HPC Project
Plotting Mandelbrot Set

Utsav Patel
Id: 201501443

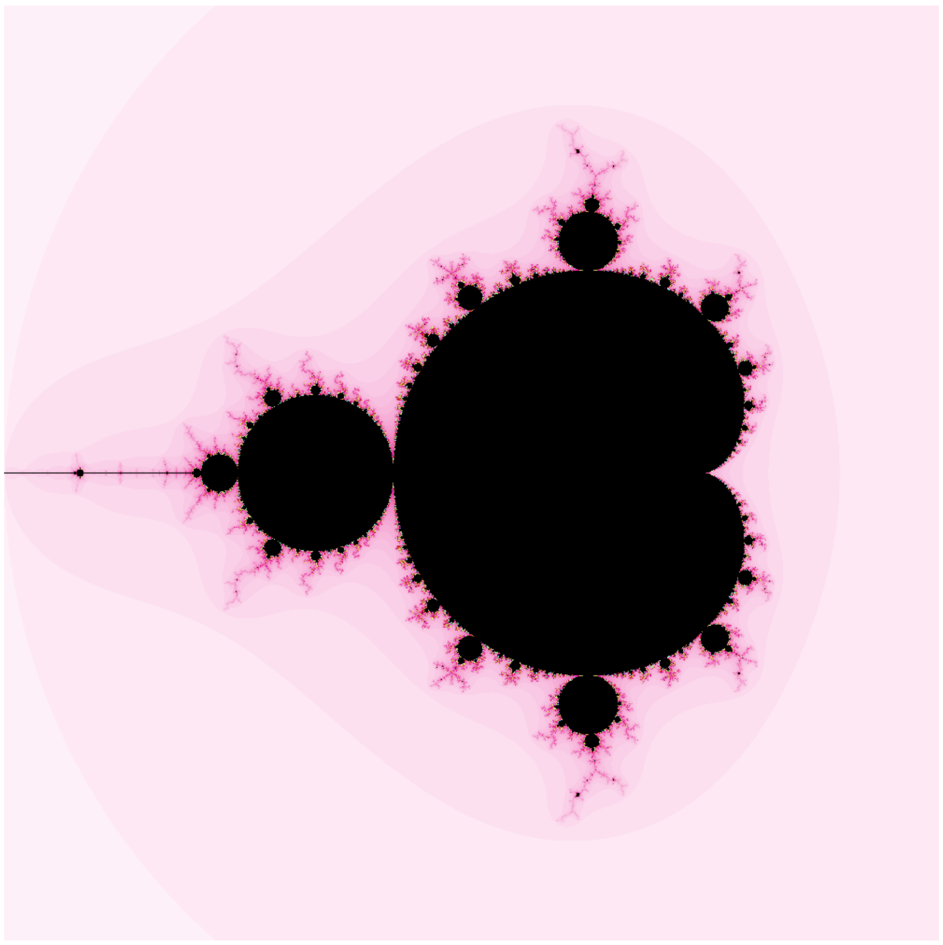
Meet Rayvadera
Id: 201501444

October 30, 2017

1 Introduction

Mandelbrot set is fractal structure, which is set of complex numbers c for which sequence $z_{n+1} = z_n^2 + c$ does not diverge, when iterated from $z_1 = 0$. Plot can be obtained by sampling complex plane according to pixel of image which we want to generate and then coloring that point according to how rapidly it converges. Black color will denote that point will belong to set and white point will denote point doesn't belong to set because for that point sequence is diverging. We have set bound of 2, so for any point to be in set, absolute value of terms of sequence should be less than 2.

For example below is Mandelbrot set for range of x between -2 to 1, range of y -1.5 to 1.5, pixels 1001*1001 and 500 iterations i.e. 500 terms of sequence for each point.



Region between points for which sequence is directly gets unbounded and point for which sequence remains bounded till end, shows some beautiful patterns. Some of these zoomed images from our code are shown below.

