UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
Departamento de Engenharia de Computação e Automação

Programação Paralela
Juliane da Silva Santos

Relatório 4:
Aplicações limitadas por memória ou CPU

1 Introdução

A crescente demanda por desempenho em aplicações científicas, de engenharia e dados intensivos impulsiona o uso de técnicas de paralelização. Entre elas, a utilização de multithreading via OpenMP se destaca por sua simplicidade e portabilidade. Este relatório apresenta uma análise comparativa entre dois tipos de programas paralelizados em C com OpenMP: um limitado por acesso à memória (memory-bound) e outro por capacidade de processamento (compute-bound).

Segundo Pacheco (2011) [1], programas memory-bound são aqueles cujo desempenho é restringido pela largura de banda e latência de acesso à memória. Já os programas compute-bound são limitados pela velocidade de execução dos cálculos. Stallings (2010) [2] complementa que o uso do paralelismo em arquiteturas modernas envolve desafios como a contenção de memória, competição por recursos internos da CPU e efeitos de hyper-threading. O código completo pode ser visto no repositório disposto na plataforma GitHub.

2 Metodologia

O estudo consistiu em implementar dois algoritmos paralelos usando OpenMP:

- Teste memory-bound: realiza operações simples (soma e subtração) em vetores grandes.

- Teste compute-bound: realiza cálculos matemáticos intensivos (funções trigonométricas e exponencial).

Para cada algoritmo, o tempo de execução foi medido variando o número de threads (de 1 até 16). Os resultados foram armazenados em um arquivo CSV para posterior análise.

A métrica utilizada foi o tempo de execução em segundos, capturado por meio da função gettimeofday(), representando o tempo real gasto pela aplicação.

2.1 Implementação

A implementação foi feita em linguagem C com diretivas OpenMP. A seguir, estão descritas as principais partes do código:

- Definições e medição de tempo

```c
#define N 10000000
#define MAX_THREADS 16

double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1e6;
}
```

Assim como no projeto passado, para medir o tempo de execução, foi recomendado em sala de aula o uso da função gettimeofday(), que captura o tempo total de execução (wall-clock time), incluindo pausas, esperas e operações de I/O. Essa abordagem difere da função clock(), que mede apenas o tempo de CPU efetivamente utilizado pelo processo, ignorando períodos de inatividade, como espera por I/O ou suspensão do programa. Para facilitar a medição do tempo, foi desenvolvida a função gettimems(), que utiliza gettimeofday() e retorna o tempo atual em milissegundos. Essa função é chamada no inicio do for e no final e para o cálculo fina do tempo de execução realizamos uma subtração do tempo final pelo inicial.

- Teste memory-bound

```c
void memory_bound_test(int num_threads, double *time_taken) {
    static double a[N], b[N], c[N];

    // Inicializa  o dos vetores
    for (int i = 0; i < N; i++) {
        a[i] = i * 1.0;
        b[i] = (N - i) * 1.0;
    }

    omp_set_num_threads(num_threads);
    double start_time = get_time();

    // C lculo simples: soma e subtra  o
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i] - i;
    }

    double end_time = get_time();
    *time_taken = end_time - start_time;

    printf("[Memory-Bound] Threads: %d | Tempo: %f s\n", num_threads, *
        time_taken);
```

```
}
```

A função cria vetores grandes com persistência (static), para que não usem a pilha da função, evitando stack overflow. Inicializa os vetores com valores reais. Define quantas threads OpenMP vai usar na próxima região paralela. E no segundo for implementa a paralelização, onde cada iteração do loop é independente, então OpenMP divide o loop entre várias threads.

A operação c[i] = a[i] + b[i] - i; é simples, então a velocidade do acesso à memória será o principal fator de desempenho.

- Teste compute-bound

```
void compute_bound_test(int num_threads, double *time_taken) {
    static double results[N];

    omp_set_num_threads(num_threads);
    double start_time = get_time();

    // C lculo intensivo: seno, cosseno e exponencial
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        results[i] = sin(i) + cos(i) + exp(i * 0.000001);
    }

    double end_time = get_time();
    *time_taken = end_time - start_time;

    printf("[Compute-Bound] Threads: %d | Tempo: %f s\n", num_threads, *
        time_taken);
}
```

Na segunda função, temos a implementação de um cálculo mais complexo, criamos um vetor para armazenar os resultados. Cada iteração realiza operações matemáticas pesadas: seno, cosseno e exponencial. Isso exige muito das unidades de ponto flutuante da CPU.

Esse código também é ideal para paralelizar com OpenMP, pois cada cálculo é independente.

- Medição e registro dos resultados

```
    for (int i = 0; i < num_values; i++) {
        int num_threads = threads[i];
        double memory_time, compute_time;

        memory_bound_test(num_threads, &memory_time);
        compute_bound_test(num_threads, &compute_time);

        // Salva os resultados no CSV
        fprintf(file, "%d,%.6f,%.6f\n", num_threads, memory_time,
            compute_time);

```

```
11            printf("\n");
12        }
```

3 Resultados e Discussão

Durante os testes, observou-se que:

O teste memory-bound apresentou melhorias até certo número de threads. Após esse ponto, o ganho marginal diminuiu devido à contenção da memória e limitação do barramento.

