A Comparative Study Between Primallity Testing Algorithms.

By,Tamis van der Laan & Author Samuel Austin

Abstract—**In this paper we compare three one sided Monte Carlo primality testing algorithms, namely the Fermats, Miller-Rabin and Solovay-Strassen primality test algorithms and two deterministic primality testing algorithms, namely the Trial division algorithm and the Agrawal-Kayal-Saxena primality testing algorithms. We will look at theoretical runtime complexity as well as practical run time complexity, Theoretical accuracy and practical accuracy. We will also look at the practicality of parallelizing the algorithms and look at their gain in runtime using our setup.**

I. Introduction

A. Fermat's Primality Test

Fermat's primality test is quite simple. It makes use of fermats little theorem [1] which is stated as followed:

$$a^{p-1} \equiv 1 \mod p \tag{1}$$

This theorem holds for any $1 \leq a < p$ when p is a prime number and a and integer. When a is composite at least half of the integer bases a will fail the test with the exception of so called Carmichael numbers. These Carmichael numbers will pass the test for most of the integer basses a. Although the test is flawed Carmichael numbers are rare and thus the test is still used in practical applications. This theorem is utilized in practice by generating k random integer bases and testing if the theorem holds. If one fails the test the number is composite and if all pass the number is prime. The accuracy of the fermat primality test given that n is not a Carmichael number is 2^{-k} where k is the number of tests performed. The run time complexity of Fermats Primality Test is $O(k \cdot log^2(n))$.

```
Input: n,k
Output: n is prime or composite
for $i \in \{1, ..., k\}$ do
    $r \longleftarrow$ random(1,n-1) ;
    if $r^{n-1} \mod n \neq 1$ then
        return Composite;
    end
end
return Prime;
```

Algorithm 1: Fermat's Primality Test

B. Solovay-Strassen Primality Test

The Solovay Strassen primality test [5] is built upon the ideas of the original fermats primality test. The Solovay-Strassen primality test indeed takes a similar approach where instead of fermats little theorem the following theorem discovered by euler is used:

$$a^{(p-1)/2} \equiv \left(\tfrac{a}{b}\right) \mod p \tag{2}$$

Were (a/b) is called the jacobi symbol. This theorem will always hold for any base and prime number p. In contrast to the fermat primality test for any composite number at least half of the integer bases pass the test stated above. The accuracy of the Solovay-Strassen primality test is therefore 2^{-k} where k is the number of random bases tested. In principle the fermats primality test has the same accuracy but it suffers from the Carmichael numbers which are composite yet will pass the fermats little theorem for most bases (far more than half). The run time complexity of the Solovay-Strassen Primality test is $O(k \cdot log^3(n))$.

```
Input: n,k
Output: n is prime or composite
if $n \neq 2 \wedge n \mod 2 = 0 \vee n < 2$ then
    return Composite;
end
for $i \in \{1, ..., k\}$ do
    $r \longleftarrow$ random(1,n-1) ;
    $jacobian \longleftarrow (n + jacobi(r, n)) \mod n$ ;
    $m \longleftarrow r^{(n-1)/2}$ ;
    if $jacobian = 0 \vee m \neq jacobian$ then
        return Composite;
    end
end
return Prime;
```

Algorithm 2: Solovay-Strassen Primality Test

C. Miller-Rabin Primality Test

The Miller-Rabin test [4] is based on the Fermat test, just like the Solovay-Strassen test, but also includes elements of Euclids lemma. Euclids lemma is based on the following:
sif $x^2 \equiv 1 \mod p$ then $(x-1)(x+1) \equiv 0 \mod p$

This means that prime p can divide the product $(x-1)(x+1)$. Euclids lemma states that prime p divides one of the factors $(x-1)$ or $(x+1)$. This implies that x is congruent to $1 \mod p$ or $(-1 \mod p)$.

Based on this, the following must hold for any prime number $p > 2$: $p-1$ must be an even number and can be written as $2^s \cdot d$ where s and d are both positive and d is an odd number. And for each a1,..,p-1the following must hold: $a^d \equiv 1 \mod p \vee a^{2^r d} \equiv -1 \mod p$ where $r \in \{0, ..., s-1\}$.

The Miller-Rabin test then uses Fermats little theorem, $a^{p-1} \equiv 1 \mod p$, in combination with the above

equalities in the following way:
If you keep taking square roots of a^{n-1} and n is prime then you should end up with either
$a^{2^r d} \equiv -1 \mod p$ where $r \in \{0, ..., s-1\}$ or if all powers of 2 have been removed $a^d \equiv 1 \mod p$

If neither of these cases are reached, n must be composite, otherwise n is probably prime. The running time of the

```
Input: n,k
Output: n is prime or composite
$s \longleftarrow n-1$ ;
while $s \mod 2 = 0$ do
    $s \longleftarrow \frac{s}{2}$;
end
for $i \in \{1,...,k\}$ do
    $r \longleftarrow$ random(1,n-1) ;
    $t \longleftarrow s$;
    $m \longleftarrow a^t \mod n$ ;
    while $t \neq n-1 \wedge m \neq 1 \wedge m \neq n-1$ do
        $m \longleftarrow m^2 \mod n$ ;
        $t \longleftarrow 2t$
    end
    if $m \neq n-1 \wedge t \mod 2 = 0$ then
        return Composite;
    end
end
return Prime;
```

Algorithm 3: Miller-Rabin Primality Test

Miller-Rabin Primality Test is $O(k \cdot log^3(n))$. The running time of the Miller-Rabin Primality Test is $O(k \cdot log^3(n))$. At least three out of four bases a fail the primality test when n is a composite number. This means that the chance of MR declaring a composite number as prime is equal to 4^{-k}.

