PrimeTime: MultiThreading Techniques in Deterministic Primality Testing Schemas

Anurag Banerjee, Josh Randall

Abstract—We implement two deterministic primality testing algorithms, Trial Division and AKS Primality. In addition, we apply multithreaded techniques to our implementations, in order to see noticeable performance runtime gains in testing large numbers for primeness. We also find that as the number of threads increase, performance gains show diminishing returns. We provide future avenues of research in optimizing our algorithms and implementing probabilistic multithreaded primality testing for comparison metrics.

1 Introduction

PRIME number generation is a topic that is important for a variety of engineering and computer science disciplines. Most notably, prime number generation is the central tenet of the RSA encryption protocol for securing messages over the Internet. Generating prime numbers requires testing numbers for primality. The simplest algorithm to determine whether a number N is prime is to test whether its divisible by any number between 2 and the square root of N. This test is called Trial Division. However, as N becomes larger, this simple test scales poorly. Therefore, we look to multi-threading, among other techniques, in order to provide greater efficiency. Primality tests are usually judged on three principal metrics: determinism, polynomial-bound running time, and generality to arbitrary numbers. Determinism ensures that for a given number as an input, we can say with absolute certainty whether or not that number will be prime. Polynomial-bound running time is a crucial metric that speaks to the scalability of the primality test. Generality allows one to test any class of number without worry of incorrect results. We implement two principal primality testing schemes: Trial Division and AKS Primality. Both of these implementations satisfy the three metrics to varying degrees and employ multi-threading techniques in order to distribute the computational workload and generate runtime gains. Our paper is organized as follows: we first describe the theory behind the two algorithms with single-threaded run-time complexities, we then describe our parallel processing methodologies for both algorithms in terms of implementation, and finally, we present the preliminary results for various thread pool sizes and numbers n to test.

1.1 Related Work

Primality Testing implementations have spanned a large space of differing approaches and parallelization schemes. Within the follow section, we describe various scientific research efforts that implement similar systems to the ones that we have implemented. We will delineate the key differences of our implementations.

1.1.1 Trial Division

Trial division is a primality testing algorithm that has seen numerous implementations in academic research [10]. In addition, we see multi-threaded applications in regards to the work of Bokari [9]. In particular, Bokari differs in his parallel implementation, in that the shared synchronized counter for testing primality for crossing is explicitly told to wait until all other threads are a set distance away from itself before picking up a number. This is accomplished using an explicit locking mechanism, while ours uses a lock free atomiclong construct. In addition, slave threads are explicitly told to cross out multiples of 2,3, and 7 initially, while our algorithm only explicitly ignores multiples of 2.

1.1.2 AKS

In regards to AKS primality, the fundamental algorithmic bounds of the necessary computations have been explored in various papers. The AKS algorithm was first developed by Agrawal, Kayena, and Saxena [2]. Tighter bounds to the algorithm were discovered by Lenstra and Pomerance [3]. Jayashima provides a parallel MPI implementation of AKS that uses a master-slave model along with a bank of precomputed values for the polynomial congruence relation stage [8]. The master is responsible for maintaining end conditions for each slave thread while the slave threads blindly perform exponentiation and polynomial expansion operations. The point of critical progress condition is that if the master thread unexpectedly stalls, there would be no way for the slave threads to continue performing the arithmetic exponentiation tasks. Our implementation scheme chiefly differs, in that it uses a shared AtomicLong object in order for each thread to determine loop end conditions on its own. We believe this ensures that the algorithm completion would be lock-free in a nonblocking progress conditional sense.

1.2 Implementations

Within the following section, we describe detailed implementation schemes for both deterministic primality testing algorithms, Trial Division and AKS Primality. For each method, we first discuss the basic algorithm and timing implications from a single threaded standpoint, then we describe the synchronization techniques we use to involve multiple threads in the calculation.

