Paralelização de uma Heurística em Aceleradores Gráficos para Resolução do Problema de Roteamento de Veículos

Péricles Pinheiro Feltrin[1], **Ana Paula Canal**[2]

[1]Acadêmico do curso de Ciência da Computação – Centro Universitário Franciscano
Rua dos Andradas, 1614 – 97010-032 – Santa Maria – RS – Brasil

[2]Orientadora. Professora do curso de Ciência da Computação – Centro Universitário Franciscano
Rua dos Andradas, 1614 – 97010-032 – Santa Maria – RS – Brasil

periclesfeltrin@gmail.com, apc@unifra.br

Abstract. *The Vehicle Routing Problem (VRP) has many applications in the real world, especially the VRP Capacitated and requires a lot of processing and time to find the optimal solution. This happens because it is a combinatorial optimization problem and belongs to the class of NP-Complete problems. Thus, one should choose to leave the optimal solution from the side, so you can get a good solution in a timely manner using a heuristic. So as to achieve a performance gain, this work uses OpenACC in order to perform parallel processing on the GPU (Graphics Processing Unit) the heuristic Iterated Local Search for Vehicle Routing Problem Capacitated. The heuristic achieved good results and parallelization in OpenACC brought performance gains for the application.*

Resumo. *O Problema de Roteamento de Veículos (PRV) possui muitas aplicações no mundo real, especialmente o PRV Capacitado e requer muito processamento e tempo para encontrar a solução ótima. Isso acontece porque é um problema de otimização combinatória e pertence à classe de problemas NP-Completos. Deste modo, deve-se optar por deixar a solução ótima de lado, para que seja possível obter uma boa solução em tempo hábil usando uma heurística. Ainda para alcançar um ganho de desempenho, este trabalho utiliza OpenACC a fim de realizar o processamento paralelo em GPU (Unidade de Processamento Gráfico) da heurística de Busca Local Iterada para o Problema de Roteamento de Veículos Capacitado. A heurística proposta obteve bons resultados e a paralelização em OpenACC trouxe ganho de desempenho para a aplicação.*

1. Introdução

O Problema de Roteamento de Veículos (PRV) é um problema de Otimização Combinatória e caracteriza-se por possuir muitas aplicações, como nas situações reais de entrega de mercadorias ou cartas, coleta de lixo e outros que demandam estabelecimento de rotas e múltiplos veículos. Assim, afeta a indústria, o comércio, o setor de serviços, a segurança, a saúde pública e o lazer [Goldbarg e Luna 2005]. Segundo [Toth e Vigo 2002, p.1] com a solução desse problema gera-se uma economia nos custos de transporte, geralmente de 5% a 20%.

Trata-se de um problema que está entre os mais complexos da área de Otimização Combinatória, pertencendo à classe dos problemas NP-Completos [Leiserson et al. 2002,

p.736–805], pois possui grande número de variáveis, diversidade de restrições e tempo de processamento, o que impõe um cuidado na escolha da taxonomia para tratar o problema [Goldbarg e Luna 2005]. Algumas classificações para o PRV são: Capacitados, com Janela de Tempo e Múltiplos Depósitos. Este trabalho pretende resolver o Problema Roteamento de Veículo Capacitado (PRVC), pois é a base para outros PRVs, visto que com algumas adaptações em suas restrições pode-se transformar em outros PRVs [Toth e Vigo 2002, p.5].

Com isso não é possível obter a solução ótima em tempo hábil, inclusive para instâncias relativamente pequenas. Assim, indo ao encontro da afirmação de [Tanenbaum 2001, p.315], embora os computadores estejam cada vez mais velozes, as exigências sobre eles crescem no mínimo mais rápido do que a sua velocidade de operação. Portanto, um algoritmo exato encontrará a solução ótima para o problema em um tempo exponencial, o qual pode facilmente ultrapassar o tempo desejado para encontrar o resultado, característica do NP-Completo.

Outra forma de resolver este problema é com métodos heurísticos, ou seja, métodos capazes de encontrar uma boa solução, próxima a solução ótima. As heurísticas, como Busca Local ou Busca Tabu, buscam uma solução viável ao problema em tempo computacional aceitável, isto é, deixam de lado a solução ótima para obter uma boa solução em tempo hábil [Rich e Knight 1993] e [Hillier e Lieberman 2006].

A escolha por utilizar um método heurístico de Busca Local para resolver o PRVC, deu-se por ser facilmente adaptável para outras heurísticas presentes na literatura, assim como Busca Tabu, Busca Local Guiada e GRASP. Além disso, encontram soluções razoáveis em grandes ou infinitos espaço de dados e operam utilizando apenas um estado, fazendo com que o método ocupe pouca memória.

Os métodos heurísticos fazem parte da Inteligência Artificial (IA) e quando os pesquisadores começaram a utilizar a computação paralela em sistemas de IA perceberam que o paralelismo poderia trazer um ganho significativo no desempenho desses sistemas, mesmo utilizando a paralelização em um único processador [Rich e Knight 1993]. Segundo [Schulz 2013] os problemas de otimização combinatória necessitam paralelizar tarefas para beneficiar o seu desenvolvimento. Uma vantagem da utilização da paralelização segundo [Rich e Knight 1993] é a melhor modularidade, pois quando as regras operam mais ou menos independentes é fácil de acrescentá-las, eliminá-las ou modificá-las sem mudar a estrutura do programa inteiro.

A paralelização pode ser feita tanto em CPU (Unidade Central de Processamento) quanto em GPU (Unidade de Processamento Gráfico). A paralelização em aceleradores gráficos também é chamada de Unidade de Processamento Gráfico de Propósito Geral (GPGPU), quando utiliza-se a Unidade de Processamento Gráfico para ir além de renderizações gráficas, ou seja, realizar aceleração em aplicações que eram executadas somente pela Unidade Central de Processamento, como aceleração de aplicações científicas.

1.1. Objetivo

O objetivo geral deste trabalho é paralelizar em GPU um método heurístico capaz de encontrar uma boa solução para o Problema do Roteamento de Veículos Capacitados (PRVC).

1.2. Objetivos Específicos

A fim de alcançar o objetivo geral, propõem-se os seguintes objetivos específicos:

- Implementar uma heurística capaz de resolver o problema;
- Paralelizar em GPU a heurística implementada;
- Analisar o desempenho do algoritmo paralelo.

1.3. Organização do Texto

O texto está organizado da seguinte forma: na Seção 2 será definido o Problema de Roteamento de Veículos. Na Seção 3 será apresentada as Heurísticas e as Meta-heurísticas. Na Seção 4 o tema abordado será a Unidade de Processamento Gráfico de Propósito Geral (GPGPU) e OpenACC, uma API (Interface de Programação de Aplicações) para GPU. Já na Seção 5 estarão os trabalhos relacionados. A proposta será apresentada na Seção 6, juntamente com a metodologia. Já a Seção 7 apresentará a Busca Local Iterada, o algoritmo sequencial e o algoritmo paralelo em GPU. Os resultados obtidos serão apresentados na Seção 8 e, por fim, a conclusão e os trabalhos futuros na Seção 9.

