Improving the Efficiency of a Best-First Bottom-Up Approach for the Constrained 2D Cutting Problem

Gara Miranda*, Jesica de Armas, Carlos Segura, Coromoto León

Dpto. Estadística, I. O. y Computación. Universidad de La Laguna.
38271 - La Laguna. Santa Cruz de Tenerife, Spain.
Phone number: +34 922 318180
Fax number: +34 922 318170

Abstract

This work introduces several improvements in the solution of the Constrained 2D Cutting Problem. Such improvements combine the detection of dominated and duplicated cutting patterns with the implementation of parallel approaches for best-first search methods. The analysis of symmetries and dominances among the cutting patterns is able to discard some non-optimal or redundant builds, thus reducing the search space to be explored. The experimental evaluation demonstrates that when the domination/duplication rules are applied to an efficient parallel approach, the obtained reduction in the number of managed nodes involves a noticeable decreasement on the computational effort associated to the final search process.

Key words: Cutting Problems, Parallel Algorithms, Pattern Dominances and Symmetries, Exact Tree Searches

*Corresponding author

Email addresses: `gmiranda@ull.es` (Gara Miranda), `jdearmas@ull.es` (Jesica de Armas), `csegura@ull.es` (Carlos Segura), `cleon@ull.es` (Coromoto León)

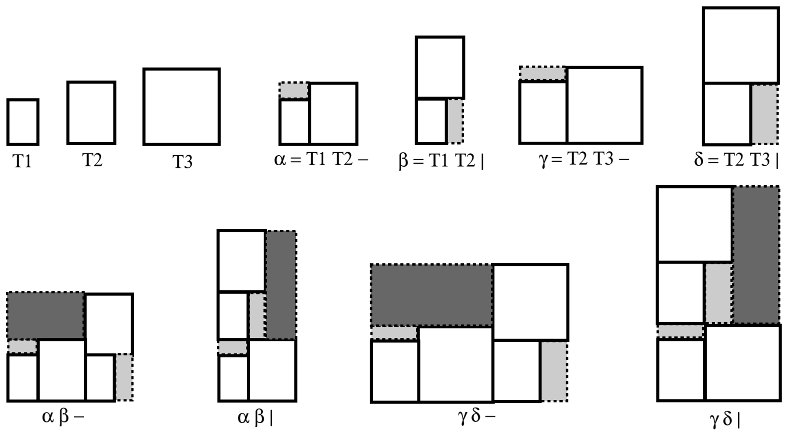


Figure 1: Vertical and horizontal meta-rectangles. Shaded areas represent waste.

1. Introduction

Cutting Problems (CPs) arise in many production industries where large stock sheets (glass, textiles, pulp and paper, steel, etc.) must be cut into smaller pieces. CPs are usually analyzed together with packing problems and can both be classified attending to several characteristics [1, 2, 3]: the number of dimensions, the number of available surfaces and patterns, the shape of the patterns, the orientation, etc. This work is focused on a variant of the Constrained 2D Cutting Problem (2DCP) which targets the cutting of a large rectangle S of dimensions $L \times W$ in a set of smaller rectangles using orthogonal non-staged guillotine cuts. That means that any cut must orthogonally run from one side of the rectangle to the other end producing two new rectangles. The produced rectangles must belong to one of a given set of rectangle types $\mathcal{D} = \{T_1 \dots T_n\}$ where the i-th type T_i has dimensions $l_i \times w_i$. Associated with each type T_i there is a profit p_i and a demand constraint b_i. The goal is to find a feasible cutting pattern with x_i pieces of type T_i maximizing the total profit:

$$\max \sum_{i=1}^{n} x_i p_i \text{ subject to } x_i \in \{0, 1, \dots, b_i\}$$

The posed problem is an output maximization problem where the large objects are only supplied in limited quantities which do not allow for accommodating all small items, so the usage of the available large objects must be maximized in order to arrange a maximum value of small items. Moreover, all dimensions are fixed and the assortment of small items is weakly heterogeneous. Analyzing the problem from this point of view, one of the most recent typologies for cutting and packing [3] suggests to denote the problem as "*Single Large Object Placement Problem*". However such a name is just a suggestion in order to unify the notation in the cutting and packing field, because the same problem appears in the related literature under several different names: Template-Layout Problem, Two-Dimensional Cutting Problem, Rectangle Packing Problem, Orthogonal Knapsack Problem, or Cutting Stock Problem. In particular, the majority of the studies related to the here presented proposals [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] use the name *Cutting Stock Problem* to denote the problem we are analyzing.

Though a large number of heuristics have been devised for the solution of this cutting problem [5, 6, 17, 18], the number of exact algorithms is not as extensive. However, since the profit obtainable from the raw material is a crucial issue for most production industries, the study and design of exact approaches may appear as an interesting research field. Similar to what happens with other combinatorial optimization problems, most exact approaches for the 2DCP are based on an enumeration of the solution search space. Such algorithms can generally assume that the state space is represented in the form of a tree. There is an evaluation function associated with the nodes of the tree, and an orderly search procedure is applied for visiting the nodes. In a branch-and-bound formulation, the order for traversing the tree is usually just a *depth-first* search, so that no evaluation function is required to establish the order among nodes [4, 19]. However, in such a case, the evaluation function is used to measure the newly created nodes - potential solutions - against the best existing solution. This way, the tree is pruned by avoiding the exploration along paths which cannot yield solutions better than the current one. On the other hand, in a *best-first* search, the order of traversal of the search graph is determined by the evaluation function [9, 11, 12]. Moreover, the search typically stops as soon as a complete solution - goal node - is found. If the evaluation function satisfies certain conditions, then this scheme guarantees optimal solutions.

Both tree-search branch-and-bound procedures has been successfully applied to the posed 2DCP and also to problems where restricted three-staged patterns are considered [20, 21, 22]. However, since best-first approaches have shown a more efficient behavior [9], in this work we have focused on the original Viswanathan and Bagchi's algorithm (VB). VB algorithm [9] is based on Wang's bottom-up proposal on building vertical and horizontal

Algorithm 1 Viswanathan and Bagchi's Algorithm

```
1: OPEN := $\{T_1, T_2, \ldots, T_n\}$;  CLIST := $\emptyset$;  f' := UpperBound();
2: repeat
3:    Choose $\alpha$ meta-rectangle from OPEN with highest $f'$ value;
4:    if $(h'(\alpha) == 0)$ then
5:       return($\alpha$);
6:    end if
7:    Insert $\alpha$ in CLIST;
8:    for all $\beta$ in CLIST do
9:       $\gamma_H = \alpha\beta-$; $l_{\gamma_H} = l_\alpha + l_\beta$; $w_{\gamma_H} = \max(w_\alpha, w_\beta)$;    /* horizontal build */
10:      $\gamma_V = \alpha\beta|$; $l_{\gamma_V} = \max(l_\alpha, l_\beta)$; $w_{\gamma_V} = w_\alpha + w_\beta$;    /* vertical build */
11:      $g(\gamma_H) = g(\gamma_V) = g(\alpha) + g(\beta)$;
12:      $x_i(\gamma_H) = x_i(\gamma_V) = x_i(\alpha) + x_i(\beta)\ \forall i$;
13:      if $((l_{\gamma_H} \leq L)$ and $(w_{\gamma_H} \leq W)$ and $(x_i(\gamma_H) \leq b_i \forall i))$ then
14:         Insert $\gamma_H$ in OPEN;
15:      end if
16:      if $((l_{\gamma_V} \leq L)$ and $(w_{\gamma_V} \leq W)$ and $(x_i(\gamma_V) \leq b_i \forall i))$ then
17:         Insert $\gamma_V$ in OPEN;
18:      end if
19:   end for
20: until forever
```

combinations of patterns [5] (see Figure 1). Wang's proposal studies a less general formulation of the problem where the values of the demanded rectangles are directly proportional to their areas. The method is heuristic and does not guarantee the achievement of optimal solutions, although several improvements have been proposed [7, 8, 23]. VB algorithm uses Gilmore and Gomory's [24] dynamic programming solution - for the unbounded version of the problem - to build an upper bound. VB resembles A* algorithms and uses two lists, OPEN and CLIST, to yield the set of feasible solutions. At each iteration of VB algorithm, the element α - with size $l_\alpha \times w_\alpha$ - from OPEN with better upper bound is chosen and combined with the elements in CLIST to produce horizontal $\gamma_H = (\alpha\beta-)$ and vertical $\gamma_V = (\alpha\beta|)$ builds (see Algorithm 1).