D. Trial Division Primality Test

Trial Division is the most trivial and straightforward deterministic primality test one can imagine. It simply exploits the fact that a prime number can only be divided with itself and 1. There is no need to test numbers beyond the square root of n and thus the worst case running time becomes $O(\sqrt{n})$.

```
Input: n
Output: n is prime or composite
for $i \in \{2,...,\sqrt{n}\}$ do
    if $n \mod i = 0$ then
        return Composite;
    end
end
return Prime;
```

Algorithm 4: Trial Division Primality Test

E. Agrawal-Kayal-Saxena Primality Test

The Agrawal-Kayal-Saxena primality test [?] is based on a generalization to polynomials of the Fermat's little theorem. The following congruence relation is used to test primality.

$$(x-a)^n \equiv (x^n - a) \mod n \tag{3}$$

Were the the congruent relation holds for all integers a coprime to n. Note that x is not specified but is an variable and that in order to test this congruence relation the first polynomial must be expanded and the coefficients must be compared. This expansion and comparison takes exponential time and therefore the algorithm makes use of the following related congruence relationship:

$$(x-a)^n \equiv (x-a) \mod (n, x^r - 1) \tag{4}$$

this is the same as:

$$(x-a)^n - (x^n - a) \equiv nf + (x^r - 1)g \tag{5}$$

Where f and g are some polynomials. This congruence can be checked in polynomial time.

```
Input: n
Output: n is prime or composite
for $i \in \{2,...,\sqrt{n}\}$ do
    if $\log_i(b) = \lfloor \log_i(b) \rceil \wedge \log_i(b) > 1$ then
        return Composite;
    end
end
for $r \in \{2,...,n-1\}$ do
    if $Multiplicative_Order(r,n) > \log_2(n)^2$ then
        break;
    end
end
for $a \in \{0,...,r\}$ do
    if $1 < gcd(a,n) \wedge gcd(a,n) < n$ then
        return Composite;
    end
end
if $n \leq r$ then
    return Prime;
end
for $a \in \{1,..., \lfloor \sqrt(Totient(r)) \log(n) \rfloor\}$ do
    if $x^n \neq (x-a)^n + a$ then
        return Composite;
    end
end
return Prime;
```

Algorithm 5: Agrawal-Kayal-Saxena Primality Test

II. Accuracy Comparison

Although the theoretical accuracies are known we tested the accuracy of the random algorithms in practice as they may deviate from the theoretical accuracy which are only upper bounds. Our first test will involve testing the first

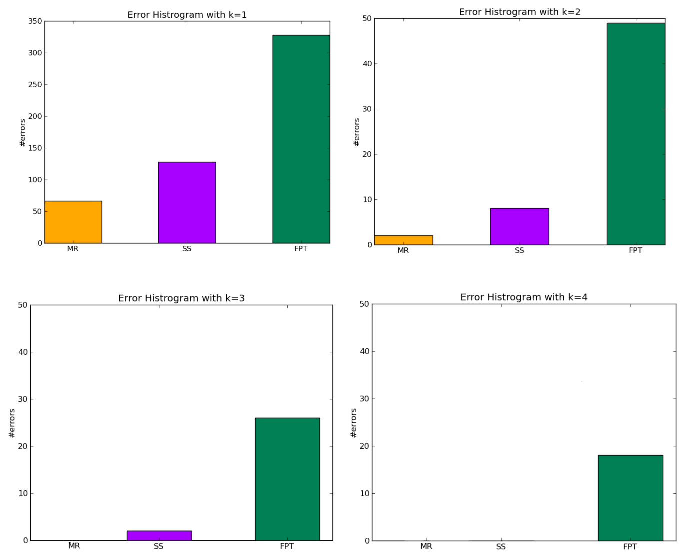

Fig. 1

500000 numbers for different number of tests k. We detect the errors by also running the deterministic Trial Division Primality Test. Under the assumption of the theoretic bounds we would expect to see 500000/2=250000 and 500000/4 =125000 errors when $k = 1$. The histograms in figure 1 show the results for $k = 1, 2, 3$. We see that the algorithms perform significantly better than expected for $k = 1$. Namely we make far viewer errors in practice than in theory. It is also clear that although the theoretical accuracy bounds of the Solovay-Strassen and the Fermat primality test are the same (except for the carmichael numbers) the Solovay-Strassen performs significantly better. We do see that the miller-rabin test is twice as accurate as the Solovay-Strassen test. This trend persists when k grows. The Fermat Primality Test seems to half for k up to 3 but then gets stuck when $k > 3$ most probably due to the remaining Carmichael numbers. There are actually 12 carmichael numbers below 50000. This can also be seen from the shaky character in the plot in Figure 2 which illustrates the number of errors with increase of k.

