Parallel Prime Sieve: Finding Prime Numbers

David J. Wirian
Institute of Information & Mathematical Sciences
Massey University at Albany,
Auckland, New Zealand
dwir001@gmail.com

ABSTRACT

Before the 19th of century, there was not much use of prime numbers. However in the 19th of century, the people used prime numbers to secure the messages and data during times of war. This research paper is to study how we find a large prime number using some prime sieve techniques using parallel computation, and compare the techniques by finding the results and the implications.

INTRODUCTION

The use of prime numbers is becoming more important in the 19th of century due to their role in data encryption/decryption by public keys with such as RSA algorithm. This is why finding a prime number becomes an important part in the computer world. In mathematics, there are loads of theories on prime, but the cost of computation of big prime numbers is huge.

The size of prime numbers used dictate how secure the encryption will be. For example, 5 digits in length (40-bit encryption) yields about 1.1 trillion possible results; 7 digits in length (56-bit encryption) will result about 72 quadrillion possible results [11]. This results show that cracking the encryption with the traditional method will take very long time. It was found that using two prime numbers multiplied together makes a much better key than any old number because it has only four factors; one, itself, and the two prime that it is a product of. This makes the code much harder to crack.

In this paper, we are going to discuss about the parallel prime sieve. In the beginning, we are going to discuss about the use of prime numbers. An introductory of prime sieve will be explained in the next section. The best known prime number sieve is Eratosthenes, finds the primes up to n using O(n ln ln n) arithmetic operations on small numbers. We are also discussing how the sieves are run with parallel computation. This will include the analysis of each sieve, and also how we improve the technique to result more efficient.

PRIME SIEVE

A. Sieve of Eratosthenes

More than two thousand years ago, Eratosthenes described a procedure for finding all prime numbers in a given range. The straight forward algorithm, known as the Sieve of Eratosthenes, is to the only procedure for finding prime numbers [3].

An example of the sieve appears in Figure 1. In order to find primes up to 100, integer multiples of the primes 2, 3, 5, 7 are marked as composite numbers. The next prime is 11. Since the square of its value is 121, it is greater than 100 then it causes an end to the sieving loop. The unmarked integers that remain are primes.

Figure 1: The Sieve of Eratosthenes, finds primes between 2 and 100

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

(a) Mark all multiples of 2 between 4 and 100, inclusive. Result: 2

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

(b) The next unmarked value is 3, then mark all multiples of 3 between 9 and 100, inclusive. Result: 2, 3

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

(c) The next unmarked value is 5, then mark all multiples of 5 between 25 and 100, inclusive. Result: 2, 3, 5

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

(d) The next unmarked value is 7, then mark all multiples of 7 between 49 and 100, inclusive. Result: 2, 3, 5, 7, and all unmarked value (11, 13, …, 97)

Below is the pseudocode to find prime numbers using the sieve [6]:

1. Create a list of natural numbers 2, 3, 4, 5, ….., *n*. None of which is marked.

2. Set k to 2, the first unmarked number on the list.
3. Repeat:
    a. Mark all multiples of k between k^2 and n.
    b. Find the smallest number greater than k that is unmarked. Set k to this new value. Until $k^2 > n$.
4. The unmarked numbers are primes.

However, the Sieve of Eratosthenes is not practical for identifying large prime numbers with hundred of digits, because the algorithm has complexity O(n ln ln n), and n is exponential in the number of digits. A modified form of the sieve is still an important tool in number theory research.

B. Sieve of Atkin

The sieve of Atkin was created by A. O. L. Atkin and Daniel J. Bernstein. It is a fast and modern algorithm for finding all prime numbers up to a specified integer. It is an optimized version of the ancient sieve of Eratosthenes, but does some preliminary work and then marks off multiples of prime squared, rather than multiples of prime.

