SHPC4002 Project 1

Sam Marsh
21324325

October 7, 2017

1 Compilation and Execution

This project was implemented in the C programming language. All parts of the project were completed. The project can be compiled using the included `Makefile`:

```
cd <project_dir>
make
```

This will produce an executable file called `main`. The program arguments are as follows:

```
usage:  ./main lattice_size seed_prob seed_what percolation_kind [num_threads]
```

where `lattice_size` is a positive integer n for a $n \times n$ square lattice, `seed_prob` is the occupation probability between 0 and 1, `seed_what` is either `s` or `b` (representing site or bond seeding), `percolation_kind` is an integer between 0 and 2 (0 for row-definition of percolation, 1 for column-definition, 2 for both), and `num_threads` is an optional argument which is passed to OpenMP as the number of threads to use (defaults to the number of logical processors). For example,

```
./main 2048 0.6 b 2 4
```

will test for both row and column percolation on a 2048×2048 square lattice, using a 0.6 seeding probability of bonds, and will use four threads to do so.

The same command line arguments apply for both the parallel and sequential programs (but the sequential program will ignore the number of threads argument, if given).

2 Implementation

First, a framework was written for supporting the construction of a lattice (see file `lattice.c` and `lattice.h`). The lattice is made up of a square array of `sites`. A `site` structure keeps track of whether it is occupied, and whether it has a bond to its four neighbouring sites. Convenience functions were written for finding the neighbour of a site in a given direction, assuming the bond exists. Wraparound connections are implemented naturally via the modulo operator – edge sites are treated no differently than any other site. There are two seeding functions – one which seeds the sites, and one which seeds the bonds. Seeding sites works by setting each site as occupied if a generated random number is less than the occupation probability. A second pass is then done through the array, this time setting bonds between neighbouring sites if they are both occupied. For bond seeding, instead the unique bonds are iterated through: for each site, the north and east directions are considered: if a generated random number is less than the occupation probability, the bond is enabled between the two sites, and both sites are set to occupied. Apart from the seeding, bond and site percolation detection works the same way (no bond or site-specific code is written), it is all done using bonds – with site percolation, the bonds are automatically filled between adjacent sites.

Next, a stack implementation was required to support the depth first search which is used to find clusters. Recursion could have been used, using the process stack for depth first search, however for larger lattices with high density (high occupation probability) a large number of sites may be on the stack and so this may cause an overflow. A simple array in memory was chosen as the 'backing' data structure for the stack (see `stack.c` and `stack.h`) over a linked list, due to the potential for some of the array to be kept in the processor cache – leading to extremely fast pops and pushes. A (dynamic length) linked list has the advantage in space used, since the array needs to in the worst case hold all n^2 elements for a n by n lattice, but speed is more important in this case. Pops, pushes, and empty checks are implemented through the array and a `size` variable.

The initial sequential implementation to check for percolation was relatively simple (see `percolation.c`). A boolean `visited` matrix of the same size as the lattice is created. Then each of the lattice sites is iterated over. For a particular starting site which is occupied and not yet visited, allocate two `bool` arrays of size n to keep track of if each row and column have been reached in the current cluster. Then a depth first search is performed from this starting site. For each site (i, j) visited, increment the current cluster size by one, and mark the ith row and the jth column as reached. The unvisited neighbours of the current site are then pushed onto the stack. Once the depth first search from the initial site is complete (i.e. when the stack is empty), if row/column percolation exists (checking if all of the `bool`s in the row and column arrays are `true`) update row/column percolation variables. Update the current maximum cluster size if needed, and move on to the next occupied and unvisited site. Once all sites have been processed, the maximum cluster size and whether the lattice percolates has been found.

The parallel implementation, while sharing the foundations of the initial sequential version, requires a lot more care. The main aim is to split up the lattice into smaller 'boxes' (slices), which then can be processed in parallel. The challenging part comes in patching the data together from these sub-lattices, where clusters may span over multiple boxes. The boxes were originally made to be square (e.g. a 2×2 overlay of square boxes over the lattice), but after an updated suggestion made in lectures, they were modified to be 'slices' of width n.

In order to accommodate a parallel version, new functions and structures were introduced into `lattice.c` and `lattice.h`. In particular a `cluster` structure was created. Each cluster has a unique integer identifier (unique even amongst different boxes). A cluster also keeps track of whether it is `global`, that is, spanning multiple boxes – and if so, the sites on its border which connect to clusters in other boxes. A cluster also tracks the columns and rows that it spans, similarly to the sequential version. Finally, a cluster has the ability to point to another cluster: the `redirect` variable has the purpose of allowing clusters from different boxes to be merged together in the 'patching' stage. Indeed, functions were also created to facilitate this merging: the `canonical` function follows a chain of redirects to find the cluster which labels the rest of the global clusters, and the `merge_clusters` function (1) redirects a cluster to another cluster (2) updates the size, row span and column span of the cluster which is being merged into. A simple `box` structure was also implemented, just holding the bounds of a box (min i, min j, max i, max j).

