Parallel Algorithms for the Third Extension of the Sieve of Eratosthenes

Todd A. Whittaker
Ohio State University
whittake@cis.ohio-state.edu

Kathy J. Liszka
The University of Akron
liszka@computer.org

Abstract

A prime number sieve is an algorithm for finding all prime numbers in the range [2,n]. For large n, however, the actual run time of existing algorithms on a single processor computer is prohibitive. Fortunately, the classic Sieve of Eratosthenes and its extensions are suitable for massively parallel architectures or networks of workstations. Two mappings of the third extension sieve are presented. One is for a massively parallel SIMD architecture and the other is for a network of workstations using the PVM message-passing library.

1. Introduction

In addition to number theorists, prime numbers are of great interest to two groups: code makers and code breakers. Public key / private key encryption is based on large prime numbers, as are the methods of decrypting. The code makers encrypt by choosing two large prime numbers *p* and *q*, whose product *n* becomes part of a key. The code breakers try to reverse this process through a variety of factoring algorithms. Previous practical parallel sieve efforts by Bokhari [2] and Landsdowne [5] demonstrated prime generation through 10^6.

2. The Classic Sieve and Other Popular Solutions

In the original algorithm attributed to the fourth century B. C. library Eratosthenes, all the numbers from 2 to n are written on a piece of paper. Starting with 2, all multiples of 2 greater than 2 are crossed off the page. The next number after 2 which is not crossed off is 3, so all multiples of 3 greater than 3 are removed. This process continues until $\sqrt{n}$ is reached, the largest possible factor of *n*. Remaining numbers, when the algorithm completes, are prime.

This sieve is classified as additive, as the composites are identified by adding *p*, the prime under consideration, to the current composite. This algorithm has super-linear time complexity because a composite number is crossed off as many times as it has prime factors. It was shown in [13] that the overall time complexity of the original sieve is O(*n* log log *n*) and the bit complexity (the number of bits of storage required) is O(*n*).

The original sieve considers the possibility that any number in the range 2..*n* may be prime, and therefore must check even obviously composite numbers such as 4, 6, 8, and so forth. Extensions, or wheels [10] eliminate these obviously composite numbers from consideration by pre-sieving them via a particular initialization pattern, or by using a function to map the reduced set onto the integers.

It is generally accepted that Eratosthenes derived the first extension, which does not contain multiples of 2 greater than 2, i.e., $V = \{2\} \cup w$ where $w = \{2i + 1 \mid i \in N\}$. In doing so, Eratosthenes not only halved the quantity of numbers he had to write and cross off, but he also halved the time to complete the sieve. The second extension [7] contains no multiples of 2 greater than 2 or multiples of 3 greater than 3, i.e., $V = \{2, 3\} \cup w$ where $w = \{6i \setminus 1 \mid i \in N\}$. Generalizing, the k^{th} extension has no multiples

of the first k primes [6]. Furthermore, if all the primes through $\sqrt{n}$ are known, then the range $(\sqrt{n}]$ may be segmented [1] into N pieces which may be sieved independently, reducing the space complexity to $O(\sqrt{n} + \delta)$ where $\delta = (n - \sqrt{n})/ N$ is the size of the segment.

The goal of a linear sieve is to reduce the time complexity by a factor of log log n over the classic sieve to $O(n)$; however, this is at the expense of additional arithmetic complexity. Examples of these are shown in [8] and [4]. It was then shown in [9] that these approaches are multiplicative and hence, have a higher time complexity.

The goal of the sub-linear sieve as given by Pritchard [9] is to reduce the asymptotic time complexity to $O(n/\log \log n)$ and to maintain the additive arithmetic complexity of the classic sieve. In linear sieve algorithms, a vector $V = (2, 3, 4, \ldots, n\}$ is initialized and each composite is removed exactly once. To make a sieve sub-linear, it is not only necessary to remove each composite once, it is also necessary to place fewer composites into V initially. Pritchard generates V dynamically, adding primes and only a select few composites per iteration. Some of the composites are removed, while others are used to generate larger primes in future iterations; however, fewer composites are added each iteration than in the linear time sieves. The final iteration removes all remaining composites from V.

