Pthread Performance in an MPI Model for Prime Number Generation

Paul Johnson
University of Colorado
CSCI 4576 – High Performance Scientific Computing
paulie@colorado.edu

Abstract

The Message Passing Interface, MPI, relies on a Symmetric Multi-Processor (SMP) architecture to divide up tasks between nodes in order to efficiently complete a task. However, this model depends on distributed memory available only to an individual processor meaning all information must be sent across an interconnect any time two or more processors need to share data. A solution to this performance setback is the POSIX threads standard (Pthreads) which allows for concurrent tasks to be executed on a single processor with the results gathered using shared memory. This paper investigates the benefits of utilizing a combination of MPI and Pthreads for discovering bounded prime numbers while running on two separate SMP architectures.

1. Introduction

The introduction of the dual-core processor into the desktop and laptop market has increased the popularity of symmetric multiprocessing (SMP) over the past few years. This architecture relies on set of processors each with access to common memory across a machine. Accessing this memory, however, is solely dependent on the programming library utilized to complete the task at hand. This paper looks at invoking the Message Passing Interface (MPI) protocol and POSIX threads (Pthread) library to analyze the performance speedup of computational and memory intensive programs running two prime number generation algorithms.

1.1. MPI

MPI is a specification protocol for message passing across a supercomputer. It was designed in such a way to scale across large parallel machines, while remaining portable to many architectures and preserving a high volume of performance [1]. A current implementation of this protocol is called MPICH which is freely available and will be analyzed in this paper.

This project uses the MPI-1.1 specification, implemented as MPICH1, rather than the current MPI-2 design, as it only supports a distributed memory scheme. This library allows for a stronger comparison between shared and distributed memory which should highlight the benefits between using just threads, just processors, or a combination of the two to handle a given task.

1.2. Pthreads

The POSIX thread library (Pthread) provides an interface to generate and interact with separate threads of execution within a program [2]. This standard is defined by the IEEE and is available across nearly all variants of the UNIX operating system.

The idea behind using threads is to fork an individual program into running tasks concurrently to more efficiently get work done than in a single execution. This is accomplished by invoking time slicing wherein each processor switches between threads to divide up the program's runtime. As a result, this context switching provides a fast solution to using only one process to control multiple functional abilities simultaneously.

Another advantage to multithreading is the ability to have each thread utilize shared, global memory while also employing private memory as necessary. This allows for communication between the threads to create a true divide and conquer environment. Consequently, these traits allow for each thread to simulate a virtual node in a uni-processor environment and should provide a decent comparison to a real multiprocessor machine.

1.3. Primality Algorithms

The development of a proficient algorithm to find large prime numbers has been an important topic of research for cryptographers over the past forty years. Its vitality has led to the creation of a $250,000 prize from the Electronic Frontier Foundation for the first person to discover a one

billion digit prime number [3]. Unfortunately, it would take over a month on a single processor Pentium 4 system just to discover if one such number is prime.

The real question, however, is why would the discovery of these numbers be so important? In a standard public key encryption method, such as RSA, two large, distinct, and random integers are necessary to encrypt and decrypt messages in a secure manner. Its security is based on the very hard and deterministic process of factoring large numbers. This project presents a comparison of two different algorithms to discover prime numbers using a parallel and multithreaded programming approach to show the advantages of both libraries.

2. Algorithms

Checking primality is a problem that is quite hard to do deterministically. The best runtime algorithm developed to date produced an $O(\log^{12} n)$ bound and was created by Agrawal, Kayal, & Saxena [4]. The goal for this project obviously will not be to break this barrier, but to provide insight as to the various choices to find prime numbers in varying sizes of processor and thread counts as it pertains to the MPICH1 and Pthread library.

2.1. Naïve

The first algorithm to be investigated utilizes a simple strategy to avoid even numbers, except the number two, and continually test the odds for primality. This test involves a basic modular exponentiation function that returns true when the correct integer is found. It does not draw any conclusions from the previously discovered primes and does not take into account the current index. As such, it should prove to not scale when the upper bound is increased. In a parallel environment, however, the load sharing should be able to compensate for the massive number of calculations needed, but it shouldn't provide an optimal solution.

The goal for this algorithm is to perform a computational stress test on the given libraries. The naïve implementation relies heavily on looking at as many numbers as possible all the while using very little memory. This should favor tests that involve as many physical processors as possible.

