Parallel Ant Colony Optimization on Multi-Core CPUs

Accelerating the TSP Solution

Douglas Sousa Jorge, Isaac Brasil Oliveira

INF - UFG

Problema

- A Otimização por Colônia de Formigas (ACO) é uma meta-heurística popular para resolver problemas de otimização combinatória, como o Problema do Caixeiro Viajante (TSP).
- Implementações sequenciais de ACO são frequentemente limitadas pelo tempo de execução, especialmente em problemas de grande escala.
- Paralelizar o ACO pode melhorar significativamente a eficiência, mas apresenta desafios em termos de acesso e controle de dados.

Solução Proposta

- Proposta de um novo modelo de ACO paralelo para arquitetura de CPU multi-core.
- Introdução de uma abordagem de seleção proporcional à aptidão chamada Roda da Roleta Vetorial (VRW).
- Utilização de instruções vetoriais para acelerar a construção do tour de cada formiga.

Metodologia

- **Inicialização:**
  - Matrizes de distâncias, feromônios e heurísticas são inicializadas aleatoriamente.
- **Construção de Solução:**
  - Cada formiga constrói um tour baseado nos feromônios e nas heurísticas utilizando a VRW.
- **Atualização de Feromônios:**
  - Feromônios são atualizados com base nos tours construídos pelas formigas.
- **Medição de Tempo:**
  - Tempo de execução é medido para diferentes números de iterações (n).

Metodologia

- **Hardware utilizado:**
  - CPU: AMD Ryzen 5 5600 (6/12)
  - GPU: AMD Radeon RX 6750 XT
  - RAM: 16gb ddr4 3600mhz

Solução

```
Random random = new Random(42);

// Parâmetros
var alpha = 1.0;  // Importância do feromônio
var beta = 2.0;   // Importância heurística
var rho = 0.5;    // Taxa de evaporação
var numFormigas = 16;
var numCidades = 25;

// Inicialização aleatória de distâncias e feromônios
var distancias = new double[numCidades, numCidades];
var feromonios = new double[numCidades, numCidades];
var heuristicas = new double[numCidades, numCidades];

InitializarMatrizes(distancias, feromonios, heuristicas, numCidades);
```

Figure: Definições do programa

Solução

```
void InitializarMatrizes(double[,] distancias, double[,] feromonios, double[,]
heuristicas, int numCidades)
{
    for (var i = 0; i < numCidades; i++)
    {
        for (var j = 0; j < numCidades; j++)
        {
            if (i != j)
            {
                distancias[i, j] = random.NextDouble();
                feromonios[i, j] = 1.0;
                heuristicas[i, j] = 1.0 / distancias[i, j];
            }
            else
            {
                distancias[i, j] = 0.0;
                feromonios[i, j] = 1.0;
                heuristicas[i, j] = 0.0;
            }
        }
    }
}
```

Figure: Inicialização das matrizes

Solução

```
Func<double[,], double[,], int, int[]> construirSolucao = (feromonios, heuristicas,
numCidades) =>
{
    var tour = new int[numCidades];
    var visitadas = new bool[numCidades];
    var cidadeInicial = random.Next(numCidades);
    tour[0] = cidadeInicial;
    visitadas[cidadeInicial] = true;

    for (var i = 1; i < numCidades; i++)
    {
        var cidadeAtual = tour[i - 1];
        var probabilidades = new double[numCidades];

        for (var j = 0; j < numCidades; j++)
        {
            if (!visitadas[j])
            {
                probabilidades[j] = Math.Pow(feromonios[cidadeAtual, j], alpha) * Math.
Pow(heuristicas[cidadeAtual, j], beta);
            }
        }

        var somaProbabilidades = probabilidades.Sum();

        // Verificação para evitar divisão por zero
        if (somaProbabilidades == 0)
        {
            for (var j = 0; j < numCidades; j++)
            {
                if (!visitadas[j])
                {
                    probabilidades[j] = 1;
                }
            }
            somaProbabilidades = probabilidades.Sum();
        }
```