From Figure 2 we see that the Fermat Primality Test has a linear character in the beginning. But when k is increased further there error starts to shake and goes to zero further out then one would have expect when only looking at linear behavior at the beginning. This behavior stems from the Carmichael numbers who are persistent and require far more iterations to be correctly labeled as composite.

In theory the accuracy is independent of n, that is the magnitude of the number being tested. We tested this in practice by comparing accuracy in two different ranges, namely from 1 to 200000 and from 200000 to 400000, both results can be seen in the plot as a complete lines and striped lines respectively. We see that indeed small or large n numbers perform approximately the same. There is a small discrepancy but this is due to the fact that the density of primes decreases in the order of $\frac{1}{\log(n)}$ (prime number theorem).

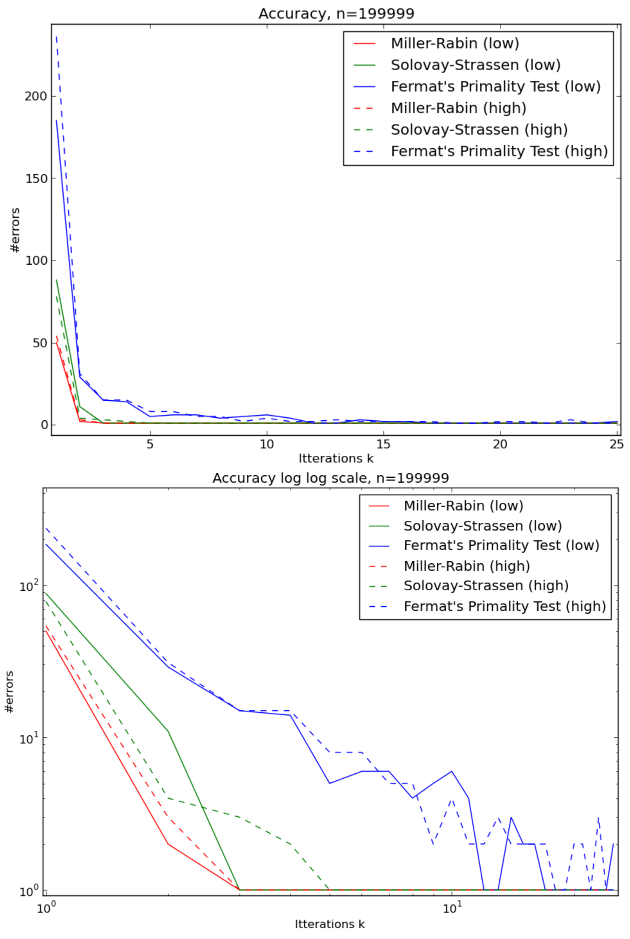

Fig. 2

III. Run Time Comparisons

We test the performance of both the Monte Carlo Algorithms as well as the Deterministic Algorithms both in a serial and parallel fashion. For this we used the following computing setup:

Processor	Intel Core i7-3630QM 2.4 GHz (8 cores)
Memory:	7.7 GB DDR3
OS:	Linux Mint 16 (Petra)

The algorithms have been implemented in C++ and we use the Open Multi Processor library [3] in order to parallelize our code. For modulo exponentiation we used the GNU Multiple Precision Library [2] as these exponentiations easily get very large.

A. Monte Carlo Primality Tests

Next to the accuracy of the randomized primality tests, it is also useful to know the run time in order to determine which algorithm is most viable. For these tests we based the value k, the number of iterations run within each algorithm, on the theoretical accuracy of each primality

