Universidade Federal do Rio Grande do Norte
Departamento de Engenharia da Computação e Automação
DCA3703 - Programação Paralela
Tarefa 4 - Aplicações limitadas por memória ou CPU
Aluno: Daniel Bruno Trindade da Silva

1 Introdução

A computação paralela tem se tornado uma estratégia essencial para melhorar o desempenho de programas, especialmente com a popularização de processadores com múltiplos núcleos. O OpenMP (Open Multi-Processing) é uma API amplamente utilizada para programação paralela em sistemas com memória compartilhada, permitindo a distribuição de tarefas entre múltiplas threads de forma simples e eficiente. Este relatório apresenta a implementação e análise de dois programas paralelos desenvolvidos em linguagem C com OpenMP: um limitado por memória (memory-bound), baseado em somas simples entre vetores, e outro limitado por processamento (compute-bound), com operações matemáticas intensivas.

O objetivo principal é avaliar o impacto da paralelização com a diretiva `#pragma omp parallel for` no tempo de execução dos programas, variando o número de threads utilizadas. A partir dos experimentos, são observadas as situações em que o desempenho melhora, se estabiliza ou até mesmo piora, permitindo uma reflexão sobre os efeitos do multithreading de hardware em diferentes cenários. Essa análise contribui para a compreensão dos limites e vantagens da programação paralela, considerando o perfil computacional das aplicações.

2 Metodologia

Realizamos a implementação de dois programas em C, o primeiro deles tem como objetivo realizar somas simples para simular uma carga de trabalho limitada pelo acesso a memória (memory-bound). Ele aloca dinamicamente um vetor de 100 milhões de elementos do tipo double e realiza uma operação simples de escrita: atribuir a cada posição do vetor o valor i + 2, onde i é o índice da iteração. A paralelização é feita com a diretiva `#pragma omp parallel` for, permitindo que múltiplas threads realizem essas atribuições em paralelo.

O tempo de execução é medido com a função `omp_get_wtime()`, antes e depois do laço paralelo. Como a operação executada em cada iteração é computacionalmente simples, o desempenho do programa depende principalmente da taxa de leitura e escrita na memória principal. Esse tipo de programa evidencia os limites do ganho de desempenho com o aumento do número de threads, já que o gargalo está na largura de banda da memória, e não na capacidade de cálculo da CPU.

Assim ficou nosso código:

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

#define SIZE 100000000

int main() {

 double *a = (double *)malloc(SIZE * sizeof(double));
 double start, end;
```

```
 start = omp_get_wtime();
 #pragma omp parallel for
 for (int i = 0; i < SIZE; i++) {
  a[i] = i+2;
 }
 end = omp_get_wtime();

 printf("Tempo de execução (Memory-bound): %f segundos\n", end - start);

 free(a);

 return 0;
}
```

O segundo programa desenvolvido simula uma carga de trabalho limitada pela capacidade de processamento da CPU (CPU-bound). Ele realiza 100 milhões de operações matemáticas intensivas, envolvendo funções como sin, cos e sqrt, seguidas de uma divisão. Essas operações são realizadas dentro de um laço for paralelo, utilizando a diretiva `#pragma omp parallel for` com uma cláusula de reduction para somar corretamente os resultados parciais de cada thread na variável `result`.

O tempo de execução também é medido com `omp_get_wtime()`. Diferente do programa *Memory Bound*, aqui o gargalo está no tempo necessário para realizar os cálculos matemáticos complexos, que demandam mais ciclos de CPU. Esse tipo de programa permite observar como o desempenho pode escalar com múltiplas threads até certo ponto, mas também evidencia como o aumento de concorrência pode gerar competição por recursos internos da CPU, como unidades de ponto flutuante e cache, o que pode limitar os ganhos ou até causar perda de desempenho.

O código para essa programa ficou assim:

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

#define SIZE 100000000

int main() {
 double result = 0.0;
 int i;
 double start, end;

 start = omp_get_wtime();
 #pragma omp parallel for reduction(+:result)
 for (i = 0; i < SIZE; i++) {
  result += sin(i) * cos(sqrt(i)) / (i + 1.0);
 }
 end = omp_get_wtime();

 printf("Tempo de execução (CPU-bound): %f segundos\n", end - start);
 printf("Resultado final: %f\n", result);

 return 0;
}
```

Para execução de ambos os programas (memory bound e cpu bound) utilizamos a diretiva `OMP_NUM_THREADS=n` onde n é o número de threads que sera utilizado para execução. Nesse estudo aumentaremos o valor de n progressivamente afim de verificar até que ponto temos ganho de desempenho.

Como os resultados desse experimento dependem diretamente do hardware utilizado se faz importante mencionar que os códigos estão sendo executado em um computador com a seguinte configuração:

- **Processador:** Intel I3 10100F
  - Núcleos: 4
  - Threads: 8
- **Memoria RAM:** 16 gigas
  - Frequência: 2400Hz

3 Resultados

Primeiro vamos conferir os resultados obtidos no código do *Memory Bound*. Executamos o código por 10 vezes sempre aumentando o número de threads, ou seja, começamos com 1 thread e fomo até 10.

