**Programação Paralela - atividade III**

**Aluno:** Benjamim Marcos Bezerra de Menezes Pereira

# Relatório de Brenchmark MPI vs OpenMP - Resolução do problema de roteamento de veículos com JSA.

**Contextualização:**

O Problema de Roteamento de Veículos (VRP) é um dos desafios mais estudados na área de otimização combinatória, sendo amplamente aplicado em setores como logística, transporte e distribuição. O objetivo principal desse problema é determinar a melhor sequência de visitas a um conjunto de clientes, minimizando custos operacionais, como distância percorrida ou tempo de entrega. No entanto, devido à sua natureza NP-difícil, encontrar a solução ótima para grandes instâncias do VRP se torna inviável utilizando métodos exatos, o que justifica o uso de abordagens heurísticas e meta-heurísticas.

Entre os diversos algoritmos bioinspirados desenvolvidos para resolver esse problema, o Jellyfish Search Algorithm (JSA) se destaca por sua simplicidade e eficiência. Inspirado no comportamento coletivo das águas-vivas em busca de alimento, o JSA alterna entre um modo passivo de deriva, baseado no movimento das correntes marítimas, e um modo ativo de busca, onde os indivíduos se movimentam em direção a soluções promissoras. Essa abordagem favorece o equilíbrio entre exploração e exploração no espaço de busca, permitindo encontrar soluções de boa qualidade em um tempo reduzido.

Entretanto, como ocorre com muitas meta-heurísticas, o JSA pode demandar um alto custo computacional, especialmente para problemas de grande escala. Para mitigar essa limitação, a paralelização do algoritmo se torna uma alternativa viável para reduzir o tempo de execução. Neste relatório, serão analisadas duas estratégias de paralelização do JSA:

- **MPI (Message Passing Interface)**: Abordagem baseada em passagem de mensagens, ideal para sistemas distribuídos, onde múltiplos processos

comunicam-se de forma explícita para dividir o trabalho de busca pela melhor solução.

- **OpenMP (Open Multi-Processing)**: Modelo de paralelismo baseado em memória compartilhada, no qual diferentes threads trabalham simultaneamente em diferentes partes do problema, otimizando o uso de múltiplos núcleos de um mesmo processador.

Dessa forma, o objetivo deste trabalho é comparar o desempenho das implementações paralelas do JSA utilizando MPI e OpenMP, analisando métricas como tempo de execução e speedup. Para isso, serão realizados benchmarks em diferentes configurações de hardware, permitindo avaliar qual das abordagens apresenta melhores resultados para a resolução do VRP.

**Código MPI:**

O código foi compilado utilizando a IDE Visual Studio, enquanto a sua execução, alterando o número de processos para paralelização do código, foi feita pelo cmd do windows. Segue abaixo a explicação das principais funcionalidades do código paralelizado com MPI.

1. **Constantes:**

O uso das constantes serve para definir os parâmetros principais da implementação do JSA e também do VRP. Veja abaixo:

```
#define POP_SIZE 700
#define DIM 100  // Número de clientes
#define MAX_ITER 1000
#define BETA 3.0
#define ALPHA 0.5
```

- POP_SIZE: usado para definir o tamanho da população das soluções.
- DIM: define o número de clientes, referente ao VRP;
- MAX_ITER: serve para definir o número de iterações dos movimentos das águas vivas com os clientes do VRP.
- BETA e ALPHA: Parâmetros do JSA, apesar de não utilizados no código.

2. **Função cálculo de custo:**

Serve para receber a rota e a matriz de distâncias retorna o custo total dessa rota. Também faz referência à função objetivo principal do código do JSA com VRP.

```
// Função de custo (distância total percorrida)
double custo(int* rota, double distancias[DIM][DIM]) {
    double total = 0.0;
    for (int i = 0; i < DIM - 1; i++) {
        total += distancias[rota[i]][rota[i + 1]];
    }
    total += distancias[rota[DIM - 1]][rota[0]];
    return total;
}
```

3. **Função de inicialização da população:**

Serve para inicializar a população de soluções aleatórias. A sequência é embaralhada para criar soluções aleatórias.

```
// Inicializa população (rotas aleatórias)
void inicializarPopulacao(int populacao[POP_SIZE][DIM]) {
    for (int i = 0; i < POP_SIZE; i++) {
        for (int j = 0; j < DIM; j++) {
            populacao[i][j] = j;
        }
        for (int j = 0; j < DIM; j++) {
            int swapIdx = rand() % DIM;
            int temp = populacao[i][j];
            populacao[i][j] = populacao[i][swapIdx];
            populacao[i][swapIdx] = temp;
        }
    }
}
```

