Short Paper

A Parallel Bucket-Based Bottom-Left-Fill Algorithm for Nesting Problems Using a Semi-Discrete Representation

Sahar Chehrazad
Department of Computer Science, KULeuven
Leuven, Belgium
Sahar.Chehrazad@kuleuven.be

Dirk Roose
Department of Computer Science, KULeuven
Leuven, Belgium
Dirk.Roose@kuleuven.be

Tony Wauters
Department of Computer Science, KULeuven
Technologiecampus, Gent, Belgium
Tony.Wauters@kuleuven.be

ABSTRACT

We discuss the parallelization of the bottom-left-fill algorithm to solve nesting problems on multicore processors. The algorithm uses a semi-discrete representation of both the 2D non-convex pieces and the strip. In case several rotation angles are allowed for each piece, the computation of the tentative placement for each angle is parallelized using OpenMP, limiting the number of threads to the number of rotation angles. The speedup is further limited due to load imbalance. Therefore, we develop a bucket-based bottom-left-fill algorithm, which subdivides the pieces in buckets and places the buckets, while computing an optimal ordering of the pieces in each bucket to improve the solution, i.e., to minimize the length of the strip. The many possible orderings of the pieces in a bucket allow to create many tasks that can be executed concurrently. Dynamic load balancing is used. We evaluate the parallel performance and the solution quality of this new algorithm, using different bucket sizes.

KEYWORDS

cutting and packing, nesting, bottom-left-fill algorithm, multicore, OpenMP

1 INTRODUCTION

Nesting problems, a branch of cutting and packing problems, are important for many industries, e.g. textile, sheet metal, leather, glass and paper industries, see [2], and also in additive manufacturing, see [7]. In this paper we deal with placing 2D, possibly non-convex, pieces without overlap on a strip, a rectangular sheet with a fixed width and a variable length, while minimizing the used length of the strip. This is known as the irregular strip packing problem, which is a branch of 'open dimension' problems, see [14]. Depending on the application, the pieces can be rotated or not. In case rotation is allowed, typically only a few rotation angles are allowed.

Many heuristic methods have been proposed to solve this NP-complete problem. Bottom-left-fill is a simple greedy heuristic algorithm, in which the pieces are placed one by one in the strip in the left-most and bottom-most position for which the piece does not overlap any of the already placed pieces ('fill' indicates that a piece can be placed in the empty space between already placed pieces). The ordering in which the pieces are placed influences the solution quality, i.e., the used length of the strip. Hence, many (meta-)heuristics have been proposed, that use the bottom-left-fill algorithm as a 'building block', e.g., by permuting the order in which pieces are placed.

In most applications the pieces are polygons and many methods use the original polygonal geometry of the pieces [9, 11], but some methods use a discretization of the pieces and/or the strip. In the latter case, a 2D discretization can be used [12, 13] or a 1D discretization, also called semi-discretization [1, 10].

In [5] we presented an efficient bottom-left-fill algorithm using semi-discretization of the pieces and the strip along the horizontal x-axis. The semi-discretized pieces and the (partially) filled strip are represented by line segments lying on equidistant vertical lines, called 'resolution lines'. The bottom-left-fill algorithm discretizes and places the pieces one by one, sorted according to e.g. decreasing bounding box area. For each piece, all allowed rotation angles are considered. For each rotation angle, the rotated piece is tentatively placed in the left-most and bottom-most free position in the partially filled strip without overlap. Eventually, the piece is placed with the rotation angle that gives the best result according to some optimality criteria (see next section).

We assume that the bottom-left corner of the strip has (strip) coordinates $(x, y) = (0, 0)$ and that the bottom left corner of the axis-aligned bounding box of each piece has (local) coordinates $(0, 0)$. Placing a (rotated) piece in the left-most and bottom-most available position in the partially filled strip without overlap requires to find a 'translation vector' $t = (x_t, y_t)$, that indicates the position in the strip of the bottom-left corner of the axis-aligned bounding box of the piece, such that all line segments of the semi-discretized representation of the piece can be placed in the strip without overlap. For each piece, the subsequent translation vectors correspond to the low endpoints of the *empty* vertical line segments in the strip, traversing the resolution lines from left to right and, for each resolution line, traversing the empty vertical segments from bottom to top.