2. Problema de Roteamento de Veículos

O Problema de Roteamento de Veículos tem como objetivo determinar o melhor conjunto de rotas a ser realizado por uma frota de veículos, a fim de servir um conjunto de clientes e também é um dos estudos e problemas mais importantes de Otimização Combinatória (OC) [Toth e Vigo 2002]. Está entre os problemas mais complexos da área de OC, pertencendo à classe dos problemas NP-Completos [Leiserson et al. 2002], pois possui grande número de restrições e variáveis. Assim, necessita de muita capacidade de processamento, tornando inviável obter uma solução ótima em tempo hábil para instâncias grandes do problema [Goldbarg e Luna 2005].

O PRV pode ser abordado de múltiplas formas, como pode ser visto em [Goldbarg e Luna 2005, p.375–377]. Para este trabalho foi escolhido o Problema Roteamento de Veículo Capacitado (PRVC), pois é a base para outros PRVs [Toth e Vigo 2002, p.5]. No PRVC todas as demandas são previamente conhecidas e há apenas um único depósito central. Os veículos da frota são todos iguais, e possuem apenas restrições de capacidade. A resolução tem como finalidade a minimização dos custos (ou seja, distância percorrida e/ou tempo de viagem) para atender todas demandas solicitadas [Toth e Vigo 2002, p.5]. A seguir será apresentada a definição para o PRVC.

2.1. Definição do Problema de Veículos Capacitados

Segundo [Toth e Vigo 2002, p.5–8] o Problema do Roteamento de Veículos Capacitados pode ser definido da seguinte forma:

Seja $G = (V, A)$ um grafo completo, onde $V = 0, ..., n$ é um conjunto de vértices (clientes, cidades ou demandas) $i = 1, ..., n$, com o vértice 0 correspondendo ao depósito e A é um conjunto de arcos (depósitos).

Tem-se um custo não negativo, c_{ij}, onde é associado a cada arco $(i, j) \in A$ e representa o custo da viagem do vértice i ao j. Se $c_{ij} = c_{ji} \; \forall \; (i, j) \in A$, considera-se um PRVC simétrico, sendo C uma matriz de custos simétrica. Caso contrário será um PRVC assimétrico, então $c_{ik} + c_{kj} \geq c_{ij} \; \forall \; (i, j, k) \in V$.

Cada cliente i ($i = 1, ..., n$) está associado a uma demanda conhecida e não negativa, d_i, para entrega, e o depósito tem uma demanda fictícia $d_0 = 0$. Para o conjunto $S \subseteq V$, tem-se $d(S) = \sum_{i \in S} d_i$, para definir o total da demanda do conjunto.

No depósito tem-se um conjunto K de veículos idênticos com capacidade C disponível. Assume-se que $d_i \leq C \; \forall \; i = 1, ..., n$. Cada veículo pode fazer somente uma rota e também assume-se que $K > K_{min}$, onde K_{min} é o mínimo de veículos necessários para atender todas demandas.

O PRVC consiste em encontrar um conjunto k de circuitos simples, cada um correspondendo a uma rota de um veículo com um custo mínimo, definido como a soma dos custos dos arcos pertencentes ao circuito, tal que:

- Cada circuito visita o depósito;
- Cada vértice cliente é visitado por exatamente um circuito;
- A soma das demandas dos vértices visitados não podem exceder a capacidade C do veículo.

A definição matemática para minimizar os custos do PRVC segundo [Goldbarg e Luna 2005, p.398–399] é:

$$\text{(PRVC) Minimizar } z = \sum_{i,j} (c_{ij} \sum_k x_{ijk})$$

Sujeito a:

$$\sum_k y_{ij} = 1 \qquad i = 2, ..., n \tag{1}$$

$$\sum_k y_{ij} = m \qquad i = 1 \tag{2}$$

$$\sum_i q_i y_{ik} \leq Q_k \qquad k = 1, ..., m \tag{3}$$

$$\sum_j x_{ijk} = \sum_j x_{jik} = y_{ik} \qquad i = 2, ..., n \qquad k = 1, ..., m \tag{4}$$

$$\sum_{i,j \in S} x_{ijk} \leq |S| - 1 \qquad \forall S \subseteq \{2, ..., n\} \qquad k = 1, ..., m \tag{5}$$

$$y_{ik} \in \{0, 1\} \qquad i = 1, ..., n \qquad k = 1, ..., m \tag{6}$$

$$x_{ijk} \in \{0, 1\} \qquad i, j = 1, ..., n \qquad k = 1, ..., m \tag{7}$$

Onde:

x_{ij} é uma variável binária que assume valor 1 quando o veículo k visita o cliente j imediatamente após o i, 0 em caso contrário.

y_{jk} é uma variável binária que assume valor 1 se o cliente i é visitado pelo veículo k, 0 caso contrário.

q_i é a demanda do cliente i.

Q_k é a capacidade do veículo k.

c_{ij} é o custo de percorrer o trecho que vai do cliente i ao j.

A restrição (1) assegura que um veículo não visite mais de uma vez um mesmo cliente. A restrição (2) garante que o depósito receba uma visita de todos os veículos. A restrição (3) obriga que a capacidade dos veículos não seja ultrapassada. A restrição (4) garante que os veículos não parem suas rotas em um cliente. A restrição (5) constitui-se das tradicionais restrições de eliminação de subtours[1] . A restrição (6) é uma variável (y_{jk}) binária que assume valor 1 se o cliente i é visitado pelo veículo k, 0 caso contrário. A restrição (7) é uma variável (x_{ij}) binária que assume valor 1 quando o veículo k visita o cliente j imediatamente após o i, 0 em caso contrário.

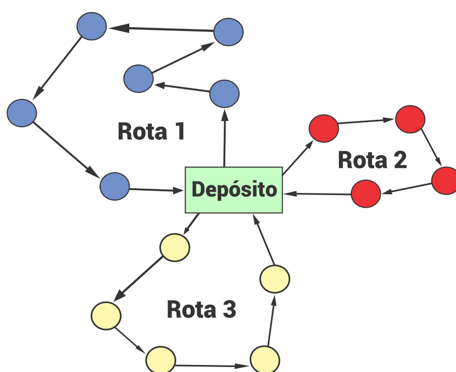

Figura 1. Solução do PRVC com 3 veículos.

Como pode-se ver na Figura 1, o PRVC possui apenas um depósito central, onde todos veículos devem iniciar e terminar a sua rota. Os veículos possuem uma determinada capacidade e não podem passar duas vezes no mesmo lugar (cidade ou cliente).

