The Par-A-MAZE-ing Labyrinth

CIS*3090 Parallel Programming
Group Report

By: Richard Break & James Kelly
Professor: William Gardner

Table of Contents

1. Introduction & Project Overview
2. Goals
3. Maze Generating Algorithm
4. Fitness Functions - Shortest Path Algorithm
5. Fitness Functions - The Primary Reconvergence Algorithm
6. Generations, Mutations, and Crossovers
7. Working with Java
8. Parallelizing The Maze
9. User Guide
10. Results
11. Experience
12. Responsibilities / Expected Grade
13. Sample Maze

1. Introduction & Project Overview

The foundation of this project is to create a genetic algorithm that generate mazes and then defines their values using different fitness functions to determine what kind of maze has been generated. Our program randomly generates the mazes and then applies fitness functions such as the shortest path to find out information about the mazes.

Our maze generating algorithm is based off of *Search-Based Procedural Generation of Maze-Like Levels*, a paper by Professor Daniel Ashlock, which provides theoretical advice on proper maze generating techniques. We are following his section on positive, indirect representation in which we use a linear integer array to fill out and create the walls inside our maze.

This project is written in java and accepts several command line arguments from the user to help determine what kind of maze the user wants to generate including the dimensions, the number of processes to use, and if the user wishes to include checkpoints inside the maze.

2. Goals

From our research, a maze that is 50x50 requires several minutes for fitness functions to finish. Our goal is to decrease the amount of time required to generate mazes and run these fitness functions by implementing parallel programming into the process. We are looking for the amount of time we can save by parallelizing the process as well as hoping to achieve a speedup of 50%.

A secondary goal is to compare parallelizing the whole algorithm vs. parallelizing the fitness function itself. To elaborate, we want to see if having every individual thread being tasked with both creating a maze and running fitness functions for on the maze will be faster than creating the same number of mazes serially and then parallelizing the fitness functions being used on each maze.

3. Maze Generating Algorithm

The maze generating algorithm we use is based on the positive, indirect representation section of Daniel Ashlock's paper "*Search-Based Procedural Generation of Maze-Like Levels*." Our maze is generating by using a linear array 80 integer in a range between 0 to 65535. The integers are used in pairs to create and fill out the layout of the maze.
The first integer of a pair is bit sliced into three values:

- The first value is a one bit number that acts as a flag to whether the line is able to pass through other lines in the maze. If the value is 1 it can pass through other walls. If the value is 0 it can't proceed when it contacts another wall.

- The second value is a two bit number that acts as a direction for the line to travel (North, East, West, South)
- The remaining number is used to determine the length of the barrier.

The second integer of a pair is split to gain the starting coordinates of the line. The second integer, *i*, is split into *x = i mod X* and *y = ((i div X) mod Y)* with *x* and *y* being the starting coordinates and *X* and *Y* being the total length and width of the maze grid.
The following figure 3.1 helps illustrate this process:

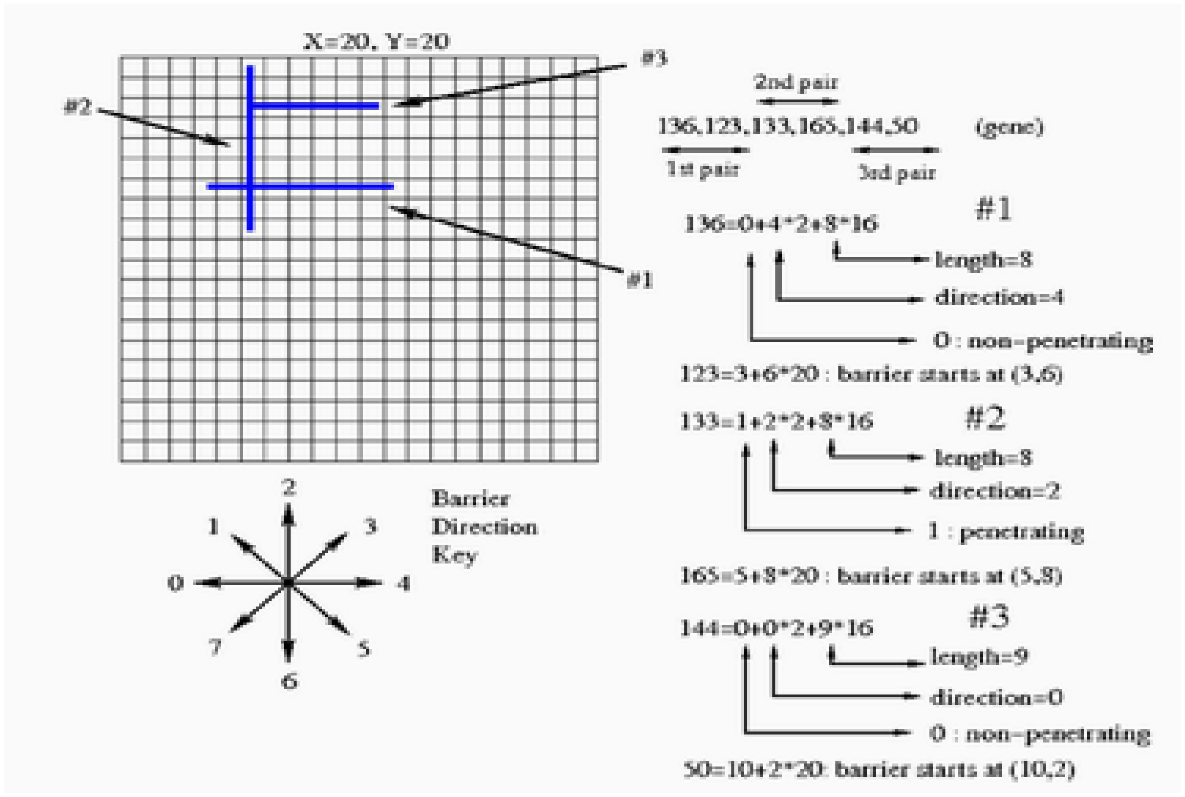


Fig 3.1. Shows how to unpack a maze from a pair of integers in an array. The integers are split apart to determine the properties of a wall. Pictures used from Search-Based Procedural Generation of Maze-Like Levels by Daniel Ashlock

4. Fitness Functions - Shortest Path Algorithm

Our program allows the user to enter the total length and width of the maze while the entrance, exit, and checkpoints are predetermined. The entrance is always in the upper left corner and the exit is always in the lower right corner of the maze. When generating mazes, it is important to note that all checkpoints will appear at the same geographical coordinates regardless of the difficulty it takes to reach them (see fig 4.2 a).

With the shortest path fitness function, our program is able to calculate how to solve a maze with the least number of steps possible. Solving the maze includes calculating the steps between the start and end of the maze. A maze is considered to be superior than another maze if, after being run through a fitness function, it returns a value closer to our intended goal. In the case of the shortest path, the fitness function will return a value of the lowest number of steps needed to reach the exit. However a lower value does not mean the maze is closer to our intended goal; we may wish to generate a maze were the shortest path requires high number of steps needed to complete. Any maze that does not manage to generate a high number of steps needed to complete would considered inferior to mazes that manage to meet that number of required steps (see fig 4.2 b).

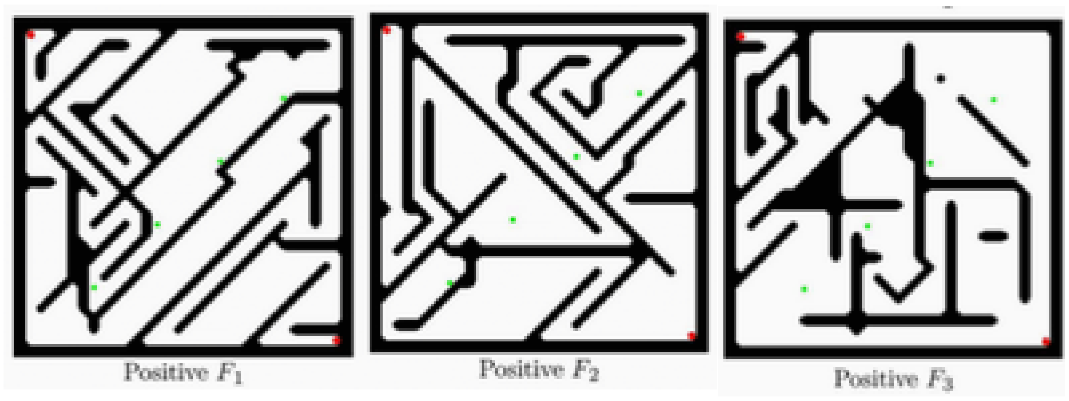


Fig 4.2. a)The red dots in the upper-left and bottom-right corners are the start and end points of the maze, respectively. The four green points are checkpoints spread throughout the maze (notice how the checkpoints are all still in the same locations on each different maze).
b) If we wanted to generate mazes with a high number of steps needed to complete, F1 and F2 would be considered superior to F3. Pictures used from Search-Based Procedural Generation of Maze-Like Levels by Daniel Ashlock

For our project we use a search algorithm called A* that, unlike other search algorithms like greedy best-first search which only considers the cost of the previously expanded node, accounts for the distance travelled from all previously traversed nodes to ensure greater speed and accuracy for finding the shortest path, and a heuristic function based on Manhattan Distance from the goal. Initially we had experimented with using a breadth-first search algorithm however it resulted in very long loading times and ran into memory constraints very quickly.

