Sieve Vector Implementation

Huda Ibeid and Enas Yunis
Supervised by Dr. Ralf Mundani

CS 292 – Project Report
Team – 01

History

Sieve of Eratosthenes is a method for identifying prime numbers. Conventionally it is implemented using regular array data structures. What has been noticed with that form of implementation is that the runtime is always $\theta(N\sqrt{N})$.

```
! Sieve usual implementation
        initialize boolean isPrime[2…N] = true
        for i 2… sqrt(N)
                for j 2 … N
                        isPrime[j]=(j!=i && j%i== 0)?false:isPrime[j]
        prime_count = 0
        for i 2 … N
                prime_count += isPrime[j]?1:0
```

The problem with this method is that it is very inefficient when it comes to usage of memory and time. When parallelized it has a time complexity of $\theta(\frac{N}{p}\sqrt{N})$ where p is the number of processors.

It is also highly dependent on having all the data available in order thus not allowing random reads or dynamic load balancing.

Use of Vectors

We have implemented a new paradigm (per instructions of our Supervisor) to reduce the N portion of the complexity with each successive cycle thus reducing the actual total running time and to also assume that the data is unordered and the solution has to account and allow for such.

```
! Serial Version of Vector Sieve (sieve_srl_v3.cpp)
      Initialize int vector primes <- empty
      Initialize int vector real_numbers <- set of numbers to consider

      while (true) do
        prime <- next smallest number in real_numbers
        if (prime > sqrt(N))
                break out of while
        primes <- prime
        for i in real_numbers
                if i%prime == 0
                        remove from real_numbers
      end
      prime_count = real_numbers.size() + primes.size();
      max_prime <- max_element in real_numbers and primes.
```

The advantage of using this format is that we can pick the next minimum element so we do not care about order, placement of data and allow for better data re-distribution.

The initialization of both original algorithm and vector-based algorithm is the same at $\theta(N)$ for serial and $\theta\left(\frac{N}{p}\right)$ for parallel. The difference now comes in with the inner for loop where *real_numbers* is constantly decreasing in size with each cycle. Thus in the first cycle we have $\theta(N)$ runtime; in cycle two we have $\theta(\frac{N}{2})$; in cycle three we have $\theta(\frac{N}{3})$ and so on. Thus our complexity got reduced to $O(N\sqrt{N})$ for serial and to $O(\frac{N}{p}\sqrt{N})$ for parallel. We can also give an estimate on the lower bounds of the system at $\Omega(prime\_count\sqrt{N})$ for serial and $\Omega(\frac{prime\_count}{p}\sqrt{N})$.

We could have improved the coding further given we are only required to report prime_count and not return the actual primes but we chose to keep the code this way for debugging and testing purposes. In the actual code we have also included tracking of initialization time and computation time the communication of that information is itself a bottleneck that we chose to keep for better result analysis.

Note that there is a semi-linear relationship between N and prime_count as depicted in figure 1

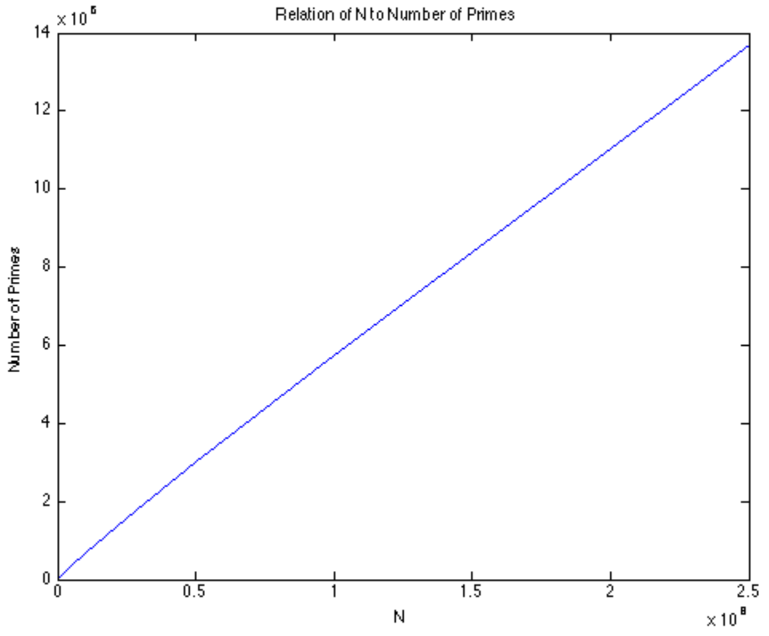

Figure 1

OpenMP Shared Memory Strategy

In OpenMP we have attempted to share the vectors but found that they do not thread safe and caused major issues when globally shared. So instead, we have chosen to start the threads and then each to create its data independently and evenly across the different threads. Then the threads share their local minimum and decide on the global_min thus the next prime to work with.

