Jamie Haddow 0705082
CMP202 Data structures and algorithms
Multithreading – Astars pathfinding algorithm and grid initialization

Small addendum added to last slide as separate video

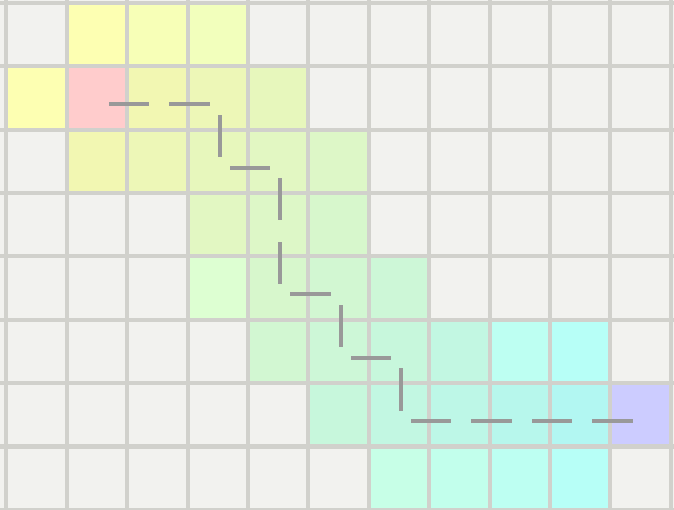
TDLR from CMP201

How do you get from A to B to C? Pathfinding!!

Strategy games are my jam.

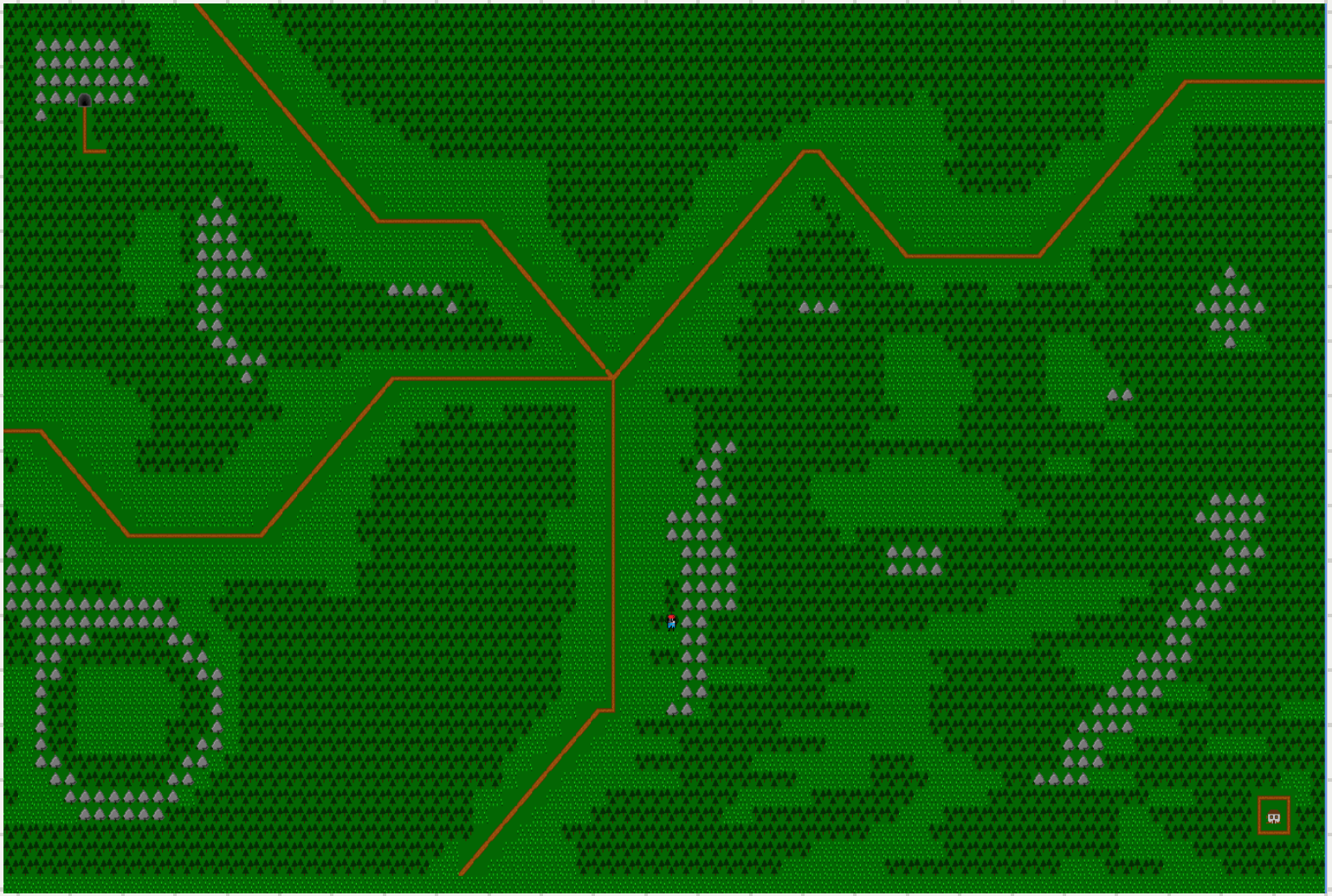
A comparison between Astar and Lee's was carried out on a range of grid sizes from 50x^2 to 500^2

