Partitioning of Complex Discrete Models for Highly Scalable Simulations

Submitted: 9th May 2024; accepted: 10th February 2013

Jakub Ziarko, Mateusz Najdek, Wojciech Turek

DOI: 10.14313/jamris-2025-033

Abstract:
The need for more and more accurate simulations of groups of autonomous beings has directed the researcher's attention towards ways of parallelizing simulation algorithms. Parallel execution of discrete simulation models update methods requires their division of labor between workers. Existing methods used for grid division aim at providing equal areas of fragments and minimizing the length of the resulting borders. However, in real-life simulations, other factors also have to be considered. This paper presents a method for grid partitioning, that also allows for defining indivisible areas, considers complex shapes of real-life environments, and supports division suitable for defined architectures of nodes and cores. The method is evaluated using several scenarios, which provided satisfactory results.

Keywords: *Simulation, Model Partitioning, Scalability, High Performance Computing*

1. Introduction

Methods for simulating complex phenomena observed in societies of autonomous beings have received significant attention over the last decades [14]. Modelling and simulation of herds of animals or groups of people in various environments and situations can lead to better understanding of patterns and laws in the surrounding world. It can also be used for predicting future states in such systems, which has numerous applications in planning and management. The most widespread approach to the problem of modelling environments and the beings in them is inspired by the assumptions used in cellular automatons. Despite the significant simplifications imposed by the discretization of space and time, the grids of cells representing the physical space, and the beings, which can occupy one cell at a time, these simulations have been proven useful in a variety of cases, from evacuations modelling [9] to traffic simulations [8].

Continuous development of these type of simulations inevitably leads to an increase in the demand for their performance. The desire to simulate larger environments and more numerous groups of beings, and to add more details to the models, increases the amount of computations. At the same time, users find new, demanding applications for such simulations and require results to be provided very fast.

For example, a simulation can be used for evaluating automatically-generated scenarios in optimization algorithms or as a basis for real-time management methods [16]. Such a demand cannot be satisfied by a single computer, and this leads directly to the concept of simulation algorithm parallelization.

The problem of parallel execution of grid-based simulation algorithms is typically solved by splitting the grid into parts, which are then updated independently by separate worker processes. This approach leads to several challenging problems related to state synchronization between workers [10].

This paper focuses on the issue of efficient utilization of available computing resources, which requires proper division of computational tasks between the workers. The division of the grid must split the task into parts, which will uniformly load the available computing resources and limit the scope of the synchronization. In the existing approaches, which will be discussed in details in Section 2, these two factors are typically translated into two metrics: the standard deviation of the fragments' sizes, and the total length of fragments' common borders (referred to as edge-cut). Although these metrics seem adequate, they do not reflect all the issues related to parallel simulation of complex models on parallel hardware. There are at least three other issues that should be considered by the grid division algorithm:

1) the structure of the environment,
2) the complexity of synchronization in different fragments of the environment, and
3) the architecture of the computing hardware.

The structure of the environment (1) in real-life scenarios is typically far more complex than the uniform, rectangular-shaped model. It contains complex shapes of spaces accessible for beings that are separated by inaccessible fragments (e.g., rooms separated by walls). These unused parts can strongly influence optimal division and should be directly addressed by the algorithm.

The problem of synchronization complexity (2) can strongly depend on the density of beings in different fragments of the environment. For example, in complex pedestrian-dynamics models, like the ones discussed in [10], conflict resolution requires additional processing and communication.

Allowing for area borders to split a potentially crowded fragment (like a narrow passage or a doorway) increases the volume of the exchanged information and burdens performance. This issue is addressed in the proposed solution by defining fragments of the environments that cannot be divided.

The typical architecture of modern computing hardware (3) provides many computing cores within a single computing node. Communication between the cores is usually far more efficient than communication over the network connecting the nodes. Therefore, the method should distinguish local and remote borders of the grid fragments and try to minimize the length of the more remote ones.

The main contribution of this paper is a new grid partitioning method that considers inaccessible and indivisible areas and provides division for local and remote parallelism. The method is based on a state-of-the-art graph partitioning method that has been extended and modified. A detailed description and justification of the introduced modifications is provided. The source code for the implementation of the proposed solution is available for download [18]. The method is evaluated using a set of complex scenarios.

2. Existing Graph Partitioning Methods

Graph partitioning is a branch of graph theory dedicated to the reduction of graphs into smaller subgraphs by dividing their nodes into smaller, mutually exclusive subgroups. The graph partitioning problems are NP-complete. There are many algorithms with which to approach this problem. Among them, the four main groups of graph partitioning methods can be distinguished:

- spectral partitioning,
- recursive partitioning,
- geometric partitioning, and
- multilevel partitioning.