1.2.1 Trial Division

The trial division algorithm is the most basic algorithm for determining whether or not a number is prime. With this method, we take the prime we are testing, n, and modulo with each prime number that is less than or equal to the square root of n. If any of the remainders are zero, then we know that the number is composite. To implement this technique, we keep a running list of prime numbers, first inserting two manually, then checking odd numbers starting from 3. For each number we check, x, we go through the current list of primes, and modulo by each one that is less than the square root of x. If we find that x is prime, we modulo n by x, to see if we have found a factor of the number we are interested. If x happens to be a factor of n, then we know that n is composite and we are done. Once we have tested for all the primes less than the square root of n, and have still not found a factor, then we know that n is prime [9]. While this method works well for small numbers, due to its simplicity, it is often not optimum for finding very high primes. We can bound the run-time of the trial division by looking at the limiting term of the harmonic series

Fig. 1. Prime harmonic bound

$$\sum_{\substack{p \text{ prime} \\ p \le n}} \frac{1}{p} \ge \log\log(n+1) - \log\frac{\pi^2}{6}$$

And evaluate that for calculating all primes below an arbitrary number n we find the bound to be [11]

$$O(n * loglog(n)) \tag{1}$$

In addition, we can bound the memory usage of the basic algorithm to be bounded by the number of bits in n and the memory usage to be calculated as

$$O(\sqrt{n} * loglog(n)/log(n)) \tag{2}$$

Combined together, we can take these results to mean that, although polynomial in running time, we would expect that as n scales up, the memory and run-time scales up proportional to greater than a linear scale.

1.2.2 Trial Division Multithreading

To allow threads to work on one trial division problem concurrently, they must share the list of primes so far, and there must be a way for them to mutually exclusively claim the next number to be tested, so that other threads do not test the same number. Also, when a thread claims a number to test, it must ensure that all the necessary primes are available by which it will need to modulo. In our implementation, the count is an odd atomic long. Threads getAndAdd 2 to get the number they will test. This ensures that no two threads work on the same number, and every odd number will get tested. When a thread wants to test a number, it will need to know every prime number less than the square root of its number. In order for such a thread to know that all required numbers have been tested so far, each threads adds their number to an active list of numbers that are currently being checked. Then, threads check whether any of the numbers in the active list are less than the square root of their number, in which case they must wait for that to be true. Because the numbers being checked only increase, the active list will include all numbers lower than the current number that are being checked. The thread will remove its number from the active list when it is done testing the number. Because the numbers being tested can only increase, if there are no numbers in the active list that this number is dependent on, then all required numbers have been tested. Now, the thread can actually test the number it has. It first copies the list of primes so far, because all the required primes must have been added to the primes list already, and iterating over a dynamic list that is changing causes problems in java. The thread goes through each prime in the list that is less than the square root of the current number, and checks the modulus of its number by that new number. If any numbers in the list are give a modulus of 0, then this number must be composite, and the thread moves on to another number after removing this number from the active list. If the thread makes it through the list of primes and has not found a factor, then this number is prime, and the thread adds it to the primes list. Then the thread checks whether the overall number, n, is divisible by this prime. If the current number is prime, but does not divide n, the thread removes its current number from the active list, and starts to test a new number. If the current number divides n, then n is composite, and the test is finished. Once the number checking count gets above sqrt(n), we know that n is a prime number.

1.2.3 AKS Primality

AKS Primality testing is an algorithm that displays the interesting properties of being general, polynomial, deterministic, and unconditional. We first describe the mathematical proof of logic for the AKS algorithm, and then show that under appropriate conditions, any arbitrary number can be deterministically proven to be either prime or composite in a polynomial bound.

$$O(log^6(n)) \tag{3}$$

It is vital that we try and maintain this logarithmic running time in order to show an eventual improvement versus the trial division primality testing scheme. Before introducing the algorithm, we must introduce a few key mathematical congruence relations. First, we introduce Fermats Little Theorem that states that for a prime number p, and any integer a,

$$a^p = a \pmod p. \tag{4}$$

For AKS, we must use an extension that generalizes to polynomial functions which states that for a given number n to be prime,

$$(x-a)^n = (x^n - a) \pmod n \tag{5}$$

For appropriately chosen integers a such that a is coprime with n. In the relation above, x is a free variable that must be evaluated symbolically. In order to reduce the computational time used to verify this congruence relation, we employ an additional modification such that

$$(x-a)^n = (x^n - a) \pmod{n, x^r - 1} \tag{6}$$

where the modulo operation applies to both arguments to both sides of the equation and r is chosen in a prior step of the algorithm such that it is the smallest integer that satisfies

$$O_r(n) > log^2(n) \tag{7}$$

where the left term is the multiplicative order of n modulo r. If this congruence relation fails for any value of a within the bounds proportional to r, then we can declare the number to be composite.