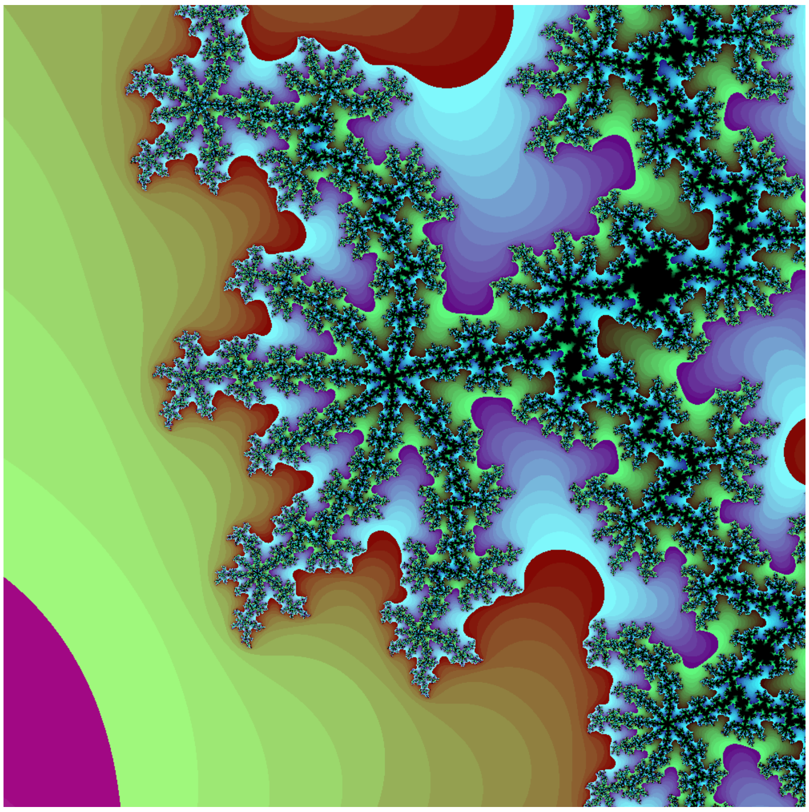


Figure 1: X-axis limit : 0.139 to 0.14, Y-axis limit : 0.639 to 0.64, Iterations : 120

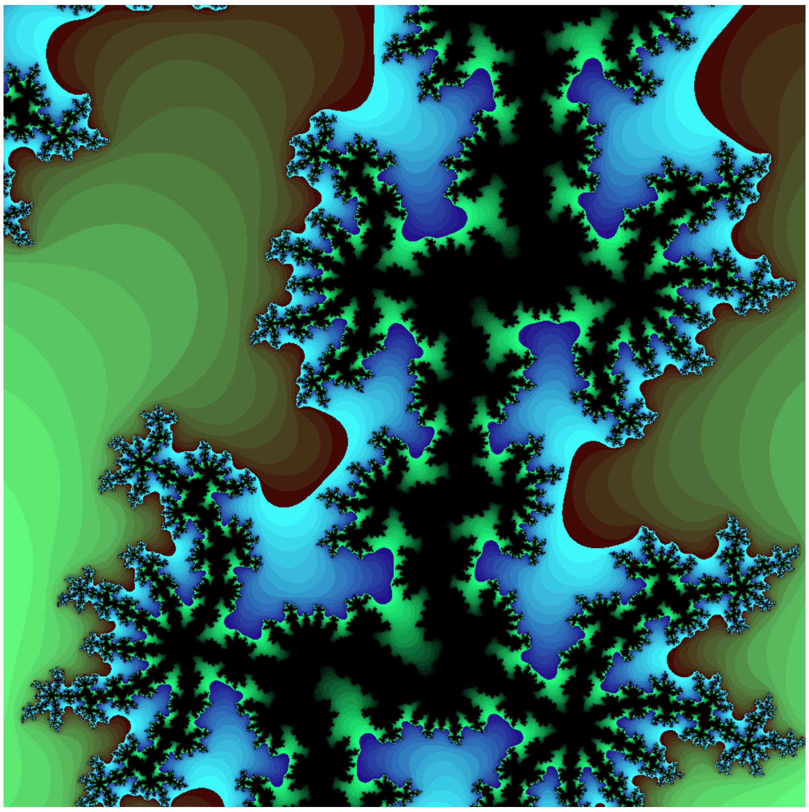


Figure 2: X-axis limit : 0.13945 to 0.1395, Y-axis limit : 0.63945 to 0.6395, Iterations : 120

As we increase number of iterations, shape and coloring of set will change like this.