5. Fitness Functions - The Primary Reconvergence Algorithm

The Primary Reconvergence Algorithm implemented in our implementation uses 2 checkpoints within the maze at points (x-x/3,y/3) and (x/3,y-y/3). These checkpoints were chosen arbitrarily, to prove that it can be implemented within the time constraints of the course. This fitness function is an extension of the Shortest Path Algorithm, except with the checkpoints as the targets instead of the exit. Each maze is required to have a path from the entrance to each of these points, as well as the exit, otherwise a value of 0 is returned.
In the paper *Search-Based Procedural Generation of Maze-Like Levels,* the fitness function is defined as the following:

“Definition 7: The primary reconvergence of checkpoints ci and cj is the smallest path length from the entrance square to any square which is a reconvergence of checkpoints ci and cj if a reconvergence exists, otherwise it is 0. We define this quantity to measure when paths from the entrance through pairs of checkpoints first happen to converge.”

In other words, the program calculates the shortest path to both checkpoints to the entrance, and compares the paths they take. The point where the paths join is where the distance is calculated. See sample maze in chapter 13 for further details.

6. Generations, Mutations, and Crossovers

This program is implemented as a genetic algorithm. As such, the goal is to determine the superior mazes, breeding and mutating their genes at random, to produce evolved versions. Each generation shall unpack the genes of each maze in the population, run a fitness test, and determine its value. The population size is set at 120.

For the first generation, the population is initialized with a restriction: each wall cannot be longer than 3 squares; otherwise the number of unsolvable mazes causes the evolution process to be hindered because of the lack of information. This is called Sparse initialization.

After the population is run through a fitness function, the population is then broken down into groups of 4 at random for a tournament selection breeding process. This means that the winners of the tournament are selected to breed, with their resulting children overwriting the ones who lost. In this program, 2 winners are determined by the desired value from the fitness function, and the 2 losers are then written over. If there is a tie for 2nd, or all the mazes have no solution, the winners are chosen arbitrarily.

The breeding of the winners is completed by swapping genes between the 2. Each child shall be given a parent they resemble, and each chromosome within the gene has a chance to be replaced by the other parent. The probability of a crossover is set by an argument within the program (b). Each chromosome also has the probability of a random mutation, where the gene is given a new value from 0 to 65535, which is also defined as an argument (m).

7. Working with Java

We chose to write our project in Java for a variety of reasons. First of all both of us have previous experience working with threads in Java and wanted to see if we could apply our knowledge of java threading with the knowledge we gained from this course. Secondly we were curious to see how well java manages to hold up in parallel programming in case either one of us decide to use it in the future. Thirdly we have learned from class that java has been outfitted with parallel libraries very early in its life cycle and we want to see if that proves to be a factor in optimizing our code. Finally we know java does not usually participate in the scientific community and we wanted to explore if there is reason behind that.

A recurring problem we struggled with while using Java was that it tended to cause several memory constraints when running our program. These constraints could prove to be too difficult for us to feasibly perform fitness functions on large size mazes. Despite this potential problem with Java we can still gain some insight for which areas of maze generating can see benefits from parallelization.

After several revisions to our code, the memory issues have been significantly improved and now should not have any issues except with extremely large mazes. Initially our maze kept unexpectedly pushing the same node over and over resulting in gigabytes worth of memory being spent. Now our program only uses a few hundred kilobytes when executed.

8. Parallelizing The Maze

Overall we found the process of learning and adding java parallel code to our program to be fairly easy. Our serial code did not need to be altered much and was mostly cut and pasted into our callable Task method. The only issue we have is that our parallel function will at times suddenly get caught on a particular maze. It seems to vary depending on which arguments are provided. We are not sure what the exact cause of this issue is but it can be solved by changing the arguments given at the start of running the process.

In our Goals section we mentioned that we wanted to compare parallelizing the whole algorithm vs. parallelizing the fitness function. However due to time constraints we could only make the former of the two options happen. Our project is parallelized by assigning each thread to both generate a maze and run the shortest path fitness function through it. Once a thread has accomplished this, it sends the maze back to be compared to other mazes and then repeats. This concept is illustrated in Fig 8.1 below.