In principle, for a given translation vector, for all line segments it must be tested whether they can be placed in the strip without overlap with already placed line segments. However, in case some line segments of the piece cannot be placed without overlap, not all tests must be performed to decide that the current translation vector does not allow placement without overlap. Indeed, as soon as a line segment cannot be placed, we update the translation vector t. Figure 1 shows the placement of a piece.

In [5] we present two ways to further reduce the number of overlap tests. First, the overlap tests of a piece covering $L + 1$ resolution lines are not performed by considering the line segments of a piece in subsequent resolution lines, but in the order $\{0, L, L/2, L/4, 3L/4, \ldots\}$ which quickly gives information about several parts of the piece. Second, if several copies of the same piece are placed consecutively, the piece is discretized only once.

10.48786/alioeuro.2022.03

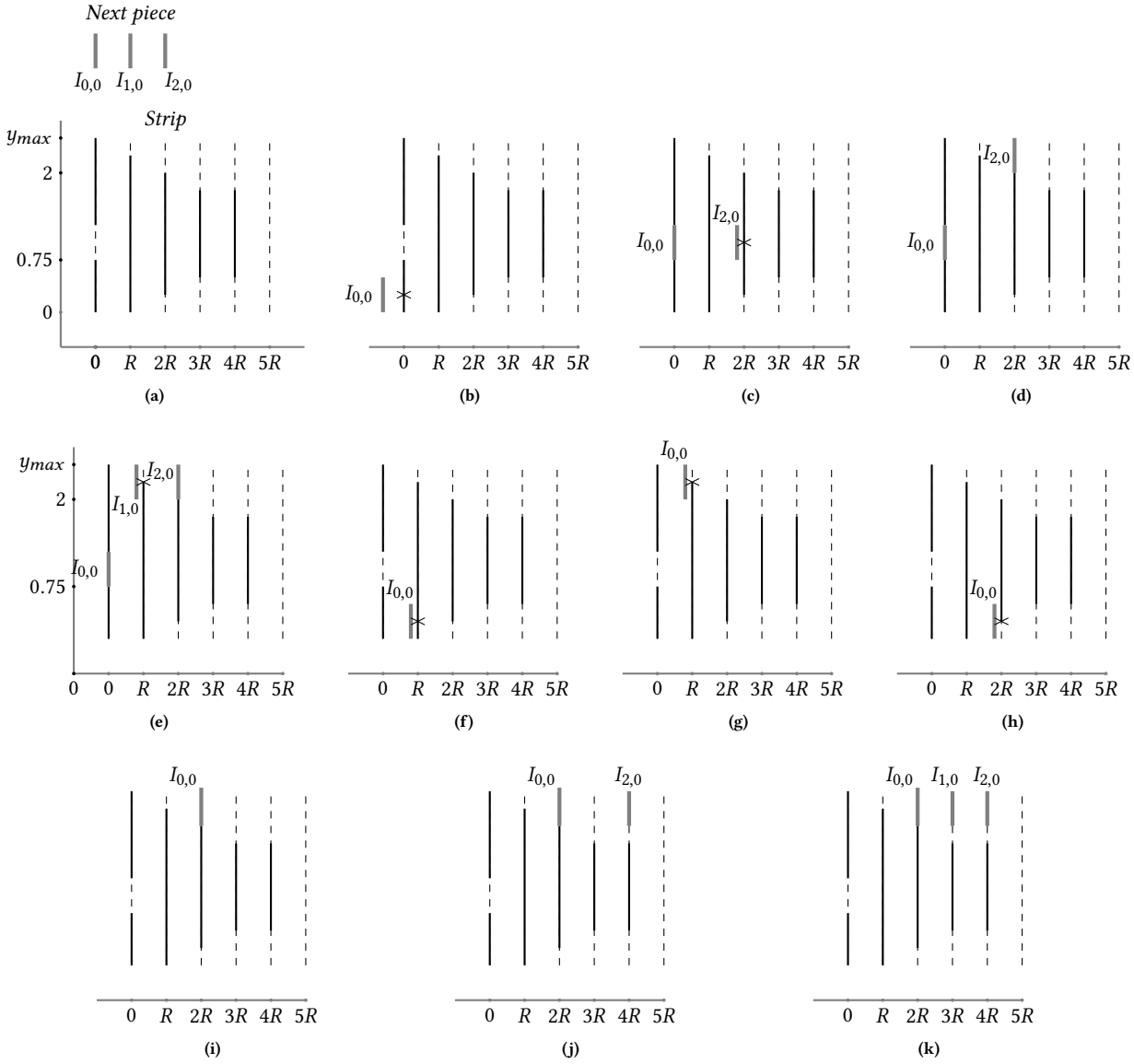
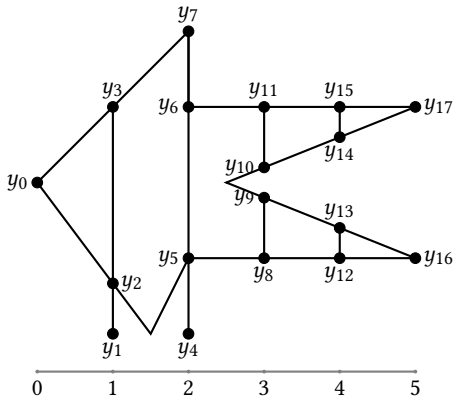


