6.8.1 Estudo de caso: método do trapézio

Em nosso primeiro estudo de caso, vamos revisar o programa de cálculo do valor da integral utilizando o método do trapézio, conforme mostrado na Seção 3.8. Vamos perceber que as alterações necessárias no código são muito pequenas e se resumem basicamente em uma única linha no código.

```
#include <stdio .h>
#include <math .h>
#include <omp .h>
double f( double x) { /* Calcula f(x) */
    double valor ;
    valor = exp(x);
    return ( valor );
}

int main ( int argc , char * argv []) {
    double integral ; /* Armazena resultado em integral */
    double a=0.0 , b =1.0; /* Limite esquerdo e direito */
    long n =8000000000; /* Número de Trapezóides */
    double h; /* Largura da base do Trapezio ide */
    h = (b-a)/n;
    integral = (f(a) + f(b)) /2.0;
    double t_inicio = omp_get_wtime ();
    /* Faz a redução e escalona as iterações para as threads */
    #pragma omp parallel for reduction (+: integral) schedule
(static)
    for ( long i = 1; i < n -1; i ++) {
          integral += f(a + i*h);
      }
   integral *= h ;
   double t_fim = omp_get_wtime ();
   printf (" Com n = %ld trapezoides , a estimativa \n", n);
   printf ("da integral de %f ate %f = %lf \n", a, b,integral);
   printf (" Tempo : \t %f com %d threads .\n", t_fim - t_inicio ,
   omp_get_max_threads ());
   return (0) ;
}
```

Vamos calcular o valor aproximado da integral da função $exp(x)$ no intervalo entre 0 e 1, com $n = 128 \times 10^9$. Esse valor tão grande de n é necessário para garantir um tempo mínimo de execução, visando a análise de seu comportamento, mas também para garantir a precisão do método utilizado, sabendo-se que o resultado esperado deve ser aproximadamente $1,71828182846 (e^1 - e^0)$.

O método do trapézio tem um grande potencial de paralelismo, pois cada iteração do laço de repetição **for** pode ser executada de forma independente. Dessa forma, podemos utilizar a diretiva **#pragma omp for** para paralelizar o cálculo da função **f()**.

Nesse exemplo, a atribuição de cada iteração do laço para as threads disponíveis foi feita utilizando a estratégia de escalonamento estática **schedule(static)** do OpenMP. Essa divisão atribui *n/p* iterações para cada thread, onde *p* é a quantidade de threads disponíveis na região paralela. Como essa divisão pode não ser exata, o ambiente de execução do OpenMP se encarrega de atribuir as iterações restantes entre as threads, conforme políticas dependentes de implementação.

A variável integral do exemplo é utilizada para acumular todos os resultados dos cálculos realizados em cada iteração. Como é uma variável compartilhada entre todas as threads, é necessário tratar a condição de corrida por meio de sincronização. Nesse caso, foi utilizada a cláusula **reduction(+:integral)** e todo esse controle de atualização e sincronização do acesso à variável foi delegado novamente para o OpenMP. A função **omp_get_wtime()** foi utilizada para medir o tempo de execução desse laço.

A seguir vamos explorar melhor este exemplo, fazendo a análise de sua escalabilidade.

Avaliação de desempenho

A análise de escalabilidade que faremos neste estudo está dividida em duas abordagens, uma sob a ótica de Amdahl e outra sob a de Gustafson, variando o número de threads (e processadores) entre 1 e 24 em ambos os casos. Primeiramente vamos variar apenas o número de threads, mas mantendo o tamanho do problema fixo. No segundo caso, vamos aumentar o tamanho do problema proporcionalmente, à medida que aumentamos o número de threads.

Os testes foram realizados em um computador com uma arquitetura de multiprocessamento simétrico com 24 processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz).

Os tempos de execução foram coletados, *speedup* e eficiência calculados, sendo esses apresentados na Tabela 6.3. Neste experimento, o tempo de execução de 2324 segundos do algoritmo seqüencial foi medido com uma thread e $N = 128.000.000.000$, sendo utilizado como referência para os demais cálculos dessa tabela.

Tabela 6.3: Métricas do método de trapézio - Amdahl

Número de *threads*	**Tempo de execução**	***Speedup***	**Eficiência**
1	2324 s	1,0	1,00
2	1206 s	1,9	0,96
4	670 s	3,5	0,87
8	339 s	6,9	0,86
12	227 s	10,2	0,85
16	171 s	13,6	0,85
20	137 s	16,9	0,85
24	115 s	20,3	0,84

A Figura 6.7 mostra a evolução do *speedup* de acordo com o aumento de processadores, de acordo com os dados obtidos a partir da Tabela 6.3. Uma linha contínua nesse gráfico indica o *speedup* linear ideal e a outra linha com pontos destacados apresenta o resultado do *speedup* calculado.

