Projeto Final SuperComp

Relatório de Implementação e Análise de Desempenho do problema de Otimização de Rotas de Veículos (Vehicle Routing Problem - VRP)

André Melo

---

1. Introdução

Este relatório descreve a abordagem utilizada para resolver o problema proposto, incluindo uma descrição das heurísticas e métodos de busca local e global adotados, bem como as estratégias de paralelização implementadas. Além disso, é apresentada uma análise de desempenho da solução proposta, incluindo tempos de execução e qualidade das soluções encontradas com base em instâncias de teste de diferentes tamanhos.

---

2. Algoritmo de busca global

Neste primeiro algoritmo foi necessário pensar em uma função exaustiva que deve gerar todas as possibilidades de solução do problema com base na combinatória dos inputs levando em conta todas as restrições. E a partir disso basta calcular a soma dos custos e escolher a solução mais barata.
Para isso, foi necessário gerar, em primeiro lugar, uma lista com todas as possibilidades de rotas que um único caminhão poderia realizar. Ou seja, todas as combinações de cidades que o veículo poderia traçar saindo e voltando ao armazém (nó *0*) e sem repetições de cidades intermediárias. O grande ponto do algoritmo é o fato de ele gerar todas as permutações de solução, como pode ser observado na a função a seguir:

```
void GerarTodasAsCombinacoesPossiveis(const vector<vector<int>>& rotas, int
num_cidades, vector<vector<int>>& resultados, vector<int>& demandas, int
capacidadeVeiculo) {
        // a função está melhor comentada no código fonte
        int n = rotas.size();
        vector<int> indices;
        for (int i = 0; i < n; ++i) {
```

```
                indices.push_back(i);
        }

        do {
                for (int i = 0; i < n; i += num_cidades) {
                        int j = min(i + num_cidades, n);
                        vector<int> sub(indices.begin() + i, indices.begin() +
j);
                        vector<int> rota_custo;
                        sub.insert(sub.begin(), 0);
                        sub.push_back(0);

                        if (calcularCusto(sub, rotas) != -1 &&
!rotaJaExiste(sub, resultados) && verificaCapacidade(sub, demandas,
capacidadeVeiculo) == 1) {
                                resultados.push_back(sub);
                        }
                }
        } while (next_permutation(indices.begin() + 1, indices.end()));
}
```

Então, o próximo passo foi permutar novamente essas rotas válidas geradas em itinerários válidos. Por exemplo: para uma rede de apenas 3 nós (0 1 2) a rede acima poderia gerar as rotas possíveis para um caminhão como (0 1 0), (0 2 0), (0 1 2 0) e (0 2 1 0). Mas essa ainda não é uma resposta final. Ainda é necessário estabelecer um itinerário para os caminhões seguirem para cumprirem com todas as demandas sem repetição, e é isso que a função *gerarTodasAsCombinacoesItinerario* faz. Nesse caso, essa função geraria (0 1 0 | 0 2 0), (0 1 2 0) e (0 2 1 0). A função está apresentada em detalhes a seguir.

```
void gerarCombinacoesRecursivo(const vector<vector<int>>& rotas, vector<int>&
combinacaoAtual,vector<vector<int>>& todasCombinacoes, int tamanho_itinerario,
int numVertices, int inicio) {

        if (combinacaoAtual.size() == tamanho_itinerario) {
                vector<vector<int>> vetor_de_verdade;
                for (int local : combinacaoAtual) {
                        vetor_de_verdade.push_back(rotas[local]);
                }
                if (checkAllNodesWithoutOverlap(vetor_de_verdade,
numVertices)){
                        todasCombinacoes.push_back(combinacaoAtual);
                }
                return;
        }
```

```
        for (size_t i = inicio; i < rotas.size(); i++) {
                combinacaoAtual.push_back(i);

                gerarCombinacoesRecursivo(rotas, combinacaoAtual,
todasCombinacoes, tamanho_itinerario, numVertices, i + 1);

                combinacaoAtual.pop_back();
        }
}

// Função principal para gerar todas as combinações de rotas
vector<vector<int>> gerarTodasAsCombinacoesItinerario(const
vector<vector<int>>& rotas, int numVertices) {

        vector<vector<int>> todasCombinacoes;
        vector<int> combinacaoAtual;
        // loop para gerar todos os tamanhos de combinação
        for (int k = 1; k ≤ rotas.size(); k++){
                gerarCombinacoesRecursivo(rotas, combinacaoAtual,
todasCombinacoes, k, numVertices, 0);
        }

        return todasCombinacoes;
}
```

Depois disso, tem-se um vetor completo com todos os arrays possíveis para cumprir todas as demandas e respeitar as restrições. Então basta calcular o menor custo somado desses itinerários e tem-se então a solução ótima do problema.

