Adam Alsegard, A0153097B CS3210

Assignment 2 - Report

Description of parallel solution

In designing the parallel algorithm I followed Foster's design principles for partitioning, communication, agglomeration and mapping.

Partitioning
The most obvious way for this problem is to partitioning the data. The input is two square matrices, a NxN world and a PxP pattern. The data can be partitioned into rows, columns or square cells with row partitioning being the choice that makes most sense in regard to the sequential algorithm.

The sequential search algorithm works through all cells of the world row-wise. For every cell a nested for-loop search through every cell in the pattern and checks to the corresponding value in the world. If a check returns "no match" it moves on to the next cell in the world. This is done four times, once for every pattern rotation. After this the world is evolved, which is also done row-wise by nested for-loops. Partitioning data in rows would mean that every processing node only searches and evolves the cells in its own data set.

The way to partitioning the functionality would be to divide the patterns into four separate groups so every node only search for one rotation of the pattern. The search part and evolution part can also be separated into two parallel groups because the world matrix is not updated until the end of the generation/iteration.

My solution is a bit of both. I partitioned the data into rows but also the patterns into four functionality partitions.

Communication
The initial data all nodes needs is the size of the world, size of pattern and number of iterations. These are all broadcasted to all nodes by the master node, which is the only node reading the data from the files.

For the searching part every node need the current world data of the partitioned cells, but also an additional *patternSize-1* rows at the bottom.

For the evolution part every node needs one additional row on top and one additional below. However, the one below is already taken care of because of the condition above.

In the initial send all rows can be sent in one chunk but for every iteration every node needs to send its bottom row to the neighboring node below and the *patternSize-1* rows in beginning to the neighboring node above. This is done after the evolution of the world.

One part of the assignment was to sort the results by iteration→rotation→row→column before printing. To achieve this the resulting lists needs to be sent back to master node in order, or at least be printed in order by nodes on the hosting computer.

Agglomeration

To be able to work on all four patterns at the same time the processors are divided into four communicators. In each communicator there is one local master. Every communicator is then changed into a Cartesian coordinate system with one column and *p/4* rows, where *p* is the number of processors. If the number of processors it not divisible by four the leftovers will be divided equally as well. However, the most efficient solution is to have the number of processors as a multiple of four, otherwise some nodes will be idle for long periods and not fully utilized.

To minimize the communication the minimum number of rows per node is *patternSize-1*. In that way there is only one node below that needs to send updated world data after every iteration. The number of rows can be calculated as: *newSize = size/ (p/4)*. The maximum number of nodes, in theory, for a world of 3000X3000 and 5X5 pattern would then be: 4 = 3000 / (p/4) <=> p = 3000.

Mapping

In the lab cluster we have a total of 16 processing nodes. The i7 Tower PC has 4 cores with 2 threads on each. All-in-one PC has 4 cores and Jetson also has 4 CPUs, alas only one on-line and 3 off-line. The off-line cores will however spring to life when needed to.

To equalize the running time of the different communicators the CPUs of each machine in the lab cluster is distributed equally among the communicators, meaning that each communicator gets one CPU from Jetson, one from All-in-one and the remaining 2 from Tower PC. The global master and all local master nodes are all allocated on the Host machine, in this case the Tower PC.

Putting it all together

As mentioned before, only the global master reads the data from file. Initially world size, pattern size and number of iterations are broadcasted to all nodes. Thereafter the read world and correctly rotated pattern is sent by non-blocking to the 3 local masters. The correct pattern is then broadcasted by the local node to all nodes in the communicator. The world is partitioned and chunks of *newSize+1+(patternSize-1)* rows are sent to the nodes in the communicator. If there are leftover rows, the last node will include them as well. After this, the initial distribution of data is complete and the iteration loop can begin.

First all nodes search for aliens in their assigned area. Then the result is sent back with blocking to the local master and local result list is purged. When done collecting the local masters will send back results to global master, by blocking as well, and thereafter purge their lists. When everything is collected the global master will print results for the iteration and then purge its list. This way, the ordering of the result (iteration→rotation→row→column) is guaranteed.