Figure: Construção da solução - primeira parte

Solução

```
        for (var j = 0; j < numCidades; j++)
        {
            probabilidades[j] /= somaProbabilidades;
        }

        // Verificação para garantir que as probabilidades são válidas
        if (probabilidades.Any(double.IsNaN))
        {
            throw new Exception("Probabilidades contém NaN");
        }

        var r = random.NextDouble();
        var acumulada = 0.0;
        var proximaCidade = -1;
        for (var j = 0; j < numCidades; j++)
        {
            acumulada += probabilidades[j];
            if (r <= acumulada)
            {
                proximaCidade = j;
                break;
            }
        }

        tour[i] = proximaCidade;
        visitadas[proximaCidade] = true;
    }

    return tour;
};
```

Figure: Construção da solução - segunda parte

Solução

```
Func<int[][], double[,], double[,], double, double[,]> atualizarFeromonios = (tours, distancias, feromonios, rho) =>
{
    var deltaFeromonios = new double[numCidades, numCidades];
    foreach (var tour in tours)
    {
        for (var i = 0; i < tour.Length - 1; i++)
        {
            deltaFeromonios[tour[i], tour[i + 1]] += 1.0 / distancias[tour[i], tour[i + 1]];
        }
        deltaFeromonios[tour[^1], tour[0]] += 1.0 / distancias[tour[^1], tour[0]];
    }

    for (var i = 0; i < numCidades; i++)
    {
        for (var j = 0; j < numCidades; j++)
        {
            feromonios[i, j] = (1 - rho) * feromonios[i, j] + deltaFeromonios[i, j];
        }
    }

    return feromonios;
};
```

Figure: Atualização dos feromônios

Solução

```
// Função ACO
Action<double[,], double[,], double[,], int, int, int> acoThread = (distancias, feromonios, heuristicas, numCidades, numFormigas,
numIteracoes) =>
{
    for (var iteracao = 0; iteracao < numIteracoes; iteracao++)
    {
        var tours = new int[numFormigas][];
        Parallel.For(0, numFormigas, i =>
        {
            tours[i] = construirSolucao(feromonios, heuristicas, numCidades);
        });

        feromonios = atualizarFeromonios(tours, distancias, feromonios, rho);
        Console.WriteLine($"Iteração {iteracao + 1}/{numIteracoes} concluída.");
    }
};

// Função para medir tempo de execução
Func<Action<double[,], double[,], double[,], int, int, int>, double[,], double[,], double[,], int, int, int, (double[,], double)>
medirTempoExecucao = (func, distancias, feromonios, heuristicas, numCidades, numFormigas, numIteracoes) =>
{
    var startTime = DateTime.Now;
    func(distancias, feromonios, heuristicas, numCidades, numFormigas, numIteracoes);
    var endTime = DateTime.Now;
    var tempoExecucao = (endTime - startTime).TotalSeconds;
    return (feromonios, tempoExecucao);
};
```