We have two different version of the OMP code depending on how we chose to setup the sharing and communication of the local minimums and global_min.

In omp_a (sieve_omp_v4.cpp) we have chosen to create 4 shared values and to use barriers and critical sections to control their modifications and controlled access. We end up with two barriers and two critical sections.

```
! OpenMP Version (a) of Vector Sieve (sieve_omp_v4.cpp)

      int local_min, prime, prime_count=0, max_prime=0


   # OMP PARALLEL
   {
      Initialize int vector primes <- empty
      Initialize int vector real_numbers <- set of numbers to consider for that specific thread

      while (true) do
        int prev_prime = prime
      # OMP Barrier
        int local_min  <- next smallest number in real_numbers
      # OMP Critical
        if (prime == prev_prime) OR (prime > local_min)
                prime = local_min
      # OMP Barrier

        if (prime > sqrt(N))
                break out of while
        if (local_mins(threadId) == prime)
                primes <- prime
        for i in real_numbers
                if i%prime == 0
                        remove from real_numbers
      end
      int local_max = max_element in real_numbers and primes or 0 if both are empty


      # OMP Critical
      {
        primes_count += real_numbers.size() + primes.size()
        if (local_max > max_prime)
                max_oprime = local_max
      }
  }
```

For omp_b (sieve_omp_v3.cpp) we have chosen to keep three shared vectors each sized at number of threads. Local_mins keeps track of the next prime to use. local_prime_count and local_max_prime contain the outputted information from each thread to sum over the counts and to choose the true max prime. Changing the code this way we were able to eliminate the use of two critical sections with a price of using three small sized vectors.

```
! OpenMP Version (b) of Vector Sieve (sieve_omp_v3.cpp)
      int vector local_mins(num_threads)
      int vector local_prime_count(num_threads)
      int vector local_max_prime(num_threads)

   # OMP PARALLEL
   {
      Initialize int vector primes <- empty
      Initialize int vector real_numbers <- set of numbers to consider for that specific thread

      while (true) do

      # OMP Barrier
        local_mins(threadId) <- next smallest number in real_numbers or N if empty
      # OMP Barrier
        int prime = min_element(local_mins) // global min


        if (prime > sqrt(N))
                break out of while
        if (local_mins(threadId) == prime)
                primes <- prime
        for i in real_numbers
                if i%prime == 0
                        remove from real_numbers
      end
      local_prime_count(threadId) = real_numbers.size() + primes.size()
      local_max_prime <- max_element in real_numbers and primes or 0 if both are empty.
  }

      prime_count <- sum over all the values in local_prime_count
      max_prime <- max_element(local__max_prime)
```

MPI Distributed Memory Strategy

We have chosen to keep the setup simple. We followed the coding strategy from what we have learned from OpenMP section by keeping each task running as independently as possible and to only share the minimum needed information. So we choose to MPI_ALL_REDUCE to share the local_mins and create the global_min (i.e. next prime to work with) and at the end to MPI_REDUCE to rank 0 (i.e. root) to have the final answer on taking the max of local_max_prime to get max_prime and another on summing local_prime_count to get prime_count.

```
! MPI of Vector Sieve (sieve_mpi_v3.cpp)
      Initialize int vector primes <- empty
      Initialize int vector real_numbers <- set of numbers to consider for that specific rank

      while (true) do
        prime <- MPI_ALL_REDUCE (MIN, next smallest number in real_numbers, N if empty)
        if (prime > sqrt(N))
                break out of while
        primes <- prime
        for i in real_numbers
                if i%prime == 0
                        remove from real_numbers
      end
      prime_count <- MPI_REDUCE (root, SUM, real_numbers.size() + primes.size());
      max_prime  <- MPI_REDUCE(root, MAX, max_element in real_numbers and primes, 0 if both empty)
```

Figures 2,3 and 4 show the communication between different cores.

