Two-Dimensional Cutting Stock Problem: shared memory parallelizations

Luis García, Coromoto León, Gara Miranda and Casiano Rodríguez
Dpto. Estadística, I.O. y Computación
Universidad de La Laguna
E-38271 La Laguna, Tenerife, Spain
{lgforte, cleon, gmiranda, casiano}@ull.es

Abstract

Cutting Stock Problems arise in many industries where large stock sheets of a given material must be cut into smaller pieces. Many algorithms have been proposed for solving each of the problem formulations. We present different implementations based on Viswanathan and Bagchi's algorithm to solve the Two-Dimensional Cutting Stock Problem (2DCSP). One approximation parallelizes the generation of new builds from different subproblems. Also a highly efficient data structures to store subproblems are introduced, allowing to provide a parallel implementation where the generation of new subproblems from a particular one can be distributed. The OpenMP tool has been used for the parallel implementations and some computational results are presented.

1. Introduction

Cutting Stock Problems (CSP) arise in many production industries where large stock sheets (glass, textiles, pulp and paper, steel, etc.) must be cut into smaller pieces. CSP can be classified [12, 6] attending to several characteristics: the number of dimensions (1D, 2D, 3D), the number of available surfaces and patterns, the shape of the patterns (regular or irregular), the orientation, etc. Even considering the simplest formulation of the problem, Cutting Stock Problems are classified as NP-Hard problems [5].

The first formulation of the Cutting and Packing Problem as a Linear Programming Problem was made in 1961 [7]. Since that moment a lot of bibliography about the different formulations of the problem has appeared. The Constrained Two-Dimensional Cutting Stock Problem (2DCSP) is one of the most interesting variants of CSP. It targets the cutting of a large rectangle in a set of smaller rectangles finding the set of pieces that get a maximum profit and a minimum loss of the available surface. The solution to the problem has been studied following multiple approximations. Though a large number of heuristics have been proposed the number of exact algorithms is not so extensive. The exact algorithms fall in two categories: depth-first searches [2] and best-first search methods [15, 8, 4]. To our knowledge, not many parallel exact algorithms have been devised [13, 11]. Wang [16] was the first to make the observation that all guillotine cutting patterns can be obtained by means of horizontal and vertical builds of meta-rectangles (Figure 2). Her idea was exploited by Viswanathan and Bagchi [15] to propose a brilliant best first search A^* algorithm (VB) which uses Gilmore and Gomory [7] dynamic programming solution - for the unbounded version of the problem - to build an upper bound. Later, Hifi [8] and Van-Dat, Hifi and Le-Cun [4] proposed a modified version of Viswanathan and Bagchi algorithm (called MVB) introducing an initial lower bound and rules to find in constant time duplicated/dominated patterns. The efficiency of MVB is also a consequence of other two novelties: the use of a bidimensional data structure and a reduced upper bound which combine the VB with the solution of a One-Dimensional Knapsack Problem.

Niklas et al. in [11] proposed a parallel version of Wang's approximation algorithm [16]. Unfortunately, Wang's method does not always yield optimal solutions in a single invocation and is slower than VB algorithm [15]. Tschöeke and Holthöfer parallel version [13] starts from the original Viswanathan and Bagchi algorithm and uses the Paderborn Parallel Branch and Bound Library (PPBB-LIB [14]). Due to the asynchronous nature provided by the PPBB-LIB skeleton, the algorithm does not guarantee the processing of best subproblems first. Another consequence is the generation of unwanted duplicates which aren't produced by the sequential version. In the worst case an exponential growth of elements may result. The authors proposed a stamp-based mechanism to hinder the generation of duplicates.

Initially, we implemented the problem using MaLLBa skeletons [10]. MaLLBa [1] skeletons have been developed in C++ and they provide sequential and parallel solvers

Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)
0-7695-2554-7/06 $20.00 © 2006 IEEE

```
1  OPEN := {T_1, T_2, ..., T_n}; CLIST := ∅; f' := UpperBound();
2  repeat
3    choose α meta-rectangle from OPEN with higher f' value;
4    return(α) if h'(α) = 0;
5    Insert α in CLIST;
6    forall β in CLIST do {
7      /* horizontal build */
8      γ_H = αβ−; γ_H^l = α^l + β^l; γ_H^w = max(α^w, β^w);
9      /* vertical build */
10     γ_V = αβ|; γ_V^l = max(α^l, β^l); γ_V^w = α^w + β^w;
11     γ_H^g = γ_V^g = α^g + β^g;
12     γ_H^i = γ_V^i = α^i + β^i ∀i;
13     if ((γ_H^l ≤ L) and (γ_H^w ≤ W) and (γ_H^i ≤ b_i ∀i)) then
14       Insert γ_H in OPEN;
15     if ((γ_V^l ≤ L) and (γ_V^w ≤ W) and (γ_V^i ≤ b_i ∀i)) then
16       Insert γ_V in OPEN;
17   }
18 forever;
```