Lee's algorithm was found to be the more appropriate choice due to the gird size being 100x100 and there being no real benefit from using astar with that grid size.



Good news!, my game company was bought out by a large company that liked the idea.
Bad news! The new project manager is being unrealistic.

They want to change my game idea from a small 100x100 grid to a much larger, open world with a potential grid size of 7000x7000.

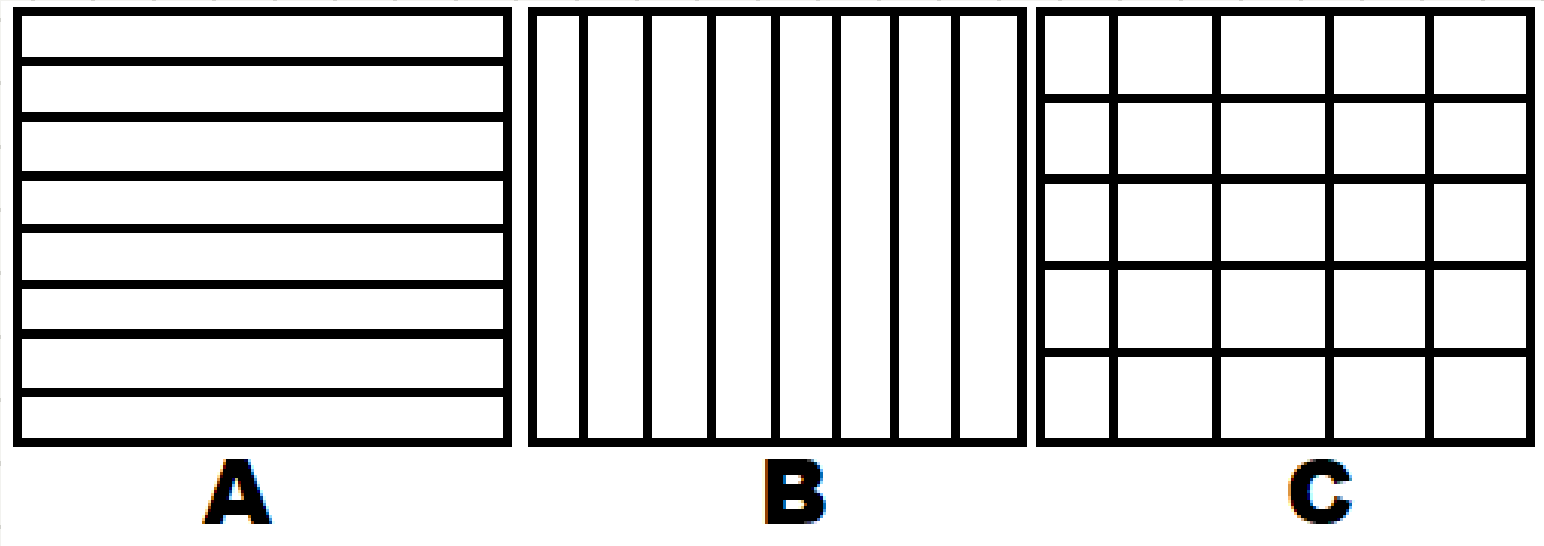



Sea of Ghosts
High Rock
Haafingar
Solitude
Dawnstar
Wintercrown
Winterhold
Dragon Bridge
Morthal
Hjaalmarch
Druadach Mountains
Karthwasten
The Reach
The Pale
Lake Yorgrim
Windhelm
Morrowind
Markarth
Riverstead
Whiterun
Whiterun Hold
Eastmarch
Throat of the World
Lake Ilinalta
Falkreath Hold
Ivarstead
Helgen
Shor's Stone
The Rift
Rift
Falkreath
Jerall Mountains
Cyrodiil

In turn, the purpose of this new application is to take the 201 astar framework, Multithread it as much as possible and test the results comparing threaded Versions of the code(non-threaded, 8 threads,16 threads, 32 threads)

The two main functions in this program are Grid initialization and pathfinding.