3. Heurísticas e Meta-heurísticas

Métodos heurísticos e meta-heurísticos são algoritmos de busca, que possuem a finalidade de encontrar soluções viáveis. Segundo [Rich e Knight 1993] e [Hillier e Lieberman 2006] a heurística é uma técnica que melhora a eficiência do processo de busca, que provavelmente vai encontrar uma excelente solução viável, mas não necessariamente uma solução ótima. Se for utilizada uma heurística bem-elaborada, pode-se esperar boas soluções (embora possivelmente não ótimas) para problemas difíceis, como o do caixeiro-viajante, em um tempo menor do que o exponencial, porém não pode-se dar garantia sobre a qualidade da solução que será obtida.

Alguns autores também tratam dos métodos meta-heurísticos, porém não há um consenso na literatura de quais são os métodos heurísticos e os meta-heurísticos. Segundo [Gendreau e Potvin 2010] as meta-heurísticas são métodos de solução que orquestram uma iteração entre os procedimentos de melhoria local e estratégias de níveis superior,

[1] Eliminação de subtours ou sub-rotas consiste na junção das sub-rotas para formação de uma única rota.

para criar um processo capaz de escapar dos ótimos locais e realizar uma busca robusta de um espaço de solução. Já para [Osman e Laporte 1996] meta-heurística é definida como um processo de gerações iterativas que orienta uma heurística subordinada, combinando inteligentemente conceitos diferentes para explorar o espaço de busca. Estratégias de aprendizagem são usadas para estruturar as informações, a fim de encontrar soluções de forma eficiente perto do ideal.

Neste trabalho serão abordados todos os métodos como sendo heurísticas. Algumas heurísticas existentes são Algoritmo Genético, Busca A*, Busca Bidirecional, Busca Profundidade, Colônia de Formigas, Subida de Colina e Vizinho Mais Próximo. Para este trabalho a heurística escolhida para resolver o PRVC foi a Busca Local Iterada (BLI) que será descrita na Seção 7. Isto, porque a BLI pode ser facilmente adaptada para outras heurísticas presentes na literatura como, por exemplo, a Busca Tabu, Busca Local Guiada e GRASP.

4. Unidade de Processamento Gráfico de Propósito Geral (GPGPU)

A programação paralela é uma forma existente na computação para realizar vários cálculos, tarefas ou métodos simultaneamente para reduzir o tempo de execução de problemas que manipulam grande quantidade de dados e/ou necessitam para sua resolução de uma grande capacidade de processamento. Assim, é utilizada para obter um ganho de performance nessas aplicações [Pasin e Kreutz 2003].

Essa paralelização pode ser realizada tanto em CPU quanto em GPU, porém quando utiliza-se aceleradores gráficos é denominada Unidade de Processamento Gráfico de Propósito Geral (GPGPU) ou Acelerador GPU, que é o uso da unidade de processamento gráfico, juntamente com uma CPU para acelerar aplicações científicas, de análise, de engenharia, de consumo e empresariais [Nvidia (b) 2015].

Uma maneira simples de compreender a diferença entre CPU e GPU é comparar como elas processam tarefas. Como pode ser visto na Figura 2, a CPU consiste em alguns núcleos otimizados para o processamento de uma série sequencial, enquanto a GPU tem uma arquitetura massivamente paralela que consiste em milhares de núcleos menores e mais eficientes, projetados para lidar com várias tarefas ao mesmo tempo [Nvidia (b) 2015].

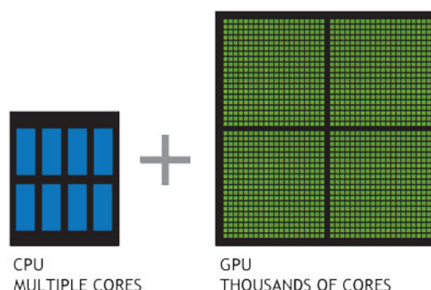

Figura 2. Diferença de núcleos entre CPU e GPU. [Nvidia (b) 2015]

Para utilizar a programação paralela em GPUs existem algumas APIs (Interface de Programação de Aplicações) como OpenCL, Cuda e OpenACC. OpenCL (*Open Computing Language*) é um padrão de programação paralela aberto e mantido pelo *Khronos*

Group. Este padrão proporciona a paralelização através de CPUs, GPUs e outros processadores, para mais informações [OpenCL 2015]. CUDA (*Compute Unified Device Architecture*) é uma plataforma de programação paralela para GPUs da Nvidia. O ambiente de desenvolvimento do CUDA permite o desenvolvimento de programas em C, C++, Fortran, Python, Java e também tem suporte a DirectCompute e OpenACC, para mais informações [Nvidia (a) 2015]. A seguir é apresentada a API OpenACC, que será a utilizada na programação paralela em GPUs deste trabalho.

4.1. OpenACC

OpenACC surgiu em 2011 como um modelo de programação de mais alto nível para paralelizar aplicações em aceleradores gráficos [OpenACC (b) 2015]. Fornece portabilidade entre sistemas operacionais, processadores e uma ampla gama de aceleradores gráficos, incluindo APUs (Unidade de Processamento Acelerado), GPUs e co-processadores com múltiplos núcleos. Assim, possui um modelo de aceleração de programação e suporte para diversos tipos de plataformas, como Nvidia, AMD e Intel [OpenACC (a) 2015].

O OpenACC trabalha com diretivas de compilador, para especificar laços de repetições, rotinas de biblioteca, variáveis de ambiente e regiões do código em C padrão, C++ e Fortran a ser descarregado a partir de uma CPU *host* para um acelerador *device*. A declaração das diretivas em C e C++ são especificadas por *#pragma acc nome-diretiva*, podendo haver outras cláusulas depois do nome da diretiva. Uma diretiva aplica-se a declaração de um bloco ou ciclo imediatamente seguinte a estrutura [OpenACC 2013].

Algumas diretivas existentes são: *#pragma acc parallel loop, #pragma acc kernels, #pragma acc data, #pragma acc atomic update e #pragma acc cache*. OpenACC também possui rotinas para integração com outras plataformas, como OpenCL, CUDA e Intel Coprocessor Offload Infrastructure (COI) [OpenACC 2013].

Existem diferentes compiladores para o OpenACC, alguns *open source* (gratuitos), como *GCC Compiler*, *accULL*, *Omni*, *OpenARC*, *OpenUH* e *RoseACC*, e outros compiladores comerciais (pagos), por exemplo, como o *PGI Accelerator Compiler*, *Cray Compilation Environment (CCE)*, *CAPS Compilers* e *PathScale Enzo Compiler*. O compilador utilizado é o *PGI Accelerator Compiler* disponibilizado pela *PGI Compiler e Tools* via `http://www.pgroup.com/`, pois é o compilador que possui maior suporte a API, tem o melhor desempenho na compilação e também é possível conseguir uma licença para universitário.

