Assignment 4 & 5, Parallel Computing

Miao Qi, qim@rpi.edu
Jinghua Feng, fengj3@rpi.edu

April 9, 2019

1 Strong scaling with parallel I/O

1.1 Number of alive cells with ticks

Note that our tick 0 is from the generation after initialization so the number of ALIVE cells does not start from the total size of cells (i.e. all cells are ALIVE). If we include in this number in the graph as the initial point, there will be a significant decrease at the beginning since most cells died compared to their initialization states.

It is clear to see from the Figure 1 that, for each different number of MPI ranks/threads,there is a significant increase in the number of ALIVE cells during the first a few ticks but then the number of ALIVE cells only oscillates slightly around 41% - 42%. And, according to the Figure 2 which is a zoomed in version of Figure 1,the oscillation shapes of the lines are similar (i.e. they increase / decrease for the same tick number). For the same tick number, as the number of threads per MPI rank (TPR) increases, the number of ALIVE cells increases. But this difference becomes smaller and smaller when the number of threads per MPI rank increases. This is more obvious in the Figure 2 after we zoomed in (i.e. y = 4e8 to 4.5e8) since the three graph lines for 64, 32, and 16 threads per MPI rank are very close to each other and hard to distinguish.

The increase of the survival rate with the increase of TPR is probably caused by the rising asynchronization generated by the increasing TPR. Based on our experiments, we have the following two facts:

- All the cells will die in 3 ticks if we set TPR as 1 (no additional threads created), with threshold of 0 (strictly following basic rules).
- There will be some cells to survive after many ticks if we set TPR larger than 1, still with threshold of 0 (strictly following basic rules).

The reason behind these two facts is that the asynchronization generated by threads can make some cells on the boundary between threads survive. For instance, suppose thread A and B are neighbor threads which share the same local universe. At tick 1, normally, most cells of last row of A dies after first tick, so as to the cells of first row of B. But if thread A runs faster or starts earlier so that the last row of A has updated their status before B starts. Then the cells in first row of B can survive because their upper neighbors have already been updated as dead. Thus the increase of asynchronization resulted by increasing TPR can increase the survival rate.

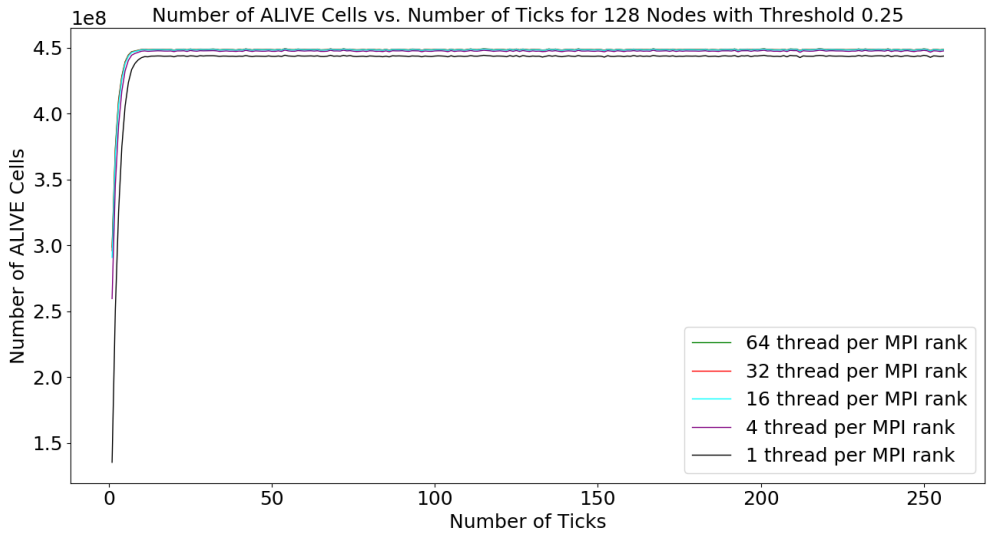

Figure 1: The number of ALIVE cells (Y-axis) as a function of the tick number (X-axis) for 5 experiments in the 128 node and 25% threshold case. Each graph line is for a different value of threads per MPI rank.

