Project-HPC-Conway's game of life

Vaibhav Patel - 201401222
Tanmay Patel - 201401409

1: Brief Introduction:

Conway's game of life is an interesting computational science problem. It is a cellular automaton based on some rules. Cellular automaton is a grid having cells and some finite states.

Now, why this problem is important?

Because it has capability of a Universal Turing Machine, it means that any algorithm can be transformed into game of life. But, we have focused only on the original game published by Conway.

We have a grid and we will observe it as the system evolves. We can observe rise and fall of the society. We can model many situations using this simple grids. Such as terrorist attack in a mall, etc.

2. Algorithm:

For understanding the algorithm, we first need to know the rules of the game.

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

Source: wikipedia

n= number of iterations we want.

The algorithm is very simple. We just calculate number if living and dead neighbors of every cell. And then base on these rules we will change the state of this cell in the next step. Here update should reflect in the next step (after calculating over all the cells) in the next step is important. **Instead of padding dead cells at border, we are considering grid as a toroid (sort of). The last and first row is neighbors of each other.**

Running time complexity of the naive algorithm:

==> O((8+4) n^2)

We can think of other alternative algorithms for this problem.

Input/Output:

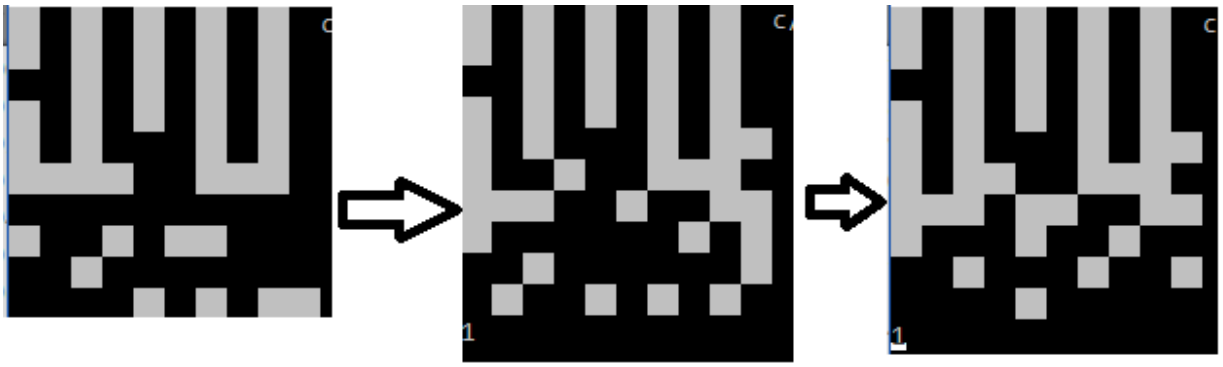
This is a game but it is a no player-game. Here the input is just number of rows, number of columns, number of steps of life we want.

So, a typical input may be 100 100 10000

Here, the life will run 10000 times on a 100*100 grid.

Output is also nothing but a state of the board evolving as the step increases.

Example:
Here a output for 10 10 2 input:



We had option of showing this output in our code, but it slows down the process. So, we comment it out. Here we can see how the life is evolving.

Maximum Theoretical Speedup:

Possible speedup = P (processing elements)
Speedup = 1 / (p / N + s)
Where N is no of core. p is parallel fraction. s is serial fraction.
In this problem serial fraction is very less so the speedup should also scale as processor increases. (The relation should be y=mx where m is close to 1 (from the left side)).

3.Profiling:

Flat profile:

% time	Cumulative seconds	Self-seconds	Self-calls	Function name
86.32	5.21	5.21	100000000	get neighbors
13.97	6.06	0.84	10000	update
0.17	6.07	0.01	-	show

Here, we can see that most of the time consumed inside the get_neighbors () function. So, we should try to improve it. Other thing is that it is called 10^8 times which can be a problem!!!