Having described the underlying theorems behind the algorithm, we now detail the step-by-step implementation at a software level. Given an arbitrary input number n to test for primeness, we check for compositeness initially by determining whether

$$n = a^b \tag{8}$$

for any integer *a,b* such that a,b¿1.

This step can be verified in essentially linear time and is bounded by

$$log_2(n) * exp(O(\sqrt{((loglogn) * (logloglogn))})) \tag{9}$$

[1]

Next, we evaluate an r such that the above conditions are satisfied. Afterwards, we perform another weak test for compositeness such that if

$$1 < gcd(a, n) < n \tag{10}$$

for some a less than r is satisfied, we can say that n is composite. However, if n is less than r then we can say n is prime.

Now, we can verify the fermats little theorem extension as described above for all integers, a, such that

$$1 <= a <= \sqrt{(\varphi(r) * log(n))} \tag{11}$$

[3]

If the modification to fermat's little theorem does not hold for any such a as described above, then, at this stage in the algorithm, we can deterministically say that n must be composite. Otherwise, n is prime, and the algorithm is complete.

Within the algorithm implementation, the first steps, (checking for perfect powers and determining a lower bound for integer r) are all relatively trivial in terms of computational scaling for larger values of n. The last step, evaluating the polynomial congruence relation, however, requires computation of the nth order polynomial.

$$(x + a)^n \tag{12}$$

This requires maintaining the binomial coefficients from the nth row of Pascals triangle and decreasing powers of a to the n. This requires computing (n+1) coefficients for each unique a tested. The combination of these operations results in a polynomial bound of

$$O(d * log^2(n)) \tag{13}$$

for each unique a tested. And with the entire recomposition of the polynomial construction, we find that the performance is dominated by

$$\sqrt{((d/3))} * log(n)/log(2) \tag{14}$$

approximately where d is the degree of the polynomial remainder [3]. We get this bound by precomputing the binomial coefficients, then only computing the unique powers of a over each iteration of the loop. These operations are the focus of where our parallelization gains should occur.

1.2.4 AKS Primality Multithreading

We describe a parallelization structure that divides up the work into generating binomial coefficients and generating powers of a up to n where the coefficient computations and power computations are multithreaded using a lock-free Atomic Reference that allows threads to dynamically perform new computations as they complete. We decide to forgo any multithreading aspects in the initial computational steps prior to the nth order binomial generation because these steps take an increasingly small fraction of the program time as the integer to be tested for primeness n approaches infinity. The bulk of our multithreaded implementation occurs in generating binomial expansions for different constants a. We first divide up this task by making use of the following identities

$$(x+a)^n = a^n * c_0 * x^0 + (a^{n-1}) * c_1 * x + ... + a^0 * c_n * x^n \quad (15)$$

where the coefficient c denotes the ith term of the binomial expansion.

Based on this relation, we can divide up the task of generating the polynomial into generating the binomial coefficients c and generating the powers of a separately.

By doing so, for a given number n to test, although we are computing a unique binomial for each iteration of the for loop, we can avoid having to recompute the c's and focus on generating powers of a and then combining the terms efficiently. For any number of threads, we must first generate the nth row of pascals triangle which contains n+1 binomial coefficients. Each term can be computed, in closed form expression by

$$(n!/((n-k)!(k)!)) \quad (16)$$

We divide up the task by having each thread use a shared AtomicLong index which they systematically try to getAndIncrement in a lock-free fashion. This ensures that regardless of the computational time distribution for different jobs, threads will divide up these portions up dynamically. Each thread will use a temporary local copy of their own index that is guaranteed to be unique from any other threads current index and compute the binomial coefficient at for the ith binomial element.