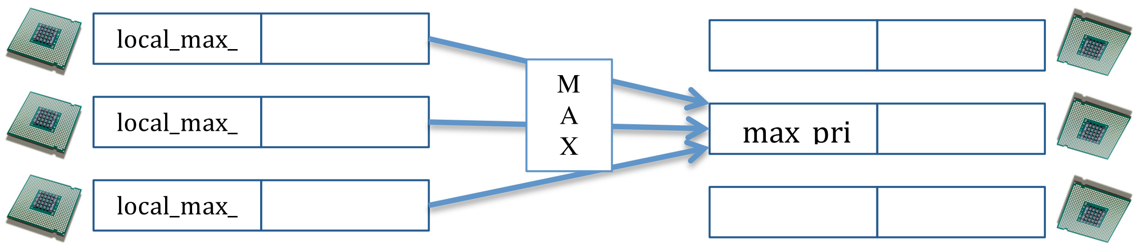

Figure 2

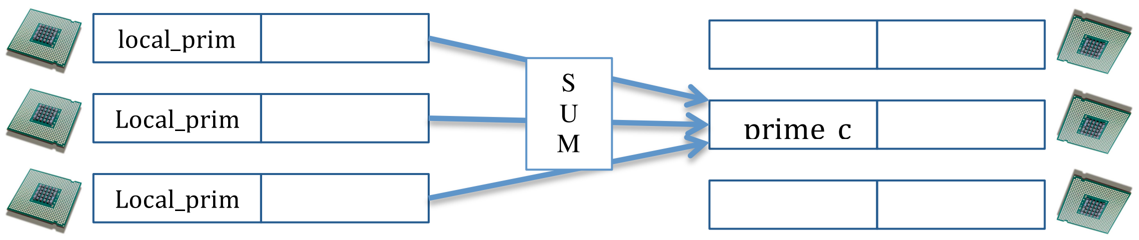

Figure 3

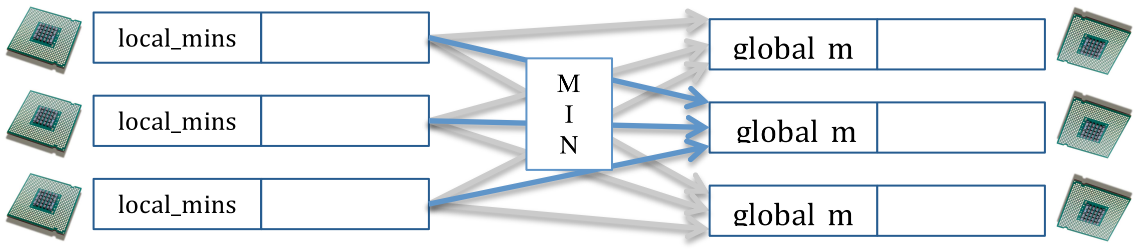

Figure 4

If you look at the actual code of the MPI implementation you will see that the system is fully parallel with the exception of Rank 0 printing out the final results. We have asked Dr. Igor Chikalov about computing serial/parallel sections of the code to find that the only serial is the output step at the end. As such we have chosen to treat our code with zero serial (given it is only for final output and we could have chosen for all of them to print the same info using ReduceAll rather than Reduce to rank zero). Using this information we know that by Amdahl law our $\boldsymbol{SpeedUp} = \frac{\mathbf{1}}{(\boldsymbol{serial}+\frac{\boldsymbol{parallel}}{\boldsymbol{NP}})}$ and thus our speedup is always NP and our Efficiency is always 100%. In the next section we will see the true speedup and efficiency and the reasons for the discrepancy.

Results and Evaluation for Single Linux Machine

Let N be the number entered as upper limit to retrieved the total primes and max prime and NP be the number of processors for OMP and MPI.

As seen in figure 5, comparing the serial execution with MPI, OMP_a, and OMP_b for NP=2 for all the different systems (Normal Plot) – We can notice a large gain in speed for N=25M and a huge gain for N=50M. We are attributing this to the way the program requests memory from the operating system and how well it is being managed. In the serial case we are not asking for the next pieces of memory fast enough for the Operating System to service. While in the NP=2 (and it is even better for higher NP) we have two processors asking for memory and both are being serviced comparatively at the same rate as the single processor but we are issuing the requests at a double speed.

