Pandora
MPI Multi Node scheduler

5.10.2020

CASE – Germán Navarro Jiménez

v0.5

Contents

1. Motivation

1.1. What is Pandora?

Pandora is a 2D Simulator for Agent-Based Modelling (ABM) written in C++. It combines 2 kind of objects: agents and grids (also named rasters) which represent the environment. Agents can interact with each other or with the environment. Rasters can depict whatever static or mutable resource, landform, geographical Points of Interest (POIs), etc.

The modeler can write its models directly in C++ on top of the Pandora's engine, which provides a number of access methods to define own rasters, types of agents and actions to be performed by these agents. A NetLogo-like interface is being developed in order to let the modeler to build a rapid prototype using the well-known syntax of NetLogo.

1.2. The problem

By definition, an ABM gathers the following properties:

1. The **level of interaction** between agents or agent-environment is **very high**. A traffic model, for instance, can be considered an ABM. But in our mind and in a general case, when we talk about ABM, it is common that agents are not strictly confined to roads, but rather they move quite freely.
2. It is a **chaotic model** in which a single modification in the initial conditions heavily affects the final outcome.
3. Because of that, they lead to **unpredictable dynamics** and it is common that agents are not uniformly distributed along all the space, i.e. **unbalances can be somehow frequent**.

Of course, when we consider toy models (~ <10.000 agents or grids < 1000^2 cells), performance problems are not usually an issue. From here on, single core runnings of a model can be unsatisfactory from a time perspective. It arises the necessity of being more efficient. Assuming that the internal data structures and methods used in the Pandora's engine are already optimal, what we can think about is parallelization.

But this parallelization task is not trivial whatsoever, mainly due to the 1st and 3rd points stated above: the high level of interaction between entities and the potential unbalances.

As for the first point, GPU-based implementation has been discarded so far, since GPUs are basically great when performing relatively simple operations with little or no dependencies among processes. So in a first stage of the development, we will bet for a CPU-based partitioning and synchronization, while for futures stages agents processing could be delegated to GPUs.

To parallelize by means of CPU we have 2 main alternatives:

1. **Shared memory**: agents would be executed by different processes accessing simultaneously to the same memory space. To apply this approach, the modeler needs to take it into account. It is not possible for the engine (i.e. in an automatic way) to know whether a race condition appears when accessing a shared variable. This operation corresponds to the modeler, which should use a variables temporary blocking system. So, **the model would be parallelization dependent**.
2. **Message passing**: agents are computed by different processes with local memories. This means that synchronization among nodes is necessary. A natural dimension for memory partitioning is the space. Since **this approach can be implemented in an almost fully transparent way for the modeler** as we will explain, we think it is the most suitable for our current purpose.

1.3. Current expectations

- In order to exploit the full potential of the message passing paradigm it is required to let the processes to run the agents in parallel asynchronously. Of course, synchronization will be necessary, but they should be minimized and only performed when strictly needed.
- For controlling the chaos, we need to set an initial seed for the RNG, as common. Besides, to produce the same results in identical simulations, we would need to maintain the coherence between agents. In other words, at a certain point in the simulation, for 2 different replications with exactly the same parameters and the same distribution, an agent should see the same state around itself, even with asynchronization (i.e. agents could arrive to nodes in different order for different simulations).
- As the performed partition will be space-based, the dependencies will arise according to space issues (agents moving or raster cells changing). Suitable models will be those for which the modeler can stablish a certain threshold for the maximum agent's visibility/movement/radius of action. If, for a given agent, any cell in the grid can be potentially updated (or it can be teleported), the model under consideration will not be appropriate for the current Pandora's implementation.
- The resulting partitioning should be as balanced among nodes as possible.

2. Initialization & distribution

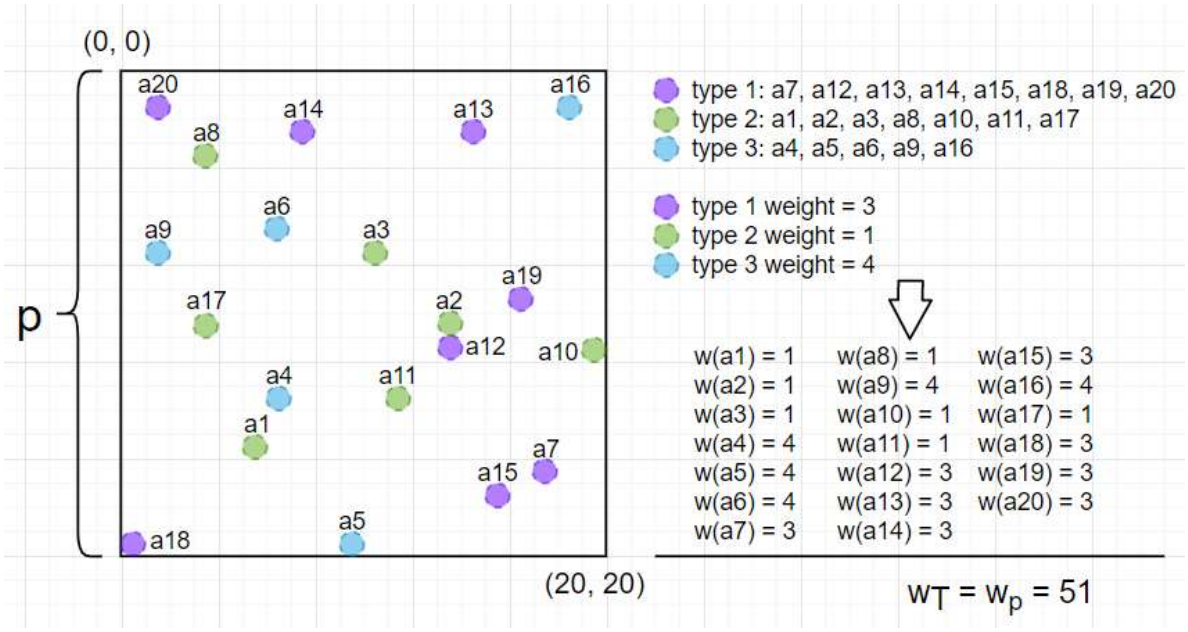
Rasters creation method (*virtual World::createRasters()*) are executed by all nodes in the MPI network. Since the initial seed is exactly the same for every node, this piece of code will generate exactly the same rasters if called before using any other random methods.

Agents creation method is only executed by the master node (0 by default) and spreaded out to the rest of the nodes.

2.1. Space partitioning: an intuitive explanation

The space is divided according to the weights of the agents. This weight may be arbitrarily defined by the modeler (e.g.: the same for all the agents, specific for type of agent, defined for each one, randomly generated, etc).

The space partitioning process is performed recursively for each subpartition in class **LoadBalanceTree.cxx**, conceptually generating a binary tree. Although the implemented algorithm is performed in a recursive way, we can also understand it as it was executed iteratively. In this way, the following example shows the approach for 4 MPI nodes and 20 randomly generated agents. Weights are defined here by type of agent.



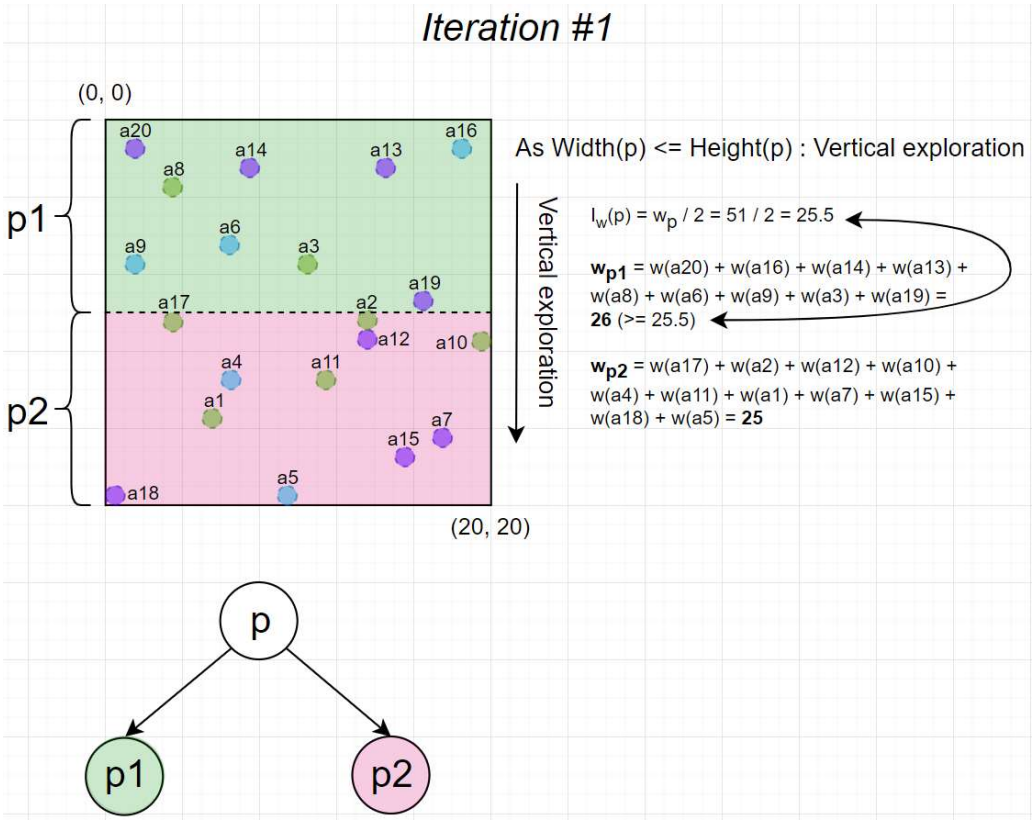
When partitioning, a particular kind of space exploration is performed:

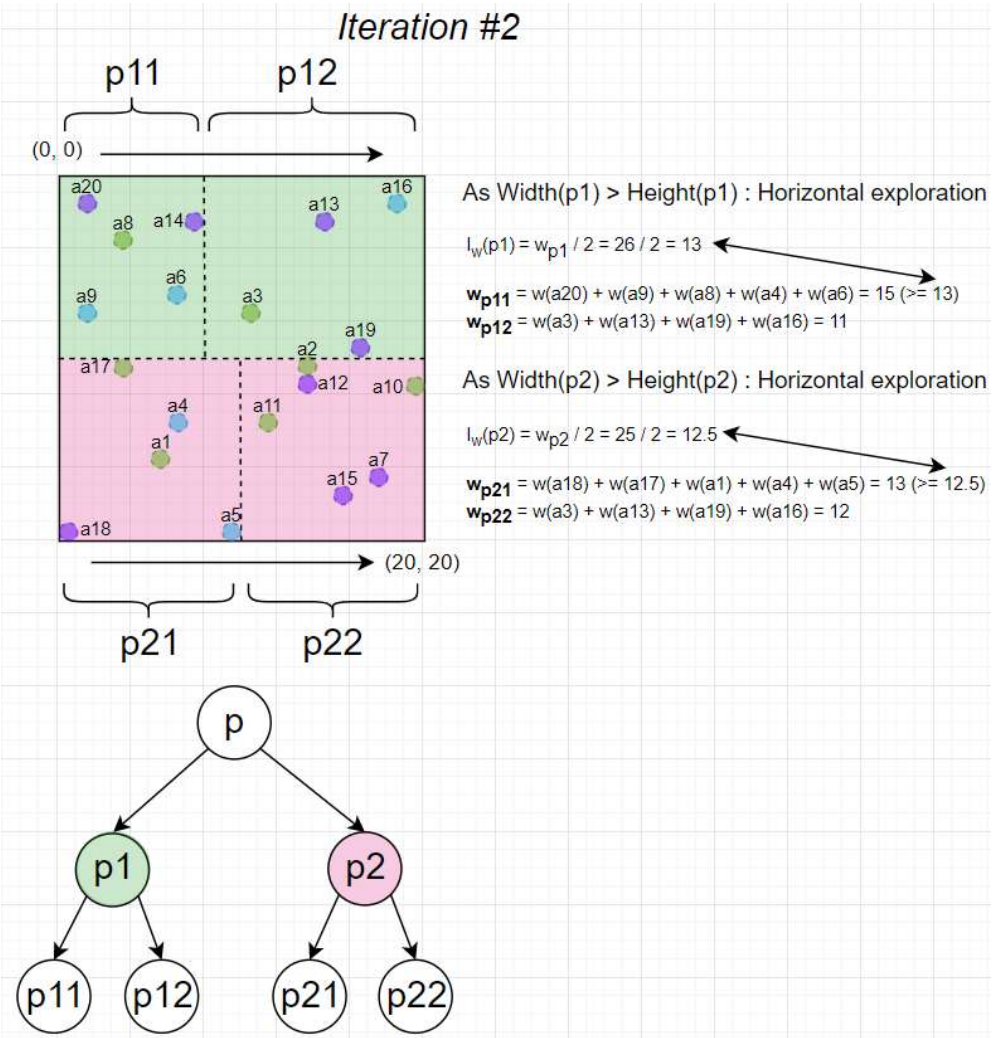
1. Vertical exploration (horizontal partitioning): the space is traversed by rows and columns, i.e. all agents in row *i* are considered before considering those on row *i+1*.
2. Horizontal exploration (vertical partitioning): the space is traversed by columns and rows, i.e. all agents in column *j* are considered before considering those on column *j+1*.

In this way, more square-like partitions are generated, which heuristically generates a lesser amount of overlap surfaces (see section 3.1. Maintaining space continuum).

In the graphics, $I_w(p)$ represents the ideal weight by partition. In other words, this is the value of reference to get a balanced partitioning in each iteration. When the exploration is performed, the weights of the encountered agents are summed up. When this total is greater or equal than the value of reference $I_w(p)$ then it stops. The remaining agents in the current row or column are considered for first partition; all the rest of the agents are considered to form the second partition.

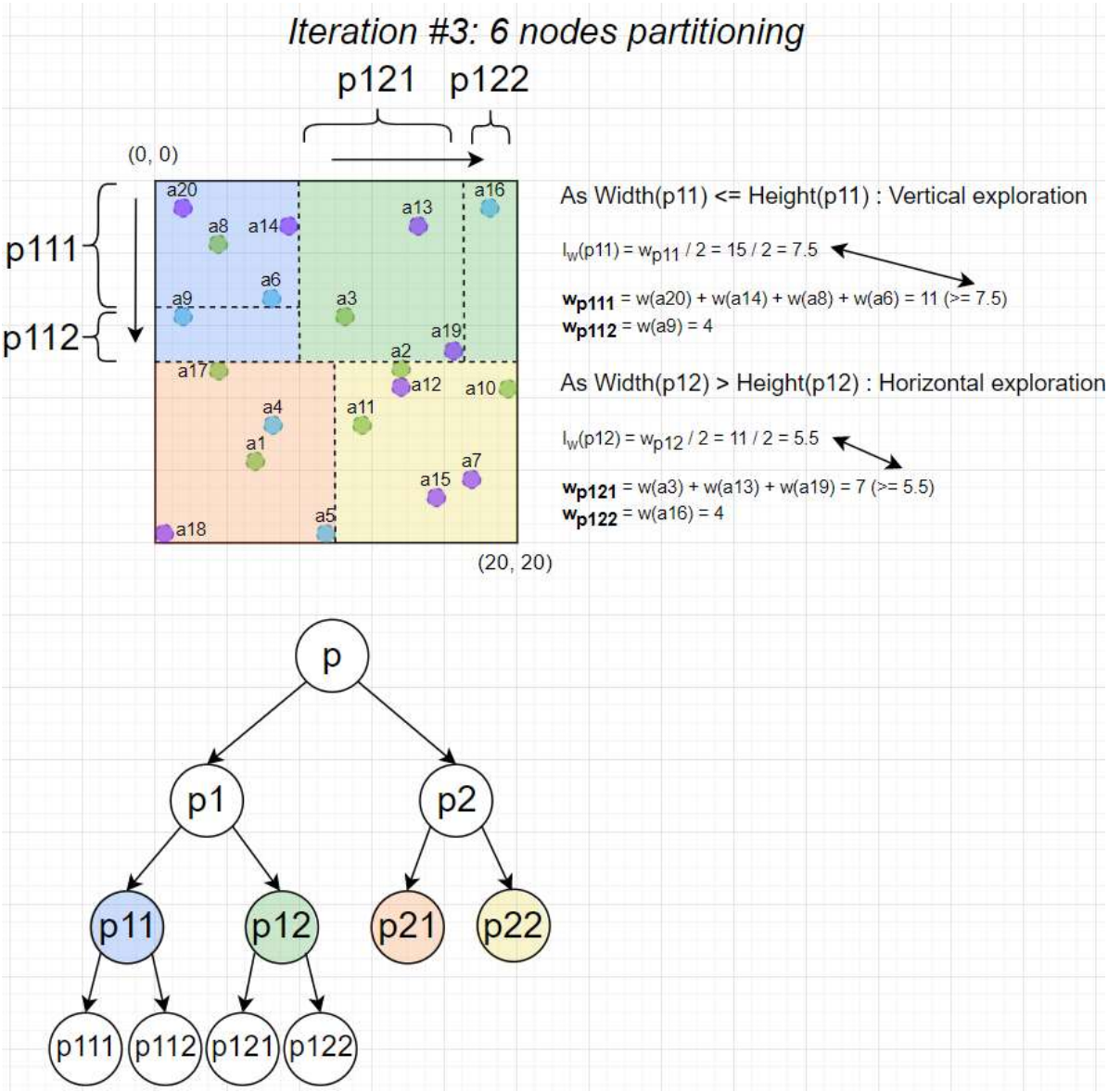
The following 2 iterations shows the process for a total of 4 partitions:





From here, we could keep dividing beyond 4. We could even state a non-multiple-of-2 number of MPI processes, but it is not recommended, since it would result in an unbalanced tree, where processing times will affect all the other nodes when synchronize with each other.

Following with the same example as illustration:



As seen in the figure, the partitions are divided again until a total amount of 6 is reached. In this particular example, we can see that this generates high unbalanced partitions (e.g. partition p112 (with weight = 4) and partition p22 (with weight = 12)). In a case like this, maybe it wouldn't be worth to split the space in 6 instead of just 4 partitions, for 2 main reasons combined:

1. The load unbalancing among partitions could lead into a situation in which some nodes are still processing agents, while some others have already sent their updates and are waiting for the rest of the nodes to finish the current step. This means a waste of resources.
2. The synchronization overhead would be surely higher because there is a bigger amount of overlap cells.

2.2. Pseudocode

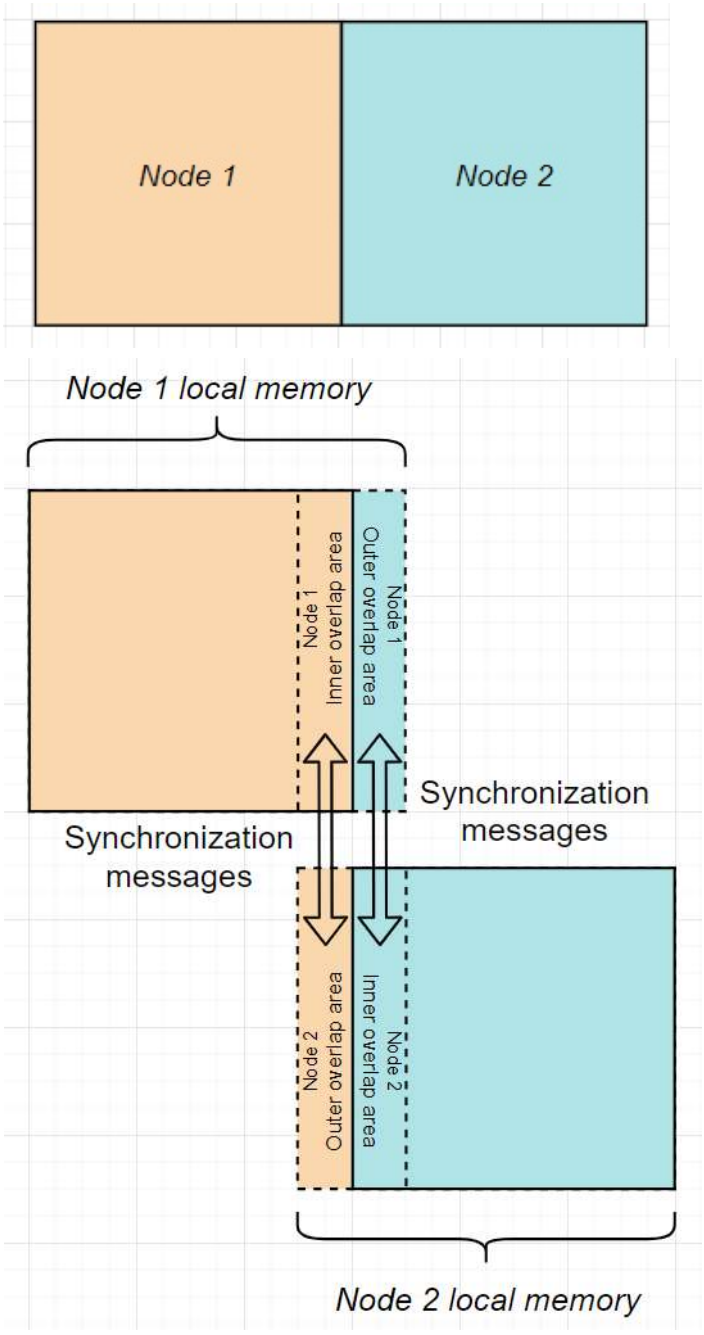
Notice that variables starting with an "_" are class members in C++ (global visibility).

```
stopProcreating(currentHeight):
   if currentHeight == 0: return true
   if numberOfLeafs(_root) == _numberOfPartitions: return true

   if not isPowerOf2(_numberOfPartitions):
       numberOfNodesNeededAtLowestLevel = 2 · (_numberOfPartitions − 2^⌈log2 _numberOfPartitions⌉)
       lowestLevel = ⌈log2 _numberOfPartitions⌉
       isLowestLevelAlreadyFull = numberOfNodesNeededAtLowestLevel == numberOfNodesAtDepth(_root,
lowestLevel)

       if currentHeight == 1 and isLowestLevelAlreadyFull: return true

   return false

exploreHorizontallyAndKeepDividing(node, totalWeight, currentHeight):
   leftChildWeight = 0

   for i in (node.value.left, node.value.right):
       for j in (node.value.top, node.value.bottom):
           leftChildWeight += getAgentsWeightInCell(j, i)

       if leftChildWeight >= totalWeight/2:
           leftPartition = Rectangle(node.value.left, node.value.top, i, node.value.bottom)
           rightPartition = Rectangle(i + 1, node.value.top, node.value.right, node.value.bottom)

           node.left = insertPartitionInLeftNode(leftPartition, node)
           node.right = insertPartitionInRightNode(rightPartition, node)

           divideSpaceRecursively(node.left, leftChildWeight, currentHeight - 1)
           divideSpaceRecursively(node.right, totalWeight - leftChildWeight, currentHeight - 1)

           break

exploreVerticallyAndKeepDividing(node, totalWeight, currentHeight):
   leftChildWeight = 0

   for i in (node.value.top, node.value.bottom):
       for j in (node.value.left, node.value.right):
           leftChildWeight += getAgentsWeightInCell(i, j)

       if leftChildWeight >= totalWeight/2:
           topPartition = Rectangle(node.value.left, node.value.top, node.value.right, i)
           bottomPartition = Rectangle(node.value.left, i + 1, node.value.right, node.value.bottom)

           node.left = insertPartitionInLeftNode(topPartition, node)
           node.right = insertPartitionInRightNode(bottomPartition, node)

           divideSpaceRecursively(node.left, leftChildWeight, currentHeight - 1)
           divideSpaceRecursively(node.right, totalWeight - leftChildWeight, currentHeight - 1)

           break

divideSpaceRecursively(node, totalWeight, currentHeight):
   if not stopProcreating(currentHeight):
       if width(node.value) > height(node.value):
           exploreHorizontallyAndKeepDividing(node, totalWeight, currentHeight)
       else:
           exploreVerticallyAndKeepDividing(node, totalWeight, currentHeight)

main():
   _root.value = Rectangle(left = 0, top = 0, right = width(p), bottom = height(p))
   _root.left = NULL
   _root.right = NULL
   totalWeight = getAllAgentsWeight()
   optimalHeight = ⌈log2 _numberOfPartitions⌉

   divideSpaceRecursively(_root, totalWeight, optimalHeight)
```

