6.8.2 Estudo de caso: contagem de números primos

O estudo de caso é um programa para determinar a quantidade de números primos entre *0* e um determinado valor inteiro N. Neste estudo, visitaremos a solução sequencial apresentada a seguir.

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

int primo (long int n) {
    long int i;
    for (i = 3; i < (int)(sqrt(n) + 1); i+=2)
    if (n%i == 0)
          return 0;
    return 1;
}

int main(int argc, char *argv[]) {
    int total =0;
    long int i, n;
    if (argc < 2) {
          printf("Valor inválido! Entre com o valor do maior
      inteiro\n");
          return 0;
      }
      else {
          n = strtol(argv[1], (char **) NULL, 10);
      }
    for (i = 3; i <= n; i += 2)
          if(primo(i) == 1)
                total++;
    /* Acrescenta o dois, que também é primo */
    total += 1;
    printf("Quant. de primos entre 1 e n: %d \n", ,→ total);
    return(0);
}
```

Esta solução sequencial descarta todos os números pares de início, pois obviamente eles não são primos. Em seguida, é verificado se N é divisível por algum número ímpar entre 0 e a raiz quadrada de N.

Neste estudo de caso, vamos paralelizar o mesmo código-fonte, alterando apenas as estratégias de escalonamento. O principal objetivo da utilização do mesmo código é verificar o impacto no tempo de execução da estratégia de escalonamento escolhida para atribuição das iterações do laço às *threads* disponíveis. A seguir é apresentado um código fonte trivial para o calculo de números primos.

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

```
#include <math.h>
#include <omp.h>

int primo (long int n) {
    long int i;
    for (i = 3; i < (long int)(sqrt(n) + 1); i+=2)
          if (n%i == 0) return 0;
    return 1;
    }

