



CENTRO FEDERAL DE EDUCAÇÃO
TECNOLÓGICA DE MINAS GERAIS

Diretoria de Pesquisa e Pós-Graduação

Programa de Pós-Graduação em Modelagem
Matemática e Computacional

ESTRATÉGIAS DE SOLUÇÃO BASEADAS EM PARALELISMO CUDA PARA O PROBLEMA DE INSTALAÇÃO DE FIBRAS EM REDES ÓTICAS

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Modelagem Matemática e Computacional do CEFET-MG, como parte dos requisitos necessários para a conclusão do Doutorado em Modelagem Matemática e Computacional.

Aluno: Daniel Moraes dos Reis

Orientador: Prof. Dr. Sérgio Ricardo de Souza (CEFET-MG)

Coorientadora: Prof. Dra. Anolan Yamilé Milanés Barrientos
(Molde University College, Molde, Norway)

Belo Horizonte - MG
Agosto de 2023

R375e Reis, Daniel Morais dos
Estratégias de solução baseadas em paralelismo CUDA para o problema de instalação de fibras em redes óticas / Daniel Morais dos Reis – 2022.
162 f.

Tese de doutorado apresentada ao Programa de Pós-Graduação em Modelagem Matemática e Computacional.
Orientador: Sérgio Ricardo de Souza.
Coorientadora: Anolan Yamilé Milanés Barrientos
Tese (doutorado) – Centro Federal de Educação Tecnológica de Minas Gerais.

1. Fibras óticas – Modelos matemáticos – Teses. 2. Caminhadas aleatórias (Matemática) – Teses. 3. Algoritmo genético – Teses. 4. Metaheurísticas – Teses. 5. Otimização – Teses. I. Souza, Sérgio Ricardo. II. Barrientos, Anolan Yamilé Milanés. III. Centro Federal de Educação Tecnológica de Minas Gerais. IV. Título.

CDD 519.2



SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
COORDENAÇÃO DO PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM MATEMÁTICA E COMPUTACIONAL

**“ESTRATÉGIAS DE SOLUÇÃO BASEADAS EM PARALELISMO CUDA
PARA O PROBLEMA DE INSTALAÇÃO DE FIBRAS EM REDES
ÓTICAS”.**

Tese de Doutorado apresentada por **Daniel Moraes dos Reis**, em 18 de agosto de 2023, ao Programa de Pós-Graduação em Modelagem Matemática e Computacional do CEFET-MG, e aprovada pela banca examinadora constituída pelos professores:

Prof. Dr. Sérgio Ricardo de Souza (Orientador)
Centro Federal de Educação Tecnológica de Minas Gerais

Profª. Drª. Anolan Yamilé Milanés Barrientos (Coorientadora)
Centro Federal de Educação Tecnológica de Minas Gerais

Prof. Dr. Igor Machado Coelho
Universidade Federal Fluminense

Prof. Dr. Gustavo Alves Fernandes
Centro Universitário de Belo Horizonte – UNI-BH

Prof. Dr. Gustavo Campos Menezes
Centro Federal de Educação Tecnológica de Minas Gerais

Prof. Dr. Marcone Jamilson Freitas Souza
Centro Federal de Educação Tecnológica de Minas Gerais

Visto e permitida a impressão,

Prof. Dr. Thiago de Souza Rodrigues
Coordenador do Programa de Pós-Graduação Stricto Sensu em
Modelagem Matemática e Computacional

Agradecimentos

Este é um momento especial para expressar minha gratidão às pessoas que desempenharam um papel fundamental em minha jornada acadêmica e na realização desta tese de doutorado. Gostaria de dedicar este espaço para expressar meu sincero reconhecimento a todos que contribuíram de maneira significativa para o sucesso deste trabalho.

Primeiramente, gostaria de agradecer aos meus filhos, Raíssa Ávila Moraes e Murilo Ávila Moraes. Vocês são minha fonte de inspiração e motivação. Agradeço por compreenderem os momentos em que precisei me dedicar intensamente aos estudos, por seu amor incondicional e por trazerem alegria aos meus dias. Vocês são meus maiores tesouros e a razão pela qual busco alcançar meus objetivos.

Agradeço também à minha mãe, Lúcia Aparecida de Moraes dos Reis, e ao meu pai, Fernando Maurício dos Reis, pelo seu amor, apoio e incentivo incondicionais ao longo de toda a minha jornada acadêmica. Sua presença constante e palavras de encorajamento foram fundamentais para que eu persistisse nos momentos mais desafiadores. Sou grato(a) por ter vocês como meus maiores apoiadores e por serem exemplos de força, coragem e determinação.

Lembro com carinho do meu irmão, André Moraes dos Reis, que infelizmente nos deixou antes que eu pudesse compartilhar esta conquista com ele. Sua memória será eternamente lembrada e agradeço por todos os momentos especiais que compartilhamos juntos. Você sempre será uma inspiração para mim.

A minha avó, Luíza Maria de Moraes, e ao meu avô, Sebastião Pires Batista de Moraes, que também nos deixaram, expresso minha gratidão pelos valores que transmitiram e pelo amor que sempre dedicaram à nossa família. Agradeço por todo o apoio e carinho que recebi de vocês.

Agradeço também minha tia, Leila Maria de Moraes de Paula, por estar sempre presente em minha vida, oferecendo seu apoio e amor incondicionais. Sua presença e palavras de incentivo foram muito importantes para mim.

A minha prima, Rafaela de Moraes Paula, expresso minha gratidão pela sua amizade e apoio constante. Agradeço por compartilharmos momentos especiais ao longo desta jornada acadêmica.

Não posso deixar de mencionar meus amigos (irmãos por afinidade) Henrique Leonardo Milagre de Faria e Bruno Magno Pfister Alves de Oliveira. Agradeço por sua amizade, suporte e momentos de descontração ao longo dessa jornada. Vocês tornaram essa caminhada acadêmica mais leve e prazerosa.

Agradeço ao meu orientador, Sérgio Ricardo de Souza, por sua orientação, conhecimento e disponibilidade em compartilhar seu tempo e experiência. Sua dedicação e apoio foram fundamentais para o desenvolvimento deste trabalho.

A minha coorientadora, Anolan Milanés Barrientos, expresso minha gratidão por sua

orientação e contribuições valiosas. Sua expertise e disponibilidade para discussões enriqueceram significativamente este trabalho.

Também agradeço ao professor Thiago Ferreira de Noronha por seu incentivo e apoio na condução deste trabalho. Sua orientação e contribuições foram fundamentais para o desenvolvimento desta pesquisa.

