

# Relatório: Aplicações limitadas por memória ou CPU

Aluno: Cristovão Lacerda Cronje

## 1. Introdução

Este relatório apresenta uma análise comparativa do desempenho de aplicações paralelas em dois cenários distintos: computação intensiva (CPU-bound) e acesso intensivo à memória (memory-bound). O objetivo é entender como o paralelismo com OpenMP se comporta em cada caso, identificando gargalos e padrões de eficiência.

## 2. Metodologia

Foram implementados dois programas em C com OpenMP:

- CPU-bound: Realiza cálculos matemáticos intensivos (funções trigonométricas e exponenciais) em um loop de 10 milhões de iterações.
- Memory-Bound: Operações em uma matriz grande (3GB), com acesso aleatório à memória.

O objetivo é entender como o paralelismo se comporta em cada cenário, identificando gargalos e padrões de eficiência. Os testes foram executados em um Intel i5-3210M (2 núcleos físicos, 4 threads com Hyper-Threading).

Código Implementado:

```
1 // CPU-bound benchmark
2 #include <stdio.h>
3 #include <time.h>
4 #include <omp.h>
5 #include <math.h>
6 #include <stdlib.h>
7 // Configuração
8 #define CPU_ITER 10000000
9 #define ARRAY_SIZE 1000000000
10 #define TRIES 5
11 // Função de tempo preciso
12 double get_time() {
13     struct timespec ts;
14     clock_gettime(CLOCK_MONOTONIC, &ts);
15     return ts.tv_sec + ts.tv_nsec / 1e9;
16 }
17 // Função de tarefa CPU-bound
18 double cpu_task(double x) {
19     double sum = 0;
20     for(int i = 0; i < 20; i++) {
21         sum += sin(x) * cos(x) * exp(x);
22         x += 0.0001;
23     }
24     return sum;
25 }
26 // Benchmark CPU-bound
27 void cpu_bench() {
28     printf("===== CPU BOUND =====\n");
29     // Serial (referência)
30     double start = get_time();
31     double serial_sum = 0;
32     for(int i = 0; i < CPU_ITER; i++) {
33         serial_sum += cpu_task(i * 0.001);
34     }
35     double serial_time = get_time() - start;
36     printf("Serial: %.4f s\n", serial_time);
37     printf("Threads | Tempo (s) | Speedup | Eficiência\n");
38     // Paralelo com 1 a 4 threads
39     for(int t = 1; t <= 4; t++) {
40         double total_time = 0;
41         double par_sum = 0;
42         for(int try = 0; try < TRIES; try++) {
43             start = get_time();
44             #pragma omp parallel for reduction(+:sum) num_threads(t)
45             for(int i = 0; i < CPU_ITER; i++) {
46                 sum += cpu_task(i * 0.001);
47             }
48             total_time += get_time() - start;
49             par_sum += sum;
50         }
51         double avg_time = total_time / TRIES;
52         double speedup = serial_time / avg_time;
53         double eff = (speedup / t) * 100;
54         printf("%7d | %.4f | %.2f | %.2f%%\n",
55             t, avg_time, speedup, eff);
56     }
57 }
58 // Memory-bound benchmark
59 void mem_bench() {
60     printf("===== MEMORY BOUND =====\n");
61     const long ROWS = 20000;
62     const long COLS = 20000;
63     double *matrix = malloc(ROWS * COLS * sizeof(double));
64     if(matrix == NULL) {
65         printf("Falha na alocação de %.1fMB\n",
66             ROWS * COLS * sizeof(double) / (1024.0 * 1024.0));
67         exit(1);
68     }
69     // Inicialização paralela
70     #pragma omp parallel for
71     for(long i = 0; i < ROWS * COLS; i++) {
72         matrix[i] = (i % 100) * 0.0001;
73     }
74     // Benchmark serial de referência
75     double start = get_time();
76     double sum = 0;
77     for(long n = 0; n < ROWS * COLS; n++) {
78         long i = (n * 32771) % (ROWS * COLS);
79         sum += sqrt(matrix[i]) * matrix[i] * 0.5 * 1.0001;
80     }
81     double serial_time = get_time() - start;
82     // Benchmark paralelo
83     for(int t = 1; t <= 4; t++) {
84         double total_time = 0;
85         double par_sum = 0;
86         for(int try = 0; try < TRIES; try++) {
87             start = get_time();
88             #pragma omp parallel for reduction(+:local_sum) num_threads(t)
89             for(long n = 0; n < ROWS * COLS; n++) {
90                 long i = (n * 32771) % (ROWS * COLS);
91                 local_sum += sqrt(matrix[i]) * matrix[i] * 0.5 * 1.0001;
92             }
93             total_time += get_time() - start;
94             par_sum += local_sum;
95         }
96         double avg_time = total_time / TRIES;
97         double speedup = serial_time / avg_time;
98         double eff = (speedup / t) * 100;
99         printf("%7d | %.4f | %.2f | %.2f%%\n",
100             t, avg_time, speedup, eff);
101     }
102     free(matrix);
103 }
```

