Daniel Garon
Dr Sanjay Rajopadhye
CS 475
2023-02-17

# PA2: The Sieve of Eratosthenes

## Algorithm Description

The sieve of Eratosthenes is an algorithm for finding prime numbers. For a selected endpoint *N*, the algorithm begins with 2, the first prime, and marks as composite every multiple of that number. The first unmarked number is the next prime, as it has not been identified as a multiple of any of the preceding numbers. The algorithm repeats for this next prime and continues the process of marking every multiple as a composite number. The algorithm terminates when the square of the next prime is greater than the selected endpoint *N*. This is because every prime less than or equal to *N* will have been marked as a multiple of a previous prime, given that at least one factor of a composite number must be less than or equal to the square root of that number.

## Experimental Setup

All data was collected using the CSU department machine with hostname “Annapolis” and the following specifications:

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              20
On-line CPU(s) list: 0-19
Thread(s) per core:  1
Core(s) per socket:  12
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               151
Model name:          12th Gen Intel(R) Core(TM) i7-12700K
Stepping:            2
CPU MHz:             3600.000
CPU max MHz:         5000.0000
CPU min MHz:         800.0000
BogoMIPS:            7219.20
Virtualization:      VT-x
L1d cache:           48K
L1i cache:           32K
L2 cache:            1280K
L3 cache:            25600K
```

## Sequential Improvements

The base code implemented the algorithm in brute-force fashion, iterating from the current prime *p* to *N* and checking every number along the way for divisibility. I added the following sequential improvements:

- marking using a stride rather than testing every element for divisibility
- starting the marking off at the square of the current prime
- marking off multiples of only those primes that are no more than $\sqrt{N}$
- marking and storing only odds

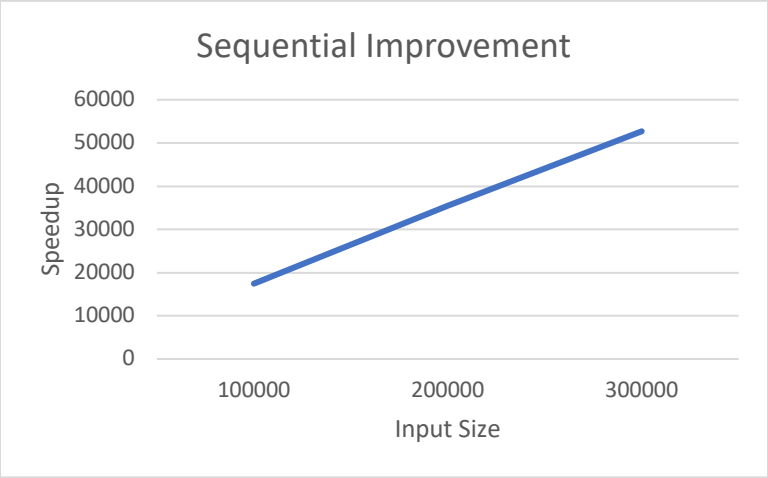


*Figure 1: Sequential Improvement*

| Input Size | Execution Time (s) | |
|---|---|---|
| | sieve.c | sieve1.c |
| 100,000 | 1.046 | 0.00006 |
| 200,000 | 3.896 | 0.00011 |
| 300,000 | 8.432 | 0.00016 |

*Table 1: Sequential Improvement*

The sequential optimization produced improvements in execution time of four orders of magnitude. The amount of improvement grows with the input size, suggesting not only improvement by a constant factor, but an improved algorithm.

The naïve version works by iterating over $N - p_i$ by the total number of primes less than or equal to $N$, approximated by $\frac{N}{\ln N - 1}$ .

The sequentially optimized sieve exhibits a (sequential) time complexity of $T(n) = O(n \ln \ln n)$.

## Parallelization

The sieve program operates using a while loop which terminates once we reach $\sqrt{N}$ that repeatedly executes a for loop which is responsible for marking the composites. Using OpenMP, I parallelized the for loop so that multiple threads could perform the marking simultaneously. The threads then synchronize and wait for the master thread to find the next prime.

