CS211 HW3 11/17/2024 Hugo Wan 862180666 twan012

https://github.com/UCR-HPC/cs211-hw3-sieving-prime-HugoWan0504

Q1

Screenshot of the sieve0‘s test outputs:

```
func=sieve0, pnum=32, count=455052511, time=17.709125
func=sieve0, pnum=64, count=455052511, time=8.972398
func=sieve0, pnum=96, count=455052511, time=5.978825
func=sieve0, pnum=128, count=455052511, time=4.495173
func=sieve0, pnum=160, count=455052511, time=3.570440
```

Organized Table of **function sieve0** and **prime count = 455052511**:

Cores	Time (seconds)
32	17.709125
64	8.972398
96	5.978825
128	4.495173
160	3.570440

Q2

Screenshot of the sieve1‘s test outputs:

```
func=sieve1, pnum=32, count=455052511, time=5.586815
func=sieve1, pnum=64, count=455052511, time=3.348253
func=sieve1, pnum=96, count=455052511, time=2.320773
func=sieve1, pnum=128, count=455052511, time=1.721402
func=sieve1, pnum=160, count=455052511, time=1.385505
```

Organized Table of **function sieve1** and **prime count = 455052511**:

Cores	Time (seconds)
32	5.586815
64	3.348253
96	2.320773

128	1.721402
160	1.385505

Sieve1.c codes:

```c
#ifndef __SIEVE1_C__
#define __SIEVE1_C__

#include "include.h"

// Follow instructions based on HW3 guide. Thanks, TA.
void sieve1(unsigned long long *global_count, unsigned long long n, int pnum, int pid)
{
    unsigned long long low_value = 3 + 2 * ((pid * ((n - 3) / 2 + 1)) / pnum); // the smallest value handled by this process
    unsigned long long high_value = 3 + 2 * (((pid + 1) * ((n - 3) / 2 + 1)) / pnum) - 2; // the largest value handled by this process
    unsigned long long size = (high_value - low_value) / 2 + 1; // number of integers handled by this process

    if (1 + (n - 1) / pnum < (int)sqrt((double)n)) // high_value of process 0 should be larger than floor(sqrt(n))
    {
        if (pid == 0)
            printf("Error: Too many processes.\n");
        MPI_Finalize();
        exit(0);
    }

    char *marked = (char*)malloc(size); // array for marking multiples. 1 means multiple and 0 means prime
    if (marked == NULL)
    {
        printf("Error: Cannot allocate enough memory.\n");
        MPI_Finalize();
        exit(0);
    }
    memset(marked, 0, size);
```

```c
    unsigned long long index = 0; // index of current prime among all primes (only works for process 0)
    unsigned long long prime = 3; // current prime broadcasted by process 0
    do {
        unsigned long long first; // index of the first multiple among values handled by this process
        if (prime * prime > low_value)
            first = ((prime * prime) - low_value) / 2;
        else if (low_value % prime == 0)
            first = 0;
        else
            first = ((prime * ((((low_value / prime) + 1) / 2) * 2 + 1)) - low_value) / 2;


        for (unsigned long long i = first; i < size; i += prime)
            marked[i] = 1;

        index++;
        if (pid == 0)
        {
            while (marked[index] == 1)
                index++;
            prime = (index * 2) + 3;
        }
        MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (prime * prime <= n);

    unsigned long long count = 0; // local count of primes
    for (unsigned long long i = 0; i < size; i++)
        if (marked[i] == 0)
            count++;
```

```c
    if (pid == 0) count++; // add prime 2 to the count
    MPI_Reduce(&count, global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // free memory
    free(marked);
}

#endif
```

Q3

Screenshot of the sieve2‘s test outputs:

```
func=sieve2, pnum=32, count=455052511, time=5.639904
func=sieve2, pnum=64, count=455052511, time=3.297908
func=sieve2, pnum=96, count=455052511, time=2.205112
func=sieve2, pnum=128, count=455052511, time=1.745691
func=sieve2, pnum=160, count=455052511, time=1.469822
```

Organized Table of **function sieve2** and **prime count = 455052511**:

Cores	Time (seconds)
32	5.639904
64	3.297908
96	2.205112
128	1.745691
160	1.469822