Later, Hifi [11] and Cung et al. [12] proposed a modified version of VB algorithm (called MVB) introducing an initial lower bound, a bi-dimensional data structure to manage CLIST (thus decoupling the generation loop in two: one for the horizontal combinations and other for the vertical ones), a reduced upper bound, and some rules to find in constant time duplicated/dominated patterns. However, most recent works based on VB algorithm have been carried out within our group [25, 26, 27, 28, 29, 30]. Such works include several improvements over the sequential VB algorithm (new lower and upper bounds, efficient data structures to manage builds, new reformulation of the problem, etc.) and also some parallel proposals.

This work introduces several improvements over the authors previous approaches. Improvements are based on the detection of dominated and duplicated cutting patterns. This way, some non-optimal or redundant builds are discarded, so that the search space can be significantly reduced. Such reduction involves a decreasement on the computational effort associated to the search process. However, for real-world instances, the search space is still quite large, so the design of parallel approaches is decisive. The introduction of dominance/duplication rules in the algorithm affects the search process, so it is necessary to design parallel algorithms which can manage the highly irregular and unpredictable search space, thus enhancing the efficiency of the search process. In this sense, it is important not only the design of the algorithm but also the implementation, so two different parallel algorithms have been designed and both have been implemented using both shared memory (OpenMP) and distributed memory paradigms (MPI). Section 2 is devoted to describe the type of domination and duplication rules applied to discard non-essential cutting patterns. The parallel algorithms where the dominance rules have been introduced are presented in section 3. Section 4 shows the obtained computational results. Finally, the conclusions and some lines of future work are given in section 5.

2. Duplicated and Dominated Cutting Patterns

The algorithm operation involves the exploration of all promising branches of the tree search, thus ensuring the achievement of the optimal solution. The usage of efficient lower and upper bounds makes it possible to reduce the search space significantly [29], although it is still very large for most real instances. In order to reduce the search space even more, another possibility lies in detecting duplicated or dominated tree branches. In this sense, there exists studies on graph-theoretical models [31], although we are going to focus on the analysis on some simple problem-specific properties and rules. For example, in the here posed problem, *duplicated* branches represent equivalent cutting patterns, i.e. physically, they represent the same usage of the stock sheet, although the same set of final pieces can be obtained through different cutting processes (see Figure 2). *Dominated* branches, on the other hand, represent non-promising cutting patterns, i.e. there exists a “similar” build which improves somehow on the dominated one. Figure 3 shows a dominated cutting pattern, γ_1, which has the same dimensions as γ_2 but includes fewer pieces. In order to improve the efficiency of the algorithm, it is essential

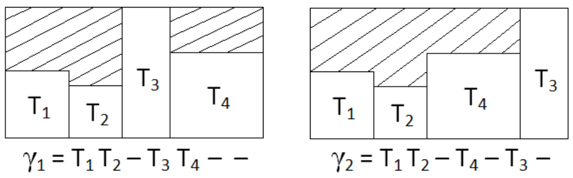


Figure 2: Duplicated patterns

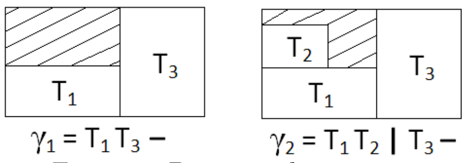


Figure 3: Dominated pattern γ_1

to find and delete duplicated and dominated patterns, so that only one of them is explored and no redundant exploration is done. There are different alternatives for introducing rules to detect duplicated or dominated patterns: they can be applied in the pattern generation process (*pre-generation rules*) or afterwards (*post-generation rules*).

2.1. Pre-generation Rules

Before doing a horizontal or vertical combination of patterns, it is worthwhile to check certain conditions in order to avoid the creation of repeated or non-promising constructions. Three different types of duplication/dominance rules are proposed. They are based on those proposed in [12], although here, some improvements are introduced and a completely new formulation is proposed in order to simplify their understanding and implementation.

2.1.1. Type D1: dominated patterns in generation

Sometimes the combination of two different patterns produces a new build which may introduce considerable waste of the stock sheet. When the empty space obtained from the combination of the two builds is large enough to accommodate one or more of the available pieces, we can avoid the generation of such a construction. It is obvious that this construction will not lead to an optimal solution because there is a possible build which has the same dimensions but employs the space better, introducing extra pieces in the free space, and so obtaining a higher profit.

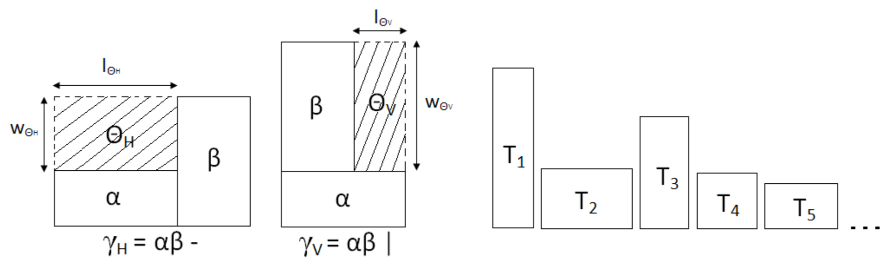


Figure 4: Dominated patterns (γ_H, γ_V) and available pieces

Let α and β be two different patterns. Suppose that γ_H and γ_V are the feasible patterns obtained from the horizontal and vertical combination of α and β. Let Θ_H and Θ_V be the resultant empty space of the horizontal and vertical construction, respectively (see Figure 4). Assume that $x_i(\alpha)$ and $x_i(\beta)$, with $i \in S = \{1, .., n\}$, represent the number of type T_i pieces used in builds α and β, respectively. Then γ_H or γ_V are dominated patterns if:

$$\gamma_H : \ \exists i \in S \ \ x_i(\alpha) + x_i(\beta) < b_i \text{ and } (l_{\Theta_H}, w_{\Theta_H}) >= (l_i, w_i)$$
$$\gamma_V : \ \exists i \in S \ \ x_i(\alpha) + x_i(\beta) < b_i \text{ and } (l_{\Theta_V}, w_{\Theta_V}) >= (l_i, w_i)$$

where $w_{\Theta_H} = |w_\alpha - w_\beta|$ and $l_{\Theta_H} = l_\alpha$ if $w_\alpha < w_\beta$ or $l_{\Theta_H} = l_\beta$ otherwise. In the case of the vertical build, $l_{\Theta_V} = |l_\alpha - l_\beta|$ and $w_{\Theta_V} = w_\alpha$ if $l_\alpha < l_\beta$ or $w_{\Theta_H} = w_\beta$ otherwise.

This domination rule must be checked during the search process, at the step dedicated to the combination of the current best build α with the already explored builds β. If it is detected that the new build γ_H or γ_V is a dominated pattern, it is not inserted in OPEN, thus avoiding a further exploration of a non-promising build. For each pair (α, β), the domination rule must check if there is any available piece which fits inside the generated trim loss. So, the complexity a single checking is $O(n)$, being n the number of pieces.

2.1.2. Type D2: symmetric patterns in opposite directions

Let α and β be patterns obtained only by horizontal combinations of pieces. Such a pattern containing only horizontal concatenations is denoted *H-pattern*. Consider $d = l_\alpha - l_\beta$ and $d \geq 0$, i.e. $l_\alpha \geq l_\beta$. Let α' and α'' be the last subpatterns combined horizontally to obtain α, and let β' and β'' be the last subpatterns combined horizontally to obtain β. Then, the pattern combinations $\gamma_1 = \beta'\beta'' - \alpha'\alpha'' - |$ and $\gamma_2 = \beta'\alpha' \mid \beta''\alpha'' \mid -$ are said to be symmetric (see Figure 5). If $l_{\alpha'} \geq l_{\beta'}$ and $l_{\alpha''} \geq l_{\beta''}$, γ_1 represents a dominated pattern and it can be discarded. As shown in Figure 5, when

