Load balanced Parallel Prime Number Generator with Sieve of Eratosthenes on Cluster Computers*

Soonwook Hwang*, Kyusik Chung**, and Dongseung Kim*

*Department of Electrical Engineering
Korea University
Seoul, 136-713, Rep. of Korea
e-mail: dkimku@korea.ac.kr

**Department of Computer Engineering & Info. Technology
Soongsil University
Seoul, 156-743, Rep. of Korea
e-mail: kchung@soongsil.ac.kr

Abstract

Algorithms of finding prime numbers up to some integer N by Sieve of Eratosthenes *are simple and fast. However, even with the time complexity no greater than O(N ln ln N), it may take weeks or even years of CPU time if N is large like over 15 digits. No known shortcut was reported yet. We develop efficient parallel algorithms to balance the workload of each computer, and to extend main memory with disk storage to accommodate Giga-bytes of data. Our experiments show that a complete set of up to 14-digit prime numbers can be found in a week of computing time (CPU time) using our 8 1.6GHz Pentium-4 PCs with Linux and MPI library. Also, by sieve of Eratosthenes, we think it is very unlikely that we can compute all primes up to 20 digits using fastest computers in the world.*

Key words: prime number, sieve of Eratosthenes, load balancing, MPI, cluster computer

1. Introduction

Prime numbers gets people's attention due to their important role in data security using encryption/ decryption by public keys with such as RSA algorithm [6]. There are a lot of mathematical theories on prime, but the computation cost of big prime numbers is usually huge even with today's high performance computers.

Sieve of Eratosthenes (abbreviated by SOE later) [1,4] is a classic but straightforward way of finding all primes in a given range. Let's review the method. The sieving of numbers starts with an array A[1:N] whose initial value is set to zero. Beginning with 2, which is the smallest prime, all multiples of the prime up to N are *marked* to 1, i.e. A[4]= A[6]= … = A[2k]= …=1 (k is an integer). Now, the next smallest unmarked integer 3 is used to mark its multiples in the same way; A[6]= A[9]= … = A[3k']= … = 1. This procedure continues until we find a new smallest prime x such that $x^2 \geq N$. Then, all unmarked integers with A[ ]=0 are gathered.

The algorithm seems serial in nature, since prime numbers are found one by one in an increasing order, and all primes lying between $\sqrt{X}$ and X can be found only after we find all prime numbers up to $\sqrt{X}$.

Let S denote the set that contains all prime numbers from 2 to Q to generate primes up to N, where Q is the greatest prime less than or equal to $\sqrt{N}$. The prime numbers in S are called *sieving primes.* For simplicity let's assume that S is indexed in increasing order. Then, $S = \{2,3,5,7,\cdots,Q\}$. Let [X:Y] denote an *interval* (or a range) that contains all the integers between X and Y inclusive. Sieving in the interval is a process of marking all multiples of each sieving prime in the interval, and then collecting only unmarked numbers as prime.

In the literature, there are a few sequential algorithms of prime generation reported to reduce the

* This research was supported by KOSEF Grant (R01-2006-000-11167-0) and by Korea University Research Grant.

complexity close to O(N) [3,4,5]. It is reported that the k-th prime is greater than $k(\ln k + \ln \ln k - 1)$ for $k \geq 2$ [3]. Many big prime numbers are reported [7,8].

This paper presents a parallel prime generation method to extend the work of Bokahari [1] with the combined (static and dynamic) load balancing algorithm that we apply SOE onto distributed memory parallel computers (cluster) and expand the range of integer N beyond the main memory capacity using hard disk storage. We develop methods of partitioning the work load, and devise the way of efficiently representing and storing big integers. The proposed algorithms are expected to deliver the best performance if they provide even distribution of the load to all computers.