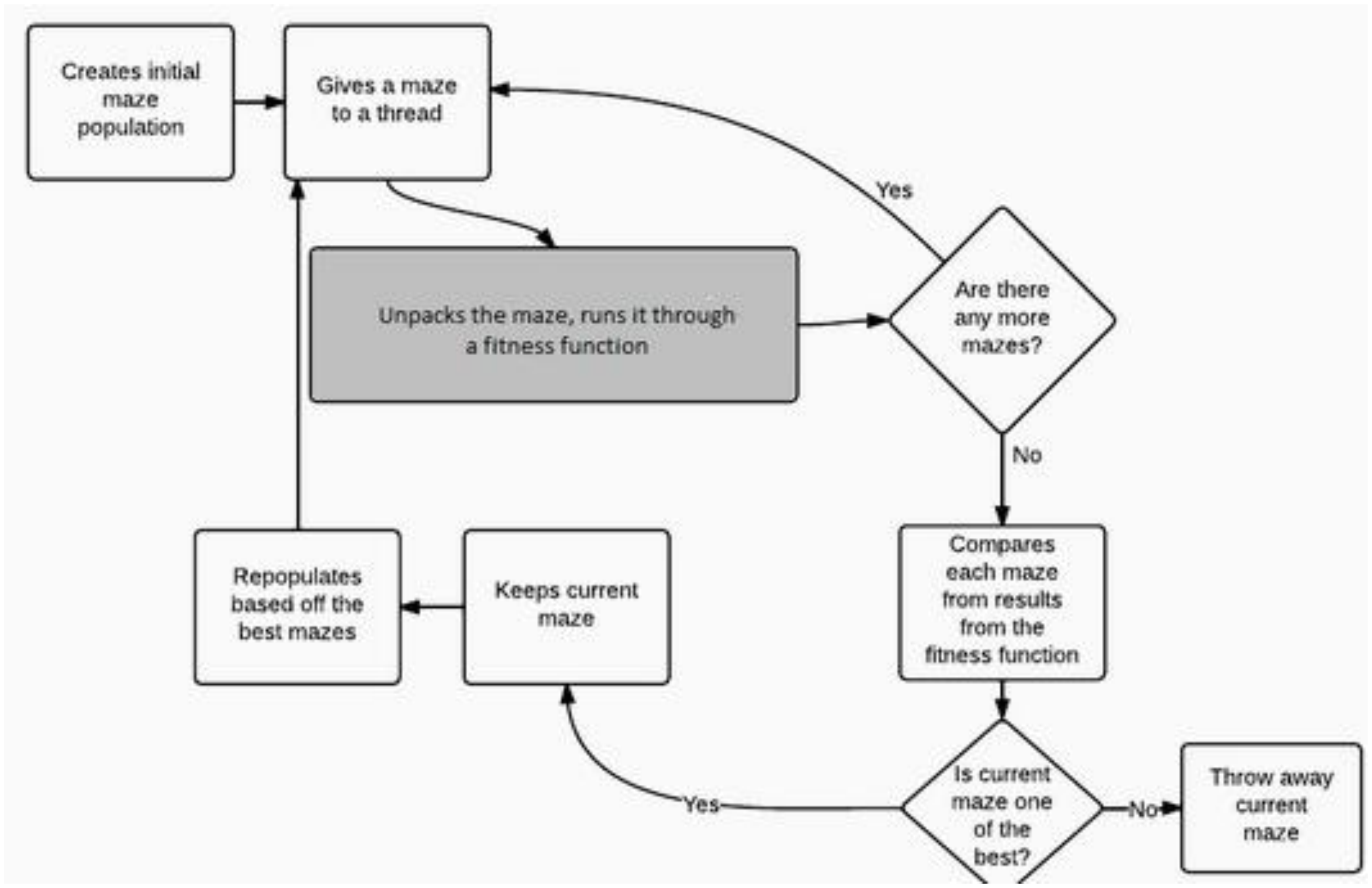

Creates initial maze population
Gives a maze to a thread
Yes
Unpacks the maze, runs it through a fitness function
Are there any more mazes?
No
Repopulates based off the best mazes
Keeps current maze
Compares each maze from results from the fitness function
Yes
Is current maze one of the best?
No
Throw away current maze

Fig 8.1. The greyed area is the part of our project that we assign each thread to complete and was parallelize our code

For parallelizing our program we used the CompletionService and the ExecutorCompletionService classes found in the java.util.concurrent library. Two methods are used for parallelizing our maze generating and fitness functions. The first method creates an ExecutorService object which determines the number of processors that will be used. The ExecutorService object is then used as a field in creating the CompletionService class which we will call cservice. The code needed to do this is:

```
        ExecutorService eservice = Executors.newFixedThreadPool(nrOfProcessors);
CompletionService < Object > cservice = new ExecutorCompletionService < Object >
(eservice);
```

We then create and submit a series of callable objects (which we refer to as tasks) into cservice. When these tasks are submitted the ExecutorService class handles which ones are send to each thread. This can be done with this line of code with index being a value we send to each thread:

```
        cservice.submit(new Task(index));
```

After all the tasks have been created and send to all the threads, we then use the command taskResult = cservice.take().get(); inside a loop to retrieve the results from each task. Finally to close all the threads afterwards we use the command:

```
eservice.shutdown();
```

9. User Guide

Reading Our Mazes: In this project mazes are displayed using a 2d integer array consisting of 1s, representing walls, and 0s, representing open space. An additional note when reading from a file (mentioned below) we also display 4s which form the shortest path possible to complete the maze.