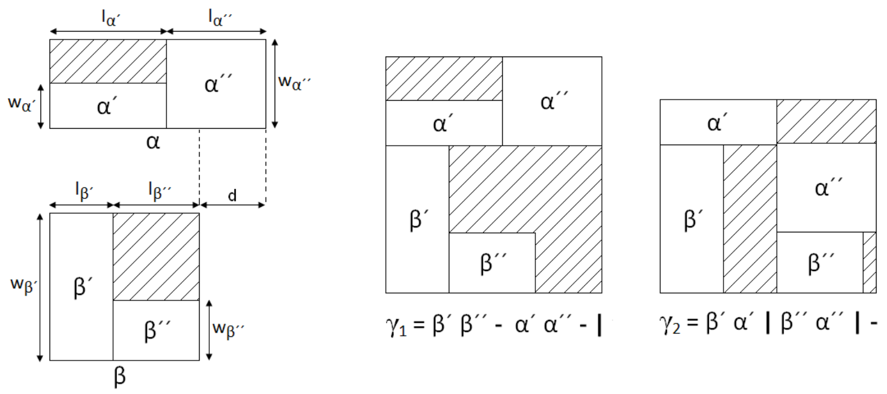


Figure 5: Patterns α and β and constructions γ_1 and γ_2

$l_{\alpha'} \geq l_{\beta'}$ and $l_{\alpha''} \geq l_{\beta''}$, γ_2 dominates γ_1. Both contain the same pieces, so they have the same profit, but γ_2 utilizes the space better. In γ_1 as well as in γ_2, the length is given by the lengths of the subpatterns α' and α'', i.e. $l_{\gamma_1} = l_{\gamma_2} = l_{\alpha'} + l_{\alpha''}$. The widths of γ_1 and γ_2 are different, however:

$$\begin{aligned} w_{\gamma_1} &= \ max\{w_{\beta'}, w_{\beta''}\} \ + \ max\{w_{\alpha'}, w_{\alpha''}\} \\ w_{\gamma_2} &= \ max\{(w_{\beta'} + w_{\alpha'}),\ (w_{\beta''} + w_{\alpha''})\} \end{aligned}$$

The width of γ_1 will always be the maximum possible, while γ_2 is able to obtain a lower width. So, γ_1 and γ_2 are symmetric patterns - they use exactly the same set of pieces - but γ_1 needs a greater width to lay down the pieces. Since the problem objective involves the maximization of the stock sheet usage, γ_1 represents a non-promising solution which is always dominated by its symmetric pattern γ_2.

This type of dominance is checked inside the search loop. It is evaluated every time the current best build α and an element from OPEN, denoted β, are going to be vertically concatenated. First, it is necessary to check whether both builds, α and β, are H-patterns. In the case where they both are H-patterns, then if conditions $l_{\alpha'} \geq l_{\beta'}$ and $l_{\alpha''} \geq l_{\beta''}$ hold, the new vertical build $\gamma = \alpha\beta|$ (which corresponds to the previously denoted γ_1) is not created or inserted in OPEN, because during the search process a pattern representing γ_2, and thus having better properties than γ, will be created. As the checking involves the analysis of two representation of patterns (to check if they have only horizontal or vertical constructions) the complexity is $O(2 * m) = O(m)$, being $m = 2 * n - 1$ the maximum size of a pattern representation (note that patterns are described with the pieces involved and the symbols $|$ or $-$ for denoting vertical/horizontal constructions).

We have developed a more accurate implementation which involves the consideration of all possible divisions of α and β patterns into subpatterns α', α'', β', and β''. Although the consideration of some other possible subdivisions of the patterns increases the associated cost of the rule checking, it makes it possible to detect more dominated nodes. Moreover, during the search not many large H-patterns are explored since the most promising cutting patterns usually involve both horizontal and vertical combinations.

2.1.3. Type D3: symmetric patterns in the same direction

Let α and β be patterns obtained by only horizontal combinations of pieces, i.e. H-patterns (or only vertical combinations of pieces, i.e. V-patterns). The horizontal construction (the vertical case could be made in the same way) between these two patterns, α (taken from CLIST) and β (taken from OPEN), is not created if either of the following cases is satisfied:

1. α is composed of different type of pieces:

$$\exists i, j \in S = \{1, .., n\},\ x_i(\alpha) \geq 1,\ x_j(\alpha) \geq 1,\ i \neq j$$

2. α is composed of only one type of piece, β is composed of different types of pieces, and one of the pieces of β is the piece composing α:

$$\begin{gathered}\exists i \in S = \{1, .., n\},\ x_i(\alpha) \geq 1 \text{ and } \nexists j \in S = \{1, .., n\},\ x_j(\alpha) \geq 1,\ i \neq j \\ \exists k, l \in S = \{1, .., n\},\ x_k(\beta) \geq 1,\ x_l(\beta) \geq 1,\ k \neq l \\ \exists m \in S = \{1, .., n\},\ x_m(\beta) \geq 1,\ m = i\end{gathered}$$

3. α is composed of only one type of piece, β is composed of only one type of piece, this piece is the same for α and β, and the difference between the number of pieces in α and β is greater than 1:

$$\begin{gathered}\exists i \in S = \{1, .., n\},\ x_i(\alpha) \geq 1 \text{ and } \nexists j \in S = \{1, .., n\},\ x_j(\alpha) \geq 1, i \neq j \\ \exists k \in S = \{1, .., n\},\ x_k(\beta) \geq 1 \text{ and } \nexists l \in S = \{1, .., n\},\ x_l(\beta) \geq 1, k \neq l \\ |x_i(\alpha) - x_k(\beta)| > 1,\ i = k\end{gathered}$$

These three duplication pattern cases are checked at each search step, before doing a concatenation of patterns α and β. If both constructions α and β are H-patterns, then if any of the three conditions is satisfied, the new build $\alpha\beta-$ is not generated. If both constructions α and β are V-patterns, then if any of the three conditions is satisfied, $\alpha\beta|$ is not generated. In this case, the checking also has a complexity of $O(2 * m) = O(m)$, being $m = 2 * n - 1$ the maximum size of a pattern representation.

We must ensure that the application of these three rules do not produce the missing of a non-redundant solution. So, we are going to analyze, on one hand the generation of patterns composed of one single type of piece, and on the other hand the generation of patterns composed of different types of pieces. Let T_{n_i} represents a pattern composed of n pieces of type i. If n_i is an even number, then the pattern T_{n_i} can be obtained by the combination of the patterns $T_{n_i/2}$ and $T_{n_i/2}$. If n_i is an odd number, then the pattern T_{n_i} can be obtained by the combination of the patterns $T_{(n_i+1)/2}$ and $T_{(n_i-1)/2}$. In both cases, the difference of pieces between the patterns to be combined is not greater than 1. This way, the generation of any single-type-of-piece pattern is ensured.

Now, let be $T_{l_i}T_{m_j}T_{n_k}$ a pattern composed of l pieces of type i, m pieces of type j, and n pieces of type k, where $l \in [1, b_i]$, $m \in [1, b_j]$, and $n \in [1, b_k]$. We have already ensured the achievement of sub-patterns T_{l_i}, T_{m_j}, and T_{n_k}, because there are composed of a single type of piece. Because of the algorithm operation, all these sub-patterns will be introduced in CLIST following an order established by the estimation function f' (line 3, Algorithm 1). Supposing an order $T_{l_i} > T_{m_j} > T_{n_k}$, i.e. T_{l_i} is the first to enter CLIST, T_{m_j} the second and so on, then the pattern $T_{l_i}T_{m_j}T_{n_k}$ can be obtaining as follows:

1. When $\beta = T_{n_k}$, T_{m_j} will be inside CLIST because $T_{m_j} > T_{n_k}$.
2. So, for $\alpha = T_{m_j}$ and $\beta = T_{n_k}$, the composition $T_{m_j}T_{n_k}$ is obtained because α doesn't include different type of pieces and there is no pieces of type m inside β.
3. The search process will continue until $\beta = T_{m_j}T_{n_k}$.
4. At that moment, the combination with $\alpha = T_{l_i}$ will be done, because α is already in CLIST since $T_{l_i} > T_{m_j} > T_{n_k}$ and because α doesn't include different type of pieces and there is no pieces of type i inside β.

For patterns containing more subsets of different type of pieces, the process can be repeated introducing the necessary extra steps. Note that the order in which the pieces may appear is not important for this analysis because we are focused on constructions of only horizontal (vertical, respectively) combinations, so that, the order will not affect to the final obtained composition.