2. **Load Model of Sieve of Eratosthenes**

Let's find the computational load for the work. The major computational load of sieving in the range 1:N is to mark (or write) on the array whose index matches some multiple of sieving primes. The number of write operations for the multiple of the first prime 2 is N/2. Similarly, it is N/3 for the second prime. Thus, $L[1:N] = \frac{N}{2} + \frac{N}{3} + \frac{N}{5} + \cdots + \frac{N}{Q} = \sum_{p \in S} \frac{N}{p}$. This load is bounded by $N \ln \ln N + O(N)$ [3]. If a number x is a multiple of some prime p, the remainder after an integer division should be zero. If x is a multiple, then $x + p$, $x + 2p$, … are also multiple. Thus, multiples can then found by addition instead of division, which saves the computational cost. It is known that there are X/ln X prime numbers [7, 8], thus, if we set X as a 14-digit number, there are 10^{12} prime numbers that easily exceed the 1000 GB hard disk capacity to store if they are represented in 64-bit format. To compute big integers using SOE, we firstly need to overcome the limit of main memory capacity. Only M integers are loaded on main memory at a time and sieving computation is done, where M is the capacity of main memory. Thus, we have to repeat the marking at least N/M times as shown in Iterate_Sieving( ) below. We assume that the set of sieving prime is available before the iteration, or it can be computed very quickly.

```
Void Iterate_Sieving(N, M)
{
 Find all sieving primes up to √N ;
 for (I=0; I < C; I++)    /* C= N/M */
                          /* Covers the array A[I*M+1:(I+1)*M] */
  {
   1) Read in working prime set up to
        √((I+1)M)  from the disk;
   2) Mark A[ ] as 1 that are multiple of
      the number in working prime set.;
   3) Write  A[I*M+1:(I+1)*M] to disk;}
 }
```

Cost-wise, throughout the iterations line 2) and line 3) in Iterate_Sieving() need O(N ln ln N) and O(N) time, respectively. However, the time complexity of the above algorithm increases since it has the term $\Omega(N\sqrt{N}/\ln N)$ because of line 1) as explained below. When N is large, we can not load the whole sieving prime numbers into memory(or in cache block), instead we may read in only necessary prime numbers, called *working prime set,* from the disk (or from main memory). At iteration i , the greatest integer in main memory for sieving is Mi , thus the number of sieving primes needed is $\sqrt{Mi}/\ln\sqrt{Mi}$. The overall time complexity $t[1:N]$ of loading the working prime set onto main memory is obtained as

$$t[1:N] = \sum_{i=1}^{C} \sqrt{Mi}/\ln\sqrt{Mi} \geq \sum_{i=1}^{C} \sqrt{Mi}/\ln\sqrt{MC}$$

$$= (\sqrt{M}/\ln\sqrt{MC}) \sum_{i=1}^{C} \sqrt{i}$$

$$t[1:N] \geq (\sqrt{M}/\ln\sqrt{MC}) \int_{1}^{C} \sqrt{x}\,dx \geq \frac{\sqrt{M}}{\ln\sqrt{MC}} \times \frac{2}{3}\left[x\sqrt{x}\right]_{1}^{C}$$

$$t[1:N] \geq \frac{\sqrt{M}}{\ln\sqrt{MC}} \times \frac{2}{3}(C\sqrt{C} - 1)$$

By substituting $C = \frac{N}{M}$, the above relation can be rewritten as

$$t[1:N] = \Omega(N\sqrt{N}/\ln N)$$

3. Parallel Algorithms

For parallel execution, the sieving computation can be partitioned to P processors in two ways. Either the marking space of array A[1:N] can be partitioned to *P segments* (subarrays), then all numbers that are multiple of any of the sieving primes are marked in each segment simultaneously. Or the entire set of sieving prime are divided into *P disjoint subsets*, and each processor marks on its local copy of A[1:N] for the multiples of sieving primes in its own subset, then only those numbers are collected as prime whose P copies have never marked by any processor. This can be implemented by "reduce" operation in MPI library. The two schemes are called *array partition* and *set partition*, respectively. For simplicity assume here that $P = 2^K$ for some integer K. Division into P equal load is done in a recursive manner as explained below. Our parallel computing adopts master/slave configuration so that the master is in charge of finding the sieving prime set S, while slaves sieve numbers in the range allocated to them. Another role of the master in the configuration is to supervise dynamic load distribution as explained later.