For grid initialization I had three choices in terms of looping through my array and assigning each element with variables. Option A and B were chosen due to not requiring an inside loop when assigning the function to threads.



During grid initialization A progress counter was added along with a function to print out a “ finished loading” text

```
void IncrementProgress()
{
    mu2.lock();
    progressCounter++;
//  std::cout << " Progress:" << progressCounter << std::endl;
    mu2.unlock();
}

void FinishedLoading()
{
    std::unique_lock<std::mutex> lck(mu);
    while (!progressisfinished)
    {
        cv.wait(lck);
    }
    //std::cout << " Array has been loaded successfully" << std::endl;
}
void initialiseGrid()
{
    int threadnum = 0;
    for (int i = 0; i < numOfThreads; i++)
    {
        threads[i] = std::thread(SingleGridThread, 0, height, i * width / numOfThreads, (i + 1) * width / numOfThreads);
    }

    for (int i = 0; i < numOfThreads; i++)
    {
        threads[i].join();
    }

    progressisfinished = true;

    if (progressCounter == numOfThreads)
    {
        progressfinished = std::thread(FinishedLoading);
        progressfinished.join();
    }

}
```

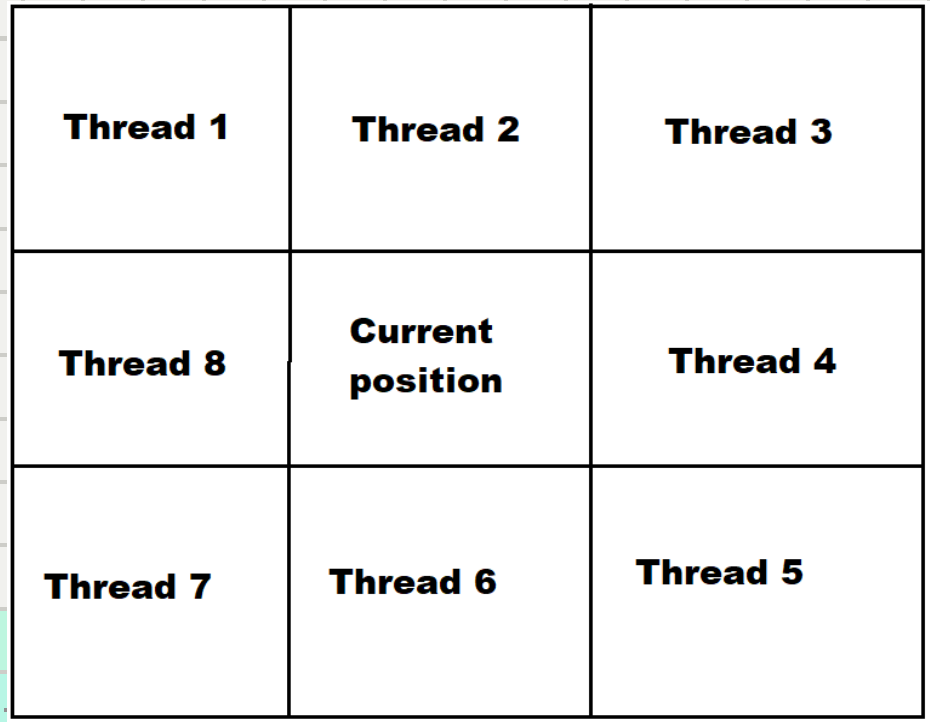
For pathfinding I used threads and a mutex./

8 threads were used to calculate the f,g and h costs. The fcost is then pushed into a priority queue that was protected by a Mutex due to it being a shared resource.

```
void SinglePathfindThread(int parentx, int parenty, int offsetx, int offs
{


    gridArray[parentx + offsetx][parenty + offsety].gCost += gridArray[xS
    gridArray[parentx + offsetx][parenty + offsety].hCost = gridArray[par
    gridArray[parentx + offsetx][parenty + offsety].setFcost(xEnd, yEnd);
    gridArray[parentx + offsetx][parenty + offsety].xParent = parentx;
    gridArray[parentx + offsetx][parenty + offsety].yParent = parenty;
    mu.lock();
    openList.push(gridArray[parentx + offsetx][parenty + offsety]);
    mu.unlock();

}
```



Both functions use loop parallelism to solve their individual problems.

For the course requirements, a condition variable was used but was not essential for the project.

