CITS5507 HIGH PERFORMANCE COMPUTING
PROJECT 1

LATTICE PERCOLATION IN PARALLEL

Jasper Paterson 22736341
Allen Antony 22706998

1.0 Introduction

In this project we simulate percolation through a variable-length square lattice, and report the key details of the percolation:

- Number of clusters
- Maximum cluster size
- Whether any cluster spans an entire row or column of the lattice
- Time taken for the simulation

The square lattice is comprised of sites. Sites are joined in a cluster in two separate ways:

- Sites percolation: sites can have a boolean occupancy value and are joined if two neighbours are both occupied.
- Bond percolation: The four bonds between each site and its neighbours can take a boolean value, and sites are joined if the bond between them is true.

The main program parameters that we will refer to are:

- Length of one side of the square lattice, N
- Probability of a site occupation or of a bond between sites, P
- Number of threads to utilise, N_THREADS

2.0 Algorithm

2.1 Sequential algorithm

2.1.1 Initialisation

The first stage of the algorithm is initialising the lattice with the parameters N and P. An array of N*N sites is allocated. In site percolation, each site in the lattice is iterated over and assigned an occupancy boolean with probability P. In bond percolation, the horizontal and vertical bonds between sites are stored in two boolean square arrays, each of size N*N, and also initialised with probability P.

Our program also supports input lattices that are defined in files. In this case the initialisation stage consists of reading the lattice information from either a site or bond lattice file, taking an assumed lattice length N from the command line. This assists in testing correctness.

2.1.2 Percolation

In this stage the simulation of the percolation through the lattice takes place. Every site is iterated over. If a site is unseen and is known to be the start of cluster (if it is occupied in site percolation or has a bond to any neighbour in bond percolation), it is added to a stack data structure. A depth-first search of that cluster begins, wherein the cluster's neighbours are checked, the cluster structure is updated, and the neighbours are added to the stack for the process to repeat.

Each site structure contains a pointer to the cluster structure that it belongs to. The cluster structure contains its size, width, height, and two boolean arrays of size N that record whether the cluster spans each row and column. For every unseen neighbour of a site, the cluster structure is updated by:

- Incrementing its size
- Setting two booleans representing the row and column of that neighbour to true
- Incrementing its width and height if that row or column was previously false

2.1.3 Scanning clusters

When the iteration through each of the N*N sites is complete, each cluster structure will contain all the information needed to report on the percolation through the lattice. In this stage we iterate over each cluster structure to find:

- Number of clusters
- Maximum cluster size
- Whether the width or height of any cluster equals N, and therefore spans all rows or columns

2.2 Parallelised algorithm

2.2.1 Initialisation

Initialisation remains the same for our parallelised implementation, as testing revealed it to be of insignificant time compared to the rest of the program. At very large lattice lengths, parallelisation of the initialisation of the lattice may yield faster performance. It is trivial to assign occupation or bond values to each site in the array in parallel, as this is a completely independent process.

We also add another way of testing correctness: a seed parameter that is used for generating the pseudo-random integer sequence. If the seed option is specified on the command line with a constant seed value, all site occupancies or bonds are identical for the same N and P. Using the same seed with different N_THREADS and the same N and P, the correctness of the parallel program can be tested.

2.2.2 Percolation

The depth-first search through clusters remains the same as in the sequential algorithm, however each thread only searches in its own separate region of the lattice. The lattice is divided into a certain number of rows per thread, ensuring that thread boundaries occur along an entire row and not halfway along a row (for ease of joining clusters). Some threads are allocated N // N_THREADS rows and the rest N // N_THREADS + 1 to ensure work is divided as evenly as possible. The number of sites per thread therefore differs by a maximum of N.

Sites that are connected but lie in different thread regions are considered unconnected in this stage. Each thread performs a completely independent search.

