Universidade Federal
de São João del-Rei

Trabalho Prático 1 - Computação Paralela

Arthur Antunes Santos Silva
Lucas Gonçalves Nojiri

Trabalho Prático 1 da disciplina de Computação Paralela, ministrada pelo professor Rafael Sachetto Oliveira, do curso de Ciência da Computação da Universidade Federal de São João del-Rei.

São João del Rei

Outubro de 2023

Sumário

1 Introdução

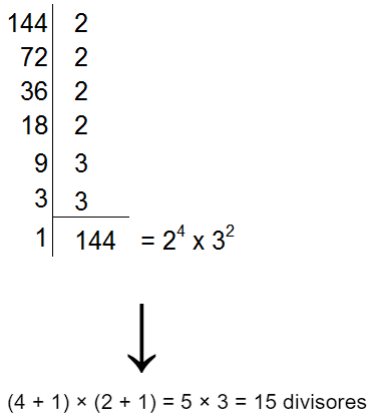
Este relatório descreve o processo de paralelização de um algoritmo de identificação de números primos em um vetor de inteiros. O objetivo deste trabalho é implementar uma versão paralela mestre-escravo desse algoritmo usando o ambiente MPI (Message Passing Interface). O algoritmo lê um vetor de inteiros a partir de um arquivo de entrada (entrada.txt), armazena-o na memória do processo mestre e, em seguida, distribui a tarefa de verificar divisores exatos para processos escravos. Após o processamento, cada escravo retorna a contagem de divisores de cada valor na sua fatia do vetor. O processo mestre, por sua vez, gera um arquivo de saída (saida.txt) contendo o número de divisores de cada elemento do arquivo de entrada na ordem original.

O objetivo final é demonstrar como a paralelização pode ser uma abordagem eficaz para acelerar a identificação de números primos em um grande conjunto de dados, aproveitando ao máximo os recursos computacionais disponíveis.

2 Perfil de desempenho sequencial

Primeiramente foi implementado um algoritmo sequencial. Nesse algoritmo, é lido um arquivo de entrada e carregado os valores contidos nele na memória principal, armazenando-os em um vetor de tamanho S, que representa o número de linhas do arquivo. Em seguida, é chamada uma função que analisa cada valor N presente no vetor. Para cada N, é percorrido os números de 1 até N/2, verificando quais deles são divisores de N. Durante essa verificação, um contador é inicializado em 1, uma vez que todos os números são divisíveis por eles mesmos.

Como esperado, a execução do programa com essa abordagem sequencial é notavelmente lenta. Com isso em mente, antes de avançar com a paralelização, foi implementado uma função dinâmica para fins comparativos a função "Fatora_Prod". Nessa função, é feito a fatoração de N em seus fatores primos e calculado o produto dos expoentes desses fatores primos, somando 1 a cada um deles. Essa técnica nos permite encontrar o número de divisores de N, como exemplificado no caso de 144, que possui 15 divisores.



3 Paralelização

3.1 Identificação das oportunidades de paralelização

A oportunidade da paralelização é vista em cada um dos itens listados

Leitura do arquivo de entrada: A leitura do arquivo de entrada "entrada.txt"é feita apenas pelo processo mestre. Essa operação de leitura pode ser paralelizada, distribuindo a carga de leitura entre os processos escravos, especialmente se o arquivo for muito grande.

Verificação de números primos: A função EhPrimo verifica se um número é primo. Essa verificação pode ser paralelizada entre os processos escravos, cada um verificando um subconjunto dos números no vetor. Os resultados podem ser combinados posteriormente.

Cálculo de divisores: A função Divisores calcula o número de divisores de um número. Assim como a verificação de números primos, esse cálculo pode ser paralelizado, distribuindo o cálculo entre os processos escravos.

Escrita no arquivo de saída: A escrita no arquivo de saída "saida.txt"também é feita apenas pelo processo mestre. Essa operação de escrita pode ser paralelizada, permitindo que cada processo escravo escreva sua parte dos resultados no arquivo de saída.

```
arthur99@arthur99-Aspire-A515-51G:~/Documentos/Computação Paralela$ mpirun -np 1 ./tp1_mpi
Tempo de execução: 1.086165 segundos
arthur99@arthur99-Aspire-A515-51G:~/Documentos/Computação Paralela$ mpirun -np 2 ./tp1_mpi
Tempo de execução: 0.539745 segundos
arthur99@arthur99-Aspire-A515-51G:~/Documentos/Computação Paralela$ mpirun -np 3 ./tp1_mpi
Tempo de execução: 0.362025 segundos
arthur99@arthur99-Aspire-A515-51G:~/Documentos/Computação Paralela$ mpirun -np 4 ./tp1_mpi
Tempo de execução: 0.283880 segundos
```

Figura 1: Execução Algoritmo Paralelo

```
89567853 - 16
56162027 - 4
89110884 - 24
211394401 - 4
147787399 - 2
212490004 - 24
146971856 - 10
172540375 - 32
143159597 - 16
60938341 - 4
127732604 - 12
92667390 - 16
64951198 - 8
85827985 - 8
230744402 - 16
241069066 - 8
242311962 - 8
140723750 - 40
76113752 - 32
97625433 - 8
81198749 - 16
240917528 - 16
217290668 - 48
198979880 - 32
134824578 - 48
126438238 - 16
186583840 - 48
223553113 - 8
181993941 - 6
Tempo de execução -> Fatora_Prod: 21.015625 segundosarthur@DESKTOP-30G3KEK:/mnt/c/Users/Arthur/Documents/UFSJ/Computação Paralela$
```

Figura 2: Execução Algoritmo Sequencial

3.2 Avaliação dos ganhos com a paralelização

Todos os testes foram executados em duas máquinas diferentes. A primeira com processador Intel Core I5-8250U, 4 núcleos de processamento de 2.50GHz, sistema operacional de 64 bits e memória RAM de 8 GB. E a segunda com processador Ryzen 5 5600g, 6 núcleos de processamento de 3.9GHz, sistema operacional de 64 bits e memória RAM de 16 GB. Em ambas as máquinas não foram encontrados resultados muito diferentes.