Hardware details:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:   0-11
Thread(s) per core:    1
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
Stepping:              2
CPU MHz:               1264.218
BogoMIPS:              4804.38
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              15360K
NUMA node0 CPU(s):     0-5
NUMA node1 CPU(s):     6-11
```

4.Scope of the parallelization:

This problem is embarrassingly parallel for most of the portion of the grid. But we have to be careful at the edges. It can be done by simply putting it in critical section. And another interesting thing about this problem is that every cell depends on its neighbor, which is a spatial property. So, we can try various different parallel implementations.

From the previous experience doing work on the same grid would have slow down the code, because the processor will try to maintain the cache coherency at the cost of speed.

So, we started with another temporary grid which is used for storing the next state. This is very convenient in terms of implementation and optimization. And here we are focusing on speed not on memory so this change is anyway justified.

We improved the naïve serial implementation to very much extent that optimized serial is ~5

times faster than naive serial.

Optimization tricks:

Time Complexity of the code is O(r*c*steps*constants)
Let's rename r*c*steps as n.
From the profiling results we came to know that the main bottleneck for the code is get_neighbors function. So, we concentrated on that part first.

→These lines are running 8n times.

```
int ni = (i  + dx[k] + r)%r;
int nj = (j +  dy[k]+c)%c;
```

We read about modulo operator and we tried another method:

```
int ni = i + dx[k];
int nj = j + dy[k];
if(ni < 0) ni += r;
if(nj < 0) nj += c;
if(ni >= r) ni -= r;
if(nj >= c) nj -= c;
```

And benchmarking results are (on single core):
Old = 0.988
New = 0.673
1.49 times faster.
Cumulative=1.49

→We were copying data from next state into current state every time but it can be done by just changing the 2d pointers. But, it is a small improvement.
→ In profiling there we so many calls to the get_neighbor() function so we merged the two function. Here, we are losing an important aspect of programming practice i.e. code should be Modular. This improvement will make the code difficult to read and debug.
Speed improvement:
Old = 0.673
New = 0.638

1.05 times faster.

Cumulative=1.564

→ Because of the borders, we have to check that ni and nj are not out of bound. This leads to so many ifs. This checking is only needed for the border; inside this grid we are free to calculate the neighbors without worrying about out of bound condition.

Speed improvement:
Old = 0.638
New = 0.445
1.443 times faster.

Cumulative=2.257

Now, the code became 3 times larger than the original one. And is difficult to read.

→ Now, each time calculating all the 8 neighbors is not optimized. So, now we are using only 3 variables v1, v2 and v3. Now in every iteration we will just calculate neighbors. Which will make it faster.

1	0	1	0
0	0	1	1
1	1	1	0

V1 2	V2 1	V3 3

alive_neigbours = v1 + v2 + v3;

v1 = v2;

v2 = v3;

v3 = 0;

//calculate v3 again

Speed improvement:
Old = 0.445
New = 0.272
1.636 times faster.

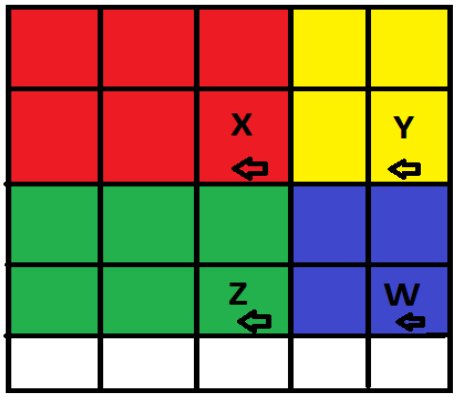
Cumulative=3.69

→ Now the last optimization is that we explicitly unroll and hardcoded the neighbor calculating algorithm. Before it was getting indexes from an 8*1 array because now it is hard coded it will be accessed from registers instead of memory or cache.

Speed improvement:
Old = 0.272
New = 0.203
1.339 times faster.
Cumulative=4.944

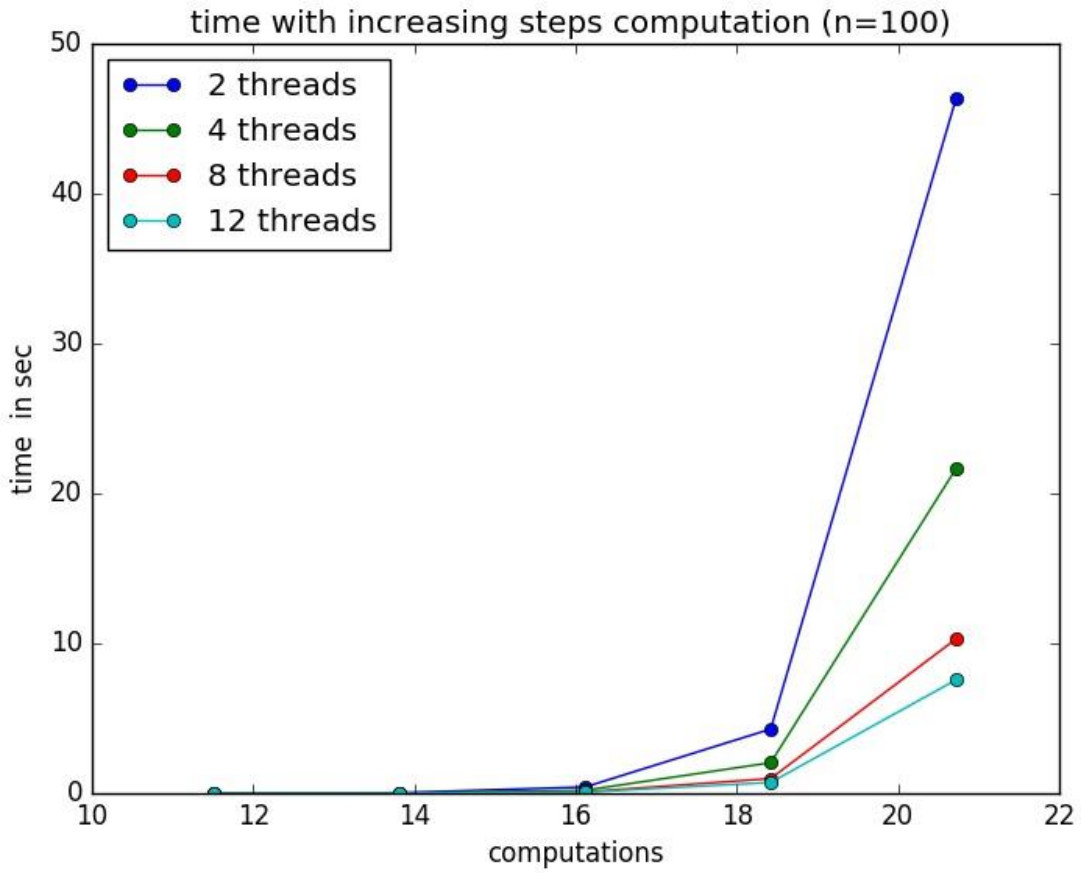
