

ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

## 7º LISTA DE ALGORITMO

1. Estude e apresente como as primitivas de paralelismo *spawn*, *sync*, and *parallel for* podem ser relacionadas com o padrão e modelo de programação OpenMP.

### Resposta:

O OpenMP é historicamente ligado ao Fortran por várias razões:

- **Origens no Fortran 77:** O OpenMP teve suas raízes no Fortran77, que era uma linguagem de programação muito utilizada na comunidade científica e de engenharia. No início, o OpenMP foi projetado principalmente para adicionar paralelismo a programas em Fortran, pois essa linguagem era amplamente usada em aplicações técnicas e científicas.
- **Demanda da Comunidade de HPC (High-Performance Computing ou Computação de alta Performance)** que refere-se a uma área da computação dedicada ao uso de supercomputadores e clusters de computadores de alto desempenho para resolver problemas computacionais complexos e intensivos em processamento. O Fortran sempre foi uma linguagem de escolha em computação de alta performance (HPC), onde o paralelismo é essencial. A comunidade de HPC desempenhou um papel significativo na demanda por uma solução de programação paralela que se encaixasse bem com o Fortran.
- **Facilidade de uso no Fortran:** O Fortran, especialmente em suas versões mais antigas, tinha características que facilitavam a adição de diretivas de paralelismo, como a estrutura de loop natural do Fortran. Isso tornou o Fortran uma escolha conveniente para experimentar e desenvolver as primeiras implementações do OpenMP.
- **Tradição de programação paralela em Fortran:** Programadores Fortran já estavam acostumados a desenvolver código paralelo devido às demandas da computação científica e técnicas. O OpenMP se encaixou naturalmente nesse ambiente.

Muito embora o OpenMP tenha suas raízes no Fortran, ele evoluiu ao longo do tempo para ser uma especificação de programação paralela amplamente aplicável em várias linguagens. Hoje o OpenMP é compatível com C, C++, Fortran, e até mesmo com algumas extensões para outras linguagens. No entanto, sua associação histórica ao Fortran permanece como uma parte importante de sua história e uso.

Dessa forma, a migração de nomenclatura de primitivas de paralelismo pode ocorrer por várias razões, incluindo a evolução das tecnologias de programação paralela, a necessidade de padronização e a adaptação a diferentes linguagens de programação. No caso da relação entre “spawn”, “sync”, e “parallel for” com o modelo de programação OpenMP, pode haver algumas razões para a diferença nas nomenclaturas:

- **Padronização e portabilidade:** OpenMP é um padrão de programação paralela que visa fornecer uma abordagem consistente e portátil para o paralelismo em várias linguagens. Isso significa que as diretivas e primitivas do OpenMP têm nomes padronizados que podem ser implementados em diferentes ambientes e linguagens. “Spawn”, “sync”, e “parallel for”

podem ser nomes específicos de uma implementação ou linguagem de programação, enquanto o OpenMP utiliza termos mais genéricos que podem ser aplicados em diversas linguagens.

- **Compatibilidade com linguagens existentes:** O OpenMP é projetado para ser compatível com várias linguagens de programação, incluindo C, C++, e Fortran, como já foi mencionado. As diretivas e primitivas do OpenMP foram escolhidas para se encaixar bem com a sintaxe e semântica dessas linguagens, tornando a migração do código mais fácil para desenvolvedores que já estão familiarizados com essas linguagens.
- **Evolução e ampliação de recursos:** O OpenMP tem evoluído ao longo do tempo para adicionar novos recursos e funcionalidades. À medida que novas capacidades foram introduzidas, os nomes e conceitos podem ter mudado para refletir essas edições, tornando o modelo mais rico e versátil.