Figure 1: Placement algorithm: (a) represents the strip with already placed pieces where full lines indicates filled line segments and dashed lines indicate empty line segments on the resolution lines with $x = i \times R$, $i = 0, 1, 2, \ldots$. In (b), line segment $I_{0,0}$ cannot be placed with $t = (0, 0)$ so we shift it upwards. In (c), $I_{0,0}$ is placed at $x = 0$ with $t = (0, 0.75)$, but $I_{2,0}$ cannot be placed with this translation t. Therefore, in (d), we shift $I_{2,0}$ upwards, t is updated to $(0, 2)$. However, in (e) $I_{1,0}$ cannot be placed with this translation t. We update t to shift the line segments to the right. We return to $I_{0,0}$. In (f) and (g), we cannot place $I_{0,0}$ at $x = R$. Therefore, we again update t to shift the line segments to the right. In (i), (j) and (k), $I_{0,0}$, $I_{2,0}$ and $I_{1,0}$ can be placed at respectively $x = 2R$, $x = 4R$ and $x = 3R$ with $t = (2R, 2)$.

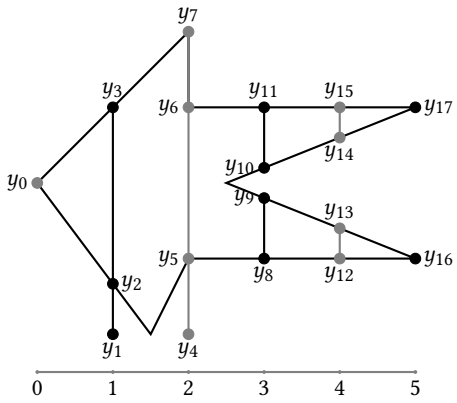
Also, the translation vectors considered for the next copy of that piece start from the translation vector that allowed to place the previous copy. More information on the discretization, discretization error, resolution and the placement algorithm and the obtained quality can be found in [5].

In [5] the bottom-left-fill algorithm is implemented sequentially. The execution time is very low in comparison with other implementations of the bottom-left-fill algorithm – without or with (semi-)discretization – due to the use of appropriate data structures and algorithm organisation in order to achieve high performance on a single core (e.g., by maximizing cache hits).

