2D Cutting Stock Problem: a New Parallel Algorithm and Bounds

Coromoto León, Gara Miranda, Casiano Rodríguez, and Carlos Segura

Universidad de La Laguna, Dpto. Estadística, I.O. y Computación
Avda. Astrofísico Fco. Sánchez s/n, 38271 La Laguna, Spain
{cleon,gmiranda,casiano,csegura}@ull.es
http://nereida.deioc.ull.es

Abstract. This work introduces a set of important improvements in the resolution of the Two Dimensional Cutting Stock Problem. It presents a new heuristic enhancing existing ones, an original upper bound that lowers the upper bounds in the literature, and a parallel algorithm for distributed memory machines that achieves linear speedup. Many components of the algorithm are generic and can be ported to parallel branch and bound and A* skeletons. Among the new components there is a comprehensive MPI-compatible synchronization service which facilitates the writing of time-based balancing schemes.

1 Introduction

The Constrained Two Dimensional Cutting Stock Problem (2DCSP) targets the cutting of a large rectangle S of dimensions $L \times W$ in a set of smaller rectangles using orthogonal guillotine cuts: any cut must run from one side of the rectangle to the other end and be parallel to one of the other two edges. 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 c_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:

$$g(x) = \sum_{i=1}^{n} c_i x_i \text{ subject to } x_i \leq b_i \text{ and } x_i \in \mathbb{N}$$

Wang [1] 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 1. Her idea was exploited by Viswanathan and Bagchi [2] to propose a brilliant best first search A* algorithm (VB) which uses Gilmore and Gomory [3] dynamic programming solution to build an upper bound. The VB algorithm uses two lists and, at each step, the best build of pieces (or meta-rectangles) is combined with the already found best meta-rectangles to produce horizontal and vertical builds. Hifi in [4] and later Cung et al. in [5] proposed a modified version of VB algorithm. Niklas et al. in [6] proposed a parallel version of Wang's algorithm. Unfortunately, Wang's method does not always

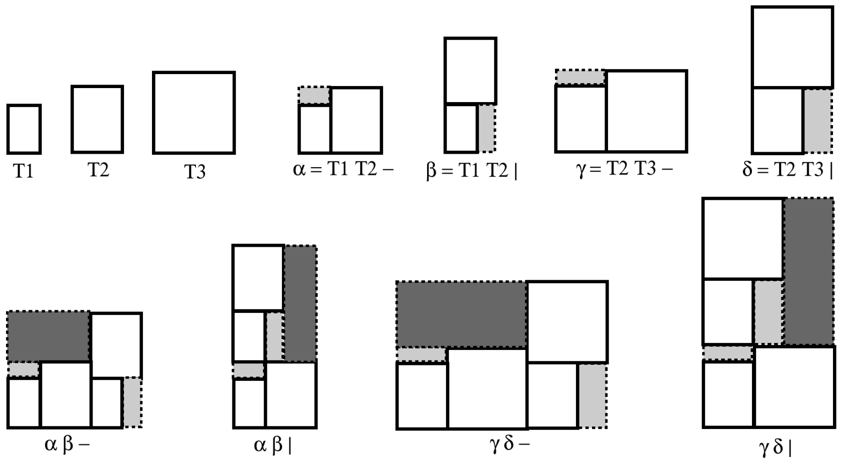

Fig. 1. Examples of Vertical and Horizontal Meta-Rectangles. Postfix notation is used: $\alpha\ \beta\ |$ and $\alpha\ \beta\ -$ denote the vertical and horizontal constructions of rectangles α and β. Shaded areas represent waste.

yield optimal solutions in a single invocation and is slower than VB algorithm. Tschöeke and Holthöfer parallel version [7] starts from the original VB algorithm and uses the Paderborn Parallel Branch and Bound Library, PPBB-LIB [8]. García et al. in [9] presented new data structures and a synchronous parallel algorithm for shared memory architectures. All the approaches to parallelize VB strive against the highly irregular computation structure of the algorithm. Attempts to deal with its intrinsically sequential nature inevitably appears either transformed on an excessively fine granularity or any other source of inefficiency.