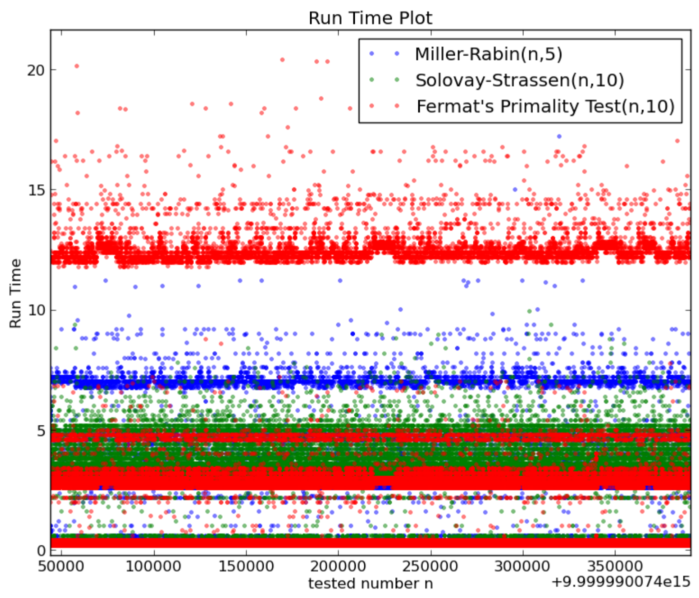

Fig. 3

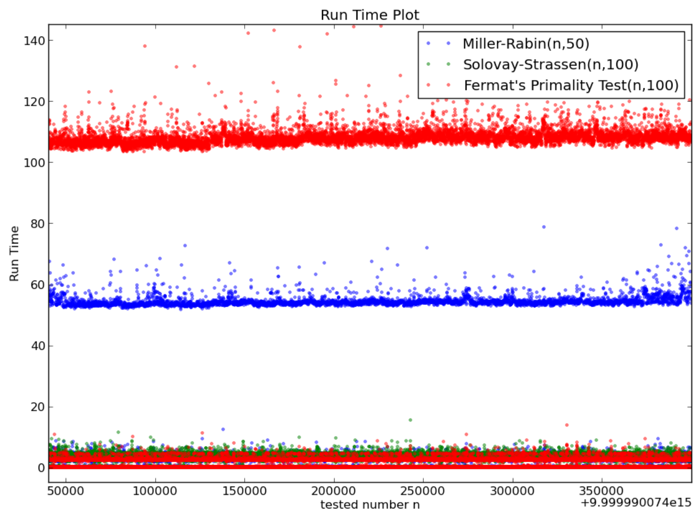

Fig. 4

test. As Miller-Rabin is twice as accurate compared to the other two we only run half the number of times.

As the speed of the randomized algorithms is quite high we used large numbers to generate the run time data. We used a range of 360000 numbers starting at 9999990074039999. For the first test we set the value k as 5 for the Miller-Rabin test and 10 for the others. Each primality test for each algorithm was run 5 times and the average run time was used to plot the graph. From figure 3 we can see that the Fermat test is the slowest of the three randomized algorithms. The Solovay-Strassen test is the fastest, even though it runs twice the number of iterations compared to the Miller-Rabin test. We also checked what the results would be if a large number of iterations was used (figure 4). We set the Miller-Rabin test to do 50 iteration and the others to 100 iterations. From the graph we can see that the results are the same,

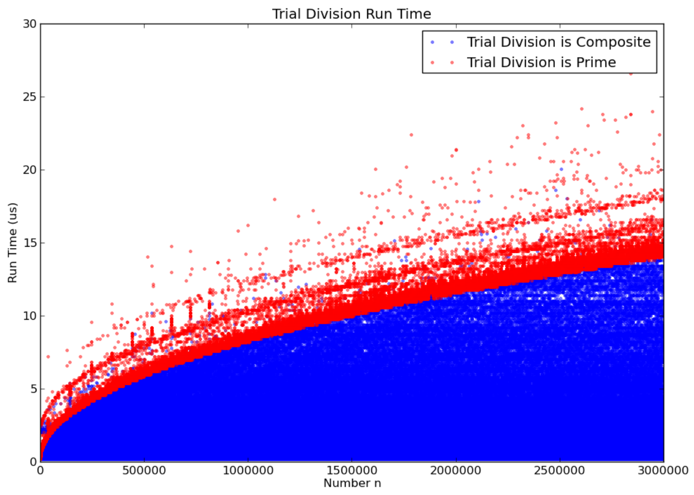

Fig. 5

but spread out further.

Based in the accuracy of the Miller-Rabin and Solovay-Strassen tests, there is no practical reason to use such a large number of iterations. However, the Solovay-Strassen is the fastest algorithm in both cases and therefore seems to be the most viable of the three.

B. *Deterministic Primality Tests*

B.1 Trial Division Primality Test

The first deterministic primality test we tested is the Trial Division Algorithm which has a theoretical runtime complexity of $O(\sqrt{n})$. We ran the algorithm from 1 up to 3000000 testing primality and recording the its runtime. The run time is an average of 5 independent runs in order to counteract cpu fluctuations which result in noise. The plotted results can seen below. We clearly see the square root behavior in the plot closely reflecting the theoretical runtime complexity in figure 5. The prime numbers are plotted in red while the composite numbers are plotted in blue. The prime numbers ride on top of the composite numbers as they form the worst case. The composite blue numbers seem to be approximately uniformly distributed between zero and the worst case.

B.2 Agrawal-Kayal-Saxena Primality Test

The second deterministic primality we tested was the Agrawal-Kayal-Saxena primality test. This test has a theoretic runtime complexity of $O(\log^{12}(n))$. The plot below shows the practical run time of the primality test. In figure 6 we see that the plot splits into two parts, the lower part seems to consist of the composite and even numbers while the upper part consists of prime numbers. The upper part clearly shows the logarithmic behavior of the algorithms also present in the theoretical time complexity. For this plot we took an average of 10 runs to compute the run time for each number. We see that never the less outliers persist above the plot. We see in figure 7 that for larger n the runtime levels off more to a constant value