Sieve2.c codes:

```c
#ifndef __SIEVE2_C__
#define __SIEVE2_C__

#include "include.h"

// Follow instructions based on HW3 guide. Thanks, TA.
void sieve2(unsigned long long *global_count, unsigned long long n, int pnum, int pid)
{
    unsigned long long low_value = 3 + 2 * ((pid * ((n - 3) / 2 + 1)) / pnum); // smallest value handled by this process
    unsigned long long high_value = 3 + 2 * (((pid + 1) * ((n - 3) / 2 + 1)) / pnum) - 2; // largest value handled by this process
    unsigned long long size = (high_value - low_value) / 2 + 1; // number of integers handled by this process

    if (1 + (n - 1) / pnum < (int)sqrt((double)n)) // high_value of process 0 should be larger than floor(sqrt(n))
    {
        if (pid == 0)
            printf("Error: Too many processes.\n");
        MPI_Finalize();
        exit(0);
    }

    char *marked = (char*)malloc(size); // array for marking multiples. 1 means multiple and 0 means prime
    if (marked == NULL)
    {
        printf("Error: Cannot allocate enough memory.\n");
        MPI_Finalize();
        exit(0);
    }
    memset(marked, 0, size);
```

```
    unsigned long long sqrt_n = (unsigned long long)sqrt((double)n);
    unsigned long long sieve_size = (sqrt_n - 3) / 2 + 1; // size for local sieving primes
    char *sieve = (char*)malloc(sieve_size); // array for local sieving primes
    if (sieve == NULL)
    {
        printf("Error: Cannot allocate enough memory for sieve.\n");
        MPI_Finalize();
        exit(0);
    }
    memset(sieve, 0, sieve_size);

    // Generate sieving primes locally
    for (unsigned long long i = 0; i < sieve_size; i++)
    {
        if (sieve[i] == 0) // Found a prime
        {
            unsigned long long prime = 2 * i + 3;
            for (unsigned long long j = (prime * prime - 3) / 2; j < sieve_size; j += prime)
                sieve[j] = 1;
        }
    }
```

```
    // Use sieving primes to mark composites in the range
    for (unsigned long long i = 0; i < sieve_size; i++)
    {
        if (sieve[i] == 0) // Found a sieving prime
        {
            unsigned long long prime = 2 * i + 3;
            unsigned long long first; // First multiple to mark
            if (prime * prime > low_value)
                first = ((prime * prime) - low_value) / 2;
            else if (low_value % prime == 0)
                first = 0;
            else
                first = ((prime * ((((low_value / prime) + 1) / 2) * 2 + 1)) - low_value) / 2;

            for (unsigned long long j = first; j < size; j += prime)
                marked[j] = 1;
        }
    }

    unsigned long long count = 0; // Local count of primes
    for (unsigned long long i = 0; i < size; i++)
        if (marked[i] == 0)
            count++;

    if (pid == 0) count++; // Add prime 2 to the count
    MPI_Reduce(&count, global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    // Free allocated memory
    free(marked);
    free(sieve);
}

#endif
```

Q4

Screenshot of the sieve3‘s test outputs:

```
func=sieve3, pnum=32, count=455052511, time=1.279721
func=sieve3, pnum=64, count=455052511, time=0.635969
func=sieve3, pnum=96, count=455052511, time=0.424961
func=sieve3, pnum=128, count=455052511, time=0.318910
func=sieve3, pnum=160, count=455052511, time=0.253828
```

Organized Table of **function sieve3** and **prime count = 455052511**:

Cores	Time (seconds)
32	1.279721
64	0.635969
96	0.424961
128	0.318910
160	0.253828

