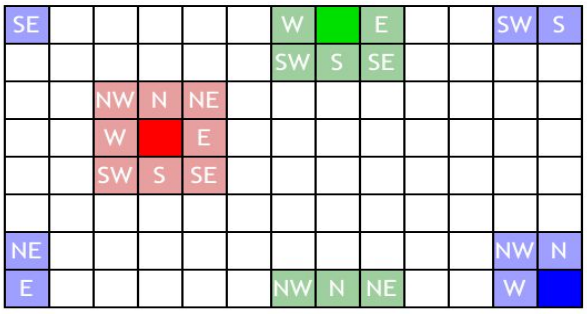
CS-301
GAME OF LIFE

RAJ JAKASANIYA 201501408
DHRUV PATEL 201501452

1: Brief Introduction

The Game of Life also known as Life is a cellular-automaton, zero player game, developed by John Conway in 1970. The game is played on an grid of square cells, and its evolution is only determined by its initial state.

The state of the game is a two dimensional grid, in our project we will connect the upper side to the lower side and the right edge to the left edge, resulting in a **toroidal structure**. Each cell in the grid has eight neighbors; we will denote these south, south-east, east, north-east etc.Each cell in the grid can be either alive or dead.



Life evolves in a sequence of generations. The state of a cell in one generation is determined by the state of the cells in the previous generation.We have four simple rules that determines the state of a cell in the next generation.

- Any live cell with fewer than two live neighbors dies, as if caused by under population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The first generation is determined by you, or by random, and the rules then determine all future generations.

Application of Game of Life:-
From a theoretical point of view, Life has the power of a universal Turing machine: that is, anything that can be computed algorithmically can be computed with Conway's Game of Life.
We will have the initial grid and we will observe it as it evolves by the rules. We can model many situations by changing rules appropriately, such as forest fires , Sieve of eratosthenes etc.

2: Hardware Details

CPU(s):	**16**
Thread per core:	**1**
L1d cache:	**32K**
L1i cache:	**32K**
L2 cache:	**256K**
L3 cache:	**20480K**

3: Profiling

Flat profile (on Naive Approach):

% time	Cumulative seconds	Self-seconds	Self-calls	Function name
82.91	105.90	105.90	2048000000	neighbours
16.96	127.57	21.66	2048	change
0.34	128.00	0.43		print
0.02	128.02	0.02		main

neighbours() function calculates the number of live cells in the neighbourhood of the given cell, it consumes most of our time so we have to improve that function.

Input/Output:
In input we have to give the number of rows and columns and also the number of iterations(or total steps).
Eg. ./a.out 1000 1000 10000
Here the 1000*1000 grid will run 10000 times.

For smaller grid size parallelization should not done because of very less number of computations involved. We have tried to maintain $10^8 - 10^9$ computations.
For output we can print the grid to show generation changes but grid being of large size (1000*1000) it is tedious to print the grid and see the generation changes on the screen.

4: Parallelization Scope

As every cell state depends on the state of its neighbours, game of life is embarrassingly parallel. Working on one single grid will result in wrong output because according to the rules a cell changes its state according to its state of old neighbours not the new one's for this reason we are using a temporary grid to store the result and then copy its content to the original grid.

Time Complexity of the serial code is **O(Row*Col*t_steps*constants)**.

Time Complexity of the parallel code is
O((Row*Col*t_steps*constants)/P).
Constants value depends on the algorithm used. Typical value of Row*Col = 10^6.

Cost of Parallel Algorithm:

The cost of a parallel algorithm is the product of its run time complexity of parallel algorithm (Tp) and the number of processors used p.
Cost = Tparallel $*$ P where T is the time complexity of parallel algorithm and P is number of Processors used.
A parallel algorithm is cost optimal when its cost matches the run time of the best known sequential algorithm Ts for the same problem.
Cost of Parallel Algorithm = **O(Row*Col*t_steps*constants).**

5: Parallel Overhead

Parallel Overhead = Parallel time(1 thread) - Serial time

Size	Naive	Optimized	Prefix	Tony Finch
256	0.157475	0.081688	0.05213	0.19616
512	0.318536	-1.564368	0.10792	0.35988
1024	0.658839	0.284162	0.22664	0.53273
2048	1.329185	0.691689	0.53702	1.261179

As the size increases the parallel overhead time also increases because as the no. of threads increases the communication time between threads also increases.

