Forest Fire Project

Abstract:

This project focuses on implementing and analysing a forest fire model using cellular automata in C++, utilizing OpenMP for parallelisation. The simulation involves a square grid with trees randomly placed, initiating a fire and spreading it based on defined rules. The forest fire model is first implemented in C++, utilizing OpenMP to parallelize a single run. The code outputs relevant information such as the number of steps, whether the fire reached the bottom, and simulation runtime for further analysis. Model convergence is then explored with respect to the number of repeat runs (M) for a fixed grid size (N = 100). Questions about the sufficiency of M = 50 repeats and its dependence on initial probability (p) are investigated, supported by data analysis and considerations of method strengths and limitations. It was determined that convergence is not reached at M=50, N=100 and after further investigation it was determined that with N=1000 and M=10 the simulation was very close to converging, with a standard deviation to mean ratio of 0.03. The model was extended by introducing wind, affecting fire spread in specific directions. The impact of wind on the number of steps before burning out and the probability of reaching the bottom was analysed for various wind directions. It was found that when comparing South wind to no wind at p=0.6 the south wind burns out with approximately half the steps that the no wind simulation requires and the probability that it reaches the bottom averages 1 at probability 0.4 whereas that occurs at probability 0.5 for no wind. The differences for other wind directions were less pronounced. Lastly, the project explores the dynamic scheduling parallel performance of the model on the BlueCrystal4 server, focusing on runtime variation with thread count for different grid sizes (N = 50, 100, 500). Results are presented and discussed, considering factors influencing performance, such as scheduling choices and their impact. The maximum thread count for each grid size is determined based on performance metrics, concluding with insights into OpenMP implementation effects. It was found that dynamic scheduling significantly increases performance for the larger grid sizes of N=500 but reduces performance for a grid size of 100 due to the overhead of managing threads.

Introduction:

Cellular automata (CA) have proven to be powerful tools in modelling complex dynamic systems across various disciplines. Originating from the seminal work of John von Neumann and Stanisław Ulam in the 1940s, cellular automata consist of a grid of cells, each evolving through discrete time steps based on a set of rules. This computational paradigm has found applications in diverse fields, ranging from physics and biology to computer science. The forest fire simulation is an example of this problem type.

Implementing the Basic Framework

Firstly, I created the forest fire program in its most basic state without any parallelisation. To make sure it was functioning correctly I developed a program that could animate my simulation results. I started off with a 10x10 grid as shown in Fig. 1 and then extended the program to a 100x100 grid as shown in Fig.2. I also created some videos of the forest fire evolving which can be found in my submission folder. These data visualization programs confirmed to me that my basic program was operating smoothly, and also lead me to some initial hypotheses about the nature of the program.

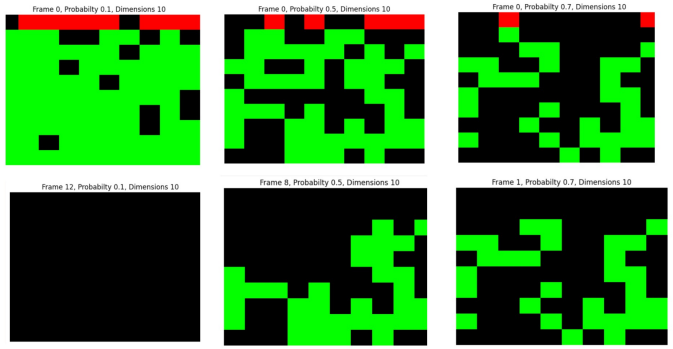


Fig 1: Shows the 10x10 grid evolving. Each column shows different probability density of a tree spawning, moving from dense to less dense as you move from left to right. The rows indicate the first and final time step.

