University of California, Riverside
CS 211: High Performance Computing - Project 3
Yuanhang Luo
Nov 19th, 2017

Part 1. Eliminate even numbers

Analysis

All the even numbers are obviously not prime numbers, so it is meaningless to set aside memory space for those even numbers. We can gain higher performance through eliminating even numbers in the algorithm. Actually, there is no big difference between the whole-number version and the odd-number version. The only thing we need to modify is the mapping from the index of the array to the number we need to compute. In the whole-number version, we use the index of the array to indicate the number we need to compute. But, in the odd-number version, the size of the array is reduced by half while the range of the number need to be computed is the same, not be halved. So, we need to change the mapping pattern. The index *i* of the array should presents the number *2*i+1*. Well, we also notice that the number we should compute starts at 3 and we do not need to consider the number 2 because it is well-known that 2 is a prime number. So, the computation only need to start at 3. In addition, another thing we need to note is that the range of number is 10000000000 which exceeds the range of int, so we need to be careful about the type of number when we declare it.

Performance

	1 nodes	2 nodes	4 nodes	8 nodes
Running Time(s)	16.524263	8.723452	7.181070	3.490537
Prime Number	455052511	455052511	455052511	455052511

Part 2. Remove Broadcast

Analysis

In the original method, we use process 0 to compute the next sieving number and broadcast to the other processes. The broadcast operation takes about $\sqrt{n}/\ln\sqrt{n}$ times throughout the whole computation period. We can reduce this part of time by letting each process to computer their own next sieving number. To achieve this approach, we need to allocate another array for each process to save the sieving numbers which is bigger than 2 and less than $\sqrt{n}$. In this

approach, each process need to compute the sieving numbers sequentially. So, each process will gain an array contains all the prime numbers which is from 3 to $\sqrt{n}$. After that, we can use our parallel algorithm to compute rest of the numbers by the local array in each process. This approach can eliminate the broadcast operations and gain higher performance.

Performance

	1 nodes	2 nodes	4 nodes	8 nodes
Running Time(s)	15.916358	8.007149	6.997060	3.428442
Prime Number	455052511	455052511	455052511	455052511

Part 3. Optimizations by Increasing Cache Reuse

Analysis

In the original method, we spend lots of time on marking the numbers in different blocks, so it leads to a very low cache hits rate. Actually, we can do more optimizations on improving the cache hits rate by rearrange the iterations in the algorithm. Take an overview of our original algorithm, it can be considered as two loops: the outer loop is iterated from 3 to $\sqrt{n}$ while the inner loop is iterated among the numbers which is assigned to this process (these numbers range from 3 to n). If we rearrange these two loops, we can gain higher cache hits rate by putting part of numbers in the big array into cache. We can firstly compute the part of numbers in the big array and mark the multiples of the primes which is less than $\sqrt{n}$. Then, we can read next part of numbers into cache and do the next computation. In this way, we can gain higher cache hits rate to achieve better performance.

Find Better Cache Size

After that, we need to find an appropriate cache size. So, I have written a program to find the fine cache size. The results are as follow:

Cache Size	4096	8192	16384	32768	65536
Running Time(s)	56.642320	31.205423	18.438662	12.123553	9.074643
Cache Size	131072	262144	524288	1048576	2097152
Running Time(s)	7.605630	6.775248	6.475376	6.378394	7.481595

So, we can find out the cache size should be 1048576.

Performance

	1 nodes	2 nodes	4 nodes	8 nodes
Running Time(s)	6.679128	3.361577	2.604060	0.858657
Prime Number	455052511	455052511	455052511	455052511

Summary

	1 nodes	2 nodes	4 nodes	8 nodes
Part 1	16.524263s	8.723452s	7.181070s	3.490537s
Part 2	15.916358s	8.007149s	6.997060s	3.428442s
Part 3	6.679128s	3.361577s	2.604060s	0.858657s