---

3. Heurística

3.1. Heurística de Clarke e Wright

Para a segunda implementação, foi escolhida a **Heurística de Clarke e Wright (Economias)**. Este método é eficiente e relativamente simples, baseado na ideia de economia de custos.
A ideia deste algoritmo é combinar duas paradas em uma única rota, em contraposição às duas individualmente e guardar isso em um array e ordená-lo. Depois, iterativamente deve-se combinar as rotas do array para encontrar a combinação com o menor custo possível. O trecho da função
clarkeWright que apresenta isso está exposto a seguir:

```
void clarkeWright(const vector<vector<int>>& distancias, int capacidade, const
vector<int>& demandas, int maxParadas) {

        int n = distancias.size() - 1;
        vector<vector<int>> rotas;
        for (int i = 1; i ≤ n; ++i) {
                rotas.push_back({0, i, 0});
        }

        vector<Economia> economias;
        for (int i = 1; i ≤ n; ++i) {
                for (int j = i + 1; j ≤ n; ++j) {
                        int valor = distancias[0][i] + distancias[0][j] -
distancias[i][j];
                        economias.push_back(Economia(i, j, valor));

                }

        }

        sort(economias.begin(), economias.end(), compararEconomias);

        for (const auto& economia : economias) {
        // itera sobre as economias salvas
}
```

3.2 Heurística greedy/gulosa (extra)

A maneira como essa heurística funciona é simples: ela sempre opta pelas rotas mais próxima, claro, sempre respeitando todas as restrições impostas no enunciado. O algoritmo falha em precisão pois ao sempre escolher a rota mais próxima ele está desconsiderando os próximos caminhos que se abrirão devido a esta escolha. Abaixo é mostrada como a função de inserção mais próxima funciona.

```
vector<vector<int>> insercaoMaisProxima(const vector<vector<int>>& distancias,
const vector<int>& demandas, int capacidade) {

    int n = distancias.size() - 1; // número de clientes (não inclui depósito)
    vector<vector<int>> rotas;
    vector<bool> visitado(n + 1, false);
    visitado[0] = true; // o depósito é sempre visitado

    for (int i = 1; i ≤ n; ++i) {
        if (visitado[i]) continue;
```

```
        vector<int> rota = {0, i};
        visitado[i] = true;
        int cargaAtual = demandas[i];

        while (true) {
            int melhorCliente = -1;
            int menorDistancia = numeric_limits<int>::max();
            for (int j = 1; j ≤ n; ++j) {
                if (!visitado[j] && cargaAtual + demandas[j] ≤ capacidade) {
                    int distancia = distancias[rota.back()][j];
                    if (distancia ≠ 0 && distancia < menorDistancia) {
                        menorDistancia = distancia;
                        melhorCliente = j;
                    }
                }
            }
            if (melhorCliente == -1) break;

            rota.push_back(melhorCliente);
            visitado[melhorCliente] = true;
            cargaAtual += demandas[melhorCliente];
        }
        rota.push_back(0); // retorna ao depósito
        if (calcularCusto(rota, distancias) ≠ -1) { // Verifica se a rota é
válida
            rotas.push_back(rota);
        }
    }
    return rotas;
}
```

---

4. Estratégias de Paralelização

4.1 Paralelização Local

A paralelização global foi feita a partir do algoritmo de busca global detalhado no ponto 2. Foram utilizados os *pragmas* para paralelizar as funções mais custosas do algoritmo, que no caso são *GerarTodasAsCombinacoesPossiveis(), gerarCombinacoesRecursivo()* e *calcula_menor_custo_itinerarios()*. Também foram adicionadas áreas críticas nas áreas de memória compartilhada para não haver condições de corrida.