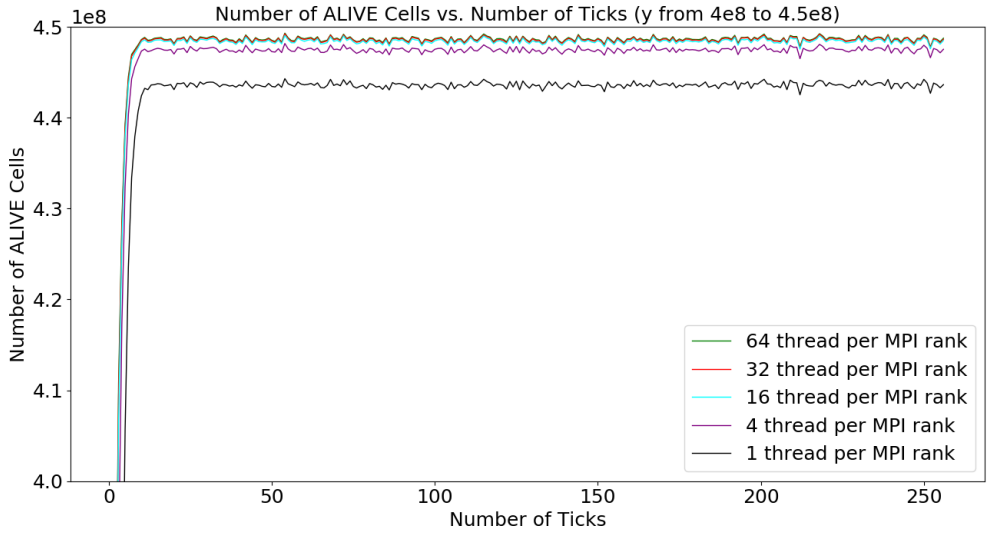

Figure 2: The number of ALIVE cells (Y-axis) as a function of the tick number (X-axis) for 5 experiments in the 128 node and 25% threshold case for Y = 4e8 - 4.5e8. Each graph line is for a different value of threads per MPI rank.

1.2 Compute execution time and speedup

According to Figure 3, in general, when the total number of MPI ranks and threads remains the same but the threads per MPI rank increases, the compute execution time increases since more overheads (e.g. pthread_create(), pthread_join(), pthread_barrier(), etc.) occur. For each single graph line (i.e. with the same threads per MPI rank value), the compute execution time decreases when we use more compute nodes

which leads to more total number of MPI ranks and threads. In general, when we double the number of compute nodes used, the compute execution time tends to be reduced to approximately half. The reason is that more work is done parallel and thus the resource contention is reduced. This achieves the strong scaling since we compute a problem with the fixed size N times faster, where N is the number of processors applied.

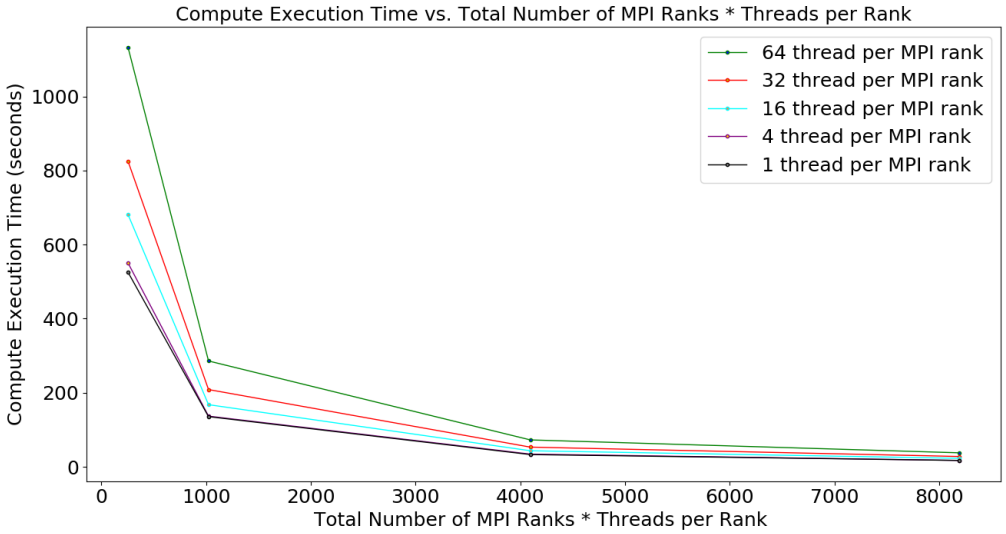

Figure 3: The compute execution time (Y-axis) as a function of the total number of MPI ranks * threads per rank (X-axis) for 20 experiments. Each graph line is for a different value of threads per MPI rank.