Figure 1. Viswanathan and Bagchi's Algorithm

based on a specific algorithmic technique. The 2DCSP implementation done with MaLLBa is also based on VB algorithm. The general structures provided by the skeletons do not allow to obtain good performance results, so we decide to do an ad hoc C implementation. Some improvements have been introduced into this implementation: new data structures for the management of builds and ad hoc parallelizations have been tested. Note that any Best First Search Branch and Bound will greatly benefit of these ideas. In particular, they can be implemented by skeletons giving support to this technique.

The article content will be organized in the following way: An exact algorithm based on VB and its parallel version will be explained in sections 2 and 3. Section 4 exposes some improvements introduced to the data structures in the sequential algorithm. The new scheme gives a chance for the implementation of a new parallelization described in section 5. Computational results of the different implementations will be shown in Section 6. Finally, conclusions and future works are given.

2. Implementation based on linked lists

The Constrained Two-Dimensional Cutting Stock Problem (2DCSP) is one of the most interesting variants of CSP and targets the cutting of a large rectangle S of dimensions $L \times W$ in a set of smaller rectangles T_i, each one with dimensions $l_i \times w_i$, and an associated profit c_i. Let's b_i the number of available rectangles of type i and x_i the number of pieces of type i that have been fit into the large rectangle.

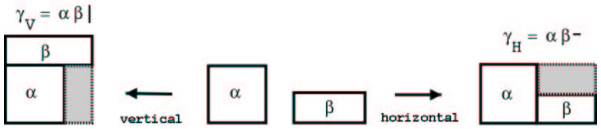

Figure 2. Vertical and horizontal builds

The problem consists in finding the set of pieces and its distribution along the surface S that get a maximum profit and a minimum loss of the available sheet, that is:

$$Maximize \sum_{i=1}^{n} x_i c_i \text{ subject to } x_i \leq b_i \text{ and } x_i \in \mathbb{N}.$$

Our first implementation is based on VB algorithm (Figure 1). The algorithm makes uses of two list OPEN and CLIST to yield the set of feasible solutions. At each step, an element α of size (α^l, α^w) from OPEN is chosen and combined with the elements in CLIST to produce horizontal $\gamma_H = (\alpha\beta-)$ and vertical $\gamma_V = (\alpha\beta|)$ builds (Figure 2).

In order to have a way to know which builds are better and which are worse, it is defined the α *build accumulated profit*, $\alpha^g \equiv g(\alpha)$, as the profit sum of all patterns belonging to the build α. Besides, $h(\alpha)$ is defined as the *maximum profit* obtainable from the remaining area of the surface. So, having a certain build α, its *total profit* is defined as: $f(\alpha) = g(\alpha) + h(\alpha)$. To calculate the *estimated total profit* $f'(\alpha)$, the algorithm uses an upper estimation of $h(\alpha)$, denoted as $h'(\alpha)$.

The presented implementation introduces some features to the original VB algorithm: Duplicated builds are detected and removed when inserting new subproblems into OPEN. Before moving any build from OPEN to CLIST, the algorithm must check dominance between the current subproblem and the ones in CLIST. All dominated rectangles will be removed. Dominance attend to the pieces employed in the build and its dimensions. The implementation also stores the best current solution in order to discard the worst builds.

3. Parallel implementation based on linked lists

The first parallel approach is based on a shared memory scheme to store the build lists (OPEN and CLIST) used during the search process. This feature allows several threads to simultaneously generate new builds from different ones. The bottleneck is to keep OPEN sorted, since threads are simultaneously accessing it. For this reason, the simple insertion of OPENMP pragmas is not sufficient.

