Assignment 2 - CS671

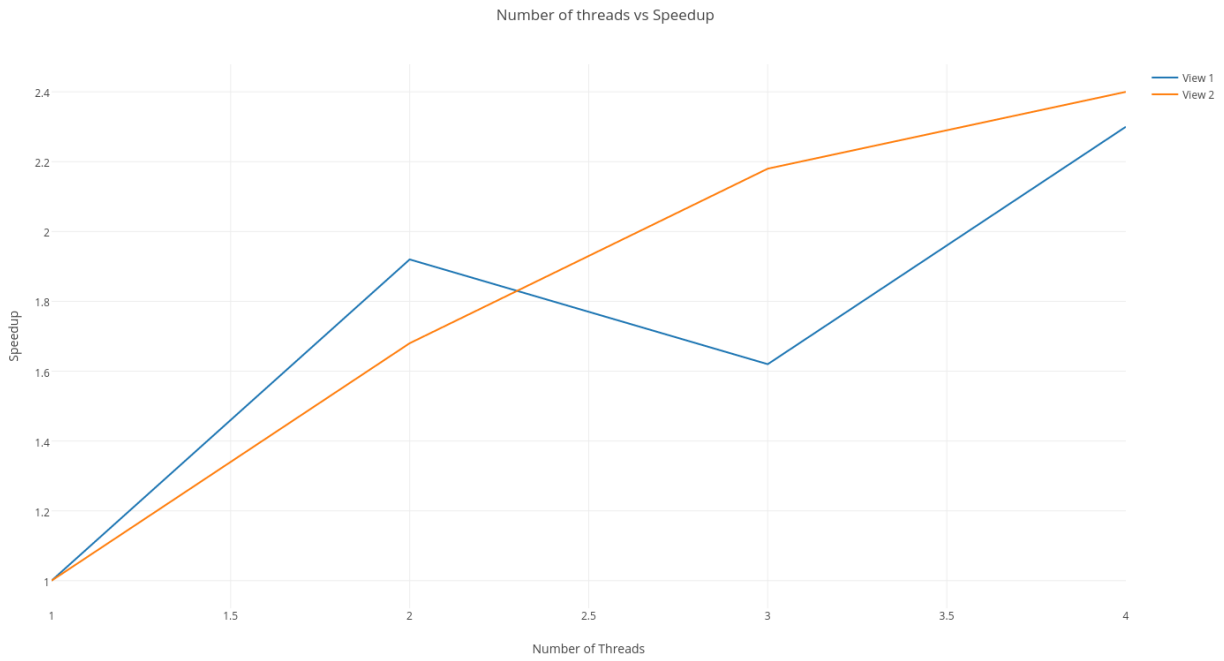
Mandelbrot set and pthreads

- Ashish Jindal (AJ523)

Part 1 -

1. Code - mandelbrot-solution-pthreads-part 1-2-3.tar.gz
2. Code - mandelbrot-solution-pthreads-part 1-2-3.tar.gz

Number of Threads	Speedup (View 1)	Speedup (View 2)
1	1	1
2	1.92	1.68
3	1.62	2.18
4	2.3	2.4



We don’t get a uniform linear speedup for increasing number of cores. We can see a dip of speedup at 3 cores, probably because in case of 3 threads, the work distribution between all threads is not same and same thread might be just waiting while other

threads are doing work. Other reason for non uniform linear speed up can be that the work distribution between various threads is not uniform. Although all the threads get equal number of rows to process (in most cases) but computationally it's not uniform as different threads run different number of iterations to computer cell value.

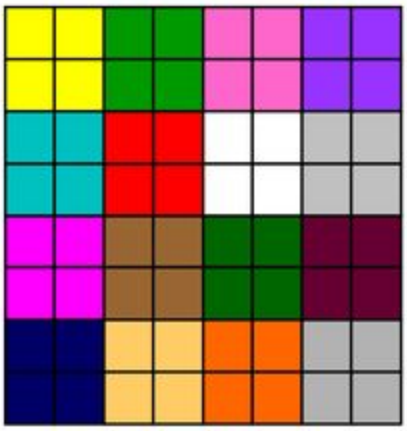
3.