We compute the speedup in two scenarios. First, in Figure 4, we compute the speedup using the formula (the compute execution time of x threads per rank using 4 compute nodes)/(the compute execution time of the same x threads per rank using 4/16/64/128 compute nodes). So we get 5 lines for each threads per MPI rank value and each line has 4 points for 4, 16, 64, and 128 compute nodes. In this figure, for the same number of MPI ranks, 4 threads per MPI rank always work the best; for the same threads per MPI rank value, the max speedup is obtained when we use more processors.

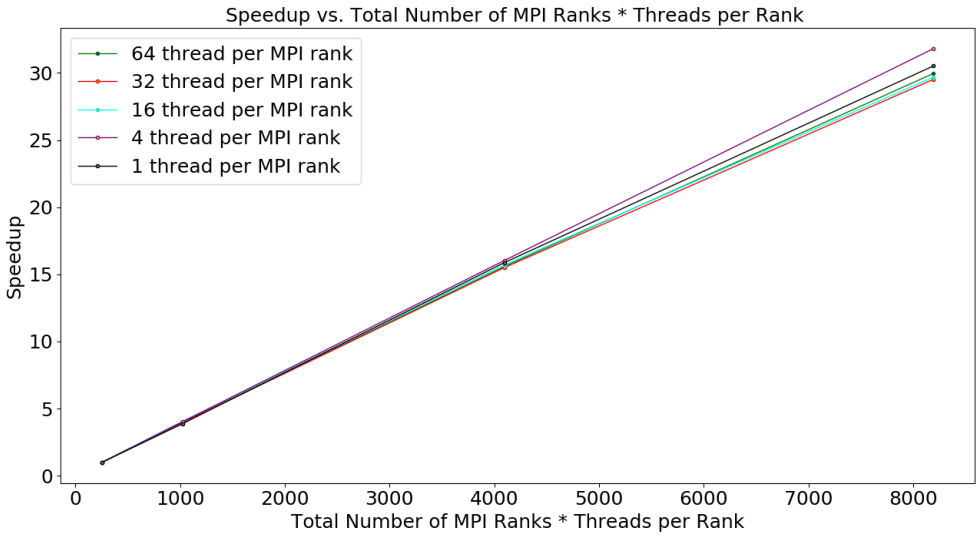

Figure 4: The speedup of execution time for 16 , 64, and 128 compute nodes relative to the execution time for 4 compute node. Each graph line is for a different value of threads per MPI rank.

Second, in Figure 5, we compute the speedup using the formula (the compute execution time of 64 threads per MPI rank using 4 compute nodes)/(the compute execution time of any one experiment). The points are still grouped based on the same threads per MPI rank value but this time all speedup are relative to the experiment with longest compute execution time. In this figure, for the same number of MPI ranks, 1 thread per MPI rank always works the best; for the same threads per MPI rank value, the max speedup is obtained when we use more processors. According to the Figure 6, the greatest parallel efficiency is achieved by 1 thread per MPI rank when we use the same number of processors; and among them, 4 compute nodes has the greatest parallel efficiency. The reason is that with fewer threads, less threads communication is needed and the time that spends on waiting for other threads to finish their work is reduced.