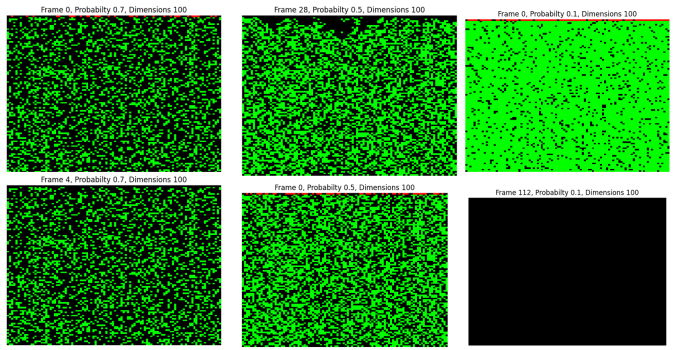


Fig 2: Shows the 100x100 grid evolving. Each column shows different probability density of a tree spawning, moving from less dense to dense as you move from left to right. The rows indicate the first and final time step.

Upon visual inspection of Fig 1 and 2 it can be seen that for smaller dimension sizes the outcome of the program will greatly depend on the initial random configuration of trees, therefore to bring the simulation closer to its true state that has small variance with regard to different random configurations one requires a larger grid size such as the one shown in fig 2. If the dimensions were increased indefinitely, the simulation would be brought closer to its true state however there the factor of computation expense needs to be considered when speaking practically. It is also clear from visual inspection of fig 2 that there is a threshold density at which point the forest is very likely to be almost completely burned.

How does probability impact convergence?

P = 0.1:

Mean: 2.4 Standard Deviation: 0.48989794855663565 ratio 0.20412414523193154

P=0.3:

Mean: 6.5 Standard Deviation: 2.3345235059857505 ratio 0.3591574624593462

P=0.5:

Mean: 35.5 Standard Deviation: 7.074602462329597 ratio 0.19928457640365063

P = 0.6:

Mean: 477.2 Standard Deviation: 89.83295609073544 ratio 0.18825011754135673

P=0.8:

Mean: 238.6 Standard Deviation: 1.8547236990991407 ratio 0.007773360012988855

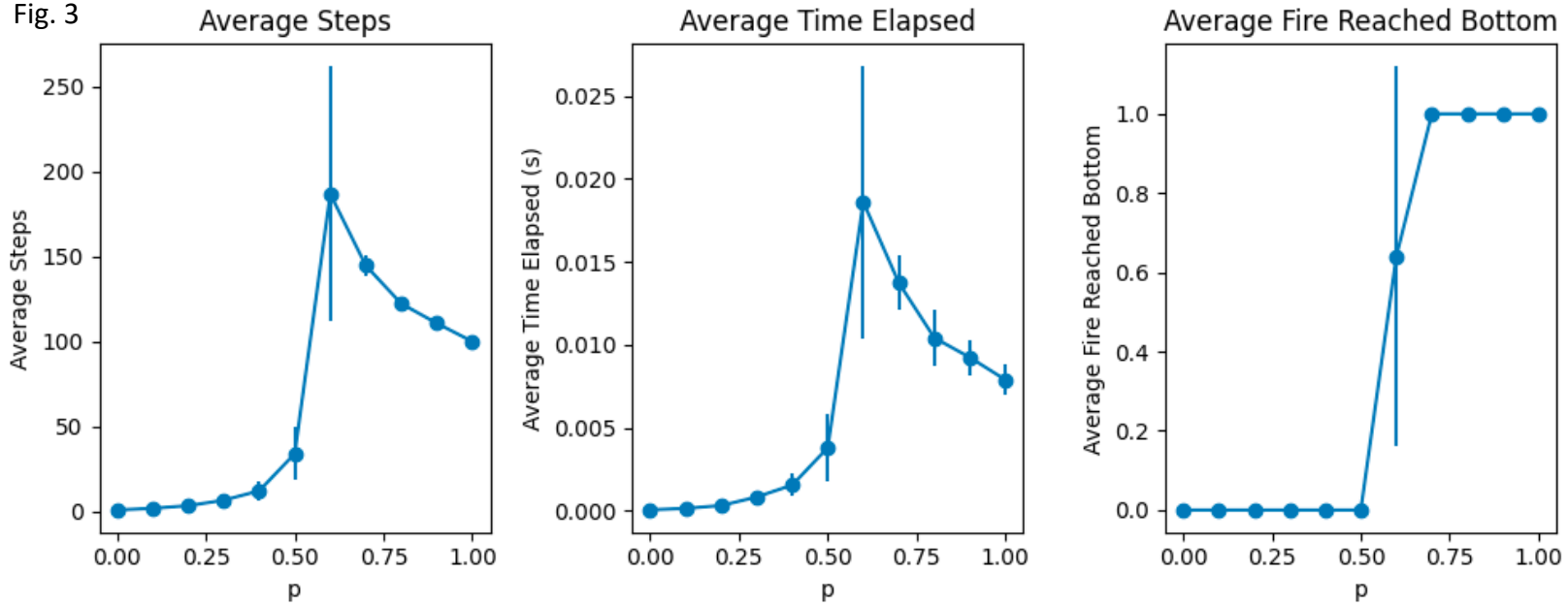
P=0.9:

Mean: 217.9 Standard Deviation: 1.3 ratio 0.005966039467645709

The data shown on the left is the simulation results of a 200x200 grid. The values indicate the mean, standard deviation and ratio between the two of steps taken before the forest fire completely burns out. At probabilities >0.8 the results do not vary significantly, whereas at probabilities less than 0.8 we see significant variability. Further investigation is required.

Convergence testing at M=50, N=100

Fig. 3



From the graph above, it can seen that M=50 runs for a grid size of 100 produces repeatable results apart from at p =0.6. The error bars are given by the standard deviation. Therefore, going forward, I will focus on trying to make sure my simulation produces repeatable results at p=0.6 by varying other factors such as N and M.

Initial Convergence Testing

Before completing speed tests, I discovered what the ideal grid size would be such that it converges on a ‘true state’ despite random fluctuation in the grid configuration. I ran a grid of size x ten times and took the mean and standard deviation of the steps. This was done at p = 0.6. Then I took the ratio of mean to standard deviation and used this as a measure of relative fluctuation of the results.

Grid Size	Mean	Standard Deviation	Ratio
50x50	60	11	0.20
100x100	163	20	0.10
500x500	986	200	0.20
1000x1000	2370	70	0.03
1100x1100	2590	96	0.04
1500x1500	3430	214	0.06

Table 1: Grid size vs mean, standard deviation and ratio between the two

It can be seen from this table that the results begin to converge at ~1000x1000 grid size for 10 runs. Better convergence could most likely be reached with greater grid size and more runs, however one is limited by time.