This work introduces a set of important improvements when compared with previous work [9]. A new upper bound that lowers the upper bounds in the literature and a new heuristic enhancing existing ones are presented in sections 2.1 and 2.2. Section 3 exposes a parallel algorithm for distributed memory machines achieving linear speedup and good load balancing through the use of a provided timing service. The timing service described in section 4 is portable through platforms and MPI libraries. Section 5 shows the obtained computational results. The conclusions and some lines of future work are given in section 6.

2 Improvements to the Sequential Algorithm

2.1 A New Upper Bound

The new upper bound improves existing upper bounds. It is trivial to prove that is lower than the upper bounds proposed in [2, 4, 5, 7, 9]. The calculus of the new upper bound is made in three steps:

1. During the first step, the following bounded knapsack problem is solved using dynamic programming [5, 7]:

$$V(\alpha) = \begin{cases} \max \sum_{i=1}^{n} c_i x_i & \\ \text{subject to} & \sum_{i=1}^{n} (l_i w_i) x_i \leq \alpha \\ \text{and} & x_i \leq min\{b_i, \lfloor \frac{L}{l_i} \rfloor \times \lfloor \frac{W}{w_i} \rfloor\}, x_i \in \mathbb{N} \end{cases}$$

for all $0 \leq \alpha \leq L \times W$.

2. Then, $F_V(x, y)$ is computed for each rectangle using the equations:

$$\overline{F}(x, y) = \max \begin{cases} \max\{F_V(x, y_1) + F_V(x, y - y_1) & \text{such that } 0 < y_1 \leq \lfloor \frac{y}{2} \rfloor\} \\ \max\{F_V(x_1, y) + F_V(x - x_1, y) & \text{such that } 0 < x_1 \leq \lfloor \frac{x}{2} \rfloor\} \\ \max\{c_i & \text{such that } l_i \leq x \text{ and } w_i \leq y\} \end{cases}$$

where

$$F_V(x, y) = min\{\overline{F}(x, y), V(x \times y)\}$$

3. Finally, substituting the bound of Gilmore and Gomory [3] by F_V in Viswanathan and Bagchi upper bound [2] the new proposed upper bound is obtained:

$$U_V(x, y) = \max \begin{cases} \max\{U_V(x + u, y) + F_V(u, y) & \text{such that } 0 < u \leq L - x\} \\ \max\{U_V(x, y + v) + F_V(x, v) & \text{such that } 0 < v \leq W - y\} \end{cases}$$

2.2 A New Lower Bound

The proposed heuristic mimics Gilmore and Gomory dynamic programming algorithm [3] but substituting unbounded vertical and horizontal combinations by feasible suboptimal ones.

Let be $R = (r_i)_{i=1...n}$ and $S = (s_i)_{i=1...n}$ sets of feasible solutions using $r_i \leq b_i$ and $s_i \leq b_i$ rectangles of type T_i. The cross product $R \otimes S$ of R and S is defined as the set of feasible solutions built from R and S without violating the bounding requirements: i.e. $R \otimes S$ uses $(\min\{r_i + s_i, b_i\})_{i=1...n}$ rectangles of type T_i. The lower bound is given by the value $H(L, W)$ computed by the following equations:

$$H(x, y) = \max \begin{cases} \max\{g(S(x, y_1) \otimes S(x, y - y_1)) & \text{such that } 0 < y_1 \leq \lfloor \frac{y}{2} \rfloor\} \\ \max\{g(S(x_1, y) \otimes S(x - x_1, y)) & \text{such that } 0 < x_1 \leq \lfloor \frac{x}{2} \rfloor\} \\ \max\{c_i & \text{such that } l_i \leq x \text{ and } w_i \leq y\} \end{cases}$$

being $S(x, y) = S(\alpha, \beta) \otimes S(\gamma, \delta)$ one of the cross sets, - either a vertical construction $S(x, y_0) \otimes S(x, y - y_0)$ or a horizontal building $S(x_0, y) \otimes S(x - x_0, y)$ - where the former maximum is achieved, i.e. $H(x, y) = g(S(\alpha, \beta) \otimes S(\gamma, \delta))$.