4. **Função de Perturbação:**

Recebe esse nome pois realiza uma troca aleatória de dois elementos da rota.

```
// Perturbação para exploração (movimento Browniano)
void perturbar(int* rota) {
    int a = rand() % DIM;
    int b = rand() % DIM;
    int temp = rota[a];
    rota[a] = rota[b];
    rota[b] = temp;
}
```

5. **Implementação Sequencial do JSA:**

É uma versão até mais simplificada da versão que está no código sequencial, servindo com o mesmo propósito. Ele se encontra no código para realizar o cálculo do SpeedUp na saída do código, para análise de resultados, funcionando de maneira coerente.

```c
// Algoritmo Jellyfish Search SEQUENCIAL
double jellyfishSearch_sequencial(double distancias[DIM][DIM], int* melhorRota) {
    int populacao[POP_SIZE][DIM];
    inicializarPopulacao(populacao);

    double melhorCusto = INFINITY;

    for (int iter = 0; iter < MAX_ITER; iter++) {
        for (int i = 0; i < POP_SIZE; i++) {
            double custoAtual = custo(populacao[i], distancias);
            if (custoAtual < melhorCusto) {
                melhorCusto = custoAtual;
                for (int j = 0; j < DIM; j++) {
                    melhorRota[j] = populacao[i][j];
                }
            }
            perturbar(populacao[i]);
        }
    }
    return melhorCusto;
}
```

6. **Função do JSA - implementada com MPI:**

A implementação do MPI ocorre dentro da função responsável pela execução do JSA para a resolução do VRP. O uso do MPI é necessário para gerenciar a comunicação entre os processos gerados, permitindo a distribuição das tarefas entre as demais funções do código.

```c
double jellyfishSearch_mpi(double distancias[DIM][DIM], int rank, int num_procs, int* melhorRotaLocal) {
    int populacao[POP_SIZE][DIM];
    inicializarPopulacao(populacao);

    double melhorCustoLocal = INFINITY;

    int inicio = (POP_SIZE / num_procs) * rank;
    int fim = (rank == num_procs - 1) ? POP_SIZE : (POP_SIZE / num_procs) * (rank + 1);

    double tempo_inicio_local = MPI_Wtime();

    for (int iter = 0; iter < MAX_ITER; iter++) {
        for (int i = inicio; i < fim; i++) {
            double custoAtual = custo(populacao[i], distancias);
            if (custoAtual < melhorCustoLocal) {
                melhorCustoLocal = custoAtual;
                for (int j = 0; j < DIM; j++) {
                    melhorRotaLocal[j] = populacao[i][j];
                }
            }
            perturbar(populacao[i]);
        }
    }

    double tempo_fim_local = MPI_Wtime();
    double tempo_exec_local = tempo_fim_local - tempo_inicio_local;

    // Redução para encontrar a melhor solução global
    double melhorCustoGlobal;
    MPI_Reduce(&melhorCustoLocal, &melhorCustoGlobal, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
    MPI_Bcast(&melhorCustoGlobal, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return tempo_exec_local;
}
```

Originalmente, a função recebe apenas a matriz de distâncias como parâmetro. No entanto, ao ser utilizada em um código paralelo com MPI, também são passados parâmetros adicionais essenciais para a comunicação entre os processos, como *rank* e *num_procs*. Além disso, um parâmetro chamado *melhorRotaLocal* é utilizado para o processamento das populações.

A função divide a população entre os processos MPI, permitindo o processamento paralelo, de modo que cada processo trabalhe em seu próprio subconjunto da população.

No *loop* principal, cada processo avalia apenas seu conjunto de rotas, atualizando sua melhor solução local. O tempo de execução é registrado para que, posteriormente, seja possível calcular o *speedup*.

Por fim, são utilizadas as funções `MPI_Reduce` e `MPI_Bcast`: a primeira permite que o processo de *rank* 0 receba a melhor solução global, enquanto a segunda garante que todos os processos tenham acesso ao melhor custo global.

7. **Função Main:**

A função principal do código, no caso do MPI, concentra a maior parte dos comandos relacionados à paralelização.

Inicialmente, são declaradas as variáveis necessárias e executadas as funções de inicialização do MPI. Em seguida, a matriz de distâncias é criada e preenchida dentro do primeiro *loop*.