6: Algorithms Implemented

Naive Approach :-

The entire grid is scanned, the number of neighbours is evaluated for each cell after applying rules the new state of the cell is stored in temporary grid, and the entire grid is updated.
Extension of Naive Approach is the **optimisation approach** in which instead of looping the 8 neighbours we have 3 variables (c1,c2,c3) whose sum gives the number of alive neighbours, this 3 variables change their value for every cell.

V1 = 1 + 5 + 9;   v2 = 2 + 10 ; v3 = 3 + 7 + 10 ;
Neighbours of 6 = v1 + v2 + v3

1	2	3	4
5	6	7	8
9	10	11	12

For calculating neighbours of 7 :-
New_v1 = v2 + 6 ;
New_v2 = v3 - 7 ;
New_v3 = 4 + 8 + 12 ;

1	2	3	4
5	6	7	8
9	10	11	12

This optimisation trick saves us from looping through all the neighbours and helps to reduce the overall time.

Prefix Technique:

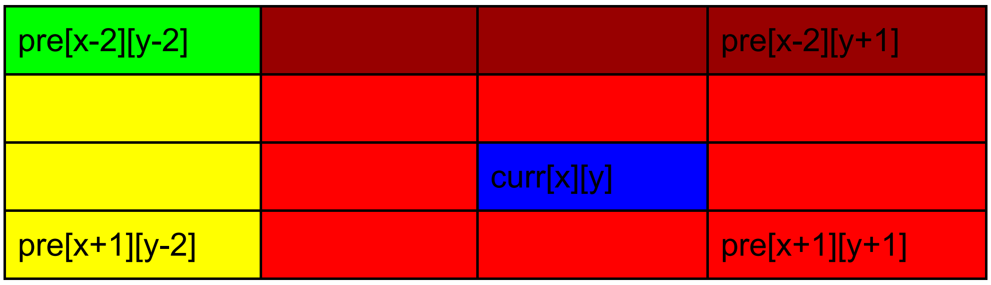
In this technique firstly we are giving a padding of 2 (i.e. 4 extra row and 4 extra columns) to the grid.

Every time before calculating next generation cell’s we fill the prefix matrix using formula :-
prefix[x][y] = curr[x][y] - prefix[x-1][y-1] + prefix[x-1][y] + prefix[x][y-1];
Here prefix sum will give no. of cells alive in the grid (x,y).

Now this prefix matrix is used to calculate the number of alive neighbours for every cell using formula :-

live = prefix[x+1][y+1] - prefix[x-2][y+1] - prefix[x+1][y-2] + prefix[x-2][y-2] - curr[x][y];

The live variable will give us the no. of neighbours alive for cell (x,y) by using above equation.



Prefix Technique helps us to avoid calculating neighbours every time for every cell of grid for next generation.

Tony Finch technique :-

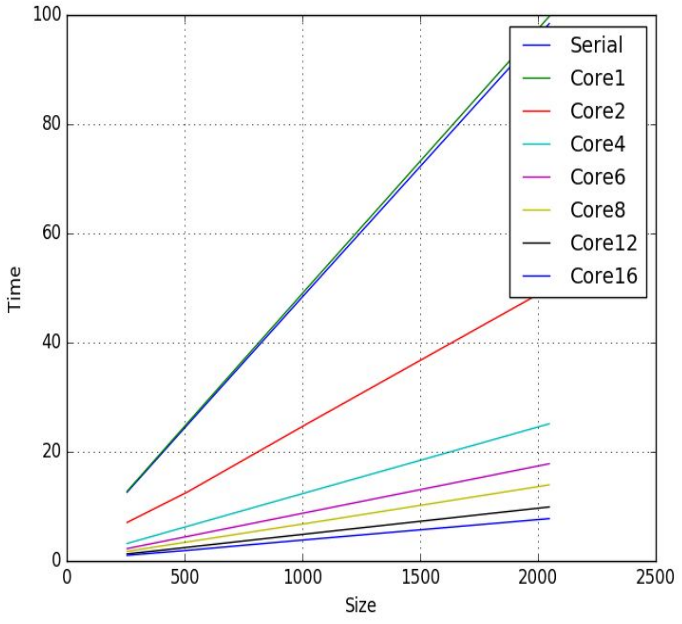
This technique involves one time calculation of no. of alive neighbours for every cell in a 2D matrix named tem.
Now while iterating through generation we change state of cell according to tem matrix and update neighbours count in duplicate matrix of tem (named next) i.e. if a live cell dies then we will subtract 1 from each of its