Though the linear and sub-linear sieves look attractive at first glance, they suffer from a number of problems when considering a practical parallel implementation. First, the fastest and most obvious implementations require one word of storage per element of V. Composites removed from V still occupy valuable space in memory. Attempting to bit pack the elements, or to use dynamic memory allocation introduces a prohibitive amount of overhead. Second, the linear and sub-linear algorithms require some form of global state, and thus are not suited for parallel message passing architectures. Finally, though both the linear and sub-linear sieves are asymptotically faster than the classic sieve, it has been shown in [3] and [13] that variations on Eratosthenes original algorithms are the most practical for feasible inputs.

3. The Third Extension of the Sieve of Eratosthenes

Extensions and wheels remove "obviously" composite numbers (numbers which are composites with at least one factor of a very small prime) from V before the sieving process begins. The first extension does not consider multiples of 2, the second does not consider multiples of 3 and the third does not consider multiples of 2, 3, and 5. Wheels define a bit pattern which can be used to initialize a bit vector so that the composites of small primes are already crossed out. Extensions take this one step further by defining a mapping between the elements in a bit vector and the integers relatively prime to the first k primes, thus, no bits are wasted.

We define and use the following symbols:

π_k the product of the first k primes defined by

$$\pi_k = \prod_{i=1}^{k} p_i$$

C_k the set of integers less than and relatively prime to π_k.

m_k the cardinality of C_k defined by
S_k the set of integers through which to sieve defined by

$$m_k = \prod_{i=1}^{k} (p_i - 1)$$

The third extension defines $k = 3$ in the above, and the following are derived:

$$S_k = \{i \cdot \pi_k + c | i \in N, c \in C_k\}$$

$\pi_3 = 2 \cdot 3 \cdot 5 = 30$
$C_3 = \{1, 7, 11, 13, 17, 19, 23, 29\}$
$m_3 = |C3| = 1 \cdot 2 \cdot 4 = 8$
$S_3 = \{ 1, 7, 11, 13, 17, 19, 23, 29, 31, 17, 41, 43, 47, 49, 53, 59, 61, \ldots\}$

By mathematically arranging the elements of S_3 a clear pattern emerges that will allow this sieve to be additive; that is, once the first composite is found, subsequent composites are a fixed offset away. Rearrange S_3 into a two dimensional grid m_k columns wide as shown in

Table 1. Composites of the prime p are always p rows away from one another *in the same column*.

Because $m_3 = 8$, it is convenient to represent a row of S_3 in a one byte data structure. For the standard sieve, all that remains is to identify the first composite of a prime p in each column. Subsequent composites will be found p rows down in the same column, as seen in Table 1 for multiples of 7. By defining two operations, we construct a table to locate the first multiple [11].

$rindex(n) = \lfloor n/\pi_k \rfloor$
$cindex(n) = i \mid (C_i \in C_k) \wedge (C_i = n \bmod \pi_k)$

Briefly, *rindex*(n) returns the row of the first multiple of n, and *cindex*(n) returns the column of the first multiple of n. However, multiples exist in all columns. These formulas can be used to construct a table to find the rows at which multiples exist in the remaining columns. The table is constructed by generating the ordered pairs of [*rindex*(n), *cindex*(n)] for all $n = i \cdot j$, $\mid$ $i,j \in C_k$ and is shown in Table 2.

As noted, S_3 has the property that each number in this table may be stored in one byte of data, so the table itself is small at 128 elements. By comparison, the table for S_4 would have 4608 elements while the table for S_5 would require well over one megabyte of storage.

