CS 211: High Performance Computing Project 3

Zhenyu Yang (862187998)

Fall 2019

Parallel Sieve of Eratosthenes 10^{10}

1.1 sieve0

The basic version is using Eratosthenes algorithm to find all prime within 10^{10}. This parallel algorithm shows that the performance has been improved by increasing the number of processing cores, and the running time is reduced to 50% when the number of processing cores is doubled.

1.2 sieve1

In this version, all even number are removed from original data, because all even number except 2 cannot be a prime. The result shows that all processing time (for different processor number) are decreased to a half of sieve0.

1.3 sieve2

In this version, all each process of the program finds its own sieving primes via local computations instead of broadcasts for saving communications time. The result shows that the processing time does not decrease so much (even cost more) when there is only few processing cores, but it shows relatively decreasing by increasing the process number.

For implementation, there is two way to calculate local primes. The first one is one-step calculation, which means the calculations are done in loops with marking those who is not prime. This implementation share the cache with marking operations, but it only calculates the primes for its own process and is limited by high_value, which means this implementation reduces the loop number.

The second one is all processes calculate all local primes first, then do loop to marking all non-prime numbers. Comparing with first implementation, this one make sure every process doing same amount of works as others for calculating local primes, saving loops for other processes may not help because the performance of a parallel program is decided by the slowest process(usually the last one).

Here is results for sieve2 with different problem size for both implementations.

Table 1: **implementation 1**

cores	10^9	10^{10}	$2 * 10^{10}$
32	1.345676	13.149903	27.425850
64	0.684752	6.769014	25.098223
128	0.419530	3.526987	6.987258
256	0.078973	1.890770	4.127071

Table 2: **implementation 2**

cores	10^9	10^{10}	$2 * 10^{10}$
32	1.361698	25.021929	27.831473
64	0.808290	6.944194	26.397712
128	0.276708	3.448321	7.020489
256	0.075725	1.944055	4.120300

There is no such huge difference between two implementations, for more cache available purpose, the implementation2 are used in sieve3.

1.4 sieve3

To use the idea of cache, blocking is a good method to decrease the cache hit miss rate.
By checking the L2 cache size for tardis, it shows the L2 cache size is **2097152**, I chose **2000000** as final block size.
Here is the cache configuration and command I used to check:

getconf -a — grep LEVEL2_CACHE
LEVEL2_CACHE_SIZE 2097152
LEVEL2_CACHE_LINESIZE 64

The results shows that the performance has been improved almost **4 folds** by increase the cache hit.

1.5 result

First, the results proving the correctness.

sieve0:
The total number of prime: 455052511, total time: 27.548977, total node 32
The total number of prime: 455052511, total time: 26.136182, total node 64
The total number of prime: 455052511, total time: 7.099779, total node 128
The total number of prime: 455052511, total time: 3.609159, total node 256
sieve1:
The total number of prime: 455052511, total time: 13.517305, total node 32
The total number of prime: 455052511, total time: 6.924514, total node 64
The total number of prime: 455052511, total time: 3.616797, total node 128
The total number of prime: 455052511, total time: 3.120928, total node 256
sieve2:
The total number of prime: 455052511, total time: 13.592637, total node 32
The total number of prime: 455052511, total time: 6.769014, total node 64
The total number of prime: 455052511, total time: 3.418407, total node 128
The total number of prime: 455052511, total time: 1.965367, total node 256
sieve3:
The total number of prime: 455052511, total time: 4.767698, total node 32
The total number of prime: 455052511, total time: 2.008644, total node 64
The total number of prime: 455052511, total time: 1.006978, total node 128
The total number of prime: 455052511, total time: 0.537084, total node 256

Table 3: **sieve0**

cores	10^9	10^{10}	$2 * 10^{10}$
32	2.837629	27.548977	57.065409
64	2.053819	26.136182	54.029782
128	0.808047	7.099779	14.476197
256	0.326352	3.609159	7.389089

Table 4: **sieve1**

cores	10^9	10^{10}	$2 * 10^{10}$
32	1.178136	13.517305	27.735075
64	0.802593	6.924514	14.082999
128	0.426171	3.616797	7.066504
256	0.114588	3.120928	3.648910

Table 5: **sieve2**

cores	10^9	10^{10}	$2 * 10^{10}$
32	1.361698	13.592637	27.831473
64	0.808290	6.769014	26.397712
128	0.276708	3.418407	7.020489
256	0.075725	1.965367	4.120300

Table 6: **sieve3**

cores	10^9	10^{10}	$2 * 10^{10}$
32	0.394069	4.767698	8.060845
64	0.222897	2.008644	4.057865
128	0.137524	1.006978	2.136247
256	0.093506	0.537084	1.114526

The result shows that, the performance has been improved 8 folds by increasing processes number from **32 - 256**, and the performance has been improved 7 folds by optimizations from **sieve0 - sieve3**, total in almost **56 folds (27.548977 - 0.537084)**.