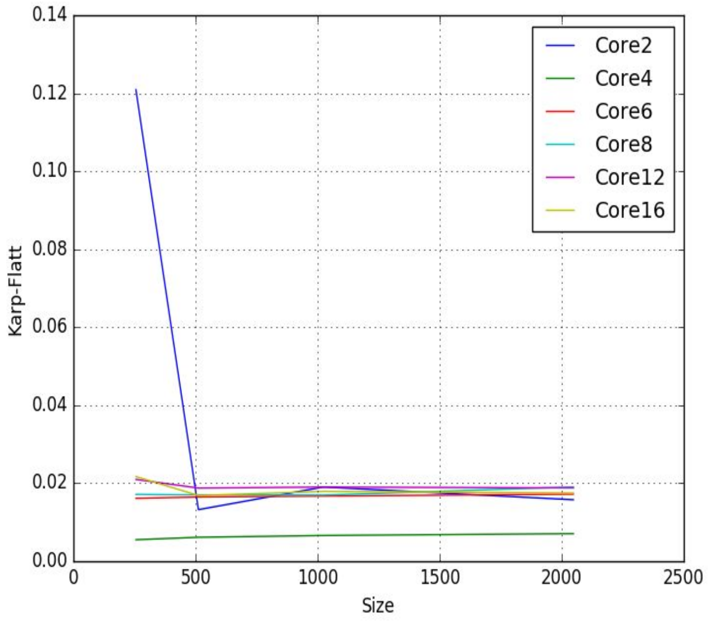
neighbour in matrix and store all this in next matrix and lastly we copy all this data to matrix tem.

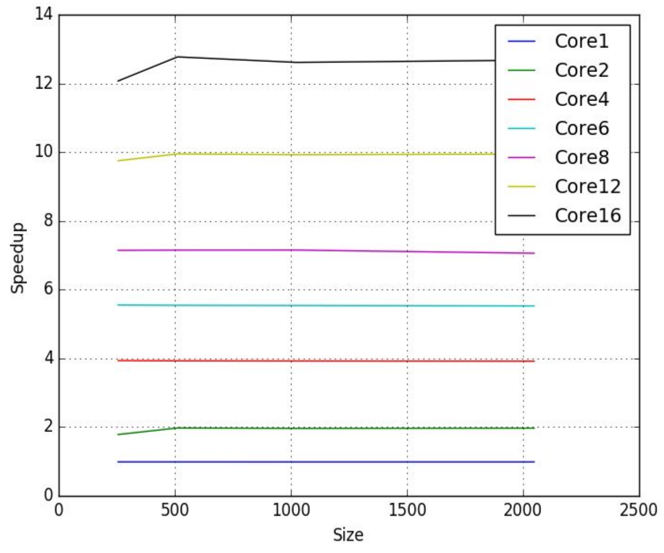
This way we just have to update neighbours of cell which change their state which reduces the computational time drastically.

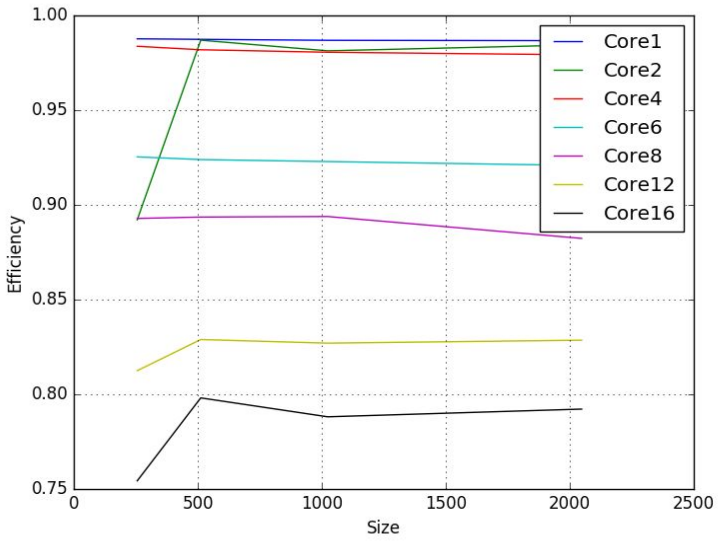
7: Graphs

Naive Approach:



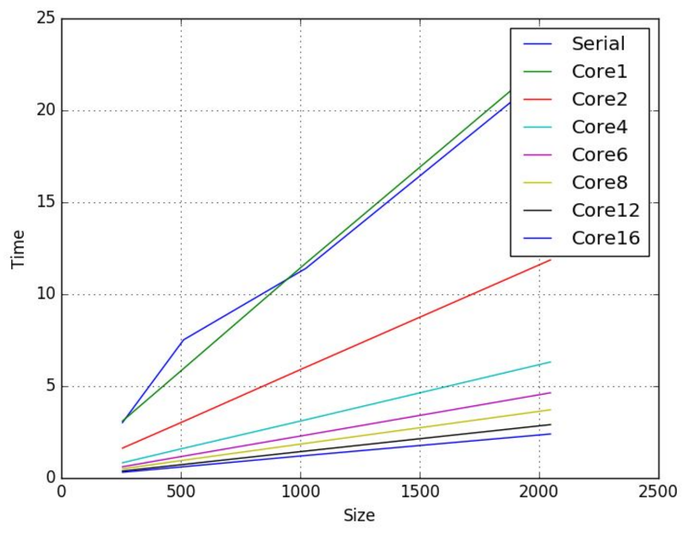


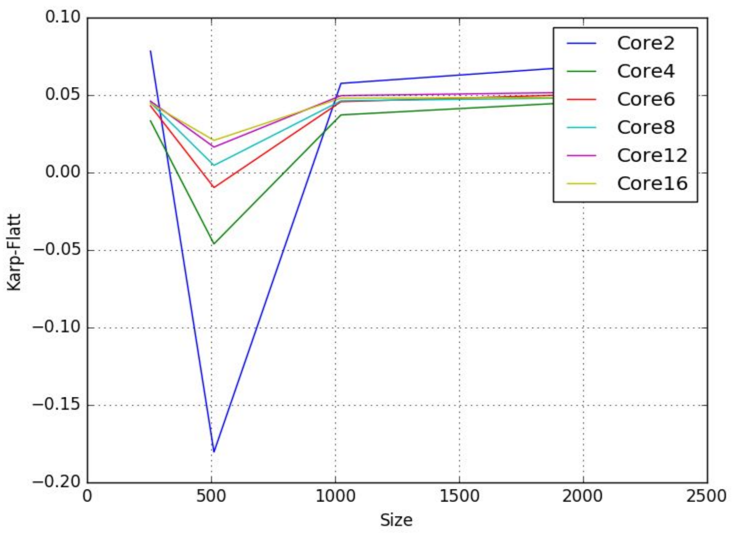


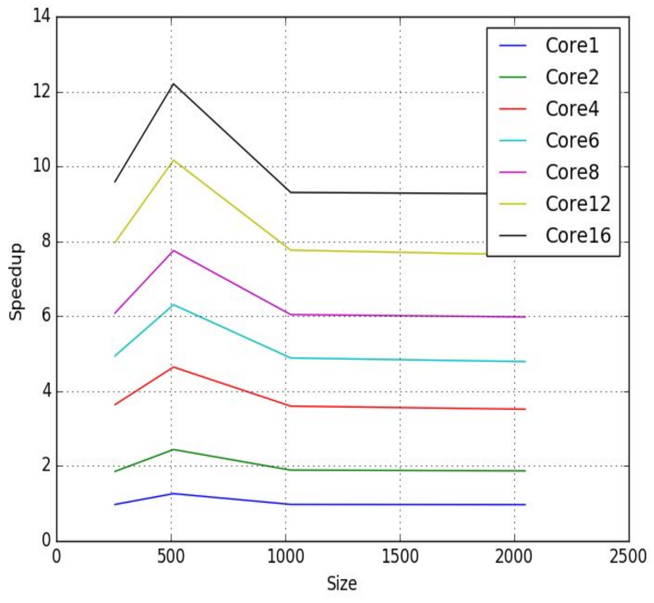


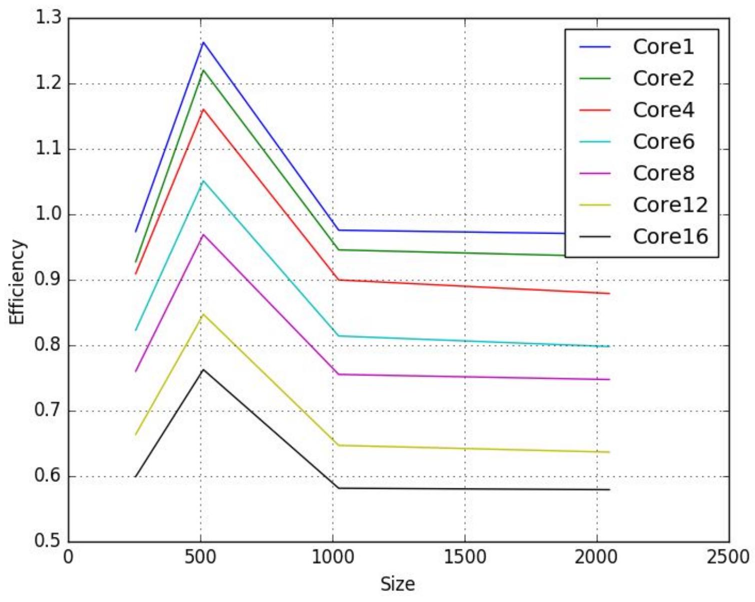
Execution time for parallel code with 1 thread is more than execution time of serial code because of large parallel overhead (creation of thread, synchronization of thread etc).

Optimise Approach:



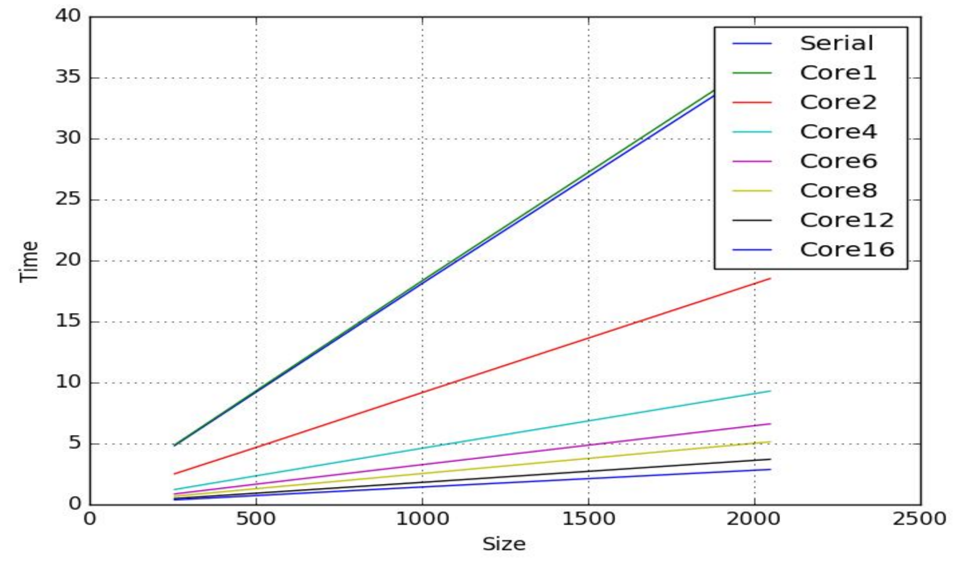


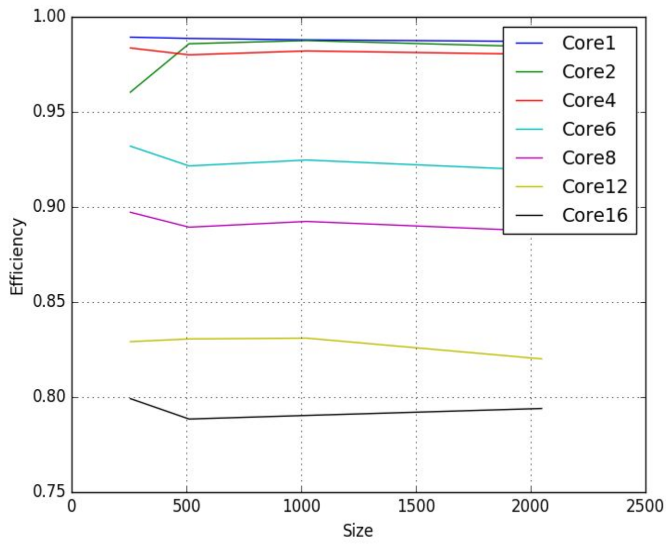


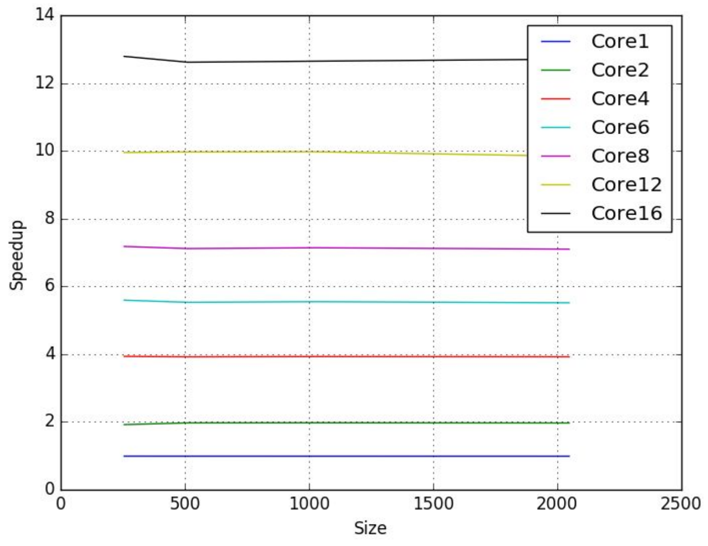


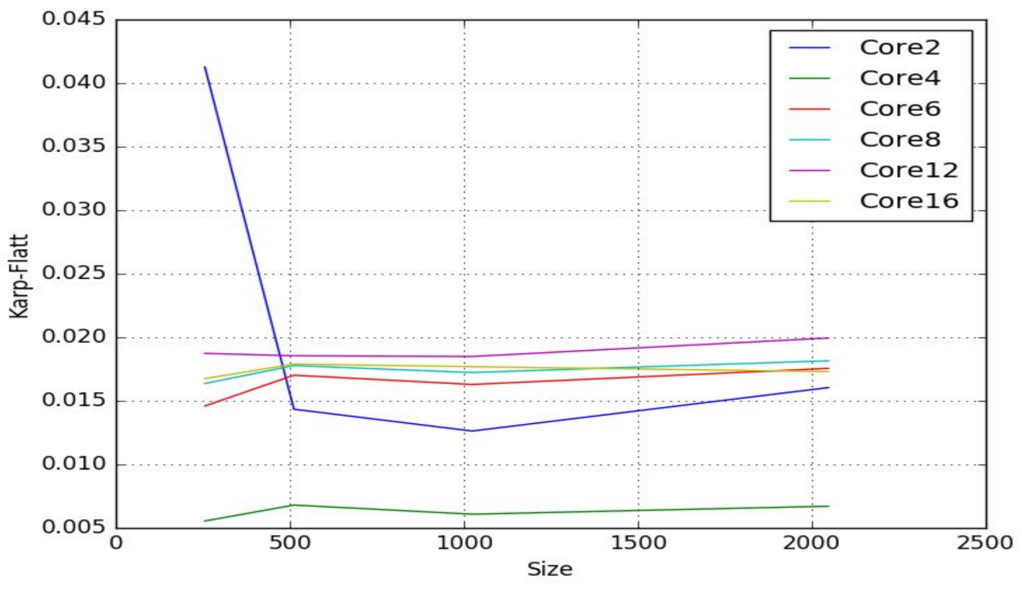
For step size of 512 we have a huge increase in speedup curve, reason being that for step size of 512 it is present in faster cache (i.e. L3 cache) and for size 1024 it overflows from L3 cache resulting in large parallel execution time contributing to very small speedup. Due to large speedup for 512 step size the efficiency also increases.