Obtivemos o seguinte resultado:

Threads	1	2	3	4	5	6	7	8	9	10
Tempo (s)	0.3745	0.2012	0.1366	0.1085	0.0999	0.0874	0.0757	0.0868	0.0817	0.0794

Table 1: Tempo de execução do programa **memory-bound** para diferentes números de threads

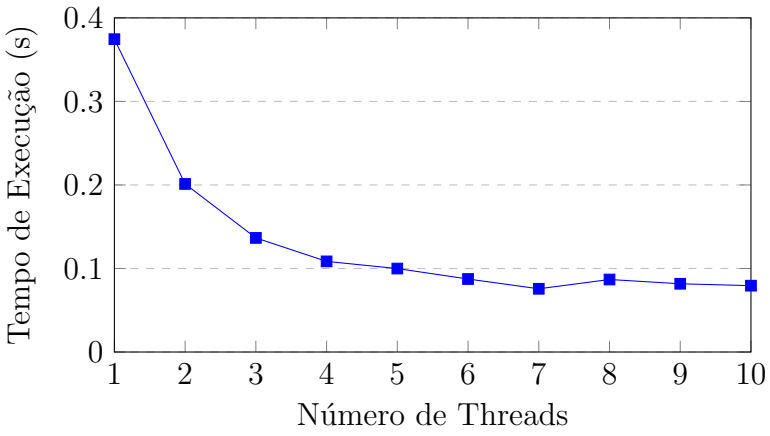

Figure 1: Desempenho do programa **memory-bound** variando o número de threads

O comportamento do tempo de execução está bem como esperado, podemos ver uma melhora significativa até a execução com 4 threads, que é exatamente o número de núcleos físicos que possuímos no processador, entre 5 e 7 threads, continuamos a ver uma melhora, mas bem modesta. A partir da execução com 8 threads começamos a ver oscilação e isso é esperado porque mais threads do que núcleos físicos não melhoram a velocidade de acesso à memória e podem até aumentar a contenção de cache e a latência da RAM.

Agora vamos aferir os resultados obtidos no programa *CPU Bound*:

Threads	1	2	3	4	5	6	7	8	9	10
Tempo (s)	2.5916	1.3193	0.9080	0.6792	0.7766	0.6214	0.5858	0.5396	0.5741	0.5706

Table 2: Tempo de execução do programa **CPU-bound** para diferentes números de threads

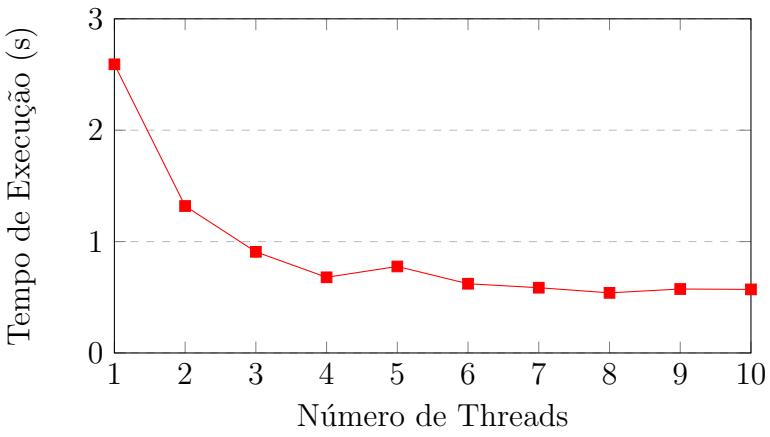

Figure 2: Desempenho do programa **CPU-bound** variando o número de threads

Da mesma forma que no *Memory Bound* o *CPU Bound* mostrou um desempenho tal como esperado. Por se tratar de um programa que precisa realizar cálculos mais complexos na CPU, seu ganho de desempenho pelo aumento de threads está limitado a quantidade de núcleos físicos. Por isso tivemos uma melhora significativa de 1 a 4 threads, a partir do 5º, o aumento de número de threads passou a causar disputa de recursos com isso o tempo de execução passou a oscilar.

4 Conclusão

Neste relatório, analisamos o impacto da paralelização em dois programas distintos utilizando OpenMP: um *Memory Bound*, que realiza operações simples em um grande vetor, e outro *CPU Bound*, que executa cálculos matemáticos intensivos. Observamos que no programa *Memory Bound*, o tempo de execução reduziu conforme o número de threads aumentou, até atingir um limite onde os ganhos se tornaram mínimos. Isso ocorre porque a principal limitação não está na capacidade computacional da CPU, mas sim na largura de banda da memória, tornando o multithreading útil apenas até que esse gargalo se manifeste.

Já no programa *CPU Bound*, a redução do tempo de execução foi eficiente até atingir o número de núcleos físicos do processador, mas após esse ponto, a competição por recursos internos fez com que os ganhos diminuíssem e, em alguns casos, o desempenho piorasse. Esse comportamento demonstra que o uso eficiente do paralelismo depende da natureza da carga de trabalho: aplicações limitadas por memória podem se beneficiar da paralelização até um certo ponto, enquanto aplicações *CPU Bound* são mais sensíveis à arquitetura do processador. Assim, entender a relação entre hardware e software é essencial para otimizar o desempenho em sistemas paralelos.