

Faculdade de Computação e Informática – FCI

Laboratório de Sistemas Operacionais

Prof. Jamilson



OpenMP

Sistemas Operacionais Turma 05N11

Nome do Aluno	RA:
Felipe Gyotoku Koike	10409640
Jônatas de Brito Silva	10403674
Bruno Viana Tripoli Barbosa	10409547

Lab 1 - Programação - Integral Regra do Trapézio (em aula)

Lab 1 - Programação - Integral Regra do Trapézio

Código fonte sequencial:

```
%%writefile integral_omp.c

#include <stdio.h>
#include <math.h>
#include <time.h>

double f(double x) {
    return sin(x);
}

double trapezoidal_rule(double a, double b, int n) {
    double h = (b - a) / n;
    double approx = (f(a) + f(b)) / 2.0;

    for (int i = 1; i <= n - 1; i++) {
        double x_i = a + i * h;
        approx += f(x_i);
    }

    approx = h * approx;
    return approx;
}

int main() {
    double a = 0.0, b = M_PI;
    int n = 100000000;

    clock_t start_time = clock();

    double result = trapezoidal_rule(a, b, n);

    clock_t end_time = clock();
    double execution_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
```

```

    printf("Resultado: %f\n", result);
    printf("Tempo de execução (sequencial): %f segundos\n",
execution_time);

    return 0;
}

```

```

!gcc -fopenmp integral_omp.c -lm -o integral_omp
!./integral_omp

Resultado: 2.000000
Tempo de execução (sequencial): 2.059622 segundos

```

Código fonte implementação paralela e melhorias:

```

%%writefile integral_paralel_omp.c

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

double f(double x) {
    return sin(x);
}

double trapezoidal_rule(double a, double b, int n) {
    double h = (b - a) / n;
    double approx = (f(a) + f(b)) / 2.0;

    #pragma omp parallel for reduction(+:approx)
    for (int i = 1; i <= n - 1; i++) {
        double x_i = a + i * h;
        approx += f(x_i);
    }

    approx = h * approx;
    return approx;
}

int main() {

```

```

double a = 0.0, b = M_PI;
int n = 1000000000;

double start_time = omp_get_wtime();

double result = trapezoidal_rule(a, b, n);

double end_time = omp_get_wtime();
double execution_time = end_time - start_time;

printf("Resultado: %f\n", result);
printf("Tempo de execução (OpenMP): %f segundos\n", execution_time);

return 0;
}

```

```

!gcc -fopenmp integral_paralel_omp.c -lm -o integral_paralel_omp
!./integral_paralel_omp

```

```

Resultado: 2.000000
Tempo de execução (OpenMP): 1.581531 segundos

```

Lab 2 - Avaliação de desempenho

Código Fonte Critical:

```

%%writefile somatorio_crit_openmb.c

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

#define SIZE (1 << 30)
#define REPS 10

// Preenchendo o vetor somente com 1.
void fill_array(int *arr, long long size) {
    for (long long i = 0; i < size; i++) {
        arr[i] = 1;
    }
}

```

```

double sum_with_critical(int *arr, long long size, int threads) {
    long long sum = 0;
    double start_time, end_time;

    start_time = omp_get_wtime();

    #pragma omp parallel num_threads(threads)
    {
        long long partial_sum = 0;
        #pragma omp for
        for (long long i = 0; i < size; i++) {
            partial_sum += arr[i];
        }
        #pragma omp critical
        sum += partial_sum;
    }
    end_time = omp_get_wtime();
    return end_time - start_time;
}

void run_experiment(int *arr, long long size) {
    int threads[] = {1, 2, 3, 4, 5, 6};
    double time_critical[6] = {0};

    // Realizando 10 execuções para média
    for (int r = 0; r < REPS; r++) {
        for (int i = 0; i < 6; i++) {
            time_critical[i] += sum_with_critical(arr, size, threads[i]);
        }
    }

    // Calculando a média
    for (int i = 0; i < 6; i++) {
        time_critical[i] /= REPS;
    }

    // Imprimindo a tabela
    printf("Threads\tCritical\n");
    for (int i = 0; i < 6; i++) {
        printf("%d\t%f\n", threads[i], time_critical[i]);
    }
}