The *master-slave* model is in the core of the implementation. Before the threads begin to work, the master generates the initial subproblems. At each step, the master removes the first subproblem from OPEN. If it is a solution,

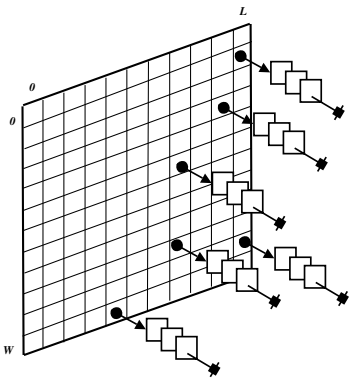

Figure 3. Data structure to store CLIST

the search finishes. In other case, and assuming the subproblem is not dominated, it verifies if someone is combining it or if it has been combined before. If the subproblem is still unbranched the master does the combination. If the subproblem is assigned, the master must wait until the thread which works on it finishes to generate its new combinations. Once all the combinations have been generated, the master inserts them into OPEN. Until the master does not notify the end of the search, each slave works generating new builds from the unexplored subproblems in OPEN.

Problems appear when different threads are simultaneously trying to modify the same shared variable. OpenMP does not provide high-level mechanisms to solve these problems when using with dynamic memory structures, so we had to incorporated extra synchronization and flush operations.

4. Improving Data Structures

In VB original version the combination is achieved traversing the whole CLIST list, discarding non feasible solutions. To alleviate this, Cun and others [4] introduced the data structure depicted in Figure 3. This way (shown in Figure 4) uses two loops, one for the horizontal combinations (lines 7-16) and another for the vertical combinations (lines 17-26) and only problems holding the geometry constraints are visited. There is one loss however. Observe that $(\alpha\beta-)^i = (\alpha\beta|)^i$ and $(\alpha\beta-)^g = (\alpha\beta|)^g$ for any α and β and any pattern i. When using the original list data structure this common values are computed only once. The decoupling on two loops implies the repetition of such calculations (lines 11 and 21). We can reduce this overhead as follows: During the horizontal loop (lines 7-16) we save a pointer to α inside the data structure representing the meta-rectangle β (for that we use an extra field, let us call it $current$). On a second field (call it $horizontal$) we store

```
1  OPEN := {T_1, T_2, ..., T_n}; CLIST := ∅; f' := UpperBound();
2  BestSol := Heuristic(); B: = BestSol^g;
3  repeat
4    choose α meta-rectangle from OPEN with higher f' value;
5    return(Bestsol) if B = f'(α);
6    Insert α in CLIST at entry (α^l, α^w);
7    for x := 0 to L - α^l do {
8      forall β ∈ CLIST_x such that β^i ≤ b_i − α^i do {
9        /* horizontal build */
10       γ = αβ−; γ^l = α^l + β^l; γ^w = max(α^w, β^w);
11       γ^g = α^g + β^g; γ^i = α^i + β^i ∀i;
12       if (γ^g > B) then { free OPEN from B to γ^g;
13                           B = γ^g; BestSol = γ; }
14       if (f'(γ) > B) then Insert γ in OPEN at entry f'(γ);
15     }
16   }
17   for y := 0 to W - α^w do {
18     forall β ∈ CLIST_y such that β^i ≤ b_i − α^i do {
19       /* vertical build */
20       γ = αβ|; γ^l = max(α^l, β^l); γ^w = α^w + β^w;
21       γ^g = α^g + β^g; γ^i = α^i + β^i ∀i;
22       if (γ^g > B) then { free OPEN from B to γ^g);
23                           B = γ^g; BestSol = γ; }
24       if (f'(γ) > B) then Insert γ in OPEN at entry f'(γ);
25     }
26   }
27   return(Bestsol) if OPEN = ∅;
28 forever;
```

Figure 4. Modified Version of Viswanathan and Bagchi's Algorithm

a pointer to $\alpha\beta-$. Now if during the vertical loop (lines 17-26) the meta-rectangle β (line 18) has its $current$ field pointing to α we can recover the values $\alpha^i + \beta^i$ stored in $\alpha\beta-$ using the $horizontal$ field of β.

On any best-first search algorithm subproblems are sorted by the value of their upper bounds. Maintaining this usually very large set is often cause for performance degradation. Since along the execution of any Branch and Bound the lower bounds keep ascending and the upper bounds descending we can state that during the search all the upper bounds fall in the interval $[best_0, upper_0]$. We denote by $best_0$ the initial heuristic value and by $upper_0$ the maximum upper bound of the initial subproblems. That suggest a natural solution: to have an array $[best_0..upper_0]$ of pointers to linked lists of subproblems. Subproblems with the same upper bound go to the same linked list. Ties in OPEN are resolved according the value of g. Insertion then can be done in constant time. Notice that insertion using the classical list approach [15, 16] leads - for VB algorithm but it is also the general case for Branch and Bound - to $\mathcal{O}(2^n)$ time since in the worst case the list grows exponentially with the num-