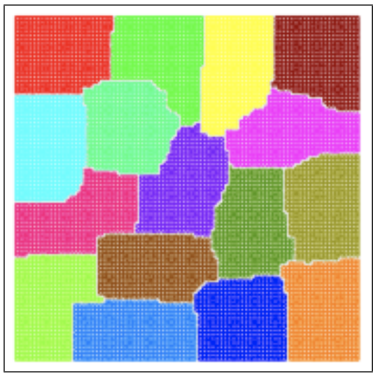
Spectral partitioning. Spectral graph partitioning methods are based on the selection of a subset of vertices that divides the graph into disjointed components. Such divisions seek to is to choose the smallest subset and divide the graph into subsets with an equal or close to equal number of vertices. Those methods use the Laplacian matrix representation of a graph. There many approaches for this method; for example, [11] presents the method using an algebraic approach to computing vertex separators, while [12] describes the spectral nested dissection algorithm (SND). These two algorithms use knowledge of the spectral properties of the Laplacian matrix to calculate the vertex separators in the graph, which determines the partitioning of the graph. These methods are expensive, however, due to the computation of the Fiedler vector, which selection criteria for vertices for the partitions. In [1], an attempt to shorten the execution time is proposed - the Fiedler vector is calculated using the multilevel spectral bisection algorithm (MSB). However, such improvements are still very computationally demanding.

Recursive partitioning. The recursive methods are generally simpler to implement, but do not work so well for more complex problems, mainly due to the greedy nature of these algorithms. The method presented in [2] assumes division of the graph into a number of areas equal to a power of two. This method can also divide the graph according to the computational capabilities of the individual processor cores. However, the underlying idea of dividing the graph into smaller and smaller parts precludes the algorithm from taking indivisible parts into account without significantly altering the idea underlying it.

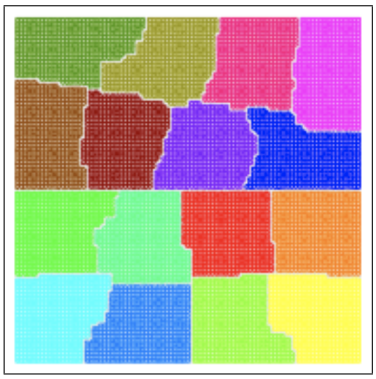
Geometric partitioning. Geometric methods use the geometric data layout of the graph for optimal partitioning. Their main advantage of these methods is their relatively short execution time, but they achieve lower-quality division results than the spectral methods. One of the best uses this method is presented in [6], where the authors describe an effective way of partitioning unstructured graph environment, which is useful for the FEMs (finite element methods) and FDMs (finite difference methods). This approach uses the geometry of a given graph to find a partitioning in linear time. They can be applied to graphs representing two- and three-dimensional grids. A characteristic feature of geometric methods, resulting from their random nature, is the need to execute the algorithm anywhere from 5 to 50 times to obtain results comparable with spectral methods, while still maintaining a shorter computing time. Geometric methods are applicable only when the coordinates of all vertices in the graph are available. For many problems (linear programming, VLSI), such coordinates are not available, limiting the applicability of this method. There are also algorithms that are able to calculate coordinates for graph vertices using spectral methods [3], but they significantly increase computing time.

Multilevel partitioning. A characteristic feature of the multilevel graph partitioning approach is the reduction of the graph size by combining the vertices and edges and dividing the reduced graph into partitions. The last phase restores the initial graph while preserving the obtained partitioning. The graph is often reduced until the number of vertices reaches the desired number of partitions [7]. The phase of restoring the graph to its initial size is accompanied by an algorithm aimed at improving the division [4]. Operating on a reduced graph has lower computational costs than other approaches. The methods in this class reduce the length of the boundary between partitions while maintaining the proportional sizes of the areas. Such methods were initially intended to reduce partitioning time at the expense of quality. The multilevel partitioning method is described in more detail in Section 3.

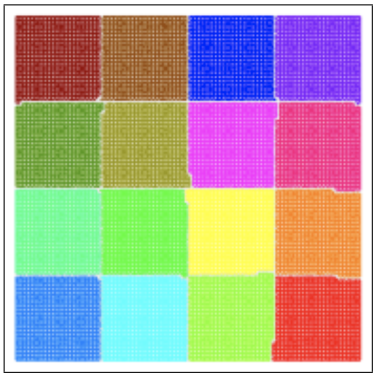
Recent research suggests that multilevel methods yield better results than spectral methods. Libraries such as Party [7], Metis [5], Jostle [17] are based on a multilevel partitioning scheme and currently give the best results in terms of partitioning quality.



(a) Jostle - 695.



(b) Metis - 688.



(c) Party - 615.

Figure 1. Partitioning a 100x100 grid into 16 areas; the edge-cut value is shown in the caption for each method. [7]

The authors of [7] compared these algorithms by partitioning a 100x100 grid area into 16 partitions, as demonstrated in Fig. 1. All of these methods reduce the graph; however, only Jostle and Party reduce the graph until they get the number of vertices equal to the required number of partitions. By comparing many different examples of grid partitioning problems, Party seems to outperform other algorithms, as shown in Fig. 1c. Moreover, Jostle has problems with partitioning the grid into appropriate areas. It can be observed that elongated, irregular partitions increase the edge-cut, even though boundaries between adjacent partitions are relatively straight.

