# Introduction

## Intro to catan:

Catan is a popular board game developed by Klaus Teuber and published in the US by Catan Studio. The main aim of the game is to develop cities and settlements, and roads between them, to gain more resources and ultimately gain the most victory points. Up to 4 players can play, and each game takes between 30 minutes and an hour. For an extended explanation of the rules, please visit: https://www.catan.com/en/download/?SoC\_rv\_Rules\_091907.pdf.

Our catan is slightly different from the original game of catan. Our board is square, and settlements are placed inside the square instead of on the edges of the square. The development cards can be bought but not all are utilized. There are no ports, but players can trade 4 of their resource cards for a different resource card. One of the main reasons a speedup of performance is possible is that the board can be any size and any number of players can be playing. In testing, the largest game we were feasibly able to make was a 1000x1000 square board with 1000 players playing.

## Intro to our approach:

We have two versions of catan: a serial version and a parallel version. The serial version looks similar to a normal computer science project: Catan is implemented in C++ and a game is run using normal basic AI features. We have timed this version to determine the speedup of the parallel version. The parallel version uses OpenMP to parallelize both the board creation and the decision making. As described below, we did not use MPI or CUDA for the parallelization.

# Describing our approach

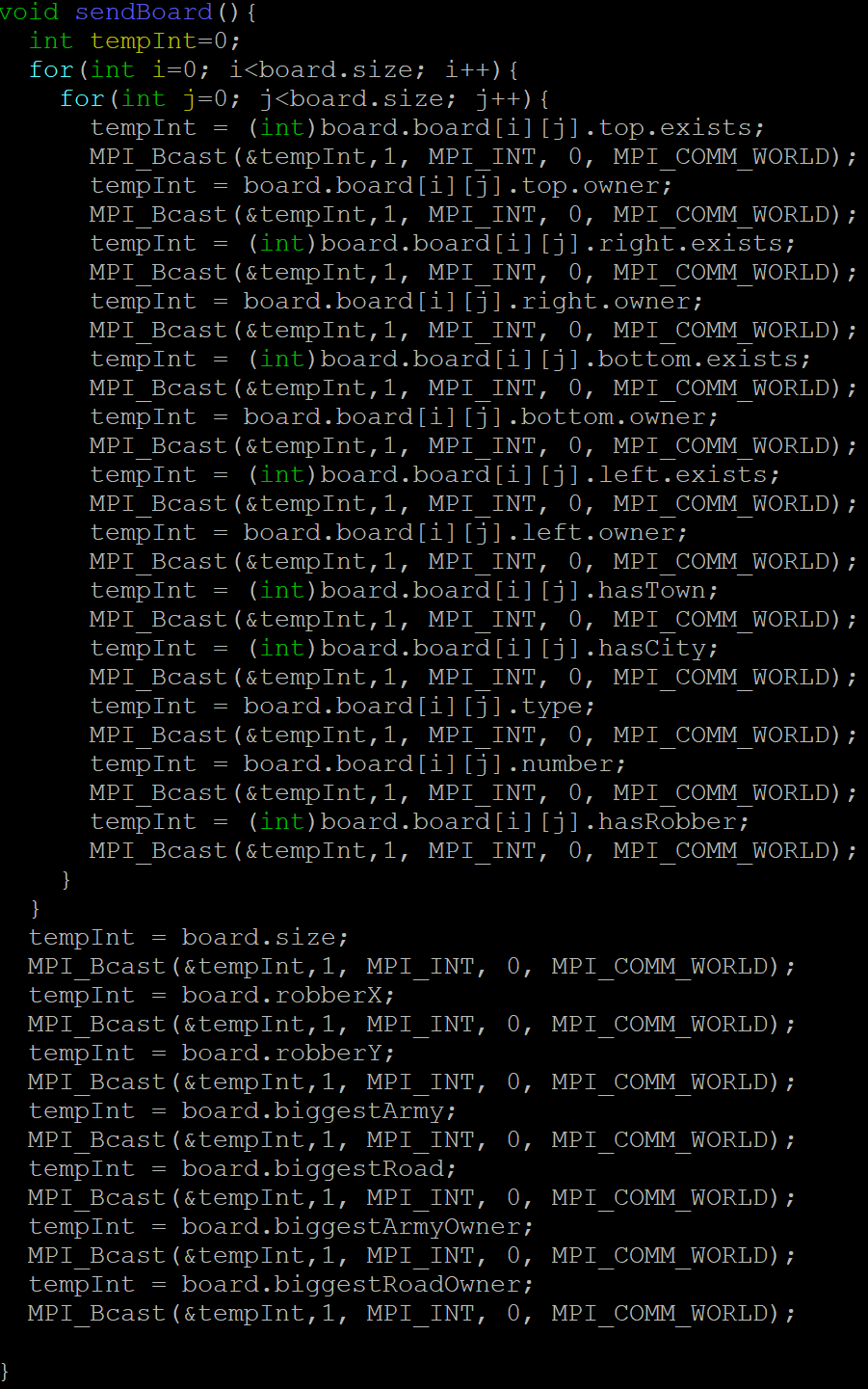
## Serial

The serial implementation is made up of a few classes that do the majority of the work. The Board class and Main file do most of the work of laying out the board and running each turn, determining who buys what things and where to place each item. The board is dynamically created as a vector so that the user can specify how large they want the board as a command line argument. Each player is also contained within a vector, so that an unlimited