2.2. Sieve

The sieving algorithm, theorized by the Greek mathematician Eratosthenes, is a primality technique that builds a check table based off of the sifting of integers [5]. A maximum size of the highest integer to be analyzed is declared and then the program starts from one. At each step of the process if a non-prime is found, each multiple of that number is automatically excluded from future searches. This eliminates the need of redundant checks on numbers that cannot possibly be prime. This methodology can also be applied to the factoring of large prime numbers which, as already stated, is a necessity to RSA encryption.

The table this algorithm maintains can get rather large as the maximum number bound the program should search up to increases. Accordingly, this allocates significantly more memory as opposed to the naïve implementation. Now while there are tricks to reduce the size of this table, it is obvious that this algorithm favors a strategy with the best memory utilization plan. When combined with the Pthread library's ability to invoke shared memory, it should prove to be the most efficient.

3. Implementation

In the creation of this project, both of the algorithms are written in C for all MPICH1 and Pthread capable systems. The machine that this program was developed and tested on is Linux based, but it should be portable enough to run with consistent results on any POSIX compliant system. For user simplicity, command line arguments are available to control the number of primes to within a specified bound, the type of algorithm to employ, and the number of threads to use as the algorithm dictates. It is assumed that this program will be running on some form of SMP environment implying that it is up to the system's job scheduling system to specify the amount of processors to employ.

3.1. Naive

The naïve algorithm gets its name according to its very simple way of solving the primality problem by knowing that even numbers with the exception two are not prime. By design this requires a large number of computations to effectively stress test the capabilities of MPI and Pthreads.

In the following sections each of these libraries will be analyzed and broken down into its pseudo-code and insight onto the benefits of their strategies. It is worth noting that in each of the examples that the variable 'Rank' implies the given processor's number in the layout scheme and 'Size' indicates the amount of processors available at execution time.

3.1.1. Serial

The serialized version of this algorithm works as expected by accepting in an upper bound and computing every number in between. It avoids even numbers by incrementing by two at every iteration.

Serial Pseudo-Code

```
for each # in bound
    check if # is prime
    increment by 2 to skip even numbers
```

3.1.2. Parallel

The parallel version creates two new variables in order to spread the workload around to all of the allocated processors. The methodology for this technique is to increment the current value in a round-robin fashion between the current rank. For example in a 4 processor model, if rank0 handles the value 9, then rank1 takes 11, rank2 takes 13, rank3 takes 15, and finally rank0 circles to the next possible value which is 17.

Parallel Pseudo-Code

```
Start ← 2*Rank+1
Divide ← 2*Size
for each # in bound from Start
    check if # is prime
    increment by Divide
Reduce
```

The *Start* variable dictates the starting point for each individual processor and the *Divide* variable represents the interval in which to increment once done with the current number. After an MPI_Reduce call, the numbers can be passed via messages to report a total prime count.

3.1.3. Multithreaded

The multithreaded algorithm inherits most of the key functionality from the parallel implementation, except it now treats each thread of execution as a separate virtual processor. For both the *Start* and *Divide* variables each is multiplied by the current Thread ID and number of threads, respectively. After each thread is completed, the local prime count is added to the global count among the processor before a reduce finishes up the work among processors if there are more than one.

Multithreaded Parallel Pseudo-Code

```
Start ← 2*Rank*Thread_ID+1
Divide ← 2*Size*Number_of_Threads
for each # in bound from Start
    check if # is prime
    increment by Divide
Add local thread counts into global variable
Reduce
```

Highlighted in red in the previous pseudo-code is an operation that requires special attention. In this situation with multiple threads it could arise that two or more of these execution instances finish at the same time and attempt to manipulate the same block of memory. This may potentially lead to inerrant results on the reported total prime count. In more appropriate terms, the atomicity of the program must be preserved throughout the changing of this resource.

In solving this, a semaphore must be utilized to encapsulate the variable to create a waiting system for each thread if the data is in use. For a simple example like this one, the only necessary operation is to create a lock, change the resource, and then unlock to allow others to do their work. The multithreaded sieving example in a further section, however, better exemplifies a race condition that makes it a more advanced and tricky problem.

3.2. Sieve

The counterpart to the naïve algorithm is the sieving design. Rather than skipping every even number, it recognizes that multiples of numbers found to be prime cannot possibly be prime themselves. For example, by determining the number 3 to be prime, the numbers 9, 15, 21, etc. are not prime themselves so there is no reason to analyze them.