The cluster structures must contain more information in preparation for joining clusters along thread boundaries: a unique identifier, and an array of its member sites that lie on a thread boundary. During percolation, new clusters are initialised with a unique identifier equal to the index of its first member site. During the DFS, if a neighbour lies on a thread boundary, its index is added to the site array of the cluster structure.

At the end of the percolation stage, we still have a collection of cluster structures. The only difference is that some clusters are considered separate when in fact they are connected across a thread boundary.

2.2.3 Joining clusters

In this stage clusters that are connected along a thread boundary are merged. To achieve this, the bottom row of each thread region was arbitrarily chosen to iterate over. For each site along the row, the site beneath it, which lies in the beneath thread region, is analysed.

If there is a connection, and their unique cluster identifiers are different, the clusters are merged. Their sizes are combined, and the row and column arrays of one cluster is filled in with the information of the other, updating the width and height in the process. Then, the boundary sites in the site array of one cluster are redirected to point to the updated cluster and appended to the site array of the updated cluster. This is necessary in case these same sites participate in future joins – they must contain the most current cluster information.

Once the bottom row of each thread region has been iterated over, all clusters are complete, and the cluster array looks much as it would in the sequential program. The only difference is it is padded with now-redundant sub-clusters, which have been marked as obsolete with an identifier of -1 and ignored.

Scanning clusters to find the percolation information can be done in the same way as in the sequential program. This was also of insignificant time compared to percolation and so was not parallelised.

3.0 Speed-up techniques

Here, we detail some of the improvements that were made over the course of the project.

3.1 Distribution of rows

Originally, our algorithm distributed rows to threads in a simple way: every thread got N // N_THREADS rows, while the last thread also received the remaining rows. We bettered performance by improving the distribution of rows so that threads differ by at most one row (N sites).

3.2 Cluster site array

Clusters were initially thought to require an array of all their member sites in preparation for the joining clusters stage. We reduced this requirement to only storing clusters on a thread boundary, as they are the only sites that will participate in cluster joins. This reduced the memory requirement of cluster site arrays from O(N*N) to O(N_THREADS*N), which corresponds to the sites along the two boundary rows of each thread region.

3.3 Scanning clusters

We increased performance by changing the scanning clusters algorithm from checking each of the N*N sites for their cluster structure, to recording a separate array of cluster pointers to be iterated over. This requires more memory, however ensures we only iterate over the number of clusters present, not every site.

4.0 Experimentation

4.1 Generating results

In this stage we created a driver program to run many site percolations with different N, P, and N_THREADS, and a graphing program (in Python) to view the results.

Two tests were completed:

- On a dual-core (four logical cores), i5, 8GB MacBook Pro 2017 we ran ~50,000 percolation simulations.
- On a hexacore (twelve logical cores), i5, 16GB ubuntu laptop we ran ~35,000 percolation simulations.

Hyperthreading remained enabled on both machines.

The parameters were varied by:

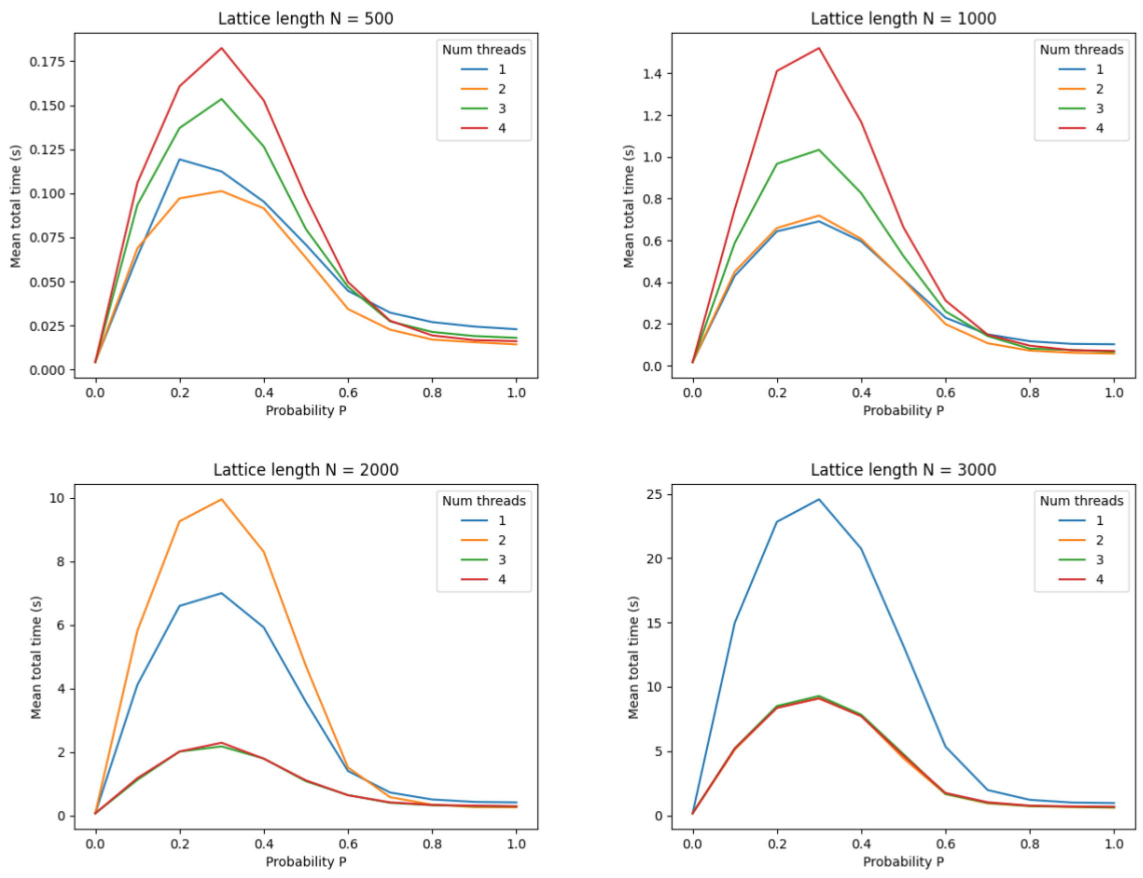
	N	P	N_THREADS
Min	100	0.0	1
Max	3000	1.0	4
Step	100	0.1	1

To graph our results, we kept either N or P as constant, and graphed the other variable against total time for all N_THREADS.

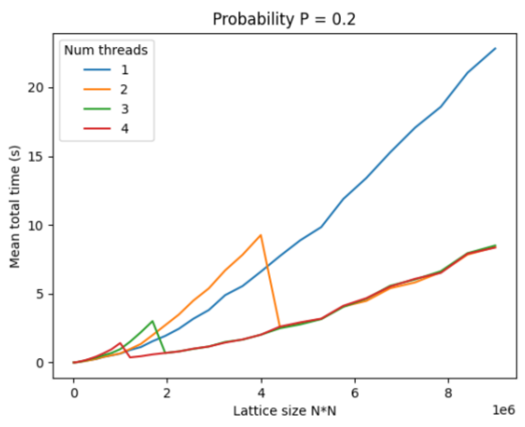
Each data point on a graph is the mean total time of all the results that match the parameter set (N, P, N_THREADS). With ~50,000 results there are ~35 results that comprise a single data point on a graph.