number of players can be playing at any given time. The majority of runtime of our game is in running the turn and all the decisions that go along with each turn. Making and printing the board can be computationally expensive when the board is very large, but this is only done once. Hundreds of thousands of turns can be run before a game ends, and each turn requires decision making.

## Parallel

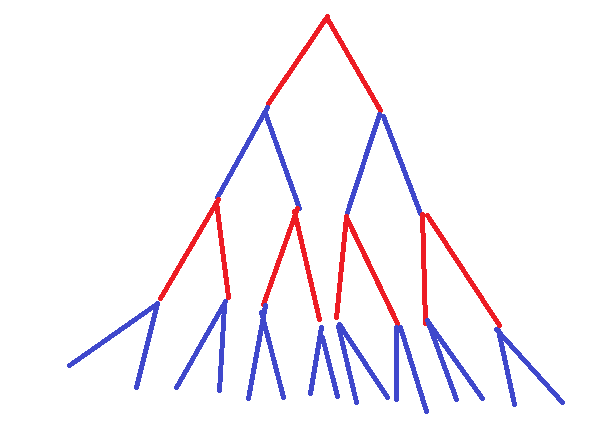
The parallel implementation of our game of Catan uses OpenMP. We were unable to utilize the message-passing ability of MPI efficiently to speedup our game, and CUDA is much more suited to a computation-intensive operation, which we do not have. The original plan for writing the game in parallel was to use MPI, but we quickly ran into a problem due to the distributed memory space that MPI uses.

We started writing our code using MPI and soon ran into a situation where one process changed the board and all other processes needed to get an updated version of the board. We soon realized that the simple sounding statement of updating the board was not so simple. First each process would need to check if another process updated the board before they could do anything to it, and if there was an update, all the work the other processes had done may need to be redone. While a broadcast is what is needed here, it also requires everyone to know where the broadcast is coming from ahead of time, but in our case any process could change the board and send it to the others. This means that processes have to send their updated board to process 0, which broadcasts to everyone else, adding more communication overhead.this slows down the system, on top of using process 0 as a communicator. Another issue is that there was no easy way to send the complex board object over standard MPI calls. With a little research, we found the boost library which people claimed could do what we wanted, but it didn’t appear to be on bluewave. We then decided to write functions that sent the board piece by piece. The sendBoard function is shown below in figure 1.



