorials on Emerging Methodologies and Applications in Operations Research*, Vol. 76 of *International Series in Operations Research & Management Science*, Springer, chapter 5.
- Correa, R. & Ferreira, A. (1995), *A distributed implementation of asynchronous parallel branch-and-bound*, in A. Ferreira & J. Rolim, eds, *Parallel Algorithms for Irregular Problems: State of the Art*, Kluwer Academic Publisher, Boston (USA).
- Crainic, T., Cun, B. L. & Roucairol, C. (2006), *Parallel branch and bound algorithms*, in E.-G. Talbi, ed., *Parallel Combinatorial Optimization*, Willey, chapter 1, pp. 1–28.
- Dorta, I.; León, C. & Rodríguez, C. (2006), *Performance analysis of branch-and-bound skeletons.*, in *Proceedings of the 14<sup>th</sup> Euromicro Conference on Parallel, distributed and Network-based Processing*, MontBéliard-Sochaux, France pp. 75–82.
- Hendrix, E. M. T., Casado, L. G. & García, I. (2006), *The semi-continuous quadratic mixture design problem*, *Technical Report 24, Mansholt Graduate School*, <http://www.sls.wau.nl/mi/mgs/publications> (submitted to the *European Journal of Operational Research*).
- Hendrix, E. M. T. & Pintér, J. D. (1991), *An application of Lipschitzian global optimization to product design*, *Journal of Global Optimization* 1, 389–401.
- Horst, R. (1997), *On generalized bisection of n-simplices*, *Mathematics of Computation* 66(218), 691– 698.
- Ibaraki, T. (1976), *Theoretical comparisons of search strategies in branch and bound algorithms*, *International Journal of Computer and Information Sciences*. 5, 315–344.
- Mitten, L. G. (1970), *Branch and bound methods: general formulation and properties*, *Operations Research* 18, 24–34.
- Williams, H. P. (1993), *Model Building in Mathematical Programming*, Wiley & Sons, Chichester.
- Zabatta, F. & Ying, K. (1998), *Dynamic thread creation: An asynchronous load balancing scheme for parallel searches*, in *Proceedings of 10th International Conference on Parallel and Distributed Computing and Systems*. Las Vegas, USA.