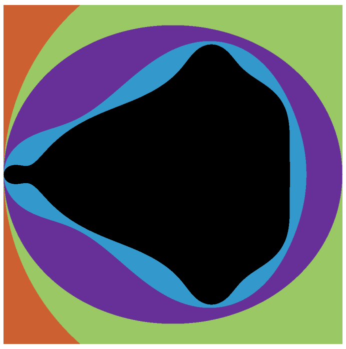


Figure 3: Iterations : 5

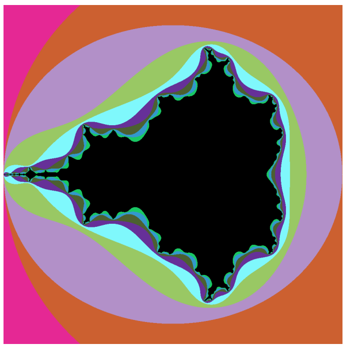


Figure 4: Iterations : 10

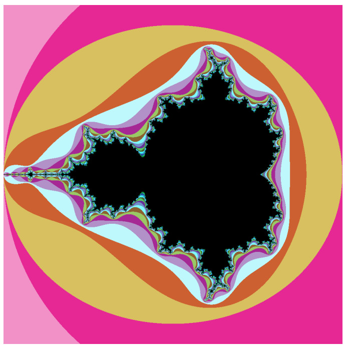


Figure 5: Iterations : 20

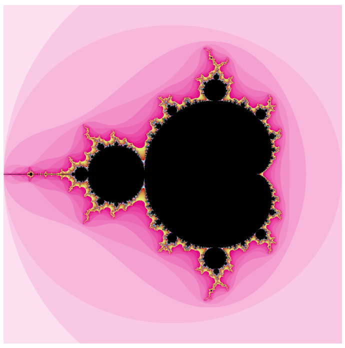


Figure 6: Iterations : 100

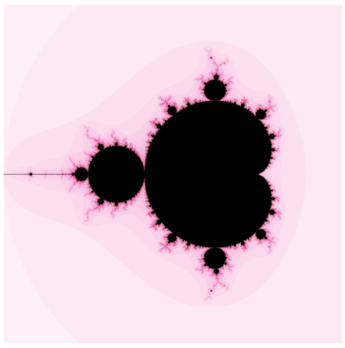


Figure 7: Iterations : 500

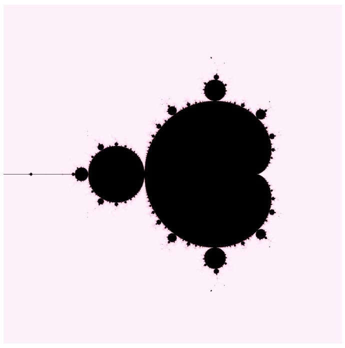


Figure 8: Iterations : 5000

1.1 Why it is computationally expensive?

Exact result will be known only when we take infinite terms of sequence because we can't know behaviour of sequence at infinity, but that is not possible, so we will take high number of iterations. All these iterations we will have to do for each point of complex plane. So it would take lot of computations. Computation will be less in case sequence get diverged for earlier terms of sequence at particular point so that we don't have to calculate all further terms.

2 Implementation Details

2.1 Brief description about the Serial implementation

For plotting Mandelbrot set on complex plane, we have to check sequence for each point (starting from that point) in complex plane, whether sequence remains bounded in absolute value or not. So for that we first have to iterate through all points of plane, number of points will be specified from input(n). We will fix the range of complex plane, -2 to 1 on X-axis and -1.5 to 1.5 on Y-axis. Number of terms in particular sequence for each point is taken to be 120. So for each point, max iterations would be 120. For sequence of each point, if sequence fells outside circle of radius 2 then there won't be need to calculate further terms of sequence. Now we have to decide the color for each point such that it tells us about how much sequence for that particular point remains in bound. For that first normalize number of iteration till when it remained in bound to 0 to 255. We want to show zero color for unboundedness and black color for boundedness. So subtract that number from 255. Now give this number to R,G,B values differently to get different patterns for points where iterations are neither too small nor too big. Finally after deciding R,G,B values for each point, put it on the plot.