*Figure 1: An MPI implementation of sendBoard()*

We then deemed that the distributed memory system of MPI would not work well for our problem because runtime would be bogged down by constantly broadcasting the board to everyone. We then looked to openMP to solve our problems since it uses a shared memory space where threads are used to do work concurrently. This leads us to the planned design of the parallelization, which would mainly take place in the players taking their turns, since most of the game is processed once at the beginning. Our plan for parallelization was originally to treat the game as a tree, where each node was a board state, with the root being the current state, and each path is a potential move. The idea here being that each process can traverse a path and evaluate the board state, then the process that finds the highest scoring board state is the move that is used. This is a common AI technique called a minimax search, since each layer of the tree would be a different players move. The problem we found with this approach is that we said we wanted our game to run with any number of players. Variations of this approach are used in games such as chess, which made us think it might be effective on Settlers of Catan. Below is an example of what one of these trees would look like, with each color representing a player. That makes this a two player game where each turn the player only has two choices.

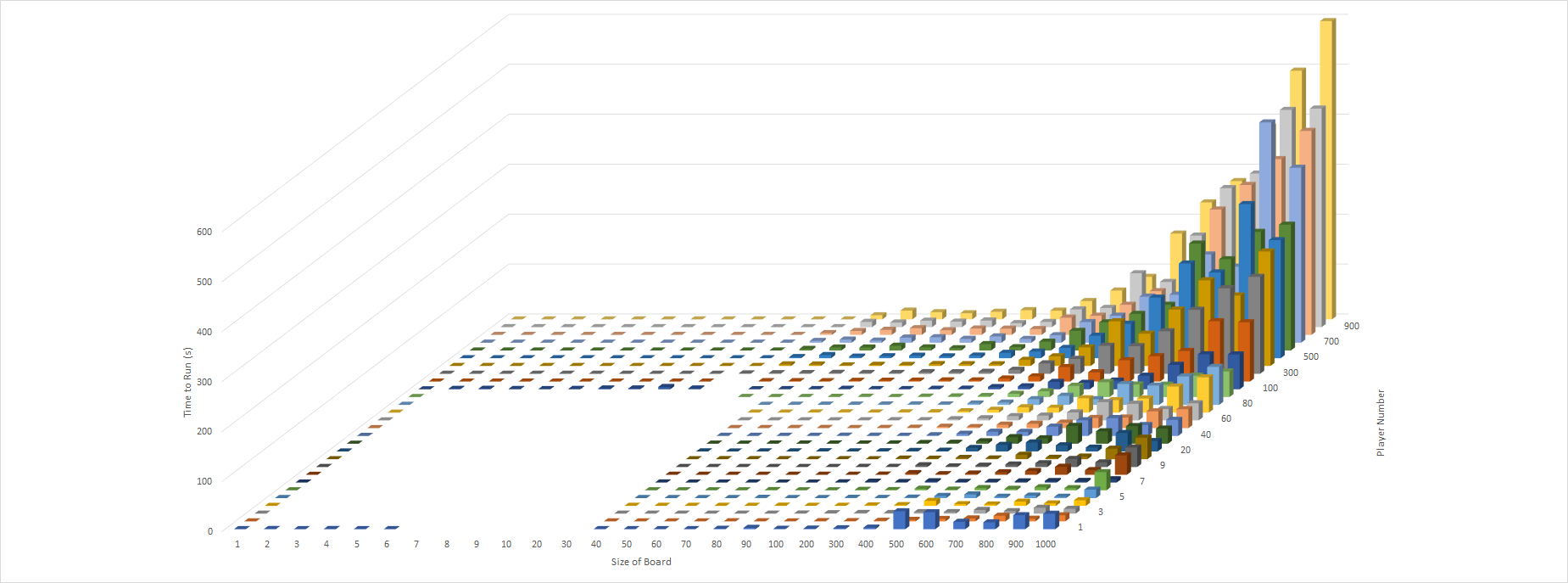


*Figure 2: Example minimax tree with only 2 players*

The problem with this approach in Catan is not that each each level would have have many more branches, but instead the problem is the number of players. In the legendary chess computer Deep Blue, the minimax tree generated for any turn has a depth of 12 (6 turns per player). If we wanted to look ahead 6 turns per person we would have a tree with depth 6\*n, which gives us a search space of x^(6n), where x is the number of potential moves. With a relatively small number of players this gives us an outrageously large search space with the number of players we plan to have the game run at (hundreds-thousand), or even far before that. This finally leads us to the final choice in our parallelization technique. To allow for any number of players and board sizes we would not be able to look ahead, instead using a greedy algorithm, taking the best move for a given board state without thinking about how the moves after that will go. After