Na *main*, são chamadas duas versões da função responsável pela implementação do Jellyfish: a sequencial e a paralelizada. Isso é feito para permitir o cálculo do *speedup*. A execução do código sequencial ocorre exclusivamente no processo de *rank* 0, sendo seu tempo de execução medido com `MPI_Wtime()`, tanto no início quanto no fim da execução, e armazenado em variáveis.

Após a execução do código sequencial, inicia-se a preparação para a execução paralela, levando em consideração o número de processos desejado. Para garantir que todos os processos começam simultaneamente, é utilizado `MPI_Barrier(MPI_COMM_WORLD)`, evitando que qualquer processo seja

executado antes dos demais. Em seguida, a função Jellyfish paralelizada é chamada, e o tempo de execução de cada processo é armazenado na variável `tempo_exec_local`, permitindo o cálculo do *speedup* e a exibição do tempo individual de cada processo. Os parâmetros da função Jellyfish são ajustados para que cada processo execute sua respectiva parte da computação.

Por fim, o tempo total de execução do código paralelo é exibido, e o cálculo do *speedup* global é realizado.

```c
int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, num_procs;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    srand(time(NULL) + rank);
    double distancias[DIM][DIM];

    for (int i = 0; i < DIM; i++) {
        for (int j = 0; j < DIM; j++) {
            distancias[i][j] = (i == j) ? 0.0 : (rand() % 100 + 1);
        }
    }

    int melhorRotaSeq[DIM];
    int melhorRotaPar[DIM];

    double tempo_inicio_seq, tempo_fim_seq, seq_time = 0.0;
    double melhorCustoSeq = 0.0;

    // Apenas rank 0 executa a versão SEQUENCIAL
    if (rank == 0) {
        tempo_inicio_seq = MPI_Wtime();
        melhorCustoSeq = jellyfishSearch_sequencial(distancias, melhorRotaSeq);
        tempo_fim_seq = MPI_Wtime();
        seq_time = tempo_fim_seq - tempo_inicio_seq;

        printf("==== EXECUCAO SEQUENCIAL ====\n");
        printf("Melhor custo sequencial: %.2f\n", melhorCustoSeq);
        printf("Tempo sequencial: %.6f segundos\n\n", seq_time);
    }

    // Todos aguardam antes de começar a execução paralela
    MPI_Barrier(MPI_COMM_WORLD);

    // Cada processo mede seu próprio tempo de execução
    double tempo_exec_local = jellyfishSearch_mpi(distancias, rank, num_procs, melhorRotaPar);

    // Todos os processos recebem o tempo sequencial para cálculo do speedup
    MPI_Bcast(&seq_time, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Cada processo calcula seu speedup local
    double speedup_local = seq_time / tempo_exec_local;
    printf("Rank %d -> Tempo local: %.6f segundos, Speedup: %.2f\n", rank, tempo_exec_local, speedup_local);

    // O rank 0 calcula o speedup global
    if (rank == 0) {
        printf("\n==== EXECUCAO PARALELA ====\n");
        printf("Tempo paralelo (media dos processos): %.6f segundos\n", tempo_exec_local);

        // Cálculo do speedup global
        double speedup_global = seq_time / tempo_exec_local;
        printf("\n==== SPEEDUP ====\n");
        printf("Speedup global: %.2f\n", speedup_global);
    }

    MPI_Finalize();
    return 0;
}
```

**Brenchmark e resultados:**

Os resultados de paralelização obtidos para MPI e OpenMP mostraram que ambas as abordagens proporcionaram melhorias no desempenho da aplicação. No entanto, a forma como cada uma escalona o speedup e a eficiência varia

significativamente. No MPI, o speedup aumentou consideravelmente para um conjunto de configurações com menor número de iterações, população e dimensão, enquanto no conjunto com valores maiores, o crescimento foi mais modesto. Já no OpenMP, o crescimento do speedup foi mais linear, aproveitando melhor a escalabilidade com o aumento de threads.

A eficiência caiu para ambos os métodos conforme mais threads e processos foram utilizados, mas no MPI essa queda foi mais acentuada, possivelmente devido ao overhead de comunicação entre processos. O OpenMP, por operar com threads dentro de um mesmo espaço de memória, manteve uma eficiência superior em comparação ao MPI para a mesma quantidade de núcleos utilizados.

