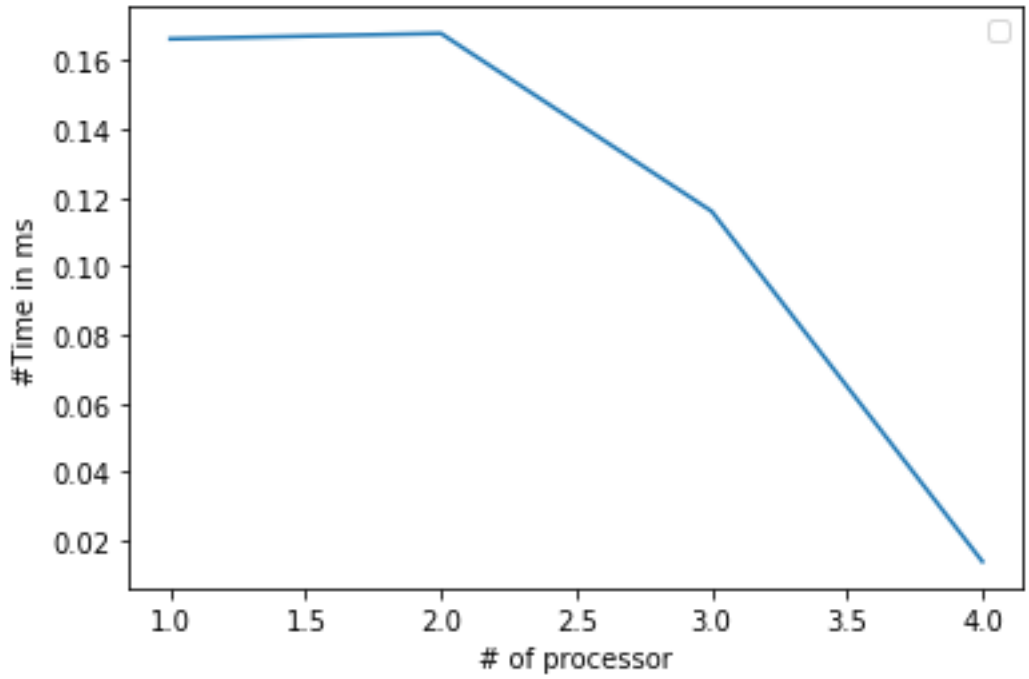
200626021

http://www.shodor.org/media/content//petascale/materials/UPModules/sieveOfEratosthenes/module_document_pdf.pdf

I used Sieve of Eratosthenes to create the Prime Number. I used an algorithm where the ith processor calculate the multiple of (ith)*(total_processor). This algorithm produce result really fast, O(n ln ln n) complexity. I have reported the number of twin prime generated under 10e4, although my code can compute upto 1e9 (I CHECKED up to this value).

#Processor = 1	#Processor = 2	#Processor = 3	#Processor = 4
0.1662	0.2097, 0.126 Avg = 0.1678	0.1242, 0.1222, 0.1038 Avg = 0.116	0.0881, 0.0988, 0.1289, 0.1424 Avg = 0.014



The problem seems to scale up, but after experimenting with code and following the reference I found this implementation that I choose is one of the two MPI implementation of Sieve of Eratosthenes, but this algorithm have load balancing problem[reference]. However, this problem is very fast as compared to the naïve algorithm used to produce prime number. I have reported all the twin prime generated in file “file1.txt”. The code for this problem is present in the file “main2.cpp”.

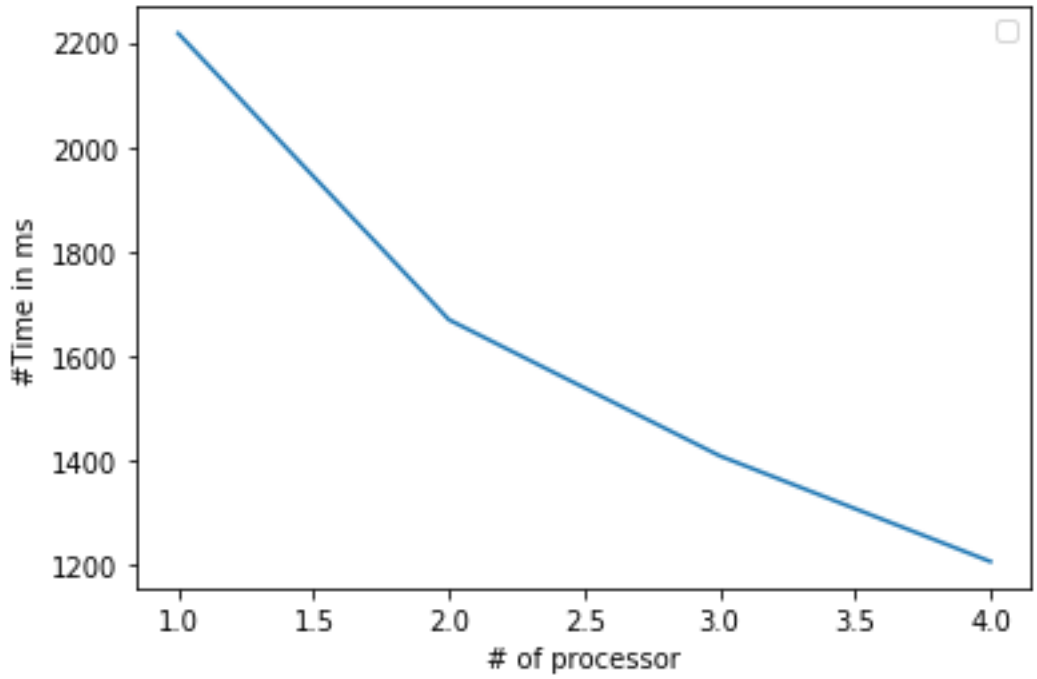
To tackle the load balancing problem I wrote “main6.cpp” which use the second MPI approach for sieve of Eratosthenes. I used domain Decomposition approach, which works on the condition that:

Range/#of Process > sqrt(N),

200626021

In sieve of Eratosthenes, we find all the prime number between(0,sqrt(range)), and remove all the prime which is multiple of these prime. If above condition satisfies, all the prime calculation will be done by 1st processor and the number get broadcasted to rest of them. Each processor have a chunk of original array and remove all the multiple of prime in their respective chunk. This algorithm works fine, but increase the communication overhead as we have to broadcast the prime number from root to all other processor. I implanted the algorithm, (building up on the work of : https://github.com/rbsteinm/MPI-Sieve-Of-Eratosthenes). This algorithm scale well and is fast as my first implementation, but there is a bug in code, which mess up the prime number sequence in root process, the number generated on other than root process are just fine. So, sometime this algorithm print wrong twin prime. I am providing a file "file2.txt" which has all twin prime pair generated under 1e3 using this implementation. I am also providing the result of how this implementation scale with #of processor for 1e8:

#Processor = 1	#Processor = 2	#Processor = 3	#Processor = 4
2217	1718, 1622 Avg = 1670	1390,1392,1450 Avg = 1410	1227,1188,1222,1196 Avg = 1208



200626021

Compilation:

To compile:

I used :

g++ main2.cpp && mpirun -np <#of Processor> .a/.out <flag>

g++ main6.cpp && mpirun -np <#of Processor> .a/.out <flag>

if flag is an int(any int) it will print out the output for example:

g++ main2.cpp && mpirun -np <#of Processor> .a/.out 1