1.O PROBLEMA

1. Crivo de Eratóstenes

O Crivo de Eratóstenes é um algoritmo clássico para encontrar todos os números primos até um dado número (n). A ideia básica é marcar iterativamente os múltiplos de cada número primo a partir de 2.

Passos:

1. **Inicialização:**
   - Crie um array `prime` de tamanho (n+1) e inicialize todos os elementos como `1` (indicando que todos os números são inicialmente assumidos como primos).
   - Defina `prime[0]` e `prime[1]` como `0`, pois 0 e 1 não são números primos.
2. **Processo do Crivo:**
   - Começando pelo primeiro número primo (2), marque todos os seus múltiplos como não primos (defina como `0`).
   - Passe para o próximo número e repita o processo. O próximo número não marcado é o próximo primo.
   - Isso continua até a raiz quadrada de (n), pois qualquer número não primo maior que (\sqrt{n}) já teria sido marcado por um de seus fatores.
3. **Resultado:**
   - Após completar o crivo, o array `prime` terá `1` nas posições correspondentes aos números primos e `0` nas outras posições.

2.ALGORITMOS

2.1. ALGORITMO SEQUENCIAL E COM OPENMP

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <time.h>

int num_threads = 4;

void crivo_sequential(int n, int *prime) {
    for (int i = 0; i <= n; i++)
        prime[i] = 1;

    prime[0] = prime[1] = 0;

    for (int p = 2; p <= sqrt(n); p++) {
        if (prime[p] == 1) {
            for (int i = p * p; i <= n; i += p)
                prime[i] = 0;
        }
    }
```

```
}

void crivo_openmp(int n, int *prime) {
    for (int i = 0; i <= n; i++)
        prime[i] = 1;

    prime[0] = prime[1] = 0;

    int sqrt_n = (int)sqrt(n);

    #pragma omp parallel num_threads(num_threads)
    {
        int id = omp_get_thread_num();
        int p;

        for (p = 2 + id; p <= sqrt_n; p += num_threads) {
            if (prime[p] == 1) {
                for (int i = p * p; i <= n; i += p)
                    prime[i] = 0;
            }
        }
    }
}

void print_primes(int n, int *prime) {
    for (int i = 2; i <= n; i++) {
        if (prime[i] == 1) {
            printf("%d ", i);
        }
    }
    printf("\n");
}

int main() {
    int n = 100000000;
    int *prime_seq = (int *)malloc((n + 1) * sizeof(int));
    int *prime_omp = (int *)malloc((n + 1) * sizeof(int));
    double start, end, time_seq, time_omp;

    // sequencial
    start = omp_get_wtime();
    crivo_sequential(n, prime_seq);
    end = omp_get_wtime();
    time_seq = end-start;
    printf("Tempo sequencial: %f seconds\n", time_seq);
    //print_primes(n, prime_seq); // descomentar para printar os primos

    // OpenMP
    start = omp_get_wtime();
    crivo_openmp(n, prime_omp);
    end = omp_get_wtime();
    time_omp = end-start;
    printf("Tempo OpenMP (%d threads): %f seconds\n", num_threads, time_omp);
    //print_primes(n, prime_omp); // descomentar para printar os primos

    printf("Speedup: %f\n", time_seq/time_omp);
    printf("Eficiencia: %f\n", (time_seq/time_omp)/num_threads);
```

```
    // Verificar se as implementacoes dao o mesmo resultado
    int results_match = 1;
    for (int i = 2; i <= n; i++) {
        if (prime_seq[i] != prime_omp[i]) {
            results_match = 0;
            printf("Mismatch at %d: Sequencial = %d, OpenMP = %d\n", i, prime_seq[i],
prime_omp[i]);
            break;
        }
    }

    if (results_match) {
        printf("Os resultados sao iguais!\n");
    } else {
        printf("Os resultados sao diferentes.\n");
    }

    free(prime_seq);
    free(prime_omp);

