CI Pathway: Parallel Computing

Assignment: Shared Memory Parallelism with OpenMP

UCLA, Statistics
Hochan Son
Summer 2025
June 28, 2025

1 Introduction

This study examines the performance characteristics and optimization strategies of OpenMP-based parallel programming for computational prime number generation. Through systematic analysis of four distinct implementation approaches, we investigate how different parallelization strategies and algorithmic optimizations impact computational efficiency and scalability.

The primary learning objectives of this OpenMP exercise include:

1. Understanding race condition avoidance in shared-memory parallel programming
2. Comparing different OpenMP synchronization mechanisms (atomic operations, critical sections, and reductions)
3. Analyzing the relationship between algorithmic complexity and parallelization benefits
4. Evaluating parallel efficiency and scalability across varying thread counts
5. Demonstrating the fundamental principle that algorithmic optimization often provides greater performance gains than parallelization alone

The exercise employs a computationally intensive prime number generation task to stress-test different OpenMP implementations and reveal their relative strengths and limitations under high-performance computing conditions.

2 Hardware Environment

2.1 System Specifications

The experiments were conducted on the NCSA Delta HPC cluster, which is a high-performance computing environment designed for parallel processing tasks. The specifications of the system are as follows:

Table 1: NCSA Delta Compute Environment

Component	**Specification**
Compute Platform	NCSA Delta HPC Cluster
Login Node	dt-login04.delta.ncsa.illinois.edu
Compute Node	cn096.delta.ncsa.illinois.edu
Operating System	Linux 4.18.0-477.95.1.el8_8.x86_64
Distribution	Red Hat Enterprise Linux 8
Architecture	x86_64
Node Interconnect	HPE Slingshot

2.2 Processor Architecture

Table 2: AMD EPYC 7763 Processor Specifications

Parameter	**Value**
CPU Model	AMD EPYC 7763 64-Core Processor
Architecture	AMD Zen 3 (Milan)
Physical Cores	64 per socket
Hardware Threads	128 (2-way SMT)
Base Clock	2.45 GHz
Boost Clock	Up to 3.5 GHz
Manufacturing Process	7nm TSMC
Socket Type	SP3
L3 Cache	256 MB
Memory Support	DDR4-3200

3 Exercises Methodology

3.1 OpenMP Exercise

1. You will find this code in `projectsbecsurbanic` as either `prime_serial.c or prime_serial.f`.

   Your job is to accelerate this code using OpenMP. You should see a dramatic speedup if you use our OpenMP directives effectively.

   If you are not careful, you could introduce a race condition and have inconsistent results. If you use the same caution we used in the examples above, you will avoid this.

   To reiterate what you did in the previous module.

   (a) Compile with something like

   ```
   nvc -mp prime_serial.c
   or
   nvfortran -mp prime_serial.f
   ```

   (b) Grab multiple cores on a compute node with something like

   ```
   srun --account=becs-delta-cpu --partition=cpu-interactive \
   --nodes=1 --cpus-per-task=32 --pty bash
   ```

   (c) Set the number of threads you wish to run with using something like

   ```
   export OMP_NUM_THREADS=16
   ```

   Submit your code and your timings for at least 1, 4, 8, 16 and 32 threads.

4 Solution to OpenMP Exercise

4.1 Method 0: Serial Code

The serial code for prime number generation is a straightforward implementation that checks each number for primality by testing divisibility against all smaller numbers. This method is inherently sequential and does not leverage parallel processing capabilities.

```
for (i = 2; i <= n; i++) {
  for (j = 2; j < i; j++) {
```

```
    if (i % j == 0) {
      #pragma omp atomic
      not_primes++;
      break;
    }
  }
}
```

4.2 Method 1: Atomic Operations

Using atomic operations for thread-safe updates to shared variables is a common approach in OpenMP. This method ensures that only one thread can update the shared variable at a time

```
#pragma omp parallel for private(i,j)
for (i = 2; i <= n; i++) {
  for (j = 2; j < i; j++) {
    if (i % j == 0) {
      #pragma omp atomic
      not_primes++;
      break;
    }
  }
}
```

4.3 Method 2: Manual Reduction with Critical Sections

This method uses a critical section to ensure that only one thread can update the shared variable at a time. While this approach is safe, it can lead to performance bottlenecks due to contention among threads.

```
#pragma omp parallel private(i,j,local_not_primes)
{
  local_not_primes = 0;

  #pragma omp for schedule(static)
  for (i = 2; i <= n; i++) {
    // computation...
    local_not_primes++;
  }
  #pragma omp critical
  {
    not_primes += local_not_primes;
  }
}
```