We also want to be clear that with each run our system is reducing real_numbers size and thus (internal implementation of vectors) causes a re-allocation of the real_numbers internal array. For a single processor we are talking about the whole

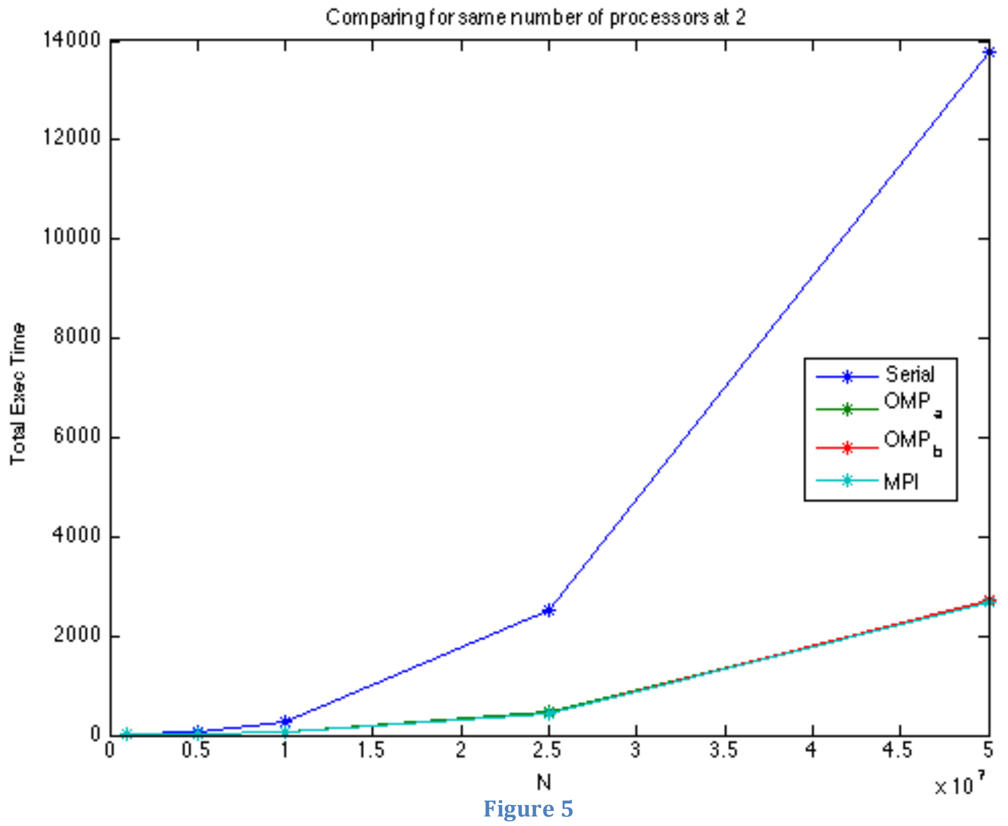

Figure 5

set of numbers being modified (new allocate, copy mem, delete old) while for the 2-processor systems each processor is only dealing with half the size of real_numbers and thus its new_allocate, copy_mem and delete_old are happening for smaller memory blocks.

We can also see from the above figure that the Total Execution Time of our three parallel algorithms is almost the same.

We show below (in figures 6,7,8) the strong scaling for OMP_a, OMP_b, and MPI for 2,3,4,5,6,7,8 processors on a Linux Fedora Machine with 8 cores using loglog plot.