    return 0;
}
```

2.2. ALGORITMO PARALELO COM OPENMPI

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

void sieve_mpi(int n, int rank, int size) {
    int sqrt_n = (int)sqrt(n);
    int block_size = (n - sqrt_n) / size + 1;

    int *prime = (int *)malloc((n+1) * sizeof(int));
    int *block_prime = (int *)malloc(block_size * sizeof(int));

    for (int i = 0; i <= n; i++)
        prime[i] = 1;

    prime[0] = prime[1] = 0;

    if (rank == 0) {
        for (int p = 2; p <= sqrt_n; p++) {
            if (prime[p] == 1) {
                for (int i = p * p; i <= n; i += p)
                    prime[i] = 0;
            }
        }
    }
```

```
    MPI_Bcast(prime, n + 1, MPI_INT, 0, MPI_COMM_WORLD);

    int start = sqrt_n + 1 + rank * block_size;
    int end = start + block_size - 1;

    if (end > n)
        end = n;

    for (int i = start; i <= end; i++)
        block_prime[i - start] = prime[i];

    for (int p = 2; p <= sqrt_n; p++) {
        if (prime[p] == 1) {
            for (int i = p * p; i <= n; i += p) {
                if (i >= start && i <= end)
                    block_prime[i - start] = 0;
            }
        }
    }

    MPI_Gather(block_prime, block_size, MPI_INT, prime + sqrt_n + 1, block_size, MPI_INT, 0,
MPI_COMM_WORLD);

    free(prime);
    free(block_prime);
}

int main(int argc, char *argv[]) {
    int n = 100000000;
    int rank, size;
    double start, end;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    start = MPI_Wtime();
    sieve_mpi(n, rank, size);
    end = MPI_Wtime();

    if (rank == 0) {
        printf("MPI Time: %f seconds\n", end - start);
    }

    MPI_Finalize();

    return 0;
}
```