Uma vez compreendido a questão da migração podemos definir que as primitivas de paralelismo “spawn”, “sync” e “parallel for” podem ser relacionadas ao padrão e modelo de programação OpenMP da seguinte forma:

- **Spawn (Criar Tarefas):** A primitiva “spawn” está relacionada ao conceito de criar tarefas paralelas em sistemas de programação paralela. No contexto do OpenMP, isso se assemelha ao uso de diretivas “parallel” e “task” para criar tarefas paralelas. O “spawn” pode ser comparado às tarefas criadas com “task”, e a sincronização subsequente pode ser controlada com “sync” em OpenMP.
- **Sync (Sincronização):** A primitiva “sync” é essencial para coordenar a execução paralela, assim como a diretiva “barrier” ou “task wait” em OpenMP. Ambas são usadas para garantir que as threads ou tarefas paralelas alcancem um ponto de sincronização antes de continuar a execução.
- **Parallel for (Laço Paralelo):** “Parallel for” é uma primitiva que permite paralelizar iterações de um loop. No OpenMP, pode-se obter funcionalidades semelhante usando a diretiva “parallel for” ou “parallel do”. Ambas permitem que você distribua iterações de um loop entre várias threads para acelerar o processamento.

Em resumo, o modelo de programação OpenMP fornece diretrizes e diretivas para criar paralelismo, controlar a sincronização e paralelizar loops, sendo semelhante em função às primitivas “spawn”, “sync” e “parallel for” mencionadas, embora os detalhes de implementação possam variar dependendo da linguagem de programação e da biblioteca utilizada.

2. Escolha um dos algoritmos que já implementou nas listas anteriores que poderiam se beneficiar de paralelismo e implemente-os utilizando OpenMP.

### Implementação da Multiplicação de Matriz de Forma Paralela com OpenMP em C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <omp.h>
5
6 // Função para multiplicação de matrizes quadradas
7 void multiply_matrices(int **A, int **B, int **C, int size) {
8     int num_threads = 1; // Número de threads OpenMP que você deseja usar
9     // Inicialize o OpenMP com o número de threads desejado
10    omp_set_num_threads(num_threads);
11    int i,j,k;
12    #pragma omp parallel private(i, j, k)
13    #pragma omp for
14    for (i = 0; i < size; i++) {
15        for (j = 0; j < size; j++) {
16            C[i][j] = 0;
17            for (k = 0; k < size; k++) {
18                C[i][j] += A[i][k] * B[k][j];
19            }
20        }
21    }
22 }
23 void print_matrix(int **matrix, int size) {
24     for (int i = 0; i < size; i++) {
25         for (int j = 0; j < size; j++) {
26             printf("%d ", matrix[i][j]);
27         }
28         printf("\n");
29     }
30 }
31 int main() {
32     int size; //≠ 600; // Tamanho das matrizes quadradas
33     printf("Digite o tamanho da matriz quadrada: ");
34     scanf("%d", &size);
35
36     int **matrixA = malloc(size * sizeof(int*));
37     for (int i = 0; i < size; i++) {
38         matrixA[i] = malloc(size * sizeof(int));
39     }
40
41     int **matrixB = malloc(size * sizeof(int*));
42     for (int i = 0; i < size; i++) {
43         matrixB[i] = malloc(size * sizeof(int));
44     }
```

