Homework 2 of CS7344

3.18

Given a list of n keys, $a[0], a[1], \cdots, a[n]$, all with distinct values, design a parallel algorithm to find the second-largest key on the list.

Answer:

1.Suppose we have p processors, then we firstly divide the list into p sublists;
2.Each processor find the max and second-largest value in its sublists;
3.Combine all values into a new list with $2p$ values;
4.Find the second-largest value in the new list.

Algorithm 1 Find Second Largest Key

Input: a[0...n-1]
Output: secondLargest
numCores ← p
localMaxSecondLargest[2p]
parallel for core from 0 to p-1
localMaxSecondLargest[2*core:2*core+1] ← findLocalMaxSecondLargest(a, core, numCores)
secondLargest ← findGlobalSecondLargest(localMaxSecondLargest)
return secondLargest

- `localMaxSecondLargest[]` array store the max and second-largest values in each sublists.
- `findLocalMaxSecondLargest()` function find the max and second-largest values of the list.
- `findGlobalSecondLargest()` function find the second-largest value of the list.

4.8

A prime number is a positive integer evenly divisible by exactly two positive integer: itself and 1. The first five prime number are 2, 3, 5, 7, and 11. Sometimes two consecutive odd numbers are both prime. For example, the odd integers following 3,5 and 11 are all prime numbers. However, the odd integer following 7 is not a prime number. Write a parallel program to determine, for all integers less than 100000, the number of times that two consecutive odd integers are both prime.

Answer: We write a parallel program as following steps:

- Firstly, we can find all prime number in n using the parallel program described in the course.

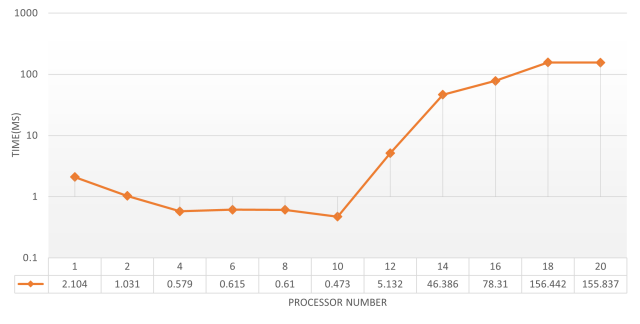

Figure 1: The Cost Time(ms) of the program (4.8) under different number of processes.

- Then, we can go through all the prime number and count the number of two consecutive odd numbers.
- To improve the performance, we can ignore all even numbers.

After implementing the aforementioned steps, the algorithm successfully determined that there are 8,169 twin prime pairs below 100,000. Profiling the algorithm with varying numbers of processors reveals insightful trends. As illustrated in the Figure 1, utilizing either 2 or 4 processes results in a significant speedup—approximately doubling the execution rate compared to a single process. When employing between 4 and 10 processes, the execution time stabilizes, indicating a plateau in performance gains. With more than 10 processes, the execution time begins to exceed that of the single-process scenario. This suggests that the overhead introduced by managing an excessive number of processes negates the benefits of parallelism. This happens maybe because our testing platform only has 6 core with 12 threads in total. Therefore, employing too many processes can lead to inefficient execution times due to the limitation of the actual number of cpu cores on the system.

5.7

Modify the parallel Sieve of Eratosthenes program presented in the book to incorporate the frrst two improvements described in Section 5.9. Your program should not set aside memory for even integers, and each process should use the sequential Sieve of Eratosthenes algorithm on a separate array to find all primes between 3 and $\sqrt{n}$. With this information, the call to MPI_Bcast can be eliminated. Benchmark your program, comparing its performance with that of the original parallel sieve program.

Answer: We write a parallel program as following steps:

- Firstly, we write the basic version of the Sieve of Eratosthenes program presented in the book (**Version 1**).
- Secondly, we refined the program by excising all even numbers from the marking array, effectively reducing the memory footprint and execution time (**Version 2**).

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

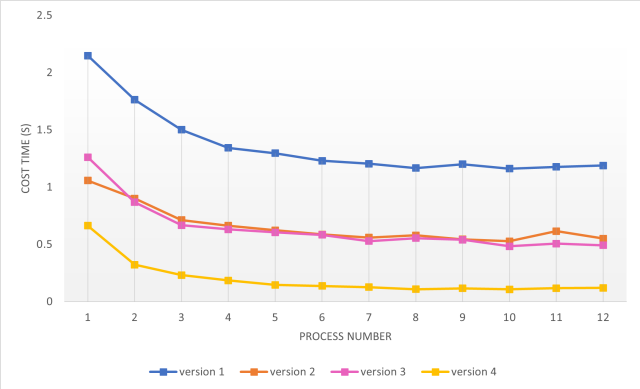

Figure 2: The Cost Time(s) of the Sieve of Eratosthenes program (5.7) under different number of processes with different optimization methods.

