MPI

```
int main(int argc, char *argv[])
{

    FILE *fpt; // file pointer to write results to csv file
    fpt = fopen("resultsMPI.csv", "a+");

    int mypid;
    int size;
    int intervalSize[9] = {1000, 5000, 10000, 50000, 100000, 500000, 1000000,
10000000, 100000000};

    double time_taken;
    double t;

    MPI_Init(&argc, &argv); /* starts MPI */

    MPI_Comm_rank(MPI_COMM_WORLD, &mypid); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (mypid == 0)
    {
        fprintf(fpt, "M, T1(ms), T2(ms), T4(ms), T6(ms), S2, S4, S6\n");
    }

    for (int i = 0; i < 9; i++)
    {
        if (mypid == 0)
        {

            t = currentTimeMillis();
        }

        parallel(mypid, size, intervalSize[i]);

        if (mypid == 0)
        {
            t = currentTimeMillis() - t;
            printf("Time taken for parallel: %f\n", t);
            fprintf(fpt, "%f \n", t);
        }
    }

    MPI_Finalize();

    fclose(fpt);
    return 0;
}
```

The main code basically does initializes MPI and calls parallel function which does the main job. Also time measurement is implemented there.

```
double currentTimeMillis() // function to get current time in milliseconds
{
    struct timeval time;
    gettimeofday(&time, NULL);
    double s1 = (time.tv_sec) * 1000;
    double s2 = ((double)time.tv_usec / 1000);
    return s1 + s2;
}
```

Time measurement function.

```
int compare(const void *a, const void *b) // compare function for sorting
{
    return (*(int *)a - *(int *)b);
}
```

Helper function for sorting.

```
void parallel(int mypid, int size, int N)
{

    int *prime;
    int *localPrime = (int *)malloc((((N / 2) + 1) / size) * sizeof(int));
    // int localPrime[((N / 2) + 1) / size];

    int j;
    int k;
    int n;
    int quo, rem;
    int serialN;
    int broken = 0;
    int l;
    int totalCount = 0;
    prime = (int *)malloc(((N / 2) + 1) * sizeof(int));

    if (mypid == 0)
    {

        prime[0] = 2;
        n = 3;
        j = 1;
        prime[j] = n;
        serialN = sqrt(N);
        for (n = 5; n <= N; n += 2)
        {
            k = 1;
            quo = n / prime[k];
            rem = n % prime[k];
            if (rem != 0)
```

```
            {
                while (quo > prime[k])
                {
                    k += 1;
                    quo = n / prime[k];
                    rem = n % prime[k];
                    if (rem == 0)
                    {
                        broken = 1;
                        break;
                    }
                }
                if (broken == 0)
                {
                    j += 1;
                    prime[j] = n;
                    if (n > serialN)
                    {
                        break;
                    }
                }
                else
                {
                    broken = 0;
                }
            }
        }
        broken = 0;
        l = n + 2;
    }
    MPI_Bcast(&l, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&j, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(prime, j + 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    int count = 0;
    for (int a = (l + 2 * mypid); a < N; a += size * 2)
    {
        k = 1;
        quo = a / prime[k];
        rem = a % prime[k];
        if (rem != 0)
        {
            while (quo > prime[k])
            {
                k += 1;
                quo = a / prime[k];
                rem = a % prime[k];
                if (rem == 0)
```

```
                {
                    broken = 1;
                    break;
                }
            }
            if (broken == 0)
            {

                localPrime[count] = a;
                count += 1;
            }
            else
            {
                broken = 0;
            }
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);

    int *all_counts;
    if (mypid == 0)
    {
        all_counts = (int *)malloc(size * sizeof(int));
    }
    MPI_Gather(&count, 1, MPI_INT, all_counts, 1, MPI_INT, 0, MPI_COMM_WORLD);

    int *displs;
    if (mypid == 0)
    {
        displs = (int *)malloc(size * sizeof(int));
        displs[0] = 0;
        for (int i = 1; i < size; i++)
        {
            displs[i] = displs[i - 1] + all_counts[i - 1];
            totalCount += all_counts[i - 1];
        }
        totalCount += all_counts[size - 1];
    }

    MPI_Gatherv(localPrime, count, MPI_INT, prime + j + 1, all_counts, displs,
MPI_INT, 0, MPI_COMM_WORLD);
    /*
    if (mypid == 0)
    {
        qsort(prime, j + totalCount + 1, sizeof(prime[0]), compare);
        for (int i = 0; i < j + totalCount + 1; i++)
        {
            printf("%d\n", prime[i]);
        }
```

```
    }
    */
    free(localPrime);
}
```