int main(int argc, char *argv[]) {
    double t_inicio , t_fim;
    long int i, n, total;
    if (argc < 2) {
          printf("Valor  inválido!  Entre  com  o  valor  do,  maior
      inteiro\n");
          return 0;
    } else {
          n = strtol(argv[1], (char **) NULL, 10);
    }
    total =0;
    t_inicio = omp_get_wtime();
    /* Solução trivial */
    #pragma omp parallel for reduction(+:total) schedule( ,static)
    /* Solução melhorada: schedule(dynamic ,10000) */
    for (i = 3; i <= n; i += 2)
          if(primo(i) == 1) total++;
            /* Acrescenta o dois, que também é primo */
            total += 1;
    t_fim = omp_get_wtime();
    printf("Quant. de primos entre 1 e %ld: %ld\n", n, ,→ total);
    printf("Tempo de execução: %f com %d threads\n", t_fim-t_inicio,
omp_get_max_threads());
}
```

Esse primeiro código apresenta uma solução para a paralelização, pois distribui uma faixa continua de números para cada *thread*, usando o escalonamento estático (**schedule(static)**). Já sabemos que, se essa divisão não for exata, o ambiente de execução do OpenMP se encarrega de atribuir as iterações restantes entre as *threads*.

Em seguida, cada *thread* conta quantos números primos existem no seu intervalo, incrementando a variável *total*, cujos respectivos valores são reduzidos para a variável de mesmo nome na *thread master* com uso da cláusula **reduction(+:total)**.

Essa solução é de simples implementação, contudo observamos um desbalanceamento de carga entre as *threads*. Esse desequilíbrio acontece porque a busca por números primos nas faixas mais altas tem custo computacional maior que nas faixas mais baixas. Ao executarmos a solução trivial com N = 700.000.000 e 48 *threads* e medirmos o tempo de execução com a função **omp_get_wtime()**. Obtemos 36252931 números primos.

Uma opção para diminuir esse problema do desequilíbrio de carga entre as *threads* seria utilizar o escalonamento dinâmico com a definição do tamanho do pedaço das iterações do laço de repetição, onde trocamos o escalonamento estático **schedule(static)** pelo dinâmico **schedule(dynamic, 10000)**, sendo a escolha do tamanho do pedaço (10.000) arbitrária. O ambiente OpenMP controla a lista com todos estes pedaços que, um a um, são atribuídos às *threads* disponíveis.

Quando uma *thread* termina sua busca por números primos, o ambiente de execução OpenMP lhe atribui um novo pedaço. Ao executarmos a solução balanceada com N = 700.000.000 e 48 *threads* e medirmos o tempo de execução.

A partir desses resultados, percebe-se a interferência do desbalanceamento de carga no tempo total das execuções: a solução trivial foi mais lenta que a segunda solução.

Para investigar melhor os motivos dessa diferença, realizamos um novo experimento com N = 700.000.000 e 8 *threads*. A Figura 6.11 mostra o tempo gasto em cada *thread* para procurar todos os números primos em seu intervalo de faixa continua na solução trivial. Pode-se notar um grande desbalanceamento no tempo de execução das *threads*. Esse gráfico evidencia que as *threads* procurando números primos nas faixas mais baixas terminam mais rápido do que as que buscam nas faixas mais altas. Por exemplo, a *thread* 0 terminou a busca em 127 segundos, enquanto a *thread* 7 terminou a busca em 405 segundos. O tempo total de execução da solução trivial foi então de 405 segundos, que é o tempo de execução da *thread* mais demorada.

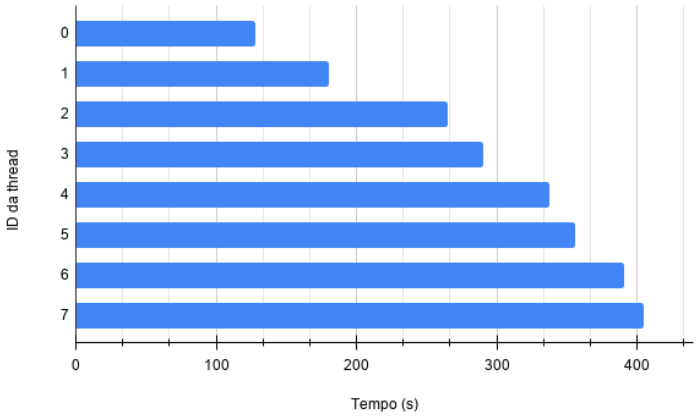

Figura 6.11: Tempo de execução de cada *thread* mostrando o desequilíbrio de carga entre elas.

Após a execução do programa com a solução balanceada, observamos que as *threads* na Figura 6.12 têm aproximadamente o mesmo tempo de execução, por volta de 291 segundos. Consequentemente, o tempo total de execução da solução balanceada diminuiu para 291 segundos.

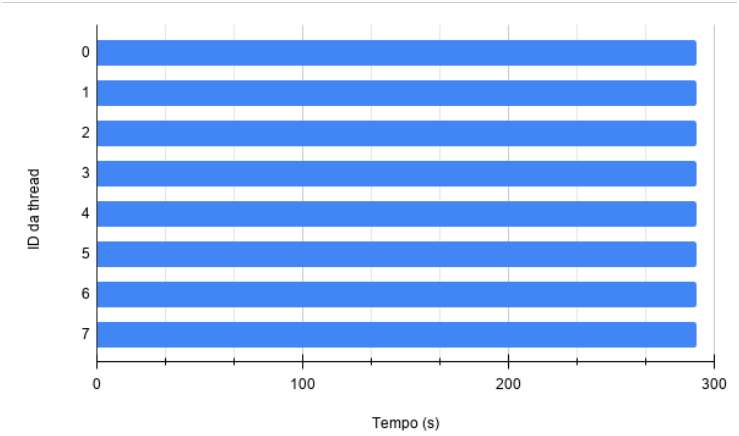

Figura 6.12: Tempo de execução de cada *thread* de forma balanceada.