writing a function that makes the move of the player in serial we simply looked for loops that could be parallelized and sections that could run at the same time as each other. Many loops could not be parallelized because they need to run in order (like the players turns, they cannot all go at the same time). Other loops like looping over a player’s resources or development cards could be ran in parallel, using omp parallel. For the parallel sections we were able to run many of the players conditional statements at the same time, sometimes these sections were even inside of parallel loops, but to fully run all iterations of a loop and all sections inside that loop would take a lot of processors. For the parallel sections we used the omp parallel section directive.

# Results

## Serial



*Figure 3: Serial timing based on board size and player number*

The two variables in the serial code are the size of the board and how many players are playing. Each variable goes from 1 to 1000, in discrete increments. The graph is not perfectly square because, for instance, a game is impossible to finish if 1000 players are playing on a 1x1 board.

One finding is that, below a certain point, the time the program takes to run is essentially random. When the program would, on average, take above a minute to run, it would always run in about the same amount of time. However, below a minute, it usually ran in the same amount of time but could vary wildly with each run. This is because of the random nature of how the board is created. If a player gets lucky, they could have exactly the board squares they need to win a quick victory. If they are unlucky, they could rely on the development cards to give them the proper amount of victory points to win, which takes much longer.

Another interesting finding is that all logic of how long the program takes to run breaks down when only one player is playing. In both the columns and rows of figure 3, a clear linear trend can be seen from left to right and up to down. However, when only one player is playing, this trend evaporates and at larger board sizes no clear pattern can be discerned.

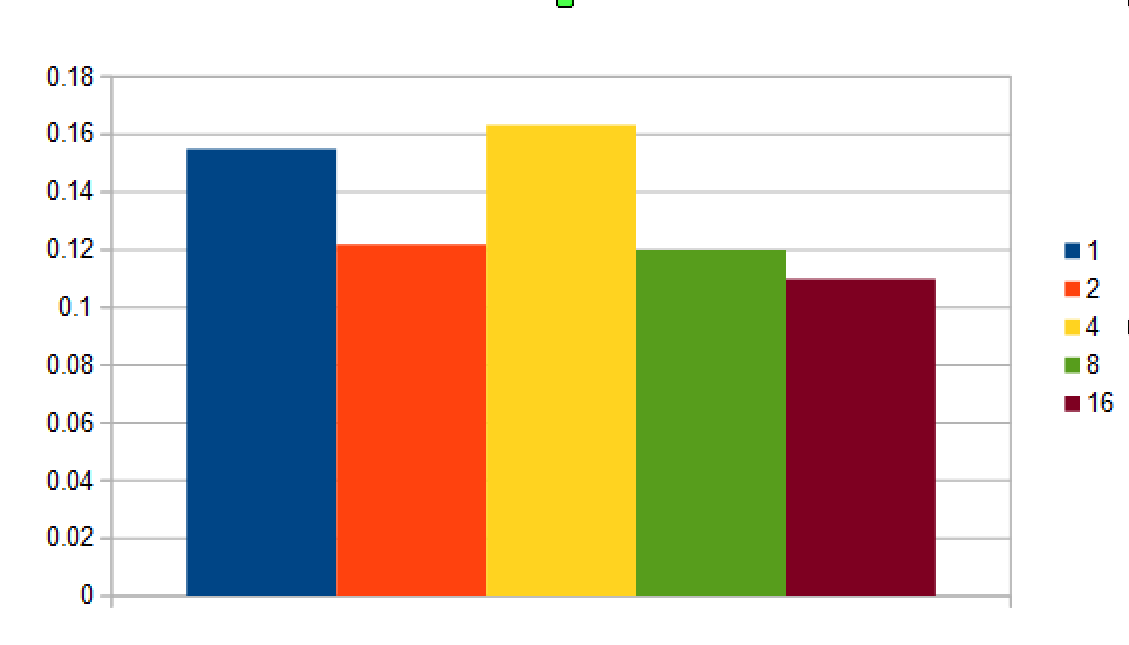
A final interesting finding is that board size has a greater impact on how long the program takes to run than player size. A 1000x1000 board with 40 players takes 32.21 seconds to run, while a 40x40 board with 1000 players takes 8.25 seconds to run. This difference can be seen in figure 3.