In the parallel implementation (`percolation.c`), the sequential code was first modified to support only exploring within a 'box'. That is, if a site is on the border of a box and has a bond leading out of the box, that neighbour is not pushed onto the stack to be processed. Rather, if this is the case, the site is added to the list of `cluster` border sites, and the cluster is marked as global. If the cluster leads out of the north border, it is recorded – the south ones can be ignored, since the south border of once slice is the north border of the one below it. If after the depth first search the cluster is not found to be global, it is discarded (only the size of the cluster is kept track of). This is because it cannot be a spanning cluster if it is fully internal to the box. After each site in the box is iterated through, there remains a list of global clusters for the box. This list of north global clusters is returned to the caller.

The processing of these boxes is completely independent, and this is where the parallel speedup comes from. The number of boxes used is chosen as the number of threads. If the lattice size, n, is not evenly divisible by this 'overlay box grid' size, then extra rows are added to the boxes as needed, so that all rows of the lattice are covered and work is as evenly distributed as possible. Each box identifier is iterated over by a single thread:

```
#pragma omp parallel
#pragma omp single
for (int id = 0; i < n_boxes; ++i)
```

and each box is processed through the OpenMP tasking framework, using

```
#pragma omp task firstprivate(id)
//find clusters for box, see percolation.c
```

Using the tasking framework was found to be faster than parallel processing of the `for` loop. However, at first there was no speedup whatsoever, because tasks were being executed sequentially by OpenMP even with multiple threads requested. This turned out to be because the task was 'tied' due to some local variables for calculating the box sizes (taking into account odd divisions into the lattice size) which were updated by the main thread each loop iteration. In order to fix this tying, the box information (width, height, starting cluster label, position) was pre-computed in a prior loop. Due to the size of the overlay lattice being (for example) four boxes with four threads, the overhead that this additional loop added was miniscule, and the tasks became untied, leading to parallel execution of tasks and a significant speedup.

Once all the global clusters have been found, the patching step is then required to aggregate results. The global cluster border sites are iterated over, and the associated neighbouring cluster is found. These two clusters are then merged, as per the description given previously. Note that since bonds are symmetric, only the north border sites need to be considered – the south border sites of one cluster are connected to the north of the other cluster.

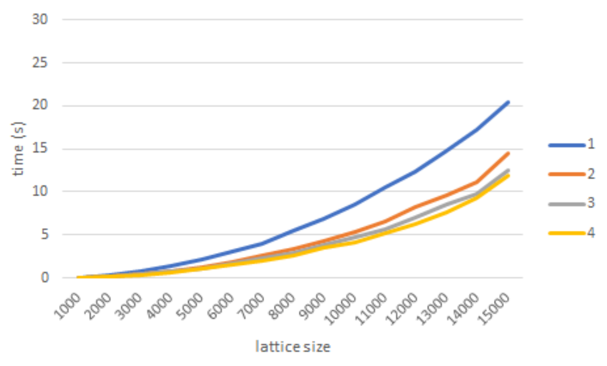
A final iteration is then performed over the *canonical* global clusters to check if any percolate (`row`/`col` array all `true`) and to update the maximum cluster size. This step is done in parallel, since it only involves a `max` reduction and updating of two boolean variables from `false` to `true` if needed. The OpenMP pragma line used is

```
#pragma omp parallel for reduction(max:full_max)
```

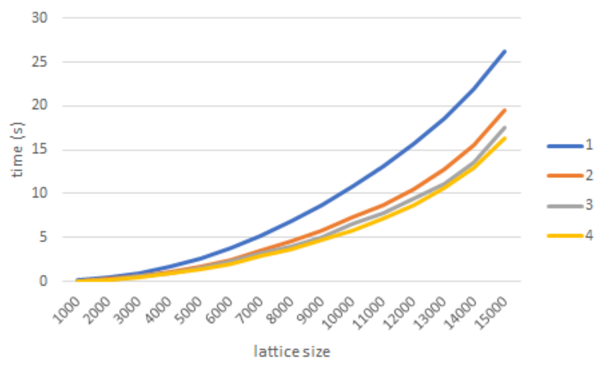
In the parallel program, I did not parallelise the seeding/generation of the lattice, since the aim was to study speed-up of the percolation checking algorithm. I also only time the percolation checking, not the lattice generation part of the program.

3 Correctness

The parallel program gives the same percolation thresholds as the sequential program – that is, for large lattices ($\gtrsim 2048$) both show site percolation for $p > 0.592746$ and don't when $p < 0.592746$. The same is



(a)



(b)

Figure 1: Average execution times for seed percolation (Fig. 1a) and bond percolation (Fig. 1b), using 1-4 threads on various lattice sizes. Note that the single-thread execution times are found using the sequential program, not the parallel program.

true for bond percolation with p close to 0.5. The cluster sizes also match as well (within a reasonable range accounting for random variation).