Algoritmo Sequencial	Tempo de execução em segundos
Teste 1	22.367823s
Teste 2	21.015625s
Teste 3	20.897621s
Teste 4	21.536489s
Teste 5	22.846241s

Figura 3: Execução Algoritmo Sequencial

Algoritmo Paralelizado	Tempo de execução em segundos
Teste 1 - 1 núcleo	1.251572s
Teste 2 - 2 núcleos	0.535188s
Teste 3 - 3 núcleos	0.355845s
Teste 4 - 4 núcleos	0.263418s

Figura 4: Execução Algoritmo Paralelo

Como se pode observar o Algoritmo Paralelo trás um benefício de desempenho muito maior que o Sequencial. Foi percebido também como a quantidade de núcleos de processamento alterou o resultado final, já que quanto mais núcleos melhor foi a performance final do Algoritmo Paralelo.

4 Algoritmo

Implementação Paralela Mestre-Escravo: O principal objetivo do código é implementar uma versão paralela usando o modelo mestre-escravo com MPI. O mestre é responsável por coordenar o trabalho dos processos escravos.

Identificação de Números Primos: O programa visa identificar números primos em um vetor de N inteiros. Isso é feito na função EhPrimo, que verifica se um número é primo.

Cálculo de Divisores: O programa calcula o número de divisores para cada número não primo no vetor. Isso é feito na função Divisores.

Leitura e Escrita de Arquivos: O código lê um arquivo de entrada chamado "entrada.txt,"realiza o processamento paralelo e escreve os resultados em um arquivo de saída chamado "saida.txt."

Avaliação do Desempenho Paralelo: O código mede o tempo de execução no processo mestre e relata os ganhos de desempenho obtidos com a paralelização.

4.1 Identificação de Números Primos

```
1     int EhPrimo(int num) {
2     if (num <= 1) {
3     // Se for menor ou igual a 1, não é primo.
4         return 0;
5     }
6     for (int i = 2; i * i <= num; i++) {
7         if (num % i == 0) {
8         // Se for divisível por algum número, não é primo.
9             return 0;
10        }
11    }
12    return 1; // É primo.
13 }
```

4.2 Cálculo de Divisores

```
    int Divisores(int num) {
    int count = 1;
    for (int i = 1; i <= sqrt(num); i++) {
        if (num % i == 0) {
        // Incrementa o contador para cada divisor encontrado.
            count += 2;
        }
    }
    if (sqrt(num) == (int)sqrt(num)) {
    // Reduz 1 se o número for quadrado perfeito.
        count--;
    }
    return count;
}
```

4.3 Leitura e Escrita de Arquivos

```
MPI_File file;
MPI_File_open(MPI_COMM_WORLD,
"entrada.txt",
MPI_MODE_RDONLY,
MPI_INFO_NULL,
&file);

// ...

MPI_File out_file;
MPI_File_open(MPI_COMM_WORLD,
"saida.txt",
MPI_MODE_CREATE | MPI_MODE_WRONLY,
MPI_INFO_NULL, &out_file);

```

4.4 Avaliação do Desempenho Paralelo

```
1 if (process_rank == MESTRE) {
2     printf("Tempo de execução: %f segundos\n",
3     end_time - start_time);
4 }
5
```

5 Conclusão

Neste trabalho, foi realizada a paralelização de um algoritmo de identificação de números primos em um vetor de inteiros utilizando a biblioteca MPI. O algoritmo foi originalmente sequencial, lento e ineficiente para grandes conjuntos de dados, o que motivou a busca por estratégias de paralelização.

O perfil de desempenho sequencial revelou que a maior parte do tempo de execução estava relacionada às operações de verificação de números primos e cálculo de divisores. Isso indicou claramente as oportunidades de paralelização nesses aspectos.

A implementação paralela seguiu a estratégia mestre-escravo, onde o mestre é responsável pela coordenação do trabalho dos processos escravos. O vetor de inteiros foi lido a partir de um arquivo de entrada, carregado na memória do processo mestre e distribuído entre os processos escravos para processamento paralelo. Após o processamento, os resultados foram coletados no mestre e escritos em um arquivo de saída.

A paralelização permitiu uma considerável melhoria no desempenho do algoritmo, especialmente quando lidando com conjuntos de dados maiores. A divisão do trabalho entre os processos escravos reduziu significativamente o tempo de processamento em comparação com a versão sequencial. A medição do tempo de execução no processo mestre permitiu avaliar os ganhos da paralelização.

Em resumo, este trabalho demonstrou como a paralelização, em particular a estratégia mestre-escravo com MPI, pode ser uma abordagem eficaz para acelerar a identificação de números primos em grandes conjuntos de dados, aproveitando os recursos computacionais disponíveis. No entanto, a paralelização também introduz desafios na coordenação de tarefas e na minimização da sobrecarga de comunicação, que devem ser cuidadosamente considerados para obter um desempenho ótimo.

6 Referências

https://www.open-mpi.org/

https://mpitutorial.com/

https://www.codingame.com/playgrounds/54443/openmp/hello-openmp