Here are the sample code for both grid initialization and path finding thread creation.

```
int threadnum = 0;
for (int i = 0; i < numOfThreads; i++)
{
    threads[i] = std::thread(SingleGridThread, 0,  height, i * width / numOfThreads, (i + 1) * width / numOfThreads);
}

for (int i = 0; i < numOfThreads; i++)
{
    threads[i].join();
}
```

```
int threadnum = 0;
for (int i = 0; i < 8; i++)
{
    if (x + offset[i][0] >= 0 && x + offset[i][0] < width && y + offset[i][1] >= 0 && y + offset[i][1] < height
        && gridArray[x + offset[i][0]][y + offset[i][1]].closed == false)
        paththreads[threadnum++] = std::thread(SinglePathfindThread, x, y, offset[i][0], offset[i][1], xStart,yStart,xEnd,yEnd);
}

for (int i = 0; i < threadnum; i++)
{
    paththreads[i].join();
}
```

The CPU I ran the test on was an Intel® Core™ i7-9700K Processor

For the measurements taken with the initialization function I used 4 versions of threads and 3 different sizes. Non-threaded, 8 threads, 16 threads and 32 threads with the sizes being 1000^2, 4000^2, 7000^2. R = row. C = Column.

	No thread R	8 Thread R	16 Thread R	32 Thread R
grid size	1000x1000	1000x1000	1000x1000	1000x1000
Average ms	7.35	4.44	7.63	14.77
	No thread C	8 Thread C	16 Thread C	32 Thread C
grid size	1000x1000	1000x1000	1000x1000	1000x1000
Average ms	7.4	4.6	7.65	16.125

	No thread R	8 Thread R	16 Thread R	32 Thread R
grid size	4000x4000	4000x4000	4000x4000	4000x4000
Average ms	119.21	20.41	14.11	17.86
	No thread C	8 Thread C	16 Thread C	32 Thread C
grid size	4000x4000	4000x4000	4000x4000	4000x4000
Average ms	118.03	19.12	14.71	18.04

	No thread R	8 Thread R	16 Thread R	32 Thread R
grid size	7000x7000	7000x7000	7000x7000	7000x7000
Average ms	366.79	44.08	23.89	21.49
	No thread C	8 Thread C	16 Thread C	32 Thread C
grid size	7000x7000	7000x7000	7000x7000	7000x7000
Average ms	351.22	44.97	24.57	21.8

Results organized by row and then column split up by grid size

8 Thread R	No thread R	16 Thread R	32 Thread R
1000x1000	1000x1000	1000x1000	1000x1000
4.44	7.35	7.63	14.77
8 Thread C	No thread C	16 Thread C	32 Thread C
1000x1000	1000x1000	1000x1000	1000x1000
4.6	7.4	7.65	16.125
16 Thread R	32 Thread R	8 Thread R	No thread R
4000x4000	4000x4000	4000x4000	4000x4000
14.11	17.86	20.41	119.21
16 Thread C	32 Thread C	8 Thread C	No thread C
4000x4000	4000x4000	4000x4000	4000x4000
14.71	18.04	19.12	118.03
32 Thread R	16 Thread R	8 Thread R	No thread R
7000x7000	7000x7000	7000x7000	7000x7000
21.49	23.89	44.08	366.79
32 Thread C	16 Thread C	8 Thread C	No thread C
7000x7000	7000x7000	7000x7000	7000x7000
21.8	24.57	44.97	351.22