O teste compute-bound escalou bem até atingir o número de núcleos físicos disponíveis. Acima disso, o uso de hardware multithreading (como o HyperThreading da Intel) piorou ou estabilizou o desempenho, devido à competição pelas unidades de execução interna da CPU.

Podemos observar essa tendencia nos gráficos a seguir:

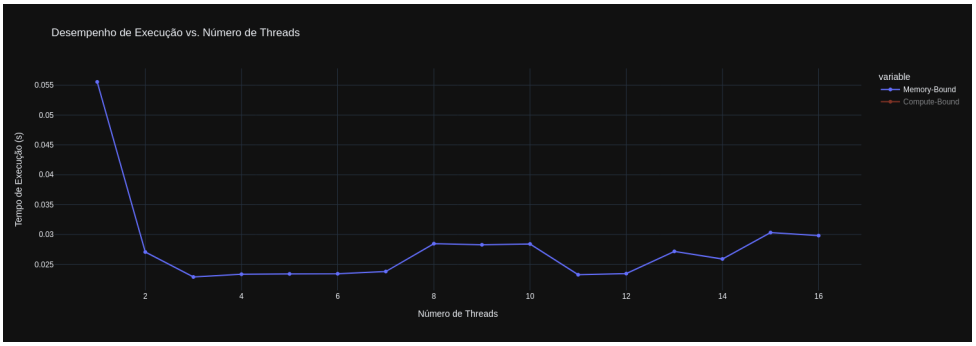

Figura 1: Gráfico de desempenho do tempo de execução para o memory-bound.

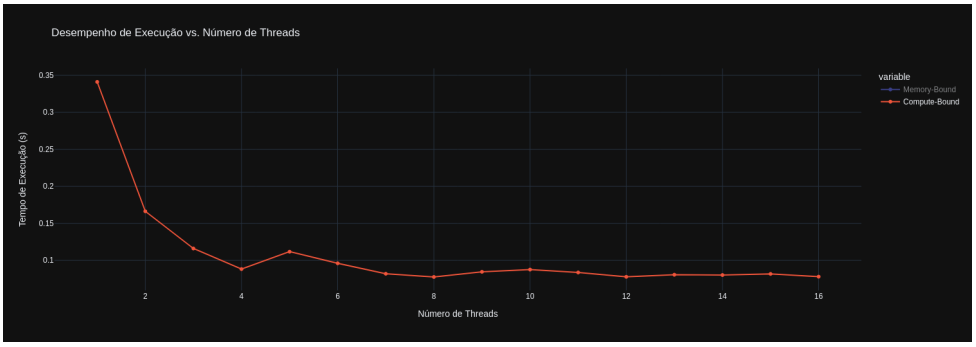

Figura 2: Gráfico de desempenho do tempo de execução para o compute-bound.

No programa compute-bound, que realiza cálculos intensivos (seno, cosseno e exponencial), houve uma melhora perceptível no desempenho com o aumento no número de threads até atingir o número de núcleos físicos da máquina. Isso se dá porque o trabalho computacional é dividido entre as threads, reduzindo o tempo de execução.

Contudo, ao ultrapassar o número de núcleos físicos disponíveis e passar a utilizar hardware multithreading, o ganho de desempenho estabilizou. Isso ocorre porque múltiplas threads competem pelos mesmos recursos internos da CPU, como registradores, caches e unidades de execução, conforme discutido por Stallings (2010) [2].

Essa contenção de recursos pode causar overhead, reduzindo a eficiência da paralelização, como também alerta Pacheco (2011) [1], ao mencionar que mais threads nem sempre significam mais desempenho — especialmente em programas compute-bound, onde a capacidade de processamento é o gargalo.

Já no programa memory-bound, que executa operações simples sobre grandes vetores, o ganho de desempenho foi observado apenas até um certo ponto (geralmente entre 4 e 8 threads). Acima disso, o tempo de execução se estabilizou ou até piorou levemente.

Isso acontece porque o acesso à memória se torna o gargalo. A largura de banda do barramento de memória, a latência e a quantidade de caches compartilhados por núcleos se tornam limitadores significativos. Mesmo que existam múltiplas threads disponíveis, elas não conseguem acessar a memória principal com velocidade suficiente para manter todas ocupadas eficientemente.

Como destaca Stallings (2010), o desempenho de programas memory-bound está fortemente atrelado à organização da hierarquia de memória (cache L1, L2, L3) e ao acesso à RAM. Pacheco (2011) também observa que programas com alta taxa de transferência de dados têm seu desempenho limitado não pela CPU, mas pela taxa com que os dados podem ser lidos e escritos na memória.

4 Conclusão

A paralelização com OpenMP permite significativas melhorias de desempenho, mas essas melhorias não são lineares nem ilimitadas. O comportamento do programa varia conforme sua natureza — memory-bound ou compute-bound — e o número de threads precisa ser escolhido com base na arquitetura da máquina.

Esse experimento reforça a importância do conhecimento da arquitetura de hardware e da natureza do problema para otimizar o desempenho de programas paralelos.

Referências

[1] Peter. PACHECO. *An introduction to parallel programming*. Morgan Kaufmann, Massachusetts, 2011.

[2] William. STALLINGS. *Arquitetura e organização de computadores.* Pearson Education, São Paulo, 8 edition, 2010.