This small logical rule dramatically decreases the cost per bound, but requires that a table be setup to keep track of these marked non-prime multiples. In the implementation of this algorithm, this table is allocated dynamically based on the size of the inputted bound.

Like for the naïve portion of this paper, this section will explain the primary sieving algorithm and how it is combined with parallel and multithreaded techniques to more efficiently solve the primality problem. For each pseudo-code box, the variables *Rank* and *Size* represent the processor number and number of processors present at runtime, respectively.

3.2.1. Serial

The serialized version starts off from the number two, denoted as the variable *k* in the below pseudo-code, and is incremented by the last prime found inside the loop. An initial sieve table is built to the size of the specified bound and each number inside is set to zero, which is denoted as unmarked. As numbers are discovered to be prime through the loop, their value in the table is changed to one to denote a marked status. Once completed, a simple pass through of the table yields the total prime count.

Serial Pseudo-Code

```
Create SieveTable with a size equal to Bound
Set all entries in table equal to 0 (unmarked)
k ← 2
while k*k <= Bound
    Set multiples of k from k*k to Bound to 1 in
       SieveTable (marked)
    k ← smallest unmarked number less than k
for each entry in SieveTable
    unmarked numbers are prime
```

3.2.2. Parallel

The parallel algorithm introduces two new variables, similar to the naïve implementation, called *Start* and *Divide*. These define a balanced workload model among allocated processors so that the computer can easily partition out marked and unmarked numbers in the table. Rank0, or processor 0, acts as the chief node which determines the smallest unmarked number and then broadcasts it out to every other processor. As each processor completes its work, a reduction is done to combine results. It should be noted that each processor in this implementation keeps its own sieve table which adds to total memory cost significantly at each access.

Parallel Pseudo-Code

```
Start=(Rank*Bound/Size)+2
Divide=(Rank+1*Bound-1)/Size -
(Rank*Bound-1)/Size
Create SieveTable with a size equal to Bound
Set all entries in table equal to 0 (unmarked)
k ← 2
while k*k <= Bound
    Set multiples of k from k*k to Bound to 1 in
       SieveTable (marked) from Start
    If Rank0
        k ← smallest unmarked number < k
        Broadcast k to other processors
for each entry in SieveTable increment by Divide
    unmarked numbers are prime
Reduce
```

3.2.3. Multithreaded

Combining the parallel model with multithreaded properties yields a change in the starting position and divide incremental value as well as a shared sieve table controlled with the use of locks. As highlighted in the pseudo-code, there are three lines in red that specify a global memory change shared between threads on a single processor.

Multithreaded Parallel Pseudo-Code

```
Divide=((Rank+1*Bound-1)/Size -
(Rank*Bound-1)/Size)/Number_of_Threads
Start=(Rank*Bound/Size)+2+(ThreadID*Divide)
Create SieveTable with a size equal to Bound
Set all entries in table equal to 0 (unmarked)
k ← 2
while k*k <= Bound
    Set multiples of k from k*k to Bound to 1 in
       SieveTable (marked) from Start
    If Rank0
        k ← smallest unmarked number < k
        Broadcast k to other processors
for each entry in SieveTable increment by Divide
    unmarked numbers are prime
Add local thread counts into global variable
Reduce
```

The first line allocates the sieve table (created by thread 0) in shared memory. The second line uses the current value of k to mark off known non-primes off of the table. This requires the correct use of locks to prevent data damage and a possible race condition. This idea is also present in the third line where each thread shares its prime count locally before a reduce sends its final combined results for the processor total.

4. Testing Machines

Two clusters are utilized in this project for the testing of both algorithms. Both of these were chosen on the basis of runtime allowed, pure processing power, and memory speed.

The first machine is Occam, a 27 node computer, each with 2 PPC970 processors and 2.5GB RAM. It is located inside the Computational Science Center at the University of Colorado. Occam invokes an unlimited time policy which makes it desirable for long test runs in excess of 30 minutes. The operating system for this cluster is Debian Linux and the program is compiled with GCC 4.1.2.

The second machine is Lonestar, a 1460 node supercomputer that is equipped with 2 Intel Xeon (dual-core) processors per node, rated at 2.6 GHz with 2GB of memory each. The raw speed and scalability of this cluster should point to faster runtimes, which is desirable since this machine is accessed via Teragrid and is limited by project time. The operating system running is Centos Linux and the program is compiled with Intel 9.1.