Our project was executed using Netbeans on Lockhart and uses 11 different command line arguments. The arguments are:
- An integer dimensions width (x) and length(y), of the maze, including the it's border.
- An integer Number of processors to be used (p), must be greater than or equal to 1
- An integer random seed (r), so results can be duplicated.
- An integer number of generations (g) for the mazes to be bred together
- {0,1} The fitness type (s), an integer, either a 0, for the shortest path from the entrance to the exit of a maze, or 1 for a primary reconvergence of 2 checkpoints.
- An integer number for the desired value (d) of the fitness functions
- A decimal number for the mutation chance (m) that randomly selects a gene and produces a new number.
- A decimal number for the breed chance (b) that decides which genes to use from a parent.
- An integer number of generations to complete before writing the current populations genes to a file (o). Use 0 to disable this feature.
- A decimal number that specifies the maximum number of filled squares in the maze(l),

An example of using this is the following:

```
x30 y30 r30 l0.05  b0.05 m0.05 g2000 o200 p4 s0 d120
```

There is another argument for reading a file and displaying the results to the screen(file:). This argument will only work with a file name. An example is:

```
file:filename.txt
```

This argument enters the program into interactive mode. It allows the user to view the entire array of integer values of the gene, the maze itself, and the maze with the paths specified with a '4' based on the fitness function specified in the file.

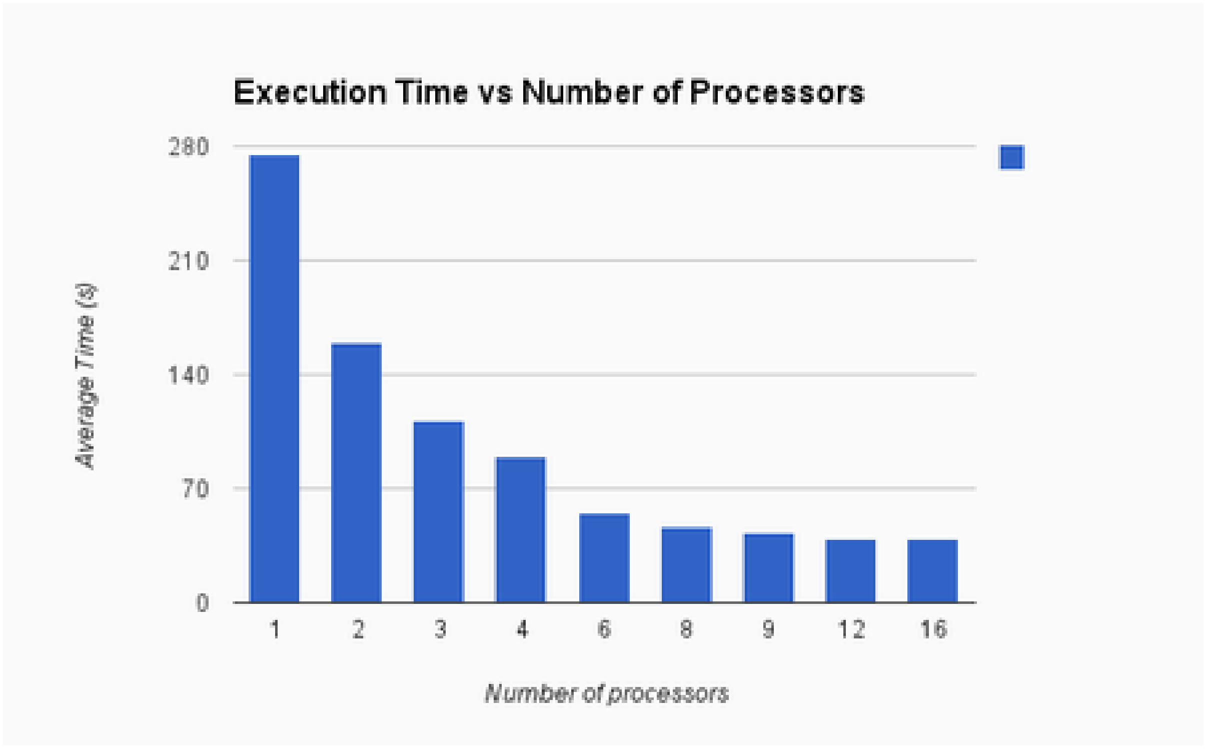
This argument is used for validation purposes only, and after it is finished reading the file, the functionality of the program is undefined.