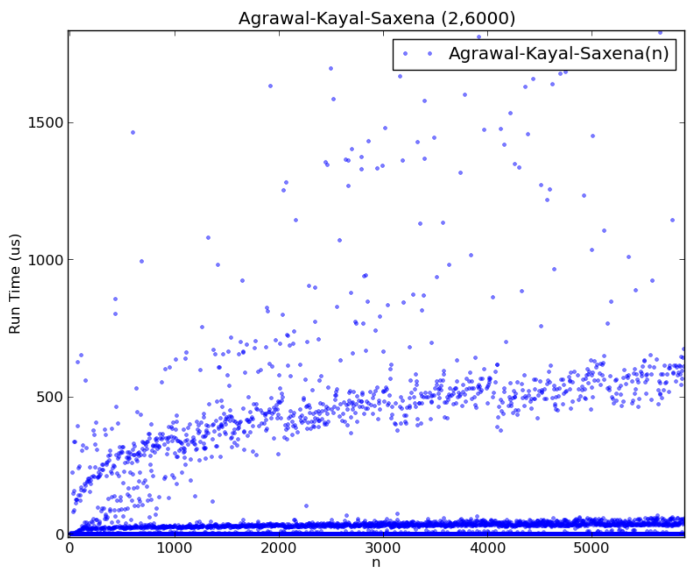

Fig. 6

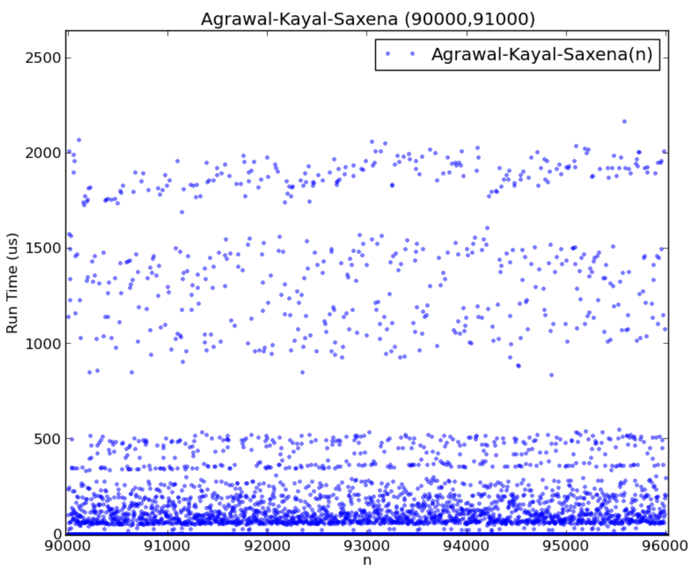

Fig. 7

although it will keep growing towards infinity when n goes to infinity. This again highlights the logarithmic nature of the algorithm.

B.3 Trial Division vs Agrawal-Kayal-Saxena

The theoretical runtime complexity of the AKS and Trial division algorithms are O(log12(n)) and $O(\sqrt(n))$ respectively. We see that the AKS algorithm is far slower than the trial division algorithm for both small and large numbers. This is an unexpected results as the AKS algorithm proves that Primes is in P we were under the impression that it would perform better than previously known deterministic algorithms for finding prime numbers.

It is possible the numbers we tested were not large enough to reach the point at which trial division surpasses AKS in run time. Unfortunately, the data structures

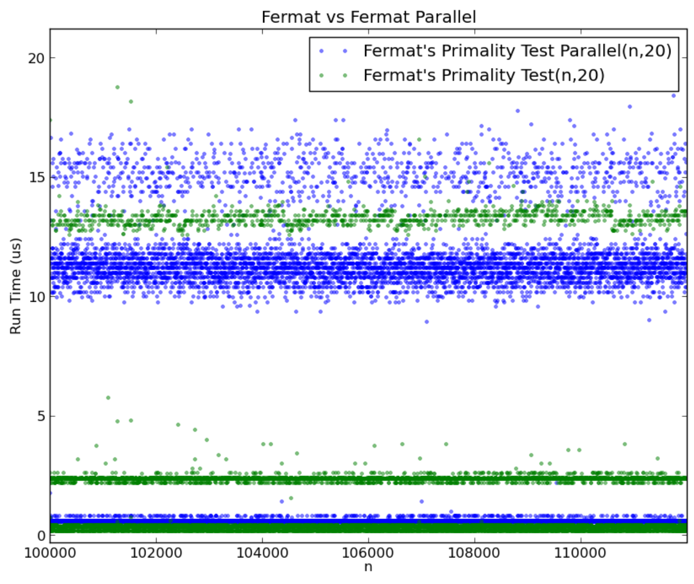

Fig. 8

we used did not allow us to test larger numbers. On top of that, the run time of both algorithm would be impractically large at that point.

IV. Parallel Monte Carlo Primality Tests

We experimented with the parallelization of the randomized algorithms to see if it would give any significant performance gain. For each algorithm we parallelized the iterations done within each algorithm as the operations within the iterations are independent of each other. For the measurements we used a range of numbers from 100000 to 120000.

.4 Fermat

In the Fermat test, with the number of iteration set to 20, we can see that the difference in run time is quite insignificant (Figure 8) . This is mainly due to the overhead of creating threads which doesnt weigh up against the time gained by running the iterations in parallel. In figure 9 we can see that running the iterations in parallel only becomes advantageous when large values of k are used. In this case we set k to 5000.

.5 Miller-Rabin

The results for the Miller-Rabin test are similar to the Fermat test. With a low number of iterations the difference between a parallelized implementation and a regular implementation are negligible (Figure 10). Running the Miller-Rabin test in parallel only becomes viable when using a high number of iterations (figure 11).

.6 Solovay-Strassen

In the case of the Solovay-Strassen test we see that a parallel version does slightly better than the regular implementation (Figure 12).

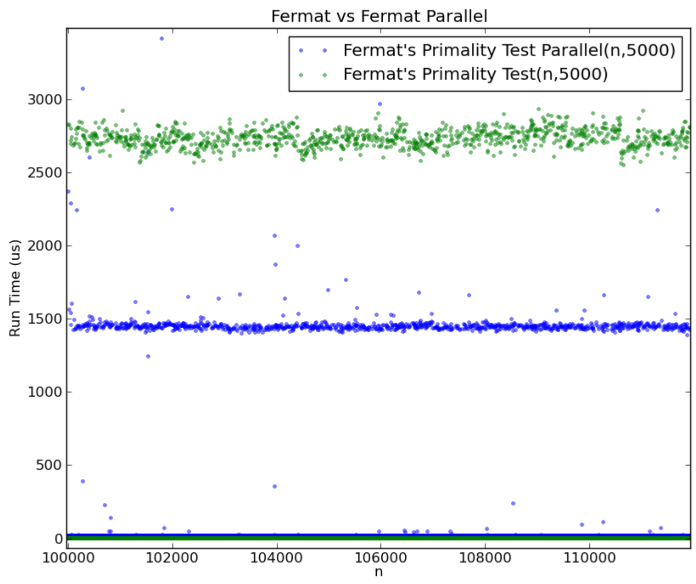

Fig. 9

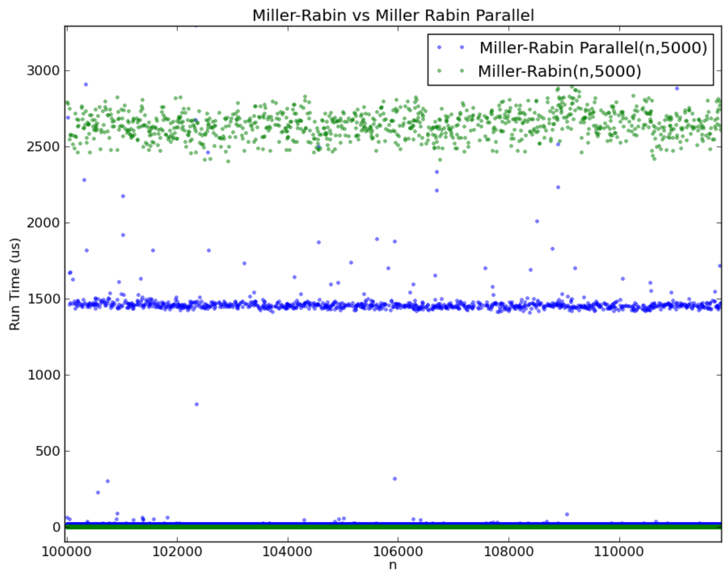

Fig. 11

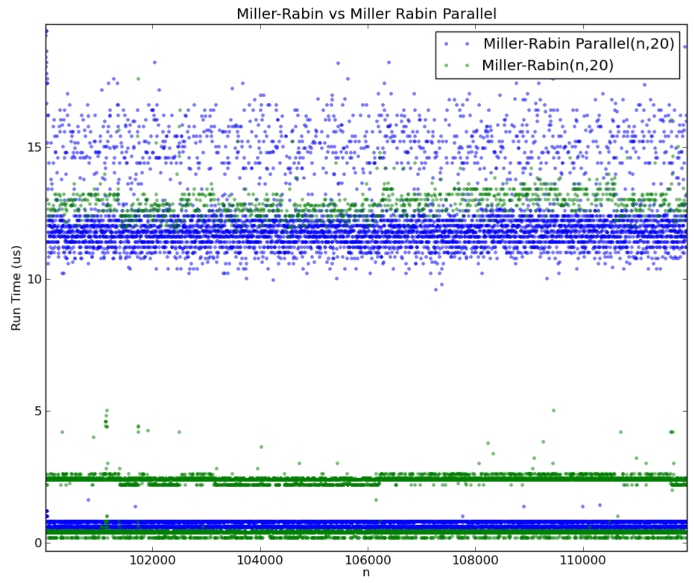

Fig. 10

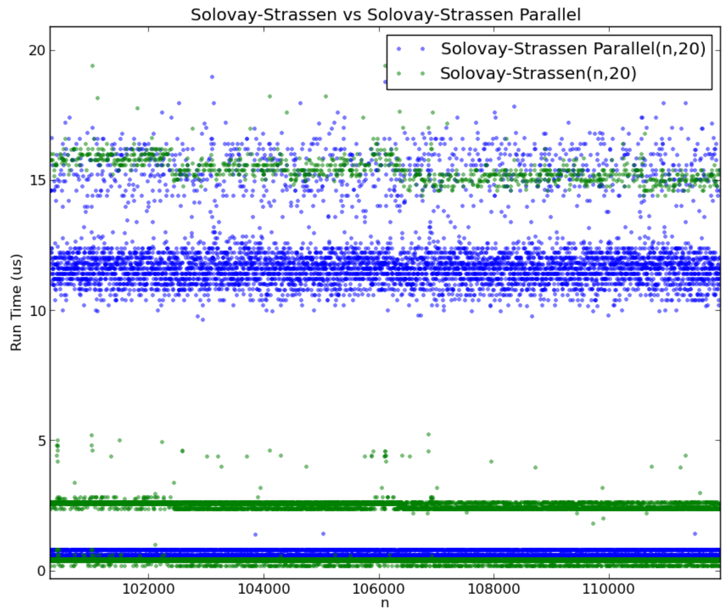

Fig. 12

.7 Solovay-Strassen

Once again, increasing the number of iteration causes a larger separation between the parallelized version and the regular implementation (Figure 13).

.8 Solovay-Strassen, Miller-Rabin and Fermat Parallel Comparison

We also compared the parallel version of the three randomized algorithms (Figure 14) to each other to see if parallelization would benefit one algorithm more than the other. When comparing the parallel versions to each other we get the same results and the regular implementations (Figure 14). Solovay-Strassen is the fastest implementation, followed by Miller-Rabin, with Fermat coming last.

.9 Conclusion

All in all, when using a relatively small number of iterations, the parallelized version does not give much of an advantage, if any. This is due to the overhead of creating threads and splitting up the work costing approximately the same amount of time as simply running the iteration in sequence.

When running the algorithms with a large number of iterations the parallel versions do give a significant advantage. However, the randomized algorithms they are generally not run with that many iterations as they already give very precise results with a relatively low number of iterations.