Number of Threads	Thread Timing (View 1) ms	Thread Timing (View 2) ms
1	224	133.14
2	114.679, 115.540	79.611, 56.623
3	52.715, 144.85, 48.514	60.947, 46.205, 43.440
4	27.001, 96.334, 97.843, 26.467	55.676, 34.010, 35.225, 34.344

As we can clearly see in the above stats, different threads have drastically different timings which is a clear symbol of unbalanced loads on different threads. This means some threads finish up very quickly and keep waiting in ideal state when others are working, thus we are not using the threads up to their full capacity leading to lesser speedups than expected.

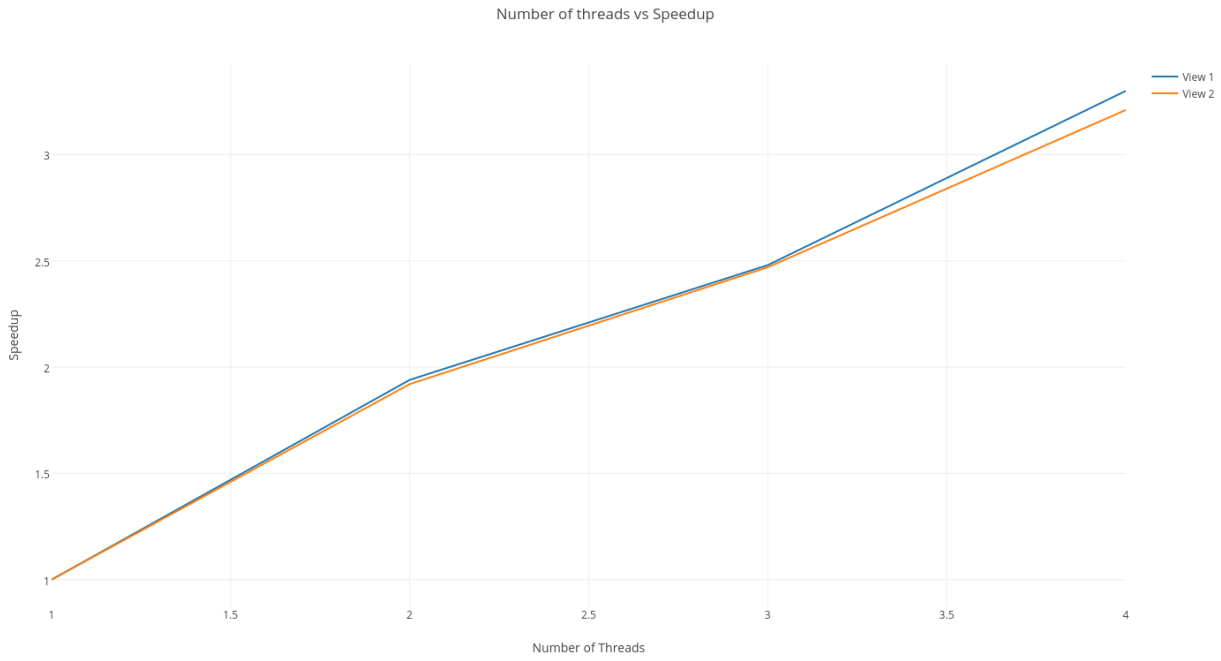
4.