A solução balanceada é parecida com o método de saco de tarefas[1] apresentado na Seção 5.8. A principal diferença reside no controle e atribuição das tarefas para as *threads*, o que é feito pelo próprio ambiente de execução do OpenMP, à medida que as *threads* se tornam disponíveis para execução. Essa solução balanceada foi obtida a partir de um breve estudo sobre o comportamento das *threads* e uma simples mudança na estratégia de escalonamento. Esses pontos devem estar sempre em mente na busca de uma estratégia para otimizar o código OpenMP.

Avaliação de desempenho

Uma arquitetura de multiprocessamento simétrico com um total de □□ processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz) foi escolhida para realizar o nosso estudo de escalabilidade.

[1] Nesse método, um processo (mestre) fica responsável por enviar as tarefas para os demais processos (trabalhadores). Assim que uma tarefa é terminada e o resultado é enviado para o mestre, uma outra tarefa será alocada para o trabalhador, e assim sucessivamente, até que não haja mais tarefas para serem executadas.

As Tabelas 6.5 e 6.6 mostram os tempos de execução da solução trivial e da solução balanceada, respectivamente. Esses tempos foram utilizados nos cálculos das métricas, como *speedup* e eficiência, conforme discutido na Seção 2.4.1, apresentados nas mesmas tabelas.

O tempo de 1316 segundos com uma *thread* e N = 700.000.000 foi utilizado como base para os demais cálculos dessas tabelas.

Tabela 6.5: Métricas números primos - solução trivial

Número de *threads*	Tempo de execução	*Speedup*	Eficiência
1	1316 s	1,0	1,00
2	847 s	1,6	0,78
4	488 s	2,7	0,68
8	261 s	5,0	0,63
12	177 s	7,4	0,62
16	130 s	10,1	0,63
20	108 s	12,2	0,61
24	90 s	14,6	0,61

Tabela 6.6: Métricas números primos - solução balanceada

Número de *threads*	Tempo de execução	*Speedup*	Eficiência
1	1316 s	1,0	1,00
2	677 s	1,9	0,97
4	377 s	3,5	0,87
8	191 s	6,9	0,86
12	127 s	10,3	0,86
16	95 s	13,8	0,86
20	77 s	17,2	0,86
24	64 s	20,6	0,86

Para entender o comportamento das soluções, desenhamos as curvas de escalabilidade na Figura 6.13. Nesse gráfico, a reta contínua representa o *speedup* linear ideal; a reta logo abaixo, anotada com pontos, indica o *speedup* da solução balanceada; a reta abaixo de todas apresenta o *speedup* da solução trivial, anotada com quadrados.

O gráfico das duas soluções tem, conforme o esperado, um crescimento linear. A escalabilidade da solução balanceada está muito mais próxima do *speedup* ideal. A escalabilidade da solução trivial se afasta da linha do *speedup* ideal conforme a

quantidade de *threads* é aumentada, devido ao desbalanceamento, que se torna cada vez mais evidente.

Figura 6.13: Gráfico de *speedup* - números primos

Esse desbalanceamento de carga fica ainda mais evidente quando são comparadas as eficiências das duas soluções, conforme pode ser visto na Figura 6.14. A curva no gráfico com marcações em círculos mostra a eficiência da solução balanceada e a curva com quadrados apresenta a eficiência da solução trivial.

A queda na eficiência da solução trivial acontece de forma rápida, mostrando o impacto do desbalanceamento. Nota-se que com 24 *threads* a eficiência é de 60% ou quase a metade da capacidade de processamento disponível.

A solução balanceada distribui a carga de trabalho de forma mais equilibrada entre todas as *threads* e, portanto, sua eficiência ficou por volta de 85%. Alguns trechos deste programa são executados de forma sequencial, como na operação de redução **reduction(+:total)**. Esses trechos afetam a eficiência, como observado a partir do uso de 2 *threads*.

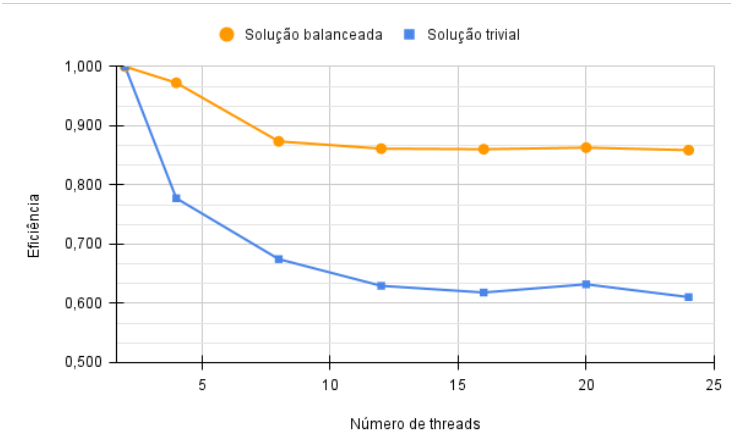

Figura 6.13: Gráfico de *speedup* - números primos.

Atividades

1 – Executar em sua máquina pessoal os códigos, para verificar o desempenho e comparar com as versões paralelas.

2 – Implementar pelo menos mais duas versões em OpenMP para comparar o desempenho com o exemplo apresentado.

3 – Realizar implementação utilizando o MPI.