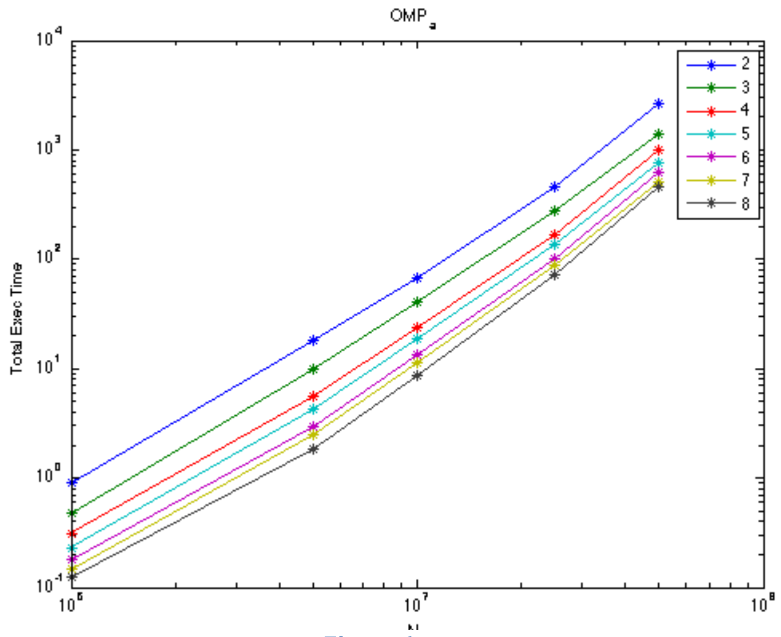

Figure 6

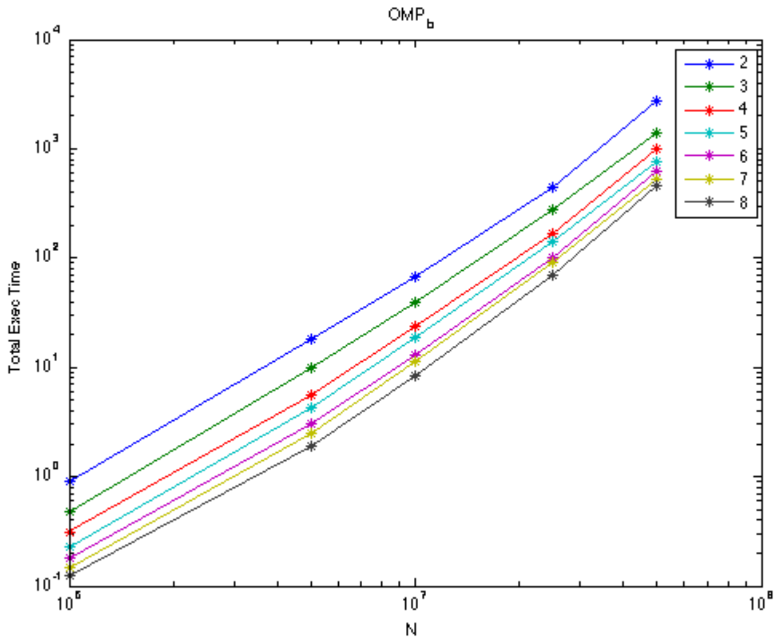

Figure 7

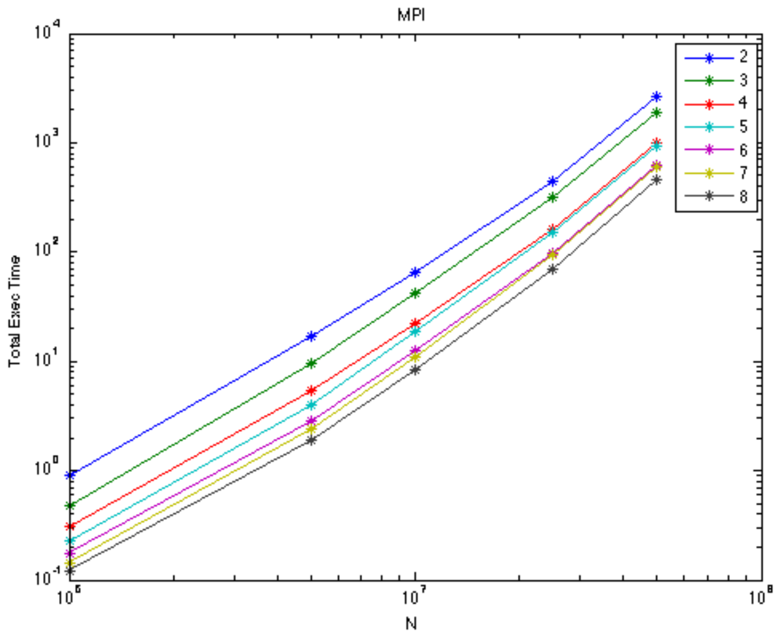

Figure 8

From any of the three figures above we can see that the gain in speed is drastic from NP=2 to NP=3. It is less drastic when going to NP=4 for larger and larger N. This is attributed to the fact that at larger and larger N communication is not the bottleneck as much as memory management by the OS and hardware is.

We also notice a trend that NP=4 is very close to NP=5 and NP=6 is very close to NP=7. From this we can make an assumption that speed gain is seen better on even numbered processors and not odd ones. This can be attributed to the architecture of the hardware and how the processors are setup to work in groups of two. So, when going to an odd numbers of processors we do not see a high gain because of the cost of communicating with the last processor. While when working with even numbered processors the last two have maximized the communication between them.

In Figure 9 we show how closely running the different parallel algorithms and in Figure 10 we show a close up of the actual behavior using NP=8.