Sieve3.c codes:

```c
#ifndef __SIEVE3_C__
#define __SIEVE3_C__

#include "include.h"

// Follow instructions based on HW3 guide. Thanks, TA.
#define BLOCK_SIZE 1024 * 32  // Based on cache optimization hints

void sieve3(unsigned long long *global_count, unsigned long long n, int pnum, int pid)
{
    unsigned long long sqrt_n = (unsigned long long)sqrt((double)n);

    // Step 1: Calculate local primes up to sqrt(n)
    unsigned long long local_sieve_size = (sqrt_n - 3) / 2 + 1; // Only odd numbers up to sqrt(n)
    char *local_sieve = (char *)malloc(local_sieve_size);
    if (!local_sieve)
    {
        printf("Error: Unable to allocate memory for local sieve array.\n");
        MPI_Finalize();
        exit(0);
    }
    memset(local_sieve, 0, local_sieve_size);

    // Generate local primes
    for (unsigned long long i = 0; i < local_sieve_size; i++)
    {
        if (local_sieve[i] == 0)
        {
            unsigned long long prime = 2 * i + 3;
            for (unsigned long long j = (prime * prime - 3) / 2; j < local_sieve_size; j += prime)
                local_sieve[j] = 1;
        }
    }
```

```
    // Store local primes for marking later
    unsigned long long *local_primes = (unsigned long long *)malloc(local_sieve_size * sizeof(unsigned long long));
    unsigned long long local_prime_count = 0;
    for (unsigned long long i = 0; i < local_sieve_size; i++)
    {
        if (local_sieve[i] == 0)
            local_primes[local_prime_count++] = 2 * i + 3;
    }
    free(local_sieve);

    // Step 2: Define the range for this process
    unsigned long long low_value = 3 + 2 * ((pid * ((n - 3) / 2 + 1)) / pnum);
    unsigned long long high_value = 3 + 2 * (((pid + 1) * ((n - 3) / 2 + 1)) / pnum) - 2;
    unsigned long long size = (high_value - low_value) / 2 + 1;

    // Allocate memory for marking composites
    char *marked = (char *)malloc(BLOCK_SIZE);
    if (!marked)
    {
        printf("Error: Unable to allocate memory for marked array.\n");
        MPI_Finalize();
        exit(0);
    }

    unsigned long long count = 0;

    // Step 3: Block processing for cache efficiency
    for (unsigned long long block_low = low_value; block_low <= high_value; block_low += 2 * BLOCK_SIZE)
    {
        unsigned long long lon  unsigned long long high_value  BLOCK_SIZE - 1;
        if (block_high > high_value)
            block_high = high_value;

        unsigned long long block_size = (block_high - block_low) / 2 + 1;
        memset(marked, 0, block_size);

        // Mark multiples of local primes in the current block
        for (unsigned long long i = 0; i < local_prime_count; i++)
        {
            unsigned long long prime = local_primes[i];
            unsigned long long first;

            if (prime * prime > block_low)
                first = (prime * prime - block_low) / 2;
            else if (block_low % prime == 0)
                first = 0;
            else
            {
                unsigned long long temp = (block_low + prime) / prime;
                if (temp % 2 == 0)
                    temp++;
                first = (temp * prime - block_low) / 2;
            }

            for (unsigned long long j = first; j < block_size; j += prime)
                marked[j] = 1;
        }

        // Count primes in the current block
        for (unsigned long long i = 0; i < block_size; i++)
            if (marked[i] == 0)
                count++;
    }

    // Include prime 2 in the count
    if (pid == 0) count++;

    // Step 4: Reduce the counts across all processes
    MPI_Reduce(&count, global_count, 1, MPI_UNSIGNED_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

    free(marked);
    free(local_primes);
}

#endif
```

Summary Tables of Execution Time (in seconds) & Speedup:

Execution Time Comparison:

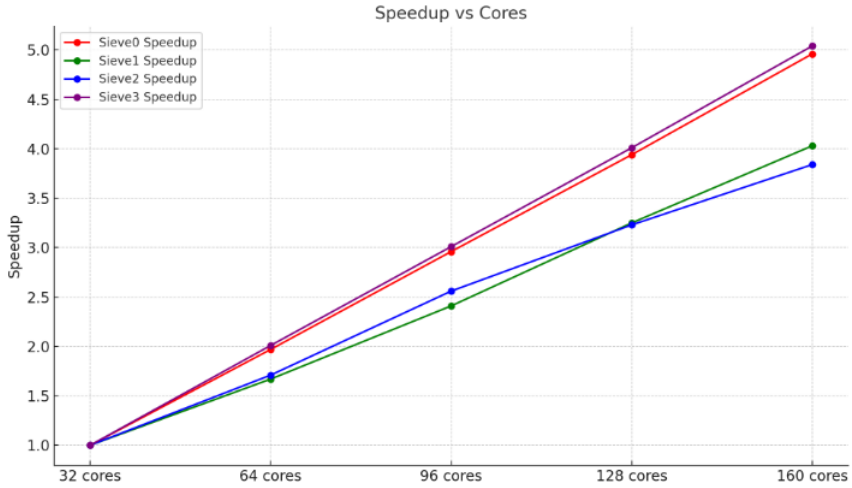
Cores	Sieve0 Time	Sieve1 Time	Sieve2 Time	Sieve3 Time
32	17.709125	5.586815	5.639904	1.279721
64	8.972398	3.348253	3.297908	0.635969
96	5.978825	2.320773	2.205112	0.424961
128	4.495173	1.721402	1.745691	0.318910
160	3.570440	1.385505	1.469822	0.253828

Speedup Comparison:

Cores	Sieve0 Speedup	Sieve1 Time Speedup	Sieve2 Time Speedup	Sieve3 Time Speedup
32	1.00	1.00	1.00	1.00
64	1.97	1.67	1.71	2.01
96	2.96	2.41	2.56	3.01
128	3.94	3.25	3.23	4.01
160	4.96	4.03	3.84	5.04

Slope of Speedups Comparison:

Sieve Version	Slope of Speedups
Sieve0	0.0309
Sieve1	0.0239
Sieve2	0.0225
Sieve3	0.0315



In conclusion:

Sieve0 is the best when it comes to scaling with more cores, making it a great choice for larger parallel computing tasks. Sieve3 works really well with fewer cores but doesn't improve as much when more cores are added, so it might need some tweaks to handle bigger systems. Sieve1 and Sieve2 give decent performance overall, but they are not as scalable as Sieve0 or as fast as Sieve3 on smaller setups. In short, Sieve0 is great for scalability, Sieve3 is better for smaller systems, and Sieve1 and Sieve2 are balanced but could be improved.