5. Trabalhos Relacionados

Nesta Seção são apresentados alguns trabalhos relacionados ao tema proposto. Estes trabalhos possuem o uso de heurísticas, meta-heurísticas, Problema de Roteamento de Veículos e/ou aceleradores gráficos e auxiliram no desenvolvimento desta pesquisa.

5.1. Uma Abordagem Heurística e Paralela em GPUs para o Problema do Caixeiro Viajante

A dissertação de mestrado de [Gazolla 2010] apresenta uma solução heurística paralela para o problema do caixeiro viajante em GPUs utilizando a API CUDA. Ainda, mostra como as unidades de processamento gráfico podem contribuir para acelerar a solução de problemas de otimização.

Para a resolução do problema foi utilizada a linguagem de programação C++ e implementados cinco algoritmos de Busca Local: 2-opt, swap, OrOpt-1, OrOpt-2 e OrOpt-3. Ainda, para cada um desses algoritmos são apresentados testes de desempenho em dois ambientes de desenvolvimento diferentes.

O autor conclui que os resultados obtidos tiveram ganhos significativos, mesmo para apenas uma GPU. Com os testes constatou-se que o modelo como um todo se mostrou eficiente e pode trazer soluções melhores, sendo várias vezes mais rápido que implementações em CPU.

5.2. Algoritmo Online para o Problema Dinâmico de Roteamento de Veículos

A tese de doutorado de [Oliveira 2011] utiliza um algoritmo *online* para resolver o Problema Dinâmico de Roteamento de Veículos (PDRV). Pode-se observar no texto que o autor também aborda as variações do PRV, como o PRVC e o Problema de Roteamento de Veículos com Janela de Tempo (PRVJT).

Foram desenvolvidos dois algoritmos híbridos com a linguagem de programação JAVA para resolver o PRVJT Estático, sendo que um deles é aplicável ao PDRV. O primeiro algoritmo é um algoritmo híbrido que combina *Simulated Annealing* Não-Monotônico (SANM) com a execução do *software* de programação matemática CPLEX que resolve o problema de particionamento de conjuntos (PPC) e também é aplicável ao PDRV. O segundo algoritmo também é híbrido, com uma combinação entre *Greedy Randomized Adaptive Search Procedure* (GRASP) e também o CPLEX.

O autor conclui que fica explícita a importância de um algoritmo de roteamento mais rápido. Aliado aos sistemas de roteamento que se preocupam em responder mais rapidamente às requisições dinâmicas, informações estocásticas podem ser utilizadas na tentativa de prever onde e quando novas aquisições poderão surgir.

5.3. Metaheuristicas para as Variantes do Problema de Roteamento de Veículos: Capacitado, com Janela de Tempo e com Tempo de Viagem Estocástico

Na dissertação de mestrado de [Miranda 2011], é apresentada a definição e um levantamento literário sobre as técnicas exatas e aproximativas de soluções do PRV Capacitado, PRV com Janela de Tempo e para o PRV Estocástico.

A resolução do PRVC Determinístico [Miranda 2011] apresenta 4 heurísticas implementadas no *Matlab*. A primeira desenvolvida foi uma Busca Local, onde é gerada uma solução inicial e a partir desta são feitas buscas locais e perturbações. Já a segunda é uma busca tabu, que é semelhante a Busca Local, com algumas mudanças para aumentar a capacidade do algoritmo de “fugir”de ótimos locais. A terceira heurística é basicamente a anterior com a inserção da memória adaptativa, que trabalha com uma lista de boas soluções. A quarta e última resolução do PRVC é a terceira solução apenas inserindo o Busca Local Guiada(GLS) como critério de solução perturbada, assim mudando o critério de escolha da solução candidata.

Para o PRV com Distância Máxima escolheu-se utilizar a quarta heurística do PRVC, apenas alterando ela para resolver problemas com restrição de distância máxima das rotas. O PRV com Janela de Tempo também foi resolvido com uma heurística baseada na última desenvolvida para o PRVC, modificando-a para a resolução do problema.

O autor conclui que os resultados mostram que a quarta heurística obteve o melhor desempenho quanto a qualidade da solução e teve custo computacional competitivo com as demais heurísticas. Por fim, [Miranda 2011] considerou que os resultados das heurísticas foram satisfatórios, mas que há espaço para melhorias.

5.4. Considerações Sobre os Trabalhos Relacionados

Na dissertação de [Gazolla 2010] há uma abordagem do Caixeiro Viajante com heurísticas de Busca Local paralelizadas em GPU, utilizando a linguagem C e a API CUDA. Já o trabalho de [Oliveira 2011] e o de [Miranda 2011] trabalham com variações do PRV, mas ambos abordam soluções sequenciais. Oliveira apresenta algoritmos híbridos com CPLEX e Miranda heurísticas de Busca Local em *Matlab*.

O presente trabalho aborda somente o Problema de Roteamento de Veículos Capacitados, que pode ser considerado semelhante à vários Caixeiros Viajantes com capacidade de carga para satisfazer uma demanda exigida e minimizar seus custos. Para obter uma melhor performance, além da CPU, é usada a GPU na paralelização de uma heurística de Busca Local Iterada, utilizando a linguagem C e a API OpenACC.

6. Metodologia

Este trabalhou iniciou com uma pesquisa bibliográfica sobre o Problema de Roteamento de Veículos, a fim de entendê-lo melhor e descobrir suas formas de resolução. Também foi realizada uma revisão bibliográfica sobre heurísticas e quais são capazes de resolver o PRV Capacitados (PRVC), bem como um estudo sobre computação paralela, focando seu uso em unidades de processamento gráfico. Deste modo, identificou-se algumas APIs existentes para programação em GPU.

A heurística implementada para o PRVC é a Busca Local Iterada (BLI), a qual é explanada na Seção 7. Optou-se pela BLI, pois ela ocupa pouca memória para encontrar um solução razoável e também pela sua modularidade que beneficia a implementação paralela. Há várias maneiras de abordar a programação paralela em GPU. A escolhida para o desenvolvimento deste trabalho foi a API OpenACC, pois favorece os desenvolvedores que possuem pouca ou nenhuma experiência em programação paralela e também é uma API que possui suporte e portabilidade de código para diferentes plataformas.

A linguagem escolhida para implementar a heurística em OpenACC foi a linguagem C, pois é uma das linguagens que possui o maior suporte. Foi utilizada a versão 2.0 do OpenACC e o compilador *PGI Accelerator C/C++ Workstation* que oferece um ambiente de desenvolvimento completo para desenvolvedores C e/ou C++, ambos disponibilizados pela PGI Compilers & Tools via `http://www.pgroup.com/`. O *hardware* utilizado foi uma CPU Intel Core i7-870 (8MB Cache e 2.93 GHz), uma GPU Nvidia GeForce GTS 240 *OEM Product* (1GB GDDR3) e 4GB de memória RAM com um Sistema Operacional Linux Ubuntu 14.04.2 LTS.