Figure: Chamada e medição do tempo de execução

Utilização CPU Paralelo

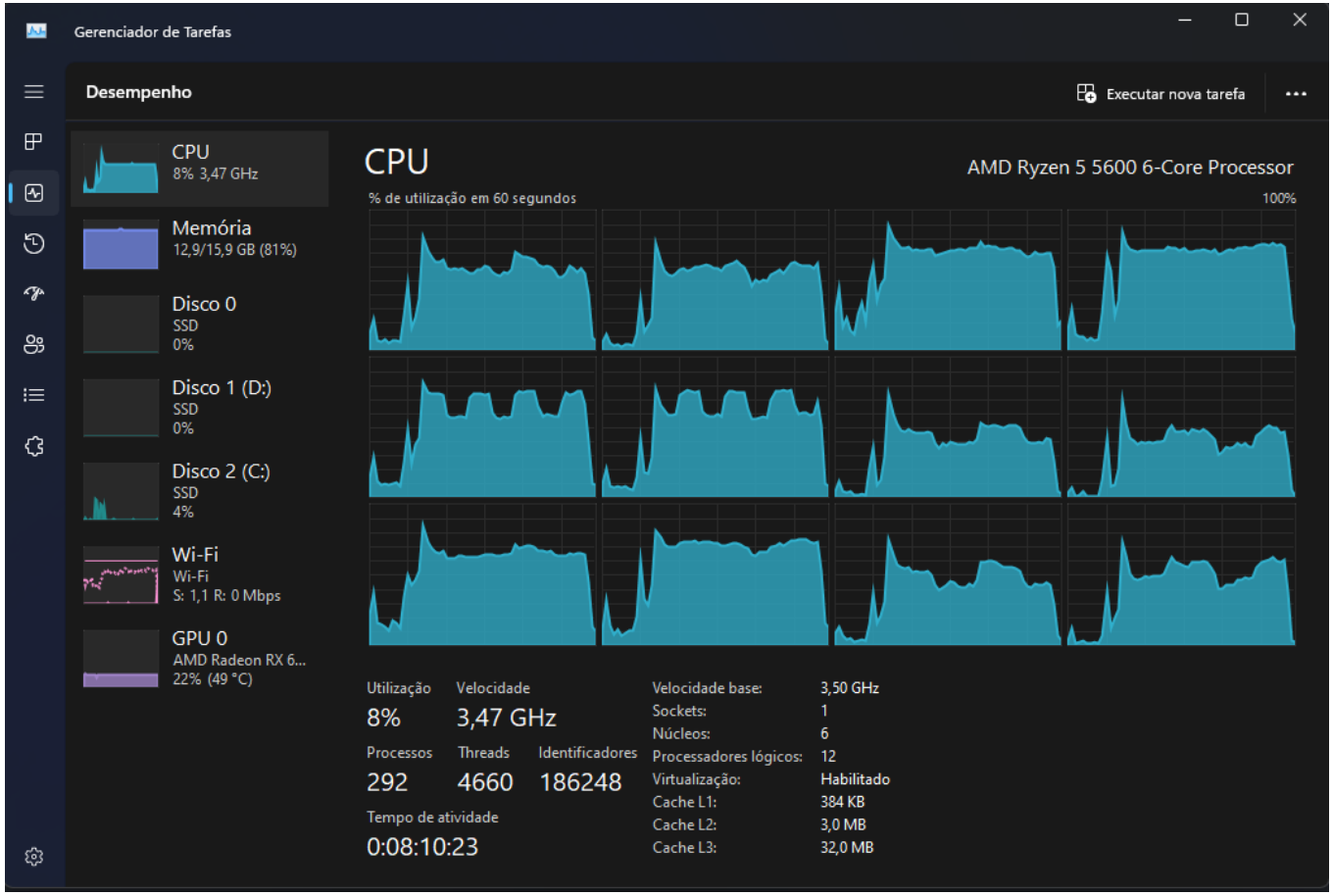

Figure: Utilização do CPU Paralelo

Resultados

- Comparação entre o algoritmo sequencial e o paralelo.
- Medição de tempos de execução e speedup.