We use a similar strategy to divide up the power computations. The power computations must be recomputed for each iteration of the loop because we are testing different values of a, while the binomial computations can be computed once and they are reused for the remainder of the primality test [6].

1.3 Results

In the following section, we describe various results for runtime execution times with various prime numbers to be tested, and different numbers of threads executing the associated primality testing algorithms. We are interested in the timing when checking actual prime numbers, because checking composite numbers causes the tests to finish early. We conducted five runs for each data point, and averaged the runtimes. We first look at the Trial Division results for increasing numbers and increasing threads, then we describe AKS Primality results for increasing numbers and increasing threads.

1.3.1 *Trial Division*

We first view the performance of the algorithm for increasing n. For trial division, we would expect the scaling for runtime to be proportional to a super log function as n increases. Based on the results, we see exactly this. We expect there to be a speedup with increasing threads and we can clearly see a pronounced downward shift in execution time between one thread and multiple threads for Trial Division Primality testing. However, as the number of threads increase, the results show diminishing returns on performance. We note the congested data for lower numbers of n in the trial division algorithm. We postulate that this congestion comes from the fact that lower numbers have much faster computational prime checks, therefore we theorize that higher contention between threads is occurring resulting in unclear gains. However, as n scales up for sufficiently large numbers, computations become more time consuming and therefore contention decreases. Indeed, the higher number tests show smoother curves than the lower number tests.

Fig. 2. Trial Division by Number

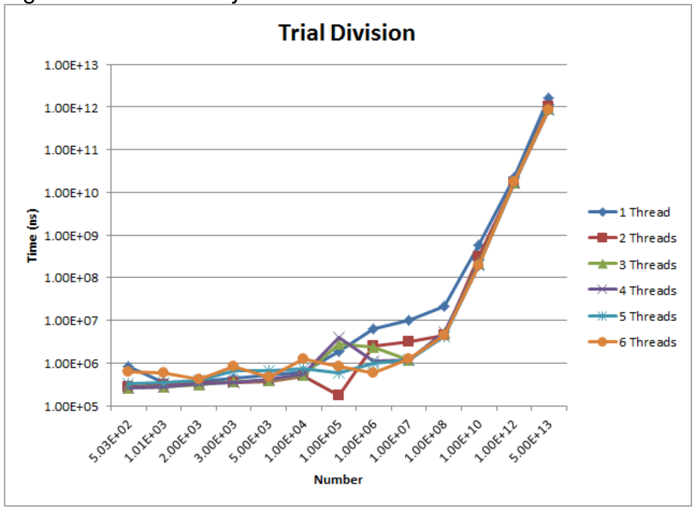


Fig. 3. Trial Division by Thread

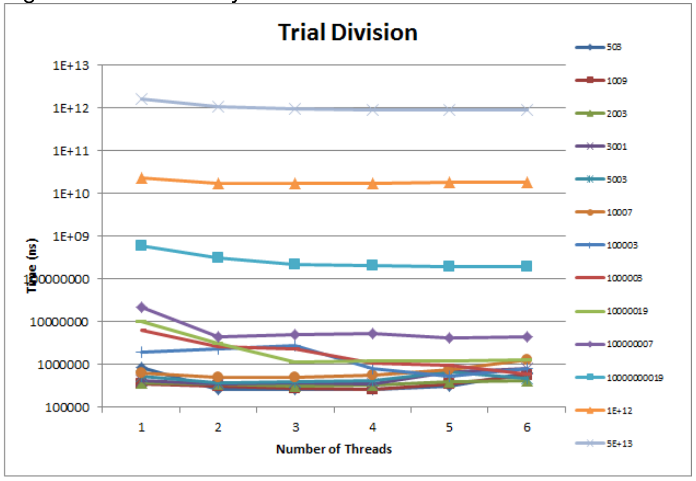
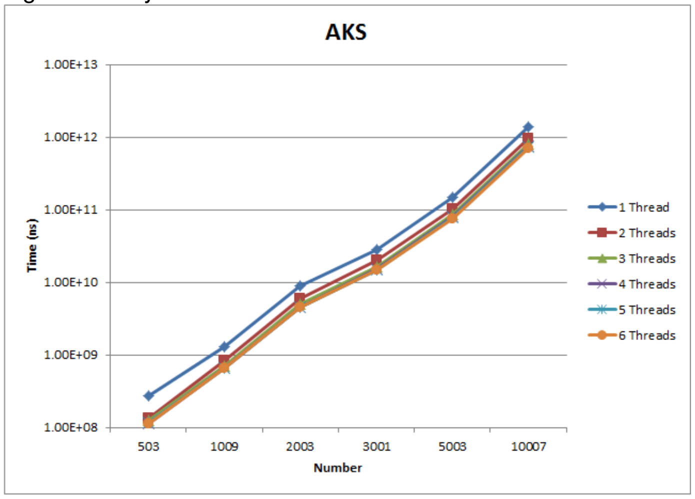