```

46  int **result = malloc(size * sizeof(int*));
47  for (int i = 0; i < size; i++) {
48      result[i] = malloc(size * sizeof(int));
49  }
50
51  // Inicialize as matrizes com valores aleatórios entre 0 e 5
52  srand(time(NULL)); // Inicializa a semente do gerador de números aleatórios
53  for (int i = 0; i < size; i++) {
54      for (int j = 0; j < size; j++) {
55          matrixA[i][j] = rand() % 6; // Valores entre 0 e 5
56          matrixB[i][j] = rand() % 6;
57      }
58  }
59  //printf("Matrix A:\n");
60  //print_matrix(matrixA, size);
61  //printf("Matrix B:\n");
62  //print_matrix(matrixB, size);
63
64  //clock_t start_time = clock();
65  double start_clock = (double)clock() / CLOCKS_PER_SEC;
66  double start_time = omp_get_wtime();
67
68  multiply_matrices(matrixA, matrixB, result, size);
69
70  double end_clock = (double)clock() / CLOCKS_PER_SEC;
71  double end_time = omp_get_wtime();
72  //clock_t end_time = clock();
73
74  printf("Resultado da multiplicação:\n");
75  print_matrix(result, size);
76
77  double cpu_time = end_clock - start_clock;
78  double wall_time = end_time - start_time;
79
80  //double time_taken = (double)(end_time - start_time) / (double)CLOCKS_PER_S
81  //printf("Tempo para multiplicação: %f segundos\n", time_taken);
82
83  printf("Tempo de CPU: %lf segundos\n", cpu_time);
84  printf("Tempo de relógio: %lf segundos\n", wall_time);
85
86  // Libere a memória alocada
87  for (int i = 0; i < size; i++) {
88      free(matrixA[i]);
89      free(matrixB[i]);
90      free(result[i]);
91  }
92  free(matrixA);
93  free(matrixB);
94  free(result);
95
96  return 0;
97 }

```

## Resultados

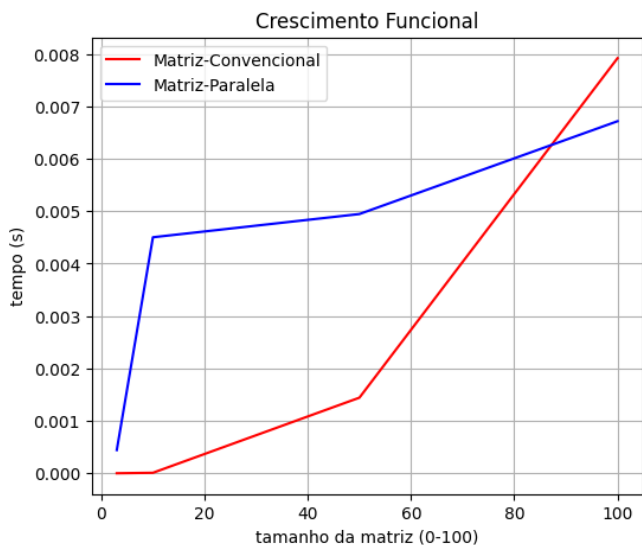


Gráfico 1

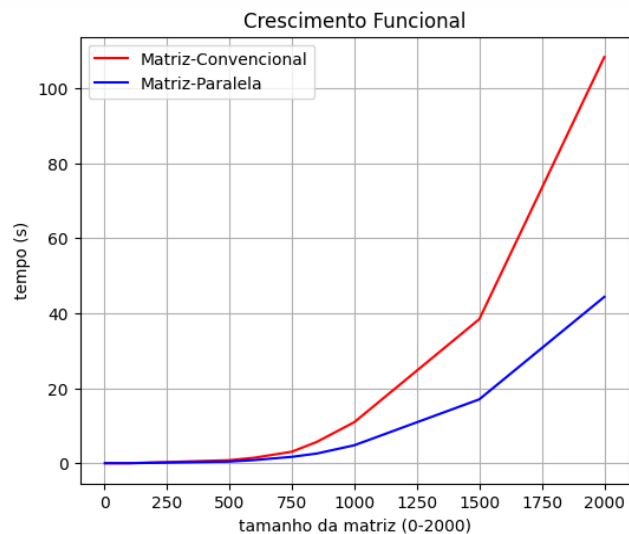


Gráfico 2

Foram feitos os testes e o resultado é o mostrado nos gráficos 1 e 2. Estamos visualizando o mesmo gráfico, no entanto como foram escolhidos valores grandes para o tamanho da matriz (eixo x), não é possível perceber detalhes para os valores iniciais. No gráfico 1 notamos que para matrizes um pouco abaixo do tamanho  $n = 90$  o processamento paralelo é ineficaz em comparação com o processamento de thread unitário. É importante frisar que tomamos dois tempos, tempo de processamento de CPU e tempo de processamento de “relógio”, para processamento de único thread esses tempos são iguais, porém quando começamos a dividir em threads, começamos a obter valores distintos.

A medida que o tamanho da matriz aumenta os gráficos se invertem e conseguimos visualizar a eficiência do processo de paralelização das tarefas, de modo que, ao final do teste, quando tínhamos uma matriz de 2000x2000 a diferença é realmente acentuada. Nesse caso com basicamente 3 linhas de códigos tornamos nosso programa significativamente mais eficiente. A priori, plotamos o gráfico de paralelização com o número máximo de “núcleos” que o computador utilizado no experimento possui, no caso 4 “núcleos”, mas realizamos testes com o aumento gradativo. Mas de modo geral tivemos um aumento proporcional a medida que aumentávamos as threads, e passamos a inserir valores acima de 4, não houve nenhuma melhora nos tempos.