3. Synchronization

3.1. Maintaining space continuum

In order to keep continuity between nodes, a replicated area must be defined. This area is called the overlap area, since it represents exactly the same area but in the different local memories of the processes. It is maintained synchronized among neighbouring nodes.



Raster cells changed in the overlap area or agents moved to/off it are sent by its modifying node to all its neighbouring nodes. In bibliography, agents in these overlap areas are called *ghost agents*. By the nature of this approach, agent collisions arise in these areas (see section 3.2.1. Agent collisions)

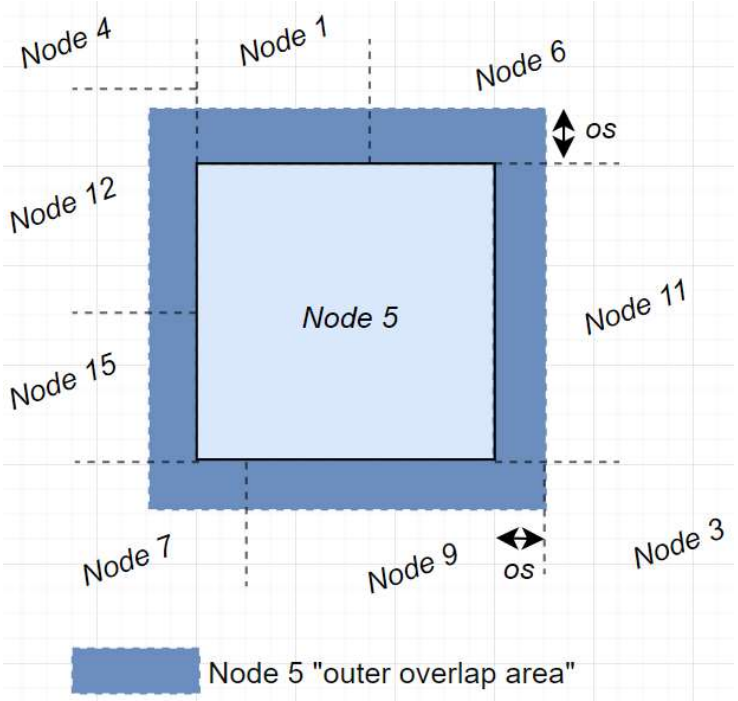
The overlap area width is fixed for all the nodes in a simulation, and it is defined by the modeler (field *overlapSizeMPI* in the config.xml file). We will refer to this value as *os* from now on. *os* should be defined as greater or equal than the maximum mobility/visibility range of the agents.

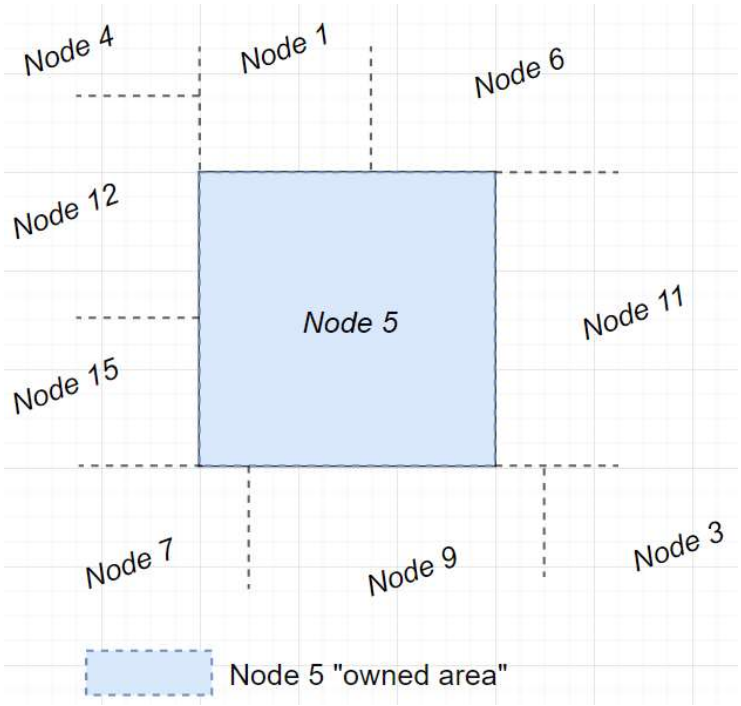
These areas grow as the number of nodes does. So does the synchronization overhead too.

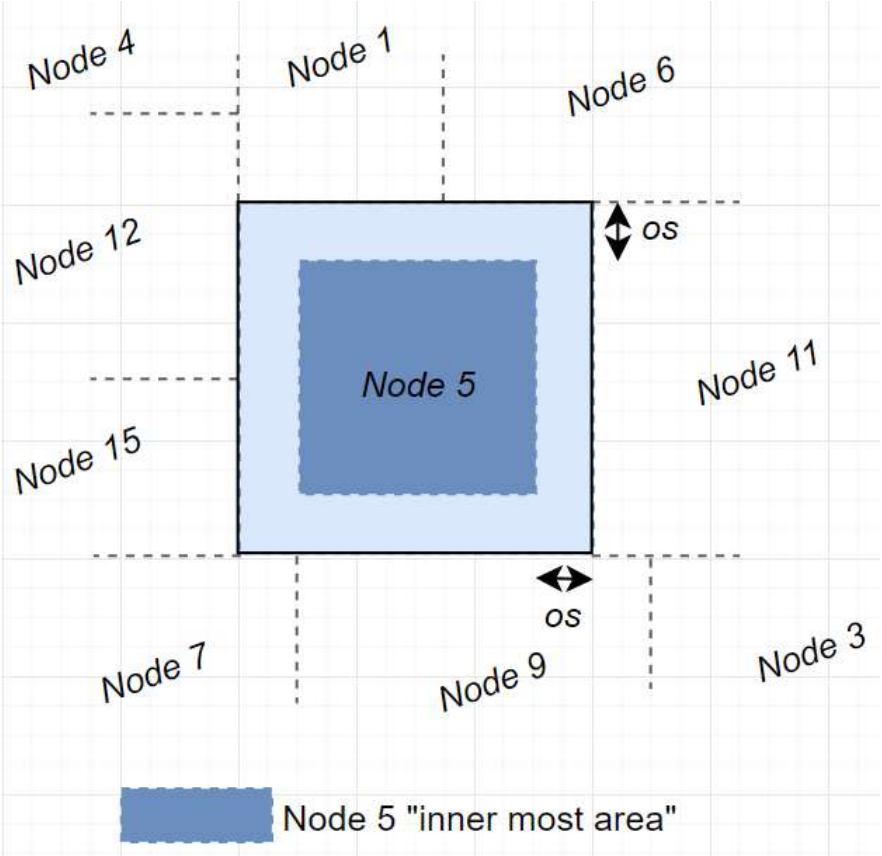
Based on this, we define 3 concentric areas for each node:

1. The outer overlap area, which corresponds to an expanded rectangle *os* cells by each side.
2. The owned area of the node
3. The inner-most area (also referred as “owned area without inner overlap”), which corresponds to a contracted rectangle *os* cells.

These areas are useful when communicating neighbouring nodes. For instance, to know whether an agent belongs to the area of influence a neighbour node or not.







3.2. Maintaining time continuum

Talking about the **state of the agents and rasters** in particular, 2 approaches exist when dealing with time consistency:

1. The **discrete** approach, in which, at step 'i', agents can only see the state of the other agents at step 'i-1'. This means that agents and rasters are only synchronized at the end of each step of the simulation. This approach is less accurate and do not match with a common sequential execution, but it is not conceptually wrong (it depends on how you understand reality or the granularity you consider). And besides, it is much faster since there is only one synchronization event per step.
    - IMPLEMENTATION: have a main data structure for step 'i-1', which is the one used by agents to retrieve data at step 'i', and a backup mirror data structure representing step 'i' that receives agent and raster updates asynchronously in the meanwhile, and that replaces the current step data structure at the beginning of step 'i+1'.

2. The **continuous** approach, in which, at step 'i', agents can see the very last state of the objects around them. In other words, at step 'i' agents will see other agents and rasters at step 'i' too (even though this will not give the same outcome for 1, 2, 4 or n MPI processes, as we will see). To achieve this, agents and rasters are synchronized multiple times in the same step. As a counterpart, this generates way more overhead, so the modeler needs to evaluate whether this option is worth it, even if it is totally accurate, natural and, to some extent, more realistic.
    - IMPLEMENTATION: make subpartitions of the overlap areas that, processed in a certain order, does not even allow collisions. The following subsections are focused on the implementation of this approach in detail.

3.2.1. Agent collisions

The approach used here to deal with agents that move in parallel is to directly avoid collisions. This means never letting 2 agents to access to the same position of the grid at the same time, nor access to the same raster cell at once.