Sorted more by size combining threads and non-threaded with rows and columns

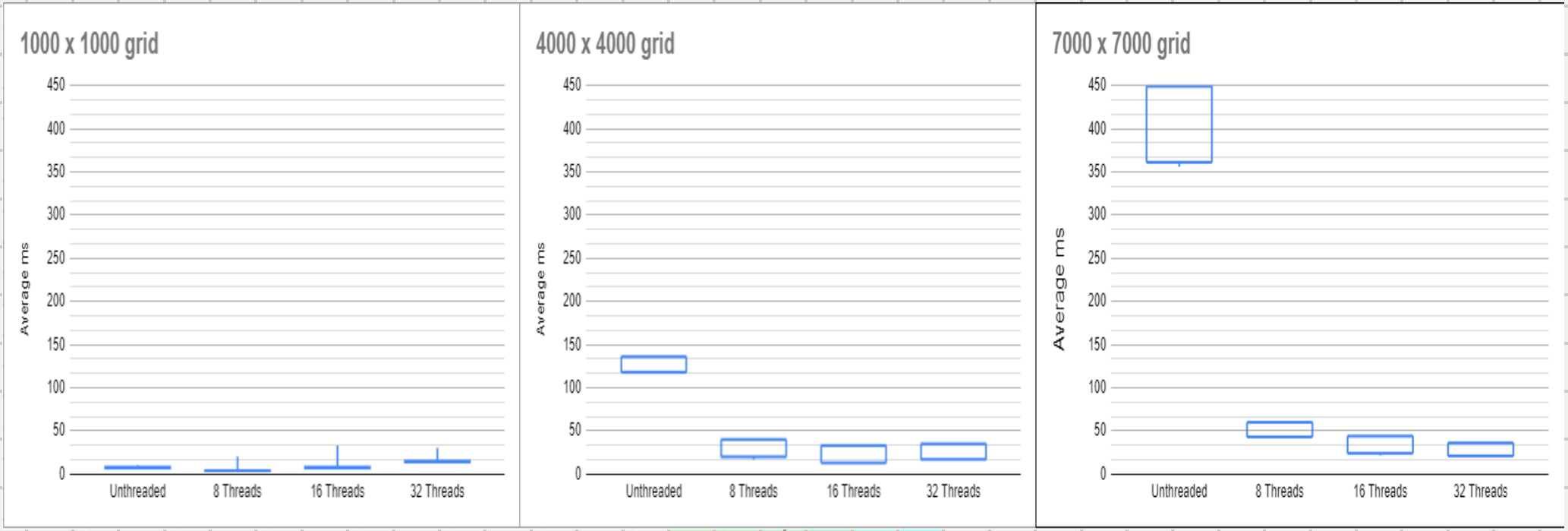
8 Thread R	8 Thread C	No thread R	No thread C	16 Thread R	16 Thread C	32 Thread R	32 Thread C
1000x1000	1000x1000	1000x1000	1000x1000	1000x1000	1000x1000	1000x1000	1000x1000
4.44	4.6	7.35	7.4	7.63	7.65	14.77	16.125
16 Thread R	16 Thread C	32 Thread R	32 Thread C	8 Thread C	8 Thread R	No thread C	No thread R
4000x4000	4000x4000	4000x4000	4000x4000	4000x4000	4000x4000	4000x4000	4000x4000
14.11	14.71	17.86	18.04	19.12	20.41	118.03	119.21
32 Thread R	32 Thread C	16 Thread R	16 Thread C	8 Thread R	8 Thread C	No thread C	No thread R
7000x7000	7000x7000	7000x7000	7000x7000	7000x7000	7000x7000	7000x7000	7000x7000
21.49	21.8	23.89	24.57	44.08	44.97	351.22	366.79

I will be using the first three in each row on the next slide for more in-depth performance checking

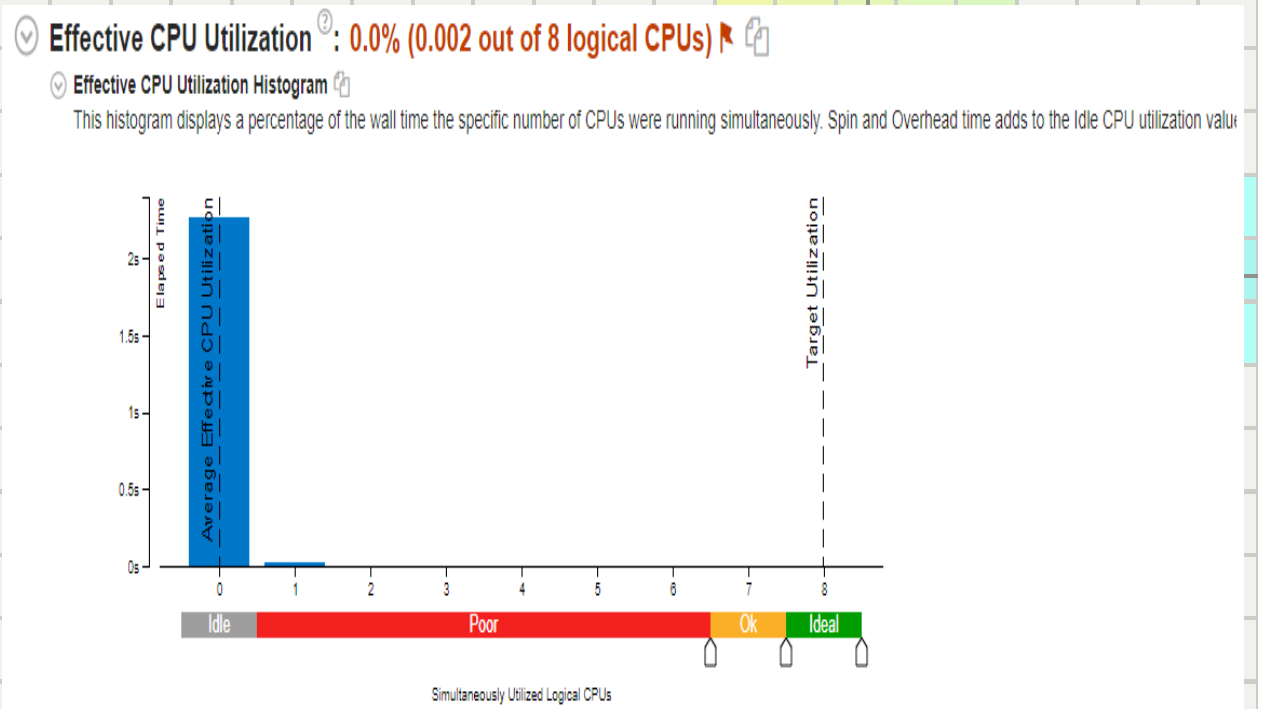
8 Thread R	No thread R	16 Thread R	16 Thread R	32 Thread R	32 Thread R	8 Thread R	32 Thread R	16 Thread R	8 Thread R	No thread R	No thread R
1000x1000	1000x1000	1000x1000	4000x4000	1000x1000	4000x4000	4000x4000	7000x7000	7000x7000	7000x7000	4000x4000	7000x7000
4.44	7.35	7.63	14.11	14.77	17.86	20.41	21.49	23.89	44.08	119.21	366.79
8 Thread C	No thread C	16 Thread C	16 Thread C	32 Thread C	32 Thread C	8 Thread C	32 Thread C	16 Thread C	8 Thread C	No thread C	No thread C
1000x1000	1000x1000	1000x1000	4000x4000	1000x1000	4000x4000	4000x4000	7000x7000	7000x7000	7000x7000	4000x4000	7000x7000
4.6	7.4	7.65	14.71	16.125	18.04	19.12	21.8	24.57	44.97	118.03	351.22