2.2 Brief description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

We will have computation for each point, so we will divide number of rows into different cores by inserting Pragma on outer loop. Here we have to declare iterating variables to be private and Number of Iterations array and PPM image to be shared (So each thread can access it).

Number of points divided between cores would be almost equal but number of iterations for each point might be different (as iterations will be stopped once value goes out of bound.)

3 Complexity and Analysis

3.1 Complexity of serial code

We have to iterate through all points, but for each point it's not necessary that it will take max iteration(max terms of sequence) as sequence can fall outside of circle of radius 2. So complexity would be $O((NumberOfPixels)^2 * (MaxIterations)) = O((n^2) * m)$.

3.2 Complexity of parallel code (split as needed into work, step, etc.)

Now we are dividing rows into different cores, so first few rows will be calculated by first core, second few by second and so on. Assuming equal load in each core, it would be $O((NumberOfPixels)^2 * (MaxIterations))/(NumberOfCores)$ $O((n^2) * m/p)$.

3.3 Cost of Parallel Algorithm

Cost = (Time for parallel algorithm) * (Number of cores). We aren't getting super linear speedup in any case, so this cost will be always greater than time for serial algorithm.

3.4 Theoretical Speedup (using asymptotic analysis, etc.)

Theoretical speedup = $\frac{1}{1-pf+\frac{pf}{p}}$, where pf = fraction of code which can be parallelized and p = number of cores.

Speedup would be less than theoretical speedup in any case. If we keep on increase number of processors, speedup will saturate at some point as

$$\lim_{p\to\infty} \frac{1}{1-pf+\frac{pf}{p}} = \frac{1}{1-pf}$$

3.5 Tight upper bound based on Amdahl's Law

Assuming whole code can be parallelized, so pf=1. So tight upper bound of speedup = p (Number of cores).

3.6 Comparison of Number of memory accesses between Serial and Parallel

Number of memory accesses would be higher because in each iteration we have to access array of iterations and PPM image from shared memory. One major benefit of multiple cores is that we will have more Caches available as each core has will have separate L1 and L2 cache, so after dividing work into parallel core total memory access load to master core will decrease. **But** in our case, it won't happen because we are not reading anything from master thread, we are just writing into it. So Cache won't be of much use and number of memory accesses would be almost same between serial and parallel.