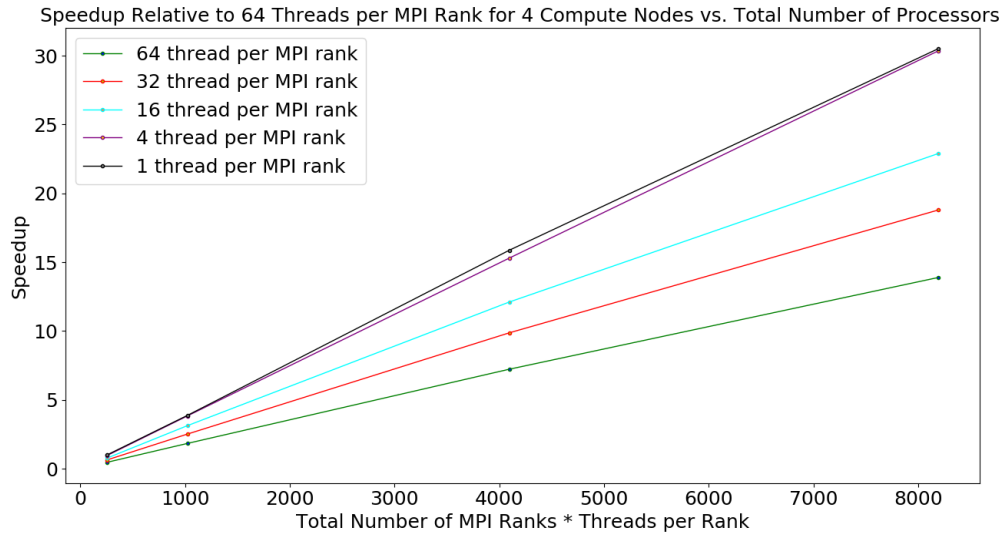

Figure 5: The speedup of execution time for all experiments relative to the execution time for 64 threads per MPI rank using 4 compute node. Each graph line is for a different value of threads per MPI rank.

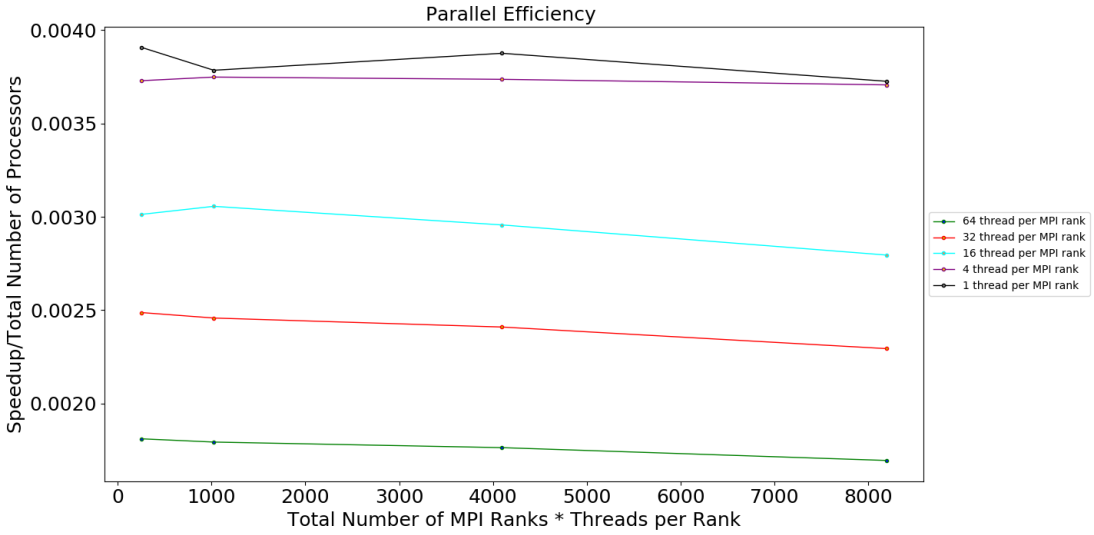

Figure 6: Parallel Efficiency using the Speedup from Figure 5

1.3 Parallel I/O time

Function MPI_File_write_at is applied to parallel write the whole universe to file. Each rank writes $chunk\_num = 32768 \times 32768/num\ of\ ranks$ with offset $mpi\_rank\_index \times chunk\_num \times sizeof(int)$. We set a fixed number of nodes as 128, and a fixed randomization threshold as 25%. We apply threads_per_rank of 1, 4, 16, 32, and 64 which corresponds to total number of ranks of 8192, 2048, 512, 256, and 128 respectively. The plot of I/O execution time versus total number of ranks is shown in Figure 7. I/O execution time increases with increase of number of total MPI ranks. Although the time consumed by writing operation should decrease

with increase of MPI ranks, the synchronization takes more time with higher ranks. Thus there is a tradeoff between these two aspects. For 128 nodes, as it contains many MPI ranks (starting from 128 ranks, up to 8192 ranks), time consumed by synchronization dominate, which generates the trend in Figure 7.