Implementation of Wind Direction

Fig. 4

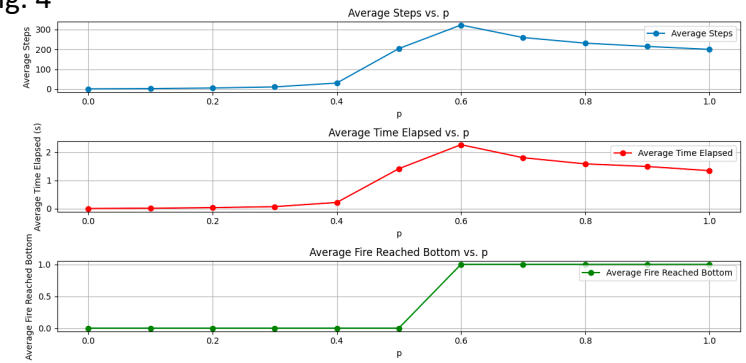


Fig. 5

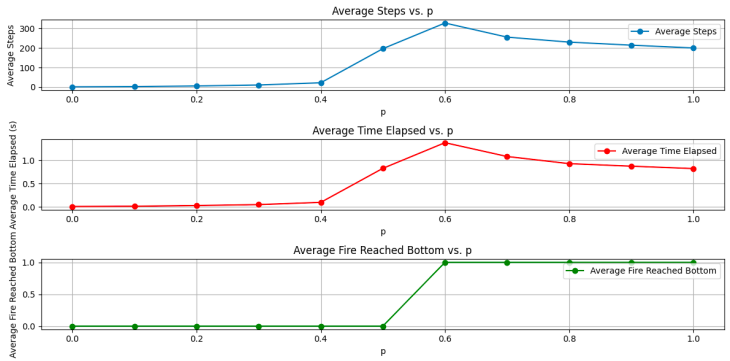


Fig. 6

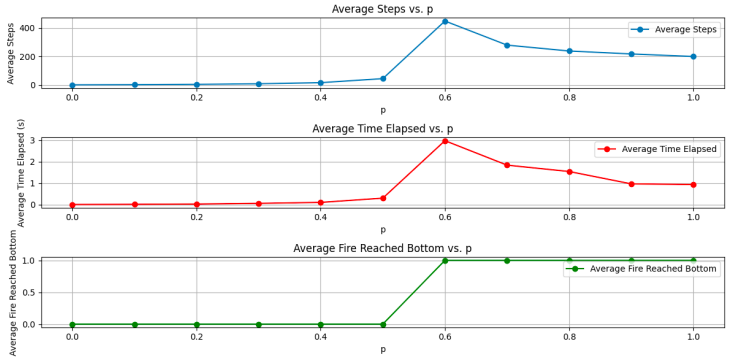


Fig. 7

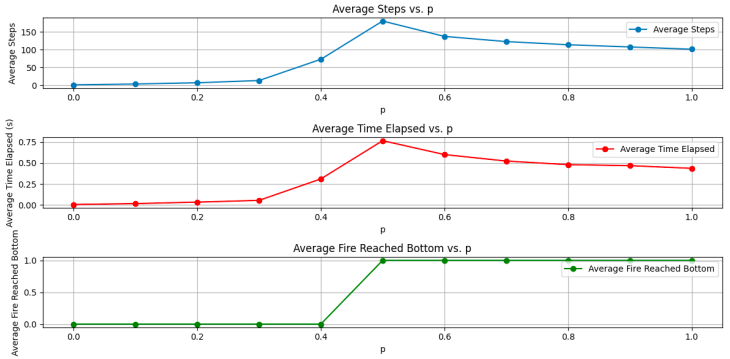
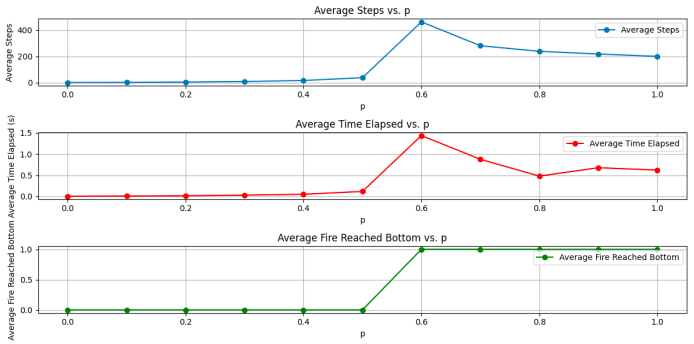