## Parallel

For scaling the problem in parallel, we changed the size of the board and the number of players, as before, but this time we added varied the number of points needed to win (victory points), which is not a command line argument but can easily be changed by changing the if statement inside the game’s main while loop. When trying to test the parallel version we were getting good results on Saturday, but after that, the run times were nothing like they were before, without any change to the code. I believe the cause of this is thread scheduling by bluewave, and everyone working to finish up their projects at the same time. As such we are unable to get reliable results from this point on, and thus only have the data already recorded. For comparison when everything was running smoothly we could run large games in around a second (64 players, 64x64 board), but now a small game take 30+ seconds (4 players, 4x4 board.

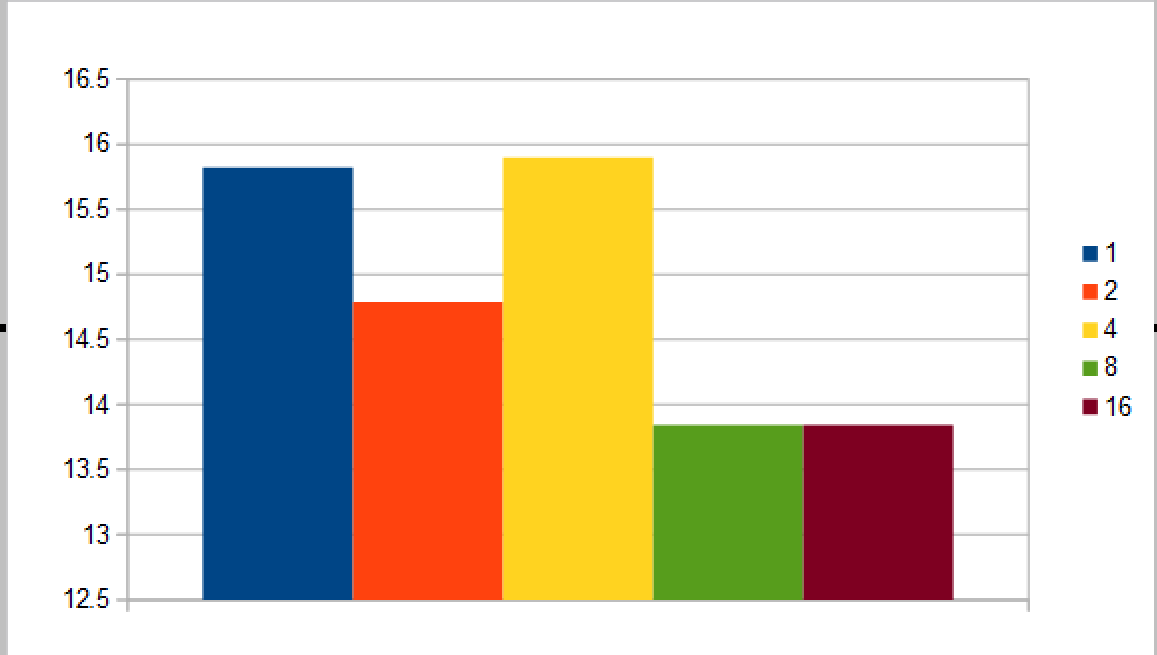
Here we have the results for size=16, players=8 and vp=50 (points to win)

And then with size=16, player=8, vp=5000, in order to show scaling as vp increases.