4.4 Method 3: OpenMP Reduction

Leverages OpenMP's built-in reduction clause:

```
#pragma omp parallel for private(i,j) reduction(+:not_primes)
for (i = 2; i <= n; i++) {
    for (j = 2; j < i; j++) {
        if (i % j == 0) {
            not_primes++;
            break;
        }
    }
}
```

4.5 Method 4: Algorithmic Optimization

Implements optimized trial division with reduced computational complexity, this method significantly reduces the number of iterations required to check for primality, especially for larger numbers. It skips even numbers and only checks divisibility up to the square root of each number.

```
#pragma omp parallel for private(i,j) reduction(+:not_primes)
for (i = 2; i <= n; i++) {
    if (i == 2) continue;
    if (i % 2 == 0) { not_primes++; continue; }

    int sqrt_i = (int)sqrt(i);
    for (j = 3; j <= sqrt_i; j += 2) {
        if (i % j == 0) {
            not_primes++;
            break;
        }
    }
}
```

5 Performance Analysis

5.1 Execution Time Results

The execution times for each method across different thread counts are summarized in Table **??**. The results demonstrate the performance improvements achieved through parallelization and algorithmic optimization.

Table 3: Execution Time Performance Comparison (seconds)

Method	**1 Thread**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
Serial Baseline	18.510	18.519	18.514	18.500	18.510
Method 1 (Atomic)	18.526	8.411	4.323	2.342	1.656
Method 2 (Manual)	24.462	10.384	5.555	3.008	1.878
Method 3 (Reduction)	20.161	8.147	4.701	2.510	1.523
Method 4 (Optimized)	0.035	0.019	0.015	0.021	0.037

5.2 Speedup Analysis

The speedup factor relative to the serial baseline is calculated for each method across different thread counts. The results are presented in Table **??**. Method 4 demonstrates a revolutionary performance improvement due to its algorithmic optimization, achieving speedups of up to 1200$\times$ compared to the serial implementation. The result also shows that Methods 1-3 achieve significant speedups through parallelization, with Method 3 (OpenMP reduction) achieving the highest speedup among trial division methods.

Table 4: Speedup Factor Relative to Serial Baseline

Method	**1 Thread**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
Method 1 (Atomic)	1.00$\times$	2.20$\times$	4.28$\times$	7.91$\times$	11.19$\times$
Method 2 (Manual)	0.76$\times$	1.78$\times$	3.33$\times$	6.15$\times$	9.86$\times$
Method 3 (Reduction)	0.92$\times$	2.27$\times$	3.94$\times$	7.38$\times$	12.15$\times$
Method 4 (Optimized)	528$\times$	974$\times$	1234$\times$	881$\times$	500$\times$

5.3 Parallel Efficiency

The parallel efficiency is calculated as the ratio of the speedup factor to the number of threads used. This metric provides insight into how effectively the parallelization scales with increasing thread counts. The results are summarized in Table **??**. Methods 1-3 maintain good efficiency up to 32 threads, while Method 4 shows diminishing returns due to its extremely fast execution time.

Table 5: Parallel Efficiency (%)

Method	**1 Thread**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
Method 1 (Atomic)	100.0	55.0	53.5	49.4	35.0
Method 2 (Manual)	100.0	58.6	54.9	50.6	40.6
Method 3 (Reduction)	100.0	56.8	49.3	46.1	38.0
Method 4 (Optimized)	100.0	24.4	15.4	5.5	1.6

5.4 Performance Analysis Discussion

5.4.1 Trial Division Methods (1-3)

The three trial division implementations demonstrate excellent parallel scalability:

- **Method 1 (Atomic Operations)**: This method achieves the best balance of performance and simplicity, with 11.19× speedup at 32 threads and maintaining good efficiency (35%).
- **Method 2 (Manual Reduction)**: The manual reduction method shows higher single-thread overhead due to OpenMP parallel region setup, but achieves comparable scalability with 9.86× speedup at 32 threads.
- **Method 3 (OpenMP Reduction)**: The OpenMP Reduction method demonstrates the highest absolute speedup (12.15×) among trial division methods, validating the efficiency of OpenMP's built-in reduction implementation.