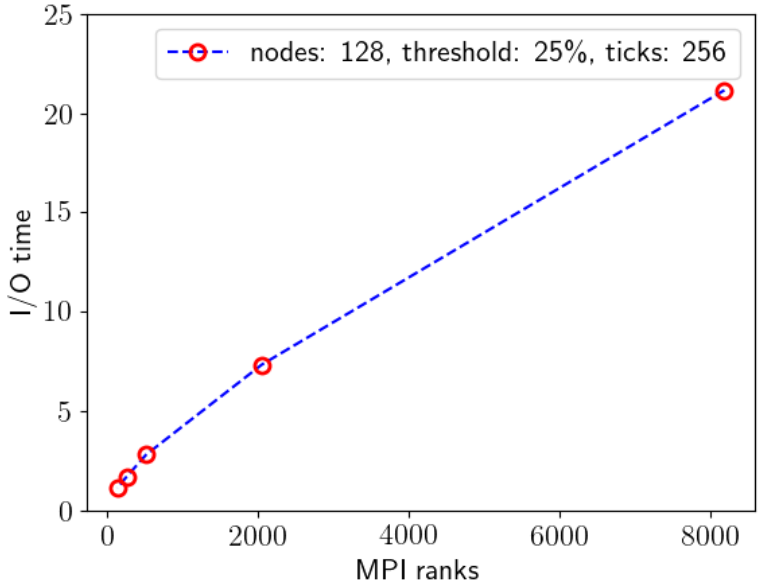

Figure 7: The parallel I/O execution time (Y-axis) to write the whole 32K×32K universe as a function of the total number of MPI ranks used in the 128 compute node.

2 Parallel I/O and Heatmap of final universe state

2.1 Approach for heatmap construction

We constructed heatmap by following the three steps below,

(a) **Read data and construct heatmap grid by MPI parallel I/O**: As each single element of the heatmap accumulates all alive cells in a grid of 32×32, we apply 1024 total MPI ranks (16 nodes with ntasks-per-node of 64) to process the whole universe. As shown in Figure 8, with this setup, each MPI rank deals with 32 lines in the whole universe, counts number of alive cells in each 32×32 grid, and at last contributes to one line in heatmap graphic. The advantage of this setup is that there is no need to exchange data (e.g. using MPI_Isend/Irecv) among different MPI ranks, thus generating better parallelism.

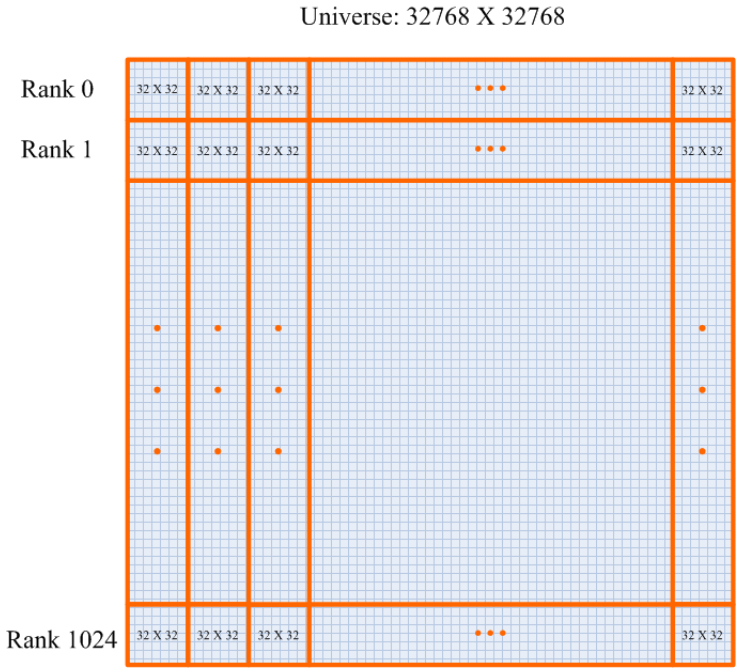

Figure 8: Heatmap construction by 1024 MPI ranks

(b) **Convert binary to text file**: The file obtained from step (a) is a binary file, which can not be processed by gnuplot. Thus we use c function **fread** to read the data in, and then apply **fprintf** to write the data to a text file.

(c) **Draw heatmap by Gnuplot**: As we can not generate picture in Blue/Gene Q, thus we use **scp** to copy the text file generated in step (b) to the local computer and then draw the heatmap graphic. In c file, function **system** is applied to call **gnuplot** command, that is,

```
system( "gnuplot -p 'draw_heatmap.gp'") ;
```

Here "draw_heatmap.gp" is the file for the gnuplot script. The main gnuplot function we used is **splot with image**. A png file is generated by command **set termial png**.