*Figure 4: Parallel implementation run using different number of pthreads*

size=16 players=8 vp=5000

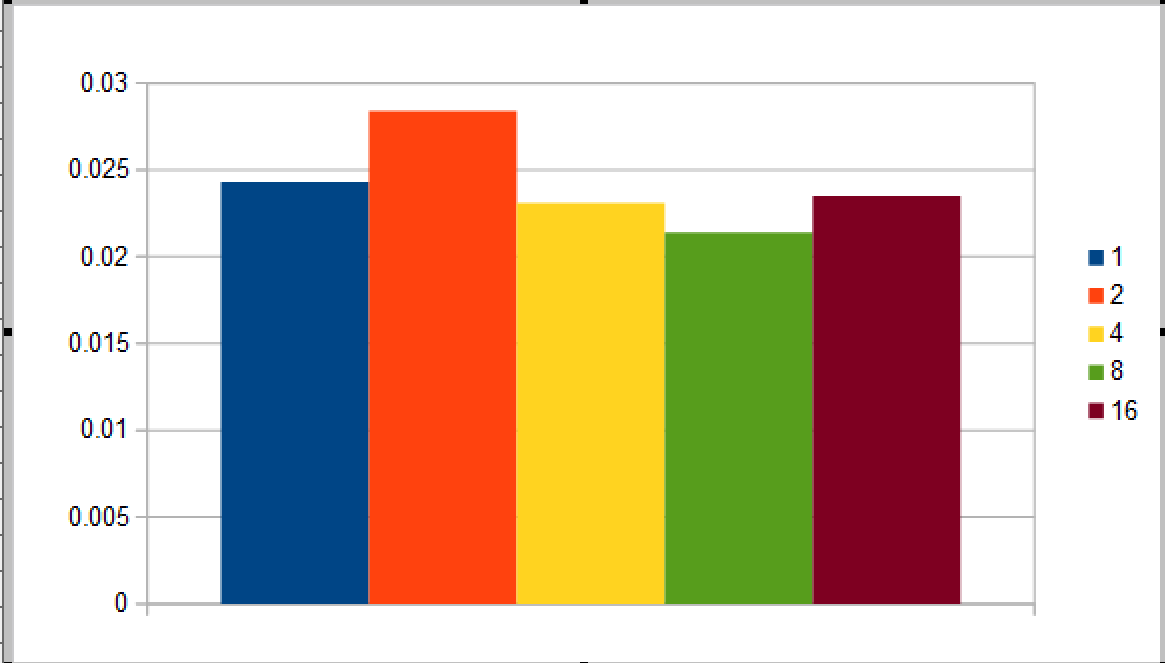


*Figure 5: Another parallel implementation using differing number of pthreads*

These values were generated using the real time from the time command, and taking the average of 5 separate trials. In individual runs we saw large variation between the times, due to the random nature of the game, since resources and development cards are generated randomly. The second graph above illustrates our best results of speedup, and even then it’s not great given 16 processors couldn’t even run in half the time of 1 processor. We also see that with 4 processors the runtime is worse than with 1 in both cases. We believe that this may be for a series of unfortunate games, since the next set of tests did not show this trend.