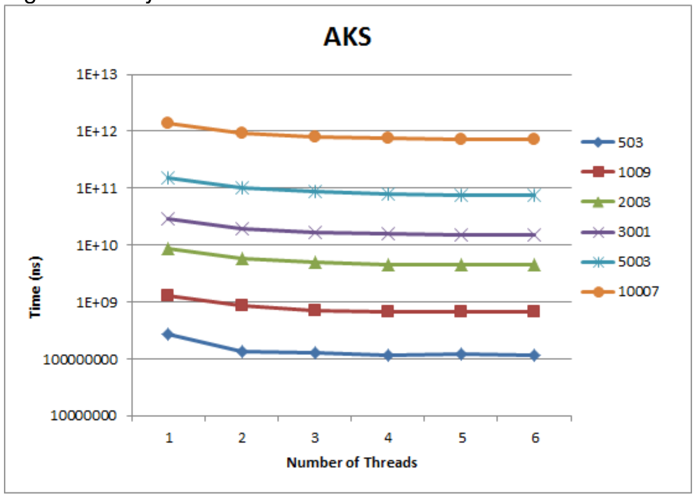
Fig. 4. AKS by Number



1.3.2 AKS Primality

We now describe the evaluated performances for AKS primality testing while independently varying the number to be evaluated for primeness and the number of threads to evaluate a constant prime number. As we claimed before, the algorithmic complexity of AKS primality should be proportional to log as n increases. Based on the scaling of the graph, we see that this holds up, and therefore we can project the AKS algorithm for increasing n to eventually outperform the trial division algorithm, while it takes much longer than Trial Division at these lower numbers. We expected there to be a speedup with increasing threads and we can clearly see a pronounced downward shift in execution time between one thread and multiple threads for AKS primality testing. However, as the number of threads increase, the results show diminishing returns on performance. We suspect this may be occurring due to the scheduler of the testing platform. In addition, we suspect that contention may occur within the binomial expansion stage.

Fig. 5. AKS by Thread



1.4 Future Work

Several avenues exist for us to examine additional primality test implementations and optimizations. The key goal for us has always been to provide performance metrics for varying thread numbers and increasingly large values of N to show throughput gains with multi-threading. However, our implementation is lacking in that it only demonstrates two different deterministic primality tests, when in fact, there are more deterministic primality testing schemes, as well as an entirely different class of primality testing algorithms that are probabilistic in nature. A probabilistic primality testing scheme is based on evaluating slightly looser congruence relations in order to determine primeness with a certain probability, as opposed to a deterministic test, which provides the certainty of the results. In particular, the Miller-Rabin test is a probabilistic primality test that could, in principle, be multithreaded, and could also could be adapted to provide a deterministic primality test by using the method in [12]

For future improvements, we would be

interested in providing multithreaded implementations of Miller-Rabin in order to compare to our trial division and AKS Primality tests. Within our own implementation of the AKS primality test, the generation of the binomial coefficients of the nth order polynomial takes a large portion of the algorithm runtime. However, Bernstein provides an alternative approach that would remove the need for binomial generation and instead relies on exponentiation of integers with appropriately chosen exponentiation functions. The exponentiation function is based on calculating an appropriate k such that

$$k = log^2(r * n) \tag{17}$$

and mapping the free variable x to

$$x- > 2^k \tag{18}$$

then applying an inverse relation to the new found integer. This algorithm would rely less on the closed-form binomial coefficient expression generation, and potentially speed up the AKS implementation.