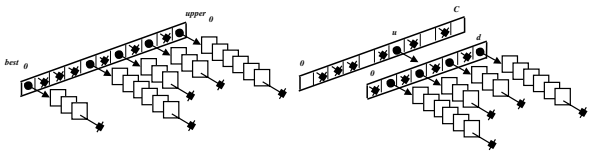

Figure 5. Data Structure to store OPEN

ber of patterns. The other main operation involved, choosing/extracting the subproblem with the largest upper bound consists now in descending the interval searching for a non void pointer. Full segments of memory can be freed any time the lower bound improves (lines 22-23, Figure 4) or the maximum upper bound decreases.

When memory is an issue and there is no space to afford to store the whole interval $[best_0, upper_0]$ the data structure becomes a tree-of-intervals (Figure 5). The root node is now a smaller interval $[0, C]$ where $C = \frac{upper_0 - best_0}{d}$. Each item in $[0, C]$ is a pointer to an interval of size $\frac{C}{d} = \frac{upper_0 - best_0}{d^2}$ and so on. The idea is extremely simple - but as far as we know it is the first time that it is proposed - and it can be applied to any best first search Branch and Bound algorithm and therefore to Algorithmic Skeletons [1, 3, 14] supporting this technique.

5. Parallel implementation based on the bidimensional structure

In this new parallel approach, each processor works in a section of the bidimensional CLIST. It combines the current best subproblem with the subproblems contained in its section. Each processor keeps a replicated copy of CLIST. The work distribution can be a dynamic or static (cyclic, block). On the other side, OPEN is distributed and only contains the builds generated by its owner. These structures allow the processors to work independently in the generation of new subproblems. After this step, each processor has its own best current subproblem. To determine global best current build, an all-to-all reduction is performed. The same reduction is used to update the best solution current value. This reduction and synchronization scheme is shown in Figure 6.

Each thread initializes its OPEN and CLIST variables. The master thread creates the initial subproblems and inserts them into its OPEN list. At every iteration of the search loop, the global best subproblem is identified. Each thread makes public its best subproblem by updating the corresponding entry of a shared static structure. The master thread determines the thread identifier having the best current subproblem and writes it to a shared variable. This value is copied by the threads into a private variable. Besides, the best subproblem owner must remove it from its OPEN list. If the subproblem is not the solution and there are no other similar or better subproblems it is inserted in the local CLIST. If the subproblem is discarded, go to the previous step. The horizontal new builds are generated in the first loop and in the second are generated the vertical ones. These loops have a *parallel-for* pragma, so each thread will do combinations of the subproblem with certain sections of the CLIST matrix. The new subproblems are inserted into the corresponding thread OPEN list. Once all the new subproblems have been created and inserted into the lists, each thread must find its current best subproblem and copy it to the static shared array. The same is done with the best solution. These steps must be followed until the solution is found.

The main problem of the implementation deals with the use of dynamic linked lists. These lists have to be modified by all threads and there is no mechanism available to ensure the integrity of data. OpenMP compilers usually ensure the integrity of static structures, that is, arrays or structs stored in the static segment or in the execution stack. But it is not the case when dealing with data structures allocated in the heap. By this reason, some additional operations are necessary to update the shared dynamic data structures used in our implementation.

6. Computational Results

The experiments have been run on a machine with 4 Intel Xeon 1.4GHz, and on an Intel Itanium 2 1.5GHz cluster. The instances have been selected for the ones available on the literature. Each experiment has been executed five times and the results presented is the average.

Table 1 shows the results for the instances $1\_$, $A4$ and $A5$ described in [8]. Columns labelled "Time" show execution times in seconds and the labelled "Comp." show the number of average computed nodes. The results grouped under the name "Initial Version" are for the sequential and parallel executions for the algorithm described in section 2 on the multiprocessor. Speedups highly depend on the input problem. Sequential times for the modified version described in section 4 are also shown in the table under the label "Improved Version". As we can see, the modified sequential implementation introduces great improvements over the original version.