Comparação de Speedup

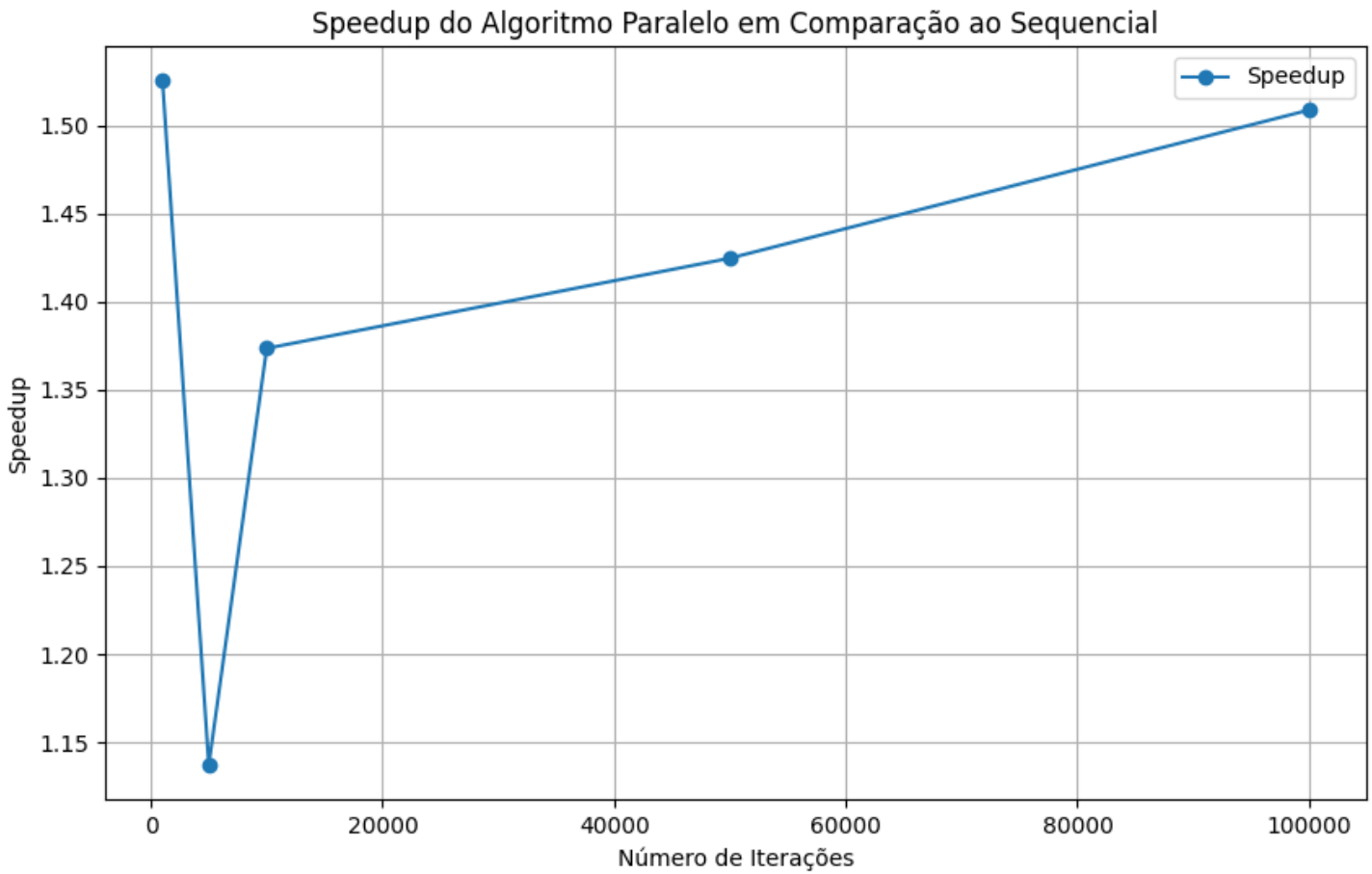

Figure: Comparação de Speedup

Comparação de Tempos de Execução

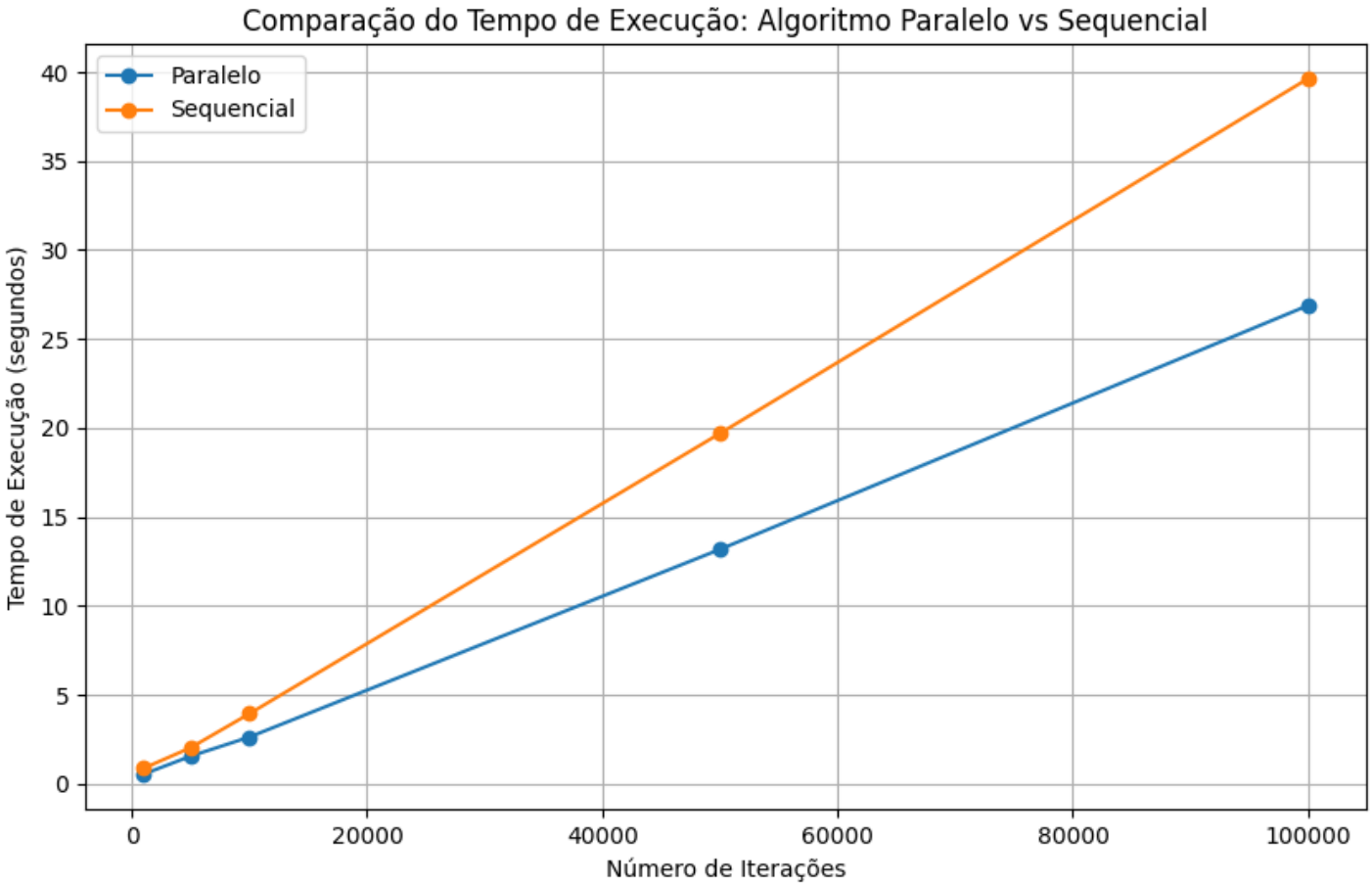

Figure: Comparação de Tempos de Execução

Conclusão

- O modelo proposto de ACO paralelo em CPUs multi-core mostrou uma aceleração significativa em comparação com a versão sequencial.
- A abordagem VRW e a utilização de instruções vetoriais melhoraram a eficiência da construção do tour.
- Resultados indicam o potencial de CPUs multi-core para resolver problemas de grande escala de forma eficiente.