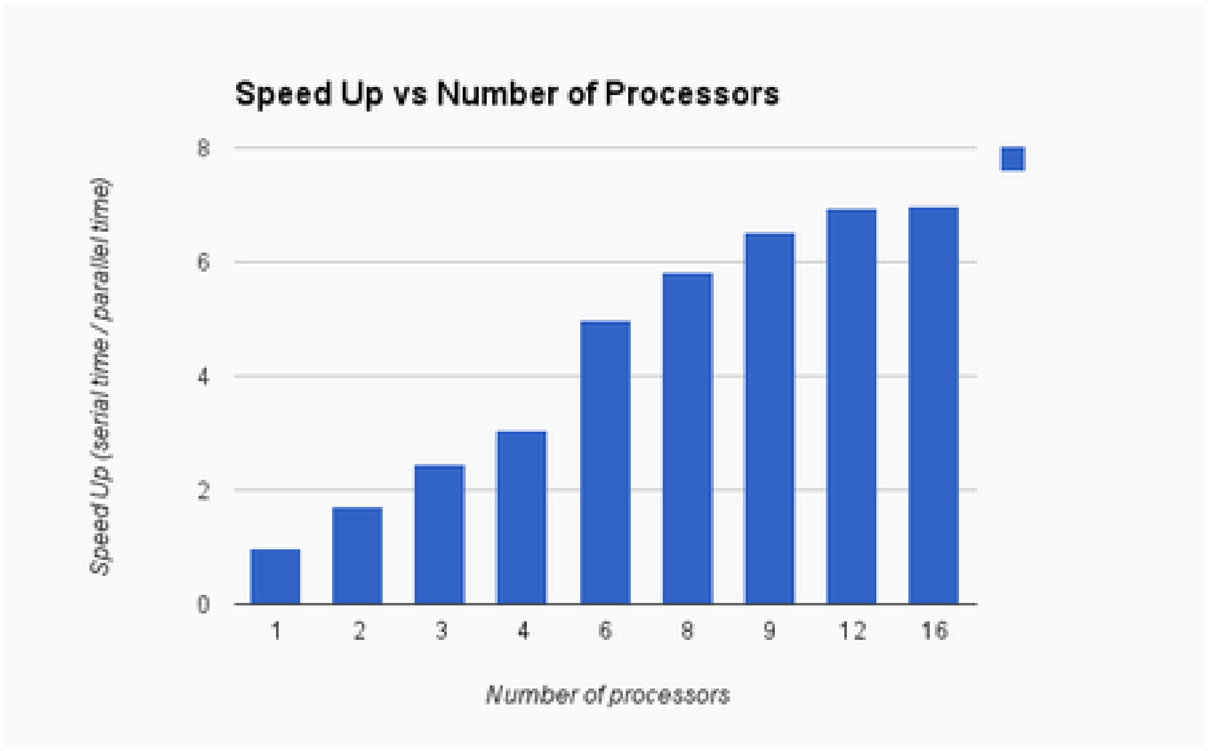
10. Results

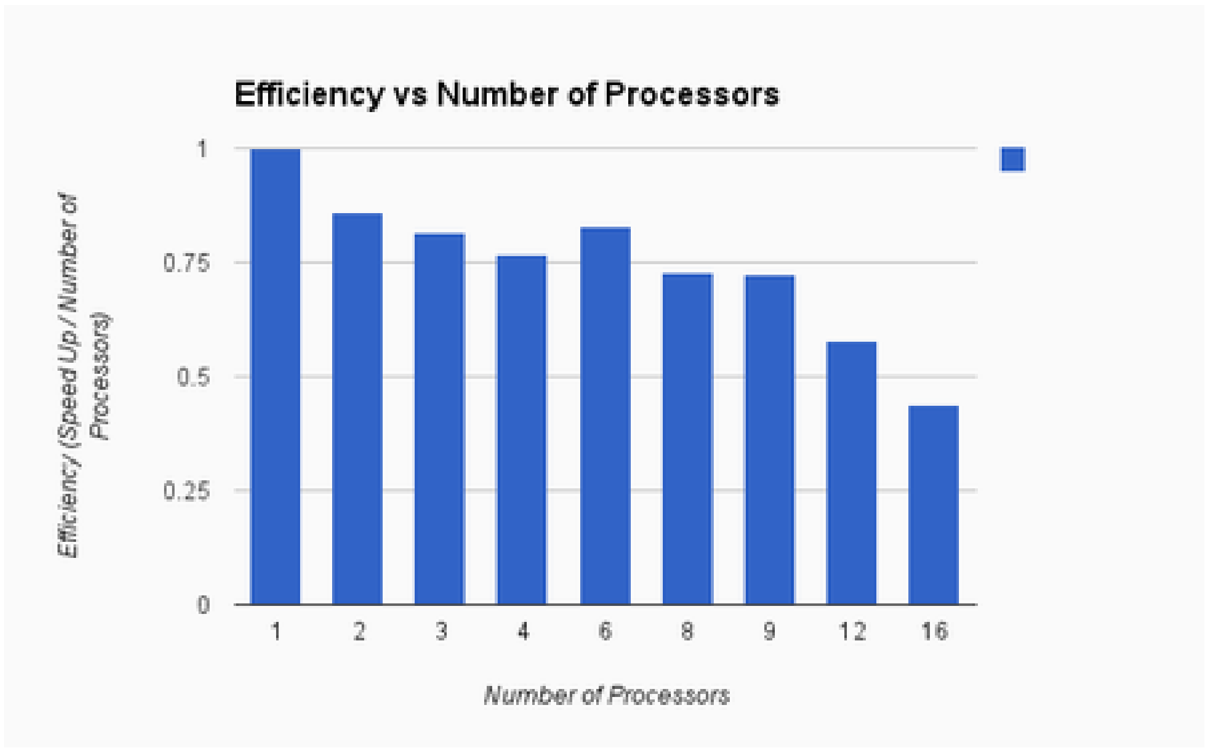
The values shown below were obtained using Netbeans on Lockhart with the following arguments (Note: the number after p indicates the number of processors used):

x50 y50 r30 g2000 o200 m0.1 b0.2 l0.05 d100 s0 p1

	Recordedtimes			Average	Speedup	Efficiency
p1 =	276.048	275.905	272.91	= 275.00	1	1
p2 =	157.454	162.206	160.785	= 160.15	1.72	0.858
p3 =	112.815	112.239	111.791	= 112.28	2.45	0.816
p4 =	87.503	89.807	91.403	= 89.57	3.07	0.767
p6 =	57.179	53.237	55.636	= 55.35	4.97	0.828
p8 =	46.881	47.156	47.445	= 47.16	5.83	0.729
p9 =	42.698	42.316	42.257	= 42.42	6.51	0.723
p12 =	39.456	39.694	39.533	= 39.56	6.95	0.579
p16 =	39.616	39.31	39.171	= 39.36	6.98	0.437




Speed Up vs Number of Processors
Speed Up (serial time / parallel time)
0
2
4
6
8
1
2
3
4
6
8
9
12
16
Number of processors


Efficiency vs Number of Processors
Efficiency (Speed Up / Number of Processors)
0
0.25
0.5
0.75
1
1
2
3
4
6
8
9
12
16
Number of Processors

<u>11. Experience</u>

How was the information you learned in this course put to use in this project?

For designing our project we knew which area of maze generating we wanted to parallelize. When the time came to implement the parallel portion of the code, very little of the code needed to be changed to accommodate. The java classes we used for parallelizing our code was very similar to OpenMP so we were able to use that knowledge to further account for how we initially write our code. Without this course, we would not be able to determine how to parallelize this algorithm in this fashion.

Was this a worthwhile parallel programming project?