3. ANÁLISE DE DESEMPENHO

3.1. RESULTADOS

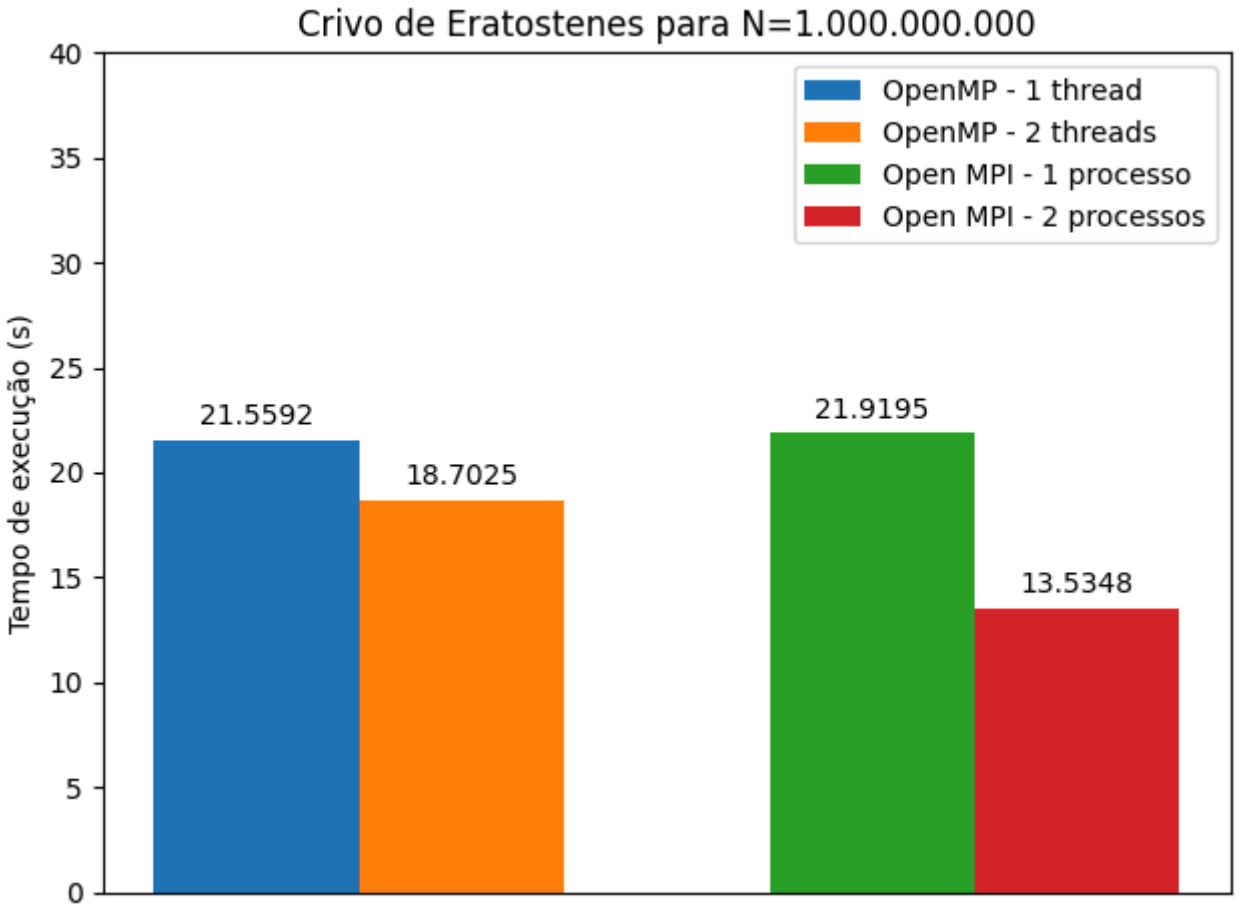
Algoritmo	Quantidade de elementos	Tempo (segundos)	Speedup	Comentários
OpenMP (1 thread)	1.000.000.000	4.086698	-	
OpenMP (4 threads)	1.000.000.000	1.852935	2.20X	Há speedup apesar da máquina virtual ter apenas 3 núcleos, suspeito que seja por causa de otimizações na biblioteca
OpenMP (2 threads)	1.000.000.000	2.829379	1.44x	
MPI (1 processo)	1.000.000.000	1.722884	-	
MPI (2 processos)	1.000.000.000	0.979400	1.77x	
MPI (4 processos)	1.000.000.000	6.756528	0,22x	Suspeito que a perda de eficiência seja causada pela máquina virtual ter apenas 3 núcleos

```
concorrente@concorrente:~/codigo$ mpiexec -np 1 ./crivo_mpi.exe
MPI Time: 1.722884 seconds
```

```
concorrente@concorrente:~/codigo$ mpiexec -np 2 ./crivo_mpi.exe
MPI Time: 0.979400 seconds
```

```
concorrente@concorrente:~/codigo$ mpiexec -np 4 ./crivo_mpi.exe
MPI Time: 6.756528 seconds
```

3.2. GRÁFICOS



4.HARDWARE E SOFTWARE

HARDWARE/SOFTWARE	MODELO
CPU	AMD Ryzen 7 7700X 8-Core Processor 4.50 GHz
RAM	A quantidade padrão da máquina virtual da disciplina | DDR5 5600MHz
SISTEMA OPERACIONAL	Debian disponibilizado para a disciplina
LINGUAGEM DE PROGRAMAÇÃO	C
COMPILADOR	gcc 11.2 https://bigsearcher.com/mirrors/gcc/releases/gcc-11.2.0/
MPI	MPI 4.1.1 https://www.open-mpi.org/software/ompi/v4.1/

5. IMPLEMENTAÇÃO

https://github.com/Casalbe/CrivoParalelo

6. DIFICULDADES ENCONTRADAS

Durante a implementação não tive muitos problemas além do básico de fazer o código funcionar, etc. Porém, hoje, dia 24/09/2024, ao testar o código antes de ir pra faculdade apresentar o projeto, não conseguia mais compilar a versão MPI do código por algum motivo, por sorte eu ainda tinha uma versão compilada antiga dele