Parallel Programming Exercise 4 – 8

Author:	邱亮茗 (r12942159@ntu.edu.tw)
Student ID	R12942159
Department	Graduate Institute of Communication Engineering

(If you and your team member contribute equally, you can use (co-first author), after each name.)

1 Problem and Proposed Approach

(Brief your problem, and give your idea or concept of how you design your program.)

The task is to count the number of consecutive odd prime pairs (e.g., 3 and 5, 5 and 7) for all odd integers less than 1,000,000. The program must use parallel computing to achieve this efficiently, leveraging MPI.

Step1. Prime Number Generation: To identify primes, the program uses a modified Sieve of Eratosthenes, optimized by only storing and processing odd integers. This reduces memory usage and computational time since even numbers are not considered.

Step2. Parallelization Strategy: The sieve's workload is split among available processes. Each process is responsible for marking multiples of primes within its range and to achieve this, the program divides the range into smaller "blocks" and allocates these to different MPI processes.

Step3. Overlap in Data Segmentation: Overlapping the range of each process slightly helps account for consecutive primes that might be split across boundaries of different processes.

Step4. Counting Consecutive Primes: Once all non-prime odd numbers are marked, each process counts pairs of consecutive indices (i.e., consecutive odd numbers) that are both unmarked.

2 Theoretical Analysis Model

(Try to give the time complexity of the algorithm, and analyze your program with iso-efficiency metrics)

Sequential Sieve of Eratosthenes: For a sequential implementation of the sieve, the time complexity is O(nloglogn), where n is the upper limit of the range within which we find primes. This complexity comes from marking multiples of each prime up to sqrt(n).

Parallel Sieve Implementation: With p processors, the range n is divided across processors, each handling n/p elements. Thus, the total parallel time complexity is approximately: O(nloglogn / p + logp) .

=>To(n, p) = plogp

=> M(n) = n; n >= Cplogp

=> M(f(p)) / p = Cplogp / p = Clogp

=> The system has good scalability

3 Performance Benchmark

(Give your idea or concept of how you design your program.)

Table 1. The execution time

Processors	1	2	3	4	5	6	7	8
Real execution time	0.002604	0.001370	0.000958	0.001052	0.001757	0.001560	0.001412	0.003551
Estimate execution time	0.002604	0.001302	0.000868	0.000651	0.000520	0.000434	0.000372	0.000325
Speedup	1	1.901	2.7181	2.4752	1.4820	1.6692	1.8441	0.7333
Karp-flatt metrics		0.052	0.0525	0.2053	0.5937	0.5186	0.4656	1.4154

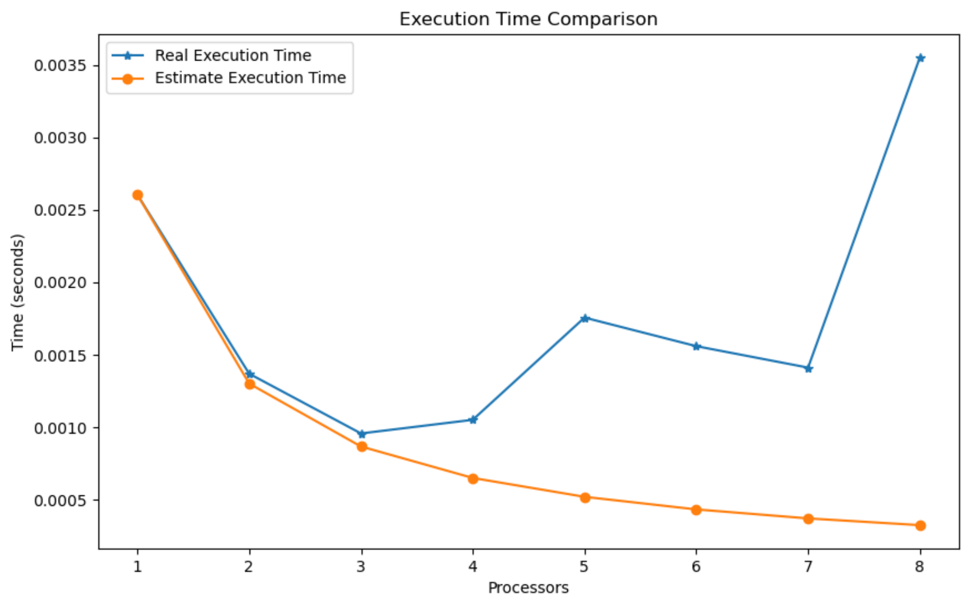

Figure 1. The performance of diagram

4 Conclusion and Discussion

(Discuss the following issues of your program

1. What is the speedup respect to the number of processors used?

   The speedup results demonstrate that the program scales well up to about 3 processors but encounters diminishing returns as the number of processors increases. This behavior aligns with Amdahl’s Law, where the speedup is limited by the serial fraction of the program. The parallel efficiency decreases as more processors are added, emphasizing the need for careful consideration of both the problem size and parallel overhead when scaling this algorithm.

2. How can you improve your program further more

   Instead of using MPI_Bcast to send the prime number to all processors, each processor can independently calculate the next prime number. By allowing each processor to determine primes locally (without relying on global communication), we can reduce the synchronization overhead and improve efficiency. This approach can also make the program more resilient to changes in the number of processors, as each processor can work autonomously. Otherwise, to improve cache performance, we can optimize the memory access patterns. One way to achieve this is by partitioning the sieve into smaller chunks that fit into the cache of each processor. By segmenting the range of numbers into blocks that are small enough to fit into the processor's cache, we can minimize cache misses and reduce the overall execution time.

3. How does the communication and cache affect the performance of your program?

   Communication overhead, especially when using MPI_Bcast, introduces latency and synchronization delays, which hinder scalability. As the number of processors increases, the overhead from frequent communication becomes more significant, resulting in diminishing speedup. Optimizing communication by allowing processors to work independently and reducing synchronization can improve performance. And, poor cache management can result in cache misses, reducing performance. By improving cache locality (e.g., through cache blocking and memory alignment) and minimizing global memory access, cache performance can be significantly improved. Using local storage for each processor can further enhance cache efficiency.

4. How does the Karp-Flatt metrics and Iso-efficiency metrics reveal?

   The increasing Karp-Flatt metric indicates that as more processors are used, the overhead increases significantly. This suggests that communication or synchronization overheads are becoming limiting factors in scaling the program efficiently. And given the rapid increase in overhead (as shown by the Karp-Flatt metric) and the decrease in speedup beyond 4 processors, the **iso-efficiency curve** will likely be steep. This suggests that the problem size must grow considerably to maintain the same efficiency when more processors are used, pointing to a limitation in the scalability of the program due to rising overheads.

)

Appendix(optional):

(If something else you want to append in this file, like picture of life game)