2.2. Post-generation Rules

Once new patterns have been generated, they are held in one of the two available data structures: OPEN or CLIST. Elements in OPEN represent generated builds that are waiting to be explored. OPEN is implemented as an array of pointers to linked lists of subproblems [27]. Subproblems with the same upper bound f' go to the same linked list, following a FIFO structure. The elements in CLIST, on the other hand, have already been explored, i.e. they have been extracted from OPEN and combined with the elements in CLIST. In order to simplify the detection of patterns exceeding the cutting stock dimensions, in the modified version of the algorithm, CLIST is implemented as a bi-dimensional data structure which groups its elements by length and width sizes [12, 27]. Elements in the same list are ordered following a FIFO structure.

2.2.1. Type D4: duplicated/dominated patterns in OPEN

Before inserting a pattern γ in OPEN, a check is done of whether OPEN already contains a pattern γ' with the same set of pieces as γ and lower or equal dimensions:

$$\forall i \in S = \{1,..,n\},\ x_i(\gamma') = x_i(\gamma) \text{ and } l_{\gamma'} \leq l_\gamma,\ w_{\gamma'} \leq w_\gamma$$

In this case, γ is dominated (or duplicated if the dimensions match, $l_{\gamma'} = l_\gamma$ and $w_{\gamma'} = w_\gamma$), so it is not inserted in the list. γ is only compared with the elements in OPEN with the same f' value, i.e. a check with the whole list would involve excessive computational effort. On the contrary, if γ dominates one or some of the elements in OPEN, they must be removed from the list. Avoiding the insertion of a duplicated/dominated element in OPEN may avoid a future exploration of a non-promising cutting pattern. The complexity of the checking is $O(p * n)$, being p the number of elements in the OPEN sublist and n the number of pieces.

2.2.2. Type D5: duplicated/dominated patterns in CLIST

Every time a pattern α is going to be inserted in CLIST, a check is done of whether there is already a pattern α' in CLIST containing the same set of pieces or more. Thanks to the CLIST structure, which stores elements by length and width dimensions, α is only compared with the elements in CLIST having the same length and width dimensions:

$$\forall i \in S = \{1,..,n\},\ x_i(\alpha') \geq x_i(\alpha) \text{ and } l_{\alpha'} = l_\alpha,\ w_{\alpha'} = w_\alpha$$

So, if an element α' with the same dimensions as α uses the same pieces, α is duplicated. If α' uses the same set of pieces as α and some other ones, then α is dominated by α'. In such cases, α is not inserted in the list. On the contrary, if α uses the same pieces and more than some of the elements in CLIST, which share dimensions with α, they are dominated by α and so they are removed from the list. Avoiding the insertion of an element in CLIST may save up to two combinations (horizontal and vertical) for each of the subsequently explored cutting patterns. The complexity of the checking is also $O(p*n)$, being p in this case the number of elements in the CLIST sublist and n the number of pieces.

Note that when the pre-generation rules are not activated, rule D5 is able to detect and eliminate the patterns avoided by rules D1 and D3. Since the pairs of non-dominated and dominated patterns, and non-duplicated and duplicated patterns analyzed in rules D1 and D3, respectively, have identical dimensions, they will be soon or later introduced in the same position of the CLIST structure, so that, they can be discarded by rule D5.

Note that these two post-generation rules are only applied for subsets of elements in OPEN and CLIST, respectively. One can think that it may be better to apply the comparison of the given pattern to all the elements in OPEN or CLIST, respectively. However, we noted that in such cases the rule checkings didn't compensated for the improvement obtained in the reduction of nodes. For this reason, we finally decided to design rules that could apply the comparisons in a more restricted group of patterns.

3. Parallel Algorithms

We have proposed a set of dominance/duplication rules which makes possible to highly reduce the problem search space, thus reducing the computational time required for the sequential search algorithm. However, the intrinsic complexity of VB algorithm still makes solving large problem instances computationally difficult. For this reason, the possibilities of parallelizing the algorithm have been carefully studied. An analysis of Algorithm 1 shows its intrinsically sequential nature. Each search step requires computing the most promising non-explored meta-rectangle. Such a computation consists of horizontal and vertical combinations of the selected meta-rectangle with *those already explored*. All these combinations are necessary to ensure the best exact solution is obtained. If some of the meta-rectangle combinations are missed, the best solution could be missed. Taking into account this kind of dependency between the problem tasks, two parallel approaches have been carefully designed, thus ensuring the achievement of the optimal solution. The first approach is a fine-grained approximation and involves the parallelization of the build-generation step. The coarse-grained alternative is based on the parallelization of the whole search loop. The study of these parallel schemes shows that any attempt to parallelize the VB algorithm must consider its highly irregular computational structure.

3.1. Fine-Grained Scheme

This implementation is based on the introduction of a parallel generation of meta-rectangles from a certain best meta-rectangle. The general operation of this scheme follows the same structure as that of the sequential scheme presented in Algorithm 1. The main difference appears in the build generation loop. Each processor involved works on a section of the bi-dimensional CLIST data structure, combining the current best meta-rectangle with a subset of the elements in CLIST. Each processor keeps a replicated copy of CLIST. Meanwhile, OPEN is distributed and only contains the builds locally generated by its own processor. This management of the structures allows the processors to work independently in the generation of new meta-rectangles.

However, after each combination of the current build with the previous best ones, each processor must identify its own local best build, i.e. the local meta-rectangle with highest upper bound value. To determine which is the global best current meta-rectangle to be expanded next, each processor communicate its local best upper build and from this subset of best builds the best global one is determined and communicated to all the processors. This *all-to-all reduction point* is shown in Figure 6. Once the processors have the new best build, they can start with the generation work. The same reduction point is used to update the global best solution value, so that non-promising cutting patterns can be properly discarded.