A todos os mencionados e também àqueles que, porventura, não foram citados individualmente, meu sincero agradecimento. Cada um de vocês desempenhou um papel importante em minha vida e na realização desta tese de doutorado. Sou imensamente grato(a) por todo o apoio, amor, incentivo e contribuições ao longo desta jornada acadêmica.

Este trabalho é dedicado a vocês. Obrigado(a) por serem parte fundamental de minha trajetória e por tornarem possível a realização deste sonho.

Resumo

Este trabalho otimiza métodos sequenciais e propõe métodos heurísticos paralelizados em GPUs Nvidia CUDA para a solucionar o Problema de Instalação de Fibras em Redes Óticas - PIFRO. O PIFRO consiste em alocar, em uma rede ótica que utiliza tecnologia *Wavelength Division Multiplexing* (WDM), um conjunto de requisições previamente conhecidas com múltiplas origens e destinos. O objetivo é minimizar o custo total dos dispositivos necessários para a operação da rede. Após provar que a versão de decisão do PIFRO é NP-Difícil, é realizada uma formulação matemática e implementadas duas metaheurísticas paralelizadas em formatos inéditos, buscando a melhoria dos resultados da literatura. Uma das heurísticas é baseada em *Biased Random Key Genetic Algorithm* (BRKGA) e a outra é baseada em *Iterated Local Search* (ILS) as quais foram selecionadas pelos bons resultados existentes para o problema. Os conceitos básicos de processamento paralelo são analisados e selecionados para execução de novas metaheurísticas em ambientes heterogêneos com GPUs.

PALAVRAS-CHAVE: Problema de Instalação de Fibras em Redes Óticas, Biased Random Key Genetic Algorithm, Iterated Local Search, Metaheurísticas, Otimização, Paralelismo, GPU.

Abstract

This work optimizes sequential methods and proposes parallelized heuristic methods on Nvidia CUDA GPUs to solve the Fiber Installation Problem in Optical Networks - PIFRO. PIFRO consists of allocating, in an optical network that uses *Wavelength Division Multiplexing* (WDM) technology, a set of previously known requests with multiple origins and destinations. The objective is to minimize the total cost of the devices required to operate the network. After proving that the decision version of PIFRO is NP-Hard, a mathematical formulation is carried out and two parallelized metaheuristics are implemented in new formats, seeking to improve the results in the literature. One of the heuristics is based on *Biased Random Key Genetic Algorithm* (BRKGA) and the other is based on *Iterated Local Search* (ILS) which were selected due to the good results available for the problem. The basic concepts of parallel processing are analyzed and selected for executing new metaheuristics in heterogeneous environments with GPUs.

keywords: PIFRO, Biased Random Key Genetic Algorithm, Iterated Local Search, Metaheuristic, Optimization, Parallelism, GPU.

Sumário

Lista de Tabelas	xi
Lista de Figuras	xiii
Lista de Algoritmos	xiv
1 Introdução	1
1.1 Apresentação Geral	1
1.2 Motivação e Justificativas	3
1.3 Objetivos	5
1.4 Metodologia de Pesquisa	5
1.5 Organização da tese	6
2 Caracterização do Problema de Instalação de Fibras em Redes Óticas	7
2.1 Problema de Instalação de Fibras em Redes Óticas	7
2.2 Equipamentos Óticos	8
2.2.1 ROADM (<i>Reconfigurable Optical Add/Drop Multiplexer</i>)	8
2.2.2 OA (<i>Optical Amplifier</i>)	9
2.2.3 OT (<i>Optical Transponder</i>)	9
2.3 Custo Total da Rede	9
2.4 Formulação matemática	10
2.5 Complexidade de PIFRO	11
3 Estratégias de Solução: Revisão Bibliográfica	14
3.1 Introdução	14
3.2 Heurísticas para PIFRO	15
3.2.1 Heurística <i>Greedy</i>	15
3.2.2 Heurística PSC	16
3.3 Heurística GA-PIFRO	18
3.4 Heurísticas ILS-PIFRO	20
3.4.1 Critérios de aceitação	22
4 Paralelismo, Placas de Processamento Gráfico e GPUs Nvidia CUDA	24
4.1 Introdução	24
4.2 <i>Hardware</i> e <i>Software</i> Paralelizados	25
4.2.1 Modificações da arquitetura de von Neumann	26
4.3 Classes de arquiteturas de paralelismo	27
4.3.1 <i>Ã</i> snica instrução, dados múltiplos	27
4.3.2 Múltiplas instruções, dados múltiplos	28
4.4 Avaliação do desempenho de algoritmos paralelos	29

4.4.1	Princípio de Brent e Speedup	30
4.4.2	Lei de Amdahl	31
4.5	Placas de Processamento Gráfico	31
4.6	GPUs Nvidia CUDA [©]	33
4.6.1	Distribuição de <i>threads</i> por <i>grid</i> , paralelismo dinâmico e <i>cooperative groups</i>	34
4.6.2	<i>Monolithic Kernels</i> vs <i>Grinde-Stride Loops</i>	35
4.6.3	<i>Race Conditions</i>	37
5	Metaheurísticas Paralelas	38
5.1	Topologias de Processamento Paralelo	38
5.2	Caracterização das Metaheurísticas Paralelas	40
5.2.1	Dimensão 1 - Cardinalidade do Controle de Busca	41
5.2.2	Dimensão 2 - Controle de Busca e Comunicação	41
5.2.3	Dimensão 3 - Diferenciação da Busca	42
5.3	Estratégias de Paralelização em Baixo Nível	43
5.4	Estratégias de Paralelização em Decomposição de Dados	44
5.5	Estratégias de Paralelização de Buscas Múltiplas Independentes	45
5.6	Estratégias de Paralelização de Buscas Múltiplas Cooperativas	46
5.6.1	Cooperação Síncrona	47
5.6.2	Cooperação Assíncrona	48
6	Paralelização das Estratégias de Solução Sequenciais	50
6.1	Introdução	50
6.2	Estruturas de dados	51
6.3	Algoritmo de Dijkstra paralelo para GPUs CUDA	53
6.4	Paralelização das heurísticas construtivas	58
6.5	Paralelização em baixo nível da busca local	59
7	Estratégias de Solução Paralelas	61
7.1	Introdução	61
7.2	Topologias de Processamento Paralelo Hierárquica	62
7.3	<i>Paths Pool</i> : Armazém de Caminhos entre Vértices	63
7.3.1	O subproblema <i>K-Shortest Paths</i>	64
7.4	O Algoritmo de Yen original e mapa de pontos de paralelização	66
7.4.1	Paralelização CUDA do Algoritmo de Yen	69
7.5	Paralelização CUDA do Algoritmo MPS	76
7.5.1	Algoritmo MPS original e mapa de pontos de paralelização	76
7.5.2	Paralelização CUDA da generalização do Algoritmo de Yen	79
7.5.3	Otimização da estrutura de dados	82
7.5.4	Algoritmo MPS Paralelo	86
7.6	Paralelização CUDA das Estratégias de Solução	89
7.6.1	Heurísticas Construtivas CUDA baseadas no <i>Paths Pool</i>	89
7.6.2	Busca Local Baseada em <i>Paths Pool</i>	90
7.7	Metaheurística Paralela CUDA GA-PIFRO	93
7.7.1	Implementações Propostas para CUDA-GA-PIFRO	94
7.8	Metaheurística <i>Iterated Local Search</i> Paralela	97