2 Conclusion

Although deterministic primality testing algorithms using Trial Division runs in super linear time, and it becomes unsustainable to use for sufficiently large numbers due to the memory usage requirements and complexity of scaled computations. AKS improves upon the scalability of Trial Division, by exhibiting logarithmic time complexity, but takes a significant amount of time, even for smaller numbers. Therefore, probabilistic algorithms generally are used over deterministic algorithms. However, much research has gone into parallelizing deterministic algorithms due to the fact that probabilistic algorithms will always fail for certain subclasses of numbers, such as Carmichael numbers. Our implementations were comparable to many existing single threaded deterministic primality testing suites. In addition, our implementations were able to overcome a variety of performance impediments for multithreading, by maintaining lock-free synchronization structures and efficient reuse of polynomial computations. For trial division, the largest component of the implementation development cycle came from providing a safe, lock-free synchronization scheme for computing the primeness of numbers. This was crucial in order for our multithreading implementation to show noticeable performance gains. For the AKS Primality testing algorithm, a significant amount of work came from efficiently generating binomial expansions for nth order polynomials. Because this is the dominating step of the algorithm, a significant portion of time was spent in order to parallelize these computations as much as possible. Based on these efforts, we found noticeable performance gains in multithreaded tests for our algorithms, with diminishing returns as the number of threads increased. We also found that our implementations matched with the theoretical runtime complexities, and thus scaled in an expected manner with respect to higher numbers. Future work in this area of research may come from exploring different primality testing algorithms for parallelization comparison metrics. In addition, optimizations can be made to our existing implementations by incorporating new research in theoretical bounds for computations and different methods for performing various computations within the algorithms themselves.

References

[1] [Bernstein 1998] D. J. Bernstein, Detecting perfect powers in essentially linear time, Math. Comp. 67:223 (1998), 12531283.

[2] [Agrawal 2004] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. Annals of Mathematics, 160(2):781793, 2004.

[3] [Lenstra 2005] H. W. Lenstra, Jr. and C. Pomerance. Primality testing with Gaussian periods. Preprint. Available as http://www.math.dartmouth.edu/ carlp/PDF/complexity072805.pdf, July 2005.

[4] Torres, M., Gold, A. and J. Barrera A parallel algorithm for numerating combinations. The 2003 International Conference on Parallel Processing Proceedings. (IEEE Computer Society Press). Taiwan, pp 1:581-588 (2003).

[5] R G Salembier and Paul Southerington. An implementation of the aks primality test. Computer Engineering, 2005.

[6] Cao, Zhengjun. A Note On the Storage Requirement for AKS Primality Testing Algorithm. IACR Cryptology ePrint Archive 2013 (2013): 449.

[7] A. Weimerskirch and C. Paar, Generalizations of the Karatsuba Algorithm for Efficient Implementations, http://www.crypto. ruhr-uni-bochum.de/imperia/md/content/texte/kaweb.pdf, 2003.

[8] Jayasimha, T. MPI Implmentation of AKS Algorithm.

[9] Bokhari, Shahid H. ”Multiprocessing the sieve of Eratosthenes.” Computer 20.4 (1987): 50-58.

[10] Hoare, C. A. R. ”Proof of a structured program:the sieve of Eratosthenes.” The Computer Journal 15.4 (1972): 321-325.

[11] Pritchard, Paul, ”Linear prime-number sieves: a family tree,” Sci. Comput. Programming 9:1 (1987), pp. 1735.

[12] Schoof, Ren (2004), ”Four primality testing algorithms” , Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, Cambridge University Press, ISBN 0-521-80854-5