For set partition we should know how to partition the sieving prime set S into P disjoint subsets to provide equal load to all processors, while in array partition we must divide the intervals 1:N into P segments each of which will have even load. We start from two equal partitions. The *half-load set boundary* is the location that divides the sieving prime set into two subsets (left subset and right subset) so that the load to mark with left subset is equal to the load to mark with right. In other words, each demands a half the overall load by the original set. The total load denoted by $L[1:N]$ is approximated as

$$L[1:N] = \sum_{p \in S} \frac{N}{p} \le \sum_{i=2}^{\sqrt{N}} \frac{N}{i} \le \int_1^{\sqrt{N}} \frac{N}{x} dx .$$

We also know the relation

$$\int_1^{\sqrt{N}} \frac{N}{x} dx = N \ln \sqrt{N} = \frac{1}{2} N \ln N .$$

The complexity may not be as tight as given in [4], but it will be applied in later computation. Let $(u \sim v)$ represent a subset of S that includes all primes between u and v. (Suppose $Z \approx z^2$) Now, we want to know the point Z where the number of sieving operations in the interval [1:N] with primes $(u \sim z)$ is a half the number with primes in $(u \sim v)$, in other words, the load of sieving with prime numbers in $(u \sim z)$ is equal to the load with those in $(z^+ \sim v)$, where z^+ is the immediate next prime number after z.

$$\sum_{p=u}^{z} \frac{N}{p} = \sum_{p=z^+}^{v} \frac{N}{p}$$

Then, the summation equations are converted to integration by approximation as $\int_u^z \frac{N}{x} dx = \int_z^v \frac{N}{x} dx$.

$N \ln z - N \ln u = N \ln v - N \ln z$,

$2 \ln z = \ln u + \ln v, \ \therefore z = \sqrt{uv}$

Now $Z = uv$, which means that the sieving computation with primes in $(u \sim \sqrt{uv})$ needs the half load of the computation with primes in $(u \sim v)$. If the half-load boundary formula is applied again to both sets $(u \sim z)$ and $(z \sim v)$, we will get four equal-load partitions. The method can be recursively applied to divide the original load to $P = 2^K$ equal-load partitions, as described in Algorithm Set_Partition.

```
Void Set_Partition(L, R, P)
{
  if (R-L > Min_interval) AND (P>=2)
    /* Min_interval is the threshold for parallel computation. */
   { mid = √LR ;
     Set_Partition(L, mid, P/2);
     Set_Partition(mid, R, P/2)}
  else
   {Allocate all (L~R) to any of P;
        /* (L~R) is a set of primes. */
    exit;}
 }
```

For another way of load partitioning, let us find the *half-load point* that is the boundary point of division of the interval [1:N] into two equal-load segments, [1:y], and [y+1: N]. The half-load point x can be found as below, where S' is the subset of S that includes all primes less than $\sqrt{y}$.

$$L[1:y] = \sum_{p \in S'} \frac{N}{p} \cong \frac{1}{2} N \ln \ln N$$

To find y that satisfies the above relationship is not easy, so we instead find an approximation Y. If N is

very large such as $N \geq 10^{10}$, the approximation is obtained as $y \approx Y = N/2$ with very small error. The relative error f a non-increasing function and is very close to 1 for $N \geq 10^{10}$ as shown below ($0 < f < 1$).

$$f \equiv \frac{L[1:y] - L[1:Y]}{L[1:y]} = \frac{\frac{1}{2}N\ln\ln N - \frac{1}{2}N\ln\ln\frac{N}{2}}{\frac{1}{2}N\ln\ln N}$$

$$= 1 - \frac{\ln(\ln N - \ln 2)}{\ln\ln N} < 1 - \frac{\ln(\ln 10^9 - \ln 2)}{\ln\ln 10^9} \approx 0.01$$

The approximation is valid for a big N.