To find the first multiples in each column of the prime p, let $r = rindex(p)$ and $c = cindex(p)$. Use c to select a row from Table 2. Let the vector A represent the *rindex* entries and the vector B represent the *cindex* entries from the *cth* row. Now, the position of all the composites of p may be found by choosing the rows from the vector $r \cdot C_3 + A$ and the columns from B. For example, to locate the first multiple of 37 in all the columns, calculate $r = rindex(37) = \lfloor 37/30 \rfloor = 1$ and $c = cindex(37) = 1$. Using zero based indexing, choose the row {[0,1],[1,5],[2,4],[3,0],[3,7],[4,3],[5,2],[6,6]} from Table 3 and create the vectors

$A = \{0, 1, 2, 3, 4, 5, 6, 7\}$
$B = \{1, 5, 4, 0, 7, 3, 2, 6\}$.

The rows in which the multiples may be found are

$S = 1 \cdot \{ 1,7,11,13,17,19,23,29\} + \{0,1,2,3,4,5,6,7\} = \{1,8,13,16,21,24,29,36\}$.

When S is recombined with the column entries in B, the ordered pairs of (row, column) of

{(1,1), (8,5), (13,14), (16,-), (21,7), (24,3), (29,2), (36,6)}

comprise the positions of the first π_3 composites of 37 in S_3.

3.1. Segmenting the Third Extension

A segmented sieve divides the numbers {2 … n} into N segments of length $\delta = (n - \sqrt{n}) / N$. Each segment is sieved independently, reducing the storage requirements by a factor of N. Segmenting requires that all the primes through $\sqrt{n}$ be known in advance, or that these be calculated in the initial stages of the algorithm and stored for use in subsequent iterations. Since these primes must be stored in $O(\sqrt{n})$ space, no space complexity is saved by choosing N less than $\sqrt{n}$, so choosing a segment of size approximately $\sqrt{n}$ is ideal [12].

Table 2 and several simple calculations produce the row and column entries in S_3 of the first multiples of a given prime. However, in a segmented sieve, S_3 is divided into sections, and the starting row number for each section will be different. Thus, the offset to the first multiple will also be different. The starting row number of a segment can be used to adjust the values in Table 2 so that the first multiples of a prime p may be located in that segment. This adjustment is used in both a single processor segmented sieve and a data parallel sieve.

Let R be the starting row number of a segment for an iteration of a segmented sieve. The first composites in this subset of S3 will be found at

$$S'[i] = \left\lceil \frac{R - S[i]}{p} \right\rceil \cdot p - (R - S[i])$$

where S is the vector or row entries calculated for the segment starting with row 0 (Table 1). For example, let R = 300 (perhaps this is the third iteration of 100 rows of the table). From the previous example for the prime 37, S = {1, 8, 13, 16, 21, 24, 29, 36} are the rows in which the first composites are found. Applying the above formula to this vector yields the following:

$\lceil (300 - 1)/37 \rceil \cdot 37 - (300 - 1) = 34$
$\lceil (300 - 8)/37 \rceil \cdot 37 - (300 - 8) = 4$
… …
$\lceil (300 - 36)/37 \rceil \cdot 37 - (300 - 36) = 32.$

The new vector S = {34, 4, 9, 12, 17, 20, 25, 32} is recombined with the B vector to yield the position pairs {(34,1), (4,5), (9,4), (12,0), (17,7), (20,3), (25,2), (32,6)}.

4. Parallel Mappings of the Third Extension

We now present a MIMD implementation using PVM (Parallel Virtual Machine) over a network of SPARCstations and a SIMD implementation on a MasPar MP-1 with 4096 processing elements for the sake of demonstrating different mapping strategies for two opposing parallel environments. It is our intent to focus on mapping designing issues and tradeoffs rather than benchmark timings that are influenced by current technology.