5.4.2 Algorithmic Optimization (Method 4)

Method 4 represents a paradigm shift in performance optimization:

- **Revolutionary Performance**: Achieves 500-1200× speedup over trial division methods
- **Algorithmic Complexity**: Reduces from $O(n^2)$ to approximately $O(n\sqrt{n})$ complexity

- **Limited Parallelization Benefits**: Shows decreasing efficiency with higher thread counts due to extremely fast execution times where overhead dominates
- **Optimal Thread Count**: Peak performance at 8 threads (0.015s) demonstrates the importance of matching thread count to problem characteristics

5.4.3 Key Performance Insights

1. **Algorithm vs. Parallelization**: The 500-1200× improvement from algorithmic optimization far exceeds the 10-12× gains from parallelization, reinforcing the principle that algorithmic efficiency is paramount.
2. **Synchronization Overhead**: All trial division methods show similar scalability patterns, indicating that synchronization strategy has minimal impact compared to computational complexity.
3. **Scalability Limits**: Method 4's efficiency degradation at high thread counts demonstrates that not all algorithms benefit from aggressive parallelization.
4. **Hardware Utilization**: The AMD EPYC 7763's 64-core architecture is well-utilized by Methods 1-3, but Method 4's speed makes thread overhead the limiting factor.

6 Conclusions

This comprehensive analysis of OpenMP parallelization strategies yields several critical insights for high-performance computing. The results demonstrate that while parallelization can yield significant performance improvements, the most substantial gains come from algorithmic optimizations that reduce computational complexity. While this exercise parallel programming is very complex in terms of the number of threads, synchronization strategies, and algorithmic optimizations. As such, the parallel debugging is also very complex. The baseline result from serial code was essential to validate the correctness of parallel implementations.

6.1 Primary Findings

1. **Algorithmic Supremacy**: The most significant performance improvement (500-1200×) came from algorithmic optimization rather than parallelization techniques, reinforcing the fundamental principle that efficient algorithms are the foundation of high-performance computing.

2. **OpenMP Strategy Effectiveness**: Among parallelization approaches, OpenMP reduction (Method 3) achieved the highest speedup (12.15$\times$), followed closely by atomic operations (Method 1), demonstrating the efficiency of OpenMP's built-in parallel constructs.

3. **Scalability Characteristics**: Trial division methods (1-3) exhibit excellent linear scaling up to 32 threads with 35-40% efficiency, while the optimized algorithm shows limited parallelization benefits due to its inherently fast execution.

4. **Race Condition Solutions**: All three synchronization strategies (atomic, critical sections, reduction) successfully eliminated race conditions while maintaining good parallel performance, with reduction showing slight superiority.

7 Appendix.code

Here's some of our code (Note the use of VerbatimInput from package `fancyvrb`):

7.1 Code A: prime_serial.c

```
# include <stdlib.h>
# include <stdio.h>

int main ( int argc, char *argv[] ){

  int n = 500000;
  int not_primes=0;
  int i,j;

  for ( i = 2; i <= n; i++ ){
    for ( j = 2; j < i; j++ ){
      if ( i % j == 0 ){
        not_primes++;
        break;
      }
    }
  }

  printf("Primes: %d\n", n - not_primes);

}
```

7.2 Code B: prime_parallel_norace_1.c

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>

#define MAX_THREADS 32

/*
  parallelism: using atomic operation
  compiler: private(i,j) -> atomic operation
  Speed: slow
*/

int main ( int argc, char *argv[] ){

  int n = 500000;
  int not_primes=0;
  int i,j;

  // Set number of threads at runtime
  int num_threads = MAX_THREADS;
  if (omp_get_max_threads() < MAX_THREADS) {
      num_threads = omp_get_max_threads();
  }
  omp_set_num_threads(num_threads);
  printf("Running␣with␣%d␣OpenMP␣threads\n", num_threads);

  // parallel running reduction with serial atomic counts
  #pragma omp parallel for private(i,j)
  for ( i = 2; i <= n; i++ ){
    for ( j = 2; j < i; j++ ){
      if ( i % j == 0 ){
        #pragma omp atomic
        not_primes++;
        break;
      }
    }
  }

  printf("Primes:␣%d\n", n - not_primes);

}
```

7.3 Code C: prime_parallel_norace_2.c

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>

#define MAX_THREADS 32

/*
  parallelism: manual reduction with scheduler with local variables
  compiler: private(i,j) -> schedule(static) -> critical
  Speed: normal
*/

int main ( int argc, char *argv[] ){

  int n = 500000;
  int not_primes=0; // global_shared_variables
  int i,j;

  // Set number of threads at runtime
  int num_threads = MAX_THREADS;
  if (omp_get_max_threads() < MAX_THREADS) {
      num_threads = omp_get_max_threads();
  }
  omp_set_num_threads(num_threads);
  printf("Running␣with␣%d␣OpenMP␣threads\n", num_threads);
  int local_not_primes;
  #pragma omp parallel private(i,j,local_not_primes)
  {
    local_not_primes = 0;

    #pragma omp for schedule(static)
    for ( i = 2; i <= n; i++ ){
      for ( j = 2; j < i; j++ ){
        if ( i % j == 0 ){
          local_not_primes++;
          break;
        }
      }
    }

    #pragma omp critical
    {
      not_primes += local_not_primes;
    }
  }
  printf("Primes:␣%d\n", n - not_primes);
}
```