4.2 Paralelização Global

A paralelização global é um incremental à paralelização local descrita anteriormente sobre o algoritmo força bruta, ela executa multiplas threads e ainda é capaz de rodas um várias máquinas diferentes. Assim distribuindo a carga ainda mais em um cluster e tornando a execução mais rápida. A implementação foi feita paralelizando a função principal, dividindo o trabalho igualmente para cada máquina requisitada. Foram adicionados 2 argumentos à função que resolve o VRP que dizem em qual partição o código está sendo executado para ele poder dividir o trabalho sem *overlap*. Como demonstrado na função *main()*:

```
int main(int argc, char* argv[]) {

        MPI_Init(&argc, &argv);
        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        int C = 15;
        int numVertices;

        vector<int> demandas = LerDestinoDemanda("grafo.txt", numVertices);
        vector<vector<int>> locais = LerRotasPossiveis("grafo.txt",
numVertices);

        auto start = high_resolution_clock::now();
        int resultado = ResolverVRPComDemanda(locais, demandas, C,
numVertices, rank, size);
        auto end = high_resolution_clock::now();

        if (rank == 0) {
                auto duration = duration_cast<milliseconds>(end -
start).count();
                cout << "Tempo de execução: " << duration << " ms" << endl;
        }

        MPI_Finalize();

        return 0;
}
```

---

5. Análise de Desempenho

5.1 Metodologia

A análise de desempenho foi feita utilizando a biblioteca *chrono* em C++ com a função *high_resolution_clock* para marcar o tempo de execução da função principal, como demonstrado a seguir:

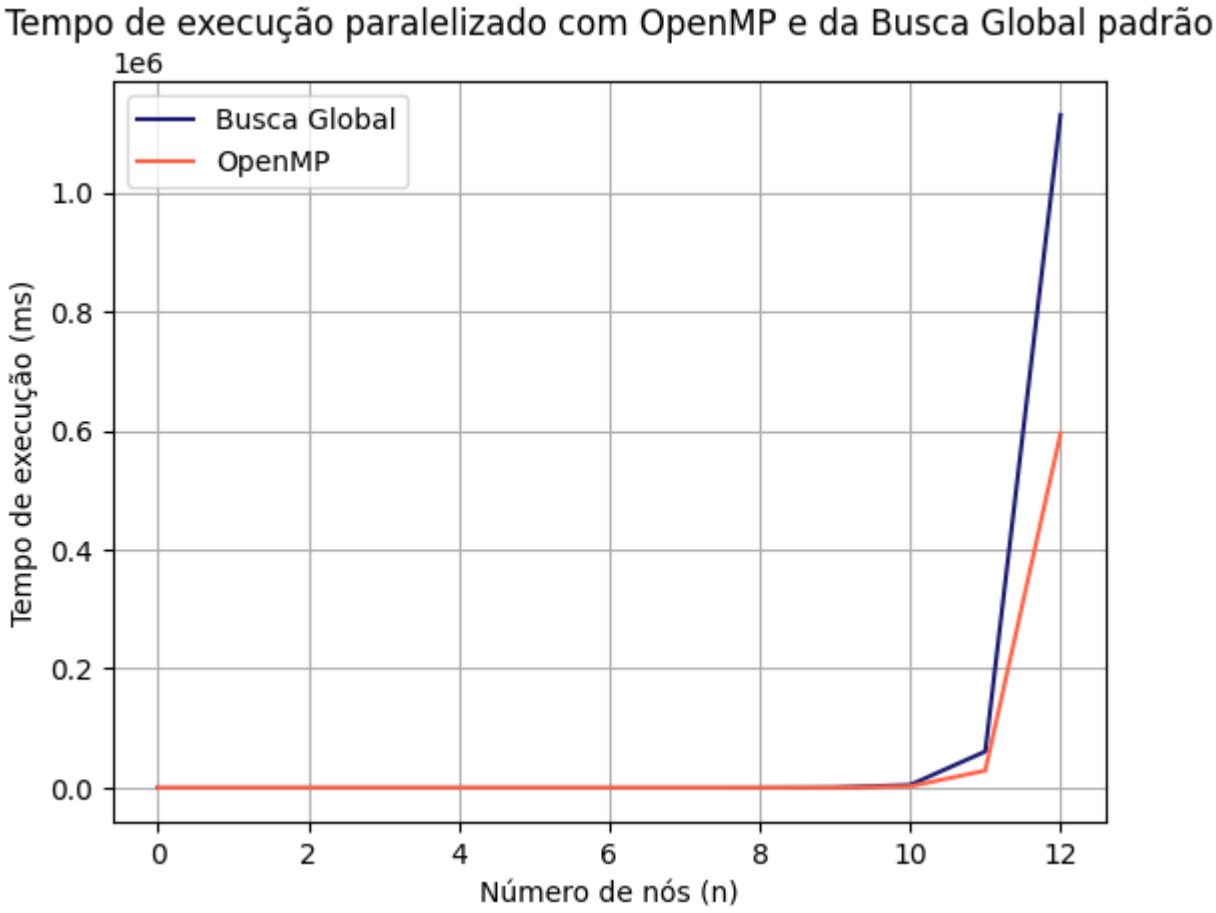
```
    auto start = high_resolution_clock::now();

    int resultado = ResolverVRPComDemanda(locais, demandas, C, numVertices);

    auto end = high_resolution_clock::now();

    auto duration = duration_cast<milliseconds>(end - start).count();
    cout << "\nMenor custo: " << resultado << "\n";
    cout << "Tempo de execução: " << duration << " ms" << endl;
```