We were initially concerned about how much parallel programming would help with generating mazes. However after consolidating Cameron McGuinness, one of the authors of *Search-Based Procedural Generation of Maze-Like Levels*, we found that mazes with a size of 50 x 50 or larger tend to take a long time to be generated. We found this to be a worthwhile project because of the amount of knowledge we gain about parallelizing in java as well as how to generate mazes and use genetic algorithms. The amount of time we managed to gain from maze generating also helps reaffirm that there are portions of maze generating that can benefit from parallel programming.

Where did you feet your knowledge and training fell short; i.e., what did you realize you *didn't* know or know how to do?

For the most part we felt our knowledge of parallel programming was sufficient for working on the project. The main area that fell short was with using the CompletionService class and the number of task methods that need to be created when the program is being run. If the number of tasks made is a wrong number there is a chance the program will get stuck on a certain thread. Unfortunately we are unable to determine what the exact cause of this is.

Tell about your experience with the tools. What worked or didn't work?

Despite our initial problems with memory constraints in using java, we found the language to actually be rather helpful in making this project. The CompletionService and ExecutorCompletionService classes proved to be very simple when adding the classes to our serialized code. In another example the PriorityBlockingQueue class helped not only with parallelizing our project but also in creating the A* algorithm. One thing that we found that didn’t work well for us was the unfortunate lack of any sort of Parallel Amplifier and Parallel Inspector equivalent in Netbeans as it would have helped in debugging and optimizing parallel java code.