None of the above-mentioned methods take the problem of indivisible areas and areas excluded from the calculations unused areas into account.

The multilevel graph partitioning methods [7] are the most promising and give the highest-quality results. As shown in Fig. 1, Party yielded the best results and as such, the presently proposed solution improves upon it. The main goal is to propose a solution that will be able to support both indivisible areas and excluded areas, while providing high-quality graph partitioning.

3. Grid Division Algorithm

The proposed multi-level scheme contains the following steps:

1) Building a graph from an image.
2) Coarsening the graph with the LAM matching algorithm.
3) Refining the partitioning and restoring the graph to its initial size.

It first reduces the graph, then applies partitioning and, then restores the graph to the initial size while propagating the partitions to a larger and larger graph until it reaches its original size. Local refinement is executed in between the restoration steps to increase efficiency. It balances partition sizes, reduces the edge-cut between partitions, and removes disconnected partitions.

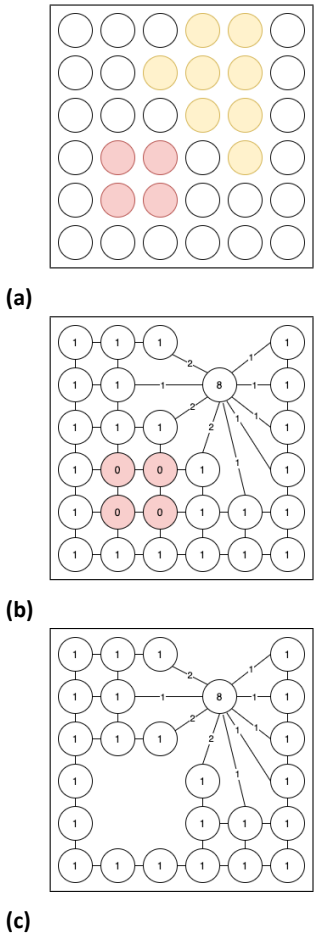
3.1. Building a graph from an image

An initial graph is built from an image. Each pixel on the image represents a vertex. Colors represent different graph area types; yellow areas are the indivisible areas, while red areas are the excluded areas. Normal areas are white. In the graph that is created, the normal area contains vertices with a weight of 1, while excluded areas are either removed from the graph or contain vertices with a weight of 0. Every indivisible area is mapped into a single vertex with a weight equal to the sum of the weights of the vertices in that area (Fig. 2). Thus, very time a set of edges is replaced by a single edge, the weight of the single edge is a sum of the replaced vertices' weights.

3.2. Coarsening phase

To create a smaller graph, a heuristic of a matching algorithm is introduced. In our case, it is the LAM [13] matching algorithm, which is executed until the number of vertices is equal to the desired number of partitions. It starts from a randomly chosen edge and checks adjacent edges. As long as it manages to find adjacent edges with a higher weight, the algorithm switches onto them and repeats the procedure until it finds the edge with the highest weight. Vertices at the end of this edge will be matched only if they satisfy a certain condition; in the Party [7] implementation, two vertices, a and b, with weights w_a and w_b, could be matched only if their combined weight did not exceed double the lowest weight (w_{lowest}) plus the heaviest weight ($w_{highest}$) that occured in the whole graph (Equation 1).

$$w_a + w_b \leq w_{highest} + 2 \cdot w_{lowest} \quad (1)$$



(a)

(b)

(c)

Figure 2. Excluded areas are either removed from the graph (c), or their weights are set to 0 (b).

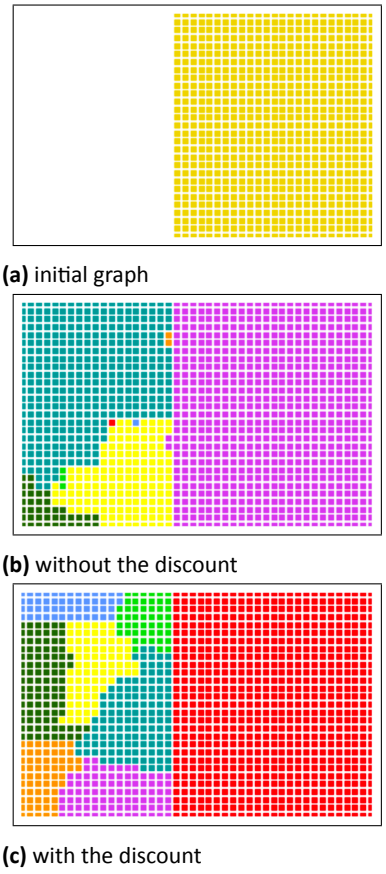
This condition allows for balanced vertex matching, avoiding vertices with very high weights along with vertices with relatively small weights. The weight of the new vertex is the sum of the weights of the matched vertices.