3. Apresente uma análise experimental do algoritmo implementado na questão 2 utilizando o NPAD para realizar suas medições.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <omp.h>
5
6 // Função para multiplicação de matrizes quadradas
7 void multiply_matrizes(int **A, int **B, int **C, int size) {
8     int num_threads = 4; // Número de threads OpenMP que você deseja usar
9     // Inicialize o OpenMP com o número de threads desejado
10    omp_set_num_threads(num_threads);
11    int i,j,k;
12    #pragma omp parallel private(i, j, k)
13    #pragma omp for
14    for (i = 0; i < size; i++) {
15        for (j = 0; j < size; j++) {
16            C[i][j] = 0;
17            for (k = 0; k < size; k++) {
18                C[i][j] += A[i][k] * B[k][j];
19            }
20        }
21    }
22 }
23 void print_matrix(int **matrix, int size) {
24     for (int i = 0; i < size; i++) {
25         for (int j = 0; j < size; j++) {
26             printf("%d ", matrix[i][j]);
27         }
28         printf("\n");
29     }
30 }
31 int main() {
32     int input[12] = {3,10,50,100, 500,600,750,850,1000,1500,2000, 5000};
33     float cpu[12];
34     float relógio[12];
35     for (int x = 0; x < 12; x++) {
36         int size = input[x]; // Tamanho das matrizes quadradas
37         //printf("Digite o tamanho da matriz quadrada: ");
38         //scanf("%d", &size);
39
40         int **matrixA = malloc(size * sizeof(int *));
41         for (int i = 0; i < size; i++) {
42             matrixA[i] = malloc(size * sizeof(int));
43         }
44
45         int **matrixB = malloc(size * sizeof(int *));
46         for (int i = 0; i < size; i++) {
47             matrixB[i] = malloc(size * sizeof(int));
48         }
49
50         int **result = malloc(size * sizeof(int *));
51         for (int i = 0; i < size; i++) {
52             result[i] = malloc(size * sizeof(int));
53         }
```

```

54
55 // Inicialize as matrizes com valores aleatórios entre 0 e 10
56 srand(time(NULL)); // Inicializa a semente do gerador de números aleató.
57 for (int i = 0; i < size; i++) {
58     for (int j = 0; j < size; j++) {
59         matrixA[i][j] = rand() % 11; // Valores entre 0 e 10
60         matrixB[i][j] = rand() % 11;
61     }
62 }
63
64 double start_clock = (double) clock() / CLOCKS_PER_SEC;
65 double start_time = omp_get_wtime();
66
67 multiply_matrizes(matrixA, matrixB, result, size);
68
69 double end_clock = (double) clock() / CLOCKS_PER_SEC;
70 double end_time = omp_get_wtime();
71
72
73 //printf("Resultado da multiplicação:\n");
74 //print_matrix(result, size);
75
76 double cpu_time = end_clock - start_clock;
77 double wall_time = end_time - start_time;
78
79 cpu[x] = cpu_time;
80 relógio[x] = wall_time;
81 //printf("Tempo de CPU: %lf segundos\n", cpu_time);
82 //printf("Tempo de relógio: %lf segundos\n", wall_time);
83
84 // Libere a memória alocada
85 for (int i = 0; i < size; i++) {
86     free(matrixA[i]);
87     free(matrixB[i]);
88     free(result[i]);
89 }
90 free(matrixA);
91 free(matrixB);
92 free(result);
93 printf("Tempo de cpu %d: %lf segundos\n", (x+1), cpu[x]);
94 printf("Tempo de relógio %d: %lf segundos\n\n", (x+1), relógio[x]);
95 }
96 return 0;
97 }