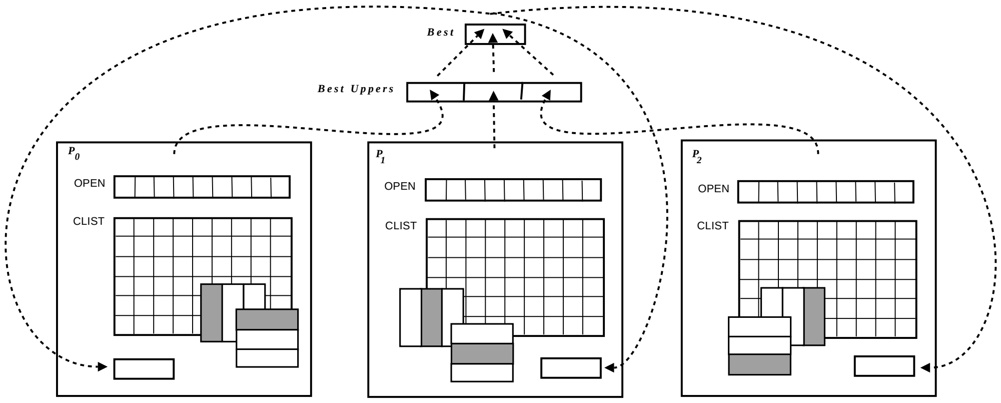


Figure 6: All-to-all reduction point

Algorithm 2 Fine-Grained Parallel Algorithm

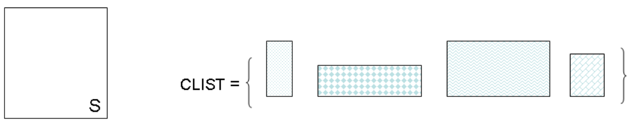
```
if (k == MASTER) then
  OPEN$_k$ = $\{T_1, T_2, \ldots, T_n\}$;
  sharedBestUpper[k] = StaticMap(element in OPEN$_k$ with highest f');   sharedBestUpId = k;
else
  OPEN$_k$ = $\emptyset$;
end if
CLIST = $\emptyset$;   BestSol = LowerBound();   BestSolValue: = $g(BestSol)$;
while ($\exists i/$ OPEN$_i \neq \emptyset$) do
  if (k == sharedBestUpId) then
    Remove $\alpha$ meta-rectangle from OPEN$_k$ with highest f';
  else
    $\alpha$ = DynamicMap(sharedBestUpper[sharedBestUpId]);
  end if
  Insert $\alpha$ in CLIST;
  # pragma omp parallel for
  for (all $\beta$ in CLIST / $x_i(\alpha) + x_i(\beta) \leq b_i \forall i,\ l_\beta + l_\alpha \leq L$) do
    $\Gamma_H = \alpha\beta-$;    /* horizontal build */
    $l_{\Gamma_H} = l_\alpha + l_\beta$;  $w_{\Gamma_H} = \max(w_\alpha, w_\beta)$;  $g(\Gamma_H) = g(\alpha) + g(\beta)$;
    $x_i(\Gamma_H) = x_i(\alpha) + x_i(\beta)\ \forall i \in [1, n]$;
    if ($g(\Gamma_H)$ >BestSolValue) then
      Clear OPEN$_k$ from BestSolValue to $g(\Gamma_H)$;
      BestSolValue = $g(\Gamma_H)$;   BestSol = $\Gamma_H$;
    end if
    if ($f'(\Gamma_H)$ >BestSolValue) then
      Insert $\Gamma_H$ in OPEN$_k$ at entry $f'(\Gamma_H)$;
    end if
  end for
  # pragma omp parallel for
  for (all $\beta$ in CLIST / $x_i(\alpha) + x_i(\beta) \leq b_i \forall i,\ w_\beta + w_\alpha \leq W$) do
    $\Gamma_V = \alpha\beta|$;    /* vertical build */
    $l_{\Gamma_V} = \max(l_\alpha, l_\beta)$;  $w_{\Gamma_V} = w_\alpha + w_\beta$;  $g(\Gamma_V) = g(\alpha) + g(\beta)$;
    $x_i(\Gamma_V) = x_i(\alpha) + x_i(\beta)\ \forall i \in [1, n]$;
    ...
  end for
  sharedBestUpper[k] = StaticMap(element in OPEN$_k$ with highest f');
  sharedBestSolValue[k] = BestSolValue;
  # pragma omp flush (sharedBestUpper, sharedBestSolValue)
  if (k == MASTER) then
    sharedBestUpId = $i/\ \nexists j$ sharedBestUpper[j] > sharedBestUpper[i];
    sharedBestSolId = $i/\ \nexists j$ sharedBestSolValue[j] > sharedBestSolValue[i];
  end if
  # pragma omp flush (sharedBestUpId, sharedBestSolId)
  if (sharedBestSolValue[sharedBestSolId] != BestSolValue) then
    Clear OPEN$_k$ from BestSolValue to sharedBestSolValue[sharedBestSolId];
    BestSolValue = sharedBestSolValue[sharedBestSolId];   BestSol = NULL;
  end if
end while
if (BestSol != NULL) then
  Return BestSol;
end if
```

The parallel algorithm is based mainly on the parallelization of two `for` loops together with a reduction point for the update of two variables: next best meta-rectangle and current best solution. Because of the simplicity of OpenMP and the ease of working with loops in parallel, the OpenMP API was initially used to implement the fine-grained approach [27] (Algorithm 2 shows the OpenMP pseudocode for thread k). The usage of OpenMP allowed for an easy customization of the work distribution. The combinations through CLIST elements are shared out based on the OpenMP *schedule clause*. Different static and dynamic scheduling options have been checked, although no important differences have been noticed among them. The main obstacle to implementation lies in using dynamic structures as linked lists (OPEN and CLIST). In attempting to devise a more immediate approximation where these dynamic lists are shared, and thus modified by all the threads, we found that there is no mechanism available to ensure the integrity of the dynamic data. In OpenMP there is a pragma to ensure the integrity of a static variable, *#pragma flush (name_of_static_variable)*, but there is no way of doing a flush over a variable allocated in the heap. This requires the introduction of some additional operations in order to share dynamic data.

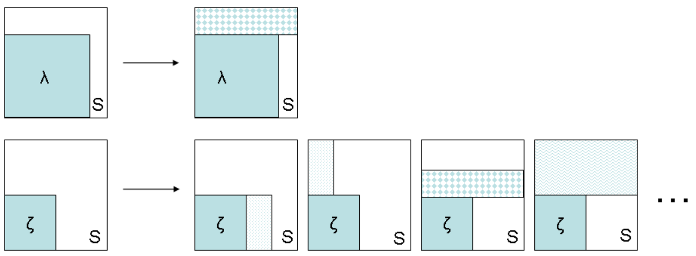
This parallel algorithm can be easily implemented in a distributed memory scheme. In this case, the processors would exchange these variables through MPI functions instead of using static shared data structures. More specifically, lines 35-36 of Algorithm 2 are replaced by a packing of the best meta-rectangle and the best solution value into a communication buffer. For the communication of the information, line 37 is replaced by an *MPI_Allgatherv* operation. Then, lines 38-46 are replaced by the following: each processor unpacks the information gathered, updates the new best solution value and also the best current meta-rectangle to be analyzed. In this implementation, we notice that any dynamic distribution of loop iterations, is more complex than the ones achieved by the OpenMP schedule clause. The user must explicitly manage all shared variables, synchronization points and locks, thus decreasing the simplicity - and possible the efficiency - of the approach. Moreover, the OpenMP implementation demonstrated that dynamic distributions of such fine-grained tasks didn't allow for better results. That is why both implementations will be compared and tested with a static and equitable distribution of the loop iterations.

3.2. Coarse-Grained Scheme

The coarse-grained approach consists of the parallel execution of the main search loop together with a flexible synchronization scheme and a load balancing strategy [29]. Algorithm 3 shows the code for processor k. The parallel scheme replicates CLIST and distributes OPEN among the available processors, p. Each processor independently follows the same steps as in the sequential algorithm: selection of the best meta-rectangle of the local OPEN for its combination with the elements in CLIST. Communication among the



(a) Available surface S and CLIST structure



(b) Example of irregular generation of builds

Figure 7: Irregular generation of meta-rectangles

processors is held at least every *tPeriod* seconds or *iPeriod* search iterations, *tPeriod* and *iPeriod* being configuration parameters specified by the user. This communication point is strictly required to generate the complete set of feasible solutions. At this point, processes perform an all-to-all communication (line 5) to exchange information about: the best solution value, the number of elements in OPEN and the sets of builds that have been analyzed since the last communication step. With the information of the best local solutions, the current best global solution can be updated, thus allowing processors to avoid - during the next search steps - the exploration of unnecessary areas of the search space. The number of elements in each OPEN structure is also exchanged in order to have an approximate idea of each processor's associated work load, which is necessary for the load balancing strategy applied (line 14).

The resulting performance depends on how the communication steps are introduced into the algorithm. If the communication step is done only when all processors have no pending work, the parallel algorithm obtains no speedup at all. Processors would spend most of their time doing unnecessary combinations and waiting for other processors to exchange information. So, the communication among processors must be frequently repeated during the search process. The simplest idea is to introduce these communication points after a certain number of search steps, *iPeriod*. The inconvenience of this approach is that the work load associated with each search iteration may differ considerably. Figure 7 shows the highly irregular work load associated with different meta-rectangles. Meta-rectangle λ only generates two new constructions, while ζ generates at least one new build when combined with every T_i item. If communications are done every certain amount of search iterations, many of the processes will spend most of their time idle while a few of them are working intensely. The other proposed alternative introduces periodically communications within a fixed time interval, $tPeriod$. The simplest idea for implementing such a proposal involves checking a timer at every search loop iteration.

The drawback to this last approach is that, as mentioned before, the work load varies considerably from iteration to iteration, so that the timer checkings will not equally occur among processors. However, increasing the time period so that several search iterations can be done before the $tPeriod$ is reached, helps to compensate the load within iterations. Another alternative time-based synchronization scheme which is independent of the structure of the search iterations was tested [29, 30], although its complexity did not compensate for the gain in synchronization effectiveness. For this reason, here we propose the usage of both criteria together - the time and iteration intervals - so as to make it possible to better balance the work executed by every processor between synchronization points.

Considering that many communication steps are performed in the parallel algorithm, a dynamic and parametric load balancing scheme was designed in order to fairly balance the work load among processors. The size of the OPEN structures is not an exact indicator of the computational effort required to solve the pending work. Nevertheless, obtaining a good distribution of the pending tasks among processes yields good performance, because each process has enough pending work to be busy until the next communication step. The resulting load balancing - sharing - scheme aims to have no idle processes by redistributing the elements of the OPEN structures when any process has less work than a certain threshold. The redistribution of work usually happens before any process becomes idle, i.e. the load sharing scheme predicts that some of the processes are going to be idle soon and performs the redistribution.