In the algorithm [1]:

1. All remainders are modulo-sixty remainders (divide the number by sixty and return the remainder).
2. All numbers, including x and y, are whole numbers (positive integers).
3. Flipping an entry in the sieve list means to change the marking (prime or nonprime) to the opposite marking.

    a. Create a results list, filled with 2, 3, and 5.
    b. Create a sieve list with an entry for each positive whole number; all entries of this list should initially be marked nonprime.
    c. For each entry in the sieve list :

        - If the entry is for a number with remainder 1, 13, 17, 29, 37, 41, 49, or 53, flip it for each possible solution to $4x^2 + y^2 = entry\_number$.
        - If the entry is for a number with remainder 7, 19, 31, or 43, flip it for each possible solution to $3x^2 + y^2 = entry\_number$.
        - If the entry is for a number with remainder 11, 23, 47, or 59, flip it for each possible solution to $3x^2 - y^2 = entry\_number$ when $x > y$.
        - If the entry has some other remainder, ignore it completely.

    d. Start with the lowest number in the sieve list.
    e. Take the next number in the sieve list still marked prime.
    f. Include the number in the results list.
    g. Square the number and mark all multiples of that square as nonprime.
    h. Repeat steps five through eight.

The sieve of Atkin computes primes up to n using O(n/log log n) operations with only n^(1/2 + O(1)) bits of memory.

PARALLEL COMPUTATION

We are going to focus on the sieve of Eratosthenes since the sieve is the one of the most popular way to search a large prime number. If we recall the previous section about the sieve of Eratosthenes, the heart of the algorithm is marking elements of the array representing integers, it makes to do domain decomposition, breaking the array into n-1 elements and associating a primitive task with each of these elements.

The key parallel computation is where the elements representing multiples of a particular prime k are marked as composite, i.e. mark all multiples of k between k^2 and n [2, 4, 5, 7, 9]. If a primitive task represents each integer, then two communications are needed to perform the step 3b each iteration of the repeat … until loop. We need a reduction each iteration in order to determine the new value of k, and then a broadcast is needed to inform all the tasks of the new value of k.

The advantage of reflecting on the domain decomposition is that there is plenty of data parallelism to exploit, but it takes lots of reduction and broadcast operations [5]. In the next section, we are going to introduce a new version of the parallel algorithm that requires less computation and less communication than the original parallel algorithm.

Interleaved Data Decomposition

Let us consider an interleaved decomposition of array elements:

process 0 is responsible for the natural numbers 2, 2 + p, 2 + 2p, …
process 1 is responsible for the natural numbers 3, 3 + p 3 + 2p, …
and so on..

The advantage of the interleaved decomposition is that given a particular array index I, it is easy to determine which process controls that index (process i mod p). However, it has a disadvantage of an interleaved decomposition for this problem. It can lead to significant load imbalances among the processes. It also still requires some sort of reduction or broadcast operations.

Block Data Decomposition Method

This method divides the array into p contiguous blocks of roughly equal size. Let n is the number of array elements, and n is a multiple of the number of processes p, the division is straightforward. This can be a problem if n is not a multiple of p. Suppose n = 17, and p = 7, therefore it will give 2.43. If we give every process 2 elements, then we need 3 elements left. If we give every process 3 elements, then the array is not that large. We cannot simply give every

process p – 1 processes [n/p] combinations and give the last process the left over because there may not be any elements left. If we allocate no elements to a process, it can complicate the logic of programs in which processes exchange values. Also it can lead to a less efficient utilization of the communication network.

Let $r = n \bmod p$
If $r = 0$, all blocks have same size
Else
    First r blocks have size n/p
    Remaining p-r blocks have size n/p

(*see figure 2a)*

Now, what is the range of elements controlled by a particular process? Which process controls a particular element?