4 Exploration

The final algorithm described above gave the best parallel performance by far, in comparison to other methods that were explored. Other strategies experimented with were:

- Parallelisation of the loop over the lattice, i.e. running depth first searches from different sites. However, this requires a lot of synchronisation in terms of ensuring that two (or more) threads don't visit the same site at the same time (race conditions). Due to the amount of synchronisation required, and the difficulty of combining results if multiple threads turn out to be processing the same cluster, this method was discarded.
- The Hoshen-Kopelman algorithm[1]. This is a simple raster scan of the lattice, combined with a similar method as used in Section 2 to merge clusters. It has the potential to be made parallel through simpling parallelising the scan through the lattice. However, the depth first search method works just as well for finding clusters and requires less merging (within a particular box, depth first search doesn't require merging of internal clusters, whereas this method does).
- Parallel speedup of the 'patching' process was also investigated. However, it was found that due to the level of synchronisation required when merging (redirecting) clusters, parallelisation resulted in worse performance than the sequential version.

5 Results

The OpenMP function `omp_get_wtime()` is used to measure the wall-clock execution time of the program. Execution times were recorded on a **2.9 GHz Intel Core i5 2016 MacBook Pro**, with two physical cores

[1]`https://en.wikipedia.org/wiki/Hoshen%E2%80%93Kopelman_algorithm`

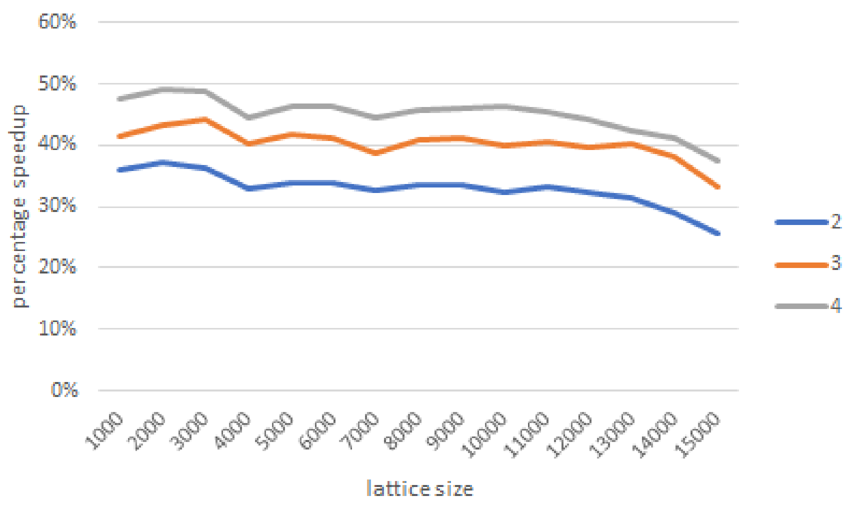


Figure 2: The percentage speedup over the sequential program using different numbers of threads.

and hyperthreading enabled $\implies$ four logical cores.

Timing was performed on the percolation thresholds: for site percolation using 0.592746 and for bond percolation using 0.5. A bash script `time-it.sh` is included with the submission of this project. This was the script used to determine the run-time of the program with various lattice sizes and numbers of threads. It is executed using `./time-it.sh s/b` to time with seed/bond percolation respectively. Each execution time is averaged over five tests. Note that the single-thread execution times are found using the sequential program, not the parallel program.

Refer to Fig. 1 for execution times as the lattice size increases.

The final parallel implementation as described above gives a notable speedup over the initial sequential version, as per Fig. 1. Moving from one to two threads results in a reasonable increase in performance. Unsurprisingly on the dual-core machine (with hyper-threading), additional threads result in less significant improvement – but with four threads, the time taken to execute the program is noticeably less than for the sequential.

The percentage speedup (given in Fig. 2) is just under 50% for on the dual-core machine with hyper-threading, using the default number of threads, equal to the number of logical cores. The percentage speedup is fairly steady, but starts to decrease at around $n = 10000$. Using four threads, the speedup decreases from 48% for $n = 1000$, to 37% for $n = 15000$. I wasn't able to find a certain cause for this, but I suspect it may be an issue with the large number of clusters stored in memory – possibly temporarily overflowing into virtual memory on the hard drive? Alternatively, it could be because of the large number of parallel allocations and de-allocations of cluster data – the program's heap could be becoming fragmented, making it slower for the OS to allocate memory for new clusters etc.

This method of parallel percolation should lend itself naturally to the second project. Using MPI, the sub-lattices could be sent to separate processes. Then once found, the global clusters could be sent to neighbouring boxes to patch together.