Since the parallel algorithm is based on a completely distributed scheme, where a single communication point is required every certain interval of time or iterations, the initial approach was implemented in MPI. However, it is beneficial to compare the initial MPI proposal with an approach based on OpenMP. For an OpenMP implementation, we know that the communication

Algorithm 3 Coarse-Grained Parallel Algorithm

```
1: OPEN$_k$ = $\{T_{k+j*p}$ / $k+j*p<n\}$;    CLIST = $\emptyset$;
2: BestSol = LowerBound();    BestSolValue = $g(BestSol)$;
3: while ($\exists i$/ OPEN$_i \neq \emptyset$) do
4:   if (OPEN$_k$ == $\emptyset$) or (time $\geq$ tPeriod) or (iters == iPeriod) then
5:     $(\lambda, C, R)$ = AllToAll (BestSolValue, sizeof(OPEN$_k$), PC);
6:     if ($max(\lambda)$ > BestSolValue) then
7:       Clear OPEN$_k$ from BestSolValue to $max(\lambda)$;
8:       BestSolValue = $max(\lambda)$;    BestSol = NULL;
9:     end if
10:    CLIST = CLIST $\cup$ (R - PC);    PC = $\emptyset$;
11:    $\Pi = \{\pi_0, \ldots, \pi_{p-1}\}$ partition of $\{S \otimes T$ / $S, T \in R$; $S \neq T\}$;
12:    Compute vertical and horizontal combinations of $\pi_k$;
13:    if (balanceRequired(C, minBalThresh, maxBalThresh)) then
14:      loadBalance(C, MaxBalanceLength);
15:    end if
16:    iters = time = 0;
17:  else
18:    Choose $\alpha$ meta-rectangle from OPEN$_k$ with highest f';
19:    Insert $\alpha$ in CLIST and in PC;
20:    for (all $\beta$ in CLIST / $x_i(\alpha) + x_i(\beta) \leq b_i \forall i$, $l_\beta + l_\alpha \leq L$) do
21:      $\Gamma_H = \alpha\beta-$;    /* horizontal build */
22:      $l_{\Gamma_H} = l_\alpha + l_\beta$;  $w_{\Gamma_H} = \max(w_\alpha, w_\beta)$;  $g(\Gamma_H) = g(\alpha) + g(\beta)$;
23:      $x_i(\Gamma_H) = x_i(\alpha) + x_i(\beta)\ \forall i \in [1, n]$;
24:      if ($g(\Gamma_H)$ >BestSolValue) then
25:        Clear OPEN$_k$ from BestSolValue to $g(\Gamma_H)$;
26:        BestSolValue = $g(\Gamma_H)$;    BestSol = $\Gamma_H$;
27:      end if
28:      if ($f'(\Gamma_H)$ >BestSolValue) then
29:        Insert $\Gamma_H$ in OPEN$_k$ at entry $f'(\Gamma_H)$;
30:      end if
31:    end for
32:    for (all $\beta$ in CLIST / $x_i(\alpha) + x_i(\beta) \leq b_i \forall i$, $w_\beta + w_\alpha \leq W$) do
33:      $\Gamma_V = \alpha\beta|$;    /* vertical build */
34:      $l_{\Gamma_V} = \max(l_\alpha, l_\beta)$;  $w_{\Gamma_V} = w_\alpha + w_\beta$;  $g(\Gamma_V) = g(\alpha) + g(\beta)$;
35:      $x_i(\Gamma_V) = x_i(\alpha) + x_i(\beta)\ \forall i \in [1, n]$;
36:      ...
37:    end for
38:    iters = iters + 1;
39:  end if
40: end while
41: if (BestSol != NULL) then
42:   Return BestSol;
43: end if
```

of the necessary synchronization data must be performed through the usage of static shared data structures. So, the general structure of Algorithm 3 is maintained, but the communication of data is done through shared memory. For the all-to-all communication (line 5), the data packing, sending-receiving, and unpacking is replaced by a write on a shared memory structure of the best solution value, the OPEN list size, and the set of best analyzed builds. In order to share the set of analyzed builds, they must be mapped onto a static structure.

Problem	Search Time			
	No Dom.	Pre-gen.	Post-gen.	All Dom.
ATP33s	2522.22	66.44	3.45	3.46
ATP36s	80.18	12.93	1.23	1.22
ATP37s	341.85	5.45	2.67	2.59
ATP39s	17.28	4.3	2.78	2.72
CL_07_25.72	140.25	20.70	6.79	5.71
CL_07_25.72	996.27	87.01	23.94	16.41
CL_07_50.89	184.55	8.24	4.22	2.78
CL_07_101.37	115.26	12.99	6.21	4.58
CW6	132.83	24.10	5.91	5.69
Hchl2	1752.04	329.92	143.24	100.86
Hchl5s	180.31	22.62	5.87	5.52
Hchl5s_	193.76	24.29	8.97	7.93

Table 1: Effect of the dominance and duplication rules

4. Computational Results

In this section we analyze the impact of the proposed dominance/duplication rules and the validity of the parallel implementations when combined with this type of sequential optimizations which involve an unpredictable variation of the search space. The computational study was performed on some instances which are available at [32, 33] and which have been widely used in many related studies [13, 17, 29, 34]. Tests were run on an SMP NovaScale 6320 node, which supports up to 32 Intel® Itanium 2 processors at 1.5GHz. The compilers used were: *gcc* 3.4.6, *icc* 9.1, and *mpicc* for MPI Bull 1.6.5. For every experiment, ten executions were repeated and average values considered. All the computational times are shown in seconds.

In previous works [27, 29], we have demonstrated the validity of our sequential approach when compared to the original ones, so here we will focus on how the detection and deletion of dominated/duplicated bounds of the search tree affects to our sequential and parallel proposals. Table 1 presents the results obtained when all dominance/duplication rules are incorporated in our version of MVB algorithm. The table shows the total execution time invested in the search of the solution when: no dominance or duplication rules are applied, only pre-generation rules are applied, only post-generation rules are applied, and both types of rules are applied. The results demonstrate that both dominance rules allow for a reduction of the initial search space, although the post-generation checks seem to have a greater effect in the global search process. Post-generation rules are less frequently checked - only on insertion or computational stages - and they are not too expensive computationally. Besides, they are able to discard a considerable quantity

Proc.	No Dominances: Fine OpenMP	No Dominances: Coarse MPI	All Dominances: Fine OpenMP	All Dominances: Coarse MPI
ATP33s				
1	4190.82	4420.71	19.3	19.53
2	3361.66	2617.68	9.29	10.82
4	2141.4	1410.84	5.43	5.88
8	1919.88	789.67	7.04	3.4
16	1678.36	394.66	5.61	1.99
Hchl2				
1	2733.7	3256.23	198.86	167.56
2	2463.14	2004.98	164.37	103.54
4	1867.8	1273.17	141.08	54.44
8	1610.41	1050.85	169.35	33.25
16	1622.06	767.03	149.59	18.2
CW6				
1	782.65	907.19	42.25	44.36
2	238.53	476.89	30.18	26.46
4	66.3	239.18	25.2	15.33
8	32.75	131.8	45.32	9.76
16	22.89	73.23	45.69	6.0
CL_07_50_09				
1	264.48	249.37	3.82	4.16
2	170.99	137.05	3.28	2.58
4	123.94	79.17	2.79	1.31
8	138.9	45.99	2.99	0.87
16	110.11	23.68	2.84	0.58

Table 2: Dominances in fine-grained and coarse-grained parallel algorithms

of nodes, even including some of the detected by some pre-generation rules. The checking of the pre-generation rules is done more frequently but even so, they are not able to discard as many nodes as in the post-generation case.

Table 2 shows the execution times obtained when both, the fine-grained (see Section 3.1) and the coarse-grained (see Section 3.2), parallel algorithms are executed with 1, 2, 4, 8, and 16 processors. In this initial experiment, we used the original implementation of the algorithms, i.e. the OpenMP implementation for the fine-grained and MPI for the coarse-grained. For the coarse-grained executions the configuration parameter $tPeriod$ is fixed to 0.1 seconds and $iPeriod = 50$. The same parameter configuration was used for every test problem, i.e. the algorithm was not configured on a per problem basis. Since the detection of dominances and duplicated nodes seriously affects the structure of the search space, two versions of the parallel algorithms were checked: one without considering the detection of dominances and another including all the dominance tests. Note that when introducing dominance checking, the execution times for most problems are drastically reduced. Even so, in cases where execution times are not too short, parallel algorithms are still able to obtain an acceptable speedup. Comparing the two parallel approaches, we realize that the coarse-grained one is able to better scale when the number of processors is increased and even when the executions are not very long. However, the fine-grained shows some difficulties when the number of processors increases and when the executions are too short. This fact is closely related to the grain associated with each parallel approach and to the work load distribution obtained.