What was easy or hard to accomplish?

The portion of the project that was the most difficult was creating a properly working A* algorithm. For creating this algorithm we had to create essentially from scratch and easily consumed the most time and energy for writing this project. This was the portion of code that would overload our computer memory and cause our program to crash. Additional problems caused by this algorithm include being stuck in very long computations and returning incorrect results.

Did you finish carrying out your original proposal? If not, why not?

Our initial proposal may have been a little too ambitious. A lot of the finer details in our proposal could not be implemented by the submission due date. Had we been a bit more humble, we would likely have achieved much more realistic goals. One portion that we did not manage to complete was comparing the results between parallelizing the whole algorithm and parallelizing the fitness function only.

However, we did manage to reach our main goal which was to decrease the amount of time needed to generate mazes. Furthermore we have easily succeeded in reaching our proposed increase in speed up with a 50% already being surpassed with just 2 processors. We also mentioned in our proposal that we wanted to include multiple fitness functions and while we didn't manage to create all of the ones we proposed (longest path, number of dead ends, number of ways to solve a maze) we did create the shortest path algorithm as well as the primary reconvergence. Aside from not managing to create an alternative way of parallelizing our program, we feel we have managed to carry out our original proposal.

12. Responsibilities / Expected Grade

Richard
- Writing most of the Proposal.
- Main coder for maze generating, shortest-path algorithm, and maze checkpoints.
- Edited report

James
- Main coder for the parallel programming portion of the code.
- Ran and recorded the timing results.
- Wrote most of the presentation.
- Wrote most of the report.

For this project we would propose a grade of 92%. Despite us not managing to create an alternate way to parallelize our project that focused on fitness functions, we were able to meet our initial proposal. The main focus of our project was to re-implement a maze generating algorithm with parallel code to increase the speed in which the mazes are created and in this regard we feel we have succeeded. From the results gathered we have greatly surpassed our goal of achieving 50% speedup, reaching execution times of up to seven times the serial version; this speedup alone proves that our project has merit.

Another reason for our proposed grade is that we tried to use a language other than C or C++. This required us to learn how to parallelize code that had not been used in any of our previous assignments in this course. Neither of us knew how to write parallel code in java but we were able to apply our knowledge gained from this course to create a functioning parallel program. In addition to using a new language for our project, this is the first time we've had create and use genetic algorithms, let alone creating a maze generator.

Our project offers insight to any student interested in the positive, indirect representation section of Daniel Ashlock's paper "*Search-Based Procedural Generation of Maze-Like Levels*." as well as to anyone wanting to see parallel programming done in Java or looking for ways to improve the speed of maze generation.

13. Sample Maze

generated using:

x50 y50 r30 g20000 o200 m0.1 b0.2 l0.05 d100 s0 p1

```
11111111111111111111111111111111111111111111111111
14000000000000000000000000000100000000000000000001
14011111111111111111111111111111110000000000000001
14000000000000000000000000000000000000000000000001
14000000000000000000001100000000000000000000000001
14400000000000000000000000000000000000000000000001
10400000000000000000000000000000000010000000000001
10444444400000000000000000000000000010000000000001
11111140400000000000000000000000000010000000000001
10000040444000000000000000000000100000000000000001
10000044004400000000000000000000100000000000000001
10000004000400000000000000000000000000000000000001
10000004000444444444444444000000000000000000000001
10000004000000000000000004440000000000000000000001
10000004400000000000000000004000000000000000000001
10000000400000000000000000004444444000000000000001
10000000400000000000000000000000004000000000000001
10000000400000000000000000000000000000000000000001
10000000400000000000000000000000000000000000000001
10000000400000000000000000000000000000000000000001
10000000400000000000000000000000000000000000100001
10000000400000000000000000000100000000000000000001
10000010400000000000000000000100000000000000000001
10000000400000000000000000000100000000000000000001
10000000400111111000000000000100000000000000000001
10000000400000000000000000000100000000000000000001
10000000400000000000000000000100000000000000000001
10000000444440000000000000000000000000010000000001
10000000000144440000000000000000000000010000000001
10000000000100044000000000000001000000000000000001
10000000000100004000000000000001000000000000000001
10000000000100004000000000000001000000000000000001
10000000000100004000000000000000000000000000000001
10000000000100004000000000000000000000000000000001
10000000000100004000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000001111111111111111100000000000111111111111
10000000000100000000000000010000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000001100000000001
10000000000100000000000000000000000000000000000001
10000000000100000000000000000000000000000011111001
11111111111111111111111111111111111111111111111111
```