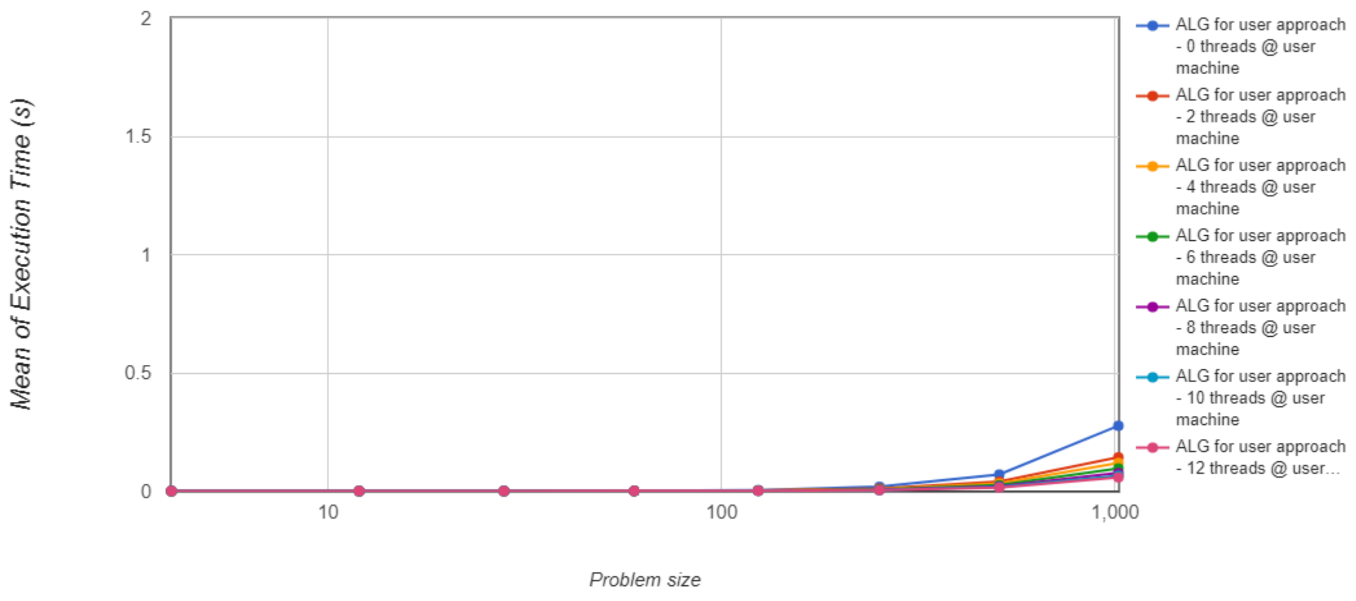
4 Curve Based Analysis

4.1 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:	1231.500
BogoMIPS:	4804.57
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

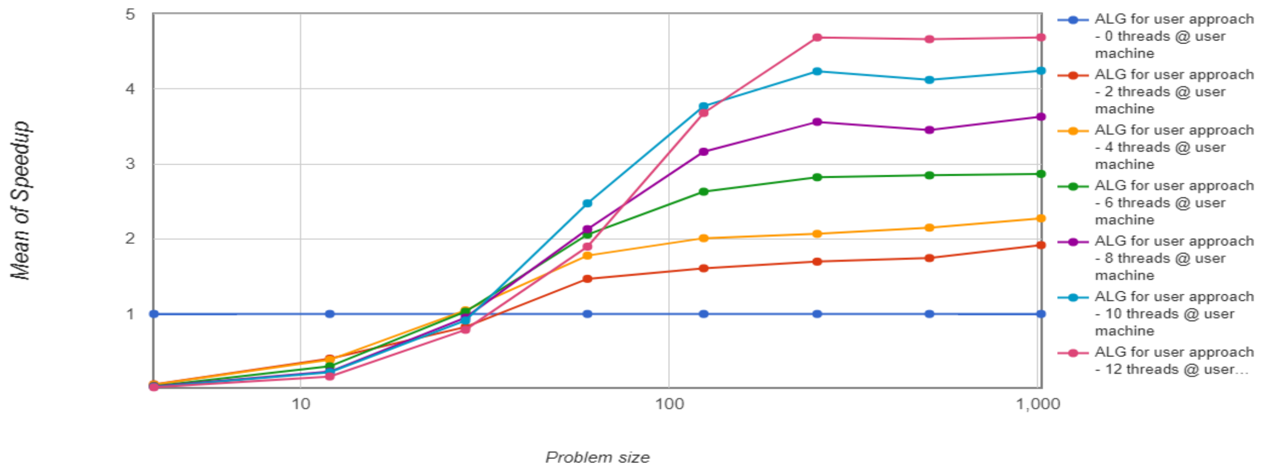
4.2 Time Curve related analysis

As we can see in below graph, execution time will decrease as number of cores increase. For smaller problem sizes there won't be much difference in execution time of serial and parallel while as problem size increases, there will be difference.