Figure 8 shows the distribution of generated nodes for the OpenMP fine-grained approach, i.e. created builds which are obtained from vertical or horizontal combinations, and the balancing of computed nodes for the MPI coarse-grained, i.e. cutting patterns combined with all previous explored nodes. Results are shown for two different problem instances - **Hchl2** and **CW6** - and for executions with 1, 2, 4, 8, and 16 processors. The load balancing scheme introduced in the coarse-grained parallelization allows for a fair distribution of work among processors (note that in the right-hand-side graphics the computed builds are almost homogeneous for the two processors involved in the 2-processor execution, for the four processors involved in the 4-processor execution, and so on. However, the distribution of load in the fine-grained approach depends on the features of the cutting patterns, which can be considerably different from iteration to iteration. A fair distribution of work load is more difficult to obtain when more threads collaborate in the search because there is not enough work to distribute and the subproblems are not uniformly spread throughout CLIST.

We studied the coarse-grained algorithm in depth, given that it has shown a more promising behavior. For the algorithm, we developed two different implementations, an original one based on MPI and another implementation using OpenMP. These two implementations of the same algorithm were compared, considering also the main configuration parameters involved in the algorithm: the synchronization time and iteration periods, *tPeriod* and *iPeriod*. The behavior of each of the approaches depend on the synchronization parameters and also on the given problem instance. That is, the best synchronization configuration for the OpenMP approach depends on the given problem, and generally it is different from the best configuration of the MPI implementation for that particular problem. Moreover, as mentioned in section 3.2, the optimal behavior of the algorithm is achieved when both synchronization parameters, time and iterations, are combined. A wide tuning of the combination of these parameters has been performed in an initial analysis. Some subsets of the best combinations of parameters are shown in

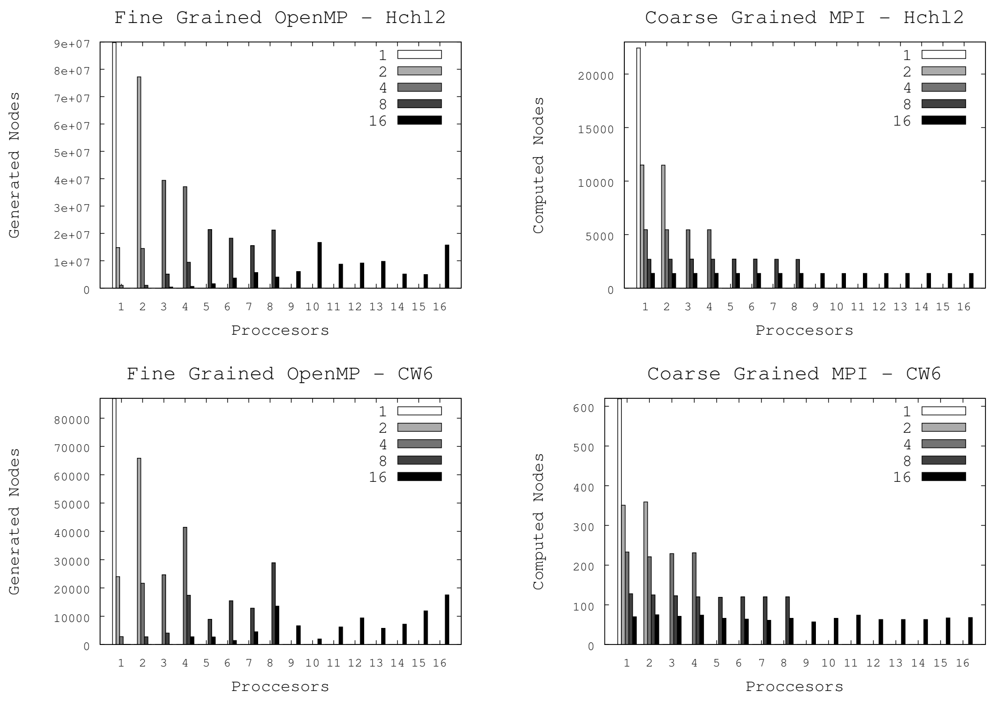


Figure 8: Load balancing: OpenMP fine-grained vs. MPI coarse-grained

Table 3 and Table 4, for the OpenMP and the MPI implementations, respectively. For each execution, both, the total time invested in synchronizations and the total search time, are shown. Note that the best configuration of the iteration interval is the same for both proposals, but with regard to the time synchronization interval, OpenMP behaves better with lower intervals than MPI. In general, OpenMP executions involve a lower search time, thus improving the performance obtained by the MPI implementation. Only when the problem is simple and the number of threads increases, is the OpenMP approach not able to improve on the MPI results. Initially we thought that this may be due to the overhead introduced by the synchronizations, but considering the synchronization times shown in the "*Syn.*" columns of the tables, we realize that the average time that each process spends on synchronizations is very similar for both approaches. However, on further analyzing the algorithms, we notice that in the OpenMP approach, all the processes usually arrive at the synchronization points at the same time, meaning the deviation among synchronization times is almost insignificant when compared to those in the MPI implementation.

Iter.	**Time**	**Processors**							
		1		2		4		8	
		Syn.	Search	Syn.	Search	Syn.	Search	Syn.	Search
				ATP33s					
10	0.3	0.000	19.264	1.246	10.097	0.994	5.240	0.990	3.526
	0.6	0.000	19.092	1.315	10.229	1.012	5.298	1.765	4.363
15	0.3	0.000	19.028	0.292	9.346	0.244	4.666	0.320	2.767
	0.6	0.000	19.081	1.115	9.968	1.129	5.514	1.800	4.573
				Hchl2					
10	0.3	0.006	183.719	17.329	101.984	11.225	47.638	8.530	30.890
	0.6	0.005	184.880	17.114	97.595	11.437	48.402	8.032	30.397
15	0.3	0.004	184.865	15.616	93.245	10.762	47.951	7.638	29.179
	0.6	0.004	183.119	15.479	92.043	11.581	49.041	8.233	30.273
				CW6					
10	0.3	0.000	43.183	0.723	22.953	0.713	13.088	0.929	8.774
	0.6	0.000	43.184	1.309	23.690	1.243	14.245	1.387	9.806
15	0.3	0.000	43.133	0.848	23.429	0.710	12.697	0.797	8.235
	0.6	0.000	43.213	0.471	22.912	0.454	13.483	1.177	9.432
				CL_07_50_09					
10	0.3	0.000	4.037	0.321	2.084	0.189	1.103	0.182	0.769
	0.6	0.000	4.030	0.312	2.065	0.189	1.093	0.146	0.749
15	0.3	0.000	4.024	0.247	2.004	0.209	1.200	0.149	0.768
	0.6	0.000	4.012	0.189	1.913	0.168	1.163	0.198	0.828

Table 3: Coarse-grained algorithm: OpenMP implementation

Iter.	**Time**	**Processors**							
		1		2		4		8	
		Syn.	Search	Syn.	Search	Syn.	Search	Syn.	Search
				ATP33s					
10	1.5	0.005	19.048	0.670	10.766	0.729	6.010	0.681	3.625
	2.0	0.005	18.953	0.680	10.859	0.730	6.007	0.652	3.595
15	1.5	0.004	18.929	0.740	10.638	0.827	6.247	0.573	3.338
	2.0	0.004	18.956	0.769	10.732	0.858	6.322	0.556	3.307
				Hchl2					
10	1.5	0.145	162.108	12.108	111.075	12.607	63.422	9.322	35.926
	2.0	0.146	162.011	12.086	111.326	12.837	64.514	9.324	35.872
15	1.5	0.116	166.109	12.870	115.838	11.851	63.919	7.773	33.496
	2.0	0.117	162.728	12.815	115.474	11.620	63.400	7.805	33.572
				CW6					
10	1.5	0.005	43.163	2.841	26.811	3.196	17.856	3.541	11.909
	2.0	0.005	42.880	2.852	26.897	3.193	17.813	3.533	11.886
15	1.5	0.004	43.143	2.053	26.642	2.304	18.261	2.357	11.028
	1.0	0.004	42.918	2.064	26.852	2.308	18.280	2.596	11.347
				CL_07_50_09					
10	1.5	0.015	4.068	0.129	2.101	0.136	1.198	0.102	0.650
	2.0	0.014	4.086	0.130	2.095	0.151	1.290	0.103	0.650
15	1.5	0.013	4.034	0.121	2.096	0.110	1.270	0.092	0.672
	2.0	0.013	4.024	0.123	2.102	0.109	1.270	0.093	0.674

Table 4: Coarse-grained algorithm: MPI implementation

5. Conclusions

This work presents a set of dominance/duplication rules that makes it possible to significantly reduce the solution search space, thus allowing for an improvement in the efficiency of the best-first algorithm for the 2D Cutting Problem. Dominance and duplication rules are based on the internal properties of the problem solutions, i.e. the combination of pieces to generate the cutting patterns. Some rules can be checked previously to the pattern generation process and other are checked after the creation of the builds. Although they are less frequently applied, the post-generation rules have shown a higher effect on the reduction of the total search time. They are not too expensive computationally and they are able to detect redundant builds that are also detected by some pre-generation checkings.