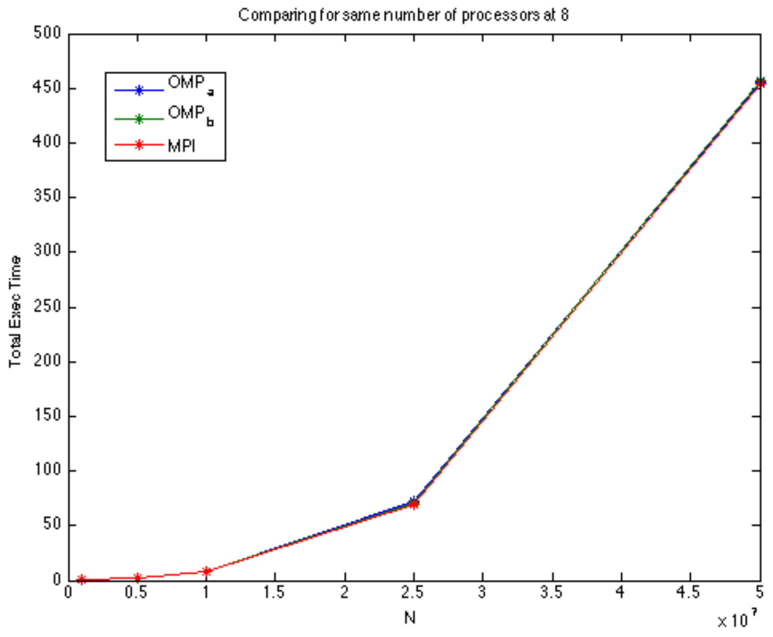

Figure 9

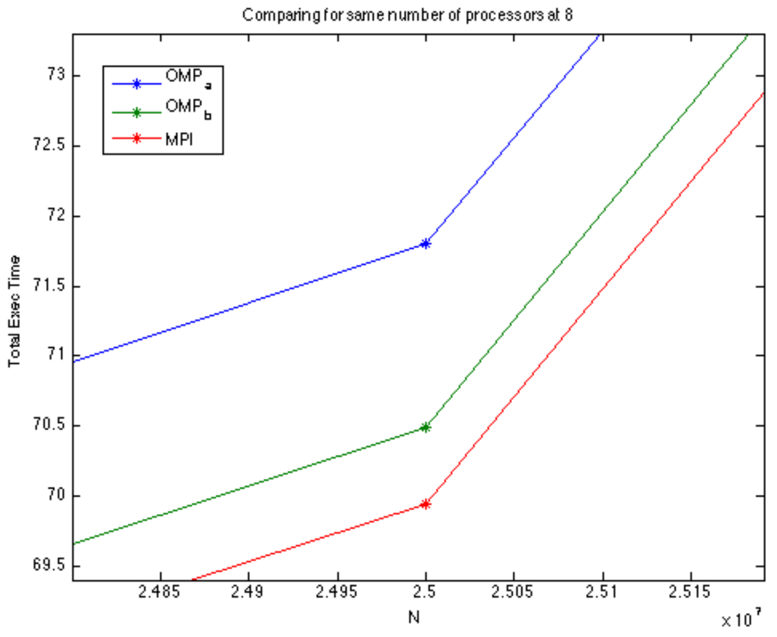

Figure 10

As can be seen OMP_a which uses more critical sections (ie. Serial portions) has a worst running time than OMP_b. This is due to the fact that OMP_a has a critical section inside of the for loop (i.e. a serial portion), while OMP_b removed such a serial portion by introducing the shared read vectors but individual write buckets in these vectors. MPI has by far the best results in time but again at relatively small improvements. OMP_b and MPI are much closer to each other than OMP_a. Our assumption is that we have modeled OMP_b to closely behave as MPI but not exact of course. We are wondering how does the OS handle each thread in OMP_b writing to its individual bucket in the Vectors and how is that dealt with when writing the data back to disk from a caching consistency prospective.

Results and Evaluation for Shaheen

Figures 11 and 12 show Speedup and Efficiency respectively using $T_2(N)$ with N in [10M, 25M, 50M] on [2,4,8,16,32] processors