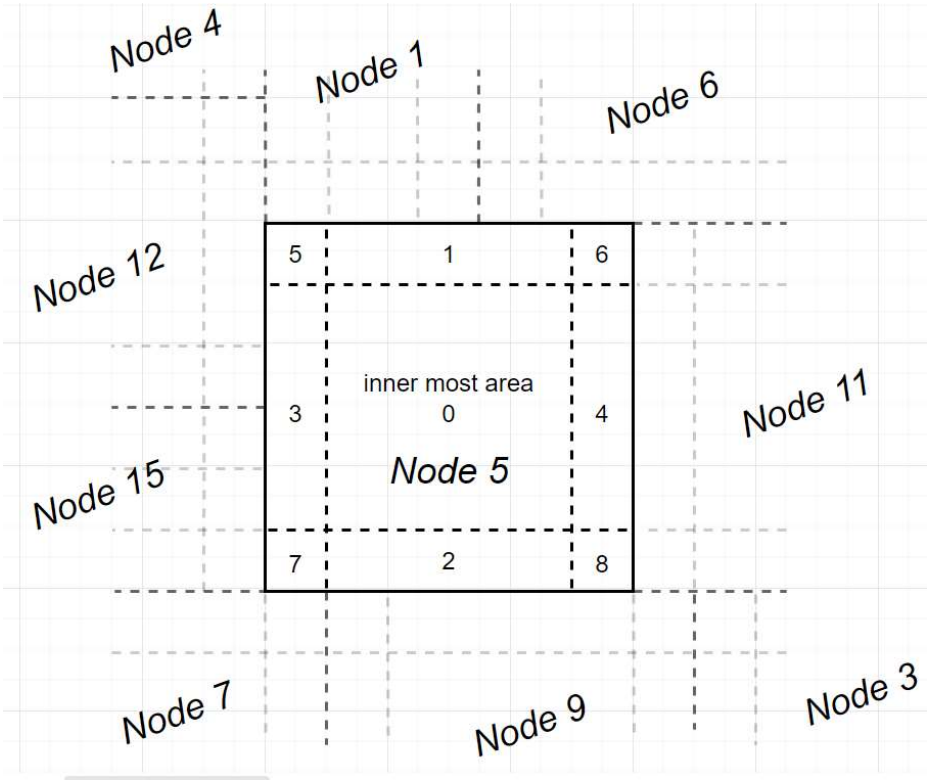
To accomplish this, we define the concept of suboverlap. A suboverlap is a section of the inner overlap area for a certain node with an identifier. If for all nodes, these suboverlaps are defined in the same way, we can process suboverlaps with the same identifier in parallel without risk of collisions.

In this way, we would process first all the inner-most areas (identified by 0) and synchronize agents and raster updates among all nodes. Then execute all the agents in suboverlap areas identified by 1 and synchronize agents and raster updates again. After this, execute agents in suboverlap areas identified by 2 and synchronize agent and raster updates among all nodes once again. And so on and so forth.

Performing it in this way, we will avoid any kind of collision, **provided that the overlap size has been properly defined as, at least, the mobility/visibility range of the agents**, as said in the previous section.

Notice that doing this, the shuffle of agents to be executed at the beginning of each step will be performed multiple times, i.e. for the agents in the inner-most area first and then for those on each one of the different suboverlaps. So this shuffle will differ from the one performed for a totally sequential execution of the very same model with the very same parameters. Furthermore, this shuffle will also differ from the one with a different number of specified partitions. In short, the number of partitions (MPI processes) affects the potential order of execution of the agents, which means a different outcome of the simulation (see section
3.2.3. A full example).

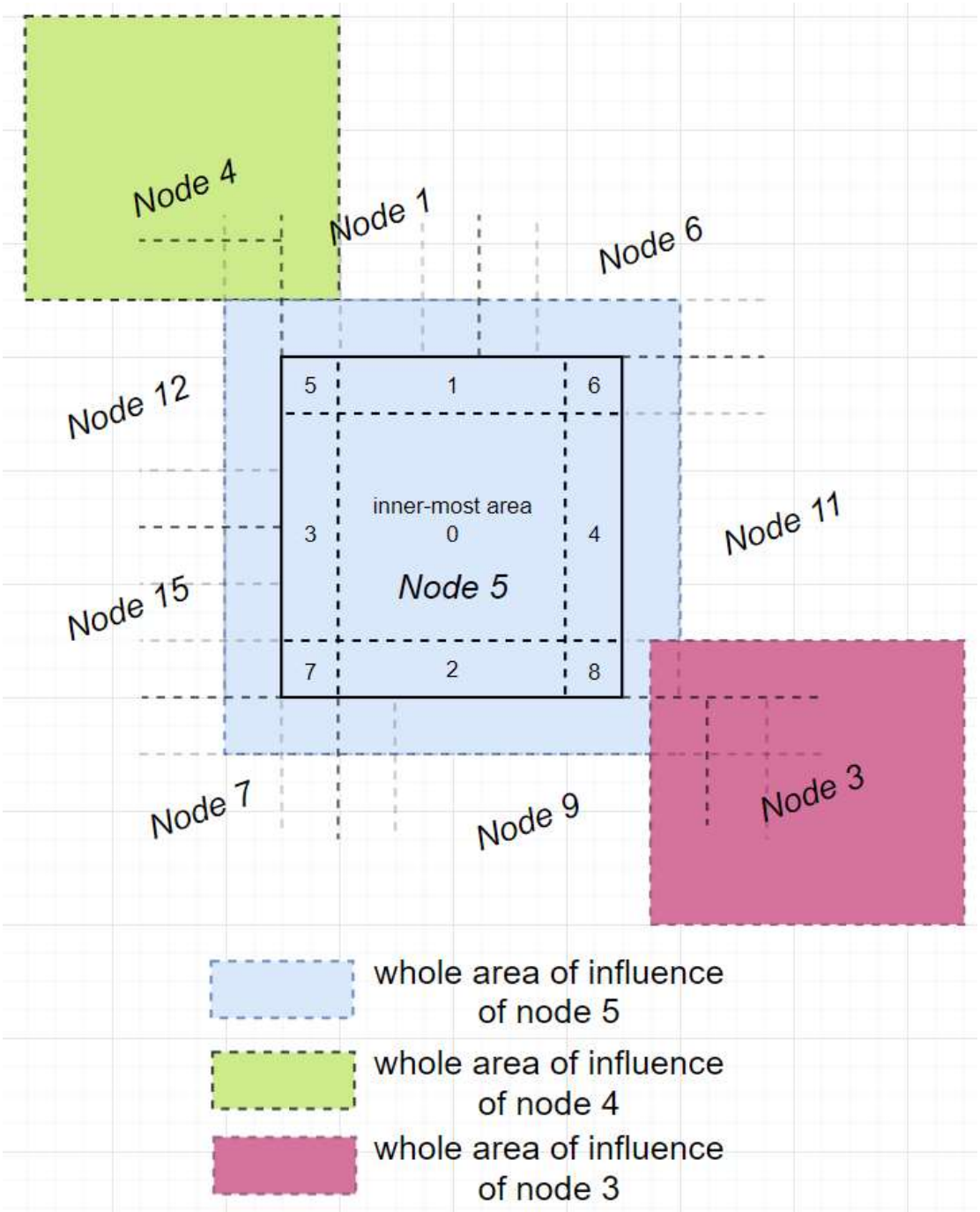
Related to this, upon receipt of the agents after their execution in each suboverlap, these are automatically sorted by ID (since they are saved in a map <id, Agent>). This measure would be necessary if agents would have received asynchronously. Otherwise, the posterior shuffling in the next step would be based in an arbitrary order of the agents. This should not happen, since we want to maintain the coherence between simulations regardless of the order of the arrivals.



When agents move, their executing node must calculate which neighbours needs to receive

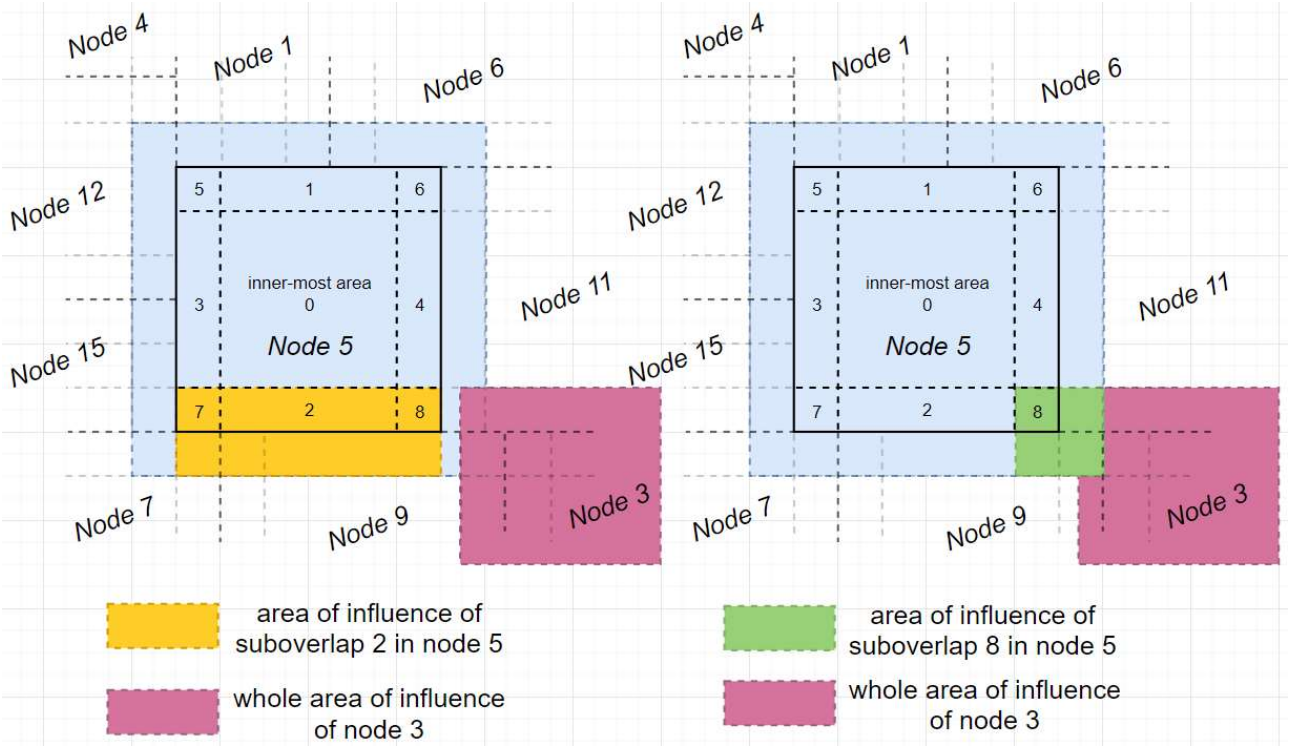
the update, and send to them. For this, we talk about areas of influence. These areas are computed for whole nodes when the full set of neighbours are needed, but also for each suboverlap when sending the executed agents to other nodes. These areas of influence are indeed precomputed and each node knows its own areas and the ones from its neighbouring nodes. In this way, a node knows whether an agent in a particular position needs to be sent to some of its neighbouring nodes.

Some examples are presented in the following figures.



In this graphic, the obvious neighbours of node 5 are: 1, 6, 7, 9, 11, 12 and 15. The area of influence for nodes 3 and 4 are displayed so it can be noticed that node 3 is also a neighbour even if they are not directly adjacent. Node 4, on the other hand, is not, because their areas of influence (or outer overlap areas) are not overlapping each other.

In the following graphic, areas of influence for suboverlap 2 and 8 are shown. Notice that suboverlap 2 does not need to communicate with node 3, while suboverlap 8 does.