4. Experiments and Discussion

The prime number generator has been experimented on cluster computers. One of the clusters consists of 8 1.83GHz AMD processors, each with a 80GB local disk and 1GB main memory. All nodes are considered fully connected by Fast Ethernet and Myrinet switches. Another cluster is planned to use, which is 64-node Myrinet cluster in KISTI [9]. Our experiments are performed on the 8-node cluster, and maximum execution time is currently limited to 24 hours. Under the constraint, we can compute the primes up to $N = 10^{12}$.

To reduce the overheads in the execution, we modify the sieving process as explained below. Firstly, to save the storage and disk/memory access time, big integers/indexes like over ten digits, (normally 32-bit integer representation covers only up to ten digits) are grouped with common prefix in binary bits and only the lower bit/digit information is specifically represented. Secondly, the order of marking is adjusted to overcome the cache limitation. Finding an integer that is a multiple of some prime (actually the index *i* of A[*i*]) is usually done by an addition; since if x is a multiple of some prime p, then $x + p$, $x + 2p$, … are also multiple. Since we have a limited size of cache (or cache line), the next multiple may not be in the cache when p is greater than cache line size. In such a case, every attempt to marking generates a cache miss. In this respect, a certain number of sieving primes adjacent each other are grouped, and marking is done for all primes together, instead of one prime by one prime.

As introduced earlier, set partitioning is simple to implement, however, it requires synchronization ("reduce" operation) of the whole array A[1:N] from P processors after individual marking operations are complete. It adds communication cost of O(N log P), which may not be neglected compared to sieving cost of N ln ln N. Thus, the algorithm is practical for small N, and our experiments focus on set partitioning.

Since the half load boundary is an approximate value, there can be some imbalance among different partitions. To get complete balancing of load throughout the computation, the *static* load distribution will be combined with *dynamic load balancing*. Static distribution allocates only a fraction q ($0 < q \leq 1$) of the interval [1:N] to P processors. The remaining interval is equally divided into $R(>> P)$ *small* segments for dynamic load distribution, thus, processors that finish sieving on the statically assigned intervals will receive small segments from the master for execution until no more are left.

Figure 1 shows the predicted and observed times of sequential execution with various ranges, where the upper limit N (= n) is chosen as $\frac{1}{2} \times 10^{11}$ for the convenience of the experiments. The plot reveals that the running times are close to $O(N\sqrt{N}/\ln N)$, whose reason was due to the limited main memory/cache capacity as explained in section 2.

Figures 2 to 4 plot speedups of parallel computation with three job partition schemes: *static only*, *static and dynamic combined*, *and dynamic only*. Static simply divides the intervals equally, which gives too inferior results. The reason is in that even though the counts of marking are equal, real load due to *cache misses* is not reflected at all. The same problem is relieved in dynamic and combined partitioning, since the scheme controls load during run time. When the problem size is big (i.e. 8*n*), parallel computation with 8 processors of dynamic and combined partition schemes has best performance since it gets smallest work load per job favorable to cache hits.

We observe in the plots super-linear speedups are obtained, and it is since the sequential executions experiences saturated operations due to the heavy disk traffic and high cache misses because of overloaded resource utilization than other parallel ones. Also, parallel computing is suitable in the computation since there is very little overhead in communication for master/slave job distribution.

As the range N grows, in addition to huge CPU time consumption, the sieving also requires other resources to the limit such as hard disk capacity. To cover 18 digit prime numbers, we expect the hard disk should be as large as 1TB. We predicts that it needs days of

computing time to get 15 digit primes with our 8 node cluster, while 18-digit needs a year of computation.

5. Concluding Remarks

This research focuses on efficient parallel computation of prime numbers in a given range. Load is distributed to each processor statically and dynamically, thus, eventually all processors spend the same amount of computation time to achieve the minimum execution time. The partition of load to processors was based on division of computing intervals and division of prime numbers in the set S of sieving primes. Saturation of the performance is observed due to full load to disk and cache misses in both sequential and parallel prime generators. We expect that the complete set of prime numbers up to 10^{14} can be obtained in days of execution, and 10^{16} in a year, although the times could vary depending on the speed of the parallel computers.