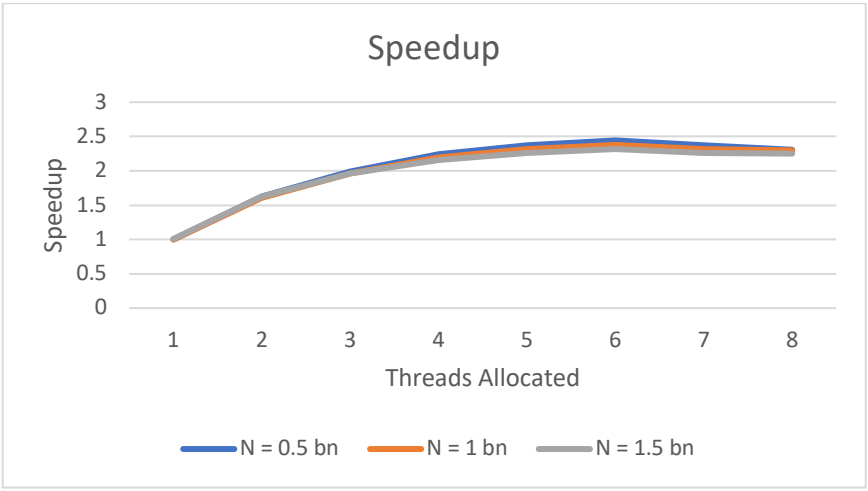


*Figure 2: Speedup for sieve2*

The parallelized version of the program exhibited a similar speedup trend across the selected input ranges. The speedup curve flattens out as more threads are allocated. I believe this trend to be due to the impact of cache misses. This memory I/O forms a bottleneck that limits the performance gains of parallelism.

I also believe that “false sharing” negatively impacts the parallel performance of this program. I believe that when any one thread writes to a segment on which one or more other threads is also operating, this forces their local caches to reload the line containing that array element to maintain cache coherency. This causes many more reads than is necessary, greatly increasing the execution time.

My results for sieve2, were also inconsistent, with the program on rare occasions taking very long to complete (up to several hundred times longer). I believe that this is the result of bad luck in the order the threads iterate over the shared array. If more than one thread is writing to the same segment, this can result in a repeated reloading of the same cache line by both threads, racking up significant memory access costs.

## Loop Tiling

I restructured the loop nest so that each iteration of the outermost loop would iterate repeatedly over a predetermined length of the *mark* array. By doing this, we can improve locality as each thread repeatedly iterates over the same section of the array marking off the multiples of all of the precomputed primes. This also mitigates any “false sharing” as only near the boundaries of the blocks will there potentially be overlap as cache line boundaries may not align with block segments.

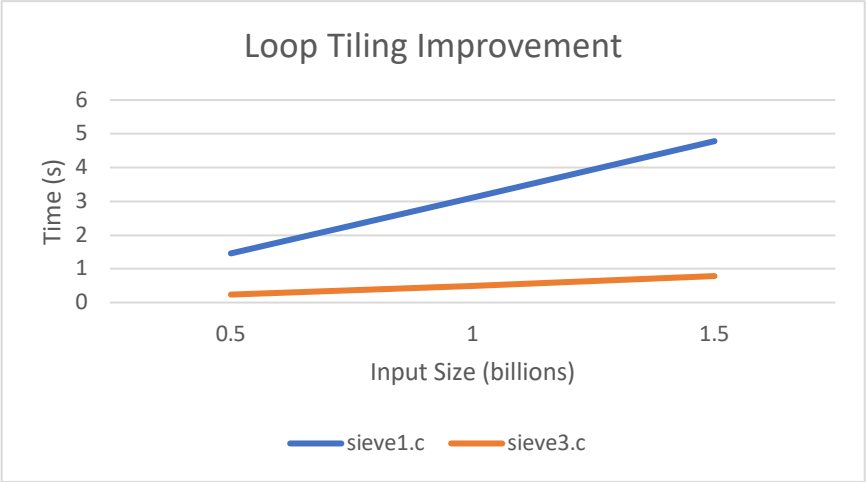


*Figure 3: Loop Tiling Improvement*

The locality improvement significantly reduced the execution times for the selected inputs. During testing, I observed that a block size of approximately 96,000 produced the fastest results. One of the sequential improvements was to store only the odd integers, reducing the space consumed by the *mark* array to $\frac{N}{2}$ bytes.