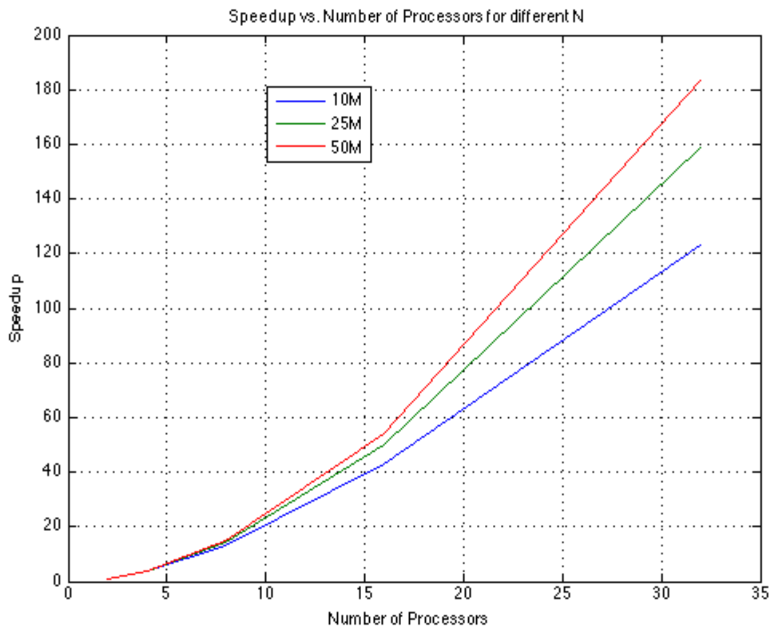

Figure 11

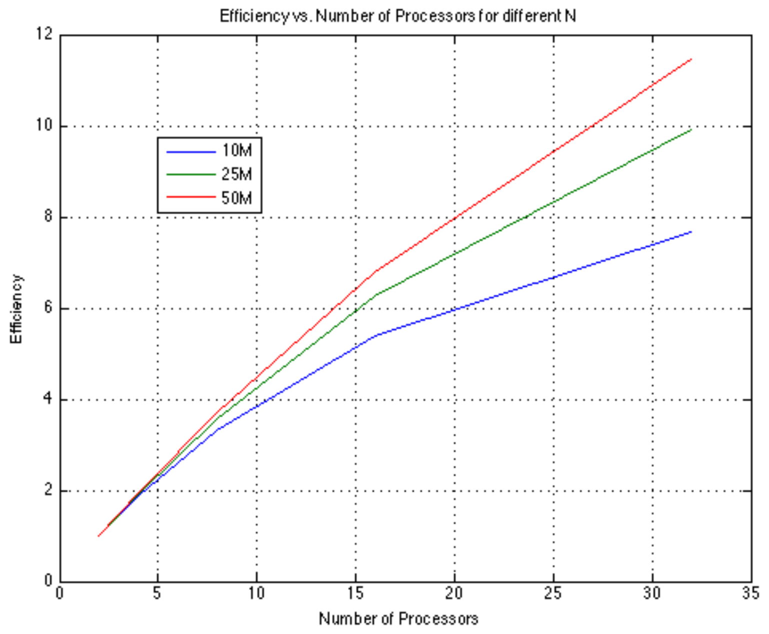

Figure 12

These graphs give the impression that our system is highly parallelizable because with every doubling of our processors we see both an increase in speedup and efficiency. We can see that our efficiency is above the unit point and that is attributed to the memory bounded-ness of this problem.

Between the different series we notice that for a higher N the speedup gain and efficiency gain is higher. This can be attributed to the fact that at high N values the system is memory management bound and the increase in number of processors (and thus distributed not-shared memory) hugely decreases the burden on the allocated memory in any given processor. The higher the N value the more overstressed the system is for small processes so the increase in number of processors has a higher effect on its speed up and efficiency.

We can also notice that the speedup at and below 8 processors is not that different across the different N values. While at 16 and 32 processors the difference in the series becomes more apparent and that is again attributed to the memory-bound issues of the system at high N all the series are stressed proportionally with their respective $T_2(N)$ starting point.

Once our system for higher NP relieved the memory-bound issues we see that the trend at higher NP's are not increasing at the same rate and that is also attributed to the fact that the communication has now re-appeared for lower N values. In short two things happen for lower N at different NP, the relief happens earlier and the communication thus starts to show its effects sooner than that for higher N values.

Figures 13 and 14 show Speedup and Efficiency respectively using $T_8(N)$ with N in [10M, 25M, 50M,100M, 250M] on [8,16,32] processors

We were unable to finish in under 24 hours for (N=100M, NP=2 ; N=250M, NP=2 and 4). So to deal with this we have displayed the Speedup and Efficiency from NP=8 for all the series to be able to plot the higher N values.