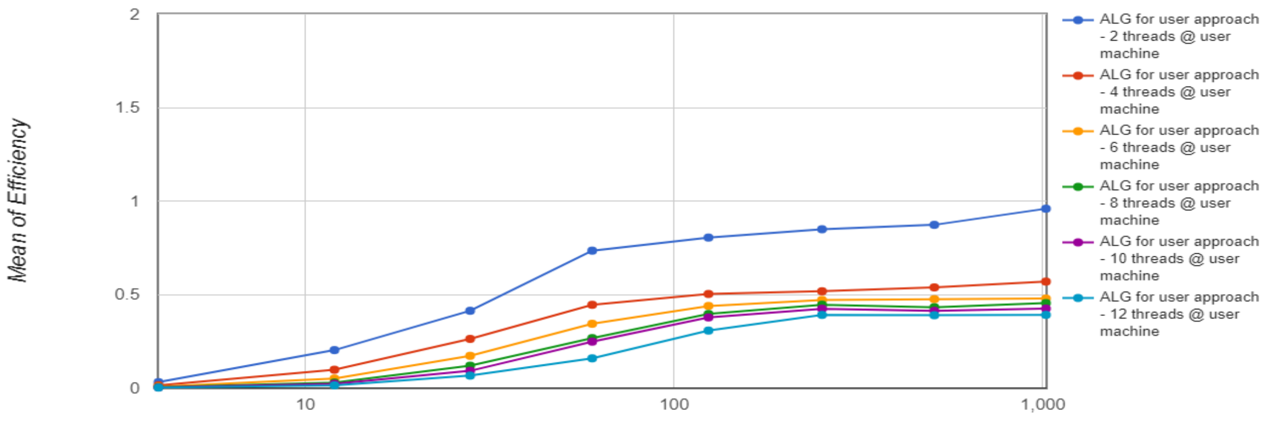
4.3 Speedup Curve related analysis (as problem size and no. of processors increase)

We are getting speedup of less than 5 even with 12 cores, so we are facing large overhead, which is described below in analysis section.



4.4 Efficiency Curve related analysis

We are getting highest efficiency for 2 cores, because for 2 cores black part of image will be divided equally, so equal load balance between cores and there won't be synchronization overhead. (This is explained below in detail). As number of cores increase, load imbalance would happen, so efficiency would decrease.