7.4 Code D: prime_parallel_norace_3.c

```
# include <stdlib.h>
# include <stdio.h>
#include <omp.h>

#define MAX_THREADS 32

/*
  parallelism: private variable with auto reduction
  compiler: private(i,j) reduction(+:not_primes)
  Speed: fast O(n^2)
*/

int main ( int argc, char *argv[] ){

  int n = 500000;
  int not_primes=0; // global_shared_variables
  int i,j;

  // Set number of threads at runtime
  int num_threads = MAX_THREADS;
  if (omp_get_max_threads() < MAX_THREADS) {
      num_threads = omp_get_max_threads();
  }
  omp_set_num_threads(num_threads);
  printf("Running␣with␣%d␣OpenMP␣threads\n", num_threads);

  #pragma omp parallel for private(i,j) reduction(+:not_primes)
  for ( i = 2; i <= n; i++ ){
    for ( j = 2; j < i; j++ ){
      if ( i % j == 0 ){
        not_primes++;
        break;
      }
    }
  }
  printf("Primes:␣%d\n", n - not_primes);
}
```

7.5 Code D: prime_parallel_norace_4.c

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>
# include <math.h>

#define MAX_THREADS 32

/*
  parallelism: private variable with reduction + algorithm optimization
  compiler: private(i,j) reduction(+:not_primes)
  Speed: fastest O(nlogn)
*/

int main ( int argc, char *argv[] ){

  int n = 500000;
  int not_primes=0; // global_shared_variables
  int i,j;

  // Set number of threads at runtime
  int num_threads = MAX_THREADS;
  if (omp_get_max_threads() < MAX_THREADS) {
      num_threads = omp_get_max_threads();
  }
  omp_set_num_threads(num_threads);
  printf("Running␣with␣%d␣OpenMP␣threads\n", num_threads);

  #pragma omp parallel for private(i,j) reduction(+:not_primes)
  for ( i = 2; i <= n; i++ ){
    if (i == 2) continue;
    if (i % 2 == 0) { not_primes++; continue; }

    int sqrt_i = (int)sqrt(i);
    for (j = 3; j <= sqrt_i; j += 2) {
      if ( i % j == 0 ){
        not_primes++;
        break;
      }
    }
  }
  printf("Primes:␣%d\n", n - not_primes);
}
```

8 Appendix.hw2_result.txt

8.1 result.txt

```
=== Basic System Info ===
Date: Fri Jun 27 15:57:36 CDT 2025
System: Linux cn096.delta.ncsa.illinois.edu 4.18.0-477.95.1.el8_8.x86_64 #1 SMP
    Fri Apr 11 09:50:48 EDT 2025 x86_64 x86_64 x86_64 GNU/Linux
CPU Info: AMD EPYC 7763 64-Core Processor
=========================
!!!!STARTING SERIAL PROCESS TEST!!!!
=== Test with 1 threads ===
Start time: 2025-06-27 15:57:36.450548839
Primes: 41539
End time: 2025-06-27 15:57:54.964368664
Total wall clock time: 18.510 seconds
----------------------------------------

=== Test with 4 threads ===
Start time: 2025-06-27 15:57:55.973709207
Primes: 41539
End time: 2025-06-27 15:58:14.498672054
Total wall clock time: 18.519 seconds
----------------------------------------

=== Test with 8 threads ===
Start time: 2025-06-27 15:58:15.512942970
Primes: 41539
End time: 2025-06-27 15:58:34.035218023
Total wall clock time: 18.514 seconds
----------------------------------------

=== Test with 16 threads ===
Start time: 2025-06-27 15:58:35.048509581
Primes: 41539
End time: 2025-06-27 15:58:53.554000535
Total wall clock time: 18.500 seconds
----------------------------------------

=== Test with 32 threads ===
Start time: 2025-06-27 15:58:54.562205897
Primes: 41539
End time: 2025-06-27 15:59:13.076595090
Total wall clock time: 18.510 seconds
----------------------------------------