Since current computers are multicore processors, a parallel implementation using several cores allows to further reduce the runtime. The parallelization should be done at the right level, such that threads perform enough work between synchronisation points, i.e. the grain size should be sufficiently large in order to achieve high parallel efficiency and thus speedup. Note that bottom-left-fill is a simple greedy heuristic to solve nesting problems, and is often combined with (meta)heuristics to improve the quality of the placement. If the (meta)heuristic allows to execute several bottom-left-fill placements concurrently, e.g., in an evolutionary algorithm [3, 6, 10], then the parallelization should be done at the level of the (meta)heuristic. However, several



(a) A discretized piece with $R = 1$.



(b) The segments of a piece are assigned to two threads, i.e. threads 0 and 1 perform the overlap tests for respectively the gray and black segments.

Figure 2: A low-level parallelization of the bottom-left-fill algorithm.

(meta)heuristics use the bottom-left-fill algorithm as a 'building block' in a sequential way, see e.g. [1, 9, 11, 12]. In these cases the bottom-left-fill algorithm itself should be parallelized.

We briefly discuss two approaches to parallelize the bottom-left-fill algorithm, with different levels of parallelism, which were not selected because of their disadvantages.

In a *high-level parallelization* approach, the N sorted pieces are assigned to P threads in a cyclic way, such that each thread handles an appropriate sample of the sorted pieces. All threads place the N/P assigned pieces concurrently on separate strips, with the same width (in the y-direction). The threads are then synchronized and all strips are concatenated along the x-direction to make the final solution, see Fig. 3. This approach typically leads to a large strip length, because of the holes around the concatenation x-coordinates. This problem could be solved by parallelizing only the placement of the M largest pieces, and sequentially placing the $N - M$ smallest pieces afterwards to fill the holes. If M is large, the solution quality is low due to the impossibility to fill all holes by the $N - M$ smallest pieces. If M is small, many pieces are placed sequentially, which limits the speedup.

In a *low-level parallelization* approach, the placement of a piece, with a given rotation angle, is parallelized. Checking whether a piece can be placed in the strip with a given translation vector

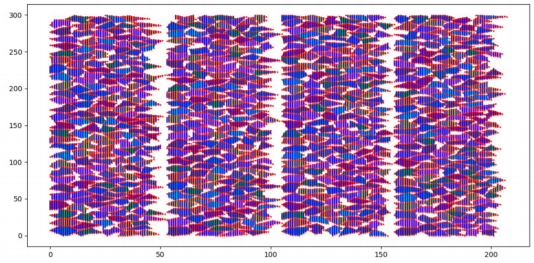


Figure 3: The pieces are divided between 4 threads. 4 strips are concatenated along the x-direction to make the final solution.

requires to perform an overlap test for each line segment of that piece. These tests could be performed concurrently, but this parallelization approach would be quite inefficient. Indeed, assume that the overlap tests are assigned to two threads, see Fig. 2. If each thread checks the placement of all assigned segments with the current translation vector t, then afterwards synchronization is needed to combine the partial results and decide whether the piece can be placed with this t or whether t must be updated. However, in the sequential algorithm the translation vector t is updated as soon as a segment cannot be placed without overlap with the current t. In the parallel algorithm, unnecessary checks are performed which reduce the efficiency of the algorithm. The unnecessary checks could be avoided by increasing the number of synchronization points. However, the overhead caused by synchronization will reduce the parallel efficiency and the speedup.

In this paper we first present an *intermediate level parallelization*, which limits the number of threads that can be used and thus the speedup, to the number of allowed angles. Afterwards, we present a bucket-based bottom-left-fill algorithm, that can improve the solution quality and that allows to execute many tasks in parallel on a multicore computer.

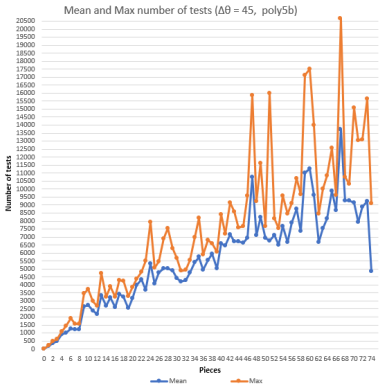


Figure 4: Placement of each piece of the poly5b data set: mean and maximum number of tests performed for the 8 allowed rotation angles.