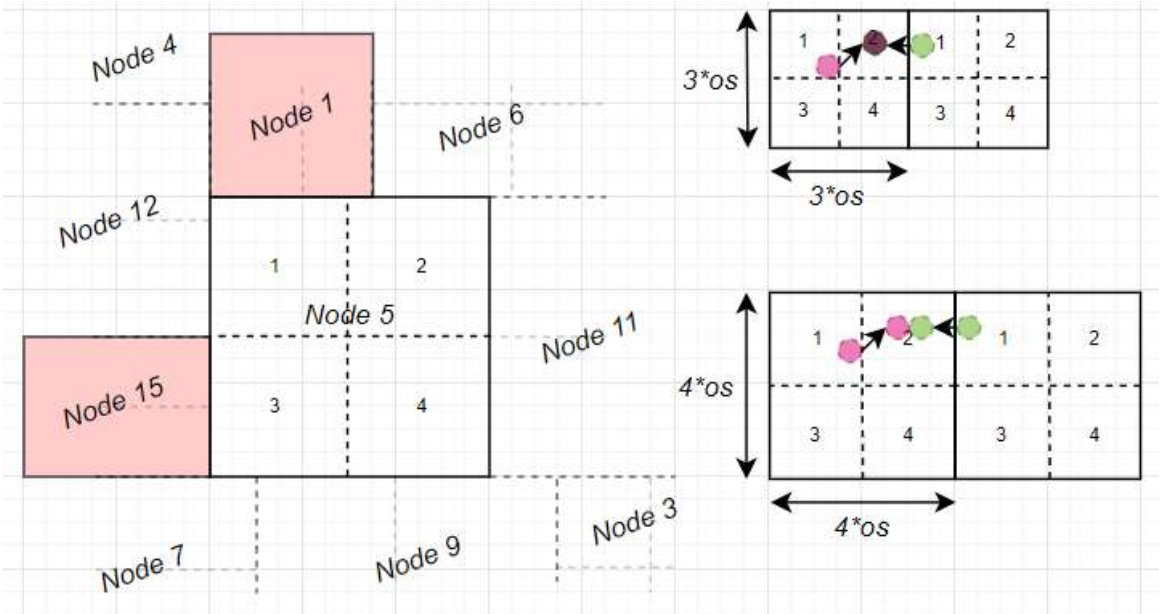
In this mode of partitioning, nodes only need to be 3 times the *overlap size* in width and height. So, theoretically, it is the most efficient partitioning avoiding collisions for small nodes.

There is another mode in which each node is partitioned into 4 subpartitions. The overlap areas are maintained to know whether a node need to send agents to each other or not, just as explained above. We have called this "mode4", and it is considerably faster than "mode9", since less synchronization events are required per step.

The communication logic is the same than the one already explained. The only inconvenience is that these nodes need to be 4 times the overlap size in width and height, unlike mode9.

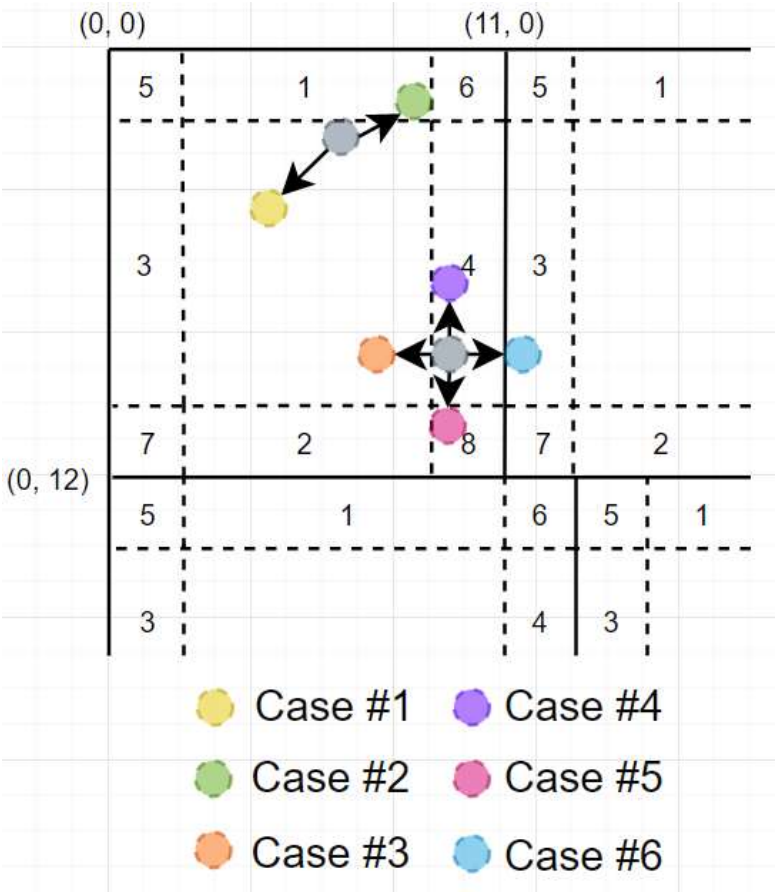
Here, Node1 and Node15 are not possible, since they are < 4**os* (*os* = 2). In the top-right picture, collisions appear, while in the bottom-right they don't, only ensuring a minimum width and height.

For this reason, the default partitioning in Pandora is mode4, even though mode 9 is fully operative too.



3.2.2. Agents movement

When moving agents in a partitioned space, 6 different cases may arise. In the picture, grey dots represent the **original positions**, while the colored ones are their **destination positions**.

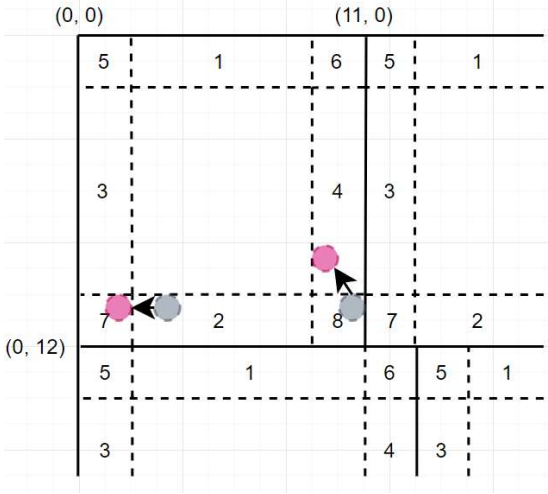


1. Case #1: agent moves within its current node inner-most area.
    - ACTION: nothing. Does not need to communicate anything to any other node.
2. Case #2: agent moves from its current node inner-most area to some of its suboverlaps.
    - ACTION: check the neighbouring nodes of the destination suboverlap (with its area of influence) and send the update to them if necessary.
3. Case #3: agent moves from some suboverlap to its current node inner-most area.
    - ACTION: check the neighbouring nodes of the original suboverlap (for deleting purposes) and send the update to them.
4. Case #4: agent moves inside its current suboverlap.
    - ACTION: check the neighbouring nodes of the current suboverlap (with its area of influence) and send the update to them if necessary.
5. Case #5: agent moves inside its current node, but to a different suboverlap.
    - ACTION: check the neighbouring nodes of both the original and destination suboverlaps (for deleting purposes) and send the update to them if necessary. NOTE: if a neighbouring node receives then the new position of an agent and it stays out of its area of influence, then it is deleted.
6. Case #6: agent moves out of its current node.
    - ACTION: check the neighbouring nodes of both the original and destination suboverlaps (for deleting purposes) and send the update to them if necessary. NOTE: if a neighbouring node receives then the new position of an agent and it stays out of its area of influence, then it is deleted.

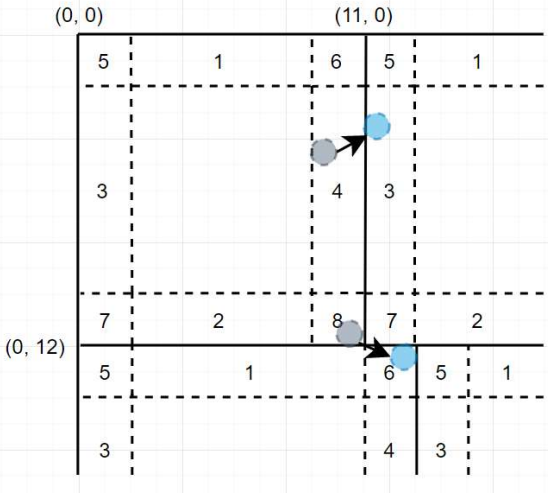
Notice that the action for cases #4 and #5 is the same.

Some examples for each one of the cases are illustrated in the following pictures:

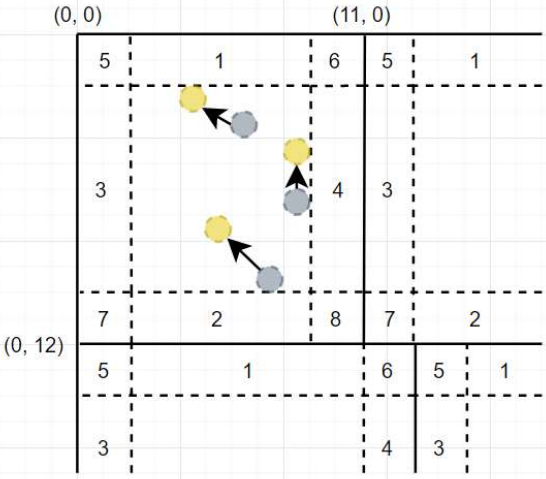
Case #5:



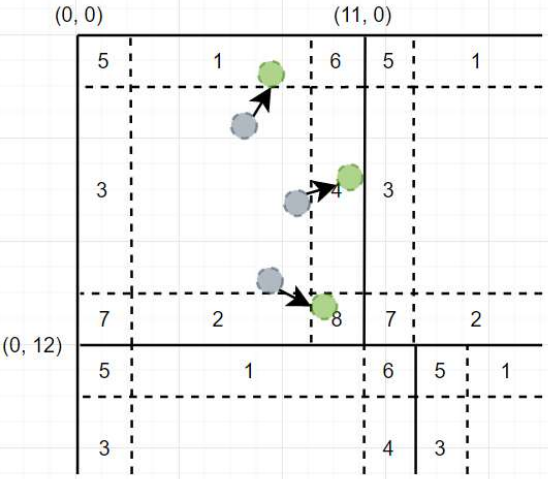
Case #6:



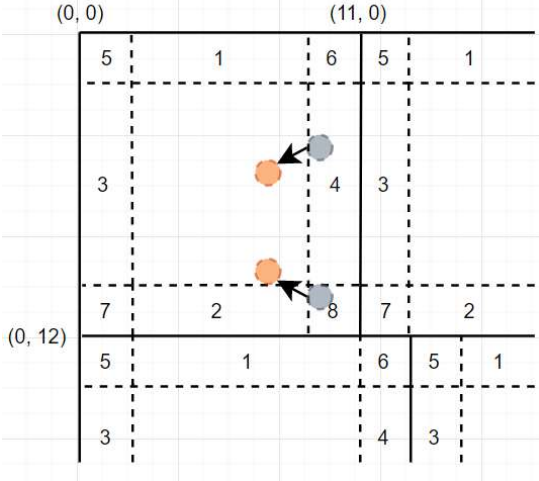
Case #1:



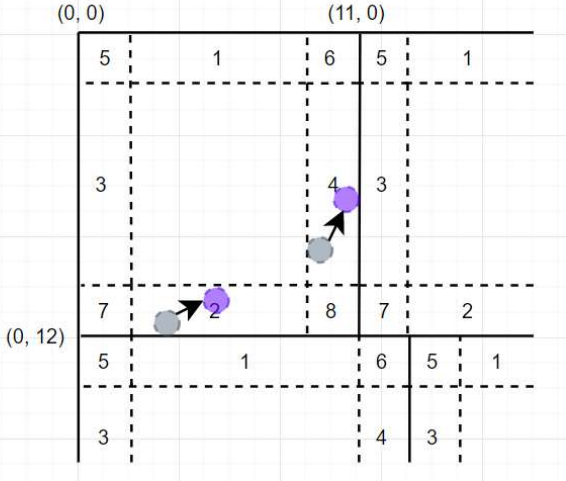
Case #2:



Case #3:

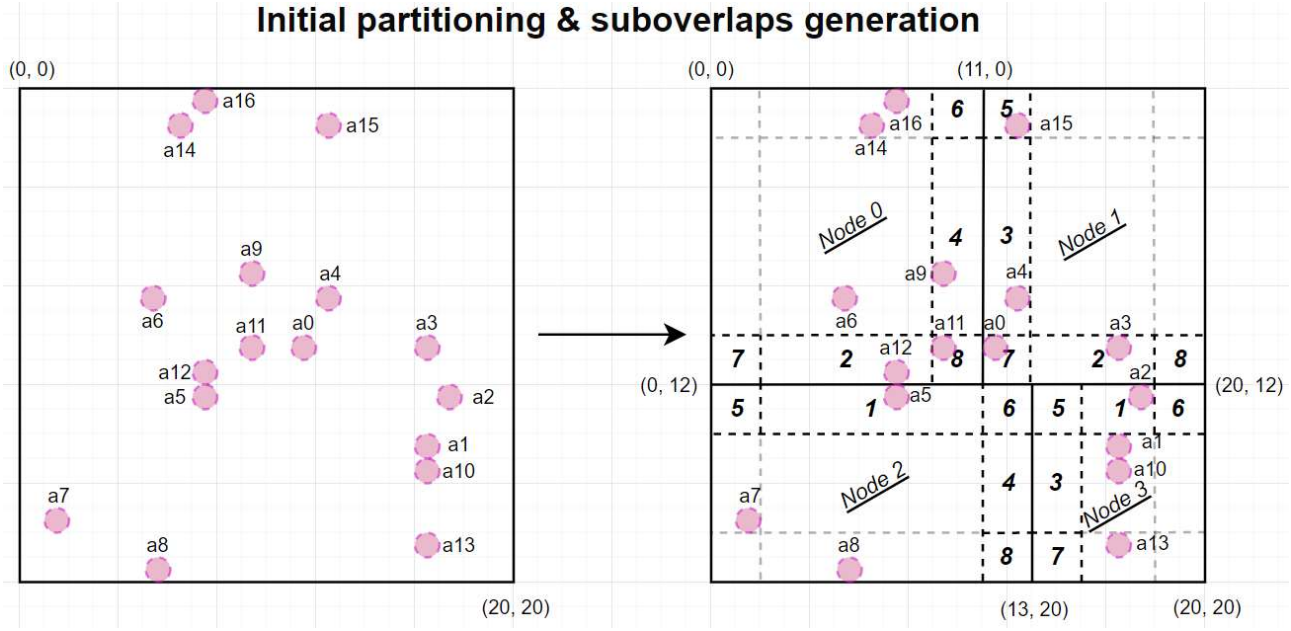


Case #4:



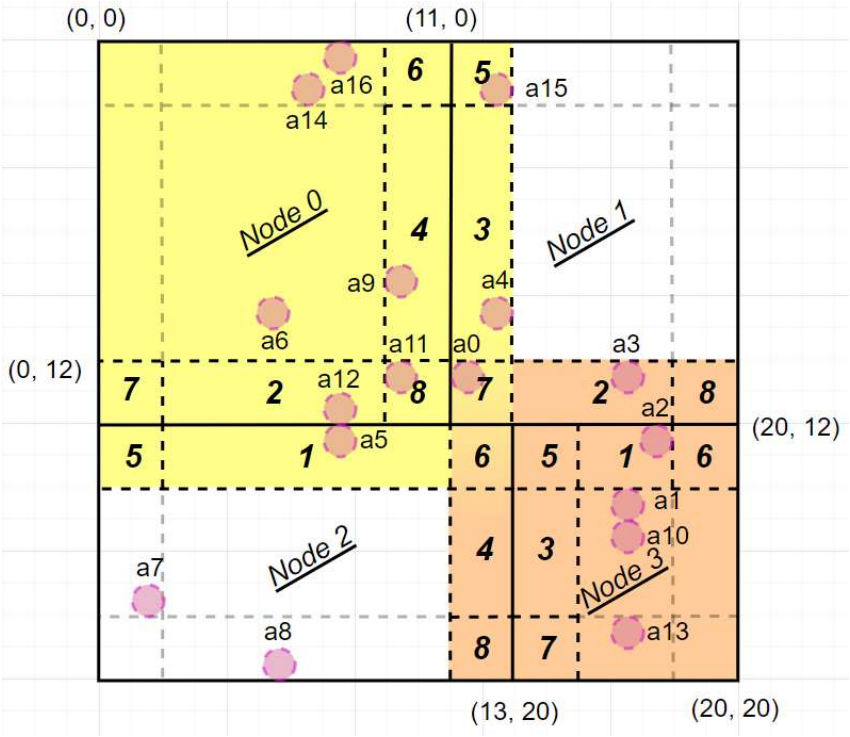
3.2.3. A full example

In this section, the first iteration for a specific example are presented. This example consists of 17 agents randomly created, asked to be divided in 4 MPI processes/nodes. The stablished overlap size *os* = 2.



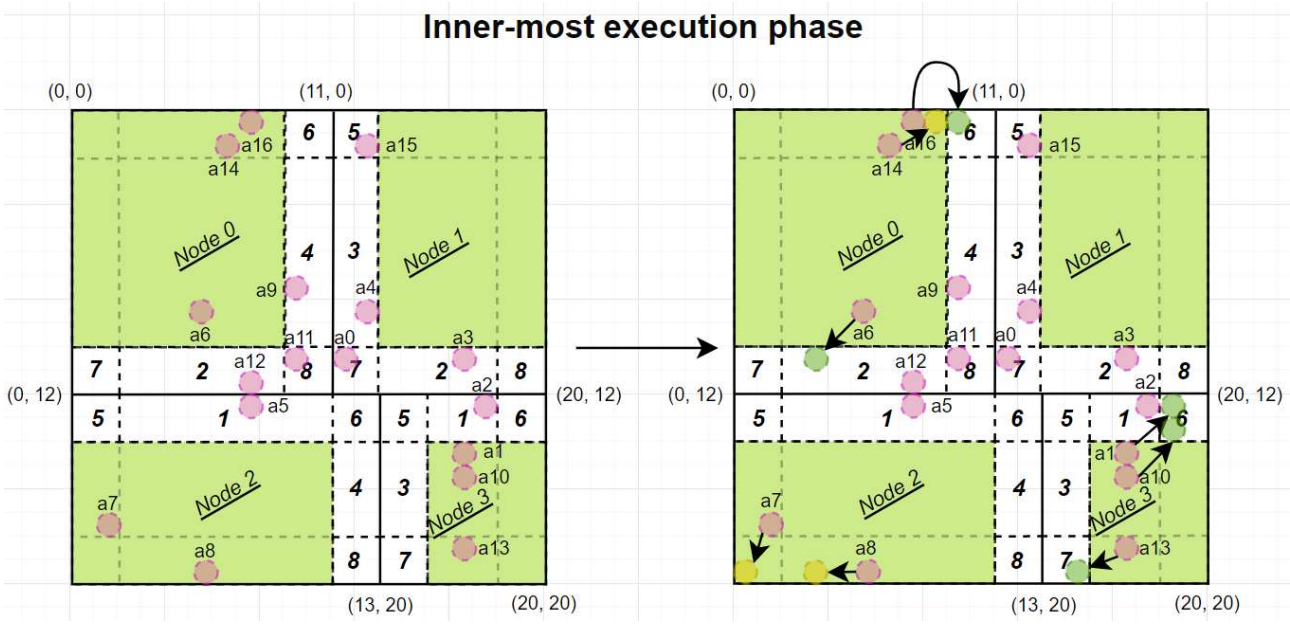
As shown in the following picture, even if node 0 and node 3 are not adjacent, they still need to communicate with each other since their outer overlap areas collide. This is a consequence of this kind of arbitrary partitioning in n nodes.

Besides, notice the situation in which a fraction of an overlap is replicated in more than just 2 nodes. See suboverlap 7 of node 1 or suboverlap 6 of node 3, which are in the local memories of the 4 nodes for this particular example.



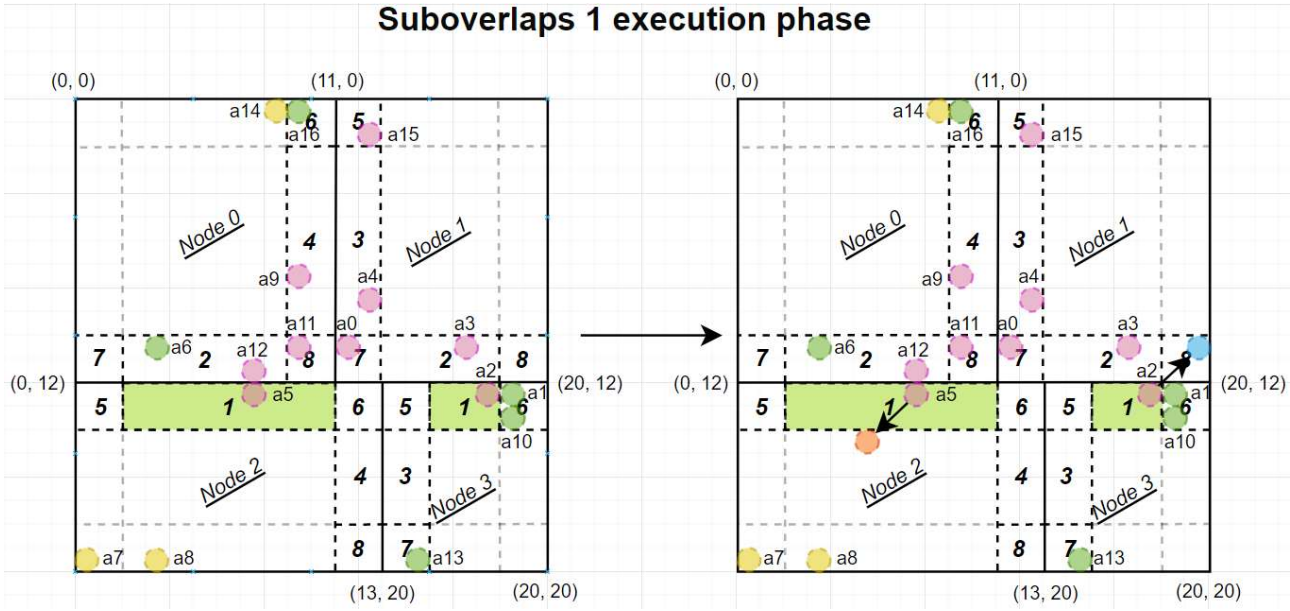
Then, first the agents in the inner-most areas are executed. In this example we have a particular kind of partition: a corner partition. As a side note, non-corner partitions are never generated when specifying a low number of partitions: <= 9, sometimes more. With > 9 partitions, it might appear some non-corner partition.

Here, we just have 4 corner partitions. For corner partitions, the inner-most area includes one or more of its own suboverlaps. This is because that corner suboverlaps does not need to communicate with any other node, so the agents in there are considered inner-most agents too and processed together in the inner-most execution phase (e.g. agents a7 and a8 in Node 2).