Suppose n is the number of elements and p is the number of processes. First, scatter large blocks among processes. The first element controlled by process i is

$$\boldsymbol{i[n/p] + \min(i,r)}$$

The last element controlled by process i is the element before the first element controlled by process $i + 1$:

$$\boldsymbol{(i+1)[n/p] + \min(i+1, r) - 1}$$

The process controlling a particular array element j is:

$$\boldsymbol{\min([j/([n/p] + 1)], [(j-r)/[n/p]])}$$

Now we compare with another scheme in the block data decomposition. The second scheme does not focus on all of larger blocks among the smaller-numbered processes. Suppose n is the number of elements and p is the number of processes. The first element controlled by process i is:

$$\boldsymbol{[in/p]}$$

The last element controlled by process i is the element before the first element before the first element controlled by process $i + 1$:

$$\boldsymbol{[(i+1)n/p] - 1}$$

The process controlling a particular array element j is:

$$\boldsymbol{[(p(j+1) - 1)/n]}$$

(*see figure 2b*)

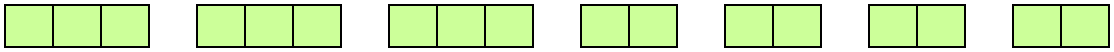

Figure 2a: 17 elements divided among 7 processes (first scheme)

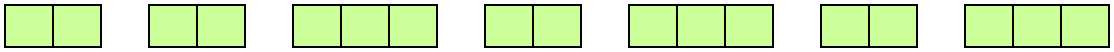

Figure 2b: 17 elements divided among 7 processes (second scheme)

Comparing block decompositions, we choose the second scheme because it has fewer operations in low and high indexes. The first scheme the larger blocks are held by the lowest-numbered tasks; in the second scheme the larger blocks are distributed among the tasks.

Block Decomposition Macros

C macros can be used in any of our parallel programs where a group of data items is distributed among a set of processors using block decomposition [5, 9].

```
#define BLOCK_LOW(id,p,n) ((i)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n) (BLOCK_LOW((id)+1)-BLOCK_LOW(id))
#define BLOCK_OWNER(index,p,n) (((p)*(index)+1)-1)/(n))
```

For macro *BLOCK_LOW*, the parameters; process rank *id*, number of processes *p*, and the number of elements *n*, expands to an expression whose value is the first, or lowest, index controlled by the process. Similarly, with the same parameters as macro *BLOCK_LOW*, macro *BLOCK_HIGH* expands to an expression whose value is the last, or highest, index controlled by the process.

Still with the same arguments, macro *BLOCK_SIZE* evaluates to the number of elements controlled by process id. Passed an array index, the number of processes, and the total number of array elements, macro *BLOCK_OWNER* evaluates to the rank of the process controlling that element of the array.

The four definitions that we just described above are the start of a set of utility macros and functions we can reference when constructing the parallel programs.

When decomposing an array into pieces distributed among a set of tasks, we need to differ between the local index of an array element and its global index. For example, we have 17 arrays elements to be distributed among three tasks. Each task is responsible for either four or five elements. Then the local indices range from 0 to either 3 or 4.

Ramifications of Block Decomposition

How does our block decomposition affect the implementation of the parallel algorithm? To answer this, first we recall the sieve that the largest prime numbers used to sieve integers up to n is $\sqrt{n}$. If the first process is responsible for integers through $\sqrt{n}$, then finding the next value of k requires no communications at all. It saves a reduction step. Let the first process has about n/p elements. If $(n/p) > \sqrt{n}$, then it will control all primes through $\sqrt{n}$. [5, 9]

In addition, it can speed the marking of cells representing multiples of k. Rather than check each array element to see if it represents an integer that is a multiple of k, it requires n/p module operations for each prime. The algorithm can find the first multiple of k and the mark that cell as well as cells $j+k$, $j+2k$, etc.., through the end of the block, for a total of about $(n/p)/k$ statements. In other words, it can use a loop similar to the one used in a sequential implementation of the algorithm, and it results much faster.

ANALYSIS OF PARALLEL SIEVE ALGORITHM

We modify the pseudocode from the previous one:

1. Create a list of natural numbers 2, 3, 4, 5, ….., n. None of which is marked. Each process creates its share of lists.
2. Set k to 2, the first unmarked number on the list. Each process does this.
3. Repeat: Each process marks its share of list
    a. Mark all multiples of k between k^2 and n.
    b. Find the smallest number greater than k that is unmarked. Set k to this new value
    c. Process 0 broadcasts k to rest of processes.