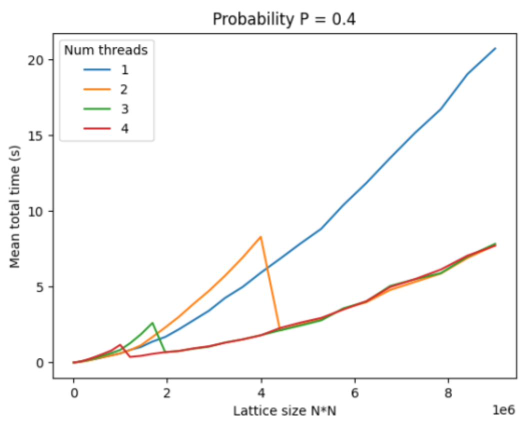
4.2 Graphing dual-core results

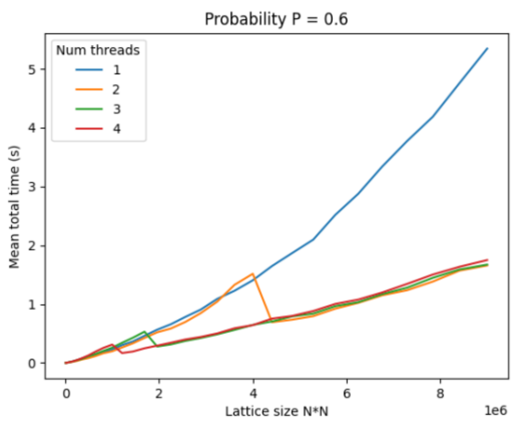
4.2.1 Mean total time (s) vs site occupation probability P