Prefix Approach

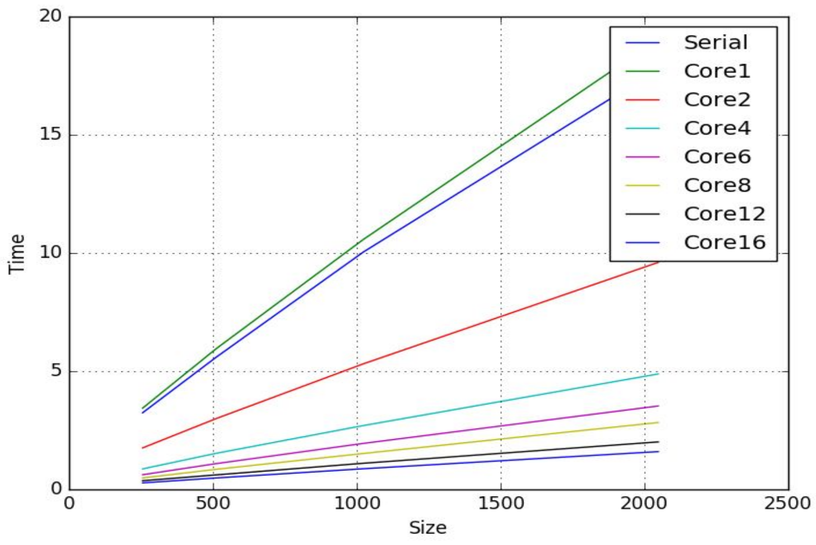


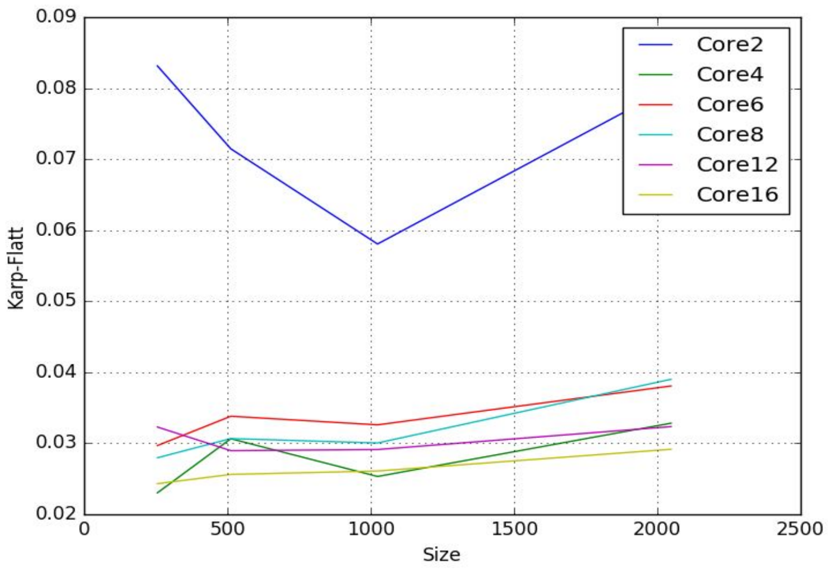


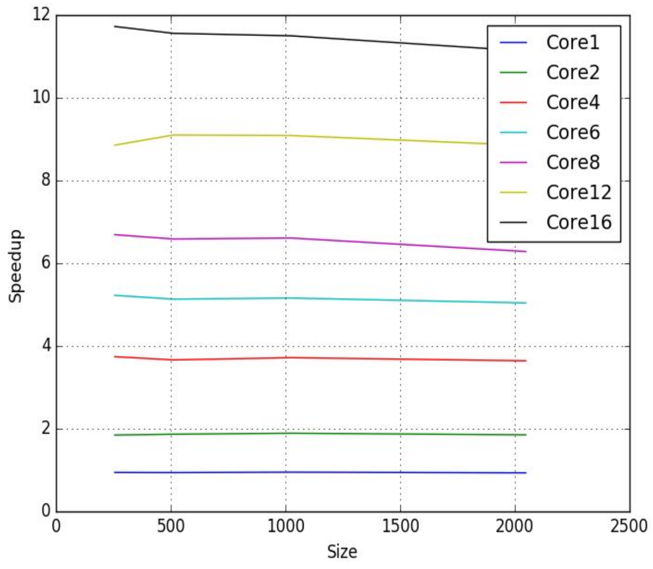


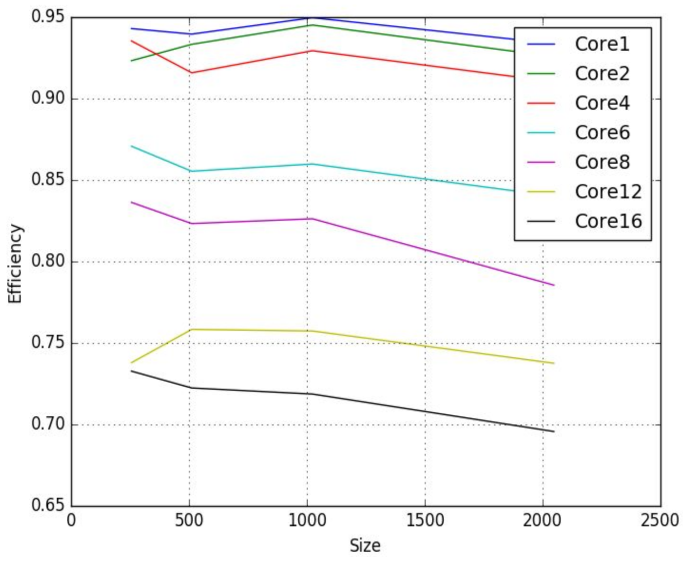


Tony Finch Approach









Tony Finch implementation is nearly 4 times faster than the Naive approach.

8: Observation:

No. of Memory Access:
As we are traversing through NXN matrix, so memory access is **O(** $n^2 * steps$ **).**

Granularity:
Ratio of computation to communication: There is more communication between the processors for calculating the neighbours because we have to calculate the neighbours from three different rows, so there is a dependency. Therefore the granularity is Fine.

Time Curve related analysis

Number of processors and the execution time are inversely proportional i.e. with the increase in the number of processors there is a decrease in the execution time,it is because with the increase in the number of processors the work is divided among more number of threads and it is done parallely i.e. at the same time which leads to decrease in the execution time. But for every code there is a optimal number of threads after which increase in the number of processors leads to the increase in the execution time, this happens because after the optimal number of threads the parallel overhead becomes much larger which cancels out the effect of decreasing the time by dividing work among threads and leads to a increase in the execution time.

With increase in the problem size there is increase in the execution time.For serial :- Increase in time with increase in problem size is very steep For Parallel :- Increase in time with increase in problem size is very steep and gradually becomes calm or gentle with increase in the number of processors.[For parallel code running with 1 thread parallel overhead is much more which leads to more execution time than the serial code]

Speedup Curve related analysis

Speedup = $\frac{T_{serial}}{T_{parallel}}$

$T_{parallel}$ = $\frac{T_{serial}}{P}$ Where P is the number of processors

Theoretical Speedup = P (Number of Processors used)