Algorithm - We divide the array into blocks of size 5x5 and then each thread processes these blocks one by one. Since we can offset the index of block to be processed by a thread based on its thread ID, so we don't need to use any synchronization between threads to achieve speedup. (Code - mandelbrot-solution-pthreads-part 4.tar.gz)



Number of Threads	Speedup (View 1)	Speedup (View 2)
1	1	1
2	1.94	1.92
3	2.48	2.47
4	3.3	3.21

Number of Threads	Thread Timing (View 1) ms	Thread Timing (View 2) ms
1	224.992	133.14
2	115.418, 115.661	69.669, 70.204
3	84.159, 81.243, 88.799	46.805, 54.323, 53.468
4	65.350, 74.446, 66.986, 81.540	40.907, 40.711, 40.802, 43.147



Part 2 -

1. Code - mandelbrot-solution-openmp.tar.gz

Number of Iteration = 1200x800

Number of Threads	Speedup View 1 (Threads)	Speedup View 1 (OpenMP)
1	1	1
2	1.94	1.94
3	2.45	2.50
4	3.25	3.25

Number of Iteration = 1600x1000

Number of Threads	Speedup View 1 (Threads)	Speedup View 1 (OpenMP)
1	1	1
2	1.94	1.89
3	2.52	2.51
4	3.14	3.06

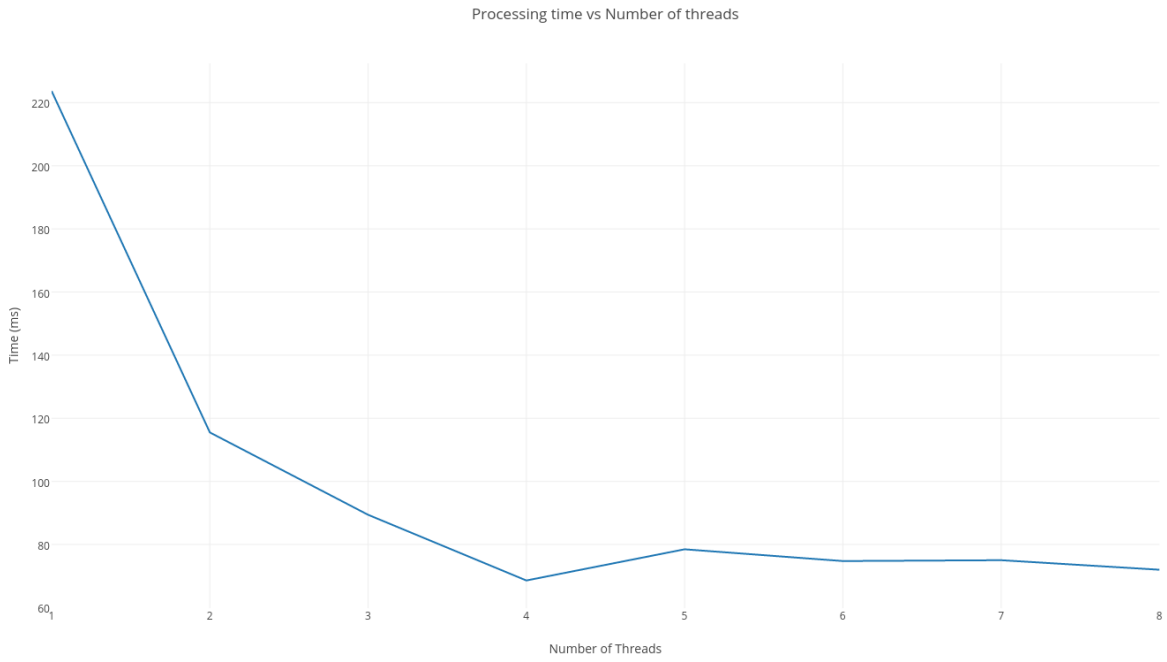
Number of Iteration = 2400x1600

Number of Threads	Speedup View 1 (Threads)	Speedup View 1 (OpenMP)
1	1	1
2	1.94	1.94
3	2.55	2.53
4	3.37	3.33