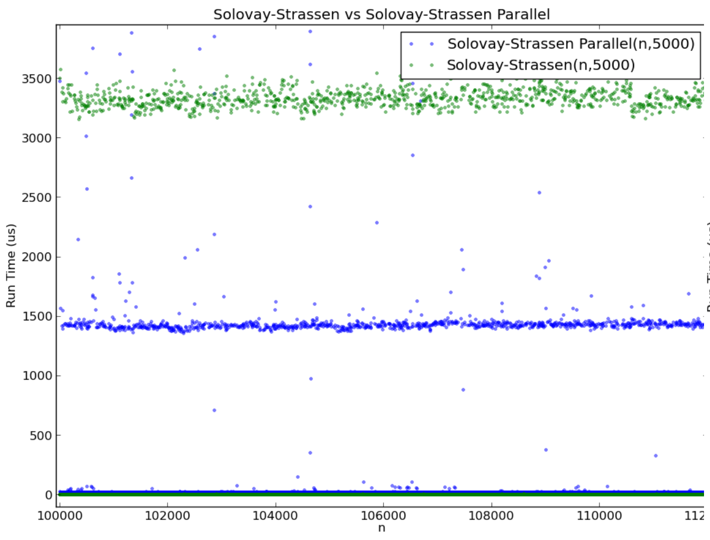

Fig. 13

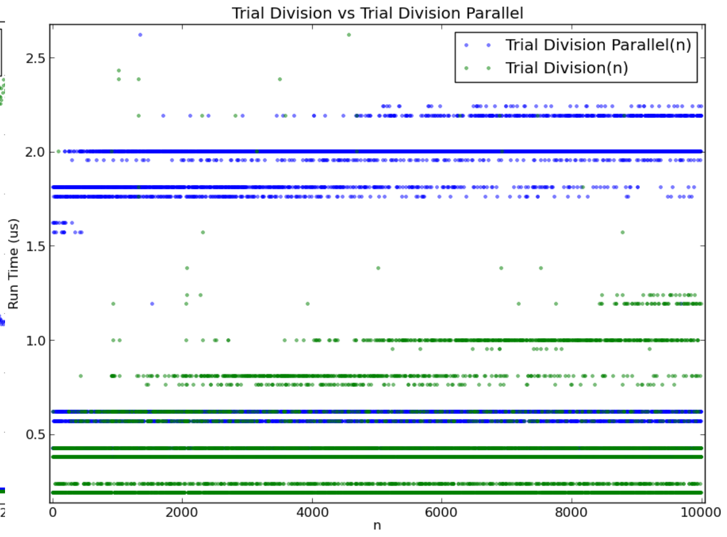

Fig. 15

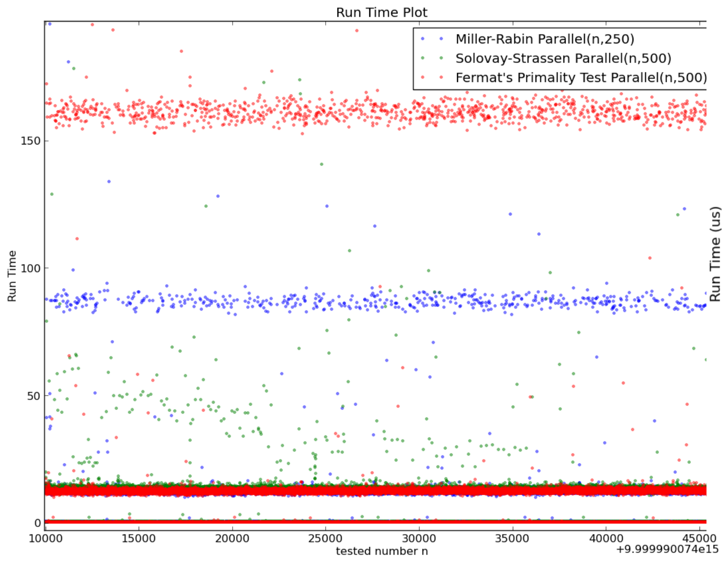

Fig. 14

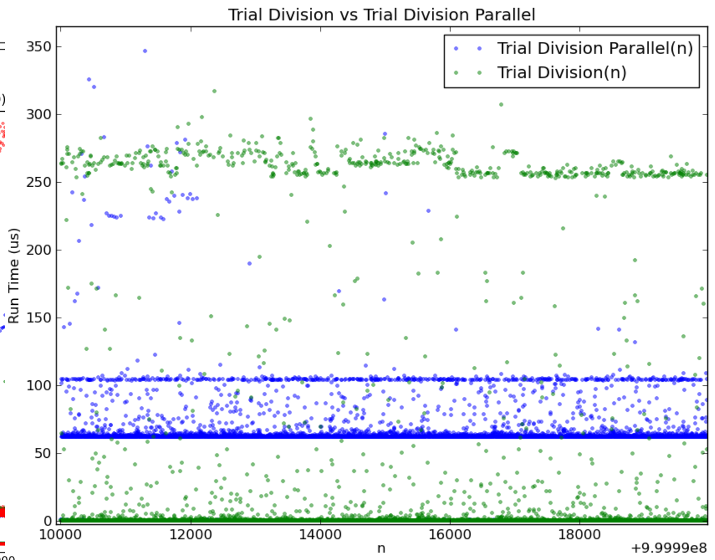

Fig. 16

V. Parallel Deterministic Primality Tests

A. *Trial Division*

Parallelization was also applied to the trial division algorithm. We did two tests, one with small values for n (range of 0 to 10000) (Figure 15) and one with large values (range of 10000 starting at 10^9) (Figure 16). When using small values of n we can see that, similarly to the randomized algorithms, the overhead of creating threads costs more time than calculating the answer in sequence. Due to the small size of the numbers and the speed of the algorithm, the boundaries of the precision of the measurement become visible but the difference in speed can still be distinguished. When using larger numbers (Figure 16) we can see that the parallelized version has a great advantage over the original implementation.

A.1 Agrawal-Kayal-Saxena

In the case of the Agrawal-Kayal-Saxena algorithm we were not able to parallelize it due to its complexity. It contains a number of loops with internal dependencies which means the iterations can not be run independently.

VI. The benefit of Randomization