A consequence of my implementation is that loop indices which correspond to integers are translated to find the corresponding location in the *mark* array. The relationship from number to location in the *mark* array is: $index\ of\ number = \frac{i-1}{2}$ . This means that a block size of 96,000 corresponds to roughly the size of the L1d cache.

The L1 data cache is 48 kilobytes in size, suggesting that there is a performance decline when the machine must access memory lower in the hierarchy. There is a slow and steady performance decline as the block size increases.

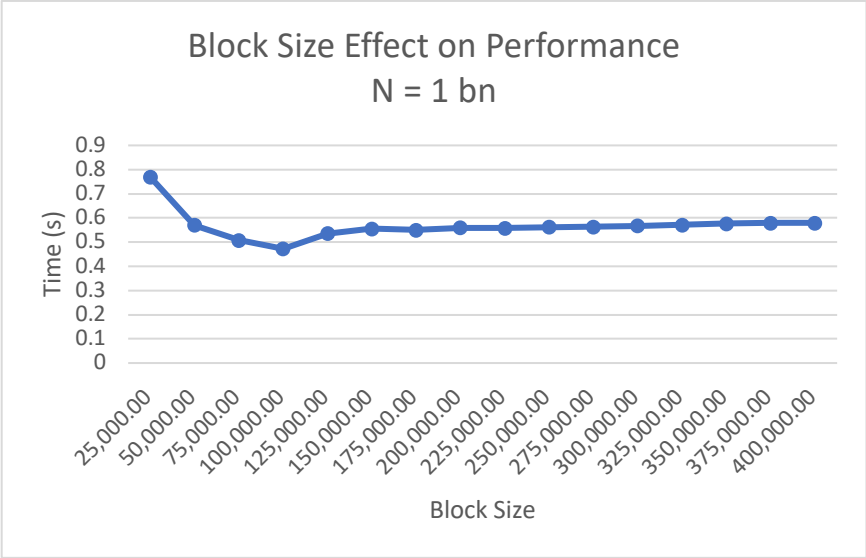


*Figure 4: Block Size Effect on Performance*

As the block size continues to increase, there is a sharper decline when the size exceeds approximately 2,700,000. This number when halved corresponds to the combined sizes of the L1d cache (48 KB) and the L2 cache (1,280 KB).

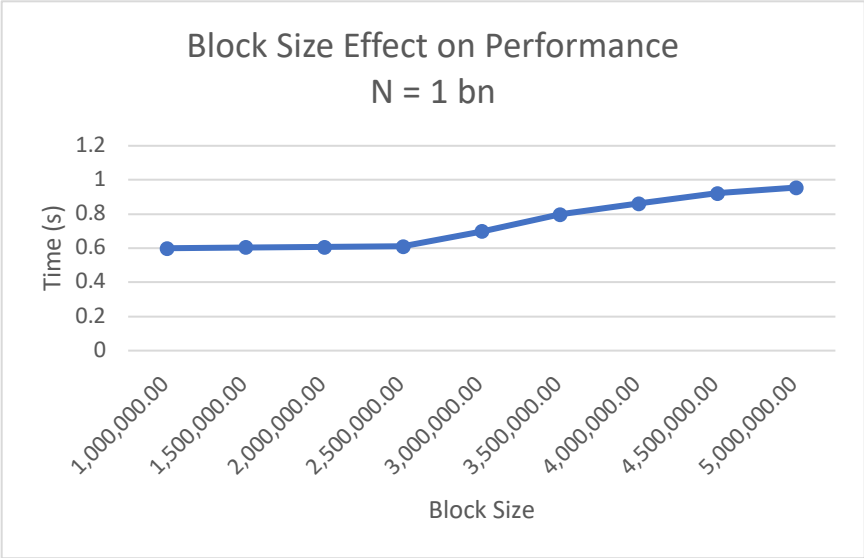


*Figure 5: Block Size Effect on Performance*

By selecting a very small block size, I believe I can essentially recreate the cache ping-ponging that I occasionally ran into with sieve2. The low block size leads to many threads concurrently writing to the same segments of the array.