→ There were so many other minor improvements were done on the serial code. **There were mainly on the idea of cache reuse. Even if we want to access a same element for 2 times we used a new variable so that second time it is used from register (because it is faster than cache).**

→ We've implemented another algorithm for calculating neighbors and it is also 4 times faster than the serial algorithm without this lengthy code. But the other version is nearly 5 times faster that is why we are parallelizing that version.
And that method is prefix sum over matrix.

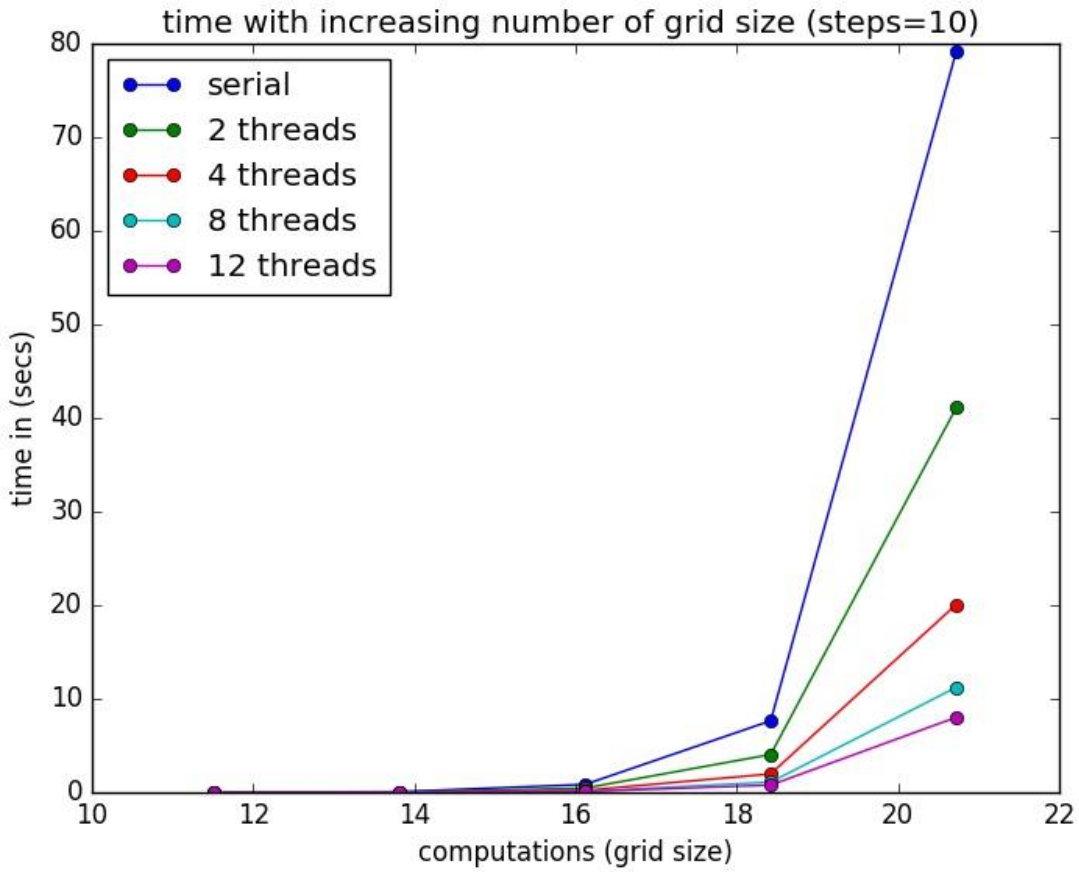


Blue portion= W -Y -Z+X

5.Speedup curves: **Problem size vs time:**

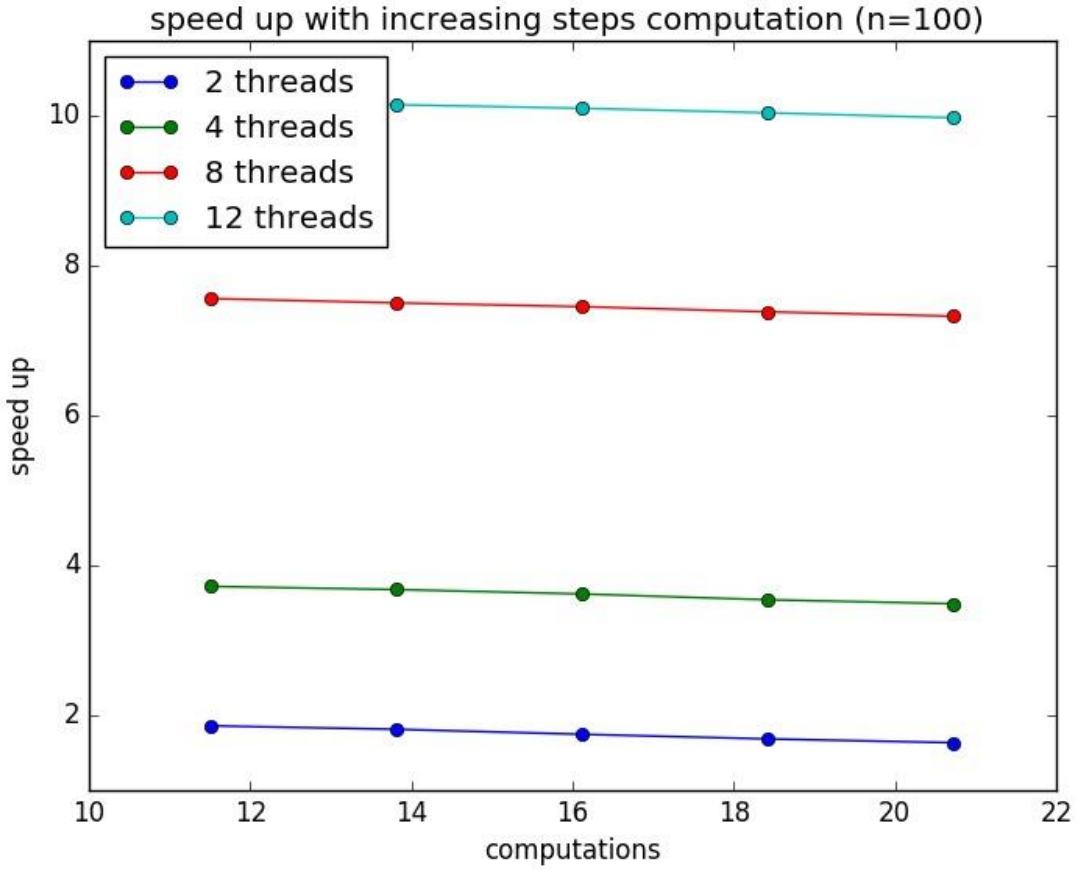


Above graph is a simple relation between time in sec and increasing computation by increasing **step size.** Time taken by program is not helping much when we want to benchmark any implementation. Here we can see that the time is decreasing as we increasing thread count but more insightful idea can be taken if we consider any other benchmarking technique such as efficiency, speedup, karp-flatt metric etc.