Esse gráfico mostra um comportamento esperado para o problema do trapézio, pois o cálculo de cada iteração pode ser realizado de forma independente.

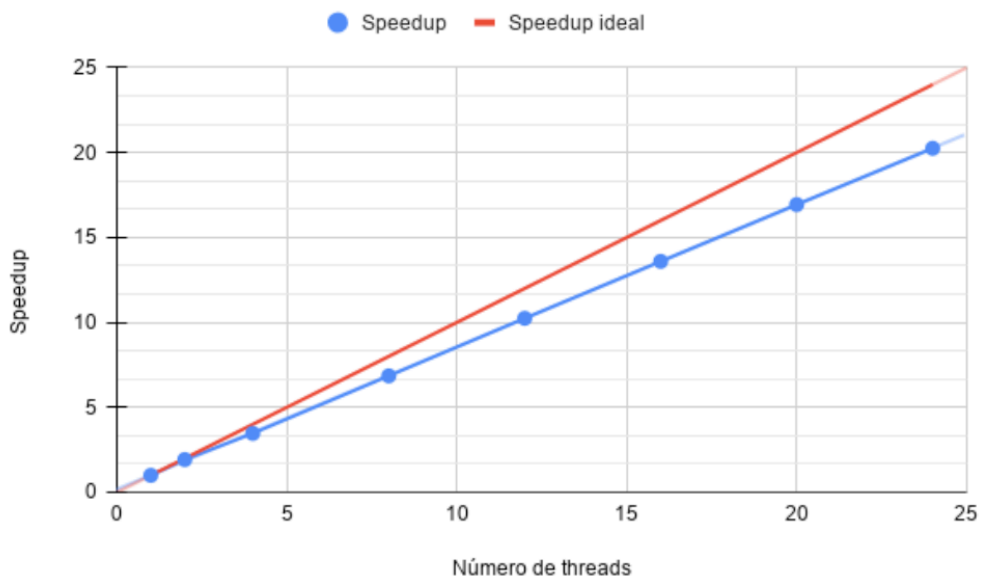

Figura 6.7: Gráfico de *speedup* - Amdahl

Apesar disso, sabemos que essa solução possui uma fração não paralelizável e, como consequência, esse trecho afeta a escalabilidade.

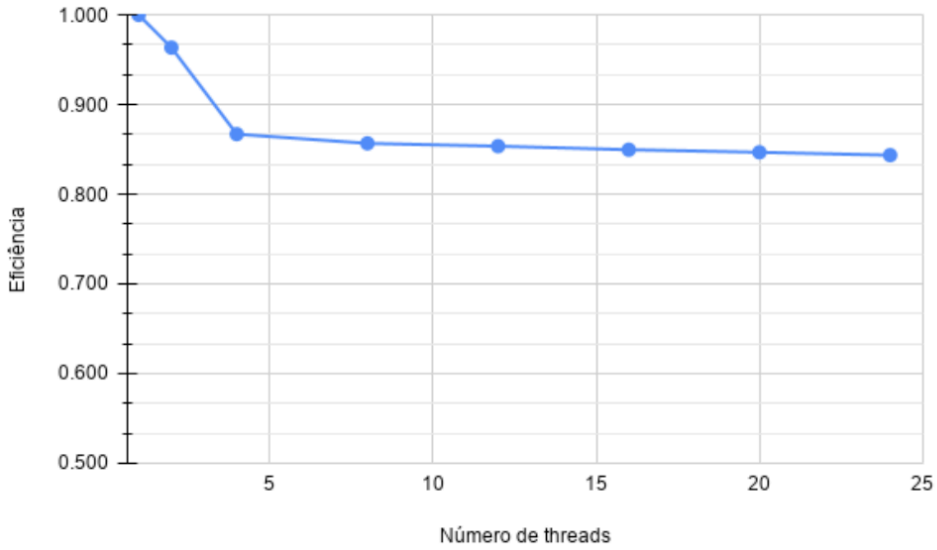

Figura 6.8: Gráfico e eficiência - Amdahl

Uma das partes importantes dessa solução é o somatório final do resultado de cada iteração feita pela cláusula de redução **reduction(+:integral)**. Essa e outras partes não paralelizáveis impactam na escalabilidade do programa.

Esse impacto é percebido também no gráfico de eficiência, como mostra a Figura 6.8. A queda da eficiência no gráfico ao adicionar 2 processadores, seguida de nova queda com 4 processadores, indica a existência de um trecho seqüencial na solução que não se beneficia da disponibilidade de mais processadores.

Para medir a escalabilidade de acordo com a definição de Gustafson, aumentamos o tamanho do problema na mesma proporção que o aumento do número de processadores. No caso do método do trapézio, o tamanho do problema está diretamente relacionado ao laço de repetição e, portanto, realizamos um aumento linear de N para obter um aumento também linear na carga de trabalho.