```

```

int main() {
    // Alocando o vetor dinamicamente para suportar 2^30 elementos
    int *arr = (int*) malloc(SIZE * sizeof(int));

    if (arr == NULL) {
        printf("Erro ao alocar o vetor\n");
        return 1;
    }

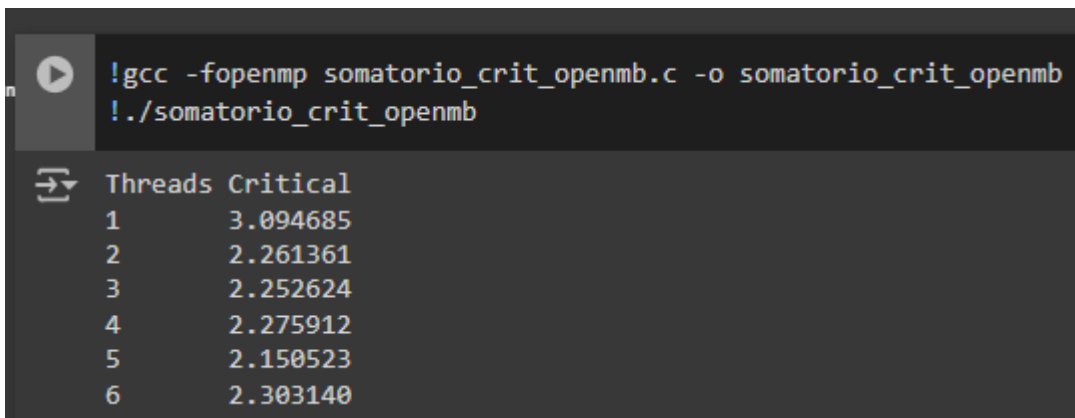
    fill_array(arr, SIZE);

    run_experiment(arr, SIZE);

    // Liberando a memória alocada
    free(arr);

    return 0;
}

```



```

!gcc -fopenmp somatorio_crit_openmb.c -o somatorio_crit_openmb
!./somatorio_crit_openmb

```

Threads	Critical
1	3.094685
2	2.261361
3	2.252624
4	2.275912
5	2.150523
6	2.303140

Código Fonte Reduction:

```

%%writefile somatorio_reduc_openmb.c

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

#define SIZE (1 << 30)
#define REPS 10

// Preenchendo o vetor somente com 1.

```

```

void fill_array(int *arr, long long size) {
    for (long long i = 0; i < size; i++) {
        arr[i] = 1;
    }
}

double sum_with_reduction(int *arr, long long size, int threads) {
    long long sum = 0;
    double start_time, end_time;

    start_time = omp_get_wtime();

    #pragma omp parallel for num_threads(threads) reduction(+:sum)
    for (long long i = 0; i < size; i++) {
        sum += arr[i];
    }

    end_time = omp_get_wtime();

    return end_time - start_time;
}

void run_experiment(int *arr, long long size) {
    int threads[] = {1, 2, 3, 4, 5, 6};
    double time_reduction[6] = {0};

    // Realizando 10 execuções para média
    for (int r = 0; r < REPS; r++) {
        for (int i = 0; i < 6; i++) {
            time_reduction[i] += sum_with_reduction(arr, size,
threads[i]);
        }
    }

    // Calculando a média
    for (int i = 0; i < 6; i++) {
        time_reduction[i] /= REPS;
    }

    // Imprimindo a tabela
    printf("Threads\tReduction\n");
    for (int i = 0; i < 6; i++) {
        printf("%d\t%f\n", threads[i], time_reduction[i]);
    }
}

```

```

    }
}



int main() {
    // Alocando o vetor dinamicamente para suportar 2^30 elementos
    int *arr = (int*) malloc(SIZE * sizeof(int));

    if (arr == NULL) {
        printf("Erro ao alocar o vetor\n");
        return 1;
    }
    fill_array(arr, SIZE);
    run_experiment(arr, SIZE);

    // Liberando a memória alocada
    free(arr);

    return 0;
}

```



2min

```
!gcc -fopenmp somatorio_reduc_openmb.c -o somatorio_reduc_openmb
!./somatorio_reduc_openmb
```

Threads	Reduction
1	2.972464
2	2.162520
3	2.098185
4	2.137001
5	2.356469
6	2.391673