The results for the other instances studied in [8] (that is, 1, 2, 3, $2\_$, $3\_$, $A1$, $A2$, $A3$ and H) are not shown because the times obtained for the improved version are more drastic and there is no chance for parallel improvements. Then, we have selected the instances problem 1 from category 1 (cat1_1) and problem 2 from category 3 (cat3_2) from the ones proposed at [9]. New data structures allow to obtain really better times. They make possible to easily sort elements in OPEN and find duplicated/dominated nodes. But,

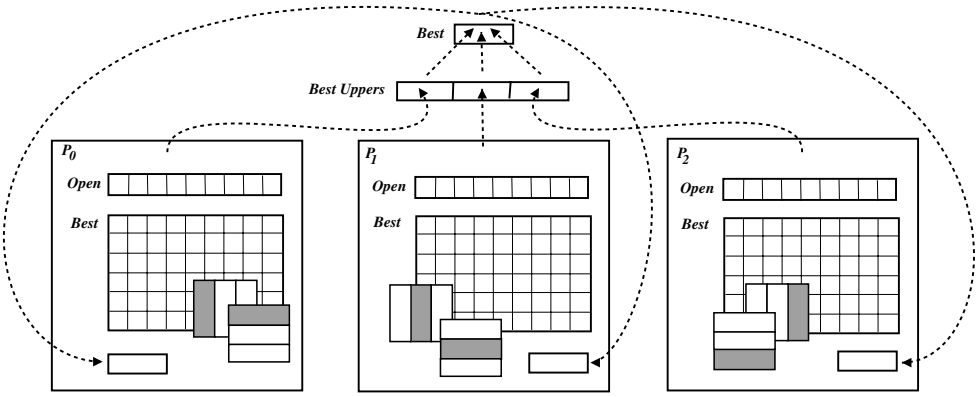

Figure 6. Reduction-Barrier Scheme

when the number of generated nodes increases, the insertion of subproblems into OPEN turns too heavy for the first implementation. By this reason, large problem instances are not approachable by this version. However, they can be solve by the parallel improved algorithm. The Table 2 presents the results obtained on the Intel Itanium cluster. The columns labelled "Ins." and "Gen." shows the number of nodes, in thousands, inserted and generated respectively. Parallel speedups strongly depend on the particular problem. That is a result of changing the search space exploration order when more than one thread is collaborating in the resolution. Even in worse cases (cat1_1) we can improve sequential times. A fair work load distribution between threads is difficult to obtain since it is not only needed to fairly distribute the subproblems to generate but also the ones to be inserted. Before doing a certain combination we are not able to know if it will be valid or not (to be inserted).

7. Conclusions

Two sequential algorithms for the resolution of the Two-Dimensional Cutting Stock Problem have been presented. Both algorithms are based on Viswanathan and Bagchi's Algorithm. The second approach introduces some improvements to the initial version. New data structures are design in order to efficiently manage insertions, combinations and dominance detections. Now insertion time does not depend on the number of nodes in the lists and can afford the resolution of larger problem instances.

Also, we have studied the parallelizations introduced on both algorithms. Parallel versions have been implemented using OpenMP. In the parallelization of the initial scheme, threads work simultaneously in the generation of new subproblems from different builds. When many different threads work over a same dynamic structure (OPEN) data integrity/consistence problems may appear. OpenMP does not provide any mechanisms to solve these problems when working with dynamic memory structures, so we had to incorporated some extra synchronization and flush operations. The extra operations involved too overhead to the scheme in comparison to the time invested in the combination of a subproblem with all subproblems in CLIST. The parallel version obtains better times because when all threads work together, they can get a better current solution earlier. This makes possible to discard a lot of builds than in the sequential scheme had to be inserted into OPEN. A parallelization of the new build generation loop is proposed. In this case, less work is distributed at each step and the load distribution is worse than in the other parallel version. Super and sublinear speedups may occur since the parallel algorithms alters the sequential order. Porting an existing C application to OpenMP even if the algorithm is straightforwardly parallel can be sometimes quite difficult due to the lack of support to qualify dynamic memory variables as shared or private.

Future works targets improvement of the modified algorithm and its parallel version. New algorithm bounds are been studied. In relation to the parallel algorithm we would like to develop a message passing implementation in order to compare with the ones presented. A depth study of the work load distribution is also required.

8. Acknowledgements

This work has been supported by the EC (FEDER) and by the Spanish Ministry of Education inside the 'Plan Nacional de I+D+I with contract number TIC2005-08818-C04-04. The work of G. Miranda has been developed under the grant FPU-AP2004-2290.