5. Performance Testing

To reiterate from a previous section, the objective to this project is not to examine the specifics of the naïve and sieving algorithms, but rather to examine the benefits and pitfalls of MPI for parallelization and Pthreads for multithreaded applications. As such, it is appropriate to examine a head to head MPI versus Pthread performance analysis and then a closer look at combining both for maximum efficiency. Each run utilizes a bound of 100 million numbers.

5.1. Head-to-Head

For directly comparing both of the libraries, Occam was employed due to the possibility of having long runtimes for the lower processor and thread counts. This test case involves scaling MPI for both algorithms in a process count from one to eight for one instance of execution. To contrast this, the Pthread implementation runs on one processor for each job, but scales from one to eight threads per run. This creates a model that can be described as head-to-head.

The final results after sixteen runs are listed below in Table 1 and are reported in total seconds. It can be seen that the worst case scenario takes over eighteen minutes to complete, while the best runs at just over two seconds.

Table 1 Performance of Naïve and Sieving Algorithm on Occam with Increasing Processor/Thread Counts

Processors/Threads	1	2
MPI Naïve	1068.009991	548.5526791
Pthreads Naïve	1101.859197	553.7542739
MPI Sieve	25.31701922	16.33148003
Pthreads Sieve	36.82761192	4.320442915

Processors/Threads	4	8
MPI Naïve	274.2667301	136.646878
Pthreads Naïve	415.109201	346.3331552
MPI Sieve	7.812529087	4.002258062
Pthreads Sieve	2.907956123	2.100058079

5.2. One-to-One

In contrasting the head-to-head ratio of processors to threads, investigating the mixture of both should provide an optimal and realistic look at solving primality for large scale bounds. This combination should point to an ideal pivot point at which shared and distributed memory levitate without unnecessary computational cost.

Testing this node and virtual node relation in a one-to-one ratio was assessed on Lonestar in processor counts from one to eight. There are two types of test cases for this comparison. The first involves testing both implementations in a pure MPI environment where there are no threads present. The second requires that the number of threads is equal to the number of processors as each scale throughout the submitted job. For example, if six processors are allocated, then it is necessary to inform the program to invoke six threads at runtime. This clearly should lead to a faster calculated speedup, but at what rate in comparison to the Occam runs should provide for interesting insight. Since this paper is primarily for parallel computing, isolated Pthreads are not tested for this assessment.

The final results after twenty runs are listed below in Table 2 and are reported in total seconds. The worst case scenario takes just under four minutes to complete with the best running under a tenth of a second. To make better sense of the results, the next section focuses on the importance of the findings.

Table 2 Performance of Naïve and Sieving Algorithm on Tungsten with Combined Increasing Processor/Thread Counts

Processors/Threads	1	2	4
MPI Naïve	219.032444	109.595355	64.973172
MPI+Pthreads Naïve	219.219765	95.045612	56.085037
MPI Sieve	4.846176	2.658205	2.174487
MPI+Pthreads Sieve	4.852561	0.191231	0.105341

Processors/Threads	6	8
MPI Naïve	44.69765	27.496463
MPI+Pthreads Naïve	37.162193	28.351757
MPI Sieve	1.244045	0.950858
MPI+Pthreads Sieve	0.081862	0.07315

6. Analysis

The tables presented on the previous page indicate a significant speedup in the test cases of both libraries for their respective process and thread counts. To complete the most detailed examination of the findings, tables and figures are generated and are complemented by relevant observations for the remainder of this section.

6.1. Occam

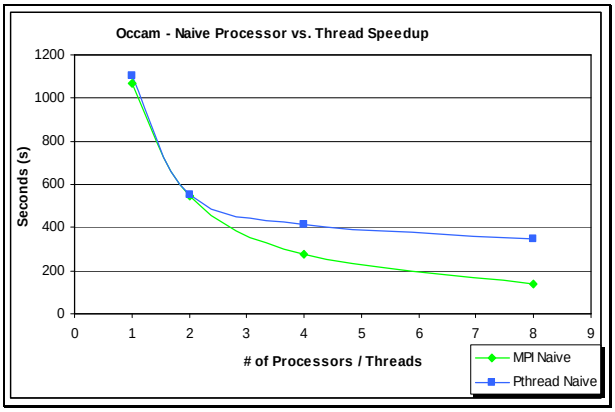
The measured speedup factors for the runs with the MPI versus Pthread schemes for both of the algorithms are shown below in Table 3 for Occam. The comparison is separated out by implementation. The control point at which speedup is observed is the MPI Naïve and MPI Sieve fields and they are set to one as indicated in blue.