Once each node is done with its communication it will continue with evolving its part of the world and then send rows by non-blocking to neighbors above and below in the coordinate system. Before the world is updated a MPI_Waitall call ensures all necessary communication is done and next iteration can begin. This way the same evolution is done 4 times, once in every communicator, but in the same time the communication between communicators is minimized.

Result analysis

Table 1 – execution time

World	Pattern	Iterations	Matches	Seq (Tower)(s)	Par (lab)(s)	Par (NSCC)(s)
Simple	Simple	100	222	0.05	0.01	-
random1000	glider3	100	51022	15.75	8.27	0.66
random1000	glider5	100	2362	12.29	4.54	0.59
random3000	glider3	100	116798	80.25	39.18	2.86
random3000	glider5	100	16823	115.54	40.99	3.86

Table 2 - speedup

			Speedup		
World	**Pattern**	**Lab, write(s)**	**Lab, write**	**Lab, print**	**NSCC (48)**
random1000	glider3	4.55	3.46	1.9	23.9
random1000	glider5	4.56	2.7	2.7	20.8
random3000	glider3	26.6	3.0	2.0	28.1
random3000	glider5	40.96	2.8	2.8	30.0

In *Table 1* we see timings for the original sequential algorithm (on the Tower PC), the parallel algorithm on 16 nodes in the lab cluster and the same parallel algorithm run on 48 cores in NSCC supercomputer.

In *Table 2* we see the timing for parallel algorithm on 16 nodes in cluster as well, but instead of printing the result on the screen it's written to a file. Then the speedup of these three approaches is shown.

The written files were used to compare the results to the results of the sequential version with the *diff* command, to make sure the output was exactly the same and not just the amount of matches.

Analysis

As can be seen in *Table 1* the parallel solution is better for all combinations of worlds and patterns with speedups between 2 and 3. When compared to the version that writes to a file it becomes apparent that writing to the screen is a major bottleneck, especially when the amount of results is huge. This was partly solved by printing every iteration instead of only at the end but it was not enough, especially when working with the smaller pattern.

In the lab cluster 16 nodes is the optimal solution, that way all CPUs run one thread each. If we use more or less threads the speedup decreases.

The results on the supercomputer are much improved with speedups around 30. This can partly be explained by not printing to any screen but foremost by the architecture of the system. Every node has 24 cores so we only work with 2 nodes for the result in *Table 1* and therefore don't have to communicate between machines as in the lab cluster.

Actually, the results on 48 cores were only slightly better then on 24 cores, only a few tenths of a second. I tried to run on 240 cores as well (random3000.w, glider5.p) but then the execution time increased to 7.66 seconds. This is probably mainly because of the blocking communication and the small workload. If the workload was increased significantly then more cores would give a better result. I also tried to submit a job with 3000 cores (random3000.w, glider5.p) to test my theory of maximum number of processors but at the time of submission the job is still waiting in line at NSCC. It will probably be significantly slower because of the extensive communication.

As hinted, one of the parallel solutions biggest bottlenecks is the blocking communication when gathering the results for every iteration. Especially the first communicator, with global master, will be slower than all the rest and the second communication, which always send it's results first, will always have to wait. I tried working around this by putting the global master outside of the communicators and let it deal solely with collecting and printing results but this turned out to be even slower. Another possible solution would be to send back the result by non-blocking and then insert them in the correct place in the list. This could work, but I'm not sure if it would actually be faster because instead of constant time O(1) insertion at end we would get linear insertion time O(n). However it is difficult to estimate the time complexity of the blocking calls so I don't know which would actually be better. The purging of the lists also introduces linear complexity, however it would be the same complexity if we had to iterate to the right spot every time we needed to print and the code complexity would also be higher.

One thing I did to optimize was to distribute the nodes in every communicator evenly among the machines in lab cluster, to have approximately the same running time, and to have all local masters on the same machine, to make the last collection faster.

Another possible optimization is to divide the evolution and searching as well. However this would mean even more communication so to utilize it we would probably need a much larger world and/or pattern.