CITS5507 HIGH PERFORMANCE COMPUTING
PROJECT 2

LATTICE PERCOLATION WITH OPENMP AND MPI

Jasper Paterson 22736341

1.0 Introduction

In this project I simulate percolation through a variable-length square lattice, and report on:

- The number of clusters
- The maximum cluster size
- Whether any cluster spans an entire row or column of the lattice
- The time taken for the simulation

The sites in a lattice are connected either by site occupation or bond occupation between sites. All experimentation was done using site occupation.

This project builds on project 1 by utilising multiple nodes as well as multiple threads during percolation.

The main program parameters are:

- The length of once side of the square lattice, N
- The probability of a site or bond occupation, P
- The number of nodes of utilise, N_NODES
- The number of threads to utilise per node, N_THREADS

2.0 Algorithm

In this section I will describe how the parallel algorithm differs from the first project to incorporate multiple nodes.

2.1 Initialisation

The site or bond occupations are assigned to the lattice by the master process and sent to workers as a short array. The master then becomes a worker. The array is split along rows and assigned to the workers.

2.2 Percolation

Each worker percolates within their process boundaries. They further divide their portion of the array along rows for each thread. Only clusters that reside on a thread boundary are retained in memory, otherwise the cluster's properties are recorded and it is freed from memory.

Clusters are no longer described by a struct, rather by a contiguous array of 2+2N integers: a unique identifier equal to the index of its first site, its size, N booleans that are true if the cluster resides on a lattice row, and N booleans for the columns. This reduces the readability

of the code, but makes creating a sending contiguous blocks of integers over MPI much simpler.

The cluster is searched depth-first as in the first project.

2.3 Joining clusters

The clusters that lie on a thread border are joined within each process. The total number of clusters over all threads is decremented for each join that takes place. The maximum cluster size and percolation booleans are also updated.

2.4 Sending/receiving clusters

Each worker packs only its clusters that lie on the process boundaries into a contiguous array of integers. The workers first send an array of size 4+2N. The first four integer are the total number of clusters for that process, the number of boundary clusters that will be sent, the maximum cluster size and column percolation boolean. The next 2N integers are the unique cluster identifiers of each boundary site along the worker's two boundaries, or -1 if there is no cluster at that site.

Then, the master allocates space for and receives the boundary clusters of all workers. The final task is to loop through all process boundary sites, and, using the size 2N array of cluster identifiers, point each cluster pointer to its corresponding boundary cluster that was received.

2.5 Joining clusters and finalisation

Now that the master's site array contains correct cluster information along each process boundary, the same join function that was used to merge threads can be reused to merge clusters along the process boundaries. Again, the total number of clusters is decremented at each join.

A final iteration though the process boundary clusters finds the global maximum cluster size and whether row and column percolation occurs.

3.0 Speed-up techniques

I improved the memory requirement of my first project by freeing the memory of any cluster that does not lie on a thread border during percolation. I record its size and whether it percolates as soon as the cluster has been searched, instead of storing every cluster and inspecting their properties at the end of the pipeline.

Similarly, I only send clusters that lie on the process borders to the master process. I also send the number of clusters, max cluster size and percolation booleans that were calculated during each process.

I use two MPI send calls to per process, the first including the number of clusters that will be sent in the second, so that the master process can allocate the minimum amount of memory required.

I switched from using a cluster struct to an array of integers to describe a cluster. This means the cluster information can easily be gathered into a contiguous integer array and sent over MPI. A cluster is described by 2+2N integers: a unique identifier, a size, N booleans that record whether the cluster lies on each row, and N booleans for the columns.

4.0 Experimentation

4.1 Correctness

I created seven lattices and stored them in files with their correct number of clusters and maximum cluster size. Some clusters in the files require multiple joins along thread and process boundaries. I created a script that loops through different values of N_NODES and N_THREADS and ensures all results match the tags in the files.

I also ensured that the same seed value is used within an iteration of the same N, P values and different N_NODES and N_THREADS values. All sets (N_NODES, N_THREADS) should yield the same results for the set (N, P, SEED). I created a script that analyses all results generated and reports on this consistency. All results were consistent.