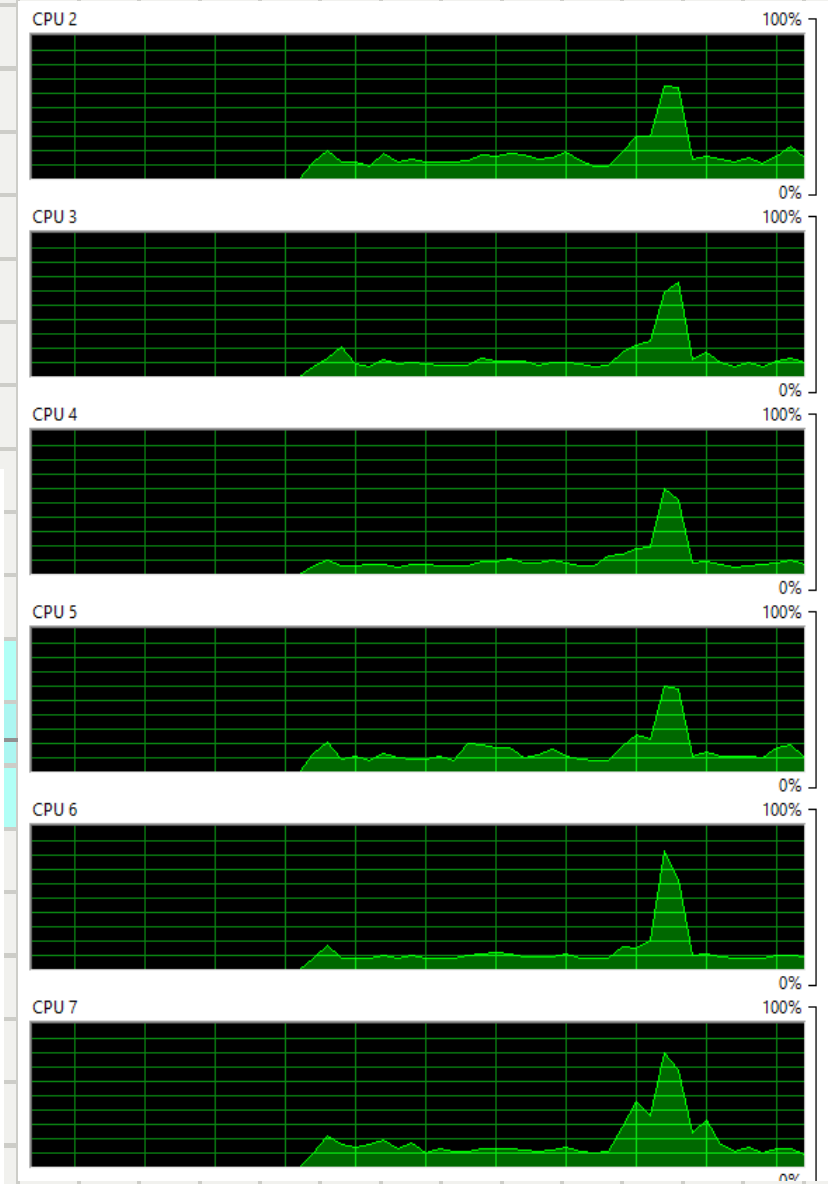
Sorted by performance in row/columns coloured by grid size

Candlestick charts to show performance for initialization with n=100 iterations



Using Intel's Ntune profiler recommended to me by Chris Trewartha after talking about cache line/misses and data oriented design, I attempted to measure the performance of threading. However, the profiler was not recognizing the threads were running so I was unable to use that to measure my performance