Para avaliar o resultado da paralelização foram comparados os tempos de execução sequencial e paralelo do algoritmo de BLI com diferentes tamanhos de instâncias do PRVC, definidas por Augerat e Uchoa et al [PUC-Rio 2015]. Também realizou-se uma comparação entre as soluções encontradas pela heurística proposta e as soluções disponíveis no repositório da [PUC-Rio 2015]. Nas próximas Seções estão os algoritmos sequencial e paralelo da BLI e os resultados obtidos.

7. Busca Local Iterada

Os algoritmos de Busca Local operam usando um único estado e em geral se movem apenas para vizinhos desses estados. Essa busca possui duas vantagens, a primeira é que usam pouca memória e a segunda é que frequentemente podem encontrar soluções razoáveis em grandes ou infinitos espaços de busca [Russell e Norving 2004].

A Busca Local Iterada (BLI) apresentada no Algoritmo 1 inicia a partir de uma solução inicial s gerada aleatoriamente ou com uma heurística gulosa. Assim que encontrada a solução inicial aplica-se a busca local em s, chegando-se então a uma solução s^*. Dada a solução s^*, aplica-se uma perturbação que gera um estado p intermediário pertencente ao conjunto solução. Logo após, a busca local é aplicada a p e assim chega-se a uma solução p^* em s^*. Se p^* é aceito no critério de aceitação torna-se o próximo elemento para a caminhada em s^*, caso contrário volta-se para s^* [Gendreau e Potvin 2010].

Algoritmo 1: Busca Local Iterada [Boussaïd et al. 2013, p.89]

s = GerarSoluçãoInicial();
s^* = BuscaLocal(s);
repita
    p = Perturbação(s^*);
    p^* = BuscaLocal(p);
    s^* = CritérioDeAceitação(s^*, p^*);
até *que o critério de parada seja satisfeito*;
retorna a melhor solução;

A BLI é composta por um conjunto de quatro métodos diferentes: Geração da Solução Inicial, Busca Local, Perturbação e Critério de Aceitação.

Geração da Solução Inicial – Nesta etapa é criada uma solução inicial para o problema, que pode ser obtida de diferentes formas: a partir de métodos aleatórios ou com heurísticas gulosas. A solução inicial gulosa tem duas principais vantagens em relação a soluções de partida aleatória: (i) combinadas com a busca local, soluções iniciais gulosas muitas vezes resultam em soluções de melhor qualidade; (ii) a busca local a partir de soluções gulosas leva em média, menos passos de melhoria e, portanto, a busca local requer menos tempo de execução. A qualidade da solução inicial é importante para alcançar a melhor solução possível [Gendreau e Potvin 2010].

Busca Local – Esta parte do algoritmo se preocupa em melhorar a solução encontrada. A cada iteração o método executa movimentos entre a vizinhança da solução atual, a fim de obter uma nova solução melhor [Gendreau e Potvin 2010].

Perturbação – A Busca Local Iterada escapa de ótimos locais aplicando perturbações no atual mínimo local. Se a perturbação é muito forte a BLI pode se comportar como uma reinicialização aleatória, por isso melhores soluções só serão encontradas com uma probabilidade muito baixa. Por outro lado, se a perturbação é muito fraca a BLI encontra muitas vezes um ótimo local já explorado [Gendreau e Potvin 2010].

Critério de Aceitação – O critério de aceitação define as condições que o novo ótimo local p^* tem que satisfazer para substituir o atual s^*. Existem diferentes maneiras de abordar este critério, dois exemplos são: (i) se p^* mais visitado que s^*; (ii) se p^* melhor que s^*. O critério de aceitação de Markovin é uma forma simples de tratar o critério de

aceitação, onde somente as melhores soluções são aceitas e é definido para problemas de minimização conforme o Algoritmo 2 [Gendreau e Potvin 2010].

Algoritmo 2: Critério de aceitação. [Gendreau e Potvin 2010]

se p^* *menor que* s^* **então**
  retorna p^*;
senão
  retorna s^*;
fim

7.1. Algoritmo Sequencial

O algoritmo começa com a leitura de um arquivo especificado pelo TSPLIB 95, que é uma biblioteca de casos de amostra para o Problema do Caixeiro-Viajante e problemas relacionados [Reinelt 1995]. O arquivo de entrada deve possuir o formato especificado pelo TSPLIB 95, ser do tipo CVRP (*"TYPE : CVRP"*), com distâncias euclidianas em duas dimensões (*"EDGE_WEIGTH_TYPE : EUC_2D"*) e conter apenas um depósito central. Esse arquivo contem o seu nome, um comentário, o tipo de problema, a dimensão ou quantidade de instâncias, o formato em que estão os dados, a capacidade do veículo, a cidade com seu x e y no plano, a cidade com sua respectiva demanda e por fim a cidade que é o depósito central.

Na leitura do arquivo obtém-se a quantidade de cidades, a demanda de cada veículo, a localização das cidades no plano (como na Figura 3) e as suas demandas. Para armazenar esses dados são criados três vetores, um vetor para guardar o x, um para o y e um para a demanda de cada cidade, e uma matriz, que irá conter as distâncias entre cidades obtidas pelo cálculo da distância bidimensional euclidiana de um ponto A até outro ponto B.

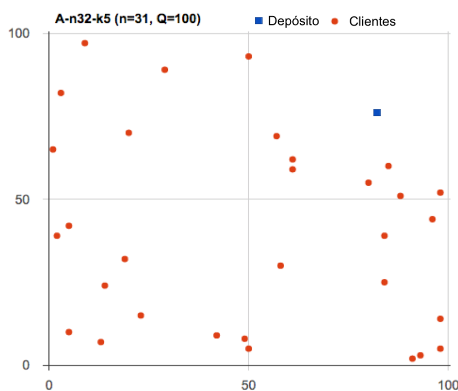

Figura 3. Plano com as instâncias. [PUC-Rio 2015]

A definição matemática para o cálculo da distância bidimensional euclidiana segundo [Bortolossi 2002, p.139] é:

$$d_{AB} = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

onde:

x_A é a posição x e y_A é a posição y do ponto A no plano.

x_B é a posição x e y_B é a posição y do ponto B no plano.

d_{AB} é a distância entre os dois pontos A e B.

O resultado do cálculo da distância entre duas instâncias (A e B) é armazenado em uma matriz e como o PRVC é simétrico não há necessidade de uma matriz densa. Por isso foi utilizado para armazenar as distâncias entre as cidades uma matriz triangular inferior alocada dinamicamente, como na Figura 4.