4.2 Generating results

I varied N, P, N_NODES, N_THREADS as follows

N	500, 1000, … 5000
P	0.0, 0.1, … 1.0
N_NODES	1, 2, 3, 4
N_THREADS	1, 2, 4, 8

I generated around 5000 results per N_NODES for a total of around 20,000 results. I increased N to its maximum value before the program timed-out or ran out of memory, and recorded the results. I also investigated the optimal N_THREADS for N_NODES = 4, N = 5000 and P = 0.3.

I graphed all sets (N_NODES, N_THREADS) for different N and P.

I created a script that calculates the speed-up between two sets (N_NODES, N_THREADS) for a certain (N, P). The speed-up value is equal to M1 / M2 where M1 and M2 are the mean total times of all results for (N, P) and the two sets (N_NODES, N_THREADS).

4.3 Performance graphs

Figure 1-4: Probability P vs mean total time (s)

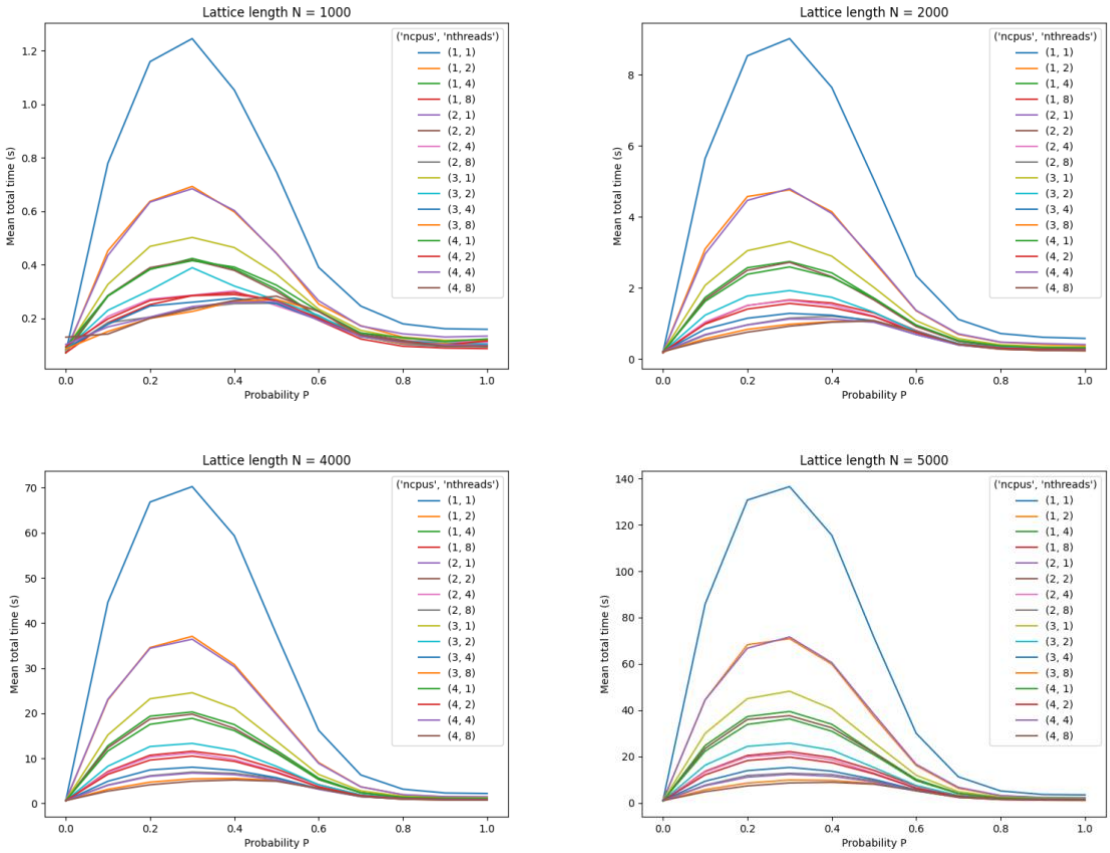
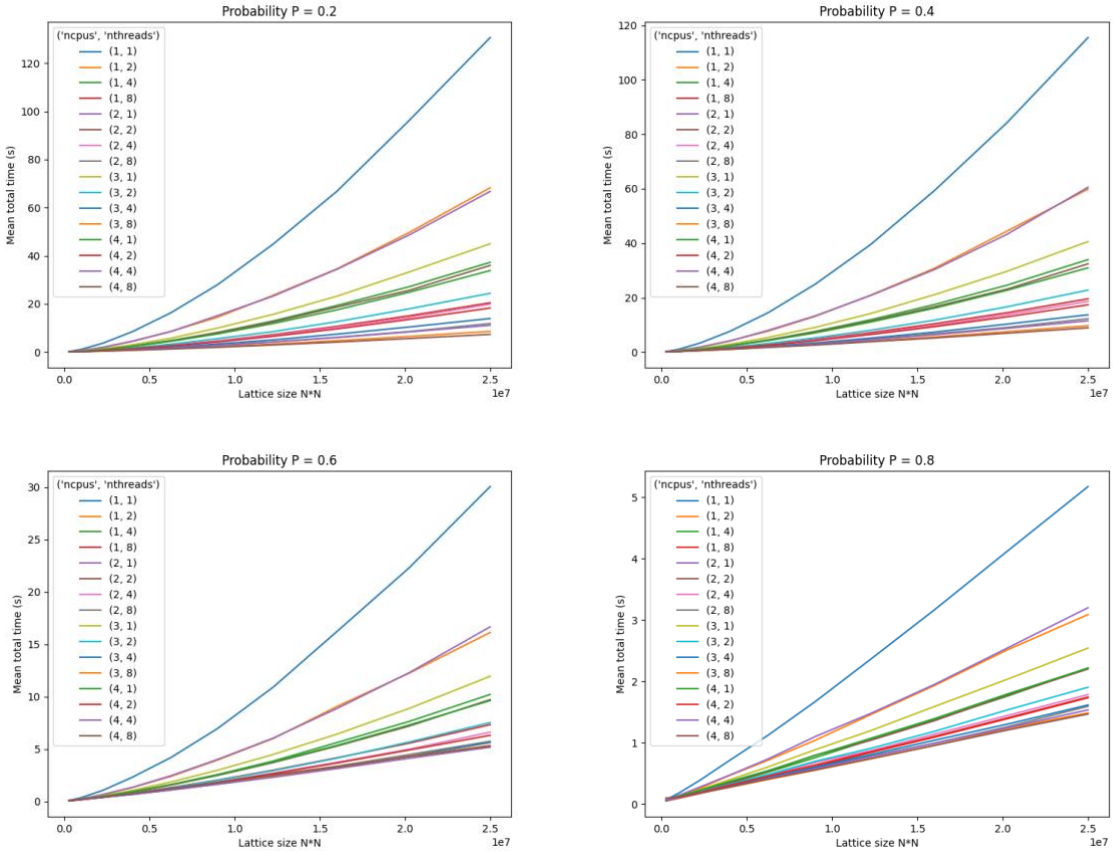