3 Parallel Algorithm

The general operation mode of the parallel scheme follows the structure of the VB algorithm [2]. This exact algorithm uses two lists, OPEN and CLIST. OPEN stores the generated meta-rectangles. At each step, the most promising meta-rectangle from OPEN is moved to CLIST. The main difference between the sequential and parallel scheme lies in the lists management. The parallel scheme replicates CLIST and distributes OPEN among the available processors, p. Figure 2 shows the code for processor k. Initially, the heuristic and the upper bounds are calculated (lines 2-3). CLIST begins empty and the initial builds, $\mathcal{D}$, are distributed among the processors (lines 4-5). At each search step, the meta-rectangle in OPEN with the highest upper bound is chosen (line 9). This build is inserted into CLIST and into the *Pending Combinations* set, PC (lines 10-11). The PC set holds the analyzed meta-rectangles that have not been transferred to other processors. The selected build must be combined with the already found best meta-rectangles to produce new horizontal and vertical builds (lines 12-23). Only the new elements with expectations of success will be inserted into OPEN (lines 16 and 22). Also, OPEN is cleared if necessary (lines 15 and 21).

In order to generate the complete set of feasible solutions, it is necessary to incorporate a synchronization at certain periods of time. That has been implemented using the *synchronization service* explained in section 4. The synchronization subroutine (lines 28-39) is called when a processor has no pending work (line 7) or when an active alarm of the synchronization service goes off. The expiration time of the alarms is fixed by the user, using the *SyncroTime* parameter (line 38). The information given by each processor consists of: its best solution value, the size of its OPEN list and the set of builds that has analyzed since the last synchronization step (line 29). The elements computed by each processor must be inserted into the CLISTs of the other processors (line 32) and also combined among them. Such combinations are uniformly distributed among processors (lines 34 and 35). The current best solution is updated with the best solution found by any of the processors, pruning nodes in OPEN if necessary (line 31). The stop condition is reached when all the OPEN lists are empty (line 6).

The search path followed by the parallel algorithm can be different from the sequential one. In cases where the initial heuristic finds the exact solution, the number of computed nodes is the same. However, in cases where the heuristic does not find the exact solution, changes in the search path may produce modifications on the number of explored nodes.