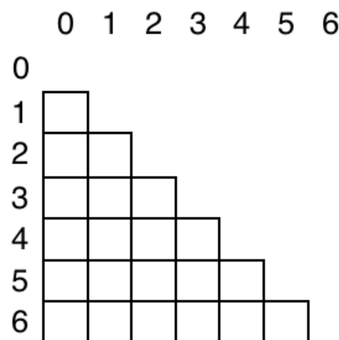

Figura 4. Matriz com as distâncias entre as instâncias.

Após gera-se um vetor para armazenar o resultado final do problema, esse vetor tem o tamanho para armazenar o mínimo de rotas possível. Para obter esse valor realiza-se o cálculo que é a demanda total do problema dividido pela capacidade de cada veículo ($demandaTotal/capacidade$) para saber o mínimo de rotas para resolver o problema onde, por exemplo, se o resultado encontrado for $3,2$ deve-se arredondar pra o próximo número inteiro, no caso, 4. Com o resultado obtido soma-se a quantidade de cidades do problema ($quantidadeCidades$) e assim tem-se o tamanho que deverá ser alocado o vetor que irá armazenar o resultado final.

A solução inicial para o PRVC é gerada a partir da heurística do vizinho mais próximo [Russell e Norving 2004]. Começando a rota sempre do depósito central e indo para a cidade mais próxima e que não ultrapasse a capacidade do veículo da rota, essa nova cidade será a referência para a escolha da próxima cidade a ser inserida na rota. Quando não é mais possível inserir uma cidade na rota por causa da sua demanda que ultrapassa o limite do veículo, é criada uma nova rota iniciando-se do depósito central, assim reiniciando o processo até todas cidades estarem incluídas em uma rota.

Após encontrar uma solução inicial aplica-se a Busca Local, executando uma heurística baseada em 2-opt [Gendreau e Potvin 2010], sendo retirada duas arestas e inseridas novamente de forma cruzada, isto é feito n vezes para cada rota, não alterando arestas em rotas distintas. A perturbação é feita por meio de trocas, trocando-se cada elemento para o próximo lugar encontrado em que reduz o custo total da rota. Nunca é trocada a primeira (0) e a última (n) posição do vetor, pois é no deposito central onde deve-se começar e terminar a solução. Pode-se verificar o processo na Figura 5, com a seta vermelha indicando a troca definitiva.

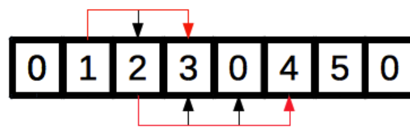

Figura 5. Perturbação por troca nas rotas econtradas.

Por fim aplica-se o critério de Markovin, que verifica se a nova solução encontrada é melhor que a solução anterior [Gendreau e Potvin 2010]. Se sim, a nova solução passa a ser a referência para a próxima iteração. A iteração da BLI irá acabar quando não houver melhoria na iteração anterior. Após o termino das iterações é exibida a solução final encontrada pela aplicação, juntamente com seu tempo de execução.

7.2. Algoritmo Paralelizado em GPGPU

Devido ao algoritmo desenvolvido possuir algumas dependências de dados não foi possível realizar a paralelização de 100% da aplicação. Foi utilizado o OpenACC nos casos que permitiam a paralelização. A diretiva básica utilizada para paralelização foi *kernels* e também utilizou-se a diretiva *loop* e a cláusula *reduction(operação:variável)*.

A diretiva *kernels* identifica uma região de código que pode conter paralelismo e confia na capacidade de paralelização automática do compilador para analisar a região, identificar quais laços são seguros para paralelizar, e, em seguida, acelerar esses laços [OpenACC (b) 2015]. A diretiva *loop* aplica-se em um laço de repetição e pode descrever que tipo de paralelismo o laço deve executar, declarar variáveis, matrizes e operações de redução. A cláusula *reduction(operação:variável)* gera uma variável privada para cada iteração do *loop*, porém ao final da iteração todas as cópias privadas são reduzidas em um resultado final, que é retornado para a região. Por exemplo, pode ser necessário o máximo de todos os exemplares particulares da variável ou, talvez, a soma. A redução só pode ser especificada em uma variável escalar e apenas comum, as operações especificadas podem ser executadas, como por exemplo, +, *, min, max [OpenACC (b) 2015].

Utilizou-se o OpenACC da seguinte forma, quando é identificada a quantidade de cidades na leitura do arquivo gera-se uma matriz para armazenar o cálculo das distâncias entre duas cidades. Nessa alocação foi utilizada a diretiva *#pragma acc kernels* antes do *for* para agilizar a alocação. No cálculo das distâncias euclidianas foi utilizada a diretiva *#pragma acc kernels* antes do primeiro *for* ou o *for* que representa o i da matriz, assim, foi paralelizado o cálculo e o armazenamento do resultado na matriz.

A Solução Inicial possui dependência de dados, pois necessita da comparação com dados da iteração anterior, assim, não sendo possível a paralelização, porém, conseguiu-se utilizar a diretiva de paralelização *#pragma acc kernels* antes do *for* que adiciona as cidades em um vetor auxiliar para encontrar a solução. A Perturbação e a Busca Local Iterada possuem o mesmo problema da solução inicial, não sendo possível a paralelização.

Por fim foi utilizada a diretiva *#pragma acc kernel loop reduction(+:CustoTotal)* antes do *for* que realiza o cálculo total do custo do resultado para o problema. A cláusula *reduction* foi utilizada para poder realizar a soma dos resultados encontrados em cada paralelização sem perder dados. Na Figura 6 disponibilizada no Anexo A pode-se observar as informações obtidas na hora da compilação das regiões que foram aceleradas em

GPU.

7.3. Considerações sobre Busca Local Iterada

O algoritmo desenvolvido para o PRVC pode ser encontrado no seguinte endereço `https://github.com/PericlesFeltrin/CVRP`, devidamente comentado e com o arquivo leia-me (*readme*) com as devida instruções para compilar e executar a aplicação.

8. Resultados

Para a realização dos testes foram utilizados seis arquivos de entrada, com diferentes quantidades de cidades (32, 44, 53, 63 e 80 instâncias de Augerat, 1995 e 1001 instâncias de Uchoa et al., 2014), disponibilizado pelo repositório mantido pela [PUC-Rio 2015]. Cada uma das instâncias foram executadas cinco vezes em paralelo e cinco vezes sequencialmente, limpando a memória cache do computador antes de cada execução.

Obteve-se tempos distintos para cada uma das execuções, assim fez-se o desvio padrão e a média de tempo das cinco execuções, a fim de comparar o tempo de execução sequencial com o tempo paralelo. A partir dos tempos de execução, calculou-se a diferença entre eles para cada instância, resultando no quão mais rápido foi encontrada a solução pelo algoritmo paralelo. Os resultados obtidos podem ser observados na Tabela 1, na Tabela 3 disponibilizada no Anexo B, no gráfico da Figura 7 disponibilizada no Anexo C e no gráfico da Figura 8 disponibilizada no Anexo D.