Figure 5-8: Lattice size N*N vs mean total time (s)



4.6 Speed-up tables

Figure 9: Speed-up between (1, 1) and (1, 2), P = 0.3

N	Speed-up
1000	1.797
2000	1.894
3000	1.945
4000	1.896
5000	1.926

Figure 10: Speed-up between (1, 1) and (2, 1), P = 0.3

N	Speed-up
1000	1.820
2000	1.881
3000	1.929
4000	1.929
5000	1.906

Figure 11: Speed-up between (1, 1) and (4, 8), P = 0.3

N	Speed-up
1000	5.297
2000	9.775
3000	12.520
4000	14.229
5000	15.793

4.8 Limit testing

Figure 12: N_THREADS vs mean total time (s) for N = 5000 and P = 0.3

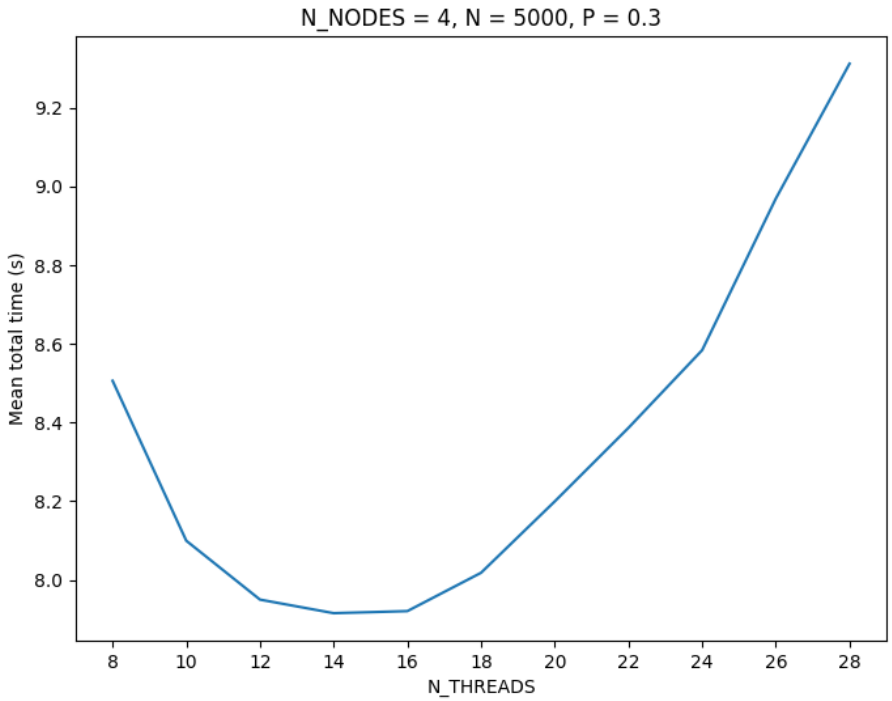


Figure 13: Results for N_NODES = 4, N_THREADS = 28 and P = 0.3

N	Total time (s)
10,000	41.7
15,000	106.4
20,000	213.7
25,000	370.7
27,500	459.2
28,000	487.9
29,000	531.4

Figure 14: Results for large N and P = 0.6

N	Total time (s)
30,000	413.4
32,000	474.5

4.9 Analysis

I observe a significant speed-up for all lattice sizes.

Figures 1-10 all show that at the computation maximum P = 0.3, both 2-node, 1-thread (1, 2) and 1-node, 2-thread (2, 1) achieve a speed-up close to 2 compared to (1, 1), while (4,8) achieves a speed-up of around 15 as lattice size increases.

The purple and orange curves (1, 2) and (2, 1) perform similarly for all N and P, indicating an equivalence between utilising nodes and utilising threads during parallelisation. Performance can be observed to be similar for all curves N_NODES * N_THREADS = k. For example, in figures 1-8, the two red and one pink curve that correspond to k = 8, (1, 8), (2, 4) and (4, 2), all perform similarly.

All graphs indicate that speed-up is proportional to the product N_NODES * N_THREADS. The curve with the highest computing power, (4, 8), performs the best for all N > 1000 and $0.1 < P < 0.9$.

Figure 12 shows that utilising 14 threads on each of 4 nodes is optimal for N = 5000 and P = 0.3. Figures 13 and 14 show the maximum N values that can be computed before time-out or memory runs out for N_NODES = 4, N_THREADS = 28. At P = 0.3 the maximum N value was 29,000 and at P = 0.6 it was 32,000.

5.0 Limitations

All results were for site occupation lattices. It is expected that speed-up would be similar for bond occupation.

All memory allocation was done dynamically. It is possible that performance could be increased if stack memory is used in certain scenarios. For example, clusters could initially be allocated on the stack, and only transferred to the heap if there lie on a thread border.

Worst-case memory allocation was used in many scenarios. Cluster memory was allocated individually for each cluster. With more time, an expression for the estimated number of clusters for a parameter set (N, P) could be found empirically. This could be used to allocate an initial block of memory for clusters, and could be resized if necessary. This would dramatically reduce the number of allocation calls.

For large N, speed-up could be increased further by applying parallelisation to other parts of the pipeline. For example, non-neighbouring lattice regions could be joined in parallel along both thread and process boundaries. The initialisation could be done using multiple threads on the master node, or done separately on each worker node. In the latter case the site or bond occupations can be assigned to the lattice by the worker; the master needs never know the state of the entire lattice. Scanning clusters to find the maximum cluster can also be done in parallel using reduction.

6.0 Conclusion

I developed a parallelised depth-first search algorithm for simulating percolation in a square lattice. I achieved speed-up values of up to 15 using four nodes and eight threads.