Então, foram salvos em uma lista *python* todos os valores testados para os diferentes tamanhos de entrada e depois esses dados foram processados para criar os gráficos e os números presentes aqui neste documento.

5.2 Resultados

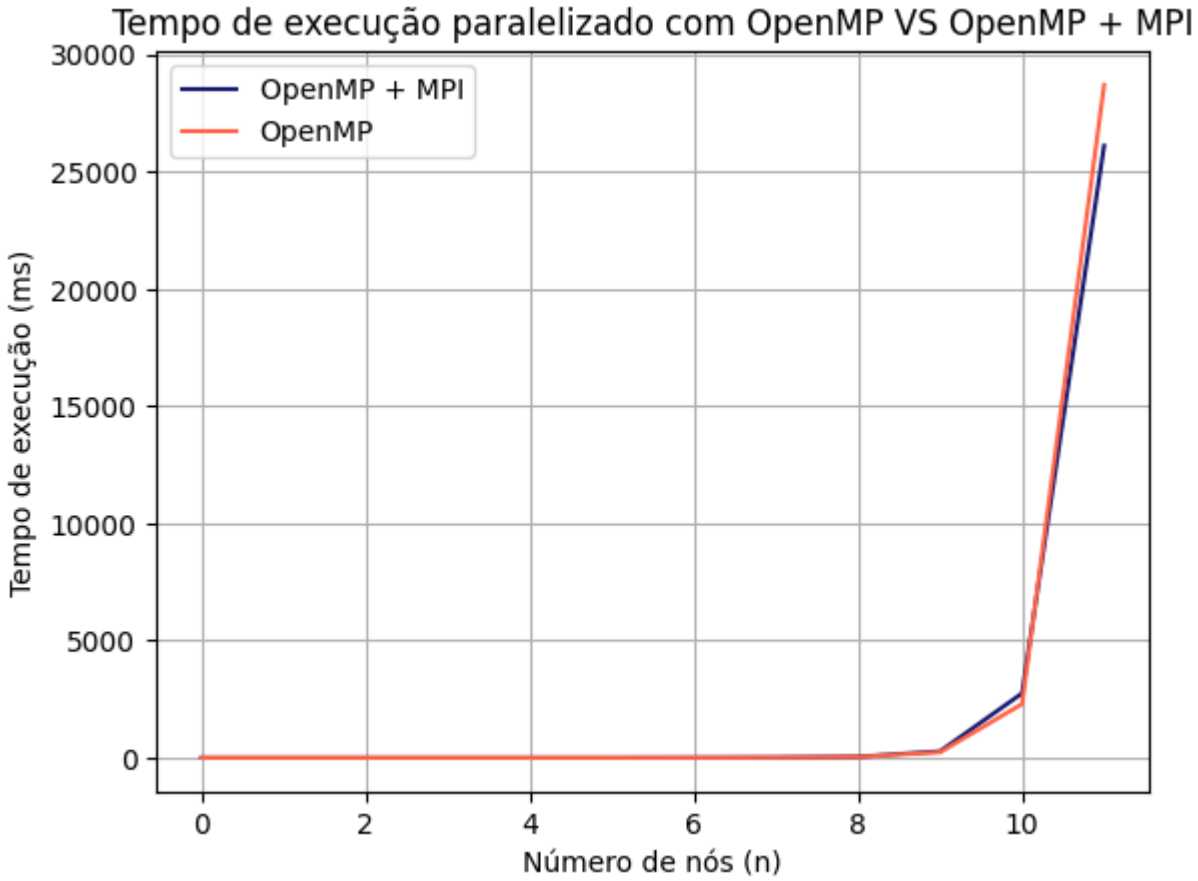
Apresentando os resultados da análise de desempenho em **Tempo de Execução**:

- Busca Global X Busca Global com OpenMP



O primeiro tempo de execução é a busca global simples, força bruta, em comparação com esse algoritmo paralelizado em threads com OpenMP, que fez com que o tempo de execução fosse reduzido em aproximadamente 52,6% para a situação de 12 nós no grafo

- Busca Global OpenMP X Busca Global OpenMP + MPI



De maneira contrária às expectativas, a utilização do MPI juntamente com o OpenMP não reduziu o tempo de execução do código de busca global. Em todos os testes, os tempos de execução permaneceram iguais ou às vezes até ligeiramente maiores do que os resultados com o OpenMP sozinho.

- Busca Global X Heurística:

A diferença entre ambos os algoritmos em termos de tempo é notável. Enquanto o tempo de execução da heurística de Clarke e Wright permanece efetivamente em 0 durante todo o teste (até cerca de 20 nós de entrada, na verdade), o algoritmo bruteforce crece exponencialmente, saindo de 8ms para n = 7 e chegando em 80s para n = 11. Fato que já era esperado já que por se tratar de um algoritmo de busca exaustiva, sua complexidade é fatorial.

- VRPY X Heurística:

A diferença também é surpreendente quando tratamos da diferença do desempenho entre a heurística desenvolvida e o VRPY, o gabarito. Apesar do algoritmo desenvolvido aqui ter apresentado resultados cerca de 20% mais caros que o VRPY, ele se mostrou violentamente mais rápido. Aos 210 nós em um grafo, o VRPY demorou cerca de 5 minutos para devolver um itinerário, enquanto o Clarke Wright apenas 200 ms!

---

6. Conclusão

Os resultados dos tempos de execução dos algoritmos seguiram exatamente como esperado, exceto pelo algoritmo MPI:

- Heurística de Clarke e Wright: A heurística de Clarke e Wright desenvolvida apresentou ótimo desemprenho em termos de rapidez de solução mas não foi tão boa assim em providenciar as melhores rotas (com menor custo) para grandes números de nós. Podemos então concluir que para entradas muito grandes ela é melhor até mesmo que a própria implementação do VRPY. Mas para entradas menores, o VRPY fornece soluções mais precisas em um tempo razoável.
- Heurística Gulosa: Apesar de apresentar resultados consideravelmente distantes da solução ótima, esta heurística se destaca pela sua velocidade. Mesmo com entradas

grandes (na casa dos 300 nós) este algoritmo foi capaz de dar uma solução para o problema quase que instantaneamente, em 6ms. Podemos concluir então que quando o tempo de solução é a prioridade máxima e para entradas muito grandes, esta pode ser uma solução válida.

- OpenMP: A utilização do OpenMP para acelerar a execução do código se mostrou um grande sucesso. A avaliação das permutações possíveis foi distribuída entre os núcleos disponíveis, permitindo um processamento mais rápido e eficiente. Esta distribuição equilibrou a carga de trabalho, reduzindo o tempo total necessário para gerar e avaliar todas as permutações de rotas em aproximadamente metade do normal.
- MPI: Enquanto OpenMP trouxe ganhos significativos ao reduzir o tempo de execução pela metade, a utilização de MPI não fez diferença significativa no desempenho. Isso pode ter acontecido por inúmeros fatores, inclusive uma implementação inadequada.

---

7. Referências

https://colab.research.google.com/drive/12HWu5McuGuov0xGIb2lecK-D2gTKoweJ?usp=sharing#scrollTo=fCj71MBieLJZ