The remainder of the paper is organized as follows. Section 2 presents the parallelization of the bottom-left-fill algorithm. Section 3 presents the parallelization of the bucket-based bottom-left-fill. In section 4 computational results are presented, providing insight in the parallel performance of the method. Finally, in section 5 we draw some conclusions.

2 PARALLELIZATION OF THE BOTTOM-LEFT-FILL ALGORITHM

The bottom-left-fill algorithm iterates over the pieces and for each piece all allowed rotation angles are considered. For each rotation angle, a separate thread checks the placement of the piece in the strip. The thread (rotation angle) that minimizes the largest x-coordinate of the tentatively placed piece wins. In case several threads lead to the minimum, the thread that minimizes the maximal y-coordinate of the tentatively placed piece wins. In case there is still a 'tie break' the thread with the smallest ID wins. The parallelization is done using OpenMP [4]. In order to limit the fork-join overhead, we create the threads only once before starting the iteration over the pieces and keep them active until the end of the algorithm. Global variables are used to store the results of the thread with the best result so far. The winning thread consolidates the placement in the strip and resets the global variables. The threads then start placing the next piece.

This approach not only limits the number of threads that can be used to the number of allowed angles, and thus the speedup, but also causes substantial load imbalance since the number of overlap tests varies with the rotation angle. As explained in the previous section, parallelization of the bottom-left-fill algorithm at a lower and a higher level are also very inefficient or can lead to a low solution quality. Therefore, we present in the next section a bucket-based bottom-left-fill algorithm, which not only can improve the solution quality, but can also be parallelized more efficiently.

3 PARALLELIZATION OF THE BUCKET-BASED BOTTOM-LEFT-FILL ALGORITHM

It is well known that the solution quality obtained by the (greedy) bottom-left-fill algorithm can be improved by reordering the pieces, see [3]. Checking all permutations of the pieces, especially considering all rotation angles for each piece in each permutation, is prohibitively expensive. Subdividing pieces in 'buckets' and placing the buckets one by one in the strip, while finding an optimal placement of the pieces in each bucket, reduces the number of permutations considered but still can improve the solution quality, compared to the classical bottom-left-fill algorithm.

Suppose that the pieces are subdivided in k buckets with size b (the last bucket might have less than b pieces). Considering all permutations of the pieces in a bucket and rotating each piece with f rotation angles leads to $b! \times f^b$ possibilities. All these possibilities could be evaluated concurrently, but we create tasks with sufficient grain-size by grouping f possibilities as follows. In each task the ordering of the b pieces is fixed, as well as the rotation angle of the first $b-1$ pieces, and all rotation angles of the last piece are evaluated, to find the rotation angle that minimizes the maximal x-coordinate of the pieces in the bucket, when placed in the strip without overlap. This leads to $b! \times f^{b-1}$ tasks that can be executed concurrently.

Compared to the parallelization of the classical bottom-left-fill algorithm, the potential speedup is not limited to the number of rotation angles and the (relative) synchronization overhead is lower, due to the larger 'grainsize' of the tasks.

We have implemented this approach using OpenMP, assuming that the number of threads can vary. Initially the empty strip is stored in global variable *Global_Strip*. Each thread gets a copy of the strip in its local memory. We create the threads only once and keep them active until the end of the algorithm. Global variables are used to store the result of the thread that minimizes the largest x-coordinate of the tentatively placed pieces. In case there is a 'tie break', the result of the thread that minimizes the maximal y-coordinate of the tentatively placed pieces will be stored in the global variables.

Each thread gets a task from the task pool and performs tentative placement of the assigned permutation and rotations of the pieces. The thread compares its result, i.e. the best result considering all rotation angles of the last piece, with the values in the global variables and updates them in case the thread obtains a better result. Finally, the main thread updates *Global_Strip* and resets the other global variables. The threads then start the placement of the next bucket. After placing all buckets, the threads are joined.