The first change introduced by us to the LAM algorithm is an extended condition for matching vertex a with vertex b (3):

$$discount = \frac{t}{T \cdot \frac{w_{highest}}{w_{lowest}+1} \cdot \log(num\_of\_partitions) + 1} \quad (2)$$

$$w_a + w_b \leq discount \cdot (w_{highest} + 2 \cdot w_{lowest}) \quad (3)$$

If $discount > 1$, it is assigned a value of 1. T is the expected number of LAM algorithm executions. It is counted as the number of times the number of graph vertices has to be divided by 2 to get a number of vertices that is less or equal to the number of partitions required to achieve it. t is the current number of LAM algorithm executions. The condition established by authors of the Party library assumes a balanced formation for the vertices' matchings from the very beginning of the algorithm execution.



(a) initial graph

(b) without the discount

(c) with the discount

Figure 3. Effects of the partitioning with and without the discount.

In our case, however, this assumption is disturbed by vertices made from indivisible areas, which may have very high weights at the beginning of the algorithm execution. As a result, without our modification, the algorithm gives poor-quality results (Fig. 3).

Adding the discount to the matching conditions enhances the condition and creates more balanced vertex matchings, especially in the initial algorithm executions. The later the iteration, the more similar it becomes to the original condition.

The other modification made to the LAM algorithm is the removal of the R set, which contains edges that are about to be removed. It is not useful for the graph shrinking application, so its maintenance can be skipped.

3.3. Local refinements and graph restoration

The second part of the graph partitioning algorithm is the refinement and restoration phase. It takes as an argument a shrunken graph and iteratively restores it to the initial size. After the graph is about 90% restored, the refinement procedure starts alongside the restoration, which aims to reduce the edge-cut and balance the sizes of the partitions. For the refinement phase, the algorithm based on the Helpful Set heuristic [15] was used.

The implemented Helpful Set heuristics (Code 1) start from the initial partitioning and reduces the edge-cut using locally performed changes.

As an example, let us take partitions V_1 and V_2. It starts from the initial set `k-helpful`, which is a subset of V_1 or V_2 that reduces the edge-cut by k if it is moved to the other partition. Let us assume that `k-helpful` was found in V_1. Then it is moved to V_2. In the next step, the algorithm starts to look for a `balancing set` in the V_2 partition. This set has to balance the partitions (reducing sizes to be similar to the initial ones), and it can increase the edge-cut maximally by $(k-1)$ edges. If a balancing set is found, then it is moved to V_1. This results in an edge-cut decrease of at least 1. Vertices that build a `helpful set` and a `balancing set` are added greedily and are stored in a sorted priority queue according to their value. The most important part of the algorithm is to figure out the l helpfulness values for the sets. If l is too small, then good sets cannot be found; if l is too big, then the execution time is longer, but the found sets are not necessarily better. To compute l value, Party uses an `adaptive limitation` technique. It contains two l limits - a separate one for each of the sets.

```
1  HelpfulSet(A, B)
2    IF cut_size_{A-B} < 0
3      RETURN;
4    l_A ← l_B ← cut/2;      /* Initialize the limits */
5    s_max = (|A| + |B|)/2) · 0.2;    /* Initialize max size of HS
       */
6    WHILE l_A + l_B ≥ 1
7      IF l_A = 0 OR 2 · l_A ≤ l_B  /* Choose better partition */
8        Swap(A, B);
9      S_A = BuildHS(A, l_A, s_max);
10     IF h(S_A) ≤ l_A /* If the h(S_A) is smaller than wanted
         ...
11       l_A ← b(S_A);  ... adjust the limit for the next
        search */
12       IF l_B > h(S_A)
13         S_B = BuildHS(B, l_B, s_max);
14         IF h(S_B) ≥ h(S_A) /* Better partition is called A
         ...
15           Swap(A, B);
16         ELSE
17           l_B ← b(S_B); ... and the other limit is
        reduced */
18       UndoBuild(S_B);
19     IF len(S_A) == 0
20       l_A = 0;
21       CONTINUE;
22     l_A ← min(l_A, h(S_A));    /* Adjust the limit */
23     MoveSet(S_A)    /* Move the helpful set */
24     min, max ← DetermineMaxAndMin(w(S_A));
25     S_B = BuildBS(B, 1 − h(S_A), min, max);
26     IF w_l ≤ w(S_B) ≤ w_h and h(S_B) > −h(S_A)  /* Checking the
         BS */
27       MoveSet(S_B);        /* Yes: Move the BS */
28       l_A ← l_A + log(l_A);       /* Increase the limits */
29       l_B ← l_B + 1;
30     ELSE
31       UndoBuild(S_B);  /* No: Undo the build operation
        and
32       UndoMove(S_A);     the movement of the helpful set
         */
33       l_A ← l_A/4;          /* Reduce the limits */
34       l_B ← l_B/2;
35   IF cut_size_{A-B} > 0.1 · longer_edge_of_a_grid
36     Balance(A, B);
```

Code 1. Modified Helpful-Set algorithm.