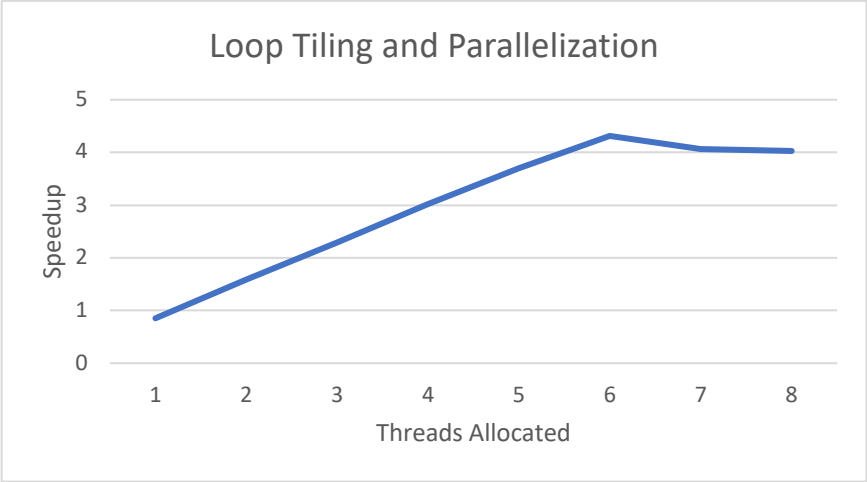


*Figure 6: Speedup for sieve4 relative to sieve3 with static scheduling*

I believe that the default static scheduling does not split up the work quite evenly. The threads responsible for marking the first chunks will encounter primes that when squared lie beyond their current segment. I have provided a conditional statement that will cause these threads to exit the *j* loop, which iterates over the primes.

I found that selecting a small chunk size more evenly distributed the work and yielded a significant performance improvement.

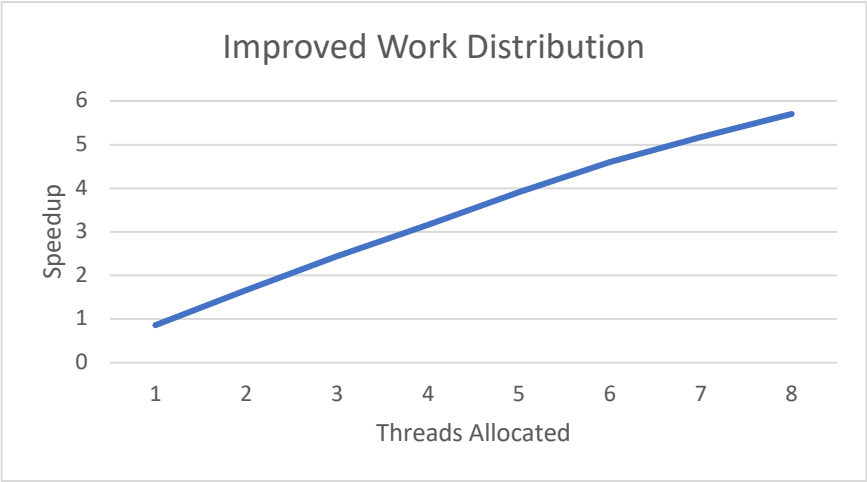


*Figure 7: sieve4 Speedup with Smaller Chunk Size*

## Conclusions

I observed the importance of considering not only whether various tasks can be performed in parallel but how those tasks interact with shared data. The relatively high cost of memory access combined with inattention to how and when the program will be required to make those memory accesses can lead to a program that not only receives no benefit from parallelism, but one that performs worse than its sequential counterpart. I also observed how important it is to carefully consider how the work will be distributed when choosing a parallelization strategy.