To have OPEN lists fairly balanced, a parametric method has been designed. This method requires three configuration parameters: *MinBalThreshold*, *MaxBalThreshold* and *MaxBalanceLength*. The method is executed (lines 36-37) after the computation of the pending combinations. It is necessary to sort the set of processors attending to their OPEN size. Processor in position i is associated with a partner located in position $p - i - 1$. That will match the processor with largest OPEN list with the processor with the smallest one, the second largest one with the second smallest and so on. Partners will make an exchange if the

```
search() {
  BestSol := $H(L,W)$; B: = $g(BestSol)$;
  $h'$ := $U_V$;
  $CLIST$ := $\emptyset$;
  $OPEN_k$ := $\{T_{k+j*p}$ / $k + j * p < n\}$;
  while ($\exists i$ / $OPEN_i \neq \emptyset$) {
    if ($OPEN_k$ == $\emptyset$) { synchronize(); }
    else {
      choose $\alpha$ meta-rectangle from $OPEN_k$ with higher $f' = g + h'$ value;
      insert $\alpha$ in CLIST;
      insert $\alpha$ in PC;
      forall ($\beta \in CLIST$ / $x_\alpha + x_\beta \leq b, l_\beta + l_\alpha \leq L$) do {
        $\gamma = \alpha\beta-$; $l_\gamma = l_\alpha + l_\beta$; $w_\gamma = \max(w_\alpha, w_\beta)$; /* horizontal build */
        $g(\gamma) = g(\alpha) + g(\beta)$; $x_\gamma = x_\alpha + x_\beta$;
        if ($g(\gamma) > B$) { clear $OPEN_k$ from B to $g(\gamma)$; $B = g(\gamma)$; BestSol = $\gamma$; }
        if ($f'(\gamma) > B$) { insert $\gamma$ in $OPEN_k$ at entry $f'(\gamma)$; }
      }
      forall ($\beta \in CLIST$ / $x_\alpha + x_\beta \leq b, w_\beta + w_\alpha \leq W$) do {
        $\gamma = \alpha\beta|$; $l_\gamma = \max(l_\alpha, l_\beta)$; $w_\gamma = w_\alpha + w_\beta$; /* vertical build */
        $g(\gamma) = g(\alpha) + g(\beta)$; $x_\gamma = x_\alpha + x_\beta$;
        if ($g(\gamma) > B$) { clear $OPEN_k$ from B to $g(\gamma)$; $B = g(\gamma)$; BestSol = $\gamma$; }
        if ($f'(\gamma) > B$) { insert $\gamma$ in $OPEN_k$ at entry $f'(\gamma)$; }
      }
    }
  }
  return(BestSol);
}
```

```
synchronize() {
  $(\lambda, C, R)$ = all_to_all (B, $|OPEN_k|$, PC);
  if ($max(\lambda) > B$)
    { clear $OPEN_k$ from B to $max(\lambda)$; $B = max(\lambda)$; }
  $CLIST = CLIST \cup$ (R - PC);
  PC = $\emptyset$;
  Let be $\pi = \{\pi_0, \ldots, \pi_{p-1}\}$ a partition of $\{R_i \otimes R'_j$ / $i \neq j\}$
  compute vertical and horizontal combinations of $\pi_k$
  if (balanceRequired(C, minBalThreshold, maxBalThreshold))
    loadBalance(C, MaxBalanceLength);
  fixAlarm(SyncroTime);
}
```

Fig. 2. Parallel Algorithm Pseudocode for Processor k

one with larger OPEN has more than *MaxBalThreshold* elements and the other has less than *MinBalThreshold*. The number of elements to be exchanged is proportional to the difference of the two OPEN sizes, but it can never be greater than *MaxBalanceLength*.

4 The Synchronization Service

All synchronizations in the model are done through time alarms (*alarm clocks*). That makes the service independent of the particular algorithm and the MPI implementation. Every process participating in the parallel algorithm fixes the alarm to a certain time value. When the alarm goes off, the corresponding process is informed. If the alarm is fixed with the same time value and then an all-to-all exchange is done when the alarm expires, a synchronous scheme is obtained. The service is initiated on each node by starting a daemon. An *alarm clock manager* is created on each node. This process is in charge of attending all the alarm clocks requests coming from the algorithmic processes. For each received request, the service manager creates a new *alarm clock process* that will communicate to the corresponding requester. Once the communication between the requester and its alarm clock is initiated, their interaction proceeds without any intervention of the manager.

Figure 3 shows the state of one computation node running two MPI processes and the synchronization service. The process at the bottom and its corresponding alarm process have already been initiated. The initialization for the process at the top is presented. First of all, each process in the parallel algorithm must ask for the alarm clock service. The manager process attends each alarm service request creating a new alarm clock process and assigning it to the requester. Then, the algorithmic process can activate the alarm clock specifying a certain amount of time. Once the specified time has passed, the alarm clock will notify the process. In this particular case, after each alarm clock notification, the MPI processes can synchronize their information. If a process finishes its work before the alarm time has expired, it can cancel its alarm and go directly to the synchronization point. If each process cancels the alarm, the synchronization will be reached earlier.