- Thirdly, we eliminate the MPI_Bcast by enabling each process to autonomously calculate primes between between 3 and $\sqrt{n}$ (**Version 3**).
- Finally, we also reorganize loops by using cache to improve the hit ratio (**Version 4**).

After implementing the aforementioned steps, we obtain the results shown in the Figure 2 by setting $n = 100000000$ on our platform with 6 cores and 12 threads. From the results, we can see that deleting all even integers can roughly cut in half execution time. However, eliminating the MPI_Bcast has a nominal effect. While reorganizing loops gain huge improvement by improving the cache hit ratio. Therefore, improving the cache hit ratio is a great way to speedup the program.

6.9

Write your own version of MPI_Reduce using functions MPI_Send and MPI_Recv. You may assume that

```
datatype=MPI_INT
operator=MPI_SUM
comm=MPI_COMM_WORLD
```

Compare the performance of your implementation with the "real" MPI_Reduce in your system's MPI library.

Answer: We implement two MPI_Reduce functions using MPI_Send and MPI_Recv with different methods.

- **Method 1**: Every process transmits its data to process 0, which then aggregates all the numbers it has collected.
- **Method 2**: In an effort to minimize the communication overhead for process 0, we employ a tree-based reduction method. Specifically, for the current count of processes N holding data, process i will receive data from process $i + \frac{N}{2}$ and combine it with its own data, subsequently awaiting the next cycle. In each cycle, the count N is halved. If N is an odd number, process 0 does not engage in communication during that cycle. Once N is reduced to 1, process 0 will have obtained the final sum.

With above two methods, we get the results of Figure 3 (**Base** is the "real" MPI_Reduce in the test system with 40 cores and 40 threads). Observing the outcomes, all three methods show similar performance for a smaller number of processes due to the negligible costs, which are not effectively captured by the timer. However, as the process count increases, **Method 1** incurs significantly higher costs ($O(N)$ complexity) compared to the others, since it places a linear communication burden on process 0. Conversely, **Method 2** demonstrates a logarithmic complexity ($O(logN)$), consistently outperforming **Method 1**. **Method 2** also tends to perform better than the **Base**. This could be attributed to the **Base** incurring additional overheads to handle various data types and operations, whereas our implementation is specialized, focusing solely on the `SUM` operation for `INT` datatype.

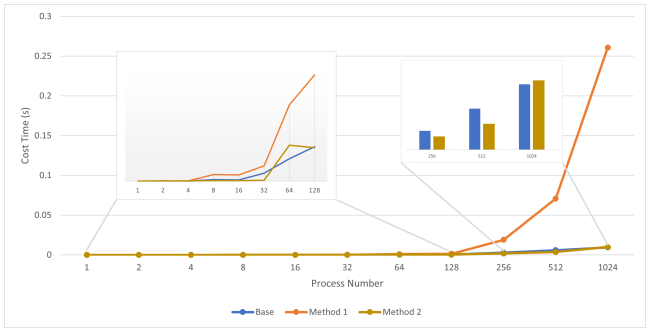

Figure 3: The Cost Time(s) of MPI_Reduce (6.9) under different number of processes with different optimization methods.

7.5

For a problem size if interest, 6 percent of the operations of a parallel program are inside I/O functions that executed on a single processor. What is the minimum number of processors needed in order for the parallel program to exhibit a speedup of 10?

Answer: We can use the Amdahl's Law to solve this problem. Because the the fraction of the program that is sequential is 0.06, we have

$$10 = \frac{1}{0.06 + \frac{1-0.06}{p}}$$

$$p \approx 23.5$$

Therefore, the minimum number of processors needed is 24.

7.8

Brandon's parallel program executes in 242 seconds on 16 processors. Through benchmarking he determines that 9 seconds is spent performing initializations and cleanup on one processor. During the remaining 233 seconds all 16 processors are active. What is the scaled speedup achieved by Brandon's program?

Answer: We use the Gustafson-Barsis's Law to solve the problem. According to the question, we can know the fraction of serial part is $\frac{9}{242}$, so we have

$$S = 16 - (1 - 16)\frac{9}{242} = 15.442$$

Therefore, the scaled speedup is 15.442.