Several modifications have been introduced to the original algorithm in order to adjust it for the considered requirements.

The first concerns the initialization of s_{max}. Originally its size was set to 128, while in the proposed method it is dependent on the sizes of the partitions (line 5). Used for this purpose, the 0.2 factor could be changed - but, according to the experiments, it should not be more than 0.4, to prevent the helpful set from getting too big. The condition for the WHILE loop was weakened from ≥ 0 to ≥ 1 to decrease number of the algorithm executions. The additional repetitions did not improve the partitioning, but worsen execution time. In Party's solution, during the search, the helpfulness of the helpful set is $> -d/2$, and its weight cannot exceed s_{max}. d is an average vertex's degree in the graph. In our case, $d = 4$. This rule happened not to work, however. The algorithm is greedy and often builds sets, which fulfilled the s_{max} condition and with $-2 \leq helpfulness \leq 0$. But when, in the original code, it was checked whether $helpfulness < 0$, and if $true$, the helpful set search had to start once again. Because of this, the algorithm was running unsuccessfully for multiple times. The solution to this issue was a change in the instruction for the condition for the size of the S_a (line 19), allowing for the building of helpful sets with a helpful value ≥ 0 (line 9 and 13). The helpful-set algorithm has a mechanism to build a helpful set from a more promising partition (line 7). After the helpful set is found - which might happen after further adjustments to the l value - the balancing procedure starts, and a $[min, max]$ range is computed. This is the minimum and a maximum weight of the balancing set. max is a little bigger than size of the helpful set and min is a little smaller. During the experiments, the algorithm was usually reaching max size - and for some reason, for each pair of refined partitions, the same one was usually chosen to build the helpful set from; the same was true for balancing set. As a result, one partition was always growing at the expense of the other. As a result, the following change to computing the min and max values was introduced:

```
1  DetermineMaxAndMin(w(S_A))
2    rand = Rand(0, 1)    /* draws either 0 or 1 */
3    IF rand == 1
4      min = |w(S_A) − 0.1 · w(S_A)|
5      max = |w(S_A) + 0.1 · w(S_A)|
6    ELSE
7      min = |w(S_A) − 0.2 · w(S_A)|
8      max = |w(S_A) − 0.1 · w(S_A)|
9    ENDIF
10   RETURN min, max
```

Code 2. Modified min and max computing.

If a balancing set's weight is in the range, it is moved to the other partition. The next modification was the condition for executing the `Balance` procedure, which prevents the formation of scattered areas. The `Balance` procedure greedily chooses vertices with the highest helpfulness. These are always vertices from a bigger partition. It does not move the whole weight difference between the partitions; instead on each execution, it moves only a percentage of this difference. In our case, it was around 10% of the weight difference.

Making local refinements can cause scattered partitions to appear. A scattered partition is a partition that is divided into unconnected sub-areas. This can be caused by indivisible areas, which are usually big chunks of a graph that can be moved in just one step of the refinement algorithm. A simple solution is to find scattered partitions; identify the main, biggest subarea; and join the other parts with their adjacent partitions. This procedure is executed after each refinement.

3.4. Partitioning into m partitions

Considering modern parallel hardware architectures - which are composed of multiple nodes, each equipped with multiple cores - requires taking into account differences in local and remote communication. Partitioning of a grid for parallel processing on m nodes with k cores each should provide m subgrids, each divided into k partitions. The simple approach of dividing the grid into m parts first and then splitting each part later cannot be used here because of the indivisible parts of the grid; a single indivisible part might become one of the m parts, preventing its further division it into k parts. Therefore, it is proposed to merge the $m \cdot k$ division into m parts instead.