Lab 3 - Nova multiplicação de matrizes

Código fonte e execução multiplicação de matriz linear :


```

%writefile mult_sequencial.c

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

void multiply_matrices(int **A, int **B, int **C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int N = 1000; // Tamanho da matriz NxN
    int **A, **B, **C;

    // Alocação dinâmica das matrizes
    A = (int **)malloc(N * sizeof(int *));
    B = (int **)malloc(N * sizeof(int *));
    C = (int **)malloc(N * sizeof(int *));
    for (int i = 0; i < N; i++) {
        A[i] = (int *)malloc(N * sizeof(int));
        B[i] = (int *)malloc(N * sizeof(int));
        C[i] = (int *)malloc(N * sizeof(int));
    }
}

```

```

// Inicialização das matrizes A e B com valores aleatórios
srand(time(NULL));
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        A[i][j] = rand() % 100;
        B[i][j] = rand() % 100;
    }
}

// Medição do tempo de execução
clock_t start = clock();
multiply_matrices(A, B, C, N);
clock_t end = clock();

double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Tempo de execução (Sequencial): %f segundos\n", time_taken);

// Liberação da memória alocada
for (int i = 0; i < N; i++) {
    free(A[i]);
    free(B[i]);
    free(C[i]);
}
free(A);
free(B);
free(C);


return 0;
}

```

```

[34] !gcc -o mult_sequencial mult_sequencial.c -fopenmp
!./mult_sequencial

```

 Tempo de execução (Sequencial): 12.811685 segundos

Código fonte e execução multiplicação de matrizes OpenMP.



%%writefile mult_mp.c

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

void multiply_matrices(int **A, int **B, int **C, int N) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int N = 1000; // Tamanho da matriz NxN
    int **A, **B, **C;

    // Alocação dinâmica das matrizes
    A = (int **)malloc(N * sizeof(int *));
    B = (int **)malloc(N * sizeof(int *));
    C = (int **)malloc(N * sizeof(int *));
    for (int i = 0; i < N; i++) {
        A[i] = (int *)malloc(N * sizeof(int));
        B[i] = (int *)malloc(N * sizeof(int));
        C[i] = (int *)malloc(N * sizeof(int));
    }
}
```

```

}

// Inicialização das matrizes A e B com valores aleatórios
srand(time(NULL));
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        A[i][j] = rand() % 100;
        B[i][j] = rand() % 100;
    }
}

int num_threads[] = {1, 2, 3, 4, 5, 6};
double time_taken[6];

for (int t = 0; t < 6; t++) {
    omp_set_num_threads(num_threads[t]);
    double start = omp_get_wtime();

    multiply_matrizes(A, B, C, N);

    double end = omp_get_wtime();
    time_taken[t] = end - start;

    printf("Tempo de execução com %d threads: %f segundos\n", num_threads[t], time_taken[t])
}

// Liberação da memória alocada
for (int i = 0; i < N; i++) {
    free(A[i]);
    free(B[i]);
    free(C[i]);
}
}
]

```

```

}
free(A);
free(B);
free(C);

return 0;
}

```

```

[36] !gcc -o mult_mp mult_mp.c -fopenmp
!./mult_mp

```

```

⇒ Tempo de execução com 1 threads: 13.497641 segundos
Tempo de execução com 2 threads: 13.946802 segundos
Tempo de execução com 3 threads: 13.409357 segundos
Tempo de execução com 4 threads: 13.149557 segundos
Tempo de execução com 5 threads: 13.105672 segundos
Tempo de execução com 6 threads: 12.735129 segundos

```

Tabela execução linear:

Execução	Tempo (s)
Execução 1	12.811685
Execução 2	13.069774
Execução 3	12.603524
Média	12.828328

Tabela execução OpenMP:

Threads	Execução 1	Execução 2	Execução 3	Média
1	13.497641	12.928768	11.846501	12.75764
2	13.946802	15.039649	13.567159	14.18454
3	13.409357	13.152921	12.930875	13.16438
4	13.149557	12.891325	12.612827	12.88457
5	13.105672	12.740014	12.674768	12.84015
6	12.735129	12.571059	12.271789	12.52599

gráfico speedup:

