COMS W4995: Parallel Functional Programming

Deterministic Integer Factorization

Hazel Flaming

haf2136@columbia.edu, haf2136

December 18, 2025

1 Introduction and Motivation

Prime factorization is simultaneously one of the simplest and most difficult problems in computer science, with no currently known polynomial-time solution. The problem is considered to be NP-intermediate, as it is not known to be in P nor NP-complete.

Given that any positive integer $i \in \mathbb{Z}$ greater than 1 can be uniquely written as a product of primes to integer powers:

$$\forall N \in \mathbb{Z}, \exists! \ \langle e_1, e_2 ... e_n \rangle \ s.t. \ \ N = \Pi p_i^{e_i}$$

where p_n is the nth prime number, and the described vector has no trailing zero components, determining arbitrary integers' prime factors is extremely difficult and relevant. If a sufficiently fast factorization algorithm is found (not necessarily polynomial-time), all modern encryption would cease to be useful and digital privacy as we know it would be nonexistent. Aside from this dramatic possibility, prime factorization is a core element of number theory, and its results are useful essentially everywhere where integers are used, like the Fast Fourier Transform, hash functions, data error correction, and many others. Thus, this problem seems like a good non-IO-bound candidate for optimization and parallelization in Haskell.

While there exist probabilistic algorithms which can be much faster at finding factors for very large primes, these algorithms present several challenges. The first is that I am not a number theory PhD student and building algorithms structured around algebraic number fields is not something I fully understand nor have the time to do, although I would like to thank Kyle for believing in me enough to suggest I implement the General Number Field Sieve. The second is that these are probabilistic algorithms with unpredictable runtimes sensitive to small input variations and are inherently random, which makes testing and benchmarking, especially across multiple cores, hard to do. The third is that most of them rely on a large set of smaller algorithms like sparse matrix solving, and thus parallelizing them would involve first parallelizing the sub-algorithms, which could be final projects of their own. The final challenge is that these algorithms are spiritually poisonous with respect to the beauty of Haskell– Haskell's strength comes from its determinism, and probabilistic algorithms are somewhat opposed to this determinism.

Thus, this project aims to specifically tackle *deterministic* factorization, utilizing algorithms which produce either a factor or assert that a number is prime within a predictable amount of time with a predictable amount of steps. With this established, we may begin.

2 Uninformed Trial Division

Trial division is the simplest and most commonly used algorithm when factoring small composites. Its process is simple: test a candidate m to see if it is a factor by determining the remainder (n mod m) of dividing n by m. If it is not a factor (nonzero remainder), move on to the next candidate. If all candidates (which range from 2 to $\sqrt{N}$) are exhausted, the number is prime. A simple implementation is as follows, using the return type of

```
Maybe(Integer,Integer)
```

where the algorithm either returns

```
Just(factor,remainder)
```

or

```
Nothing
```

to indicate the number is prime.[1]

```
tdum :: Integer -> Integer -> Maybe(Integer,Integer)
tdum n m = f 2
  where f i | i > m = Nothing
            | (n `rem` i) == 0 = Just (i,n `div` i)
            | otherwise = f (i+1)
```

[1] As a side note, these algorithms are designed to find a *single* factor of N, rather than all factors, as finding the first factor of any N is much harder than finding additional factors. A recursive factoring method (which we will call `rfactor`) based on the given return signatures is simple: if the algorithm returns a `Just (x,y)`, return `x : rfactor y`, otherwise just return `[N]`.

The algorithm accepts N to be factored, and a max bound m, which should be set to $\sqrt{N}$ to explore the entire possibility space.

Despite its simplicity, this algorithm performs remarkably well, with a 64-bit worst-case factoring time of $\approx$ 115.03 seconds. For this and all following variants of trial division, an n-bit worst-case factoring time refers to the amount of time the algorithm takes to verify an n-bit integer is prime, as the algorithm must explore all factor possibilities to make this determination. Data is averaged over 10 runs, with one cache-warming run.

3 Informed Trial Division

An obvious optimization to Uninformed Trial Division is to first test 2 as a factor, and if it is not, only test $3, 5, 7, 9, 11$ and so on. However, this can be optimized further– we can then test 3, and then only test numbers which are not multiples of 2 and 3: $5, 7, 11, 13, 17$ and so on. This process motivates the design of a better algorithm: informed trial division.

This algorithm makes use of the observation that if you take the first n primes and multiply them together,

$$c = p_1 p_2 ... p_n$$

then any number above c which can be represented as c plus one of the first n primes can be described as

$$\begin{aligned} c + p_i &= p_1 p_2 ... p_{i-1} p_i p_{i+1} ... p_n + p_i \\ &= (p_1 p_2 ... p_{i-1} p_{i+1} ... p_n + 1) p_i \end{aligned}$$

and thus has a factor p_i.

Thus, to factor some N, if we first fix n and ensure the first n primes are *not* factors of N, then we know any integers in $[1, c]$ with a factor in the first n primes (we will name this set S) will *not* be a factor of n, nor will any member of S plus c, $s + kc, s \in S, k \in \mathbb{Z}$, as under this pattern we know some prime in the first n primes can be factored out of $s + kc$.

Thus, we create a 'wheel' with circumference c (product of first n primes) and a list of all integers $[1, c]$ with *no* factors in the first n primes (spokes). Then, we only need to check those candidates offset by integer multiples of c up to $\sqrt{N}$, as all other numbers will be guaranteed to have a factor in the first n primes, and thus cannot be a divisor of N. An implementation of the algorithm is as follows:

```haskell
spokes :: [Integer] -> [Integer] -> [Integer]
spokes b c = filter (\x -> all (\p -> gcd x p == 1) b) c

wheelc :: Integer -> (Integer,[Integer])
wheelc cMax = (circumference,spokes basis [1,2..circumference-1])
  where basis = last . takeWhile (\x -> product x <= cMax) $ inits primes
        circumference = foldr (*) 1 (basis)

tdi :: Integer -> Integer -> Maybe (Integer,Integer)
tdi n cMax = tdim n (ceilSqrt n) cMax

tdim :: Integer -> Integer -> Integer -> Maybe(Integer,Integer)
tdim n m cMax | isJust s = s
              | otherwise = f c
  where s = tdum n c
        (c,o) = wheelc cMax
        f i | i > m = Nothing
            | any (\x -> (n `rem` (x + i) == 0)) o = fromCandidates n $ map (+i) o
            | otherwise = f (i+c)
```

The algorithm accepts N to be factored, a max bound which should be set to $\sqrt{N}$, and an additional argument `cMax` which serves as a circumference bound for the wheel.

This circumference bound, then, introduces our first hyperparameter. Because our test cases are arbitrarily fixed at 64-bit primes, we attempt to tune this hyperparameter to minimize the runtime of this test case (Figure 1).

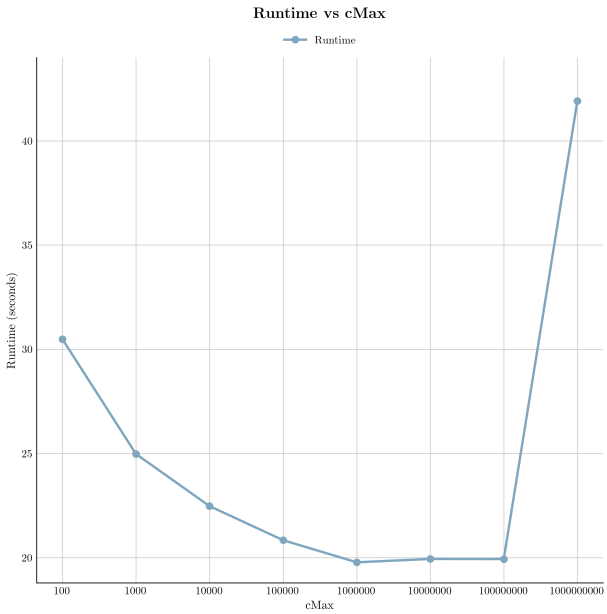

Figure 1: Runtime vs `cMax` of serial informed trial division on $N = 18446744073709551557$, a 64-bit prime

On this test case, `cMax` $=$ `1000000` performs best, with a worst-case 64-bit factoring time of ≈ 19.77 seconds, which is ≈ 5.8 times faster than brute force (Uninformed Trial Division).

4 Parallel Informed Trial Division

4.1 Attempt 1

The motivation behind the first attempted parallelization stems from the idea of checking several 'wheels' at once. In the following function, the spokes and wheel are computed in the same way as Informed Trial Division, under the idea that under sufficiently large N, this computation will be negligible compared to the rest of the algorithm. Then, the

`Strategies` library is used to test for factors in parallel among sequential wheel offsets:

```
tdip1 :: Integer -> Integer -> Integer ->
↪  Maybe(Integer,Integer)
tdip1 n cMax nT | isJust s = s
                | otherwise = p 1
  where s = tdim n c $ ceilSqrt c
        m = ceilSqrt n
        (c,o) = wheelc cMax
        p i | (c*i) > m = Nothing
            | any isJust il = asum il
            | otherwise = p (i+nT)
            where il = (map pc
            ↪  [c*i,c*(i+1)..c*(i+nT-1)] `using`
            ↪  parList rseq)
        pc i' | any (\x -> (n `rem` (x + i') == 0)) o
        ↪  = fromCandidates n $ map (+i') o
              | otherwise = Nothing
```

The main problem with this attempt is the tradeoff between the number of pseudo-threads `nT` (i.e. possible independent parallel computations created by the program) and the circumference of the wheel. If the number of threads is high, the sequential part of the algorithm that checks if any of the parallel parts found a factor becomes non-negligible, and if the circumference of the wheel is large, the initial serial computation of the wheel and spokes is non-negligible. If both are low, then parallelization overhead becomes non-negligible and work balance becomes uneven due to low `nT`.

Using arbitrarily[2] chosen hyperparameters of `cMax` = `100000000` and `nT` = `256`, using all available cores, the 64-bit worst-case factoring time is ≈ 5.06 sec (≈ 22.7 times faster than brute force), with a single core running time of ≈ 22.95 sec, indicating a parallelization overhead of about 3 seconds. Thus, while the algorithm parallelizes somewhat well despite its weaknesses, the parallelization overhead suggests this algorithm can be improved. Figure 2 plots the speedup of this algorithm as a function of available cores.

4.2 Attempt 2

A second attempt at this algorithm attempts to overcome the problems in Attempt 1 by allowing each parallel computation to check for factors in multiple offsets of the wheel. The algorithm accepts the N to be factored, a maximum bound `m`, a circumfrence bound `cMax`, the number of pseudo-threads `nT`, and the 'rotations per thread' `rpc`.

```
tdip2m :: Integer -> Integer -> Integer -> Integer ->
↪  Integer -> Maybe(Integer,Integer)
tdip2m n m cMax nT rpc | isJust s = s
                       | otherwise = p 0
  where s = tdim n c (ceilSqrt c)
        c' = rpc*c
        (c,o) = wheelc cMax
        p i | (c*c'*i) > m = Nothing
            | any isJust il = asum il
```

[2]These hyperparams seemed to work well; less time was spent choosing them because we improve this algorithm in the next section.

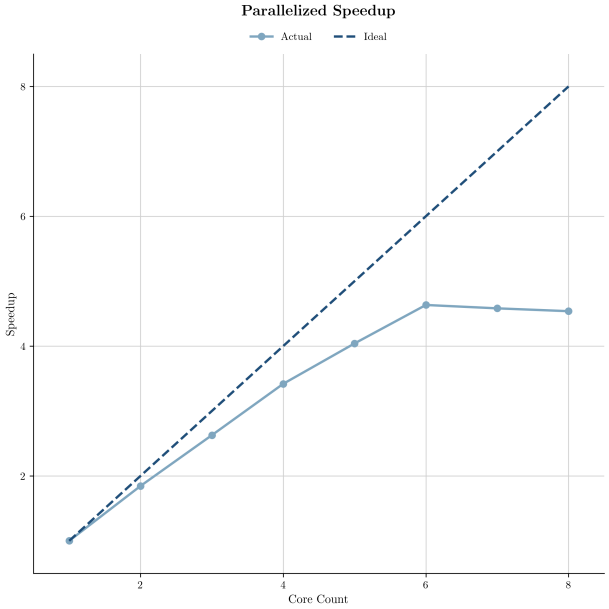

Figure 2: Runtime speedup vs core count on parallelized informed trial division on N = 18446744073709551557

```
            | otherwise = p (i+nT)
            where il = (map (pc 0)
            ↪  [c*c'*i,c*c'*(i+1)..c*c'*(i+nT-1)]
            ↪  `using` parList rseq)
      pc j k | j >= c' = Nothing
             | any (\x -> (n `rem` (x + (j + k)) ==
             ↪  0)) o = fromCandidates n $ map (+
             ↪  (j+k)) o
             | otherwise = pc (j+c) k
```

This algorithm introduces both additional complexity and hyperparameters. Based on the results in Informed Trial Division, in an attempt to reduce complexity that naturally comes with three hyperparameters, we fix `cMax` to `100000`, which is just under the tuned hyperparameter for serial informed trial division, under the intuition that the serial `cMax` should be higher than the parallel one, as wheel calculation is a serial process and thus should be minimized when parallelizing the algorithm.

Then, we must minimize the two remaining hyperparamters, `nT` and `rpc`. Figure 3 plots a runtime heatmap as a function of the improved parallelized trial division using all available cores.

Thus, based on the results of this runtime heatmap, we set `nT` to `256` and `rpc` to `16`, with a 64-bit worst case factor time of ≈ 3.84 sec (≈ 30.0 times brute force). Using these hyperparameters, we can now plot speedup as a function of the number of cores available in Figure 4. Under this improved algorithm, the single-core running time improves to ≈ 20.82 sec, reducing parallelization overhead from ≈ 3 to ≈ 1 seconds.

However, in Figure 4 (and also in Figure 2), we notice a distinct change in the marginal parallel performance as a consequence of adding cores 7 and 8. This divergence from what otherwise looks like efficient parallelism could be a

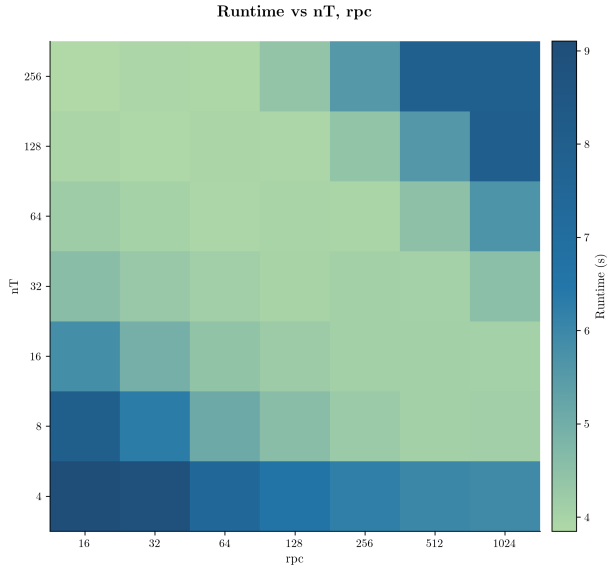

Figure 3: Runtime heatmap of parallelized informed trial division on $N = 18446744073709551557$, tuning `nT` and `rpc` using all available cores and `cMax` $= 100000$

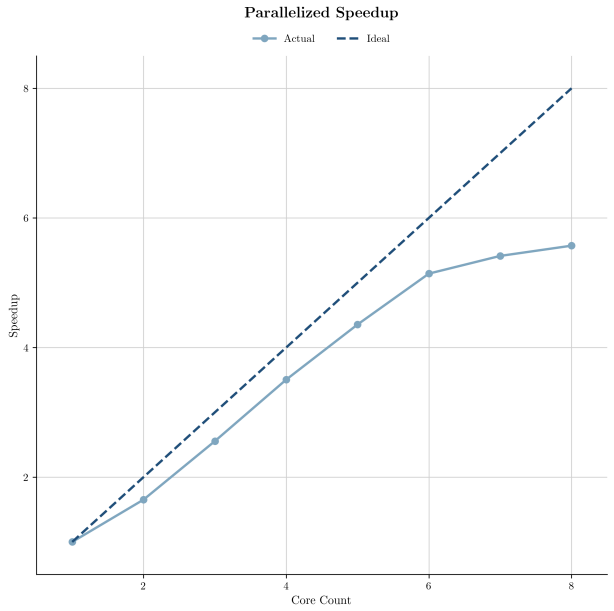

Figure 4: Runtime speedup vs core count on parallelized informed trial division on $N = 18446744073709551557$

consequence of poor hyperparameter selection in Attempt 1, however given the optimizations and tuning used on Attempt 2, the fact that this set of hyperparameters was *the best* out of 64 hyperparameter configurations almost guarantees the existence of an external factor causing this quirk.

The answer is simple hardware limitations– These tests were benchmarked on a 2021 Mac Book Pro M1, which has six 'performance cores' and two power-efficient but less powerful 'efficiency cores.' Thus, performance is limited once efficiency cores are always under load, which happens when running benchmarks using seven or eight cores.

Thus, because the reduction in marginal benefit of additional cores is because of hardware issues, rather than load balancing or overzealous garbage collection, and aditionally because this parallelized time is ≈ 5.15 times faster than serial trial division and the 8-core time is ≈ 5.58 times faster than the 1-core time, we expect decent parallel work balance and productivity, which is exemplified in Figure 5 using `threadscope`.

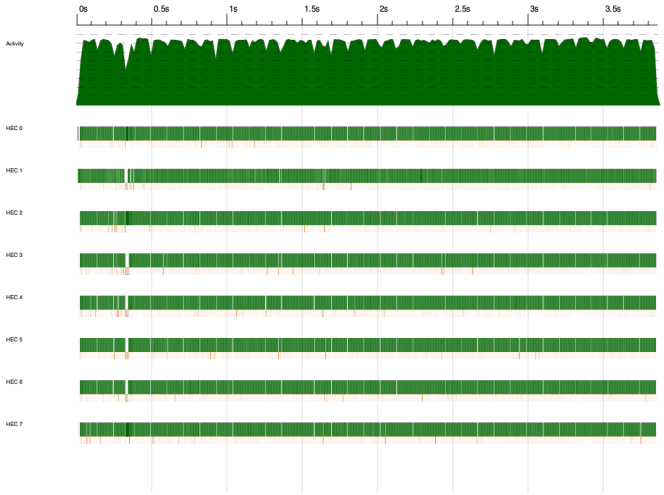

Figure 5: Threadscope log of parallelized informed trial division on $N = 18446744073709551557$

Additionally, spark creation looks reasonably bounded with minimal fizzles and garbage collects and no duds, as seen in Figure 6.

HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	8960	8921	0	0	11	28
HEC 0	0	1222	0	0	0	0
HEC 1	8960	736	0	0	11	20
HEC 2	0	1190	0	0	0	7
HEC 3	0	1129	0	0	0	1
HEC 4	0	1185	0	0	0	0
HEC 5	0	1173	0	0	0	0
HEC 6	0	1145	0	0	0	0
HEC 7	0	1141	0	0	0	0

Figure 6: Threadscope spark log of parallelized informed trial division on $N = 18446744073709551557$

5 Fermat's Method

While we got extremely good results out of the parallelization in Parallel Informed Trial Division, trial division still is best known for finding *small factors* of given composites,

which is obvious given that it starts at the beginning of $\mathbb{Z}$ and works its way upwards. Its main weakness is when a given composite has two factors, both close to $\sqrt{N}$. If the given composite is a perfect square, the factor time under trial division is equivalent to the worst-case factor time created by feeding the algorithm a prime number.

Pierre de Fermat made the observation that any odd integer can be represented as

$$N = a^2 - b^2 = (a - b)(a + b)$$

where the trivial representation has either $a + b$ or $a - b$ set to 1.

Within the context of prime factorization, under the conditions indicated in the last paragraph, if N has two factors close to $\sqrt{N}$, by this property we know these factors can be written as $a - b$ and $a + b$. Given that N is assumed to be odd, they must be decomposable in this way, as the only way that this would be impossible is if the difference in the factors is odd, which is impossible for two odd numbers. Thus, based on Fermat's observation,

$$b^2 = a^2 - N$$

means that if $a^2 - N$ is a perfect square, then we can find two factors, $a - b$ and $a + b$. Thus, under Fermat's method, if the given N is a perfect square or close to a perfect square, the algorithm will find a factor almost instantaneously, in large contrast to Informed Trial Division.

Fermat's method, for this reason, involves starting a at $\sqrt{N}$, testing $a^2 - N$ to see if it is a perfect square, and incrementing a if not. We can make the observation that for sufficiently large N, for every increment of a close to $\sqrt{N}$, we eliminate

$$\approx \sqrt{(\sqrt{N}+1)^2 - N} = \sqrt{2\sqrt{N}+1}$$

possible factors (which is ~ 250 for $N = 1$ billion), effectively performing trial division $\sqrt{2\sqrt{N}+1}$ times in a single Fermat step. However, it should be noted that this efficiency drops off quickly, and a single step of Fermat's method is more expensive, due to working with extremely large numbers. Thus, the optimal strategy to aim for seems to involve using Fermat's Method and Trial Division on different sections of the domain $[1, \sqrt{N}]$. An implementation of Fermat's method is given below:

```
fermatm :: Integer -> Integer ->
↪  Maybe(Integer,Integer)
fermatm n m = f $ floorSqrt n
  where f a | b2 < 0 = f $ a+1
            | a - b < m = Nothing
            | b*b == b2 = Just (a-b,a+b)
            | otherwise = f $ a+1
            where b2 = a*a-n
                  b = floorSqrt b2
```

The algorithm accepts an N to be factored, and a *minimum* bound `m` (as opposed to maximum as in Uninformed Trial Division, Informed Trial Division), such that the algorithm will search for factors in $[m, \sqrt{N}]$.

Similar to Informed Trial Division, the worst-case test case is still a prime number. However, because trial division is more computationally and algorithmically efficient over most of the domain of possible factors, testing for Fermat's method will involve testing $[0.8 \cdot \sqrt{N}, \sqrt{N}]$ rather than the entire possible factor domain to capture the idea that we only intend to use Fermat's method to rule out candidates on the higher end of the possible factor domain. In practice, $0.8 \cdot \sqrt{N}$ is a little too low, but setting it to a more accurate $0.9 \cdot \sqrt{N}$ makes runtimes fast enough that IO fluctuations and system noise make comparison and benchmarking hard.

Running this algorithm on $N = 18446744073709551557$, we get a worst case 64-bit factoring time[3] of ≈ 8.60 seconds.