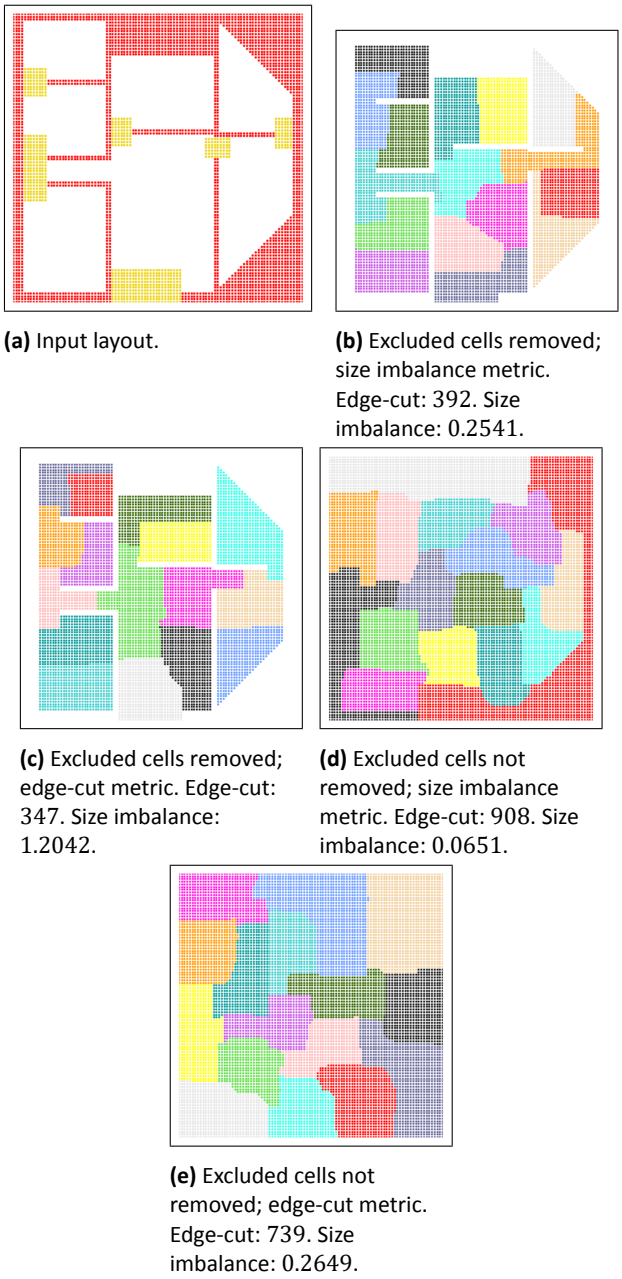
This part is handled by the LAM weighted matching with an additional greedy algorithm. First, after the partitioning into $m \cdot k$ parts, the graph is reduced to m vertices using the LAM matching algorithm. After the reduction, each vertex is assigned to a new partition. Next, the graph is restored to $m \cdot k$ partitions, but the m partitions yielded by LAM are kept. At this point, if not all of the m partitions have the same amount of subpartitions, the greedy method is used: first, the adjacent partitions are balanced. Next, if there are no more imbalanced adjacent partitions, but there are still some imbalances, the algorithm chooses the two partitions with the highest and lowest number of vertices and balances them greedily. This process is repeated until all the partitions have a proper size.

4. Experimental Evaluation

This section shows the results of experiments performed using the presented method. In each experiment type, the same input was processed 100 times, followed by selection of the best result (an approach used also in [7]). Two metrics were used to determine the output quality:

- *Edge-cut*: the length of the border between partitions.
- *Size imbalance*: the inequality of the resulting partition sizes. The excluded areas are not counted in the size of the area. This inequality is expressed as the standard deviation of all percentage partition shares, to allow the metrics to be compared regardless of the total grid size.

4.1. Partitioning into $m.k$ areas



(a) Input layout.

(b) Excluded cells removed; size imbalance metric. Edge-cut: 392. Size imbalance: 0.2541.

(c) Excluded cells removed; edge-cut metric. Edge-cut: 347. Size imbalance: 1.2042.

(d) Excluded cells not removed; size imbalance metric. Edge-cut: 908. Size imbalance: 0.0651.

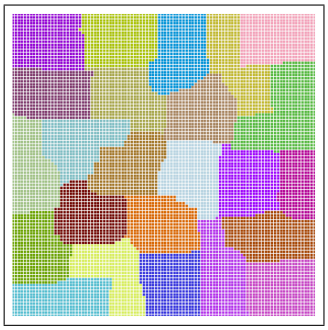
(e) Excluded cells not removed; edge-cut metric. Edge-cut: 739. Size imbalance: 0.2649.

Figure 4. Results of a building floor - plan partitioning.

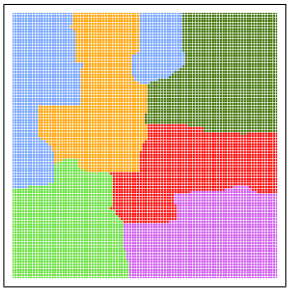
In the first experiment, the grid is a building floor plan, with several rooms connected via hallways (see Fig. 4a). The grid contains both excluded areas and indivisible areas, which are represented as red and yellow pixels, respectively. Walls and the area outside of the building are marked as excluded areas, while doorways are marked as indivisible. The size of the grid is 100×100 cells, and divided into 16 partitions.

The experiment was executed in four variants, determined by two parameters:

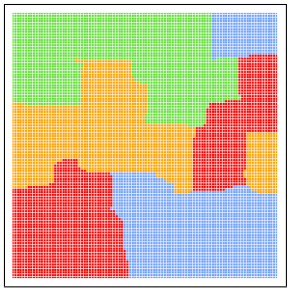
- Excluded areas were either assigned a zero weight or removed from the graph.
- The metric used to determine the best result was either edge-cut or size imbalance.



(a) Partitioned input. Edge-cut: 925. Size imbalance: 0.4535.