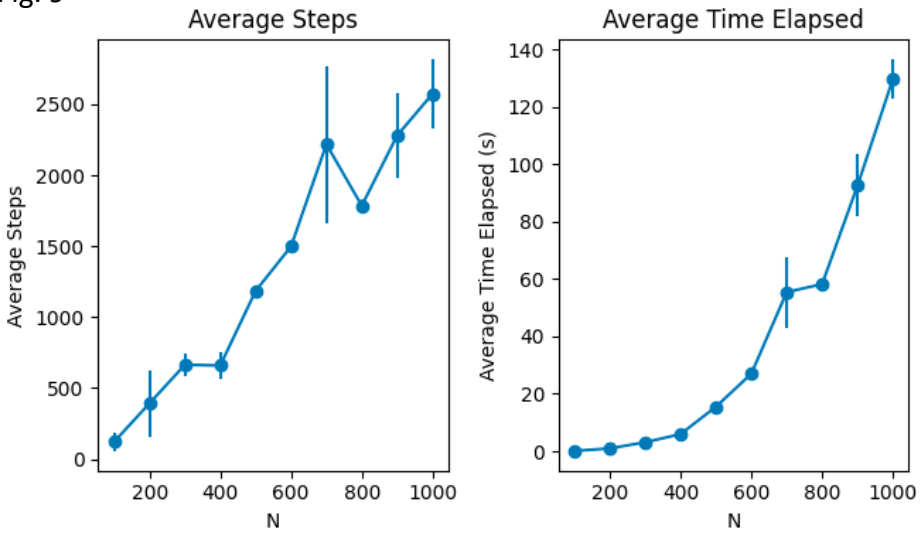
Fig. 8



Figures 4,5,6 and 7 correspond to wind directions West, East, North and South respectively. Fig 8 is no wind. These simulations were performed on a 200x200 grid. When comparing South wind to no wind it can be seen at p=0.6 the south wind burns out with approximately half the steps that the no wind simulation requires and the probability that it reaches the bottom averages 1 at probability 0.4 whereas that occurs at probability 0.5 for no wind. The differences are less pronounced for the other wind directions, however the amount of steps is reduced by approximately 100 for each of the other wind directions compared with no wind, apart from wind in the north direction which seems to make very little difference, which is expected considering it is the top row that is set alight. There is an mp4 file attached with the report that animates the impact of wind direction.

How grid size impacts serial performance

Fig. 9



The figure on the left shows average steps and average time elapsed for the serial program at probability 0.6 with a varying N on the x-axis. M was taken to be 2 to speed up computation. Average time elapsed increases exponentially whereas average steps increases linearly. This means that the time taken to compute a single step increases for larger grid sizes, which is expected. This fact becomes important when considering the impact of parallisation later in the report.

Implementation of Parallisation

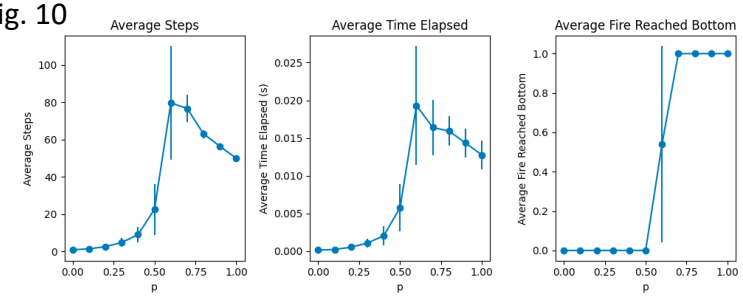
One method of parallising the code is to make the program run multiple grids at the same time across different cores. Another method would be to split the grid up into different rows that correspond to different threads, this is done automatically by the scheduling automation function of `#pragma omp parallel for schedule(dynamic).`
Dynamic scheduling is a task scheduling strategy used in parallel programming to distribute loop iterations or tasks among multiple threads dynamically at runtime. Unlike static scheduling, where the distribution is determined before the execution begins, dynamic scheduling allows the runtime system to adapt to the workload and the availability of resources during execution.

I implemented dynamic parallisation over a single grid and investigated the relative the time taken to run simulations.

Speed tests for openMP

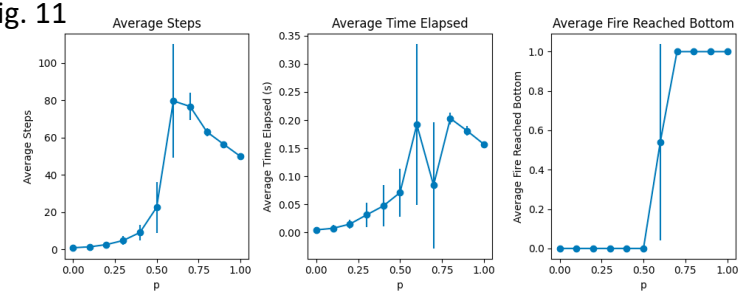
Simulations for: N=50, M = 50

Fig. 10



No parallisation

Fig. 11



Dynamic Parallisation

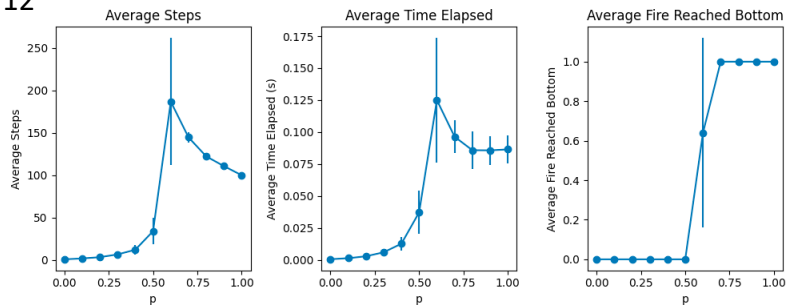
For this slide and the next two I display figures that have the same format. For all three slides, the simulations are repeated 50 times and average values are taken. Top figure shows the results for the serial program and the bottom program shows parallising the grid into different threads by using dynamic scheduling on each grid. This page is for dimensions 50x50, the next is 100x100 and the last is 500x500.