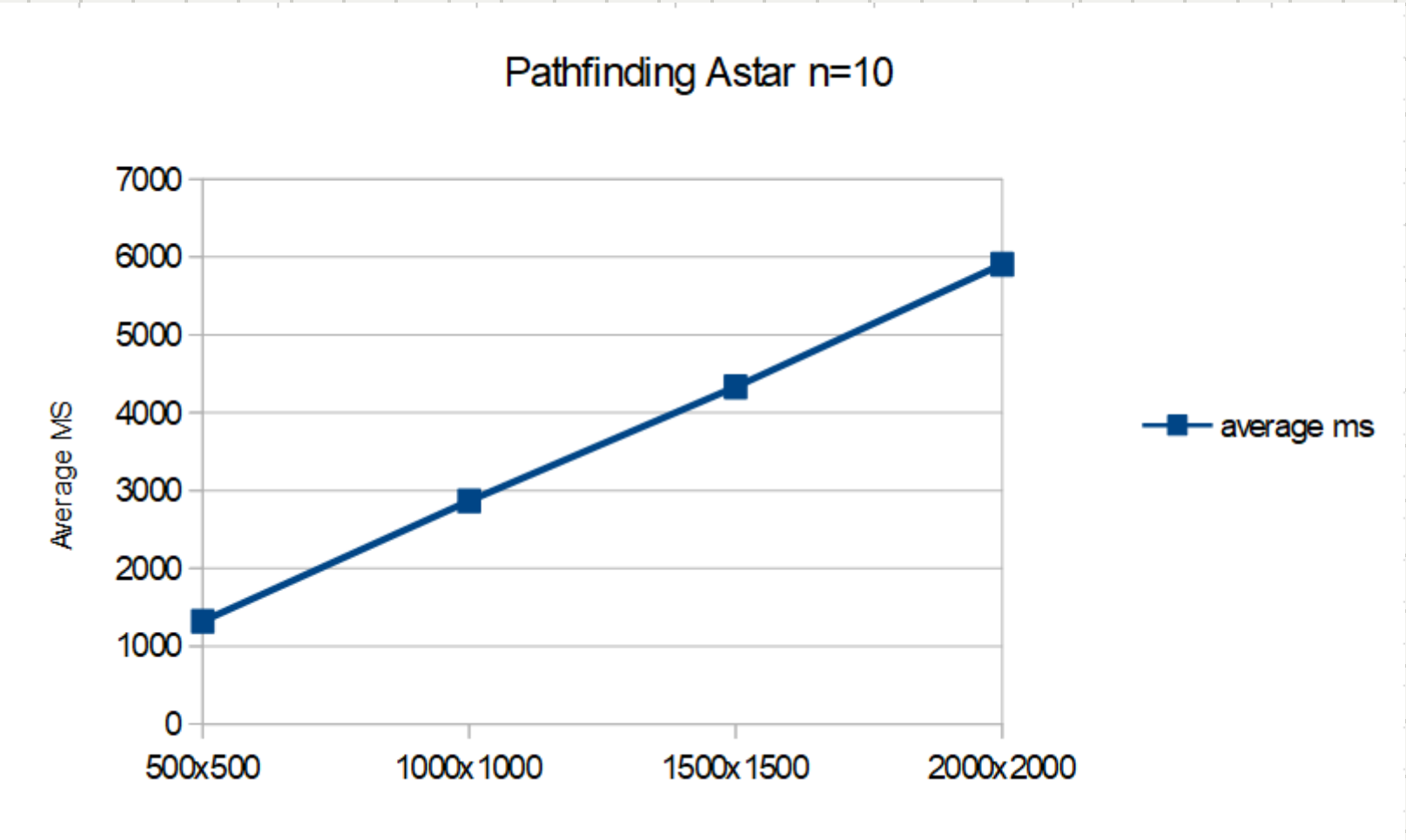


Running the Visual studios profile for both functions gave these results

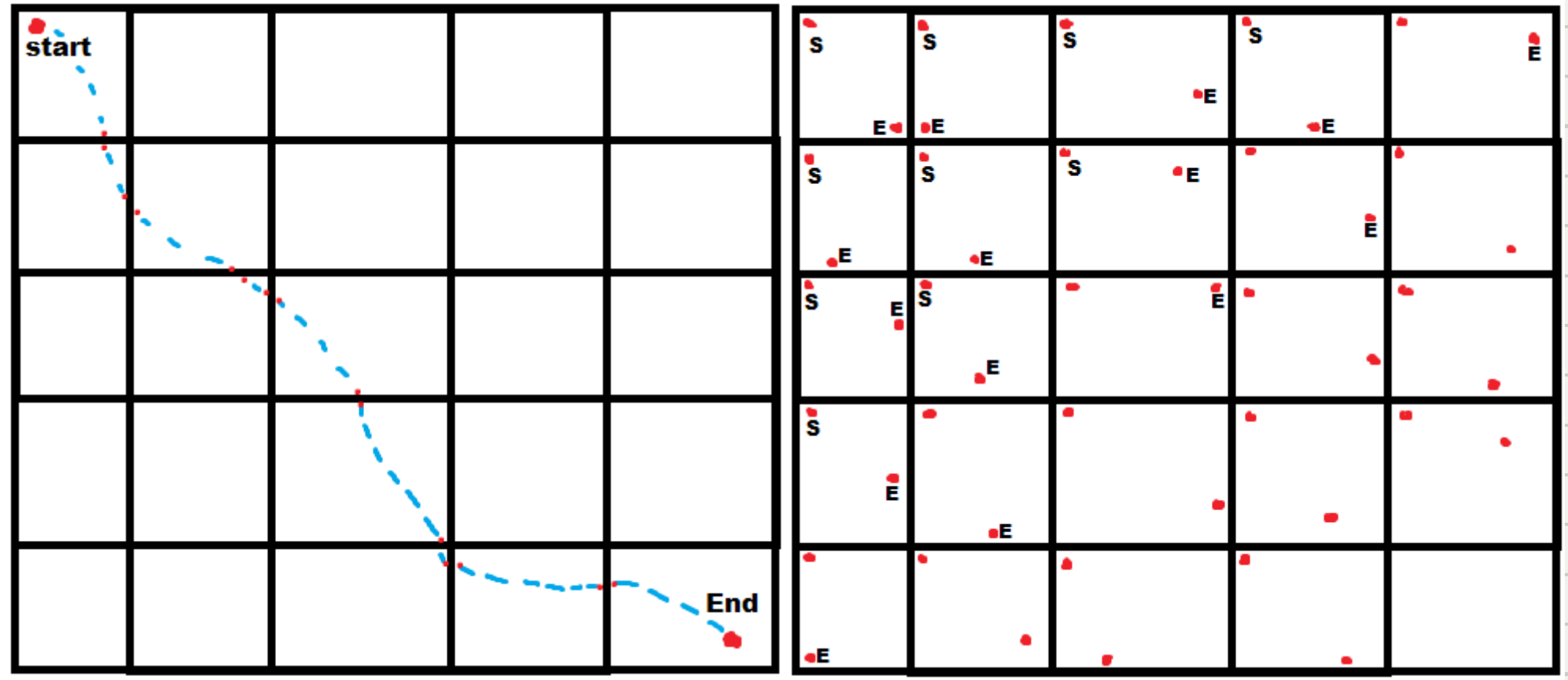
```
    1 (0.05%)  158      for (int j = lowX; j < highX; j++)
               159      {
    5 (0.23%)  160          for (int i = lowY; i < highY; i++)
               161          {
               162              gridArray[i][j].gCost = 1;
               163              gridArray[i][j].closed = false;
  638 (28.73%) 164              gridArray[i][j].xPos = i;
  575 (25.89%) 165              gridArray[i][j].yPos = j;
               166          }
               167      }
               168
               169      for (int i = 0; i < (highY - lowY) * (highX - lowX) / 6; i++)
               170      {
  803 (36.15%) 171          gridArray[rand() % 1 + ((highX - lowX) - 1)][rand() % 1 + ((highY - lowY) - 1)].closed = true;
               172      }
```

```
               237      int threadnum = 0;
               238      for (int i = 0; i < 8; i++)
               239      {
               240          if (x + offset[i][0] >= 0 && x + offset[i][0] < width && y + offset[i][1] >= 0 && y + offset[i][1] < height
    3 (0.06%)  241              && gridArray[x + offset[i][0]][y + offset[i][1]].closed == false)
  984 (21.30%) 242              paththreads[threadnum++] = std::thread(SinglePathfindThread, x, y, offset[i][0], offset[i][1], xStart, yStart, xEnd, yEnd);
               243      }
               244
    1 (0.02%)  245      for (int i = 0; i < threadnum; i++)
               246      {
  146 (3.16%)  247          paththreads[i].join();
               248      }
```

Unfortunately, in this specific scenario, Astar is less than optimal.



A few options where Astar could be useful in this project



Why does the current Astar path-finding function with threads run so slow?

Due to the threads requiring synchronisation, this leads to thread queueing

With such a low computation requirement, a single core can handle this data efficiently, so involving 8 cores is more of a memory bottleneck situation. We are no longer use cache memory but main memory if we need to synchronize

Please see Video “Addendum 1 – thread queueing”