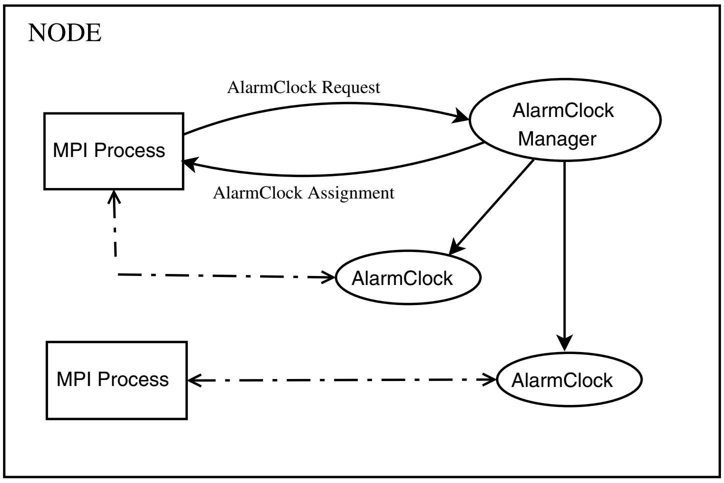

Fig. 3. Synchronization Service Operation

This allows the user to better adapt the alarm service behaviour since the alarm can be activated or cancelled at any moment.

The communication between the algorithmic processes and the alarm manager is done through system message queues. The user activation and cancellation of alarms is done through the message queue that was assigned to it. Alarm expirations are notified by using a variable allocated in the system shared memory. System signals can be used although it may produce conflicts when combined with some available libraries.

The implemented service can scale to any number of sequential or parallel processes. Users can implement their own time alarms through the system time functions or signals, but then they would have to deal with every implementation detail. Managing multiple alarms inside the same process can be quite complex but doing it with the synchronization service is as easy as for the single alarm case.

5 Computational Results

The instances used in [2, 4, 5, 7, 9] are solved by the sequential algorithm in a negligible time. For that reason, the computational study here presented has been performed on some selected instances from the ones available at [10]. Tests have been run on a cluster of 8 HP nodes, each one consisting of two Intel(R) Xeon(TM) at 3.20GHz. The interconnection network is an Infiniband 4X SDR. The compiler and MPI implementation used were *gcc 3.3* and MVAPICH *0.9.7* [11].

Table 1 presents the results for the sequential runs. The first column shows the exact solution value for each problem instance. The next two columns show the solution value given by the initial lower bound and the time invested in its calculation (all times are in seconds). Note that the designed lower bound highly approximates the final exact solution value. In fact, the exact solution is directly reached in many cases. Last column compares two different upper bounds: the one proposed in [4] and the new upper bound. For each upper bound, the time needed for its initialization, the search time, that is, the time invested in finding the exact solution without including bounds calculations, and the number of computed nodes are presented. Computed nodes are the nodes that have been transferred from OPEN to CLIST and combined with all previous CLIST elements. The new upper bound highly improves the previous bound: the number of computed nodes decreases, yielding a decrease in the execution time.

Table 2 presents the results obtained for the parallel algorithm. The search time and the number of computed nodes are shown. For 16-processors, the speedup in relation to the sequential algorithm is also presented. Both algorithms make use of the improved bounds. Figure 4 represents the speedups for the three problems with best, worst and intermediate parallel behaviours. Note that the sequential algorithm and the 1-processor parallel algorithm compute exactly the same number of nodes, but the parallel implementation introduces an overhead over the sequential algorithm. When the number of processors increases the parallel algorithm improves its behaviour. In those cases where the

	Solution	Lower Bound		Upper Bound					
				V			U_V		
Problem	Value	Value	Time	Init	Search	Nodes	Init	Search	Nodes
07_25_03	21693	21662	0.442	0.0309	2835.07	179360	0.0312	2308.78	157277
07_25_05	21693	21662	0.436	0.0311	2892.23	183890	0.0301	2304.78	160932
07_25_06	21915	21915	0.449	0.0316	35.55	13713	0.0325	20.83	10310
07_25_08	21915	21915	0.445	0.0318	205.64	33727	0.0284	129.03	25764
07_25_09	21672	21548	0.499	0.0310	37.31	17074	0.0295	25.49	13882
07_25_10	21915	21915	0.510	0.0318	1353.89	86920	0.0327	1107.18	73039
07_50_01	22154	22092	0.725	0.1056	2132.23	126854	0.0454	1551.23	102662
07_50_03	22102	22089	0.793	0.0428	4583.44	189277	0.0450	3046.63	148964
07_50_05	22102	22089	0.782	0.0454	4637.68	189920	0.0451	3027.79	149449
07_50_09	22088	22088	0.795	0.0457	234.42	38777	0.0428	155.35	29124
07_100_08	22443	22443	1.218	0.0769	110.17	25691	0.0760	92.91	22644
07_100_09	22397	22377	1.278	0.0756	75.59	20086	0.0755	61.84	17708