First part is same as OpenMpi project. To give true results, I implemented a serial part. After serial part, the number of found prime numbers, the smallest number which will be executed in parallel part and found prime numbers list is broadcasted to parallel processes. With same way, the prime numbers are generated. To ensure that all processes finished their parts, I put barrier function. The important difference here is that each process have different prime number list which is called as localPrime. Each localPrime array has different number of primes. Therefore, I used gatherv function which enables us to different sized arrays to be gathered. Before it, I created a list for number of primes generated by each process and these are gathered with gather function. Displs array calculates displacement for addresses of elements on the list.

M, T1(ms), T2(ms), T4(ms), T6(ms), S2, S4, S6
1000, 0.022949, 0.035889, 0.058105, 0.076904, 0.639444, 0.394957, 0.298411
5000, 0.092041, 0.076904, 0.039795, 0.041016, 1.196830, 2.312879, 2.244027
10000, 0.204834, 0.177979, 0.083984, 0.087158, 1.150889, 2.438965, 2.350146
50000, 1.660889, 0.854980, 0.482910, 0.523926, 1.942606, 3.439334, 3.170083
100000, 3.461182, 1.790771, 0.967041, 1.424072, 1.932789, 3.579147, 2.430482
500000, 27.615234, 14.094971, 6.786865, 10.545898, 1.959226, 4.068923, 2.618576
1000000, 64.648926, 37.979004, 16.723145, 24.414062, 1.702228, 3.865835, 2.648020
10000000, 1314.414062, 702.841064, 386.466797, 436.792969, 1.870144, 3.401105, 3.009238
100000000, 27316.297852, 15617.278076, 8575.965088, 9754.791016, 1.749107, 3.185216, 2.800296

I was not able to calculate with 8 threads in Mpi due to hardware limitations of my computer. Therefore I tested 6 threads instead. In 100.000.000 case the best result belongs to 4 threads. These results are generated via makeFile. In Mpi, there is no possibility to set threads in the code as I researched. Therefore I wrote a makeFile which runs compiled code with different threads.

- make

These commands enable user to run code files. Parallelizing with one thread or two threads give distinct result as expected. However, there is no such distinction between four threads and six threads. Surprisingly, 4 threads give better performance generally. It might be resulted as this due to more difficult of parallelization with 6 threads. Additionally, user can run code with these commands to test with one specific thread number.

- mpicc -o ./a.out mpi.c -lm
- mpirun ./a.out -n <NUMBER OF THREADS>

You can see the results from resultsMPI1.csv. **If you want to rerun, you must delete *resultsMPI.csv* file if it exists.**

THRUST

```
int main()
{
    FILE *fpt; // file pointer to write results to csv file
    fpt = fopen("resultsThrust.csv", "w+");
    fprintf(fpt, "M, T1(ms), T2(ms), T4(ms), T8(ms), S2, S4, S8\n");

    int intervalSize[7] = {1000, 10000, 50000, 100000, 1000000, 10000000,
100000000};
    int threads[4] = {1, 2, 4, 8};
    for (int i = 0; i < 7; i++)
    {
        omp_set_num_threads(1);
        double start = currentTimeMillis();
        parallel(intervalSize[i]);
        double end = currentTimeMillis();
        double time_taken1 = end - start;

        omp_set_num_threads(2);
        double start1 = currentTimeMillis();
        parallel(intervalSize[i]);
        double end1 = currentTimeMillis();
        double time_taken2 = end1 - start1;

        omp_set_num_threads(4);
        double start2 = currentTimeMillis();
        parallel(intervalSize[i]);
        double end2 = currentTimeMillis();
        double time_taken3 = end2 - start2;

        omp_set_num_threads(8);
        double start3 = currentTimeMillis();
        parallel(intervalSize[i]);
        double end3 = currentTimeMillis();
        double time_taken4 = end3 - start3;

        fprintf(fpt, "%d, %f, %f, %f, %f, %f, %f, %f\n", intervalSize[i],
time_taken1, time_taken2, time_taken3, time_taken4, time_taken1 / time_taken2,
time_taken1 / time_taken3, time_taken1 / time_taken4);
    }
    fclose(fpt);

    return 0;
}
```