4 RESULTS

In this section we discuss the performance of the parallel algorithms described in the previous sections, implemented in C++ using the Qt environment with OpenMP, compiled using gcc. The experiments were carried out on Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz (18 cores).

We present results for the 'poly5b' data set [8] and a data set with 550 randomly generated pieces. In the latter data set, the pieces are generated with 4 different diameters, 80% of the pieces have the smallest diameter and most pieces are non-convex. Both data sets contain both non-convex and convex pieces, varying in size. Table 1 shows characteristics of the data sets. The resolution R, i.e. the distance between consecutive resolution lines, see Fig. 1, is chosen such that the non-vertical edge with the smallest x-projection in the data set is represented (discretized) by 10 line segments. The pieces are sorted according to decreasing axis-aligned bounding box area; this allows to place small pieces in the gaps between already placed large pieces.

4.1 Parallelization of the bottom-left-fill algorithm

For the parallelization of the bottom-left-fill algorithm we consider 4 different cases: no rotation allowed ($\theta = 0°$), two rotations with $\Delta\theta = 180°$, four rotations with $\Delta\theta = 90°$ and eight rotations with $\Delta\theta = 45°$. Since the (tentative) placement for each angle is executed by a separate thread, the number of threads (and cores of the multicore processor) equals the number of rotation angles.

Table 2 shows the execution times (time to compute the placement, including the semi-discretization, averaged over 100 runs) of the parallel execution of the bottom-left-fill algorithm. Since the computations for each rotation angle are performed by a separate thread, the parallel execution time ideally would be independent of the number of rotation angles. However, in practice the execution times grow with the number of allowed rotations, mainly due to load imbalance. Hence the speedup is smaller than the number of rotation angles.

To illustrate the load imbalance, the number of tests to tentatively place a piece with each allowed rotation angle is counted, for each piece of the poly5b data set. Fig. 4 presents the mean and the maximum number of tests performed for each piece. The sometimes large difference between the mean and maximum values results in substantial load imbalance during the placement of a piece.

Table 1: Information on the data sets used for computational experiments.

Data set	Number of pieces	Number of different pieces	Minimum and maximum x-projection of pieces	Resolution
poly5b	75	75	[3,12]	0.1
550-random	550	550	[1,27.2]	0.1

Table 2: Execution times of the parallel bottom-left-fill algorithm for poly5b and 550-random data sets, time: time (ms) to compute semi-discretization + time of the placement.

Data set		Allowed rotations = number of threads			
		1 ($\theta = 0°$)	2 ($\Delta\theta = 180°$)	4 ($\Delta\theta = 90°$)	8 ($\Delta\theta = 45°$)
poly5b	Parallel time (ms)	4.9	5.9	6.9	7.6
	Speedup	1	1.6	2.5	4.6
	Strip length	*72.4*	*68.5*	*66*	*65.9*
550-random	Parallel time (ms)	82.0	100.5	113.3	132.3
	Speedup	1	1.7	2.9	5.3
	Strip length	*112.6*	*111.9*	*111.4*	*111.1*

Table 3: Information on the parallelization of the bucket-based bottom-left-fill algorithm.

Data set	Number of rotation angles f	Size of buckets	Number of buckets	Number of tasks per bucket
poly5b	8 ($\Delta\theta = 45°$)	3; 4	25; 19	384; 12288
550-random	8 ($\Delta\theta = 45°$)	3; 4	184; 138	384; 12288

Table 4: Execution times of the parallel bucket-based bottom-left-fill algorithm for poly5b and 550-random data sets, time: time (ms) to compute semi-discretization + time of the placement.