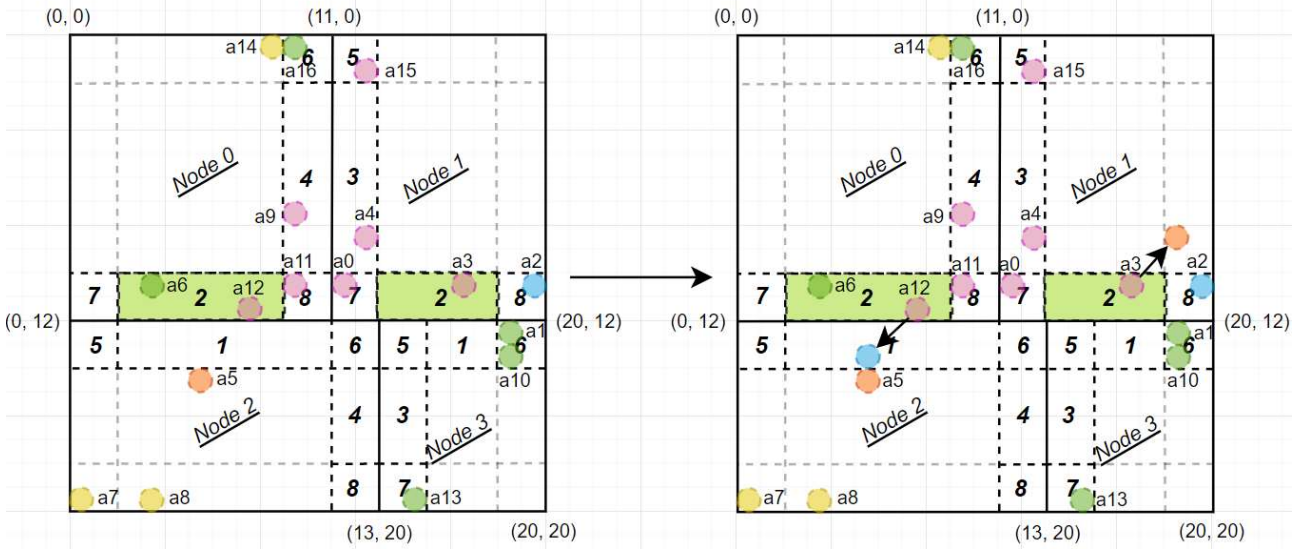


So, agents moved in this example according to a modeler-defined logic considering an underlying raster that represents field resources. We are not interested in the coded behaviour. The fact is that the agents move.

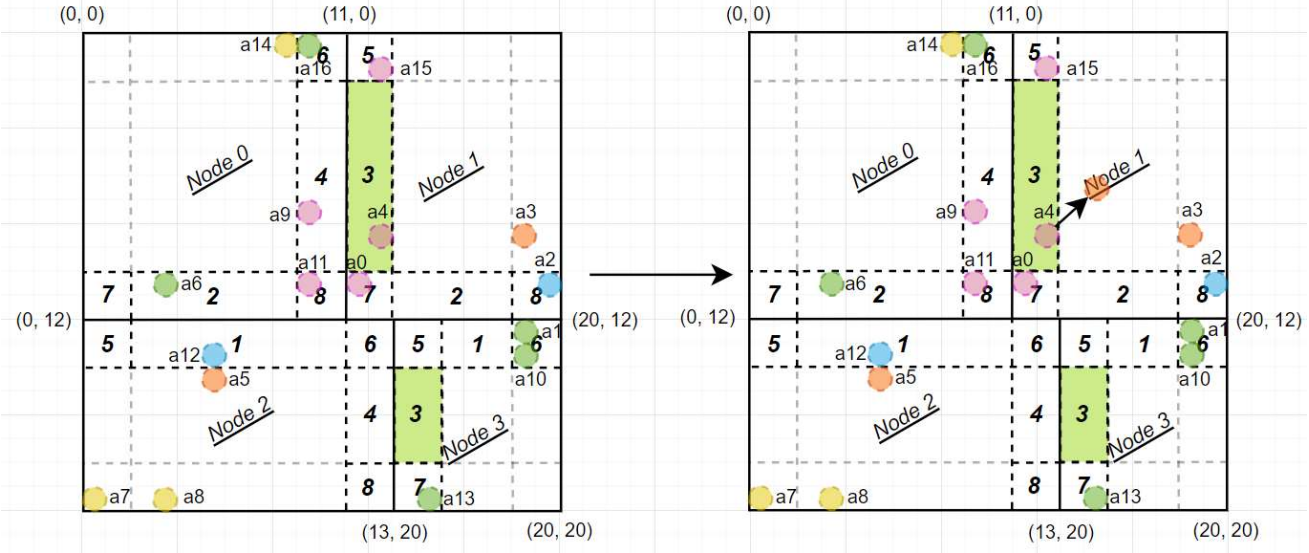
Then it comes suboverlap 1 execution phase, then suboverlap 2 execution phase, and so on.