7.8.1	Implementações Propostas para CUDA ILS-PIFRO	99
7.9	Análise de dominância do espaço de buscas com soluções geradas via <i>Paths Pool</i>	101
8	Experimentos Computacionais	103
8.1	Configuração dos Parâmetros	103
8.2	Instâncias	104
8.2.1	Instâncias SNDLib	104
8.2.2	Instâncias SNAP	107
8.2.3	Instâncias DIMACS	108
8.2.4	Adaptações realizadas nos grupos de instâncias	108
8.3	Experimento I: Utilização do Algoritmo de Dijkstra nas estratégias de solução sequenciais	109
8.4	Experimento II: Apuração do desempenho do Algoritmo CUDA-Yen	110
8.5	Experimento III: Desempenho dos Algoritmos CUDA-Yen-Generalization e CUDA-MPS	113
8.6	Avaliação do desempenho das heurísticas paralelas para PIFRO	118
8.6.1	Experimento IV: Desempenho do Algoritmo CUDA- <i>Multistart</i> PIFRO	118
8.6.2	Experimento V: Desempenho do Algoritmo CUDA GA-PIFRO	119
8.6.3	Experimento VI: Desempenho do Algoritmo CUDA ILS-PIFRO	121
8.7	Avaliação do desempenho das melhores heurísticas paralelas para PIFRO .	124
8.7.1	Experimento VII: Melhor desempenho da heurística CUDA <i>Multistart</i> -PIFRO	124
8.7.2	Experimento VIII: Melhor desempenho da heurística CUDA GA-PIFRO	125
8.7.3	Experimento IX: Melhor desempenho da heurística CUDA ILS-PIFRO	126
8.7.4	Experimento X: Gráficos do desempenho das heurísticas paralelizadas CUDA PIFRO	126
8.8	Apuração da qualidade de solução das heurísticas paralelas para PIFRO . .	130
8.8.1	Experimento XI: Melhoria da qualidade de solução da heurística CUDA <i>Multistart</i> -PIFRO	130
8.8.2	Experimento XII: Melhoria da qualidade de solução da heurística CUDA GA-PIFRO	131
8.8.3	Experimento XIII: Melhoria da qualidade de solução da heurística CUDA ILS-PIFRO	133
8.9	Análise da qualidade de solução das melhores heurísticas paralelas para PIFRO	135
8.9.1	Experimento XIV: Detalhamento da versão de melhor qualidade de solução da heurística CUDA <i>Multistart</i> -PIFRO	136
8.9.2	Experimento XV: Detalhamento da versão de melhor qualidade de solução da heurística CUDA GA-PIFRO	136
8.9.3	Experimento XVI: Detalhamento da versão de melhor qualidade de solução da heurística CUDA ILS-PIFRO	137
8.9.4	Experimento XVII: Detalhamento gráfico das qualidades de soluções das heurística paralelizadas de PIFRO	138

9	Conclusões	140
9.1	Principais Contribuições	140
9.2	Trabalhos publicados	143
9.3	Recomendações para Pesquisas Futuras	143
	Referências	144

Lista de Tabelas

7.1	Notação para Figura 7.2.	68
8.1	Percentual de tempo utilizado para execução do Algoritmo de Dijkstra pelas estratégias de solução para PIFRO	109
8.2	Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 10$	111
8.3	Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 100$	111
8.4	Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 1000$	112
8.5	Avaliação do Algoritmo CUDA-Yen-Generalization: Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 10$	114
8.6	Avaliação do Algoritmo CUDA-MPS: Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 10$	114
8.7	Avaliação do Algoritmo CUDA-Yen-Generalization: Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 100$	115
8.8	Avaliação do Algoritmo CUDA-MPS: Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 100$	115
8.9	Avaliação do Algoritmo CUDA-Yen-Generalization: Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 1000$	116
8.10	Avaliação do Algoritmo CUDA-MPS: Tempo de execução para atingir <i>K-Shortest Paths</i> para $K = 1000$	116
8.11	Desempenho obtido entre as diferentes versões de CUDA- <i>Multistart</i> PIFRO	118
8.12	Desempenhos obtidos pela Heurística CUDA GA-PIFRO - <i>Greedy</i>	119
8.13	Desempenhos obtidos pela Heurística CUDA GA-PIFRO - PSC	120
8.14	Desempenhos obtidos pela Heurística CUDA GA-PIFRO - Hierárquica com <i>Paths Pool</i>	121
8.15	Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO- <i>Greedy</i>	122
8.16	Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO-PSC	123
8.17	Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO- <i>Paths Pool</i>	123
8.18	Detalhamento do desempenho da versão mais eficiente da heurística CUDA <i>Multistart</i> -PIFRO	124
8.19	Detalhamento da performance da versão mais eficiente da heurística CUDA GA-PIFRO	125
8.20	Detalhamento do desempenho da versão mais eficiente da heurística CUDA ILS-PIFRO	126
8.21	Desempenho obtido entre as diferentes versões da heurística CUDA- <i>Multistart</i> PIFRO	130
8.22	Desempenhos obtidos entre as diferentes versões da heurística CUDA GA-PIFRO- <i>Greedy</i>	131

8.23	Desempenhos obtidos entre as diferentes versões de CUDA GA-PIFRO-PSC	132
8.24	Desempenhos obtidos entre as diferentes versões da heurística CUDA GA-PIFRO- <i>Paths Pool</i>	132
8.25	Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO- <i>Greedy</i>	133
8.26	Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO-PSC	134
8.27	Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO- <i>Paths Pool</i>	135
8.28	Detalhamento da qualidade das soluções da versão mais eficiente da heurística CUDA <i>Multistart</i> -PIFRO	136
8.29	Detalhamento da qualidade das soluções da versão mais eficiente da heurística CUDA GA-PIFRO	137
8.30	Detalhamento do desempenho da versão mais eficiente da heurística CUDA ILS-PIFRO	138

Lista de Figuras

2.1	Representação gráfica da função $F_{WDM}^{ij}(\omega_{ij}, l_{ij})$.	10
2.2	Rede de PIFRO obtida com a transformação à partir de uma instância de SC $U = \{1, 2, 3, 4, 5\}$ e $S = \{S_1 = \{1, 2\}, S_2 = \{1, 2, 3\}, S_3 = \{2, 3, 4\}, S_4 = \{3, 4, 5\}, S_5 = \{4, 5\}\}$.	13
3.1	Representação gráfica da função $F'_{WDM}(\omega, l, p, P)$ para $p = 1$ e $P = 5$.	17
3.2	Transição entre gerações sucessivas (Noronha et al., 2011).	19
3.3	Funcionamento da heurística ILS-PIFRO.	21
4.1	Arquitetura de von Neumann.	25
4.2	Arquitetura <i>Single Instruction, Multiple Data</i> - SIMD.	28
4.3	Sistema Paralelo com Memória Compartilhada.	29
4.4	Sistema Paralelo com Memória Distribuída	29
4.5	Diferenças entre as estruturas da CPU e GPU.	32
4.6	Estrutura lógica de blocos e <i>threads</i> em GPUs.	32
5.1	Topologia de processamento paralelo “Completamente Conectada”.	39
5.2	Topologia de processamento paralelo “Estrela”.	39
5.3	Topologia de processamento paralelo “Anel de Processamento Unidirecional”.	40
5.4	Taxonomia para classificação de metaheurísticas paralelas (Crainic et al., 1997; Crainic e Hail, 2005)	41
6.1	Exemplo de grafo em <i>compact adjacency list</i> .	52
7.1	Topologia de processamento paralelo “Hierárquica”.	63
7.2	<i>Flowchart</i> para Algoritmo de Yen.	67
7.3	<i>Flowchart</i> do Algoritmo MPS.	77
7.4	<i>Segmented Sorting Process</i> usando <i>Parallel Fix Strategy</i> .	85
7.5	Funcionamento da heurística CUDA <i>Greedy</i> utilizando o <i>Paths Pool</i> .	89
7.6	Funcionamento da busca local baseada no Algoritmo de Dijkstra.	91
7.7	Ilustração do conjunto de soluções viáveis possíveis de acordo com os caminhos para cada $r \in R$	102
8.1	Representação da instância abilene–D-B-S-N-S-A-N-N.	107
8.2	Tela de pesquisa das instâncias para PIFRO.	107
8.3	Aceleração média obtida por cada configuração TMPPC vs 100%	112
8.4	Aceleração média obtida por cada configuração TMPPC.	117
8.5	<i>Gap</i> entre o tempo demandado pelas heurísticas para atingir o valor.	127

8.6	TTT-Plot das diferentes versões de CUDA Multistart PIFRO para as instâncias CollegeMSG e Johnson32-4.	128
8.7	TTT-Plot das diferentes versões de CUDA-GA PIFRO para as instâncias CollegeMSG e Johnson32-4.	128
8.8	TTT-Plot das diferentes versões de CUDA-ILS PIFRO para as instâncias CollegeMSG e Johnson32-4.	129
8.9	TTT-Plot das diferentes versões da heurísticas CUDA para PIFRO para as instâncias CollegeMSG e Johnson32-4.	129
8.10	<i>Gap</i> entre a qualidade das soluções apresentadas pelas heurísticas ao atingir o critério de parada.	139

Lista de Algoritmos

1	Greedy	16
2	Busca Local	16
3	PSC	18
4	GA-PIFRO	20
5	ILS-PIFRO	22
6	KernelMultiply	34
7	Exemplo de Dynamic Parallel Kernel Call	35
8	Kernel Multiply Cooperative	35
9	Monolithic Kernel Example	36
10	Grid-Stride Loop Kernel Example	36
11	Kernel 1: Dijkstra	54
12	DeviceFunction 1: THRESHOLD	55
13	DeviceFunction 2: RELAX	56
14	DeviceFunction 5: ExtractPreviousEdges	57
15	DeviceFunction 6: ExtractPath	58
16	DeviceFunction: <i>CUDA Parallel Greedy Shortest Path Based</i>	59
17	DeviceFunction: <i>CUDA Parallel Local Search Shortest Path Based</i>	60
18	Kernel Yen	70
19	DeviceFunction YenGenerateCandidates	71
20	DeviceFunction getThreadsSetsSize	72
21	DeviceFunction YenSubgraphSetup	74
22	DeviceFunction YenRestoreGraph	75
23	DeviceFunction: Yen Generalization	80
24	DeviceFunction: Generate Candidates	81
25	Kernel 7: getThreadsSetsSize	82
26	DeviceFunction: <i>MPS Forward Sorted Star Form Nested Sort Call</i>	84
27	DeviceFunction: <i>MPS Forward Sorted Star Form Parallel Fix Strategy</i>	85
28	DeviceFunction: MPS	87
29	DeviceFunction: <i>MPS Generate Candidates</i>	88
30	DeviceFunction: <i>Parallel Greedy Paths Pool Based</i>	90
31	DeviceFunction: <i>Parallel Local Search Paths Pool Based</i>	92
32	DeviceFunction: Initial_Population_Shortest_Path_Based	95
33	DeviceFunction: Initial_Population_Paths_Pool_Based	95
34	DeviceFunction: CUDA GA-FIP	96
35	DeviceFunction: CUDA ILS-PIFRO	100
36	DeviceFunction: CUDA ILS-PIFRO	100

Capítulo 1

Introdução

1.1 Apresentação Geral

A “Internet das Coisas” se refere a uma revolução tecnológica que tem, como objetivo, conectar os itens usados no cotidiano das pessoas e empresas à rede mundial de computadores. Cada vez mais surgem eletrodomésticos, meios de transporte e até mesmo tênis, roupas e maçanetas conectadas à Internet e a outros dispositivos, como computadores e *smartphones*. Estes dispositivos, atualmente, executam diversas aplicações simultâneas (como, por exemplo, vídeo conferência, transmissões de eventos com alta qualidade de áudio e vídeo, voz sobre IP, dentre outras aplicações multimídia), atendendo, de forma convergente, diversas necessidades de seus usuários.

Por este motivo, a Internet, cada vez mais, tem demandado melhorias, com tecnologias que permitem trocas de dados cada vez mais velozes, com altas taxas de transmissão, eficientes e confiáveis, requisitos estes que atualmente somente podem ser atendidos de forma satisfatória com o uso de redes de transmissão de dados via fibras óticas.

Além de sua alta velocidade de transmissão, as redes de fibras óticas contam com uma baixa interferência do meio onde estão instaladas, graças à transmissão de informações na forma de sinais luminosos, o que, por consequência, também faz a perda de dados ser muito pequena. Tais vantagens, no entanto, trazem consigo um fator nocivo, pois o custo de implantação e manutenção de infraestrutura de redes com essa tecnologia é muito alto.

Cada enlace de uma rede de fibra ótica pode conter várias fibras, cada uma com taxa de transmissão da ordem de terabits por segundo. Para um melhor aproveitamento da grande largura de banda disponível nestes enlaces, tecnologias como *Wavelength Division Multiplexing (WDM)* (Dutta et al., 2004) e suas variações (como *Dense WDM (DWDM)* (Dutta et al., 2004) e *Coarse WDM (CWDM)* (Dutta et al., 2004)) são empregadas para garantir a boa utilização da infraestrutura instalada (Dutta et al., 2004). A tecnologia WDM divide cada fibra ótica logicamente, fazendo com que o sinal luminoso seja particionado em diferentes canais de transmissão, cada um multiplexado em um comprimento de onda diferente.

A maior parte dessas redes, normalmente, compõe-se de um conjunto de comutadores óticos, interconectados por um conjunto de enlaces WDM, dando o nome à arquitetura de redes óticas mais utilizada, chamada de Arquitetura Baseada em Comutação de Circuitos Óticos (Dutta et al., 2004). Os comutadores são configurados para estabelecerem canais de comunicação, que percorrem vários enlaces, sem que o sinal ótico sofra conversão para o domínio elétrico nos comutadores intermediários. Estes canais são conhecidos como

caminhos óticos (*lightpaths*, em inglês).

Seja uma rede ótica qualquer modelada em um grafo não direcionado $G(V, E)$. Os caminhos óticos são definidos pelos vértices de origem e destino, e pela quantidade de comprimentos de onda que será alocada nos enlaces definidos para estabelecer a troca de dados entre os vértices requeridos. Os caminhos óticos se originam em vários pontos da rede ótica e podem ter, como destinos, equipamentos distribuídos geograficamente pela topologia física da rede. Múltiplos caminhos óticos podem utilizar o mesmo comprimento de onda, desde que não compartilhem de um mesmo enlace da rede. Esta restrição é conhecida como *Wavelength Crash Constraint* (Dutta et al., 2004).

A topologia física de uma rede ótica é formada por seus equipamentos e fibras óticas interconectadas. A topologia virtual é formada pelo conjunto de caminhos óticos nos quais trafegam dados na rede ótica, independentemente da distribuição física das fibras óticas. Portanto, o projeto de topologias virtuais consiste em determinar os caminhos óticos que devem ser estabelecidos para criar uma topologia virtual (Skorin-Kapov, 2006). Definida a topologia virtual, uma rota e um comprimento de onda devem ser atribuídos a cada caminho ótico.

O *Problema de Instalação de Fibras em Redes Óticas (PIFRO)*, proposto por um conjunto de pesquisadores da *Bell Labs* em Antonakopoulos e Zhang (2007), consiste em rotear, através da tecnologia *WDM*, um conjunto de requisições por conexões previamente conhecidas, que se originam em vários pontos da rede e que podem ter, como destinos, equipamentos distribuídos geograficamente. O objetivo das estratégias de solução existentes para o problema é minimizar o custo total dos dispositivos óticos necessários para a operação da rede. Para este problema, considera-se que, em uma única requisição, pode haver necessidade de alocação de vários comprimentos de onda que ligam uma origem a um destino. Vários caminhos óticos podem ser utilizados para conectar quaisquer dois pares de origem/destino na rede. Por outro lado, mesmo que a necessidade de conexão seja inferior à largura de banda disponível em um comprimento de onda, um comprimento de onda inteiro será utilizado.

As estratégias de solução para o PIFRO apresentadas em dos Reis et al. (2019), dos Reis et al. (2013) e dos Reis (2013), no entanto, sofrem de forte limitação no tocante à aplicação em instâncias de grande porte, ou seja, a estruturas de redes que, quando modeladas em grafos, resultam em grandes quantidades de vértices e arestas, que podem ser da ordem de milhões. Deve-se ressaltar que, nos trabalhos citados, as maiores instâncias avaliadas têm grafos com quantidade de vértices e arestas na ordem de apenas centenas e, ainda assim, sinalizam a impossibilidade de encontrar soluções viáveis mesmo em tempo consideravelmente alto.

Técnicas de processamento paralelo vêm sendo empregadas para a solução de instâncias de grandes dimensões em problemas diversos. As *Graphic Processing Units* (GPUs) são *hardwares* dotados de milhares de núcleos de processamento, que podem executar simultaneamente um ou vários conjuntos de instruções (Cook, 2013). Apesar da capacidade reduzida de cada núcleo de processamento isolado, quando comparado às *Central Processing Units* (CPUs) da atualidade, a execução paralela de tarefas, caso devidamente adaptadas para GPUs, pode acelerar superlinearmente a execução de algoritmos (*speedup*) (Amdahl, 1967), como visto em Djinevski et al. (2013), Otte e Correll (2013), Ichnowski e Alterovitz (2012), dentre outros trabalhos. Isso se justifica uma vez que, dependendo da operação do algoritmo que terá sua complexidade de tempo analisada, ela será completa ou parcialmente dividida por uma constante C , tal que C representa o total de unidades

de processamento paralelo disponíveis, o que, apesar de manter na maioria das vezes o grau de complexidade polinomial inalterado, na prática reduz consideravelmente o tempo de execução.

GPUs fazem seu processamento de instruções em sua coleção de múltiplos *Streaming Processors* (SM), os quais executam uma instrução em um *stream* de dados em cada uma de suas *threads*. Devido a isso, as GPUs têm se destacado na execução de algoritmos que possuem padrões recorrentes de acesso a dados e alta intensidade de execução aritmética (Luebke, 2008). É importante destacar as GPUs Nvidia, dotadas da tecnologia CUDA[®], que têm se popularizado, dada sua relação custo benefício e tendo algoritmos projetados de forma que considerem suas especificidades, são capazes de ampliar o desempenho no processamento de dados e/ou tarefas, quando comparadas a *hardwares* concorrentes e até teoricamente superiores (Nvidia, 2018).

1.2 Motivação e Justificativas

Esta Seção apresenta as motivações e justificativas desta tese, os quais se originam à partir da caracterização do problema apresentada no Capítulo 2, da revisão bibliográfica apresentada no Capítulo 3 e dos conceitos sobre processamento paralelo e GPUs apresentados nos Capítulos 4 e 5. A aplicação das propostas advindas das questões apresentadas nesta Seção resultam na paralelização CUDA das estratégias de solução sequenciais existentes, exibidas no Capítulo 6 e na proposta e implementação de estratégias de solução paralelas projetadas e, levando-se em consideração as especificidades, para plataformas CUDA apresentadas no Capítulo 7.

A visão geral apresentada nos Capítulos 4 e 5 busca distinguir as diferentes estratégias de paralelização de metaheurísticas para solução de problemas de otimização combinatória e identificar lacunas existentes na literatura, principalmente as relacionadas à utilização de paralelismo heterogêneo com GPUs. Para alcançar este objetivo, a investigação da literatura foi realizada e os trabalhos com inovações foram classificados através da taxonomia introduzida em Crainic et al. (1997), elaborando assim, um estudo comparativo.

Partindo da análise de tal estudo comparativo, observa-se que grande parte das metaheurísticas paralelas possuem propostas com características semelhantes. Isso confirma o que é dito por Sörensen (2015), Crainic et al. (2014), Crainic (2015) e Davidović e Crainic (2015), os quais afirmam que existem muitos estudos sobre heurísticas e metaheurísticas na literatura, de modo que elas são capazes de resolver praticamente quaisquer problemas de otimização existentes, sejam ou não com a qualidade de solução e tempo esperados. No entanto, ainda segundo os mesmos autores, existem diversas lacunas na literatura que devem ser analisadas de forma mais profunda pelos pesquisadores da atualidade. Dentre estas lacunas, estão:

- (i) Desconstrução dos métodos e testes destes em diferentes versões, sendo geradas de forma específica para um problema e/ou arquitetura e comparadas com as versões existentes, para ter, assim, uma justificativa para seu melhor funcionamento em determinadas circunstâncias;
- (ii) Conseguir responder com exatidão quando e porque em um determinado problema, uma metaheurística teve seu melhor desempenho, comparando suas diversas formas

de implementação, e não apenas compará-la se é ou não de desempenho superior a outra;

- (iii) Encontra-se em falta na literatura trabalhos que apliquem uma metaheurística com diferentes formas de paralelização para um problema em específico;
- (iv) Os recursos disponíveis em *hardwares* recentes não são totalmente explorados para estratégias de paralelização, especialmente quando se trata das GPUs, as quais têm evoluído constantemente;
- (v) As estratégias de paralelização que fazem uso de decomposição de espaços de busca têm sido pouco exploradas nos últimos anos e, por este motivo, há muito a ser conhecido;
- (vi) Para arquiteturas com grandes números de unidades ou núcleos de processamento disponíveis, pode valer a pena investigar um pouco mais as estratégias de paralelização que utilizem a estratégia de difusão;
- (vii) A comunicação assíncrona entre os *solvers* paralelizados provê, em muitos casos, grande aumento de desempenho, mas é pouco investigada pela literatura;
- (viii) A troca de contexto em *solvers* paralelizados pode acarretar em uma boa forma de guiar a execução dos processos e precisa ser melhor investigada;
- (ix) A literatura tem dado preferência a métodos híbridos (usando mais de um *solver* ao mesmo tempo), do que várias instâncias de um mesmo *solver* explorando de formas diferentes o espaço de busca;
- (x) Variações de topologias lógicas de execução de solvers paralelizados são possíveis, mas basicamente, a literatura se concentra em estrelas, anéis e completamente conectadas.

Por menores que sejam quaisquer melhorias na qualidade das soluções para o PIFRO em termos percentuais, elas podem significar grande redução no custo final de implantação de uma rede de fibras óticas. Como pode ser visto no Capítulo 3, todas as estratégias de solução presentes na literatura para o problema são processadas de forma sequencial e têm, como critério de parada, um limite de tempo. Buscando tal redução do custo, esta tese propõe a utilização dos recursos de processamento paralelo atualmente disponíveis nas GPUs Nvidia CUDA.

A motivação deste trabalho dá-se pelo fato de que, ao se aplicar técnicas de processamento paralelo às estratégias de solução para o PIFRO, diversas formas mais eficientes de exploração do espaço de busca podem ser propostas, permitindo, assim, localizar, em um menor tempo, soluções de melhor qualidade. Assim sendo, além de utilizarem os recursos de processamento paralelo em GPUs, as metaheurísticas expostas no Capítulo 7 motivam o debate sobre as lacunas encontradas na literatura apontadas anteriormente para o tema, uma vez que algumas das estratégias de solução paralelizadas para o PIFRO nesta tese serão inéditas, segundo o conhecimento do autor.

1.3 Objetivos

O objetivo geral deste trabalho é projetar, para funcionar através de paralelismo em GPUs Nvidia CUDA, estratégias de solução para o PIFRO, fazendo assim com que melhores soluções sejam encontradas e mais rapidamente, dadas as diferentes formas possíveis de exploração do espaço de busca propostas.

Para atingir o objetivo geral deste trabalho, é necessário atingir os seguintes objetivos específicos:

- (i) Conhecer a literatura referente à métodos de paralelização de metaheurísticas populacionais e de trajetória;
- (ii) Conhecer a literatura referente a padrões de projetos de metaheurísticas paralelas, para garantir a correta utilização das estruturas de dados entre os métodos e assim obter comparações justas;
- (iii) Detalhar a modelagem, linguagem de programação, API's e frameworks que são utilizados na implementação do PIFRO e suas estratégias de solução;
- (iv) Exibir a implementação do PIFRO e de suas estratégias de solução sequenciais;
- (v) Modelar, projetar e implementar as estratégias de solução sequenciais existentes em formato paralelo em GPUs Nvidia CUDA;
- (vi) Modelar, projetar e implementar estratégias de solução paralelas e que levem consideração as particularidades existentes no paralelismo em GPUs Nvidia CUDA;
- (vii) Paralelizar de forma compatível com GPUs Nvidia CUDA quaisquer algoritmos e subestratégias necessárias para a implementação dos dois itens anteriores;
- (viii) Implementar as metaheurísticas ILS-PIFRO e GA-PIFRO com recursos de paralelização em ambientes heterogêneos com GPU Nvidia CUDA seguindo a literatura atual e aplicá-los em conjunto com as novas estratégias de solução;
- (ix) Elaborar documentação detalhada de todo o projeto das implementações;
- (x) Executar testes e validar os resultados das implementações;
- (xi) Fazer análise estatística dos resultados obtidos.

1.4 Metodologia de Pesquisa

Conforme dito na Seção 1.3, o presente trabalho visa planejar, implementar, testar e analisar os resultados de estratégias de solução paralelizadas para o PIFRO em ambientes heterogêneos com GPU, para obter soluções de melhor qualidade e reduzir assim, o custo de instalação de redes de fibras óticas do PIFRO.

Para tal, inicialmente foi feita uma revisão bibliográfica do PIFRO e suas estratégias de solução. Tal estudo permitiu a otimização destas estratégias de solução, testes foram realizados e os seus respectivos resultados analisados.

Após isto, o levantamento bibliográfico e a investigação foram direcionados para o funcionamento dos ambientes de paralelização em memória compartilhada e heterogêneos com GPU. Em seguida, foram verificadas implementações prévias de metaheurísticas para solução de outros problemas de otimização combinatória nestes ambientes. Implementações iniciais do BRKGA e ILS foram testadas em outros problemas de otimização combinatória e confirmaram que diferentes implementações de estratégias de paralelização influenciam diretamente no *speedup* das metaheurísticas.

1.5 Organização da tese

Esta tese está organizada da seguinte forma. O Capítulo 2 apresenta os equipamentos óticos utilizados na operação deste tipo de rede, a versão aprimorada do modelo matemático de PIFRO como um problema de otimização (dos Reis, 2013) e a prova de sua complexidade. No Capítulo 3 é realizada uma revisão bibliográfica sobre as heurísticas para o PIFRO. No Capítulo 4 são apresentados os conceitos fundamentais sobre processamento paralelo e processamento paralelo em GPUs Nvidia CUDA. No Capítulo 5 são apresentadas as topologias de processamento paralelo mais comuns e uma taxonomia para classificação de estratégias de solução paralelas para problemas de otimização. O Capítulo 6 exibe a paralelização em baixo nível das estratégias de solução previamente existentes para PIFRO. No Capítulo 7 são apresentadas formas de paralelização das estratégias de solução para PIFRO, pensadas em topologias que levam em consideração as especificidades das GPUs Nvidia CUDA. O Capítulo 8 mostra os resultados dos experimentos computacionais realizados para avaliar as estratégias de solução paralelas para o PIFRO. O Capítulo 9 apresenta as conclusões gerais desta tese, bem como propostas de continuidade dos esforços de pesquisa aqui relatados.

Capítulo 2

Caracterização do Problema de Instalação de Fibras em Redes Óticas

O Problema de Instalação de Fibras em Redes Óticas, o qual é objeto de interesse desta tese, é apresentado neste Capítulo. A Seção 2.1 introduz o problema. As seções 2.2.1, 2.2.2 e 2.2.3 caracterizam os equipamentos óticos utilizados. A Seção 2.3 apresenta a função objetivo do problema. As seções 2.4 e 2.5 exibem respectivamente o modelo matemático e a demonstração da complexidade do problema, conforme apresentado em dos Reis et al. (2019) e dos Reis (2013).

2.1 Problema de Instalação de Fibras em Redes Óticas

Segundo Antonakopoulos e Zhang (2007), a representação do Problema de Instalação de Fibras em Redes Óticas é realizada conforme descrito a seguir. Dada uma rede ótica, sua topologia física é representada por um grafo ponderado não direcionado $G = (V, E)$. Cada nó $v \in V$ representa um ROADM (ou seja, um dos chaveadores óticos da rede). Cada aresta $[i, j] \in E$ representa um enlace da rede, o qual possui um peso, que está associado a uma distância em quilômetros l_{ij} , entre os nós de suas extremidades. Obrigatoriamente, para cada chaveador ótico, até k fibras óticas podem ser instaladas (tipicamente $k \leq 8$).

Em uma rede ótica, devem ser alocadas requisições por caminhos óticos. Seja R o conjunto de requisições por caminhos óticos. Cada requisição $r_i \in R$ é definida por:

- (i) um nó de origem s_i ;
- (ii) um nó de destino t_i ; e
- (iii) uma quantidade de comprimentos de onda w_i , necessários para transportar o tráfego entre s_i e t_i .

O Problema de Instalação de Fibras em Redes Óticas (PIFRO) consiste, então, em encontrar um caminho ótico P_i e atribuir a este todos os comprimentos de onda w_i de cada requisição $r_i \in R$, de modo que diferentes caminhos óticos possam compartilhar enlaces da rede. No entanto, para tal, devem usar comprimentos de onda diferentes. PIFRO considera que o custo total da rede ótica é dado pela soma do preço de cada um dos equipamentos necessários para o funcionamento dela. Também considera somente os três equipamentos óticos de maior valor, os quais são descritos nas seções 2.2.1, 2.2.2 e 2.2.3. Este problema

foi proposto inicialmente em Antonakopoulos e Zhang (2007) como um esforço da *Bell Labs* em desenvolver métodos de otimização para redes óticas. O objetivo do PIFRO é, portanto, minimizar o custo de instalação de equipamentos necessários para a implantação e funcionamento da rede.

Uma vez que as tecnologias atuais disponibilizam vários comprimentos de onda nas fibras, Antonakopoulos e Zhang (2007) explicam que a tarefa de atribuição destes pode ser realizada, satisfatoriamente, modelando-a como um Problema de Coloração de Grafos (PCG) em um grafo de conflitos, conforme feito em Galinier e Hertz (2006); Hertz e de Werra (1987); Hyttiä e Virtamo (1998); Noronha e Ribeiro (2006). Uma vez que existem algoritmos muito eficientes para tal, Antonakopoulos e Zhang (2007) sugerem que as estratégias de solução para o PIFRO devem restringir-se apenas ao problema de roteamento.

O Problema de Coloração de Grafos pode ser definido na seguinte forma. Seja $G' = (V', E')$ um grafo de conflitos, em que os vértices em V' correspondem aos caminhos óticos associados a cada requisição $r \in R$, ou seja $V' = \{v_i : r_i \in R\}$. Considere que exista uma aresta $[v_i, v_j] \in E'$ entre cada par de vértices cujas rotas associadas compartilham algum enlace da rede. Se os vértices que representam caminhos óticos forem coloridos com uma mesma cor, os caminhos óticos utilizam um mesmo comprimento de onda. A solução para o problema é encontrar uma configuração com um número mínimo de cores para o grafo, de forma que não exista nenhuma aresta entre dois vértices de mesma cor. Este problema é discutido de forma mais ampla em Noronha (2004).

2.2 Equipamentos Óticos

Nesta seção, são descritos os três equipamentos óticos de maior custo utilizados nas redes óticas do PIFRO.

2.2.1 ROADM (*Reconfigurable Optical Add/Drop Multiplexer*)

O equipamento ROADM (*Reconfigurable Optical Add/Drop Multiplexer*) permite que as requisições R atravessem quaisquer nós da rede sem a necessidade de conversão do sinal do domínio ótico para o domínio elétrico, ou vice versa. Cada nó da rede ótica possui um ROADM, o qual o representa. Podem ser adicionadas h portas em um ROADM e, em cada porta, pode ser conectada uma fibra ótica WDM, com capacidade para transmitir μ comprimentos de onda (normalmente, $\mu = 100$). Um ROADM pode ter no máximo b_{ij} portas (normalmente, $b_{ij} = 8$).