Table 3 Speedup for Head-to-Head MPI versus Pthread on Occam for Naïve and Sieving Algorithms

MPI Naïve	**1**	1.94	3.89	**7.81**
Pthread	0.96	1.92	2.57	3.08
MPI Sieve	**1**	1.55	3.24	6.32
Pthread	0.68	5.85	8.70	**12.05**

This table exhibits a reasonable advantage of the pure parallel implementation as opposed to multithreaded code as the number of processors and threads increase up to eight. It is at this point where the rate of acceleration reaches over twice as fast. Utilizing a large number of calculations in this situation benefits the strategy employing the largest number of processors. While the threaded version running on one processor does perform better in certain circumstances, it definitely cannot scale to meet the computational demand necessary to compete with a cluster.

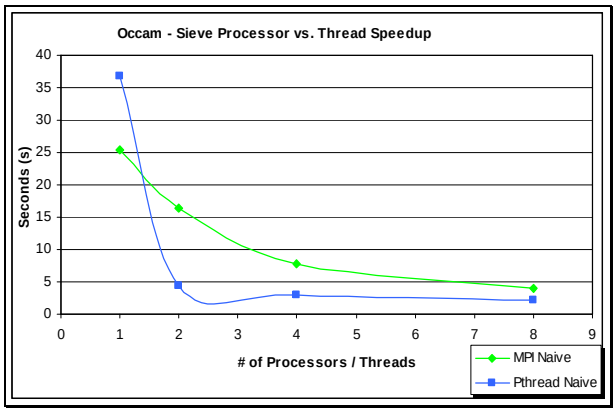
Figure 1 Performance Graph of Naïve Algorithm on Occam with Increasing Processor/Thread Counts



Generated in Figure 1 is a visual display of the time chart gathered in Table 1 for both naïve platforms. Each of the lines noticeably decreases exponentially with time. The Pthread library is able to reasonably keep up at two threads before flattening out as the MPI version employs more processors.

The sieve algorithm, however, offers startlingly results that are nearly opposite in its favoritism. The observed speedup indicates that after a poor single thread performance, possibly due to overhead, the multithreaded version outperforms MPI at an average rate of two to one. As highlighted in Figure 2, the jump from one thread/processor to two threads/processors displays a huge descending slide down to two before leveling off exponentially. This can definitely be attributed to benefits of the shared memory model. By using a global sieve table, unnecessary message passing of discovered primes in the parallel platform puts it at a significant disadvantage to multithreaded code.

Figure 2 Performance of Naïve and Sieving Algorithm on Occam with Increasing Processor/Thread Counts



Sieving through the bounded numbers exemplifies an amazing accomplishment of threads. In the final test run, one processor running eight threads outperformed eight processors in parallel by simply invoking a shared memory model. This points to the high cost of passing messages through the cluster and the superb product yield of efficient memory usage.

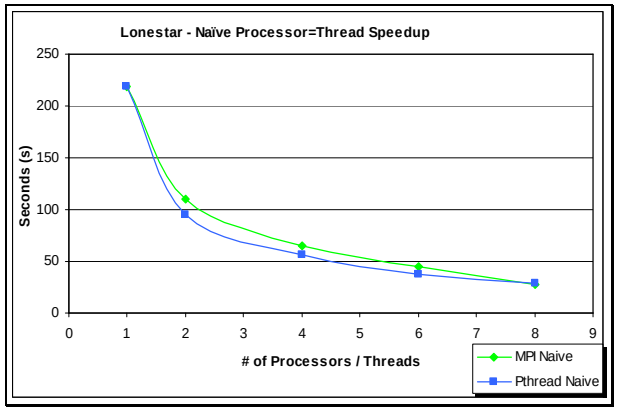
6.2. Lonestar

While Occam showed the consequences of using just one particular strategy over the other, the test runs on Lonestar were designed in such a way to exemplify the speedup

gains of merging both to achieve peak efficiency. As described in section 5.2, these were simulated in a one-to-one ratio where the amount of threads employed at runtime is equal to the number of processors allocated. Lonestar was chosen mainly for its utilization of dual-core processors which are presently becoming more and more prevalent.

