CITS3402 High Performance Computing

Project 1:

2D Percolation of a Square Lattice

By:

Belinda Edmunds | 19422753

Matthew Fitzpatrick | 21297239

# Introduction

This report will outline the methodology, analysis and results found from creating a 2D lattice percolation algorithm through either bond or site seeding. The results from a sequential algorithm will be compared with parallel programming by introducing threads to perform parallel work on the algorithm.

# Usage

In order to compile the program, the following command line usage should be employed.

For the c file which contains the threading components:  
gcc -fopenmp -o t threading1.c

To run the resultant file, the following data should then be entered:

./t seedingType probabilility

where seedingType should be submitted as either ‘s’ or ‘b’, which represents either seeding at the node sites or bonds respectively. The probability entered should be a floating point number between 0 and 1. For example, to generate a cluster search with a probability of 0.55 with seeding at the node sites, enter “./t s 0.55”.

For the c file containing the sequential only algorithm, enter the following into the command line:

gcc -fopenmp -o s sequential1.c

The resultant file can then be run similarly to the threading file:

./t seedingType probabilility

In order to alter the size of the lattice grid, the value L should be modified within the c files on line 16. L represents the edge length of a single map, which is half the total lattice dimension. Therefore, in order to create a lattice of 32 x 32 for example, L should be changed to 16.

# Methodology

It was assumed that the lattice should be divided into 4 equal sections or ‘maps’, in order to allow for ease of introducing parallel threading after the sequential algorithm was completed. An edge length of ‘L’ is defined, and each of the 4 maps has a side length of L x L.

## Seeding

Initially, the seeding of the lattice is performed, which depends on the user input of ‘s’ for site percolation or ‘b’ for bond percolation. A struct developed for the map allows for initialisation and seeding of a flag (0 or 1 depending on whether the node is seeded); up, down, left and right for progress of the bonds through the cluster search process; and x and y values to allow for transferral of map coordinates.

## Site Percolation

A random probability is generated, and depending on the user input for probability threshold, the flag for each node is set at 1 if the random probability is below the user input probability, or 0 if above. Each bond within the LxL map is set at 0, and the outside edges set at ‘2’ in order to limit the cluster search to the extremities of each map. 2 denotes an exhausted search parameter.

## Bond Percolation

For bond percolation, a randomly determined probability is applied to each bond, whereby a ‘2’ is applied to the up, down, left or right struct values for bonds which receive a probability value above the user input probability, meaning that these bonds will not be included in the cluster search. All other bonds retain an initialised value of 0, which then have adjacent nodes initialised to a ‘1’, or seeded, value.

## Depth First Search

A depth first search (DFS) is subsequently conducted on each individual map. If a node has a flag value of 1, a DFS will commence by checking adjacent flags; if the flag is also seeded and the bond has a value of 0 (unsearched), or 1 (searched but not exhausted), then the search can continue. Each iteration of the depth first search adds a counter to the cluster size and records which direction the previous node was, in order to allow the DFS to backtrack when searching is exhausted. If the DFS reaches a dead end, all immediately adjacent bonds are allocated a ‘2’, which means they will excluded from further searching. The clusterID is recorded, and each separate cluster given a separate ID, which allows recording of the cluster size and the location of any cluster nodes located the edge of the individual map.

The DFS function has several helper functions, including push(), pop(), peek(), deadEnd(), getNextNode() and isStackEmpty(). The latter is the prerequisite for continuing the individual cluster DFS. Once the stack is empty, the DFS exits and the clusterID count increases and searches for the next seeded node not included in an existing cluster.

## Cluster combinations

Once the DFS is completed across the entire map, the cluster edge node locations are transferred into a struct with their clusterID number and corresponding size.

## Parallelisation

# Results

# Discussion and Conclusion