6 Parallel Fermat's Method

Similar to the methodology expressed in Parallel Informed Trial Division, the following parallel implementation functions by, for each individual 'thread,' checking $a^2 - N$ for $a = \sqrt{N} + o + k \cdot \mathrm{nT}, k \in \mathbb{Z}$, where o is some offset $[1, \mathrm{nT}]$ so that threads don't overlap work. The algorithm accepts N to be factored, some minimum bound `m` as in Fermat's Method, a pseudo-thread count `nT`, and a computations per thread bound of `nC`. The algorithm is below:

```
fermatmp :: Integer -> Integer -> Integer -> Integer
↪  -> Maybe(Integer,Integer)
fermatmp n m nT nC = p $ floorSqrt n
  where p a | b2 < 0 = p $ a+1
            | a - b < m = Nothing
            | b*b == b2 = Just (a-b,a+b)
            | any isJust il = asum il
            | otherwise = p $ a+nT*nC
            where il = (map (p' (a+nT*nC))
            ↪  [a+1..(a+nT)] `using` parList rseq)
                  b2 = a*a-n
                  b = floorSqrt b2
        p' m' a' | b2' < 0 = p' m' $ a'+nT
                 | a' > m' = Nothing
                 | b'*b' == b2' = Just (a'-b',a'+b')
                 | otherwise = p' m' $ a'+nT
                 where b2' = a'*a'-n
                       b' = floorSqrt b2'
```

Again presented with two hyperparameters, we create another runtime heatmap to optimize runtime for a worst case 64-bit input in Figure 7.

Thus, based on the results of this heatmap, we set `nT` to and `nC` to 256 and 16 respectively, which corresponded to a worst case factoring time of ≈ 1.60 sec (≈ 5.2 times serial Fermat's method).

Using these hyperparameters, we may again plot runtime versus the number of available cores in Figure 8, with almost perfect parallel performance until the hardware limitations kick in past six cores, benching a single-core runtime of ≈ 8.12 sec indicating almost no parallel over-

[3] (for a fifth of the possible factor domain)

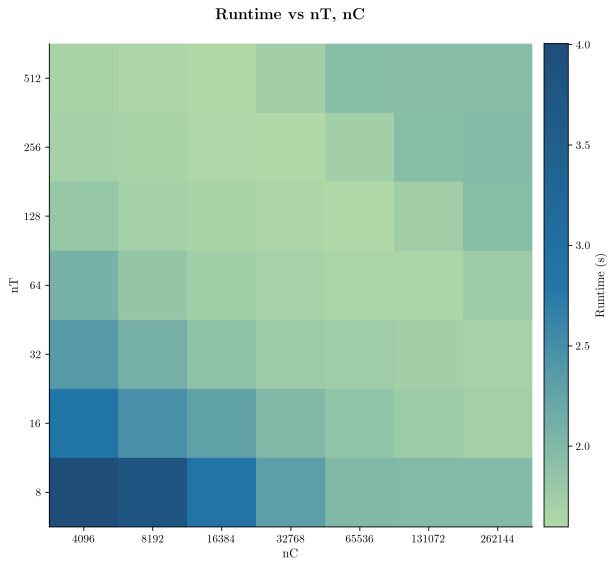

Figure 7: Runtime heatmap of parallelized Fermat's method on $N = 18446744073709551557$ tuning `nT` and `nC` using all available cores

head and an eight-core runtime of ≈ 1.56 sec[4] (≈ 5.4 times serial Fermat's method). Thus, we expect good work balance, verifiable via `threadscope` in Figure 9, and bounded spark magnitude, minimal fizzles and garbage collects, and no duds as seen in Figure 10.

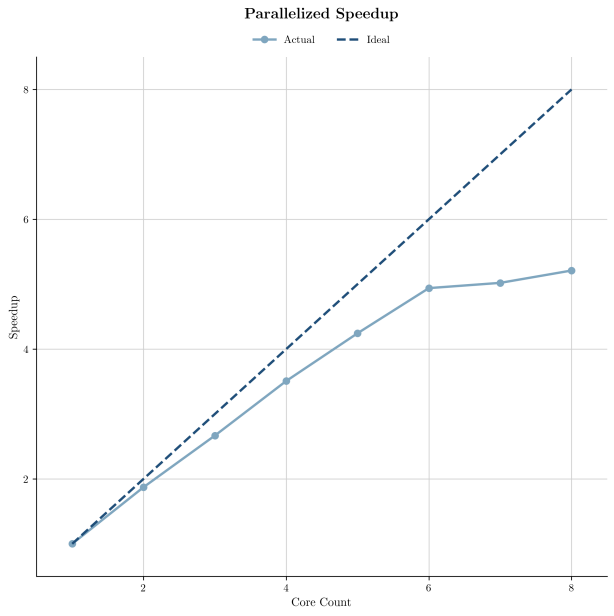

Figure 8: Runtime speedup vs core count on parallelized Fermat's method on $N = 18446744073709551557$