Instâncias	Tempo de Execução Sequencial	Tempo de Execução Paralelo	Diferença
32	5933,6μs	3863,8μs	34,88%
44	8353,6μs	4092,8μs	51,00%
53	10922,2μs	4837,4μs	55,71%
63	13738,8μs	6015μs	56,21%
80	31088μs	14150,4μs	54,48%
1001	63,1944s	28,7375s	54,52%

Tabela 1. Resultado dos testes de tempo médio de execução.

Com a realização dos testes de tempo de execução pode-se concluir que mesmo havendo limitações no algoritmo para implementar a aceleração houve um grande ganho de desempenho, visto que a média da diferença entre os tempos de execução ficou em 51,13%. Significando que as soluções contento as diretivas OpenACC foram 51,13% mais rápidas que as soluções executadas sequencialmente.

Já na Tabela 2 temos um comparativo entre o custo final das soluções encontradas pela heurística proposta e as soluções existentes no repositório da [PUC-Rio 2015]. Ainda possui um comparativo de quão inferior ficou a solução encontrada pela heurística proposta. Um exemplo de solução encontrada tanto pela heurística proposta e pelo repositório da [PUC-Rio 2015] pode ser visto na Figura 9 disponibilizada no Anexo E.

Instâncias	Repositório PUC-RIO	Heurística Proposta	Diferença
32	784	945	20%
44	937	1182	26%
53	1010	1307	29%
63	1616	1916	18%
80	1763	2093	18%
1001	72477	81280	13%

Tabela 2. Comparativo do custo final das soluções encontradas pelo repositório da [PUC-Rio 2015] e da Heurística proposta.

A heurística proposta consegue obter bons resultados visto que o custo final das soluções obtidas fica em média 20% inferior se comparado às soluções do repositório da [PUC-Rio 2015]. Um número que pode ser considerado baixo, dado que o algoritmo ainda precisa de melhorias, já que em algumas situações a solução final possui uma rota que tem somente uma cidade como destino, como é o caso da rota 5 que pode ser observada na Figura 9 disponibilizado no Anexo E.

9. Conclusão e Trabalhos Futuros

O presente estudo realizado sobre o Problema de Roteamento de Veículos mostra que, independente da sua classificação (com Janela de Tempo, Capacitados, entre outros), demanda muito processamento. Um método heurístico pode obter uma boa solução ao PRVC. Para tentar obter um melhor desempenho na aplicação pode ser utilizada a paralelização de tarefas ou de dados na implementação da Busca Local Iterada. O paralelismo pode ser empregado em CPU ou em aceleradores gráficos, que possuem mais núcleos que a CPU, frequentemente utilizada para paralelização.

Um ponto negativo do algoritmo desenvolvido é a dependência de dados, fazendo-se impossível a paralelização de 100% da heurística. O ponto positivo foi que mesmo havendo este problema as paralelizações realizadas obteram um ganho de desempenho, assim, alcançando um dos objetivos do trabalho.

Com uma paralelização de mais alto nível, utilizando diretivas, verificou-se que o OpenACC é um bom início para desenvolvedores que não possuem conhecimento em paralelização de aplicações em GPU. A diretiva *kernels* é um exemplo de como a API pode ajudar o programador. Com os testes de tempo de execução sequencial e paralelo, conclui-se que a paralelização em aceleradores gráficos pode trazer um ganho significativo no desempenho da aplicação, porém, para realizá-la os métodos não podem ser implementados com dependência de dados ou possuir o mínimo possível de dependência.

Para os trabalhos futuros pretende-se melhorar o algoritmo para que não haja rotas com apenas uma cidade como destino, reduzir a dependência de dados para implementar mais paralelizações, realizar testes em diferentes placas gráficas (com diferentes capacidades e fabricantes) e estudar o paralelismo de dados em OpenACC para reduzir o tempo de leitura dos dados de entrada.

Referências

Bortolossi, H. (2002). *Cálculo diferencial a várias variáveis*. Coleção Matmídia. Editora PUC-Rio.

Boussaïd, I., Lepagnot, J., e Siarry, P. (2013). A Survey on Optimization Metaheuristics. *Information Sciences*, Vol. 237:p. 82 – 117.

Gazolla, J. G. F. M. (2010). Uma Abordagem Heurística e Paralela em GPUs para o Problema do Caixeiro Viajante. Dissertação de Mestrado, Universidade Federal Fluminense – UFF.

Gendreau, M. e Potvin, J. (2010). *Handbook of Metaheuristics*. International Series in Operations Research & Management Science. Editora Springer US.

Reinelt, G. (1995). TSPLIB95. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`. Acesso em Agosto de 2015.

Goldbarg, M. e Luna, H. (2005). *Otimização Combinatória E Programação Linear - 2ªEdição*. Editora Elsevier.

Hillier, F. S. e Lieberman, G. J. (2006). *Introdução à Pesquisa Operacional*. Editora McGraw Hill Brasil.

Leiserson, C., Cormen, T., RIVEST, R., e STEIN, C. (2002). *Algoritmos: Teoria e Prática*. Editora Elsevier.

Miranda, D. M. (2011). Metaheurísticas para as Variantes do Problema de Roteamento de Veículos: Capacitado, com Janela de Tempo e com Tempo de Viagem Estocástico. Dissertação de Mestrado, Universidade Federal de Minas Gerais – UFMG.

Nvidia (a) (2015). CUDA Toolkit Documentation – Version 7.0. `http://docs.nvidia.com/cuda/`. Acesso em Maio de 2015.

Nvidia (b) (2015). What is GPU Accelerated Computing? `http://www.nvidia.com/object/what-is-gpu-computing.html`. Acesso em Maio de 2015.

Oliveira, H. C. B. d. (2011). *Algoritmo Online para o Problema Dinâmico de Roteamento de Veículos*. Tese de Doutorado, Universidade Federal de Minas Gerais – UFMG.

OpenACC (2013). The OpenACC Application Programming Interface – Version 2.0. `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf`. Acesso em Maio de 2015.

OpenACC (a) (2015). About OpenACC. `http://www.openacc.org/about-openacc`. Acesso em Maio de 2015.

OpenACC (b) (2015). OpenACC Programming and Best Pratices Guide. `http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf`. Acesso em Setembro de 2015.

OpenCL (2015). The OpenCL Specification – Version 2.1. `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`. Acesso em Maio de 2015.

Osman, I. e Laporte, G. (1996). Metaheuristics: A bibliography. *Annals of Operations Research*, Vol. 63(N. 5):p. 511 – 623.