Os testes foram conduzidos em dois hardwares distintos. O OpenMP foi testado em um notebook com processador Core i5 de 12ª geração, 16GB de RAM e uma placa de vídeo Nvidia GTX 1080 Ti. O MPI, por outro lado, foi testado em um hardware mais modesto, com um processador Core i5 de 5ª geração, 8GB de RAM e uma placa de vídeo Nvidia GTX 910m. As diferenças de hardware podem ter influenciado nos resultados, especialmente no comportamento do MPI.

1. **Testes de SpeedUp:**

Os testes para calcular as médias de speedup foram realizados com base no número de processos (para MPI) e no número de threads (para OpenMP) utilizados durante a execução do programa. Os experimentos foram conduzidos separadamente para cada abordagem, analisando o impacto da paralelização no desempenho do sistema. Abaixo, seguem as tabelas com os resultados obtidos.

- **MPI:**

Foram realizados testes com duas configurações distintas de parâmetros (*defines*), variando os valores da população, da dimensão (*DIM*) e do número máximo de iterações. No primeiro conjunto de testes, os valores utilizados foram 70, 10 e 100, respectivamente. No segundo, os valores foram aumentados para 700, 100 e 1000, tornando a carga computacional significativamente maior. Os resultados das execuções para ambas as configurações estão apresentados nas tabelas a seguir.

| Número de Processos | Média de Speedup - 10 execuções |
|---|---|
| 2 | 1,60 |
| 4 | 2,34 |
| 6 | 2,75 |
| 8 | 2,95 |

Defines: Pop_Size 700; DIM 100; MAX_ITER 1000

| Número de Processos | Média de SpeedUp - 10 execuções |
|---|---|
| 2 | 1,54 |
| 4 | 3.07 |
| 6 | 6.77 |
| 8 | 11.15 |

Defines: Pop_Size 70; DIM 10; MAX_ITER 100

Logo, é possível perceber que quanto menor o nível de iterações, população e dim, maior o crescimento de SpeedUp com variação do número de processos.

- **OpenMP:**

No caso do OpenMP, os testes foram conduzidos com um conjunto de parâmetros diferentes, utilizando os valores 1000, 200 e 10000 para população, *DIM* e iterações, respectivamente. Para cada configuração, foi realizada uma média do *speedup* ao longo de 10 execuções, variando o número de threads para analisar o impacto da escalabilidade no desempenho.

| Número de Threads | Média de SpeedUp - 10 execuções |
|---|---|
| 2 | 1,80 |
| 4 | 3,15 |
| 6 | 4,42 |
| 8 | 5,58 |

Defines POP_SIZE 1000; DIM 200; MAX_ITER 10000

2. **Valor de Eficiência.**

Os cálculos de eficiência apresentados a seguir foram obtidos a partir dos testes que utilizaram os *defines* com os valores mais altos, impondo uma carga de processamento mais intensa ao sistema. A eficiência foi analisada para ambas as abordagens (MPI e OpenMP), considerando diferentes quantidades de processos e threads ao longo da execução do programa. A tabela abaixo apresenta os valores obtidos, permitindo uma comparação entre as duas metodologias de paralelização.

| Nº de Processos/Threads | OpenMP (%) | MPI (%) |
|---|---:|---:|
| 2 | 90% | 80,00% |
| 4 | 79% | 58,50% |
| 6 | 74% | 45,83% |
| 8 | 70% | 36,88% |

**3. Conclusões:**

- Speedup:

No MPI, os testes mostraram que o speedup cresce mais rapidamente quando os valores de iterações, população e dimensão são menores. No caso com valores mais altos de Defines, o speedup cresce de forma mais contida. No OpenMP, o crescimento do speedup foi mais estável e previsível, sem grandes saltos, indicando um aproveitamento mais linear do paralelismo.

- Eficiência:

A eficiência diminui conforme o número de processos ou threads aumenta. Isso ocorre porque o overhead de comunicação no MPI se torna mais significativo à medida que mais processos precisam trocar informações. No OpenMP, a eficiência também diminui, mas de forma menos abrupta do que no MPI. Isso sugere que o OpenMP gerencia melhor o compartilhamento de recursos dentro de um único sistema, enquanto o MPI sofre mais impacto devido à necessidade de comunicação entre processos independentes.

- Comparação de Hardware:

O fato de o OpenMP ter sido testado em um hardware mais potente pode ter influenciado na sua eficiência e no crescimento do speedup.O MPI, rodando em um

hardware mais fraco, pode ter sofrido mais com gargalos no processamento, o que explica a queda de eficiência mais acentuada.