Data set	Bucket Size	Sequential time (ms)	Parallel time (ms) (10 threads)	Parallel time (ms) (18 threads)	Speedup (10 threads)	Speedup (18 threads)	Strip length
poly5b	3	4870	520	320	9.4	15.2	63.1
	4	118912	12482	7724	9.5	15.4	63.5
550-random	3	131418	14530	8860	9.0	14.9	108.9
	4	3389784	369454	225834	9.2	15.0	109.5

4.2 Parallelization of the bucket-based bottom-left-fill algorithm

Table 3 shows the information on the parallelization of the bucket-based bottom-left-fill algorithm. For both data sets $f = 8$ rotation angles ($\Delta\theta = 45°$) are allowed. As explained in the previous section, every task evaluates the placement of a bucket with a given permutation of the pieces in the bucket with fixed angles, except for the last piece, for which $f = 8$ rotation angles are considered.

Table 4 shows the execution times for the parallel bucket-based bottom-left-fill algorithm. The results show that, using 10 threads, we obtained a speedup close to 10, due to the dynamic load balancing of tasks by using a task pool. Note that the overhead caused by creating threads is minimal since threads are created only once. Using 18 threads, the speedup increases, but the parallel efficiency decreases, due to e.g. the critical section needed to access the task pool.

Table 4 also shows that using buckets improves the solution quality, i.e., the strip length, compared to the simple bottom-left-fill algorithm, since the pieces within each bucket are placed in an optimal way. However, increasing the bucket size does not necessarily lead to a better global solution due to the greedy nature of the heuristic.

For bucket size equal to two, the small number of tasks (i.e., 16) limits the number of threads that can be used. Also, tests with bucket size 2 indicate that the resulting strip length is worse than when using larger bucket sizes.

Note that in our experiments the buckets consist of pieces that are consecutive in the ordering of the pieces according to decreasing axis-aligned bounding box area. We expect that the solution quality of the bucket-based bottom-left-fill algorithm can be improved by selecting the pieces that form a bucket in a more sophisticated way.

For the poly5b data set, we can compare the performance and the execution time of the parallel bucket-based bottom-left-fill algorithm with those of the algorithms presented in [3]. The

sequential bottom-left-fill algorithm in [3] uses exact geometry of the pieces and a semi-discrete strip and requires 14700 ms (on Pentium 4, approximately 30 times slower than 1 core of the processor used in our experiments; hence we expect that it would require ≈ 490 ms on 1 core of our processor). The achieved strip length should be equal to the length obtained with our bottom-left-fill algorithm, see [5] for more details. Using meta-heuristics on top of the bottom-left-fill algorithm, a strip length of 60.5 is achieved in [3], but the sequential execution time increases to 676610 ms (i.e., ≈ 22550 ms on 1 core of our processor). The strip length obtained by the bucket-based bottom-left-fill algorithm with bucket size 3 is worse (i.e., 63.1) but requires a much lower execution time, especially when many cores are used (i.e., 320 ms on 18 cores).

Since the parallel bucket-based bottom-left-fill algorithm, with bucket size 3, typically results in a shorter strip length than the original bottom-left-fill algorithm, while the execution time is still low, it can be used as an efficient 'building block' to implement various (meta)heuristics already proposed in the literature.

5 CONCLUSION

We presented the parallelization of the bottom-left-fill algorithm and a variant of it to solve 2D nesting problems using semi-discretization of both the pieces and the strip. We started from a very fast sequential code, that substantially outperforms other implementations of the bottom-left-fill algorithm, with or without semi-discretization.

In the parallel bottom-left-fill algorithm, the computations required to (tentatively) place a piece for each allowed rotation angle are performed concurrently. This approach limits the number of threads, and thus the speedup, to the number of allowed rotation angles. We have shown that the speedup of the algorithm is mainly limited by the load imbalance in the computations for each rotation angle.

We developed a variant of the bottom-left-fill algorithm which subdivides the pieces into buckets and improves the quality of the solution by selecting the best ordering of the pieces in each bucket. This method allows to create many tasks that can be executed in parallel. By using dynamic load balancing a high speedup is achieved on a multicore processor with up to 18 cores. Increasing the bucket size does not always lead to a better global solution.

