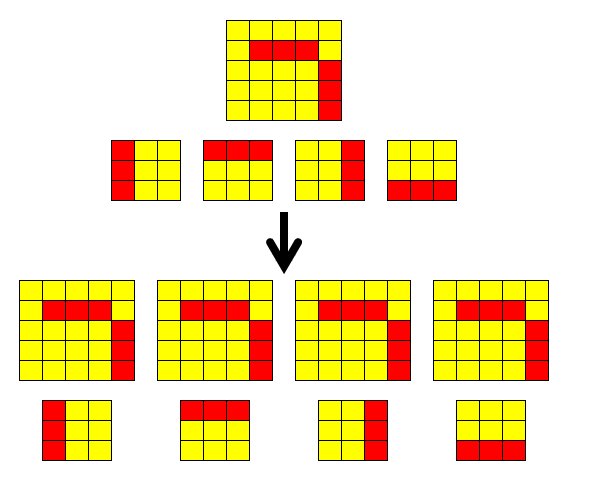
**Parallelization Approach for SETL**

The goal of the execution of this program is, given a map of the world, the design of a specific ‘alien’, and a number of iterations, to search through the world, and its subsequent ‘evolved’ iterations, for occurrences of said alien, even if it is on a different rotated orientation. Thus, the tasks could be simply divided into two: the searching of aliens, and the evolution of the world towards its next iteration.

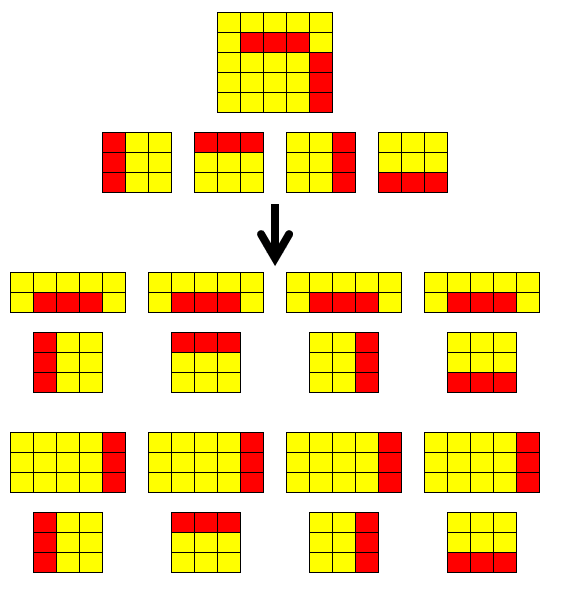
Due to the fact that there are four rotated orientations of the same alien, the most obvious and natural method of parallelizing the searching for said alien would be to have four slave processes, each searching for the alien in a different, specific orientation. Figure 1 below illustrates the parallelization of a world with a pattern among four slave processes.



*Figure 1: The parallelization of a world with a pattern among four slave processes.*

However, this simple method will not allow us to take advantage of additional hardware resources as the amount of code that can run in parallel at the same time is limited to five processes – one master and four slaves. Thus, the next step in parallelization would be to split up the world itself for different slave processes to search through, using the same alien orientation.

For example, in our world with a pattern example, if we were to have a total of nine processes – one master and eight slaves – we would be able to split the slave processes into groups of four, with each group responsible for each pattern. Thus, the example would look like what Figure 2 illustrates.

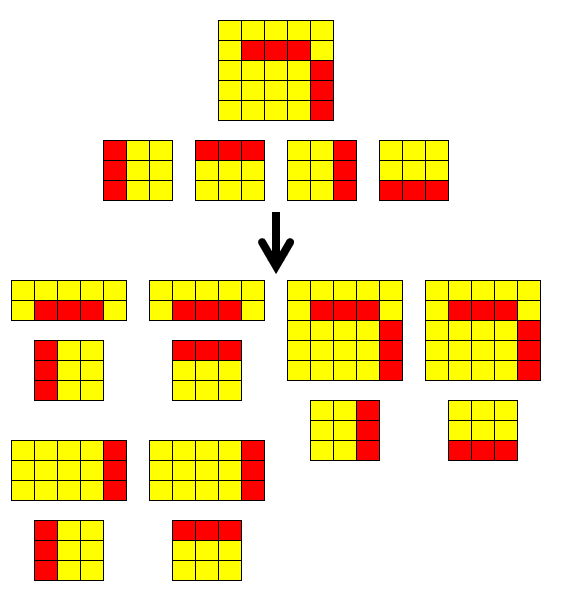


*Figure 2: The parallelization of a world with a pattern among eight slave processes. The first four slaves would receive a top half of the world, while the last four slaves receive the rest of the world, a bottom half.*

Due to the fact that the world is cleanly split into parts, and that the search for some aliens may require looking at the cells of more than one part of the world (for example, with our world, pattern and eight slave processes, to start a search at the second row of the top half of the world, we require information regarding the cells of the first two rows of the bottom half), communication between slave processes will be required.