Table 1. First Approximation - Sequential and Parallel results

	Initial Version						Improved Version	
	Sequential		2 Threads		4 Threads		Sequential	
Problem	Comp.	Time	Comp.	Time	Comp.	Time	Comp.	Time
1_	3502	2,78	4136	4,90	4044	3,39	578	0,079
A4	864	75,94	854	9,51	860	7,36	374	0,174
A5	1674	6,17	1510	6,63	1526	3,13	670	0,212

Table 2. Improved version - Parallel results

	Thread 1		Thread 2		Thread 3		Thread 4		Thread 5		Thread 6		Thread 7		Thread 8			
	Ins.	Gen.	Ins.	Gen.	Ins.	Gen.	Ins.	Gen.	Ins.	Gen.	Ins.	Gen.	Ins.	Gen.	Ins.	Gen.	Comp.	Time
cat1_1																		
Th 1	122	71631															42	112.09
Th 2	72	35551	52	36079													50	73.49
Th 4	39	18896	28	19874	35	18093	26	17515									56	73.96
Th 8	20	11443	16	13898	14	8216	15	9427	18	7068	10	5657	20	9503	10	7894	57	68.99
cat3_2																		
Th 1	145	5067															10	32.08
Th 2	43	2621	44	2919													11	5.59
Th 4	32	175	31	179	23	148	23	110									3	0.77
Th 8	30	200	19	126	21	184	17	128	25	172	28	228	12	112	22	117	5	1.01

References

[1] E. Alba and et al. MaLLBa: A Library of Skeletons for Combinatorial Optimization. In *Proceedings of Euro-Par*, volume 2400 of *Lecture Notes in Computer Science*, pages 927–932, Paderborn (GE), 2002. Springer-Verlag.

[2] N. Christofides and C. Whitlock. An Algorithm for Two-Dimensional Cutting Problems. *Operations Research*, 25(1):30–44, January-February 1977.

[3] B. L. Cun and C. Roucairol. BOB : a unified platform for implementing branch-and-bound like algorithms, June 29 1995.

[4] V. D. Cung, M. Hifi, and B. Le-Cun. Constrained Two-Dimensional Cutting Stock Problems: A Best-First Branch-and-Bound Algorithm. Technical Report 97/020, Laboratoire PRiSM - CNRS URA 1525. Universit de Versailles, Saint Quentin en Yvelines. 78035 Versailles Cedex, FRANCE, November 1997.

[5] K. A. Dowsland and W. B. Dowsland. Packing Problems. *European Journal of Operational Research*, 56(1):2–14, 1992.

[6] H. Dyckhoff. A Typology of Cutting and Packing Problems. *European Journal of Operational Research*, 44(2):145–159, 1990.

[7] P. C. Gilmore and R. E. Gomory. The Theory and Computation of Knapsack Functions. *Operations Research*, 14:1045–1074, March 1966.

[8] M. Hifi. An Improvement of Viswanathan and Bagchi's Exact Algorithm for Constrained Two-Dimensional Cutting Stock. *Computer Operations Research*, 24(8):727–736, 1997.

[9] E. Hopper and C. H. Turton. An Empirical Investigation of Meta-heuristic and Heuristic Algorithms for a 2D Packing Problem, 1999. http://people.brunel.ac.uk/˜mastjjb/jeb/orlib/files/strip1.txt.

[10] G. Miranda and C. Len. An OpenMP skeleton for the A* heuristic search. In *High Performance Computing and Communications (HPCC'05)*, pages 717–722. Springer-Verlag, 2005.

[11] L. D. Nicklas, R. W. Atkins, S. K. Setia, and P. Y. Wang. The Design and Implementation of a Parallel Solution to the Cutting Stock Problem. *Concurrency - Practice and Experience*, 10(10):783–805, 1998.

[12] P. E. Sweeney and E. R. Paternoster. Cutting and Packing Problems: A categorized, application-orientated research bibliography. *Journal of the Operational Research Society*, 43(7):691–706, 1992.

[13] S. Tschöke and N. Holthöfer. A New Parallel Approach to the Constrained Two-Dimensional Cutting Stock Problem. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, pages 285–300, Berlin, Germany, 1995. Springer-Verlag.

[14] S. Tschöke and T. Polzer. Portable parallel branch-and-bound library - PPBB-lib, Dec. 19 1996.

[15] K. V. Viswanathan and A. Bagchi. Best-First Search Methods for Constrained Two-Dimensional Cutting Stock Problems. *Operations Research*, 41(4):768–776, 1993.

[16] P. Y. Wang. Two Algorithms for Constrained Two-Dimensional Cutting Stock Problems. *Operations Research*, 31(3):573–586, May-June 1983.