Pasin, M. e Kreutz, D. L. (2003). Arquitetura e Administração de Aglomerados. *Anais do 3ª Escola Regional de Alto Desempenho*, Santa Maria – RS, Brasil.

PUC-Rio (2015). Capacitated Vehicle Routing Problem Library. `http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`. Acesso em Agosto de 2015.

Rich, E. e Knight, K. (1993). *Inteligencia artificial*. Editora Makron Books.

Russell, S. e Norving, P. (2004). *Inteligência Artificial – Tradução da Segunda Edição*. Editora Elsevier.

Schulz, C. (2013). Efficient Local Search on the GPU – Investigations on the Vehicle Routing Problem. *Journal of Parallel and Distributed Computing*, Vol. 73(N. 1):p. 14 – 31.

Tanenbaum, A. (2001). *Organização Estruturada de Computadores*. LTC Editora.

Toth, P. e Vigo, D. (2002). *The Vehicle Routing Problem*. Monographs on Discrete Mathematics and Applications. Editora Society for Industrial and Applied Mathematics.

Anexo

A.

```
pericles@pericles-Studio-XPS-8100:~/tfg/new/CVRP$ pgcc -acc -Minfo=accel CVRP.c -o paralelo
calcDistancia:
     10, include "calcDistancia.c"
           6, Generating copyin(cidadeY[1:quantCidades-1],cidadeX[1:quantCidades-1])
           7, Loop is parallelizable
           8, Accelerator restriction: size of the GPU copy of distancia,cidadeX,cidadeY is unknown
              Loop is parallelizable
              Accelerator kernel generated
              Generating Tesla code
               7, #pragma acc loop gang /* blockIdx.y */
               8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
              Generating copyin(cidadeY[:],cidadeX[:])
              Generating copyout(distancia[i1+1][:])
calcCusto:
     11, include "calcCusto.c"
           8, Generating copyin(rotas[:quantRotas])
           9, Accelerator restriction: size of the GPU copy of distancia is unknown
              Loop is parallelizable
              Accelerator kernel generated
              Generating Tesla code
               9, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
                  Sum reduction generated for custoTotal
              Generating copyin(distancia[:][:])
solucaoInicial:
     12, include "solucaoInicial.c"
          18, Generating copyout(cidades[:quantCidade])
          19, Loop is parallelizable
              Accelerator kernel generated
              Generating Tesla code
              19, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
main:
     92, Generating copyout(distancia[:quantCidades])
     93, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         93, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pericles@pericles-Studio-XPS-8100:~/tfg/new/CVRP$
```

Figura 6. Informações sobre o resultado da paralelização ao compilar o código no *PGI Accelerator Compiler*.

B.

Instâncias	Desvio Padrão Sequencial	Desvio Padrão Paralelo
32	40,3336μs	525,2924μs
44	2521,4026μs	852,1588μs
53	1266,0994μs	930,5744μs
63	3015,7481μs	706,7340μs
80	2655,1918μs	1654,4220μs
1001	0,6474s	0,3704s

Tabela 3. Desvio padrão do tempo de execução sequencial e paralelo

C.

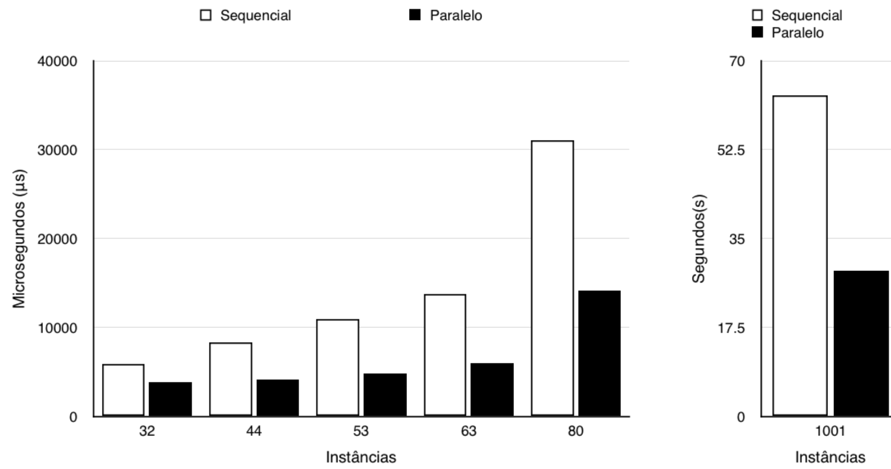

Figura 7. Gráfico com a comparação do tempo médio de execução sequencial e paralelo.

D.

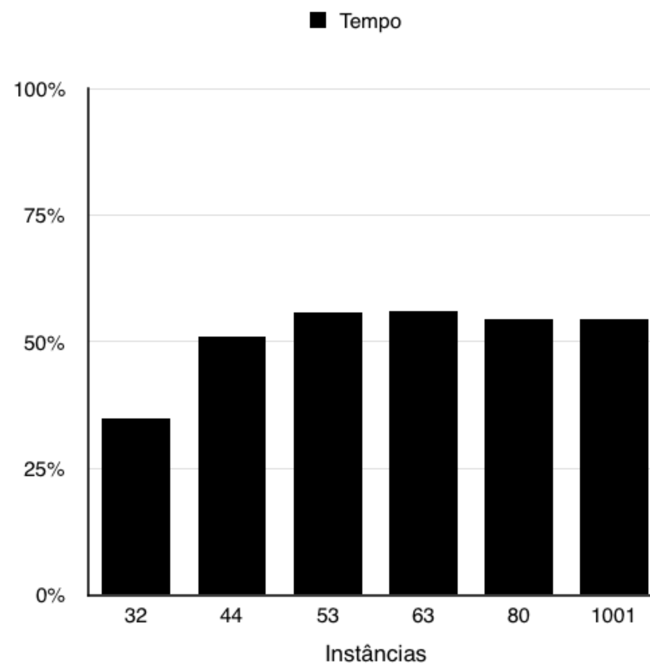

Figura 8. Gráfico com a diferença do tempo médio de execução da aplicação sequencial e paralela.

E.

Solução repositório PUC-RIO	Solução Heurística proposta
Rota #1: 21 31 19 17 13 7 26	Rota #1: 30 26 16 12 1 7 8 18 14
Rota #2: 12 1 16 30	Rota #2: 29 22 9 15 10 25 5 20
Rota #3: 27 24	Rota #3: 13 21 31 19 17 3 6
Rota #4: 29 18 8 9 22 15 10 25 5 20	Rota #4: 2 23 28 4 11 27
Rota #5: 14 28 11 4 23 3 2 6	Rota #5: 24
Custo 784	Custo 945

Figura 9. Comparação entre as soluções encontradas pelo repositório da [PUC-Rio 2015] e a heurística proposta para 32 instâncias.