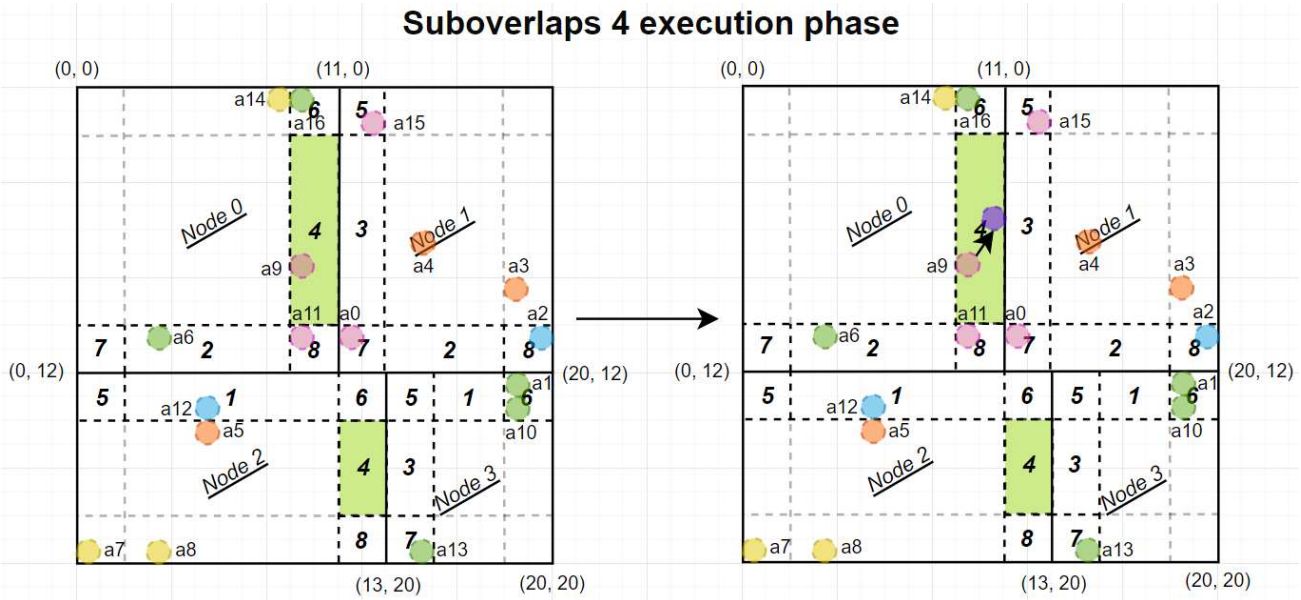
Suboverlaps 2 execution phase

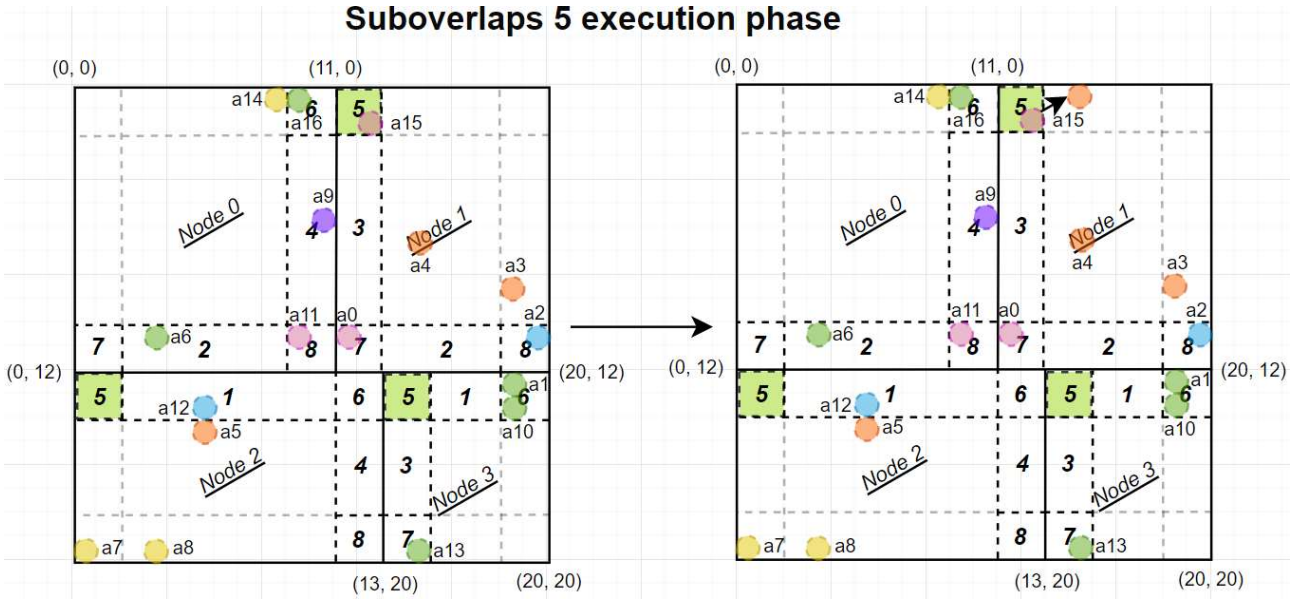


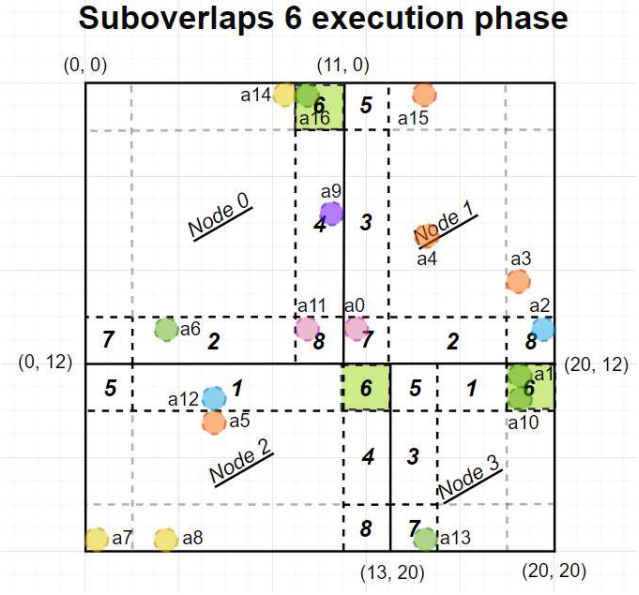
Suboverlaps 3 execution phase

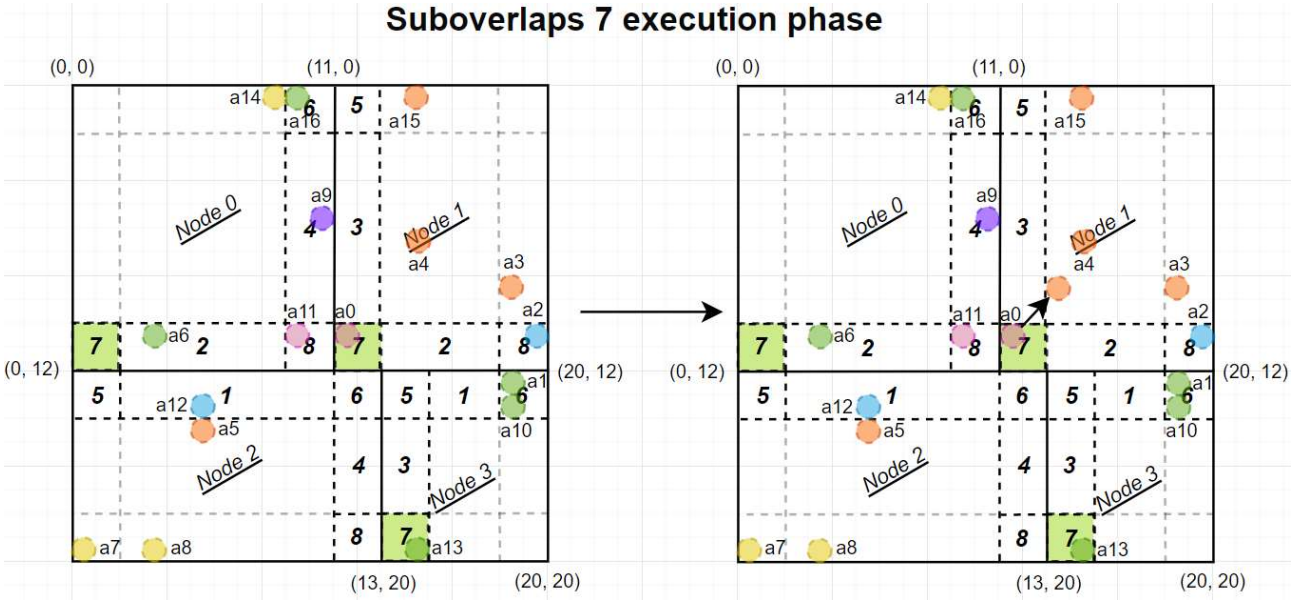


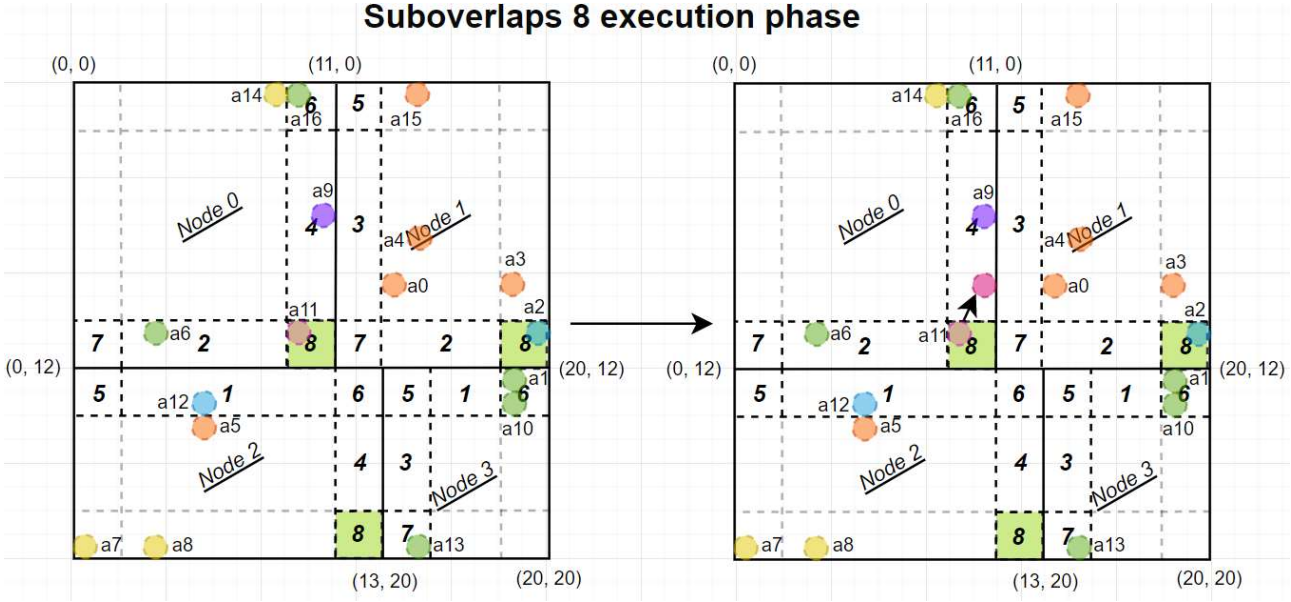
Suboverlaps 4 execution phase

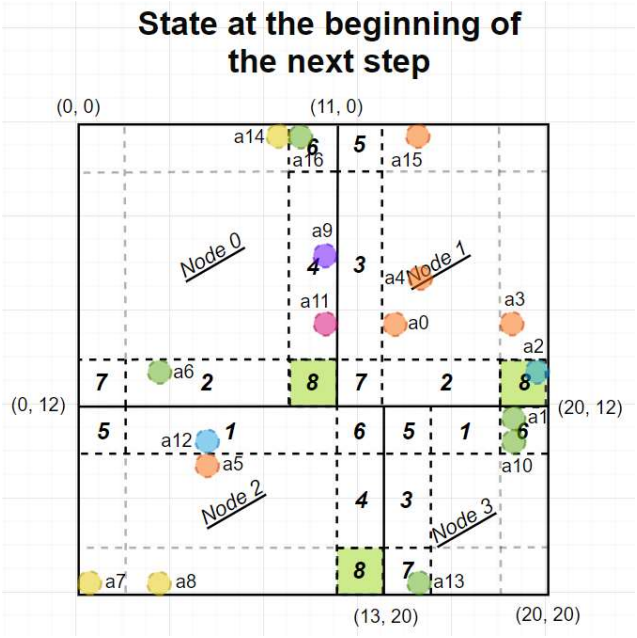



Suboverlaps 5 execution phase
(0, 0)
(11, 0)
(0, 12)
(20, 12)
(13, 20)
(20, 20)
Node 0
Node 1
Node 2
Node 3
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a14
a15
a16


Suboverlaps 6 execution phase
(0, 0)
(11, 0)
(0, 12)
(20, 12)
(13, 20)
(20, 20)
Node 0
Node 1
Node 2
Node 3
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a14
a15
a16


Suboverlaps 7 execution phase
(0, 0)
(11, 0)
(0, 12)
(20, 12)
(13, 20)
(20, 20)
Node 0
Node 1
Node 2
Node 3
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
a12
a13
a14
a15
a16





At each suboverlap processing, non-yet-executed agents for each node are run and sent to other nodes if necessary. In this example, the 6 different situations appear, being colored each one by its color code.

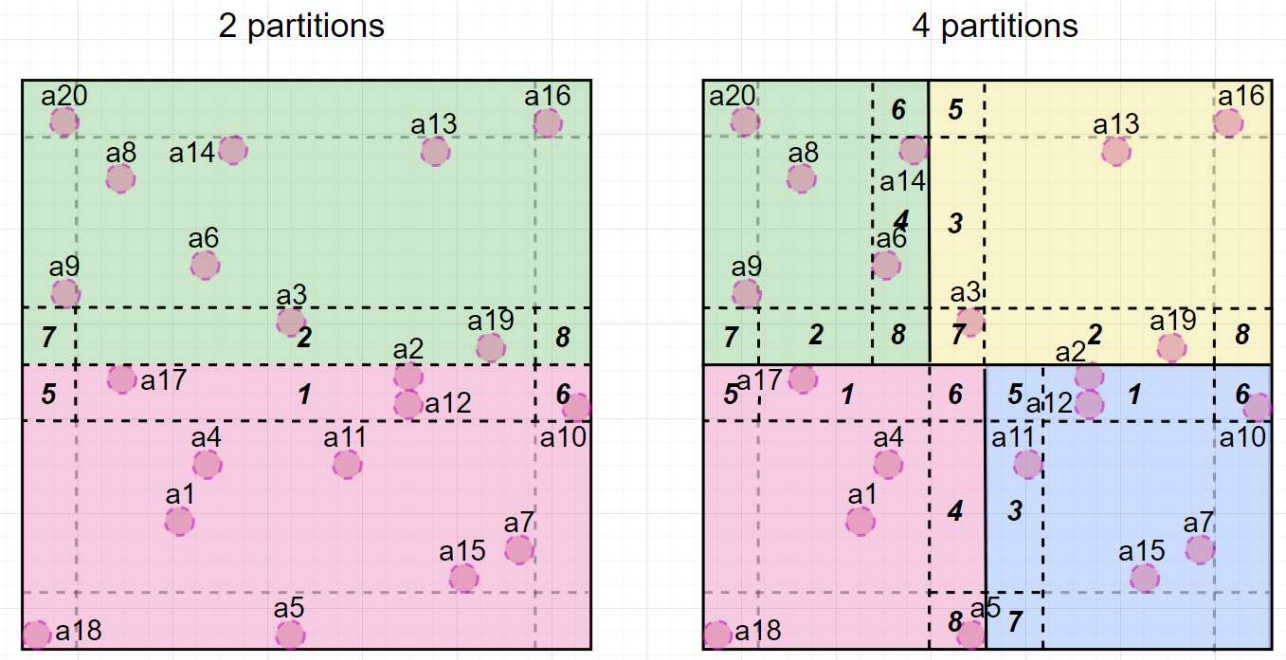
At the end of suboverlap 8 execution, the resulting state is the initial one for the next step of the simulation.

Running it in this way, no conflicts are possible between agents and each execution runs in a deterministic way, provided the same initial randomness seed and the same number of partitions (*). For different runs, the state of the simulation at the end of each suboverlap execution is consistent among them, BUT not necessarily when being in the middle of a suboverlap execution.

(*) For a different number of specified partitions:

1. Agents in a specific node will belong to different suboverlap domains, which will imply that their order of execution will be different.
2. When shuffling agents to be executed, the result depends on the original list of agents to shuffle. If there are, for instance, less agents now because the number of MPI processes are

greater, then the resulting order of the agents after shuffling will be different.



For these 2 reasons, the resulting simulation with 1, 2, 4 or n MPI processes will vary. Both cases are illustrated in the picture.

1. Agent a11 belongs to the inner-most subpartition in the left case, while it belongs to suboverlap #3 in right case. This will cause a different order in the execution of the agents. For instance, in the left case a11 will always be executed before a12 and a2, while in the right case a11 will always be executed after those 2. Just a consequence of this approach.
2. For the yellow node in the right case, the inner-most list of agents is {a13, a16}, while for the green node of the left case is {a13, a16, …}. The shuffle applied to these 2 lists of agents will not ensure that a13 and a16 will respect their order of execution.

3.3. Raster updates

At the beginning of each simulation step, the inner overlap areas of the dynamic rasters are sent to the neighbouring nodes, since they can be updated by means of the virtual function *World::stepEnvironment()*.

3.4. The discrete approach

4. References

1. **Rubio-Campillo, Xavier, Lázaro, Quim and Navarro, Germán.** GitHub Pandora. *https://github.com/HPC4SC/PANDORA.* [Online]
2. **Navarro, German.** DockerHub Pandora. *https://hub.docker.com/repository/docker/genajim/pandora.* [Online] 2020.
3. *Scalable HPC Enhanced Agent Based System for Simulating Mixed Mode Evacuation of Large Urban Areas.* **Wijerathne, Lalith, et al.** s.l. : Elsevier, 2018. International Symposium of Transport Simulation. pp. 275-282.