```

## Resultados:

Para esse experimento fizemos o cadastro de conta junto ao Nucleo de Processamento de Alto Desempenho da UFRN – NPAD, para que pudessemos fazer uso do Super-PC. Uma vez cadastrados fizemos o login da conta e já dentro do servidor foi criado uma pasta de trabalho para guardar o script e demais arquivos.

```
[asdsilva@headnode0 ~]$ ls -l
total 4
drwxrwxr-x. 3 asdsilva macfernandes 4096 out 28 00:39 estudos
lrwxrwxrwx. 1 root      root          24 out 26 16:02 scratch -> /scratch/global/asdsilva
[asdsilva@headnode0 ~]$ ls
estudos  scratch
[asdsilva@headnode0 ~]$ cd estudos
[asdsilva@headnode0 estudos]$ ls
build_matriz_paralela.sh  matriz_paralela.out  sbatch_matriz_paralela.sh  slurm-298577.out  slurm-298579.out
matriz_paralela.c        matriz_paralela.sh   slurm-298574.out          slurm-298578.out  slurm-298581.out
[asdsilva@headnode0 estudos]$ cd ..
[asdsilva@headnode0 ~]$ cd scratch/
[asdsilva@headnode0 scratch]$ ls
[asdsilva@headnode0 scratch]$ cd ..
[asdsilva@headnode0 ~]$
```

Na pasta criada nomeada de “estudos” foram salvos arquivos de configuração necessários para “subir” o script para processamento no super computador.

	build_matriz_paralela.sh	66 bytes
	matriz_paralela.c	3,1 kB
	matriz_paralela.sh	175 bytes

build\_matriz\_paralela.sh

```
1 #!/bin/bash
2
3 gcc -fopenmp matriz_paralela.c -o matriz_paralela.out
```

matriz\_paralela.sh

```
1 #!/bin/bash
2
3 #SBATCH --job-name=matriz_paralela
4 #SBATCH --time=0-0:30
5 #SBATCH --cpus-per-task=16
6 #SBATCH --hint=compute_bound
7
8 export OMP_NUM_THREADS=16
9
10 ./matriz_paralela.out
```

Após o comando

```
[asdsilva@headnode0 ~]$ sbatch matriz_paralela.sh
```

o programa **matriz\_paralela** foi submetido a um JOB.

Com relação ao programa, fizemos uma pequena alteração na linha do **Main()** a partir da linha 32 do programa. Foi inserido um **for** para que o programa ficasse em loop lendo um array contendo os valores do tamanho da matriz, dessa forma foi possível fazer o teste no super computador de forma adequada. Abaixo é mostrado os resultados dos testes.



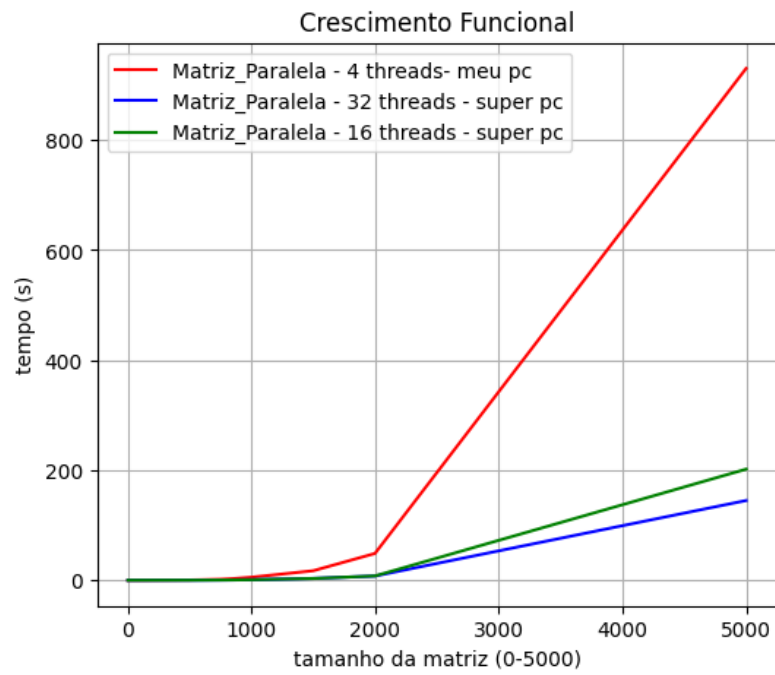


Gráfico 3

Baseado no gráfico 3 é possível notar que a diferença de tempo na execução do algoritmo do super computador com relação ao meu PC é muito grande. Foi feito teste com 16 e 32 threads.