2.2 Experimental results

We used the second configuration (4 threads per MPI rank, 16 MPI ranks per node) to conduct the experiments below. The final universe state after executing 128 ticks with randomization threshold 0%, 25%, 50%, and 75% are shown in Figure 9, 10, 11, and 12 respectively.

When randomization threshold is set to be 0%, which means that the experiment follows basic rules strictly, the number of alive cells in each heatmap element is 0 except the last column. With basic rules, all the 32768 cells in first row become dead after first tick due to each live cell is surrounded by more than 3 cells. In second row, except the last cell, all the cells die after first generation. This is because all the cells in second row are surrounded by more than 3 alive cells, except last cell is surrounded by exactly three alive cells (the below three cells, as the upper and middle 6 cells has been updated to be dead when reaching the last cell). Every next two rows repeat this process. Thus, after first tick, we should only have the last cell alive in every other row. During tick 2, as all neighbours of the alive cells are dead, the alive cells will die. While the dead cells can't become alive as there are at most 2 alive neighbors.

Thus ideally, all the cells die after second generation and the universe will keep this state with increasing ticks due to no chance for reproduction. In practice, as there are 4 threads in each MPI rank, there is some chance that cells in the boundary rows between every two threads can survive because of the parallel running of these threads. This can be justified by the fact that count becomes 0 after second tick by setting threads per rank as 1.

The number of survived cells increases with arising randomization threshold. The count has a big jump from 0% to 25%, then an increase of around 20 from 25% to 50%, and 50% to 75%. The reason is that as randomization increases, more cells can randomly choose their states without following basic rules, which makes the cells very hard to survive. Suppose we set the threshold as 100%, then there should be about 512 ($32 \times 32/2$) alive cells in each heatmap grid. The heatmap with threshold 75% is very close to this scenario.