No significant time changes are displayed between the no parallisation and dynamic parallisation.

Speed tests for openMP

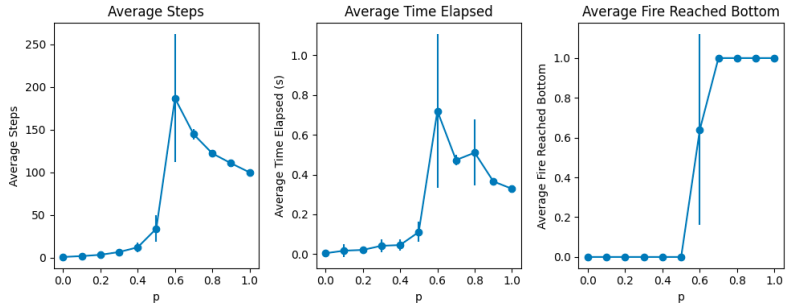
N = 100, M = 50

Fig. 12



No parallisation

Fig. 13



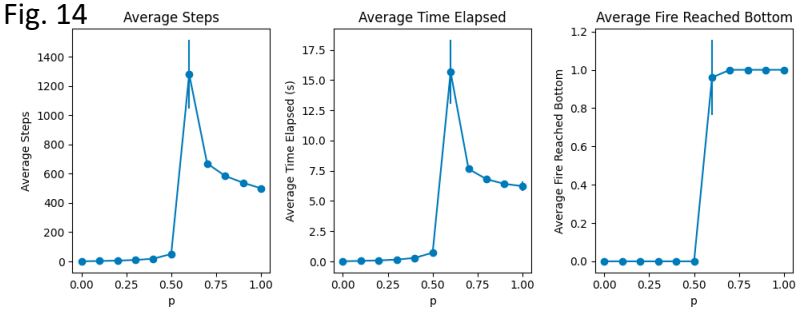
Dynamic Parallisation

Here, the dynamic program takes significantly longer, by 0.575 seconds.

Speed tests for openMP

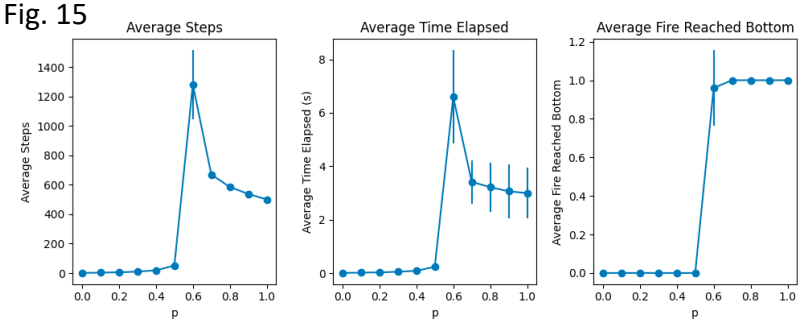
N = 500, M=50

Fig. 14



No parallisation

Fig. 15



Dynamic Parallisation

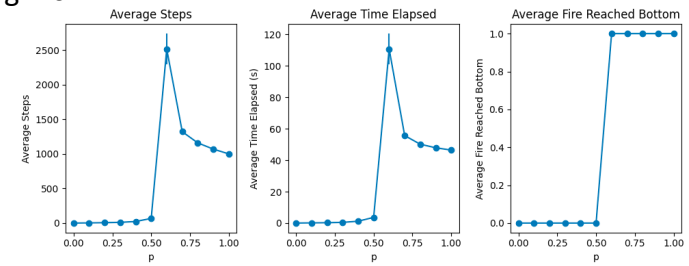
There is a significant speed up in the dynamic parallisation program. The dynamic program has a speed up of 9 seconds.