    Until $k^2 > n$.
4. The unmarked numbers are primes.
5. Reduction to determine number of primes

It gives:

$$\mathbf{X\,(n \ln \ln n)/p + (\sqrt{n} / \ln \sqrt{n})\,\lambda\,[\log p]}$$

Where X represents the time needed to mark a particular cell as being the multiple of a prime. This time includes not only the time needed to assign 1 to an element of the array, but also time needed for incrementing the loop index and testing for termination. $O(n \ln \ln n)$ is the sequential algorithm complexity [5, 9].

Since only a single data value is broadcast each iteration, the cost of each broadcast closely approximated by λ [log p], where λ is message latency. The approximation to the number of loop iterations is $\sqrt{n} / \ln \sqrt{n}$.

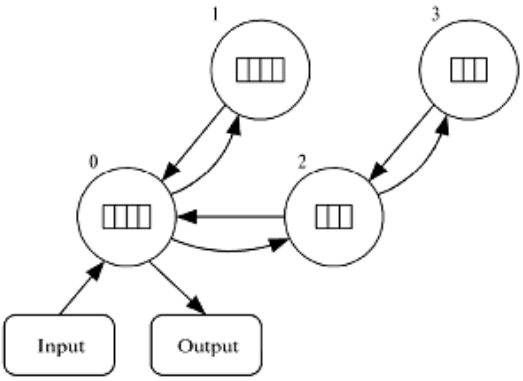

Figure 3: Task graph for the Sieve of Eratosthenes using Parallel Algorithm

Optimizations

There are few modifications to the program that can optimize the performance [5, 9].

a. Delete even integers

Since 2 is the only even prime number, so we take out 2 first. Then we change the sieve algorithm so that only odd integers are represent halves the amount of storage required and doubles the speed at which multiples of a particular prime are marked. If we change this, then the execution time of the sequential algorithm becomes:

$$\Theta(n \ln \ln n)/2$$

And the estimated execution time of the parallel algorithm would be:

$$\Theta(n \ln \ln n)/(2p) + (\sqrt{n} / \ln \sqrt{n}) \lambda \log p$$

By deleting even integers, we cut number of computations in half, and also free storage for larger values of n.

b. Each process finds own sieving primes

Suppose in addition to each task's set of about n/p integers, each task also has a separate array containing integers 3, 5, 7, …, $\sqrt{n}$. Before finding the primes from 3 to n, each task will use the sequential algorithm to find the primes from 3 through $\sqrt{n}$. Each task then has its own private copy of an array containing all primes between 3 and $\sqrt{n}$. Now the tasks can sieve their portions of the larger array without any broadcast steps. This is basically to replicate computation of primes to $\sqrt{n}$ and eliminate broadcast steps.

c. Reorganize loops

In this part, we want to increase cache hit rate by exchanging the inner and outer loops. If we recall the part that is the key of the algorithm a while ago, there are two large loops. The outer loop iterates over prime sieve values between 3 and $\sqrt{n}$. while the inner loop iterates over he process's share of the integers between 3 and n. We can fill the cache with a section of the larger sub array, and then strike all the

multiples of all the primes less than √n on that section before bringing in the next section of the sub array.

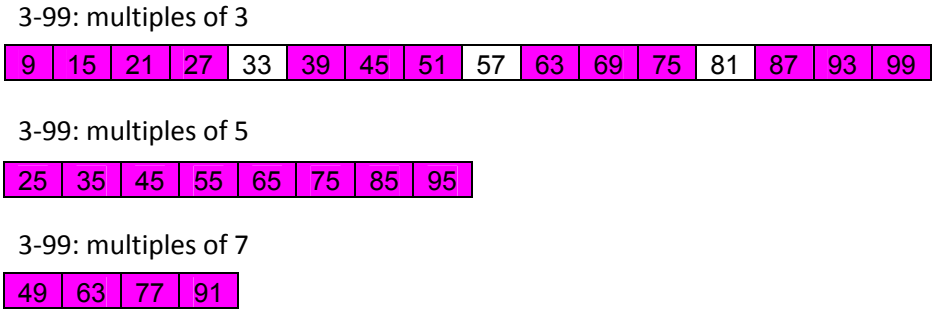

Figure 4a: Sieving all multiples of one prime before considering next prime

Figure 4a; For example, we would like to find prime numbers between 3 and 99, and suppose the cache has four lines, and each line can hold four bytes. One line contains bytes representing integers 3, 5, 7, and 9; the next line holds bytes representing 11, 13, 15, and 17; and so on.