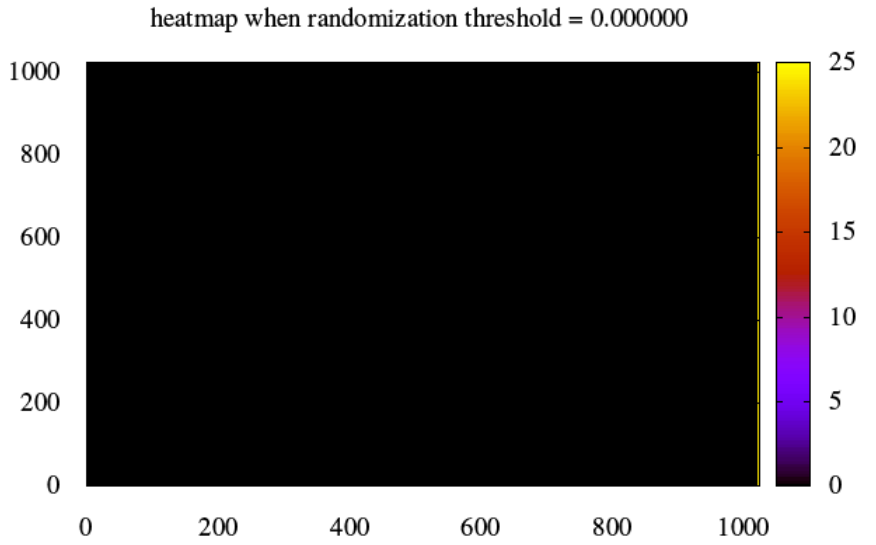

Figure 9: 1K×1K heatmap for the final universe state when threshold is 0%

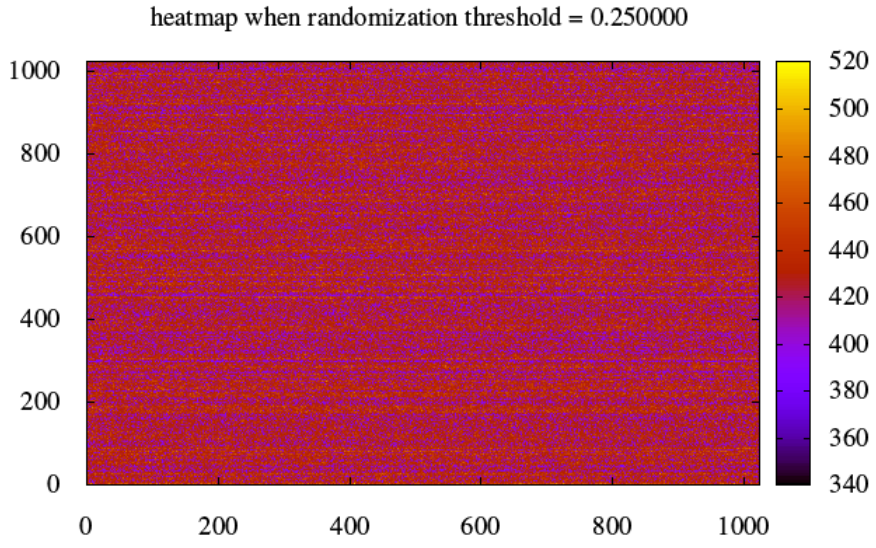

Figure 10: 1K×1K heatmap for the final universe state when threshold is 25%

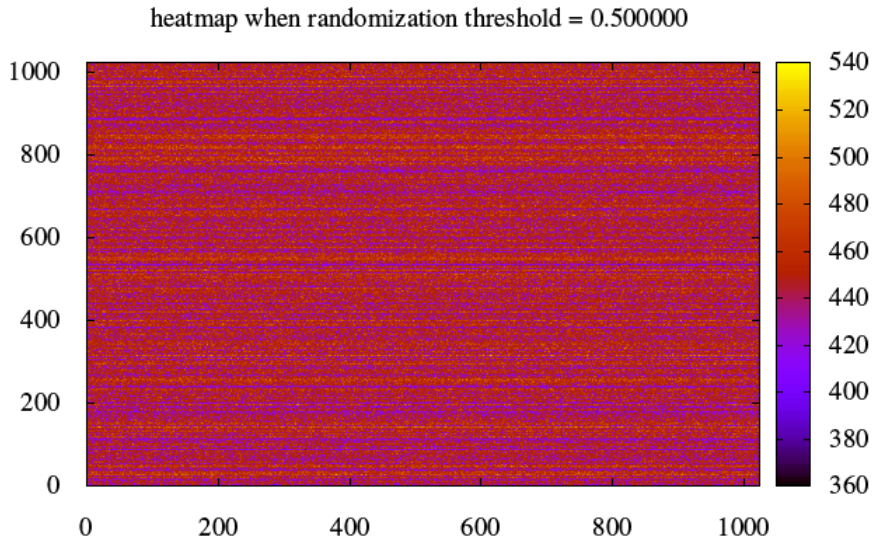

Figure 11: 1K×1K heatmap for the final universe state when threshold is 50%

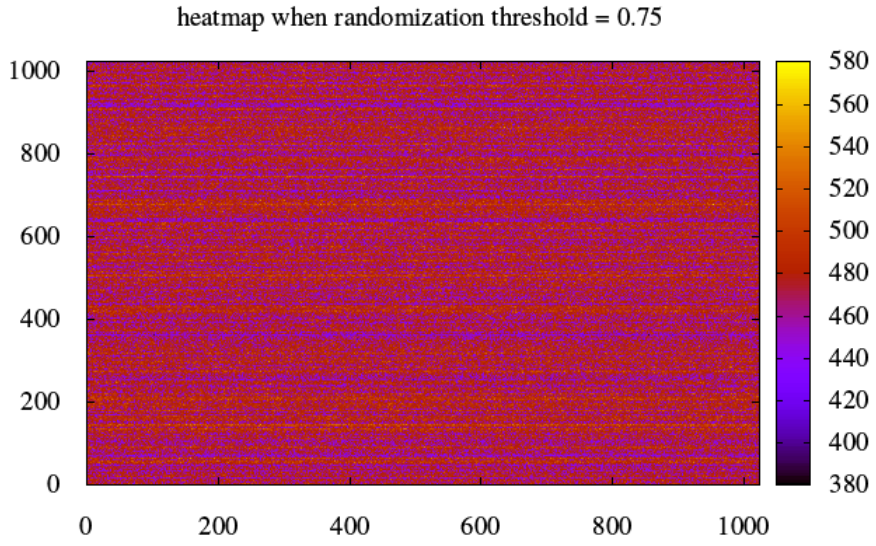

Figure 12: 1K×1K heatmap for the final universe state when threshold is 75%