!!!!STARTING PARALLEL PROCESS TEST (fixed race:method-1)!!!!
=== Test with 1 threads ===
Start time: 2025-06-27 15:59:14.088347841
Running with 1 OpenMP threads
Primes: 41539
End time: 2025-06-27 15:59:32.620570332
Total wall clock time: 18.526 seconds
----------------------------------------

=== Test with 4 threads ===
Start time: 2025-06-27 15:59:33.630192093
Running with 4 OpenMP threads
Primes: 41539
End time: 2025-06-27 15:59:42.049976006
Total wall clock time: 8.411 seconds
----------------------------------------

=== Test with 8 threads ===
Start time: 2025-06-27 15:59:43.079146224
Running with 8 OpenMP threads
Primes: 41539
End time: 2025-06-27 15:59:47.406235593
Total wall clock time: 4.323 seconds
----------------------------------------

=== Test with 16 threads ===
Start time: 2025-06-27 15:59:48.415049547
Running with 16 OpenMP threads
Primes: 41539
End time: 2025-06-27 15:59:50.760977114
Total wall clock time: 2.342 seconds
----------------------------------------

=== Test with 32 threads ===
Start time: 2025-06-27 15:59:51.770523994
Running with 32 OpenMP threads
Primes: 41539
End time: 2025-06-27 15:59:53.431851186
Total wall clock time: 1.656 seconds
----------------------------------------

!!!!STARTING PARALLEL PROCESS TEST (fixed race:method-2)!!!!
=== Test with 1 threads ===
Start time: 2025-06-27 15:59:54.448569524
Running with 1 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:00:18.921800264
Total wall clock time: 24.462 seconds
----------------------------------------

=== Test with 4 threads ===
Start time: 2025-06-27 16:00:19.931034687
Running with 4 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:00:30.319875746
```

```
Total wall clock time: 10.384 seconds
----------------------------------------

=== Test with 8 threads ===
Start time: 2025-06-27 16:00:31.328475970
Running with 8 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:00:36.888727473
Total wall clock time: 5.555 seconds
----------------------------------------

=== Test with 16 threads ===
Start time: 2025-06-27 16:00:37.903255190
Running with 16 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:00:40.914943145
Total wall clock time: 3.008 seconds
----------------------------------------

=== Test with 32 threads ===
Start time: 2025-06-27 16:00:41.924359630
Running with 32 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:00:43.810975887
Total wall clock time: 1.878 seconds
----------------------------------------

!!!!STARTING PARALLEL PROCESS TEST (fixed race:method-3)!!!!
=== Test with 1 threads ===
Start time: 2025-06-27 16:00:44.822165891
Running with 1 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:04.987657596
Total wall clock time: 20.161 seconds
----------------------------------------

=== Test with 4 threads ===
Start time: 2025-06-27 16:01:05.997037263
Running with 4 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:14.147453508
Total wall clock time: 8.147 seconds
----------------------------------------

=== Test with 8 threads ===
Start time: 2025-06-27 16:01:15.155724353
Running with 8 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:19.860829984
Total wall clock time: 4.701 seconds
----------------------------------------

=== Test with 16 threads ===
Start time: 2025-06-27 16:01:20.870281625
Running with 16 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:23.383875555
Total wall clock time: 2.510 seconds
----------------------------------------

=== Test with 32 threads ===
Start time: 2025-06-27 16:01:24.391649188
Running with 32 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:25.917885911
Total wall clock time: 1.523 seconds
----------------------------------------

!!!!STARTING PARALLEL PROCESS TEST (fixed race:method-4)!!!!
=== Test with 1 threads ===
Start time: 2025-06-27 16:01:26.927240631
Running with 1 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:26.972756752
Total wall clock time: 0.035 seconds
----------------------------------------

=== Test with 4 threads ===
Start time: 2025-06-27 16:01:27.980335568
Running with 4 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:28.002686024
Total wall clock time: 0.019 seconds
----------------------------------------

=== Test with 8 threads ===
Start time: 2025-06-27 16:01:29.014081263
Running with 8 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:29.035905411
Total wall clock time: 0.015 seconds
----------------------------------------

=== Test with 16 threads ===
Start time: 2025-06-27 16:01:30.043787076
Running with 16 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:30.067917702
Total wall clock time: 0.021 seconds
----------------------------------------

=== Test with 32 threads ===
Start time: 2025-06-27 16:01:31.076661835
Running with 32 OpenMP threads
Primes: 41539
End time: 2025-06-27 16:01:31.116858273
```

```
Total wall clock time: 0.037 seconds
----------------------------------------
```