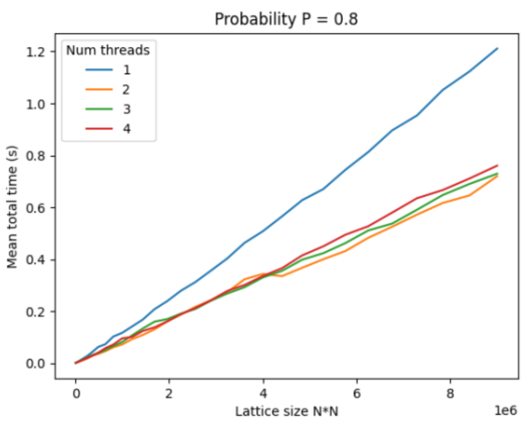


*4.2.2 Mean total time (s) vs lattice size N*N*



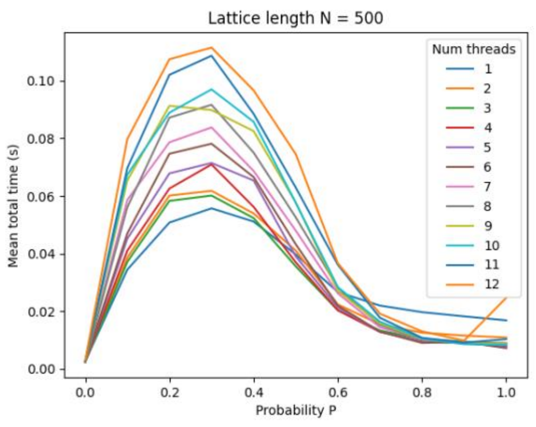


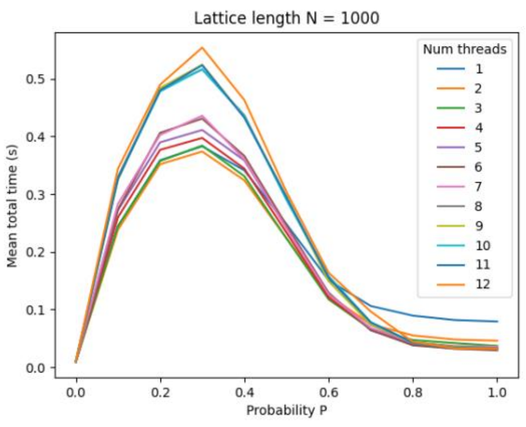




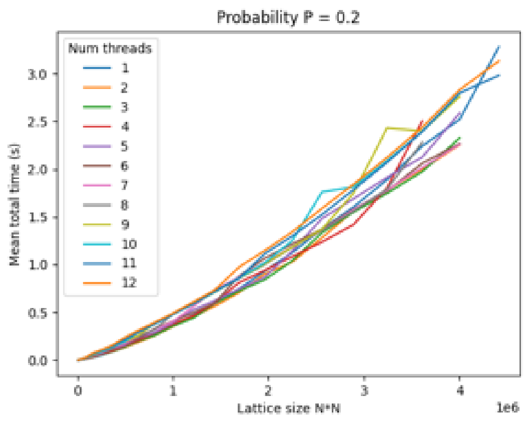
4.3 Graphing hexacore results

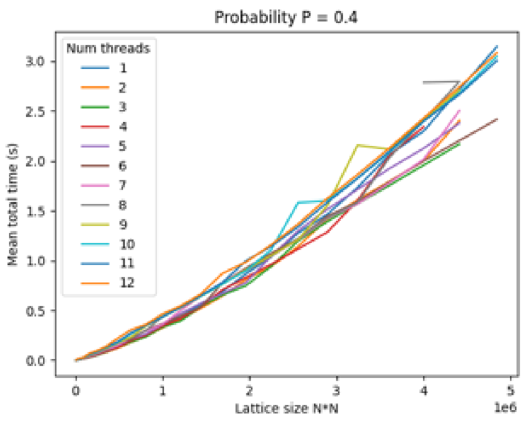
4.3.1 Mean total time (s) vs site occupation probability P

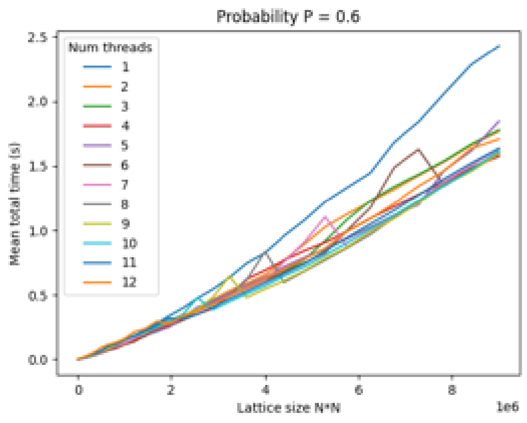


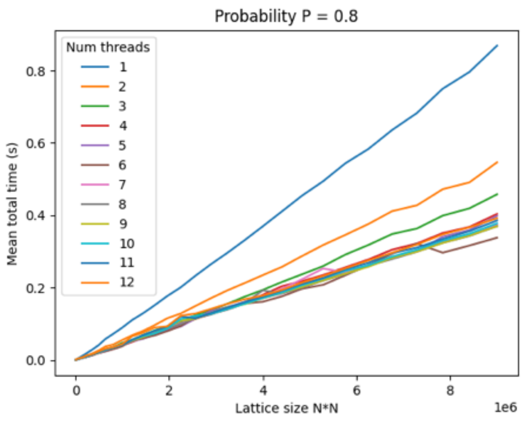


*4.3.2 Mean total time (s) vs lattice size N*N*









4.4 Results analysis

4.4.1 Mean total time (s) vs site occupation probability P

The graphs for constant lattice size form parabolic curves with a local maximum at a probability ~0.3. The curves approach zero at low probabilities and high probabilities.
At low probabilities the execution time is low because percolation skips over a greater number of unoccupied or unbonded sites, resulting in fewer and smaller clusters. As probability nears ~0.3, the number of separate clusters in the lattice is maximised. Cluster initialisation is an expensive operation, requiring three separate memory allocations of O(N) to the heap. This dominates execution time and causes the maximum.

At higher probabilities, a greater proportion of execution time lies within the cheaper DFS operation, as there exist larger but fewer clusters. The number of cluster initialisations decreases, leading to lower execution time. This pattern is observed for all values of N and N_THREADS.

On dual-core, we observe the curves N_THREADS > 1 gradually approaching and overtaking the curve N_THREADS = 1 as the constant N is increased. At N = 3000 there is speed-up factor of around ~2.5 at the maximum between the parallelised and the sequential algorithm.