Since the parallel bucket-based bottom-left-fill algorithm, e.g. with bucket size 3, achieves a shorter strip length than the original bottom-left-fill algorithm, while the execution time is still low, especially when many cores are used, it can be used as an efficient 'building block' for integration in various (meta)heuristics proposed in the literature.

ACKNOWLEDGMENTS

We acknowledge the financial support of Interne Fondsen KU Leuven / Internal Funds KU Leuven, project C24/17/048.

REFERENCES

[1] Rajasekhar Akunuru and N. Ramesh Babu. 2013. A semi-discrete geometric representation for nesting problems. *European Journal of Operational Research* 51, 14 (2013), 4155–4174. https://doi.org/10.1080/00207543.2012.751508

[2] Julia A. Bennell and Jose F. Oliveira. 2008. The geometry of nesting problem: a tutorial. *European Journal of Operational Research* 184 (2008), 397–415.

[3] Edmund Burke, Robert Hellier, Graham Kendall, and Glenn Whitwell. 2006. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operational Research* 54, 3 (2006), 587–601. https://doi.org/10.1109/TASE.2006.874973

[4] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. 2001. *Parallel Programming in OpenMP.* Academic Press, San Diego.

[5] Sahar Chehrazad, Dirk Roose, and Tony Wauters. 2021. A fast and scalable bottom-left-fill algorithm to solve nesting problems using a semi-discrete representation. *European Journal of Operational Research* Accepted (2021). https://doi.org/10.1016/j.ejor.2021.10.043

[6] A.Miguel Gomes and Jose F.Oliveira. 2002. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research* 141, 2 (2002), 359–370. https://doi.org/10.1016/S0377-2217(02)00130-3

[7] Valeriya. Griffiths, James P. Scanlan, Murat H. Eres, Antonio. Martinez-Sykora, and Phani. Chinchapatnam. 2019. Cost-driven build orientation and bin packing of parts in Selective Laser Melting (SLM). *European Journal of Operational Research* 273 (2019), 334–352.

[8] Eva Hopper. 2000. *Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods.* Ph.D. Dissertation. University of Wales, Cardiff School of Engineering.

[9] Bonfim Amaro Junior, Plácido Rogerio Pinheiro, Rommel Dias Saraiva, and Pedro Gabriel Calíope Dantas Pinheiro. 2014. Dealing with None regular Shapes Packing. *Mathematical Problems in Engineering* 2014 (2014), 587–601. https://doi.org/10.1155/2014/548957

[10] Heng Ma and Chia-Cheng Liu. 2007. Fast Nesting of 2-D Sheet Parts With Arbitrary Shapes Using a Greedy Method and Semi-Discrete Representations. *IEEE Transactions on Automation Science And Engineering* 4, 2 (2007), 273–282. https://doi.org/10.1109/TASE.2006.874973

[11] Placido R. Pinheiro, Bonfim Amaro Junior, and Rommel D. Saraiva. 2016. A random-key genetic algorithm for solving the nesting problem. *International Journal of Computer Integrated Manufacturing* 29, 11 (2016), 1159–1165. https://doi.org/10.1080/0951192X.2015.1036522

[12] Andre Kubagawa Sato, Thiago Castro Martins, Antonio Miguel Gomes, and Marcos Sales Guerra Tsuzuki. 2019. Raster penetration map applied to the irregular packing problem. *European Journal of Operational Research* 279, 2 (2019), 657–671. https://doi.org/10.1016/j.ejor.2019.06.008

[13] Andre Kubagawa Sato, Marcos Sales Guerra Tsuzuki, Thiago Castro Martins, and Antonio Miguel Gomes. 2016. Study of the grid size impact on a raster based strip packing problem solution. *IFAC-PapersOnLine* 49, 31 (2016), 143–148.

[14] Gerhard Waascher, Heike Haußner, and Holger Schumann. 2007. An improved typology of cutting and packing problems. *European Journal of Operational Research* 183 (2007), 1109–1130.