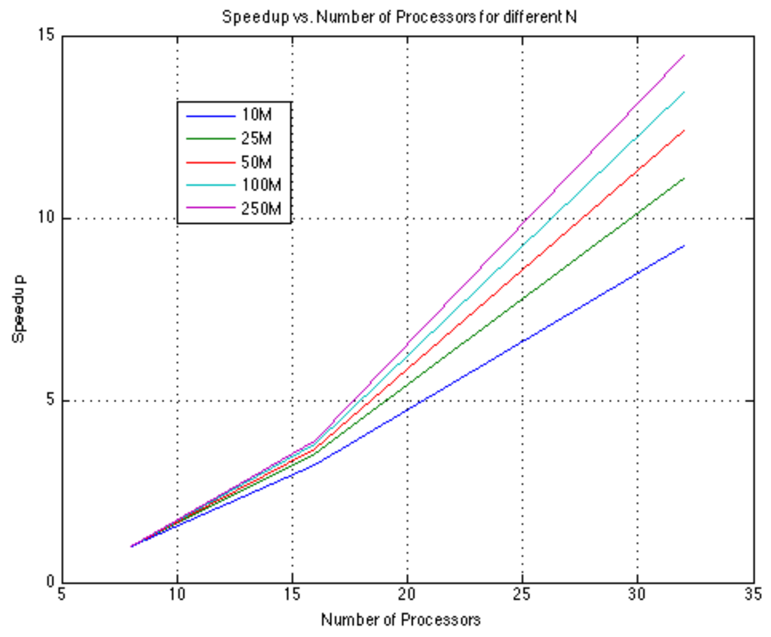

Figure 13

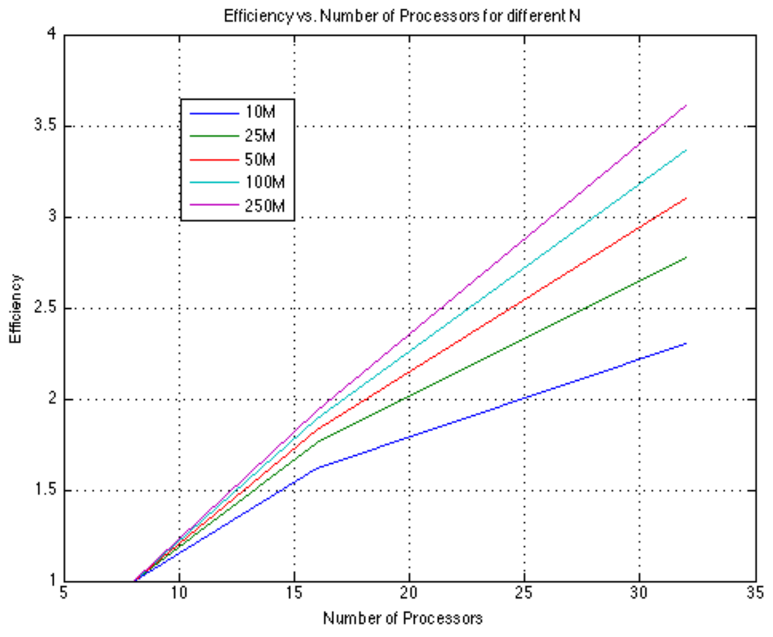

Figure 14

We can see that across the different series the higher the N value the more the speedup moves away from linear function to a quadratic one and the less square-root-like function the efficiency is until it becomes linear. Again this is attributed to the memory bounded-ness of the system and the behavior of memory management under high loads and not our specific algorithm.

System Improvements

We have improved the code by immediately removing all multiples of 2,3, 5 and 7. Thus reducing the size of our data by around ~3/4N to ~4/5N! This has allowed us to run larger sets within the time limit for smaller NPs.

Given our code is mostly memory bound then massive improvement has to be done in that area at a cost of more communication points.

We can have each individual thread only create a portion of the data at a time. Meaning, at round 1 all processes only together create real_numbers from 1 to N/Z where N/(Z*NP) keeps the system balanced between memory-bound and communication-bound.
In other words at round 1 each rank creates real_numbers of size N/(Z*NP). These ranks do their work as specified in the current MPI algorithm.

At round 2 the ranks agree together to create N/Z to 2N/Z but this time merge primes vector with real_numbers and start-eliminating non-primes using the same process in round 1. And so on with the next rounds. Note that with each round the primes list increases (can potentially decrease for non-sorted data) such causing more communication events because of repetitive use of a given prime across multiple rounds. We will have a total of Z rounds and thus a prime such as 2 will be used at most Z times for elimination.

When data is not sorted the size of primes will be worse for the data that came before their multiples.
We can see that with this method load balancing is also needed because some ranks will have more data in their primes at the end of a round causing both longer execution time for that rank and memory strain for larger and larger N's. Balancing the sizes of these private primes vectors before the start of the next round is very beneficial when we are running for yet higher and higher N.