N_THREADS values between two and four seem to perform similarly for large N. This is likely due to hyperthreading; we can only achieve the equivalent of two-core performance on the dual-core laptop.

*4.4.2 Mean total time (s) vs lattice size N*N*

The graphs for constant site occupation probability show mean total time increasing in proportion to the square of lattice length. At high lattice lengths on dual-core, the curves N_THREADS > 1 converge, with mean total time consistently half of the curve N_THREADS = 1.

At low probabilities P < 0.6, execution time is dominated by cluster initialisation. Since heap memory allocation has non-deterministic complexity, the execution time increases exponentially with lattice length. At high probabilities P > 0.8, we observe that execution time is directly proportional to the lattice size N*N, due to the O(N*N) complexity of the dominating DFS operation.

Utilising up to twelve logical cores on the hexacore machine shows a higher discrimination between the speeds of varying N_THREADS, with all thread numbers experiencing at least a factor of two speed-up. At high probabilities, N_THREADS > 4 converge to a single curve, likely caused by diminishing returns of performance due to L3 cache contention.

On hexacore at probability P = 0.8, when DFS dominates, we see clear speed-up as N_THREADS increases. N_THREADS = 2 achieves speed-up around a factor of two, while increasing N_THREADS improves this further. At probability P = 0.4, there is a noticeable optimal number of threads between three and six. This can be explained by diminishing returns caused by the overhead of maintaining increasing N_THREADS.

On dual-core, sharp decreases in execution time were present for each curve with N_THREADS > 1, after which the curves converged with N_THREADS = 4. Similar behaviour was observed on hexacore. Since execution time at lower site occupation probability is dominated by heap memory allocation, we predict that the unusual overhead was due to parallel memory allocation calls. The reason for the more pronounced behaviour in the dual-core may be slower RAM speeds and lower bandwidth.

5.0 Limitations

We had only four logical threads available to us for easy results generation, and that included hyperthreading. Hexacore results were generated later. Perhaps, on dual-core, curves N_THREADS > 1 converged because we could only get the equivalent of two-core performance out of the dual-core laptop. We could have generated results on Kaya, however it would have been more difficult to generate the same number of results with the one-hour time limit.

We also only generate results for site percolation. There is no indication that shape of any curves would have been different for bond percolation. All testing showed equivalent speed-up and correctness. Perhaps a difference would have been the location of the maximum on the mean total time vs probability graphs.

We employed a policy of worst-case memory allocation with no memory reallocation. With more time we could have found empirically expressions for the estimated cluster size and estimated number of clusters for any parameter set (N, P). With correct estimations, memory allocation would be reduced, and reallocation would be required sparsely.

The join clusters stage could be parallelised by joining non-neighbouring thread regions in parallel. The scan clusters stage can also be parallelised using reduction. Both of these stages were of insignificant time compared to percolation, and so these did not seem important for this project.

Cluster initialisations dominated the execution time as the number of clusters increased. We could have investigated storing clusters on the stack as opposed to the heap to reduce memory allocation time.

We could have further investigated removing hyperthreading to see if speed-up became more pronounced.

6.0 Conclusion

We developed a sequential DFS algorithm to analyse two-dimensional square lattice percolation. We then achieved a speed-up of factor ~2.5 for large cluster lengths $N > 2000$ at the point of interest $P = 0.3$ using a parallelised DFS percolation algorithm on dual-core (four logical cores). We also achieved speed-up for all N at probabilities higher than the critical percolation probability ~0.6, as well as at low enough probabilities $P < 0.1$. Our parallelised DFS algorithm performs well at high percolation probabilities, achieving less speed-up near the local maxima at probability $P = 0.3$. On hexacore, there was an optimal N_THREADS range between three and six across most parameter sets, while on dual-core all N_THREADS > 1 converged to a similar, but significant, speed-up value.