Randomization has clear benefits, although the deterministic primality tests are polynomial or even sub-polynomial in nature, for large prime numbers computation time becomes increasingly impractical. Random algorithms provide a arbitrary degree of accuracy with a constant amount of run time. This makes randomized in our opinion superior to the deterministic algorithms. In the graph of Figure 17 the randomized algorithms have been compared to the trial division algorithm on a range of 10000 numbers starting from 10^9. As can be seen, the trial division algorithm is slower than the randomized

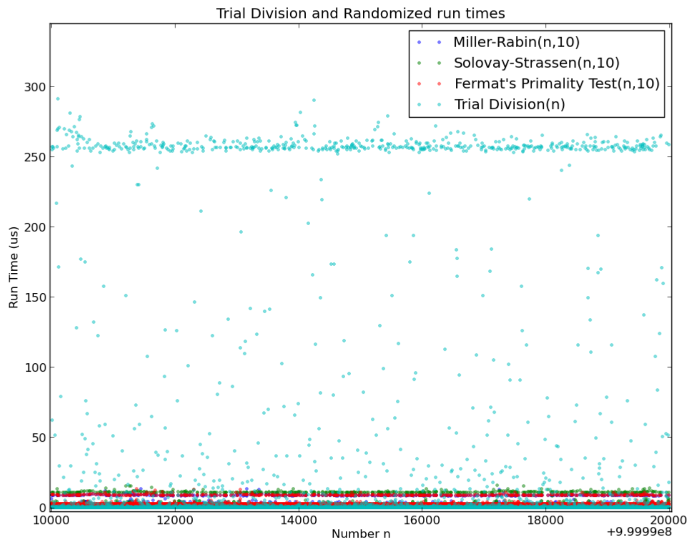

Fig. 17

algorithm by a large factor.

On top of the massive speed increase, the randomized algorithms also offer an impressive accuracy. Even at a low number of iterations, both the Miller-Rabin algorithm and the Solovay-Strassen algorithm make virtually no mistakes. The combination of high speed and high accuracy make the randomized algorithms very potent for finding prime numbers and generally a better choice than deterministic algorithms.

References

[1] Fermat's little theorem.
[2] Gnu mutiple precision arithmetic library.
[3] Openmp library.
[4] M. Rabin. Probabilistic algorithm for testing primality. *Journel of number theory*, 12:128–138, 1977.
[5] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM J. COMPUT.*, 6:84–85, 1977.