As in Mpi part, in the main code initializations and time measurement is implemented. The critical difference at this, I can set threads in code file, so multiple running is not necessary.

```
void parallel(int N)

{

    thrust::device_vector<int> prime;
    ;

    int j;
    int k;
    int n;
    int quo, rem;
    int serialN;
    int broken = 0;
    int l;
    int totalCount = 0;

    prime.push_back(2);
    n = 3;
    j = 1;
    prime.push_back(n);
    serialN = sqrt(N);

    for (n = 5; n <= N; n += 2)
    {
        k = 1;
        quo = n / prime[k];
        rem = n % prime[k];

        if (rem != 0)
        {
            while (quo > prime[k])
            {
                k += 1;
                quo = n / prime[k];
                rem = n % prime[k];

                if (rem == 0)
                {
                    broken = 1;
                    break;
                }
            }

            if (broken == 0)
            {
                j += 1;
                prime.push_back(n);

                if (n > serialN)
```

```
                {
                    break;
                }
            }
            else
            {
                broken = 0;
            }
        }
    }

    broken = 0;
    l = n + 2;

    thrust::device_vector<int> numbers(N / 2);

    thrust::copy(thrust::device, prime.begin(), prime.end(), numbers.begin());

    thrust::sequence(numbers.begin() + j + 1, numbers.end(), l, 2);
    thrust::remove_if(thrust::device, numbers.begin() + j + 1, numbers.end(),
is_prime_functor(numbers));

    /*
    std::cout << "Sorted \n";
    int ind = 0;
    while (numbers[ind] < N)
    {
        std::cout << numbers[ind] << " ";
        ind++;
    }
    */
}
```

In this part I implemented serial part as Mpi. Then, new device vector is created for parallelization. Also, the found primes are copied and, odd new numbers are added. At the functor, if the number is not prime it is deleted from the list.

```
struct is_prime_functor
{
    thrust::device_vector<int> primes;

    __host__ __device__
    is_prime_functor(thrust::device_vector<int> &_primes) : primes(_primes) {}

    __host__ __device__ bool operator()(int n)

    {
        int k = 1;
        int quo = n / primes[k];
        int rem = n % primes[k];
        int broken = 0;

        if (rem != 0)
        {
            while (quo > primes[k])
            {
                k += 1;
                quo = n / primes[k];
                rem = n % primes[k];

                if (rem == 0)
                {
                    broken = 1;
                    return true;
                }
            }

            if (broken == 0)
            {
                return false;
            }
        }
        return true;
    }
};
```

The only difference from sequential part here, I add Boolean to indicate number is prime or not. The code can be compiled and run with these commands:

- g++ -O2 -o saxpy thrust.cpp -fopenmp -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_OMP -lgomp -I /usr/local/cuda-12.3/targets/x86_64-linux/include
- ./saxpy

You can see the results from resultsThrust.csv

M, T1(ms), T2(ms), T4(ms), T8(ms), S2, S4, S8

1000, 0.052979, 0.080078, 0.105957, 0.166992, 0.661585, 0.500000, 0.317251

10000, 0.427979, 0.249023, 0.242188, 0.324951, 1.718627, 1.767137, 1.317055

50000, 1.804932, 1.193115, 0.717041, 0.813965, 1.512789, 2.517194, 2.217457

100000, 3.582031, 1.883057, 1.202148, 1.212891, 1.902243, 2.979691, 2.953301

1000000, 43.010986, 33.501953, 15.833984, 11.228027, 1.283835, 2.716372, 3.830681

10000000, 681.812988, 421.891113, 246.100830, 180.712158, 1.616088, 2.770462, 3.772923

100000000, 13239.718994, 8544.338867, 4958.271973, 3241.725098, 1.549531, 2.670228, 4.084158

These results are better compared to Mpi. Additionally, I could use 8 threads there. While the number of threads increases, the time is less. Speedup also increases in all cases.