Tabela 6.4: Métricas método do trapézio - Gustafson

Tamanho N	Tempo Seq.	Threads (p)	Tempo Paralelo	Speedup	Eficiência
$2,6 \times 10^9$	49 s	1	49 s	$1,0$	$1,00$
$5,3 \times 10^9$	97 s	2	50 s	$1,9$	$0,97$
$1,0 \times 10^{10}$	194 s	4	56 s	$3,5$	$0,87$
$2,1 \times 10^{10}$	387 s	8	56 s	$6,9$	$0,86$
$3,1 \times 10^{10}$	581 s	12	57 s	$10,2$	$0,85$
$4,2 \times 10^{10}$	775 s	16	57 s	$13,5$	$0,84$
$5,3 \times 10^{10}$	971 s	20	57 s	$17,0$	$0,85$
$6,3 \times 10^{10}$	1162 s	24	57 s	$20,2$	$0,84$

Em nosso experimento definimos inicialmente que $N = 2,\ 6 \times 109$ para um processador e continuamos aumentando o tamanho de problema em razão da quantidade de processadores utilizados, conforme mostra a Tabela 6.4. Esses resultados também podem ser observados no gráfico da Figura 6.9, onde a linha contínua indica o *speedup* linear ideal e a linha com pontos indica o resultado do *speedup* calculado.

O comportamento da curva de *speedup* da Figura 6.9 se mostra muito similar à Figura 6.7. Isso indica que o tempo gasto com a parte sequencial da solução tem um impacto proporcional conforme o tamanho do problema aumenta. No exemplo apresentado, a cláusula de redução **reduction(+:integral)** pode ser o principal elemento para que a curva de *speedup* tenha esse comportamento.

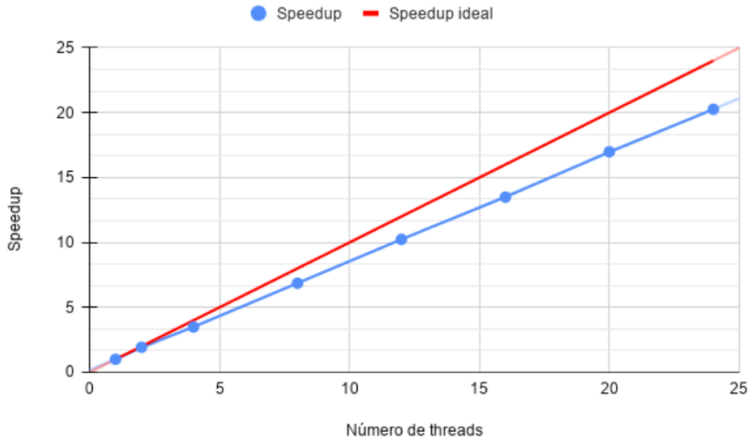

Figura 6.9: Gráfico de *speedup* - Gustafson

Na Figura 6.10 apresentamos a curva de eficiência para esse novo experimento, bastante semelhante à da Figura 6.8. Mais uma vez, percebe-se que a eficiência da nossa solução diminui a partir de 2 processadores e até o limite inferior de 85% de eficiência.

Como no exemplo com uso do MPI, os resultados obtidos em ambos métodos de avaliação, Amdahl e Gustafson, são muito similares, dado que a fração sequencial do problema é muito pequena. A queda inicial de desempenho, com duas e quatro threads, provavelmente se deve à sobrecarga para a criação e controle de threads, que é amortizada quando temos um número maior de threads em execução

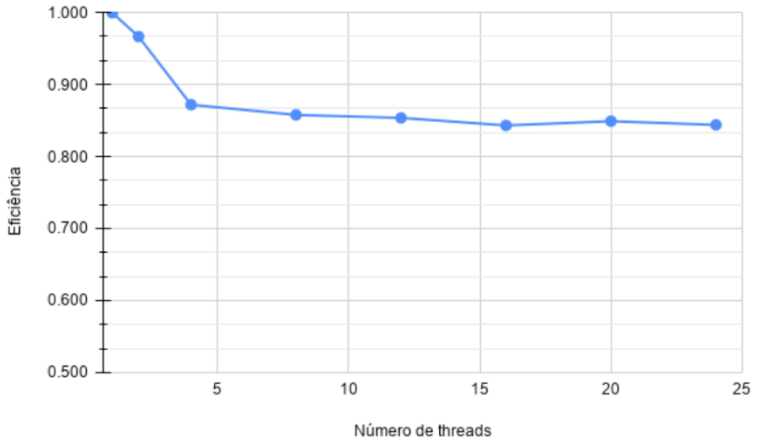

Figura 6.10: Gráfico da eficiência - Gustafson

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.