Then we sieve all multiples of one prime before considering next prime. Shaded circles represent cache misses. By the time the algorithm returns to the bytes representing smaller integers, they are no longer in the cache.

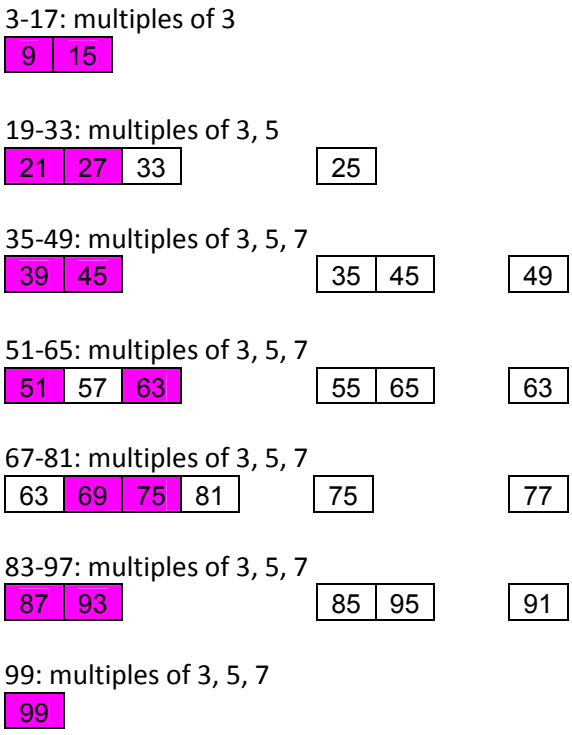

Figure 4b: Sieving multiples of all primes for 8 bytes

Figure 4b shows sieving multiples of all primes for 8 bytes in two cache lines before considering the next group of 8 bytes. Fewer shaded circles indicate the cache hit rate has improved.

SUMMARY

At beginning, we discussed a bit about the use of prime numbers, why it has become important in the computer world. We also introduced two sieves for finding large prime numbers; the Sieve of Eratosthenes and the Sieve of Atkin. We also have discussed about a sequential algorithm for the Sieve of Eratosthenes and used the domain decomposition methodology to identify parallelism.

In addition, we discussed the three improvements to the original parallel version. The first improvement is to eliminate even integers, since all prime numbers are odd except 2. We also cut the broadcast operations by making redundant the portion of the computation that determines the next prime. Then lastly, we improved the cache hit rate by striking all composite values for a single cache-full of integers before moving on to the next segment.

REFERENCES

[1] Atkin, A.O.L., Bernstein, D.J., *Prime sieves using binary quadratic forms*, Math. Comp. 73 (2004), 1023-1030.

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

[3] Hawkins, D., 1958. Mathematical sieves. *Scientific American* 199(December):105-112

[4] Hwang, S., Chung, K., Kim, D. Load Balanced Parallel Prime Number Generator with Sieve of Eratosthenes on Cluster Computers. IEEE 2007.

[5] Quinn, M.J. Parallel Programming in C with MPI and OpenMP. pg 115-134. McGraw-Hill Professional, 2004

[6] Wilkinson, B., Allen, M., Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Prentice Hall, 2005.

[7] Sorenson, J., Parberry, I. Two Fast Parallel Prime Number Sieves. Academy Press, Inc. 1994

[8] http://mathforum.org

[9] http://parlweb.parl.clemson.edu/~walt/ece493/ece493-4.pdf

[10] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

[11] http://www.cpaadvisor.us/sub/8_encryption.htm