[4]This result pulls from the data testing runtime vs core count, whereas the above 1.60 figure pulls from the hyperparameter tuning data; These are results of the same benchmark at different times.

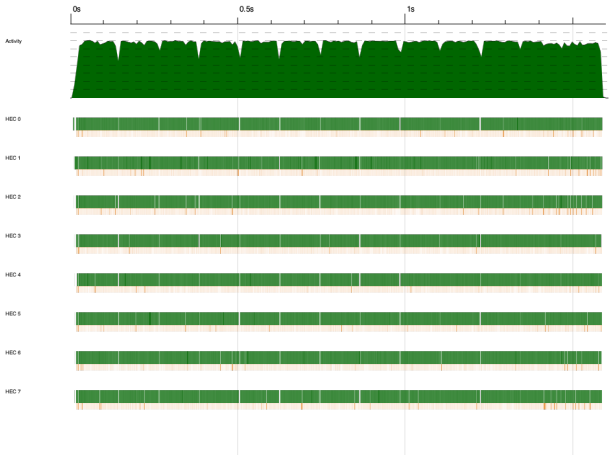

Figure 9: Threadscope log of parallelized Fermat's method on $N = 18446744073709551557$

HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	3328	3319	0	0	0	9
HEC 0	0	436	0	0	0	0
HEC 1	3328	352	0	0	0	9
HEC 2	0	415	0	0	0	0
HEC 3	0	434	0	0	0	0
HEC 4	0	433	0	0	0	0
HEC 5	0	419	0	0	0	0
HEC 6	0	426	0	0	0	0
HEC 7	0	404	0	0	0	0

Figure 10: Threadscope spark log of parallelized Fermat's method on $N = 18446744073709551557$

7 Extensions

Non-arbitrary automated hyperparameter tuning is the first major obvious extension of this work, given that hyperparameters were set manually, and some were fixed after a single benchmarking run (`cMax`). By estimating expected runtimes based on input size and core count as functions of each hyperparameter, more accurate parameter settings can be linearly or nonlinearly related to $\sqrt{N}$, thus ensuring each algorithm is as fast as possible over larger domain windows and large members of $\mathbb{Z}$.

Additionally, as defined in Parallel Informed Trial Division and Parallel Fermat's Method, we have two well-parallelized algorithms for deterministically finding factors of N. Thus, following more accurate hyperparameter tuning as discussed in the previous paragraph and the observations about the decreasing efficiency of Trial Division for large factors and Fermat's Method for small ones, by strategically setting the Trial Division maximum and Fermat's method minimum which are both represented as `m` in their respective algorithms at some value in $[2, \sqrt{N}]$ (likely as some function of $\sqrt{N}$), we can maximize efficiency of factor finding over the entire domain by running both parallel algorithms *in parallel*, thus making `m` a hy-

perparameter of this larger function. This can be achieved by performing runs of each algorithm over portions of the domain (logging the `m` at which Fermat's method becomes more efficient) over integers of increasing size and regressing on `m` as a function of $\sqrt{N}$.

References

[1] Wikipedia contributors. Fermat's factorization method. https://en.wikipedia.org/wiki/Fermat%27s_factorization_method, 2025. Accessed: 2025-12-15.

[2] Wikipedia contributors. Trial division. https://en.wikipedia.org/wiki/Trial_division, 2025. Accessed: 2025-12-15.

[3] Wikipedia contributors. Wheel factorization. https://en.wikipedia.org/wiki/Wheel_factorization, 2025. Accessed: 2025-12-15.