4.5 Karp Flatt Metric

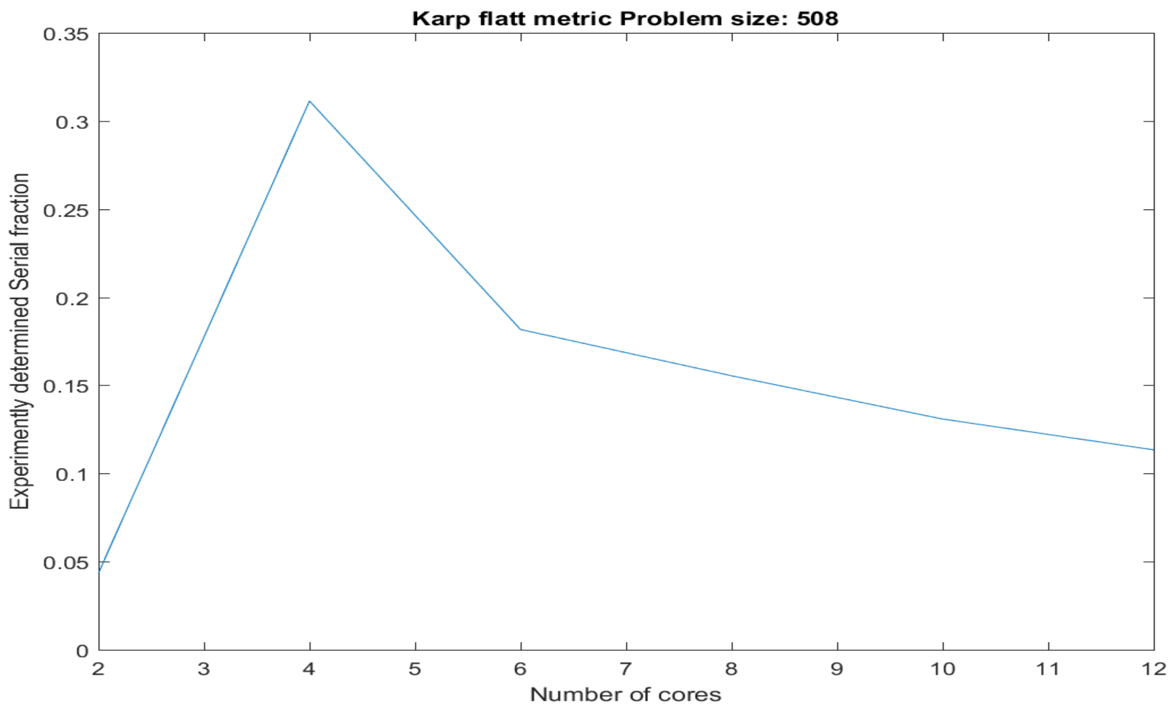


Figure 9: Problem Size: 508

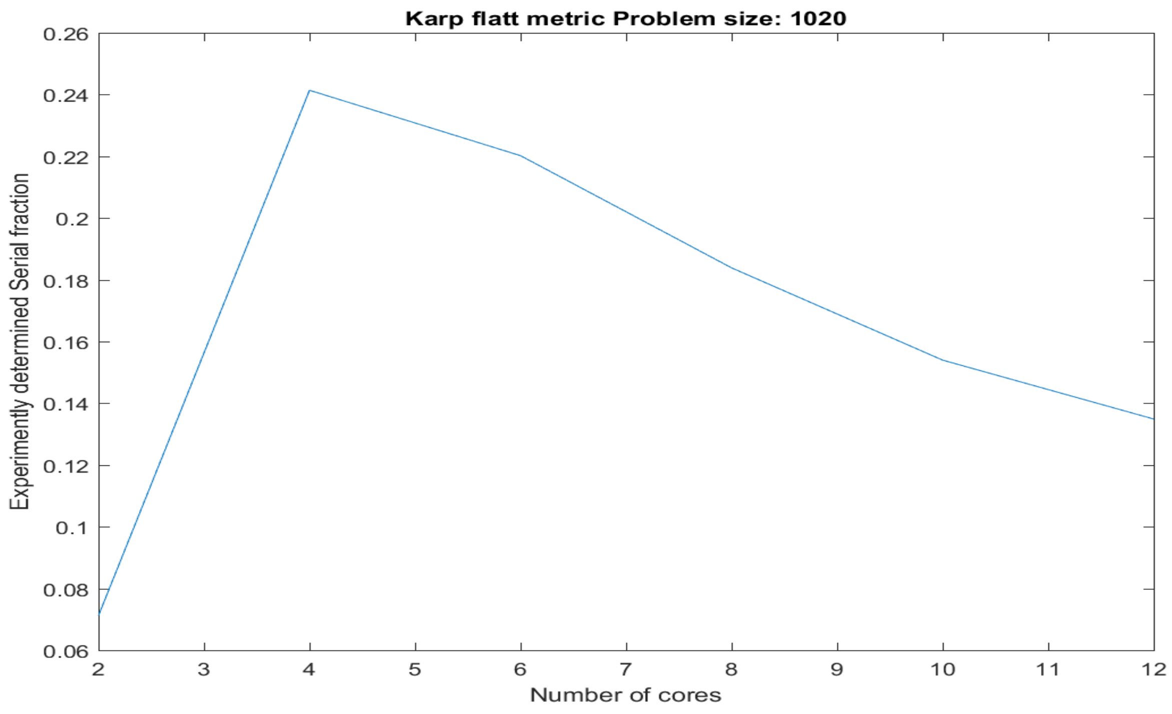


Figure 10: Problem Size: 1020

5 Further Detailed Analysis

5.1 Major serial and parallel overheads

Major serial overhead will be writing into shared memory. For each point we are writing 4 different values into shared memory. So for 1001*1001 image, communication would be 1001*1001*4. So this will be serial overhead.

Major parallel overhead will be due to **Synchronization**, which is explained in below subsection 'Synchronization related analysis'.

5.2 Cache coherence related analysis

There won't be much use of cache because as we said each core is just writing to memory, not much to read from shared memory.

5.3 False sharing related analysis

We are dividing rows into different cores, but number of rows need not be multiple of number of cores, so division of rows into multiple cores won't be equal. Some core has to compute more rows.

5.4 Load balance analysis

Number of rows divided into different cores would be almost same except last core but load can be different because at each point number of iterations can vary at large level. Because each core is getting different points to work upon, load won't be same across all cores.

5.5 Synchronisation related analysis

Synchronization is the major factor for decreasing speedup. Because we are dividing number of rows to calculate almost equal across all cores but each point will take different time so each core will finish computation at different time. For example, above generated image doesn't have black part equally for all rows (Black part denotes sequence for that point remains bounded, so that point will go until max iterations. So complexity would be higher for black part). And as we go towards central rows, black part is increasing, so cores which are calculating this part will take much more time to finish than other cores. So other cores have to wait until these cores finish computation **resulting in lower speedup**.