In this graph also the time is more for fewer number of threads.

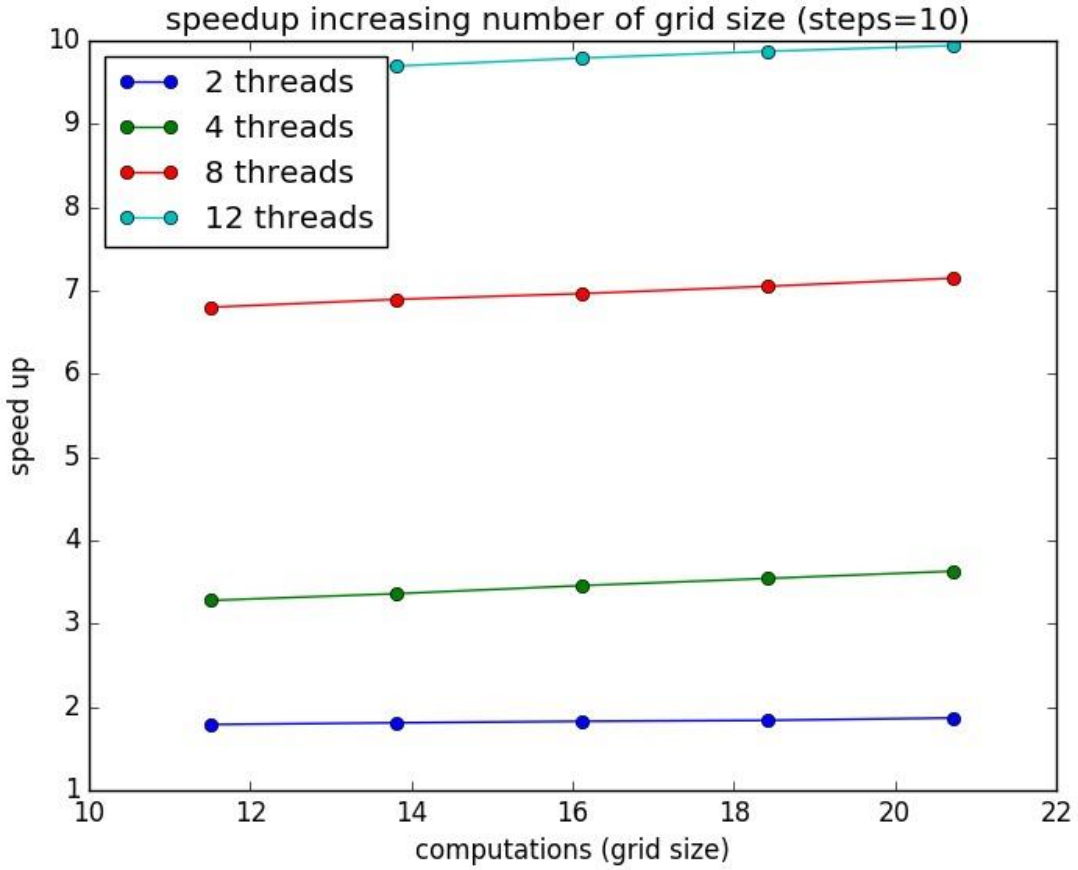
Speedup vs step-size:



While parallelizing, we should focus on outer most loop.
In our problem, the main loop structure is:

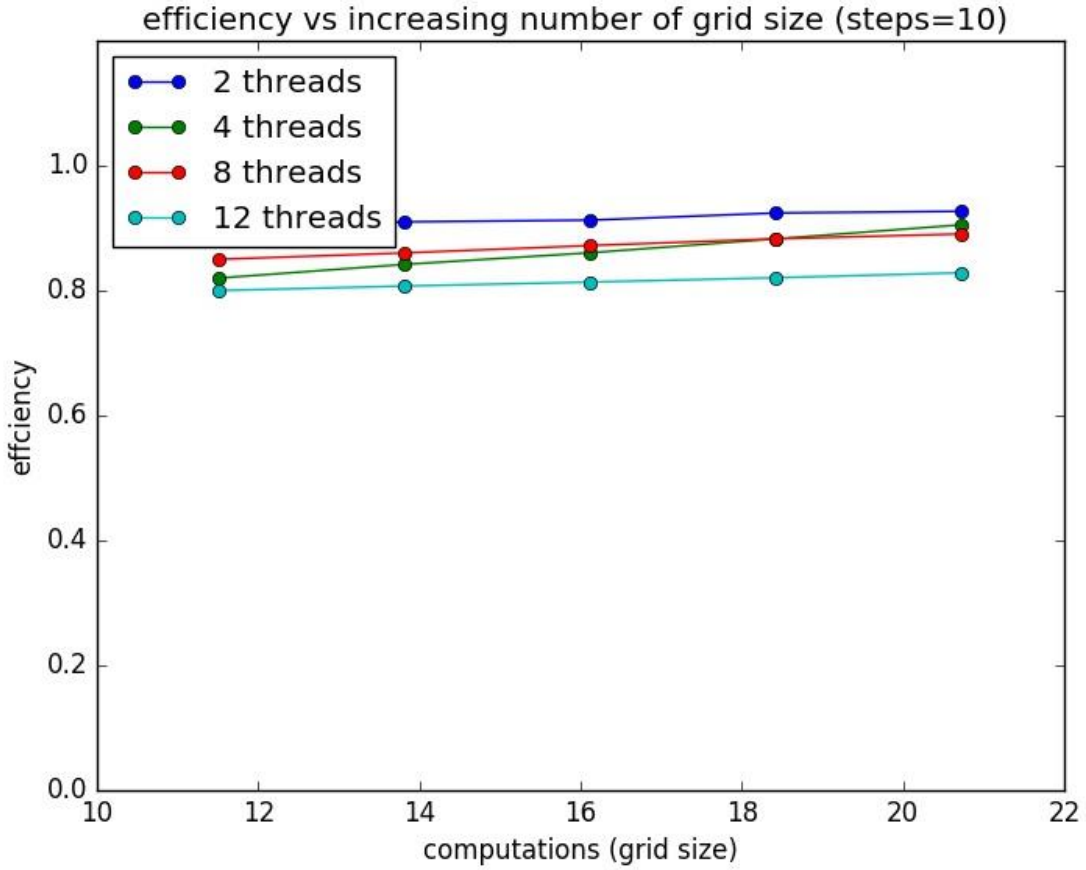
```
for(steps = 0; steps < total_steps; steps++){
    for(i = 0; i < r; i++){
        for(j = 0; j < c; j++){
            //code here
        }
    }
}
```

Here, next state is dependent on the previous state. So, we can't parallelize the outer loop. Anyway, we want to see life evolving so we should not try to parallelize it. In the above graph, grid size is 100*100 and step size is increasing. Here threads are launching at every step, which is **why the problem has become slower as the step size increases due to overhead in launching threads.**



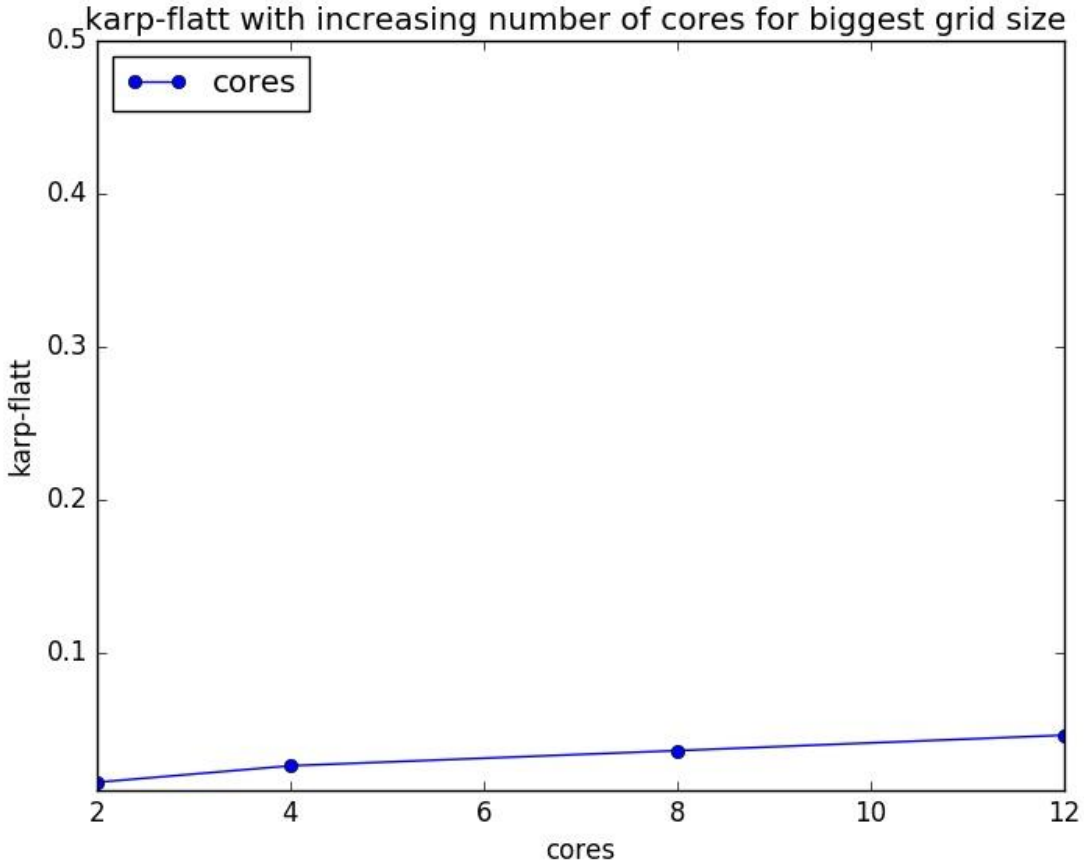
In the above graph, grid size is increasing starting from 100*100 and the total steps is 10. **Due to Amdahl's law and our problem being embarrassingly parallel (as a function of grid size) the speed up is increasing.**

Efficiency vs problem size:



As the speedup was increasing in the above graph the efficiency is also an increasing function of grid size.

Karp-Flatt metric vs problem size:



e is increasing as number of cores are increasing. But increase in e is very small. In our implementation, the inherently serial part is negligible. And this graph also shows that e is steadily increasing due to overhead. As number of cores increases the overhead increases. Giving data to all the threads. Whenever two threads try to access adjacent rows there can be situation where processor will try to maintain cache coherency. And which will lead to slow performance. We tried to parallelize the edge part of the code but it is cumbersome and not worth the effort. Its effect is negligible.