4.1. MIMD Row Farming

Consider a network of workstations and a farming algorithm. The approach is to treat S_3 as a standard segmented sieve and assign a contiguous block of numbers (a subset of the rows of S_3) to each processor. With each row of S_3 eight columns wide, a single row of S_3 is represented in one byte. The algorithm will make eight passes through memory and must calculate a vector of eight (*row*, *column*) pairs per prime. A good degree of parallelism is present in this approach. However, since several different architectures are present in the network, it is best to distribute the work dynamically as processors become available rather than assign work statically. In essence, the farming algorithm starts N worker processes, sends them a set of initial data and a job assignment, and waits for the processes to return their results. The master process also performs tasks, which differentiates a farming algorithm from a master/slave algorithm. When the master process receives a result, it sends out a new job assignment to the available worker. The dangers in this type of setup are not easily resolved: the master process may become a bottleneck if too many workers are spawned; the master process may respond too slowly to a return value from a worker; the master process is a single point of failure; and given that communication is inherent, parallelism can be limited.

The row farming algorithm is divided into two major components: the master algorithm and the worker algorithm. The master algorithm spawns N worker processes and sends them each a section of S_3 to process. By sending the starting row (R), the number of rows through which to sieve (δ) and the number of iterations through those rows (*itr*), the worker process has enough information to create and sieve its segment. The master process sends the same number of iterations and array size to each worker, however, this need not be the case. Given the memory and processor characteristics of the worker machine, it may be more efficient to iterate more times through a smaller array, or fewer times through a larger array. Additionally, the same sizes need not be sent every time to the same worker, provided the performance penalty of dynamic memory allocation between iterations is small. Each worker pre-generates all the primes through $\sqrt{n}$ which will be used in the segmented sieving process. It is more efficient to generate them locally and in parallel rather than pay the communication price for the master process to generate them and send the set to each worker. The workers then enter a loop where they receive (R, *itr*, δ) as their work assignment. They iterate *itr* times over δ simulating a segment of size $itr \cdot \delta$, removing multiples of each p from S_3 starting at row R.

In the MIMD farming model, each processor is assigned a new task as soon as it finishes its last task, which increases concurrency and efficiency. Higher numbered iterations still take longer than lower numbered iterations, but all processors are in use. A distinct advantage to this approach is that the model scales well to more processors. In empirical studies, we found that when the tasks are evenly divided among the processors, the speedup is nearly linear. Obviously, when slower processors are added to the pool they limit the total speedup.

4.2. SIMD Column Segmenting

On a massively parallel architecture, we find that there are several naïve mappings. Row segmenting, as described above, is one of them. The MIMD farming algorithm could be easily adapted to a synchronized environment such as the MasPar where some of the master process

duties are performed by an array control unit and the worker processes are performed by the processing elements (PEs). There are many limitations to this approach, the most striking being lower numbered PEs are idle more than higher numbered PEs , thus limiting parallelism. The synchronization of this model does not permit lower numbered PEs to sieve another range while the higher numbered ones complete the previous range. This is why we found the approach suitable for a MIMD environment but not massively parallel SIMD.

Rather than segment S_3 by rows, consider the columns: each PE may hold a portion of a single column, reducing the number of calculations for offsets from eight to one. Since S_3 has eight columns, the assignment of column segments to the 4096 PEs in the MasPar, for example, is simple. Instead of representing a contiguous segment of S_3 in each PE, the 0^{th} PE holds as many numbers as memory will permit of the form $1 + \pi_3 \cdot x$, the 1^{st} PE holds numbers of the form $7 + \pi_3 \cdot x$, and so forth. In general, let R be the number of rows of S_3 skipped to generate all primes through $\sqrt{n}$. Let δ be the number of elements of S_3 to be help in each PE. Finally, let j be any number in $[0 .. \delta]$. The general column based mapping for 4096 PEs is given in Table 3. In the algorithm, a main loop iterates over the small set of pre-generated primes and the inner sieve calculates the starting position of the composites and additively removes them.

SIMD column segmenting is more efficient than row segmented mapping. In column segmenting, there is no need to calculate a vector of offsets, nor is there need to make eight passes through memory to remove the composites. However, the sets of primes left in each PE are not contiguous. To produce a sorted list of primes, it is necessary to bitwise interleave the segments of S_3 in each set of eight adjacent PEs.