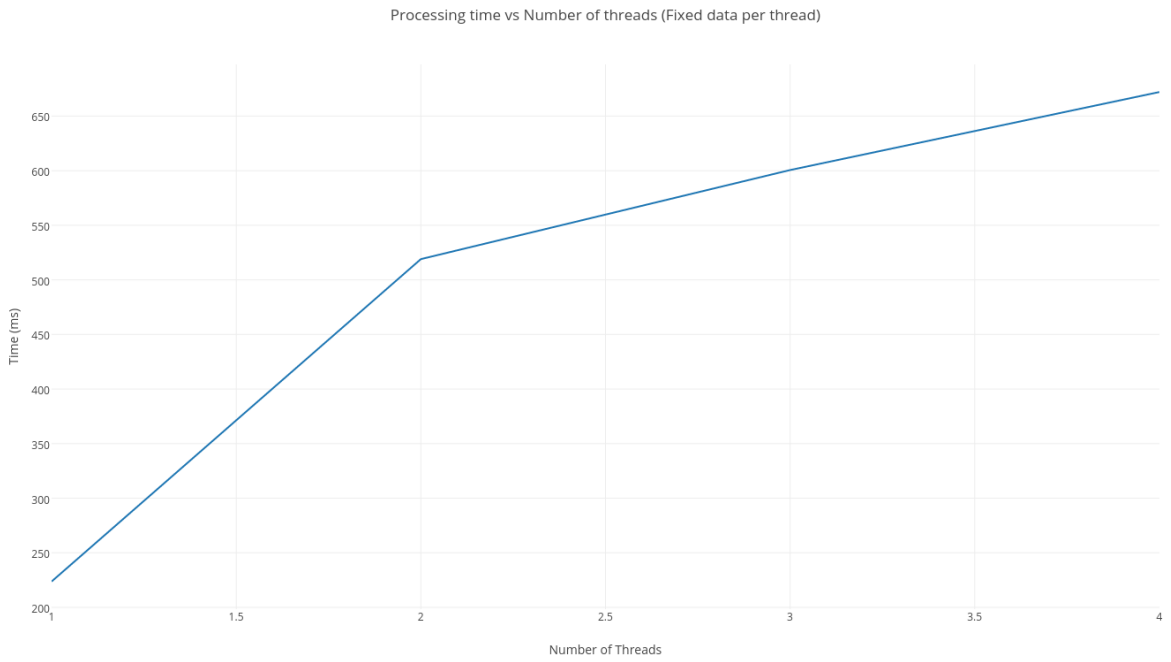
Till the number of threads = 4, both OpenMP and pthreads have approximately same performance. I couldn't test on more cores as there were no free nodes on elf.

2. We could use Dynamic schedules in our code. Dynamic schedule suits our case because each iteration takes different amount of time and thus dynamic schedule can split the work among thread in more fair way.
Dynamic schedule also has an overhead since after each iteration the thread must stop to read new value of loop variable which can slow the processing.
To tackle this we can also use guided scheduling and change the chunk size to suit our problem for maximum performance or let it be dynamic to be adjusted at runtime.

3. Scalability Analysis -
Strong Scaling: Total workload remains fixed vs increasing number of processors



Weak Scaling: The problem size per processor stays fixed but the number of processors increases.



4.
Parallelizable problem - Downloading and parsing of web-page by a browser.

I would use task decomposition to parallelize this problem. Web page has mainly 3 components HTML, JS and CSS, which can be downloaded parallely. We can start rendering the HTML on the view even while JS is still not downloaded completely. Once we have the CSS, we can re-render the view or partially render it depending on how much HTML/CSS has been fetched by the thread. In parallel browser keeps fetching the JS.

No, this problem is not load balanced. HTML/CSS/JS files are all of different sizes so threads will take different amount of time to fetch them and also processing time of all 3 of them for the corresponding threads are different.

The DOM tree of the page will be a critical section. Threads rendering HTML and CSS will be adding/removing nodes to the tree and parallely Javascript events will also be manipulating the DOM tree according to user code.

Room for improvement - Rendering thread can be run in parallel to fetching threads and can keep rendering UI based on partial input from fetching threads. Also multiple threads can be used to fetch multiple files and then assembled in order by a master thread when all are fetched. Instead of using just one thread for parsing a document (HTML/CSS) maybe we can use multiple threads which do that parallely.