In order to handle even larger instances, two parallel schemes - based on MVB algorithm - were also presented. The first approach is based on a fine-grained model which executes the generation loops in parallel. The other parallel scheme relies on a coarse-grained model. which is based on the parallel execution of the search loop and on the introduction of efficient synchronization and load-balancing schemes. For the coarse-grained algorithm, different synchronization schemes were proposed: based on search iterations, time intervals, or on a combination of both.

The results obtained with the fine-grained approach are not good enough due to the highly irregular distribution of the work load. The real work load depends on the number of builds stored in each CLIST matrix position, and usually the builds are not uniformly spread throughout the structure. However, in spite of the highly irregular computational structure and the difficulties in breaking the sequential nature of best-first search approaches, the combination of a bulk synchronous methodology with the use of a load-balancing strategy resulted in a fair work load balance and a linear speedup for the coarse-grained algorithm.

On the other hand, the comparison between OpenMP and MPI implementations shows that OpenMP performs better when the synchronizations are done very frequently and involve the communication of a reduced amount of data. MPI implementations showed better performance when the quantity of data to communicate compensates for the network latency. Moreover, MPI results demonstrate that it is able to better scale when compared to the OpenMP implementations. We have demonstrated that, even for algorithms with inherent sequential structure, it is possible to design suitable parallel schemes capable of improving the efficiency of the approach. Furthermore, it is important to note that the efficiency of the parallel schemes mainly depend on the work grain and load distribution, although the selection of a suitable parallel programming tool may be also decisive.

6. Acknowledgements

This work was funded by the EC (FEDER) and the Spanish Ministry of Science and Technology as part of the 'Plan Nacional de I+D+i' (TIN2008-06491-C04-02). The Canary Government has also funded this work through the PI2007/015 research project. The work of Jesica de Armas was funded by grant FPU-AP2007-02414.

References

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

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

[3] G. Wäscher, H. Haußner, H. Schumann, An improved typology of cutting and packing problems, European Journal of Operational Research 183 (3) (2007) 1109–1130.

[4] N. Christofides, C. Whitlock, An Algorithm for Two-Dimensional Cutting Problems, Operations Research 25 (1) (1977) 30–44.

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

[6] V. Zissimopoulos, Heuristic methods for solving (un)constrained two-dimensional cutting stock problems, Methods of Operations Research 49 (1985) 345–357.

[7] F. Vasko, A computational improvement to Wang's two-dimensional cutting stock algorithm, Computers and Industrial Engineering 16 (1) (1989) 109–115.

[8] J. Oliveira, J. Ferreira, An improved version of Wang's algorithm for two-dimensional cutting problems, European Journal of Operational Research 44 (1990) 256–266.

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

[10] S. Tschöke, N. Holthöfer, A New Parallel Approach to the Constrained Two-Dimensional Cutting Stock Problem, in: Parallel Algorithms for Irregularly Structured Problems, Springer-Verlag, 1995, pp. 285–300.

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

[12] V. D. Cung, M. Hifi, B. Le-Cun, Constrained Two-Dimensional Cutting Stock Problems: A Best-First Branch-and-Bound Algorithm, Tech. Rep. 97/020, Laboratoire PRiSM, Université de Versailles (1997).

[13] M. H. Didier Fayard, V. Zissimopoulos, An Efficient Approach for Large-Scale Two-dimensional Guillotine Cutting Stock Problems, JORS 49 (1998) 1270–1277.

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

[15] V.-D. Cung, M. Hifi, B. Le-Cun, Constrained Two-Dimensional Cutting Stock Problems: A Best-First Branch-and-Bound Algorithm, ITOR 7 (2000) 185–210.

[16] M. H. Van-Dat Cung, B. Le-Cun, Constrained Two-Dimensional Cutting Stock Problems: The NMVB approach and the Duplicate Test Revisited, Tech. Rep. 2000.127, Université de Paris (2000).

[17] R. Alvarez-Valds, A. Parajn, J. Tamarit, A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems, Computers and Operations Research 29 (7) (2002) 925–947. doi:http://dx.doi.org/10.1016/S0305-0548(00)00095-2.

[18] D. Fayard, V. Zissimopoulos, An approximation algorithm for solving unconstrained two-dimensional knapsack problems, European Journal of Operational Research 84 (3) (1995) 618–632.

[19] M. Hifi, V. Zissimopoulos, Constrained Two-Dimensional Cutting: An Improvement of Christofides and Whitlock's Exact Algorithm, The Journal of the Operational Research Society 48 (3) (1997) 324–331.

[20] Y. Cui, Heuristic and exact algorithms for generating homogenous constrained three-staged cutting patterns, Computers and Operations Research 35 (2008) 212–225.

[21] Y. Cui, An exact algorithm for generating homogeneous two-segment cutting patterns, Engineering Optimization 39 (2007) 365–380.

[22] Y. Cui, An exact algorithm for generating homogenous t-shape cutting patterns, Computers and Operations Research 34 (2007) 1107–1120.

[23] F. Vasko, C. Bartkowski, Using Wang's two-dimensional cutting stock algorithm to optimally solve difficult problems, International Transactions in Operational Research 16 (2009) 829–838.

[24] P. C. Gilmore, R. E. Gomory, The Theory and Computation of Knapsack Functions, Operations Research 14 (1966) 1045–1074.

[25] G. Miranda, C. León, An OpenMP skeleton for the A* heuristic search, in: Springer-Verlag (Ed.), High Performance Computing and Communications, Vol. 3726 of LNCS, Naples, Italy, 2005, pp. 717–722.

[26] G. Miranda, C. León, OpenMP skeletons for tree searches, in: 14th Euromicro Conference on Parallel, Distributed and Network-based Processing, IEEE Computer Society, Montbeliard-Sochaux, France, 2006, pp. 423–430.

[27] L. García, C. León, G. Miranda, C. Rodríguez, A Parallel Algorithm for the Two-Dimensional Cutting Stock Problem, in: European Conference on Parallel Computing (Euro-Par), Vol. 4128 of LNCS, Springer-Verlag, Dresden, Germany, 2006, pp. 821–830.

[28] L. García, C. León, G. Miranda, C. Rodríguez, Two-Dimensional Cutting Stock Problem: shared memory parallelizations, in: 5th International Symposium on Parallel Computing in Electrical Engineering, IEEE Computer Society, Bialystok, Poland, 2006, pp. 438–443.

[29] C. León, G. Miranda, C. Rodríguez, C. Segura, 2D Cutting Stock Problem: a New Parallel Algorithm and Bounds, in: European Conference on Parallel Computing (Euro-Par), Vol. 4641 of LNCS, Springer-Verlag, Rennes, France, 2007, pp. 795–804.

[30] C. León, G. Miranda, C. Rodríguez, C. Segura, A distributed parallel algorithm to solve the 2D cutting stock problem, in: 16th Euromicro Conference on Parallel, Distributed and Netword-Based Processing, IEEE Computer Society, Toulouse, France, 2008, pp. 429–434.

[31] F. Clautiaux, A. Jouglet, A. Moukrim, A New Graph-Theoretical Model for k-Dimensional Guillotine-Cutting Problems, in: Experimental Algorithms, Vol. 5038 of LNCS, Springer Berlin / Heidelberg, 2008, pp. 43–54.

[32] DEIS - Operations Research Group, Library of Instances, http://www.or.deis.unibo.it/research_pages/ORinstances/2CBP.html.

[33] M. Hifi, 2D Cutting Stock Problem Instances, ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/2Dcutting/.

[34] A. Caprara, P. Toth, Lower bounds and algorithms for the 2-dimensional vector packing problem, Tech. Rep. OR/97/3, University of Bologna (1997).