Table 1. Lower and Upper Bounds Results

	processors										
	1		2		4		8		16		
Problem	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Sp.
07_25_03	2922.94	157277	1665.26	161200	770.47	157281	384.05	159424	197.82	157603	11.67
07_25_05	3068.19	160932	1738.02	168941	863.23	168867	408.39	165323	206.10	162029	11.18
07_25_06	23.82	10310	11.51	10310	6.36	10310	3.01	10310	1.57	10310	13.26
07_25_08	129.02	25764	61.38	25764	29.98	25764	15.52	25764	8.33	25764	15.48
07_25_09	29.44	13882	13.69	14257	7.02	13916	3.57	13916	2.09	14150	12.44
07_25_10	1140.41	73039	539.89	73039	266.96	73039	132.32	73039	67.94	73039	16.16
07_50_01	1651.51	102662	963.07	102662	598.67	116575	240.93	103545	123.72	102965	12.53
07_50_03	4214.54	148964	2084.77	148964	1057.70	151362	512.12	150644	258.51	149039	11.78
07_50_05	4235.27	149449	2141.41	149449	1077.47	153813	512.43	150937	260.03	149450	11.64
07_50_09	161.38	29124	77.65	29124	40.34	29124	19.45	29124	10.34	29124	14.94
07_100_08	98.96	22644	48.74	22644	25.83	22644	12.60	22644	6.98	22644	13.31
07_100_09	60.05	17708	38.29	19987	18.74	18509	10.59	20584	4.77	18100	12.58

Table 2. Parallel Algorithm Results

heuristic reaches the exact solution, the parallel and the sequential versions always compute the same number of nodes and consequently better speedups are achieved. A few superlinear speedups appear due to cache effects.

The number of computed nodes per processor is shown in Figure 5. The chosen problems were those with the best and worst speedups. It clearly indicates that work load is fairly balanced even though the configuration parameters were not tuned in a per-problem basis.

6 Conclusions

This work presents a new lower bound and a new upper bound for the 2DCSP. Computational results prove the quality of such new bounds. A new parallel distributed and synchronous algorithm has been designed from the basis of the inherently sequential VB algorithm. Parallel results demonstrate the almost linear speedups and verify the high scalability of the implementation. Furthermore, a totally application-independent synchronization service has been developed. The