Actual speedup = $\frac{1}{\frac{p}{N} + s}$

Where the N is no of cores, p is parallel fraction and s is serial fraction.

For Serial :- From above formula we get Speedup = 1

For Parallel :- With the P processors, execution time= T_serial/P,
Which leads to Theoretical Speedup = P.
But in real time taken > Theoretical time taken which leads to **real speedup < P** , which implies with the increase in the number of processors there is increase in speedup. But For smaller Problem size there is very large parallel overhead which leads to very large time for parallel execution time which leads to a very low speedup which can be seen in the graph.

Here we are getting the maximum speedup of ~13 (for 16 core). As we can see from graph as the size increases we are getting almost constant speedup, which indicates that our problem is **strongly scalable** implying it does not change much with the problem size.

Efficiency Curve related analysis

The definition of parallel efficiency is defined as the speedup divided by the number of units of execution (processors, cores): E = S/P Efficiency should be nearly 1.

Efficiency greater than 1 implies a superlinear speedup.

With the increase in problem size number of computations increases without much increase in overhead,hence we get more efficiency. Also, efficiency decreases as number of cores increases for given problem size.

Karp-flatt Curve related analysis

Karp-flatt is a measure of parallelization of code in parallel processor systems, in simple language how further we can parallelize the code. This implies that the smaller the value of Karp-flatt, less is the serial fraction and better is the code. Almost in every approach we are getting the constant value of Karp-flatt which indicates that we cannot parallelize the code further.

From all the approaches done the best implementation is that of the Tony Finch.

9: Further Works

- Hash Life (Recursive quad tree algorithm)
- List data structure

10: Reference

- https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- http://www.conwaylife.com/
- http://web.stanford.edu/~cdebs/GameOfLife/