(b) Merged into 6 partitions. Edge-cut: 667. Size imbalance: 1.0459.



(c) Merged into 4 partitions. Edge-cut: 624. Size imbalance: 1.0872.

Figure 5. Synthetic example of partitioning.

Fig. 4 shows the input and the results of the partitioning. In each case, the input constraints were preserved, i.e., the excluded area did not cause large imbalances in area sizes, and indivisible areas remained undivided.

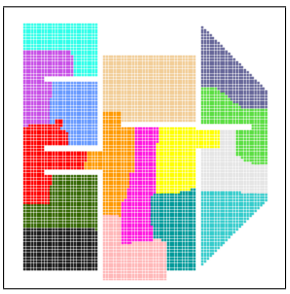
It is easy to observe that removing the excluded cells yielded better edge-cut metric values (see Fig. 4b and Fig. 4c). However, it is also important to notice that a lot of connections are removed from the graph, which significantly reduces the ability of different areas to be connected. At the same time, the borders that would touch excluded areas on one or both sides might have no communication in the simulation system, as those areas do not take active part in the simulation by definition. Another interesting piece of information is that preserving the excluded cells yielded much better size - imbalance metric values (see Fig. 4d and Fig. 4e). This might be caused by the ability to connect remote areas via a patch of zero-weight cells that do not cause the area to stop expanding.

In opposition to the usual LAM algorithm behavior, which tends to yield partitions with smaller perimeter to area ratios, the use of zero-weight cells caused more elongated shapes to emerge. One consequence is that a non-excluded area taking part in the simulation might become disconnected within a single partition - this is easiest to observe with the red partition in Fig. 4d.

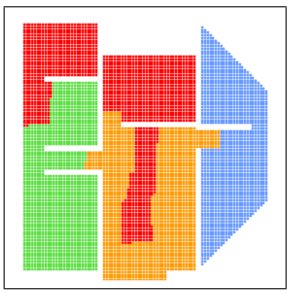
The conclusions from this experiment are evidence in favor of removing the excluded areas from the graph entirely. It leads to better division of the actual, used parts of the grid.

4.2. Partitioning mk areas into m areas

In the second experiment, the given partitioning into $m \cdot k$ areas was partitioned into m areas. This operation is strictly connected to the underlying idea of partitioning the grid for paralellization of computations. In the scenario in which the simulation will be executed using m nodes with k cores each, the first step creates the partitions for each core, while this step assigns them to the nodes.



(a) Partitioned input.



(b) Merged output.

Figure 6. Plan of building floor example of partitioning. Edge-cut: 200. Size imbalance: 1.6641.

This experiment was performed using two example scenarios. The first one is a synthetic example, in which a grid of size 100×100 was already partitioned into 24 areas. The experiment consists of two executions of partitioning those areas into 6 and 4 larger areas (thus $m = 6$, $k = 4$ and $m = 4$, $k = 6$). Fig. 5 shows the results of this partitioning. The input partitioning (see Fig. 5a) has a better size-imbalance metric value, but a much worse edge-cut metric value than both outputs. This is to be expected, as division of any area creates a new border, and thus the edge-cut should be larger for any partitioning with more areas. At the same time, any inequality in area sizes might be magnified by combining them into larger groups. Both outputs show some disconnection in the final partitions, which contributes towards larger edge-cut, but the metric value is improved in both cases. As the interpretation of this metric for merged output is the amount of inter-node communication, these results are very promising.

The second scenario is a building floor plan, based on the same layout as shown in the first experiment (see Fig. 4a). The size of the grid is 100×100, $m = 4$, $k = 4$. Fig. 6 shows the results of this experiment. The most important quality of the resulting division is that it consists of 4 areas of almost identical sizes, which can be further divided into 16 smaller areas, enforcing the rules of the excluded and indivisible cells in both cases. Not all areas are continuous - the red area in Fig. 6b is divided into three parts. However, it can be easily shown that there is no partitioning of the input that avoids fragmentation in the resulting partitions. The red area in Fig. 6a isolates two branches of the floor plan - one containing three partitions, and the other containing two. Merging the red area with any of the groups will leave the other unable to connect to sufficient number of areas to form a 4-area group.

5. Summary of Results

The proposed algorithm is probabilistic, and as such it is suitable for parallel processing to produce better results. Repeating the processing on the same input can be easily parallelized. As the partitioning itself implies that the simulation is intended to be executed on multiple cores, using those resources as early as during partitioning seems advisable. It is also possible to perform more repetitions of the first stage followed by multiple repetitions of the second phase for each result, as the second phase is much less computationally demanding. However, the quality of the initial partitioning will have a large impact on the second phase. If area sizes are balanced, the merging results are likely to be balanced as well. Therefore, it is advisable to perform more repetitions of the first stage and a similar number of repetitions of the second phase for each result, as opposed to generating fewer results from the initial phase and more repetitions of the second one.