Thus, it is implemented for each slave process to first receive information from the previous slave process, if any, regarding possible aliens beyond the top border of the current part of the world that it possesses, act on this information and continue the search, while also starting new searches in parts of the world that it has information over, before sending unfinished results to the next slave process, if any.

If the number of slave processes is not divisible by four, and thus cannot be split into even groups with each responsible for a different alien orientation, we will simply split them into groups where the numbers of slave processes in each group differs only by a maximum of one. Figure 3 illustrates an example of the parallelization of a world with a pattern among six slave processes.

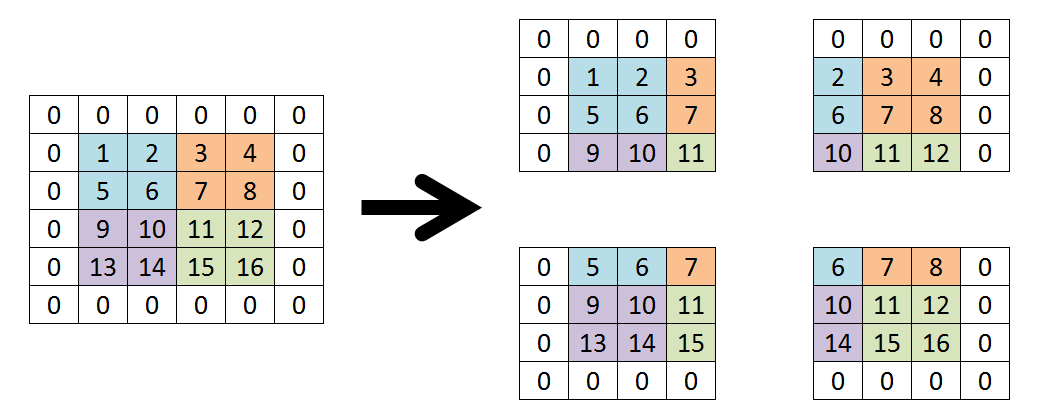


*Figure 3: The parallelization of a world with a pattern among six slave processes. Processes 1 and 5 would be in the same group and processes 2 and 6 be in another same group, and each member of the same group receive a different half of the world and the same alien orientation, while processes 3 and 4 would each be in a group of their own and thus each possess the entire world.*

While the evolution of the world itself could be parallelized as well, it is concluded to not do so, and instead simply leaving it to the master task, which would work on evolving the current world to the next after it has finished passing around the current world for the slave processes to perform their searches on.

There are several reasons to the making of this decision. Firstly, it is not as computationally expensive as the searching part (which has four nested for-loops and thus comes to a worst case complexity, with two for-loops to traverse through every cell in the world, while two more for-loops search for the pattern with each cell as the start, while evolution is simply at complexity, since the number of neighbors each cell has is constant no matter how large the world grows).

Secondly, communication overhead increases proportionately the more evolution is parallelized, as each cell needs information about its neighbors, and thus the world cannot be cleanly split into non-overlapping parts (as illustrated by the act of attempting to parallelize the evolution of a world with four processes in Figure 4).



*Figure 4: An example of how we would have to split up a world to four different processes for them to execute the evolving of the world in parallel. Notice that the four cells in the middle (6, 7, 10, 11) are in each part of the split world, while the eight cells that are neither at the corner nor in the middle of the world (2, 3, 5, 8, 9, 12, 14, and 15) are in two of the four split worlds.*

Thirdly, this adds a further additional layer of complexity into the code, as the master process would then have to be able to decide, given the number of processes there are, which slave processes to search, which to evolve, and which on what part of the world.

Overall, the considerations of these disadvantages result in the conclusion and decision that the benefit of parallelizing this action is hardly worth its costs.

Finally, the output of the results requires that the results be printed in a specific order. As parallelization of the program ensures that tasks will be performed in arbitrary order, the results that the master process collects may not remain in the same order during different executions, and it is thus necessary for us to sort the results before outputting them.

Due to the fact that the results are stored in linked lists, thus making it slightly more challenging to sort them (due to lack of a direct access to any nodes in a linked list), and for the sake of ease in coding, the output of the results involves a function that repetitively iterates through the entire linked list, looks for the result that should be printed first (since results are to be sorted in terms of iteration, followed by rotation, row, and column, the algorithm thus searches for the result with the lowest integer value representing iteration, and if two or more results hold the same value, it will then used the value for rotation next to determine, followed by row, and finally column), and then print it.

After a result has been printed, its iteration will be marked as to allow the algorithm to know to ignore it in successive iterations (as the program exits shortly after the output of the results, without using them anymore, thus it is decided acceptable to alter the results, though if the altering of results is unacceptable then we can add an additional Boolean variable into each node of the linked list denoting whether said item has been printed or not).

This execution was deemed satisfactory, especially since sorting algorithms like the insertion sort, the bubble sort, and the selection sort have average complexity anyway.

**Execution Results and Analysis**

blah