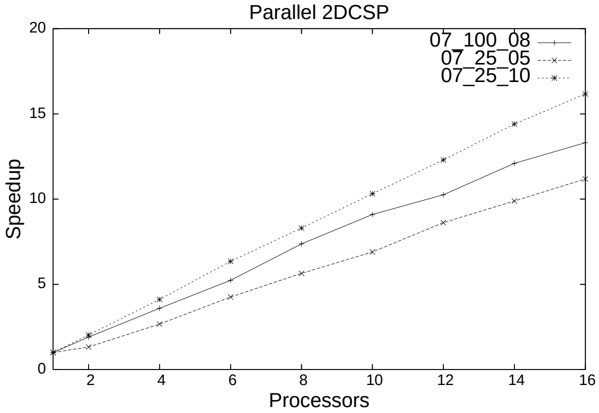

Fig. 4. Parallel 2DCSP: Speedup

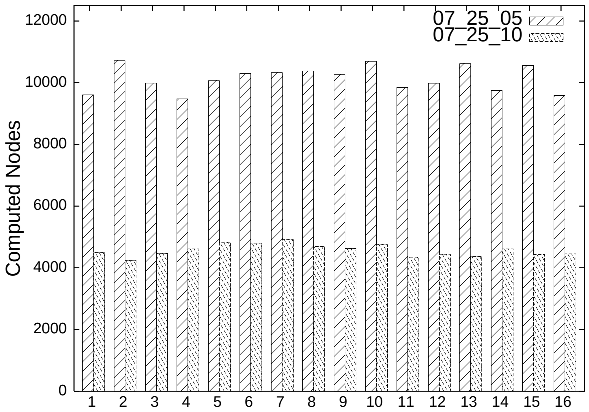

Fig. 5. Parallel 2DCSP: Load Balancing

service provides an easy way of introducing periodic synchronizations in the user programs. The synchronization service has been decisive for the well operation of the parallel scheme and for the right behaviour of the load balancing model in the presented application.

Some improvements can be added to the current implementation. The first one refers to the load balancing scheme and lies in introducing some method to approximately calculate the work associated to each of the meta-rectangles in OPEN. Instead of considering only the size of the lists, it would be better to consider the work load that they will generate. The other concern is related to the synchronization scheme. At the initial and latest stages of the search, many of the alarms are cancelled because processors do not have enough work. It would be interesting to have an automatic and dynamic way of fixing the time between synchronizations while the search process is progressing.

7 Acknowledgements

This work has been supported by the EC (FEDER) and by the Spanish Ministry of Education and Science inside the 'Plan Nacional de I+D+i' with contract number TIN2005-08818-C04-04. The work of G. Miranda has been developed under the grant FPU-AP2004-2290.

References

1. Wang, P.Y.: Two Algorithms for Constrained Two-Dimensional Cutting Stock Problems. Operations Research **31**(3) (May-June 1983) 573–586
2. Viswanathan, K.V., Bagchi, A.: Best-First Search Methods for Constrained Two-Dimensional Cutting Stock Problems. Operations Research **41**(4) (1993) 768–776
3. Gilmore, P.C., Gomory, R.E.: The Theory and Computation of Knapsack Functions. Operations Research **14** (March 1966) 1045–1074
4. Hifi, M.: An Improvement of Viswanathan and Bagchi's Exact Algorithm for Constrained Two-Dimensional Cutting Stock. Computer Operations Research **24**(8) (1997) 727–736
5. Cung, V.D., Hifi, M., Le-Cun, B.: 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)
6. Nicklas, L.D., Atkins, R.W., Setia, S.K., Wang, P.Y.: The Design and Implementation of a Parallel Solution to the Cutting Stock Problem. Concurrency - Practice and Experience **10**(10) (1998) 783–805
7. Tschöke, S., Holthöfer, N.: A New Parallel Approach to the Constrained Two-Dimensional Cutting Stock Problem. In Ferreira, A., Rolim, J., eds.: Parallel Algorithms for Irregularly Structured Problems, Berlin, Germany, Springer-Verlag (1995) 285–300
8. Tschöke, S., Polzer, T.: Portable Parallel Branch-and-Bound Library - PPBB-LIB, Department of Computer Science, University of Paderborn, D-33095 Paderborn, Germany. (December 1996)
9. García, L., León, C., Miranda, G., Rodríguez, C.: A Parallel Algorithm for the Two-Dimensional Cutting Stock Problem. In: European Conference on Parallel Computing (Euro-Par). Volume 4128., Dresden, Germany, Lecture Notes in Computer Science, Springer-Verlag (2006) 821–830
10. Group, D.O.R.: Library of Instances (Two-Constraint Bin Packing Problem) http://www.or.deis.unibo.it/research_pages/ORinstances/2CBP.html.
11. Network-Based Computing Laboratory, Dept. of Computer Science and Eng., The Ohio State University: MPI over InfiniBand Project. (2006) http://nowlab.cse.ohio-state.edu/projects/mpi-iba.