Unfortunately, it is also possible to obtain partitioning containing unconnected areas within a single node partition. However, for some inputs such behavior might be unavoidable. The algorithm presented here correctly preserves the excluded and indivisible area constraints, while at the same time maintaining the balanced - area division and minimizing the edge-cut size. Thi all leads to less time spent on waiting for other cores to finish computations and lower communication overhead.

The proposed method for partitioning grid-based simulation models provides satisfactory results. It is designed as an extension of an advanced and efficient graph - partitioning algorithm; therefore, it provides similar partitioning quality for simple environment shapes. It is extended with proper handling of the complex structure of real environment shapes and the possibility of defining indivisible areas. The division can also be optimized for local and remote parallelism, taking into account the architecture of computing nodes and cores. As a result, the proposed method can provide initial divisions of the simulation model that are better tailored to the needs of real-world simulations.

Further optimization of the implementation of the algorithm [18] is planned in order to reduce the time required to partition a given grid. Another modification of the method under consideration would allow defining different weights of the grid cells. This could reflect real simulation costs more accurately, providing a better - balanced workload.

AUTHORS
Jakub Ziarko – Poland, e-mail: kuba.ziarko@gmail.com.
Mateusz Najdek – AGH University of Krakow, A. Mickiewicza Av. 30, 30-059 Kraków, Poland, e-mail: najdek@agh.edu.pl.
Wojciech Turek* – AGH University of Krakow, A. Mickiewicza Av. 30, 30-059 Kraków, Poland, https://skos.agh.edu.pl/osoba/wojciech-turek-6937.html, e-mail: wojciech.turek@agh.edu.pl.
*Corresponding author

ACKNOWLEDGEMENTS

The research presented in this paper was funded by the National Science Centre, Poland, under the grant no. 2019/35/O/ST6/01806.

References

[1] S. T. Barnard and H. D. Simon, "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems", *Concurrency: Practice and Experience*, vol. 6, no. 2, 1994, pp. 101–117.

[2] M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors", *IEEE Transactions on Computers*, vol. C-36, 1987.

[3] T. Chan, J. R. Gilbert, and S.-H. Teng, *Geometric spectral partitioning*, Citeseer, 1994.

[4] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions". In: *Proceedings of the 19th Design Automation Conference*, 1982, pp. 175–181.

[5] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs", *SIAM Journal on scientific Computing*, vol. 20, no. 1, 1998, pp. 359–392.

[6] G. L. Miller, S. Teng, W. Thurston, and S. A. Vavasis. "Automatic mesh partitioning". In: A. George, J. Gilbert, and J. Liu, eds., *Graphs Theory and Sparse Matrix Computation*, The IMA Volumes in Mathematics and its Application, pp. 57–84. Springer-Verlag, 1993. Vol 56.

[7] B. Monien and S. Schamberger, "Graph partitioning with the party library: Helpful-sets in practice". In: *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004)*, 2004, pp. 198–205.

[8] K. Nagel, "Cellular automata models for transportation applications". In: S. Bandini, B. Chopard, and M. Tomassini, eds., *Cellular Automata*, Berlin, Heidelberg, 2002, pp. 20–31.

[9] M. Paciorek, A. Bogacz, and W. Turek, "Scalable signal-based simulation of autonomous beings in complex environments". In: *International Conference on Computational Science (ICCSA 2020)*, 2020, pp. 144–157.

[10] M. Paciorek and W. Turek, "Agent-based modeling of social phenomena for high performance distributed simulations". In: *International Conference on Computational Science (ICCSA 2021)*, 2021, pp. 412–425.

[11] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs", *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 3, 1990, pp. 430–452.

[12] A. Pothen, H. D. Simon, L. Wang, and S. T. Barnard, "Towards a fast implementation of spectral nested dissection". In: *Supercomputing'92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992, pp. 42–51.

[13] R. Preis, "Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs". In: *Annual Symposium on Theoretical Aspects of Computer Science*, 1999, pp. 259–269.

[14] S. F. Railsback and V. Grimm, *Agent-based and individual-based modeling: a practical introduction*, Princeton University Press, 2019.

[15] S. Schamberger, "Improvements to the helpful-set algorithm and a new evaluation scheme for graph-partitioners". In: V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, eds., *Computational Science and Its Applications (ICCSA 2003)*, Berlin, Heidelberg, 2003, pp. 49–53.

[16] W. Turek, L. Siwik, and A. Byrski, "Leveraging rapid simulation and analysis of large urban road systems on hpc", *Transportation Research Part C: Emerging Technologies*, vol. 87, 2018, pp. 46–57.

[17] C. Walshaw and M. Cross, "Mesh partitioning: A multilevel balancing and refinement algorithm", *SIAM Journal on Scientific Computing*, vol. 22, 2004.

[18] J. Ziarko. "Grid partitioning source code". https://github.com/WKD622/grid-partitioning/tree/master. [Online; accessed 01-20-2024].