Using the numbers in Table 2, a graph in Figure 3 was generated to display results pertaining to the naïve platform. The blue line represents the combining of MPI and Pthreads into the same running job as opposed to the green which is just parallel. For the most part it appears that both lines decrease exponentially at around the same rate. The computational heavy naïve algorithm does very little to benefit from extra running threads and this is relatively consistent with the results gathered from Occam.

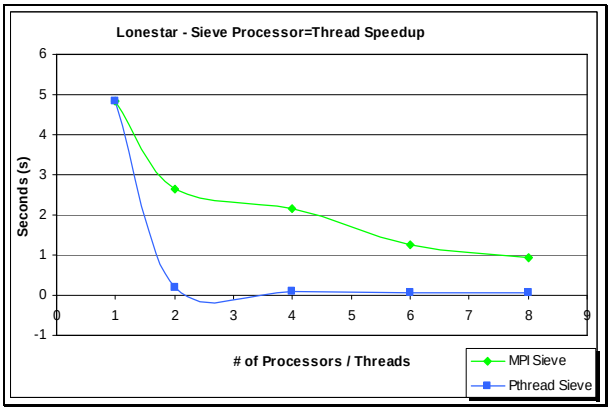
Figure 3 Performance of Naïve and Sieving Algorithm on Tungsten with Combined Increasing Processor/Thread Counts



It does not appear that inserting Pthreads into this computational-heavy model is favorable enough to warrant a recommendation for its widespread use in conjunction with parallel models. Combining both of these libraries does not save any total memory per processor and the differences between the two can most likely be attributed to overhead between process communication and thread creation.

The sieving algorithm, on the other hand, definitely leads to contrasting results which are similar to those found on Occam. As highlighted in Figure 4, both start out in a similar place with a single thread before dipping down significantly when an additional thread and processor is added into the test. The rate of speedup between these steps is calculated at nearly 25 times faster. Out of all runs between systems and job iterations, this proves to be the most impressive of the observed speedups.

Figure 4 Performance of Naïve and Sieving Algorithm on Tungsten with Combined Increasing Processor/Thread Counts



This sieving algorithm is an ideal look as to how to choose between MPI, Pthreads, or a mixture of the two. In this example, after two processors and threads are utilized, the speedup decreases drastically, but still outperforms a pure parallel implementation by a wide margin. The intensive use of memory is clearly advantageous to a multithreaded program's ability to use shared resources to complete a task.

7. Conclusions

This paper introduced the benefits of using the MPI and Pthread libraries as a function of comparing shared and distributed memory models. The parallel scaling of MPI allows for explicit message passing to divide and conquer tasks among a group of processors. Pthread's ability to time slice and create concurrent execution instances on a single processor allow it to share global memory and maximize a machine's resources. It is shown throughout the analysis of both investigated algorithms that utilization of either library is strongly dependent on the number of resources needed to complete a task.

It is also worth noting the difficulty required in turning both of the serialized algorithms into parallel and multithreaded forms. The message passing architecture generally necessitates more work from the programmer in order to determine a changing instruction flow to compensate for data exchange between all of the given processes. Conversely, the Pthread library comes with a much simpler API to create and destroy threads with the

only complexity resulting from maintaining a barrier of mutex exclusion locks to prevent multiple instances from accessing the same block of memory simultaneously. From this standpoint, if a quickly developed solution is necessary, a multithreaded model is definitely worth exploring.

And finally, while taking a step back and looking at the benefits of the libraries, it is not wise to completely disregard the choosing of an ideal algorithm. A naïve implementation running on a single processor took nearly twenty minutes to complete on a large bounded test, which was surpassed easily by an advanced sieving design running with a combination of parallel and multithreaded techniques. Its final runtime took less than a tenth of a second to complete for 100 million numbers. The blending of these two methodologies in conjunction with a cost efficient algorithm and a high-end machine yield surprisingly faster results for this project model and are good choice in future endeavors.

References

[1] Message Passing Interface Forum (MPIF) *MPI: A Message-Passing Interface Standard*. (November 2003)

[2] David R. Butenhof (1997). *Programming with POSIX Threads*. Addison-Wesley, 1997.

[3] Electronic Frontier Foundation (EFF) *Cooperative Computing Awards*. (March 1999) http://www.eff.org/awards/coop.php

[4] H. N. Gabow (2006). *Introduction to Algorithms*. University of Colorado, 2006.

[5] H. Halberstam and H.E. Richert. (1974). *Sieve Methods*. Academic Press, London, 1974.