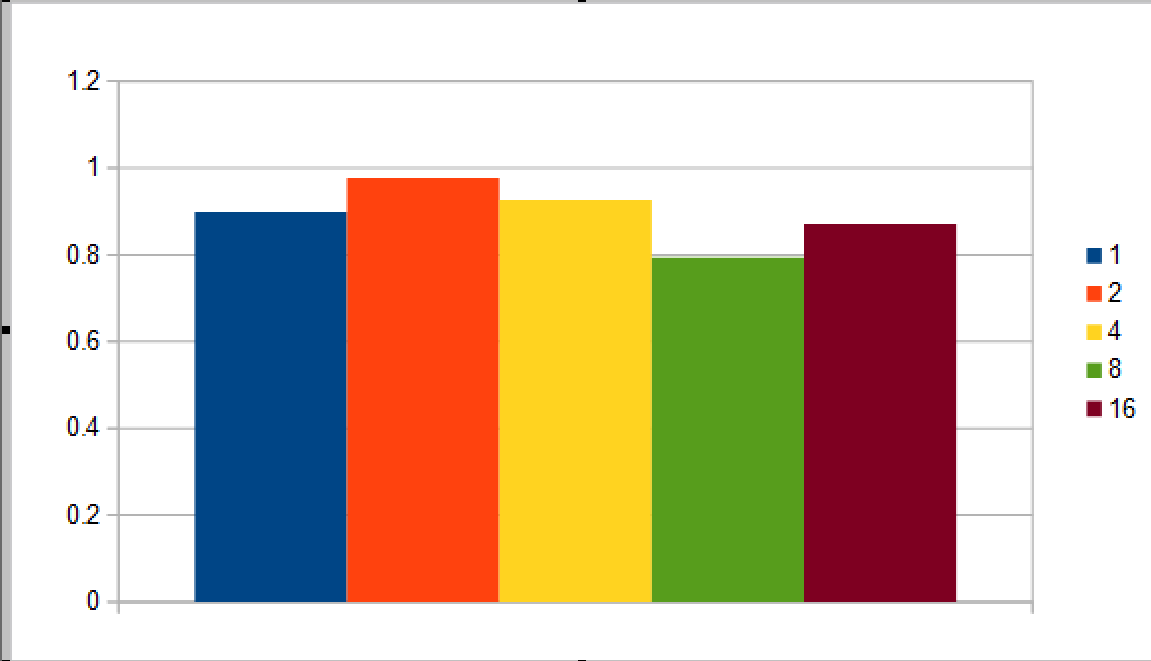
Here we show the scaling as players and board size increases.

size=4, players=4, vp=10 (this is our ‘standard game’, approximately what would be played by actual players)



*Figure 6: A larger parallel implementation of Catan*

And the next chart size=64, players=64, vp=10



*Figure 7: Another larger parallel implementation of Catan*

In these graphs the values are taken from averages of 10 runs instead of 5 like last time. In these charts we see poor performance, especially with 2 and 4 processors. We believe that at small amounts of victory points it is easier for a player to just happen to come by a low number and win quickly, diminishing the speedup. In the case of 2 and 4 here we believe that some of the parallel sections required more than 4 processors, so 2 and 4 processors are unable to run all of the sections at once, and thus does not perform as it should, but still incurring the overhead of thread creation. In both these instances we see that 8 processors yields the best results, we acreddit this to it having enough to run all sections at once, while not wasting any like in 16, where the additional processors just add more overhead for thread creation. In general our performance increase was very poor, and that the game lacks intense computational components, which are where parallel code is most beneficial.

# Conclusion

## What we learned

While most games can be efficiently parallelized, the fact that we had a board of unlimited size and an unlimited amount of players hobbled our ability to make a computation-intensive AI, limiting our ability to parallelize Catan. However, we found that OpenMP was the best technology to parallelize our code. Our speedup is not perfectly linear, but it was also hard to get good data on the parallelized version of our code.

Most importantly, we learned about the benefits and pitfalls of parallelizing code. Not all code should be parallelized, and not all parallelization is going to generate a significant speedup. The system that the code runs on is also important, because supercomputers are not as forgiving or transparent as a normal linux system.

## Future work

This project can be brought closer to the catan board game, including making the board hexagonal and placing settlements between the board squares instead of in them. Multiple complexities of the board game can be implemented too, including the ports that allow for different trading of resources and some of the board complexities of the expansions.