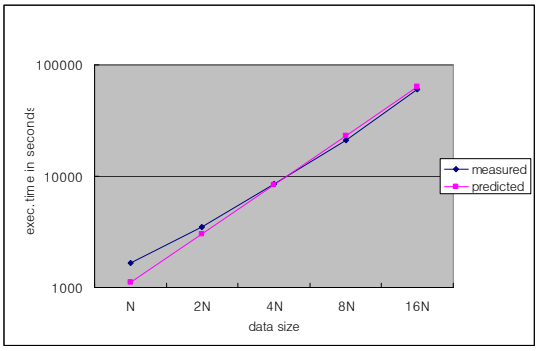

Figure 1 Sequential time complexity

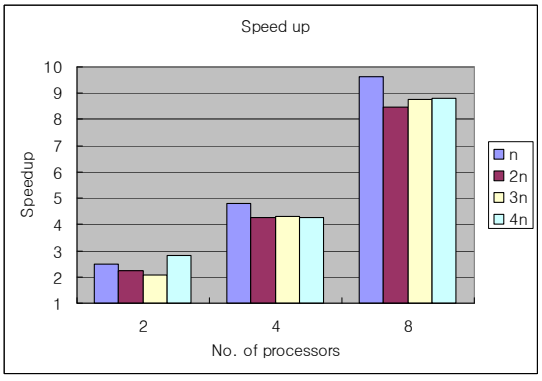

Figure 2 Speedup (static & dynamic combined)

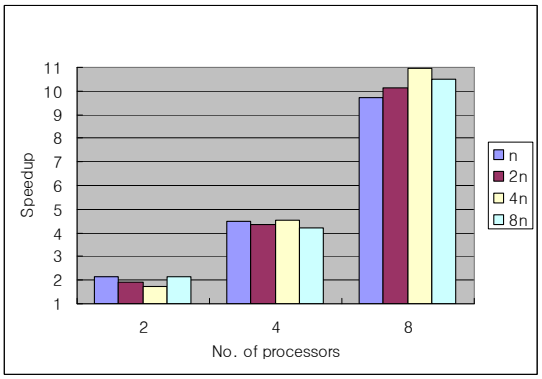

Figure 3 Speedup (dynamic load distribution)

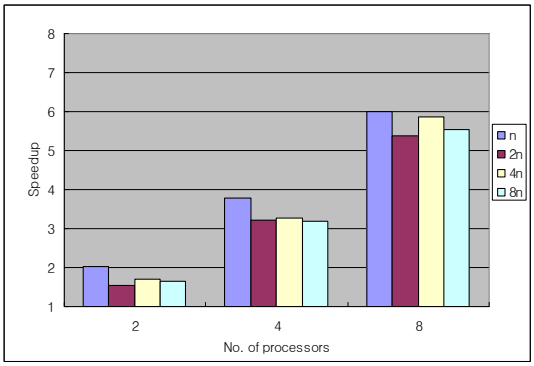

Figure 4 Speedup (static load distribution)

References

[1] S.H. Bokhari, Multiprocessing the sieve of Eratosthenes, *IEEE Computer* 20(4): April 1984, pp.50-58

[2] P. Dusart, The K-th prime is greater than k(ln k + ln ln k-1) for k >= 2, Mathematics of Computation, Vol 68, No. 225, pp. 411-415, Jan. 1999.

[3] G. Hardy and E. Wright, *An introduction to the theory of numbers*, Clarendon Press, 1979.

[4] X. Luo, A practical sieve algorithm for finding prime numbers, *Communications of the ACM*, Vol. 32 No. 3, pp. 344-346, March 1989.

[5] G. Pailard, C. Lavult, and R. Franca, A distributed prime sieving algorithm based on Scheduling by Multiple Edge Reversal, *Int'l Symp. On Parallel and Distributed Computing*, Univ. Lille 1, France, July 4-6, 2005.

[6] Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public key cryptosystems, *Comm. ACM* 21 (Feb. 1978), pp. 120-126

[7] http://www.prime-numbers.org

[8] http://primes.utm.edu/howmany.shtml, How many primes are there?

[9] http://www.ksc.re.kr KISTI Supercomputing Center.