5.6 Granularity related analysis

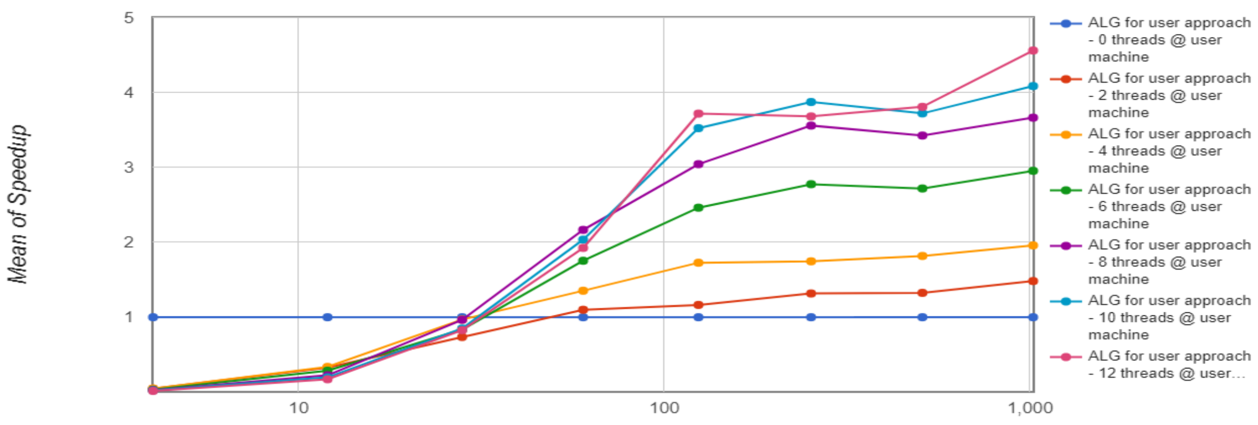
We are doing much like **Coarse grained parallelism** because there will be large amount of computational work as compared to communication between threads.

5.7 Scalability related analysis

This won't be much scalable if we keep on increasing number of rows because load balance between cores is not equal. Speedup can increase with number of rows only in case it divides black part of set equally.

6 Another approach for parallelization

Just like we have divided rows into multiple cores, we can divide columns into multiple cores. Here also we would face same issue Synchronization because of load imbalance. So this approach will give us same results as can be seen in below graph.



One difference can be seen is that for 2 cores we aren't getting speedup of 2 because column-wise black part of graph is not symmetric. So computations won't be equally divided between cores.

7 Further Scope

7.1 Approach by dividing iterations across all cores

Problem is computationally expensive because of 2 parameters, size of image(number of pixels) and size of sequence for each point. We divided first one across different cores in previous approaches. What if we divide second one across different cores?

For that we can fix range of iterations for each core i.e. one particular core has to do iterations in that specified range only for all points. For example, there are 120 iterations and 4 cores then 1st core will have 1 to 30, 2nd will have 31 to 60 and so on.

7.1.1 Dependency issue

First problem will be dependency. 2nd core won't start execution until 1st core has calculated last term of sequence in its range. Just like this will happen for other cores. To **resolve** this issue we can use **Pipe-lining approach** i.e. when 2nd core is doing execution of first point, 1st core will start execution for second point.

7.1.2 Synchronization issue

Second problem is that here still won't be Synchronization. Because each point doesn't have to go through all cores. It will stop as soon as its value gets diverged. So last core will have least load and load increases as we go to 1st core. So each core will have to wait until 1st core finishes all points. **This can decrease speedup to considerable extent.**

7.2 Solution to Synchronization issue of all 3 previous approaches

Task Scheduling can be used to solve Synchronization issue. A scheduler is what carries out the scheduling activity. Schedulers are often implemented so they keep all computer resources busy (load balancing), so maximum Speedup can be achieved.

In row-wise parallelism, we can first divide upper half of image across different cores and then divide further computation as cores calculating upper half gets finished. In this approach, none of the core will remain idle however some of them can remain idle at the end of execution but that won't matter much. Just like this, other 2 approaches can be implemented using Task scheduling.