5. Conclusion

Many different algorithms for finding prime numbers exist. The classic sieve of Eratosthenes, though by nature additive, is of super-linear time complexity. Linear and sub-linear algorithms offer a better asymptotic time complexity, but are not as suitable for parallel implementation. Extensions and wheels improve over the classic sieve by removing composites of very small primes, but this only reduces the proportionality constant of its time complexity. The third extension to the sieve of Eratosthenes is an excellent algorithm for parallel implementation. By defining a mapping between the elements of a bit vector and the set of integers relatively prime to 2, 3, and 5, this extension preserves its additive nature and reduces both time and space requirements. Additionally, since $m_3 = 8$, a one byte data structure may conveniently hold a row of S_3, or a set of 2^i processors may evenly segment the columns of S_3. Both a row base segmented mapping and a column based segmented mapping were presented.

The row mapping is a standard segmented sieve, in which each processor receives a continuous section of S_3. Unfortunately, this solution leads to a high percentage of time spend calculating the positions of composites in the columns of S_3. This is not good candidate algorithm for a SIMD synchronized architecture because the work distribution is static. Using a MIMD farming environment with current CPU technology increases the concurrency and arithmetic efficiency, greatly reducing runtime. It is also a more scalable and portable approach. The SIMD column mapping produces good empirical results even on a dated MasPar MP-1. There is still restriction on concurrency where the workload among PEs becomes unbalanced, but it is not as significant as a SIMD row segmented algorithm.

Future work includes studying the column segmented algorithm on a large cluster of workstations for very large n, in the range 10^{12} through 10^{18}. We are also working on the design of efficient parallel algorithms to begin prime generation from very large numbers instead of starting with single digit composites and primes as is required by well known prime generating algorithms.

6. References

[1] C. Bays and R. H. Hudson, "The segmented sieve of Eratosthenes and primes in arithmetic progressions to 10^{12}", *BIT*, vol. 17, 1977, pp. 121-127.

[2] S. Bokhari, "Multiprocessing the sieve of Eratosthenes", *Computer*, vol. 20, no. 4, April 1987, pp. 50-58.

[3] B. Dunten, J. Jones, and J. Sorenson, "A space-efficient fast prime number sieve", *Information Processing Letters*, vol. 59, no. 2., July 1996, pp. 79-84.

[4] D. Gries and J. Misra, "A linear sieve algorithm for finding prime numbers", *Communications of the ACM*, vol. 21, no. 12, December 1978, pp. 999-1003.

[5] S. Landsdowne, "Reprogramming the sieve of Eratosthenes", *Computer*, vol. 20, no. 8, August 1987, pp. 90-91.

[6] K. Liszka and A. Quesada, "On the parallel k^{th} extension of the sieve of Eratosthenes", *Parallel Algorithms and Applications*, vol. 10, 1996, pp. 111-125.

[7] X. Luo, "A practical sieve algorithm for finding prime numbers", *Communications of the ACM*, vol. 32, no. 3, March 1989, pp. 344-346.

[8] H. Mairson, "Some new upper bounds on the generation of prime numbers", Communications of the ACM, vol. 20, no.9, Sept. 1977, pp. 664-669.

[9] P. Pritchard, "A sublinear additive sieve for finding prime numbers", *Communications of the ACM*, vol. 24, no. 1, Jan. 1981, pp. 18-23.

[10] P. Pritchard, "Explaining the wheel sieve", *Acta Informatica*, vol. 17, 1982, pp. 477-485.

[11] A. Quesada, Technical correspondence, *Communications of the ACM*, vol. 35, no. 3, March 1992, pp. 11-13.

[12] J. Sorenson, "An introduction to prime number sieves", Technical Report CS-TR-90-909, Dept. of Computer Sciences, University of Wisconsin-Madison, Jan. 1990.

[13] J. Sorenson, "An analysis of two prime number sieves", Technical Report CS-TR-91-1028, Dept. of Computer Sciences, University of Wisconsin-Madison, June 1991.