The results seen in the speed tests for openMP for N=50 indicate that parallisation in any configuration produces no change to the results. This is attributable to the small size of the grid making impact of the dynamic parallisation of the grid obsolete. The results seen in N=100 indicate that the time taken for the communication between the threads in the dynamic program outweigh any speed up effects, thus producing a worse outcome for the dynamic parallisation. For N = 500, the dynamic parallisation performs very well, with a significant reduction in time compared to no parallisation. Here the speed up due to the simultaneous computation of threads within a grid outweighs the communication time between threads. Therefore, the effectiveness of dynamic parallisation depends on the size of the grid.

Speed tests for openMP

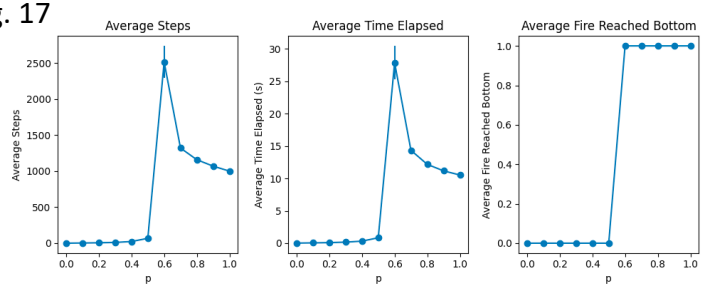
N = 1000 M= 10

Fig. 16



No parallisation

Fig. 17



Dynamic Parallisation

I decided to complete another speed test for N=1000 and M = 10 as this is where the best convergence was observed in the previous section. There is an even more significant speedup seen in the dynamic parallisation program compared to no parallisation. There is a speed up of 83 seconds, which is greater relative speed up than the 500x500 grid. An interesting concept to investigate if more time was available would be the relative speed ups of the dynamic program with respect to grid size.

For a relatively small grid size, the overhead of managing a large number of threads may outweigh the benefits of parallelism. A moderate number of threads (e.g., 4 to 8) can utilize available resources efficiently without excessive contention. As the grid size increases, there is more parallelism to exploit. Using a higher number of threads (e.g., 8 to 16) can effectively distribute the workload and enhance performance. Larger grid sizes offer more opportunities for parallelism. However, the law of diminishing returns applies. A careful balance is needed to avoid excessive thread overhead. Using a moderate to high number of threads (e.g., 16 to 32) could provide benefits up to a point, beyond which further increases might lead to diminishing returns.

Conclusion and Discussion

Model convergence was explored and discussed throughout the report. Understanding model convergence is crucial for robust analysis and reliable insights into the simulated phenomena. Model convergence refers to the stabilization of simulation outcomes as a function of key parameters, such as the number of repeat runs (M) and the grid size (N). The exploration of convergence provides valuable information on the reliability and stability of the simulated results. It is important to determine when the model converges because if one was running test on a simulation that did not converge all the results obtained would be unreliable in nature. Convergence increases in proportion to N and M, however you reach a threshold where further increases in N and M do not result in significant increases in convergence, and thus the computational expense for attempting to reach these levels of convergence outweigh the positive effects of having a more repeatable model.

The introduction of wind introduces an additional layer of complexity to the forest fire model, influencing the propagation and behaviour of the simulated fire. This is a step in the direction of making the simulation more realistic, however a more realistic simulation often scales in proportion to computational expense, and thus the same law of diminishing returns is encountered. One has to consider how realistic they want their simulation to be in order to produce the desired results. The forest fire simulation implemented in this report is not very accurate to reality, however it does serve the purpose of simulating an interesting model from a more mathematical perspective. Therefore, in this respect the simulation performs very well as it allows analysis of the problem under that light.

Dynamic parallelization, as implemented through OpenMP, plays a pivotal role in optimizing the computational efficiency of the forest fire simulation. In the context of this project, where the cellular automaton model is used to simulate the dynamics of a forest fire, dynamic parallelization offers a scalable and adaptive approach to harnessing the computational power of multiple threads. As previously discussed, it performs better on larger grid sizes and is actually a detriment to smaller grid sizes. This introduces nuance as to when and where the powers of parallisation should be implemented.

If more time and further research was put into this project, avenues such as enhanced environmental factors, advanced wind modelling, comparison with real world data and more sensitivity analysis could be explored in greater depth. Computation times could be negated by employing a supercomputer.