**Compilação:** gcc -o 004\_tarefa 004\_tarefa.c -fopenmp -lm -lrt

### 3. Análise dos Resultados

```
=====
CPU-BOUND vs MEMORY-BOUND
i5-3210M (2 núcleos, 4 threads)
=====

=== CPU-BOUND (Cálculos Intensivos) ===
Serial: 11.8834 s
Threads | Tempo (s) | Speedup | Eficiencia
-----|-----|-----|-----
1 | 10.9769 | 1.08 | 108.26%
2 | 5.4682 | 2.17 | 108.66%
3 | 5.0689 | 2.34 | 78.15%
4 | 4.6032 | 2.58 | 64.54%

=== MEMORY-BOUND (Matriz 3051.8MB) ===
Serial: 36.2546 s (sum=282880284.88)
Threads | Tempo (s) | Speedup | Eficiencia
-----|-----|-----|-----
1 | 37.1582 | 0.98 | 97.57%
2 | 22.0641 | 1.64 | 82.16%
3 | 17.6941 | 2.05 | 68.30%
4 | 15.8039 | 2.29 | 57.35%
```

#### 3.1. Comportamento CPU-Bound

- Eficiência >100% com 1-2 threads: Devido a otimizações do compilador e uso eficiente de recursos.
- Speedup de 2.17x com 2 threads: Próximo do ideal (2x), confirmando bom uso dos núcleos físicos.
- Queda para 64.54% com 4 threads: Hyper-Threading

introduz competição por unidades de execução, reduzindo eficiência.

Conclusão:

- Paralelismo eficaz até 2 threads (núcleos físicos).
- Hyper-Threading traz ganhos menores (2.58x), mas com custo de eficiência.

#### 3.2. Comportamento Memory-Bound

- Eficiência de 82.16% com 2 threads: Abaixo do CPU-bound, mas ainda razoável.
- Speedup de 2.29x com 4 threads: Melhor que o esperado (teoria previa 1.5-2x), possivelmente devido a:
  - Cache L3 compartilhado (3MB) mitigando gargalo de memória.
  - Otimizações de hardware (prefetching).
- Eficiência cai para 57.35% com 4 threads, alinhado à saturação do barramento de memória.

Conclusão:

- Gargalo de memória é evidente, mas menos severo que o teórico.

- Paralelismo ainda útil, mas com retornos decrescentes.

#### 4. Quadro Comparativo

Característica	CPU-Bound	Memory-Bound
<b>Gargalo principal</b>	Unidades de execução da CPU	Banda/latência de memória
<b>Escalonamento ideal</b>	Threads = núcleos físicos (2)	Threads $\leq$ canais de memória (1-2)
<b>Eficiência observada</b>	108% (2 threads) $\rightarrow$ 64% (4 threads)	82% (2 threads) $\rightarrow$ 57% (4 threads)
<b>Speedup máximo</b>	2.58x (4 threads)	2.29x (4 threads)
<b>Impacto do HT</b>	Queda de eficiência (64%)	Queda menos acentuada (57%)

#### 5. Discussão sobre Desvios do Esperado

##### 5.1. CPU-Bound: Eficiência >100%

- Possíveis causas:
  - Melhor uso de pipeline e registradores na versão paralela.
  - Baixo overhead no escalonamento para poucas threads.

##### 5.2. Memory-Bound: Speedup Melhor que o Teórico

- Fatores:
  - Acesso pseudo-aleatório não foi totalmente eficaz em evitar prefetching (Técnica que pré-carrega dados na cache para acelerar acessos futuros à memória). Acesso a dados ainda parcialmente previsível (prefetching).
  - Operações simples (sqrt) não saturaram totalmente a banda de memória.

##### 5.3. Comparação com Resultados Esperados

Métrica	Esperado	Observado
<b>CPU-Bound (4 threads)</b>	Eficiência: 50-70%	64.54% (dentro do esperado)
<b>Memory-Bound (4 threads)</b>	Eficiência: 40-55%	57.35% (ligeiramente acima)

## 6. Conclusões

### 1. CPU-Bound:

- Paralelismo é altamente eficaz até o número de núcleos físicos.
- Hyper-Threading traz benefícios limitados (speedup 2.58x, eficiência 64%).

### 2. Memory-Bound:

- Gargalo de memória é claro, mas otimizações de hardware melhoram o desempenho.
- Speedup de 2.29x com 4 threads é útil, mas a eficiência cai para 57%.

### 3. Recomendações:

- Para CPU-bound: Use até 2 threads para máxima eficiência.
- Para memory-bound: Limite a 2 threads para evitar saturação do barramento.

## 7. Reflexão Final

O multithreading mostrou-se mais eficaz em CPU-bound (eficiência de 108% com 2 threads), enquanto em memory-bound os ganhos foram limitados pelo gargalo de memória (speedup de 2.29x com 4 threads, mas eficiência de 57%). O Hyper-Threading trouxe benefícios marginais em ambos os casos, mas a competição por recursos compartilhados (ULAs em CPU-bound, banda de memória em memory-bound) confirma que o paralelismo ideal nesta arquitetura não ultrapassa o número de núcleos físicos (2).