A um custo de uma fração de milhão de dólares, à medida que necessita-se de novas fibras, novas portas podem ser instaladas para recebê-las. A instalação de novas portas em um ROADM faz que ele tenha um custo linearmente proporcional. Na modelagem proposta em Antonakopoulos e Zhang (2007), o custo total de utilização de equipamentos ROADM na rede ótica é calculado somando-se o custo de duas portas ROADM por cada fibra ótica instalada na rede, pois é necessário ter uma porta ROADM conectada a cada extremidade da fibra ótica.

Seja a constante $ROADM_{cost}$ o custo de uma porta ROADM. O custo total de ROADMs por fibra ótica em um enlace da rede é definido por:

$$c_1 = 2 \text{ ROADM}_{cost} \quad (2.1)$$

2.2.2 OA (*Optical Amplifier*)

A dispersão cromática causa a diminuição de potência do sinal ótico, impedindo-o, muitas vezes, de chegar ao seu destino (Born e Wolf, 1999). O equipamento OA (*Optical Amplifier with Dispersion Compensation*) permite compensar sua dispersão, quando for necessária a amplificação do sinal ótico.

A constante L_{OA} representa a distância, em cada fibra ótica, em que deve ser instalado um equipamento OA. Já a constante OA_{cost} representa o custo de instalação de cada equipamento OA. Assim, o custo total de instalação de equipamentos OA em cada fibra ótica por enlace da rede é definido por:

$$c_2 = \frac{1}{L_{OA}} OA_{cost} \quad (2.2)$$

2.2.3 OT (*Optical Transponder*)

O equipamento OT (*Optical Transponder*) tipicamente é instalado apenas em locais da rede onde é feita a conversão de sinais para haver troca de dados com outros tipos de redes, dado que este equipamento faz a conversão de sinais do domínio ótico para o domínio elétrico e vice versa.

Não há forma de se saber a quantidade exata de OTs que devem ser instalados em cada enlace da rede ótica. Desse modo, em Antonakopoulos e Zhang (2007) é considerada uma distância média em cada ponto da rede ótica em que deve ser instalado um OT.

O custo de cada equipamento OT é entre 5 a 10% do valor de uma porta ROADM. Seja então:

- L_{OT} : distância em cada fibra ótica em que deverá ser instalado um equipamento OT;
- OT_{cost} : custo de instalação na rede ótica de cada equipamento OT; e
- l_{ij} : comprimento da fibra ótica na aresta $(i, j) \in E$.

O custo total de equipamentos OT por fibra ótica em cada enlace da rede é dado por:

$$c_3 = 2 \frac{1}{L_{OT}} OT_{cost} \quad (2.3)$$

2.3 Custo Total da Rede

Uma vez que todos os caminhos óticos estão roteados, o custo da instalação dos equipamentos necessários em um enlace da rede ótica $[i, j] \in E$ é dado por:

$$F_{WDM}^{ij}(\omega_{ij}, l_{ij}) = c_1 \left\lceil \frac{\omega_{ij}}{\mu} \right\rceil + c_2 l_{ij} \left\lceil \frac{\omega_{ij}}{\mu} \right\rceil + c_3 \omega_{ij} l_{ij} \quad (2.4)$$

em que:

- ω_{ij} : quantidade de comprimentos de onda a serem alocados no enlace;
- μ : número máximo de comprimentos de onda que podem ser alocados no enlace simultaneamente;

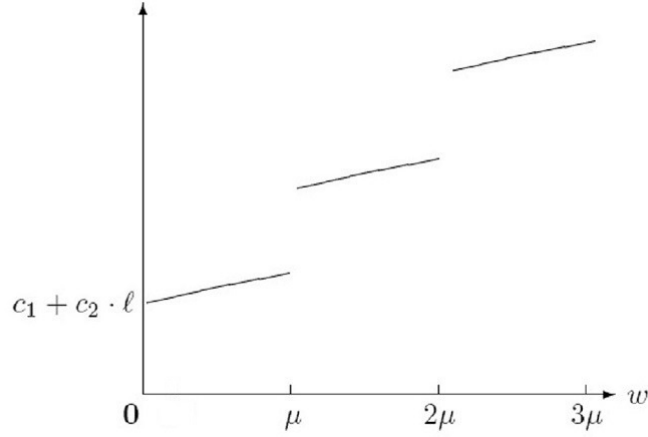


Fig. 2.1: Representação gráfica da função $F_{WDM}^{ij}(\omega_{ij}, l_{ij})$.

- l_{ij} : comprimento da fibra ótica.

A representação gráfica do comportamento da função $F_{WDM}^{ij}(\omega_{ij}, l_{ij})$ em termos do número de comprimentos de onda a serem transmitidos é apresentada na Figura 2.1. Deve-se observar o caráter linear por partes desta função.

O custo total da instalação de equipamentos necessários da rede ótica é dado pela soma do valor da função $F_{WDM}^{ij}(\omega_{ij}, l_{ij})$ para todo enlace $(i, j) \in E$.

2.4 Formulação matemática

Dado que o PIFRO é um problema de otimização, dos Reis (2013); dos Reis et al. (2019) apresentam uma formulação matemática, baseada em Programação Linear Inteira (*Integer Linear Programming*) (ILP) (Schrijver, 1986).

Dada uma instância $I^{FIP} = \langle G = (V, E), R, c_1, c_2, c_3, \mu, b_{ij}, l_{ij} \rangle$, são definidas as constantes $\pi_i^r \in \{-1, 0, 1\}$, tal que $\pi_i^r = -1$ se o nó $i \in V$ é a origem do caminho ótico para a requisição $r \in R$; $\pi_i^r = 1$, se i é o destino de r , e $\pi_i^r = 0$, caso contrário.

Esta formulação possui dois conjuntos de variáveis de decisão. As variáveis $y_{ij} \in \mathbb{N}$ representam o número de fibras óticas requeridas para alocar o tráfego que passa através da aresta $[i, j] \in E$. Além disso, para cada aresta $[i, j] \in E$, há duas variáveis $x_{ij}^r \in \{0, 1\}$ e $x_{ji}^r \in \{0, 1\}$, tal que $x_{ij}^r = 1$, se o caminho ótico da requisição $r \in R$ vai desde i até j ; $x_{ji}^r = 1$ se o caminho ótico da requisição $r \in R$ vai de j até i ; e $x_{ij}^r + x_{ji}^r = 0$, caso contrário. Nota-se que o valor das variáveis y_{ij} são definidos a partir da equação não-linear:

$$y_{ij} = \left\lceil \frac{\sum_{r \in R} w_r (x_{ij}^r + x_{ji}^r)}{\mu} \right\rceil \quad (2.5)$$

ou seja, o número de fibras óticas por aresta é igual ao teto do número de comprimentos de onda que cruzam a aresta, dividido pelo número de comprimentos de onda suportados por cada fibra. Nesta equação, w_r é o número de comprimentos de onda necessários

para transportar o tráfego associado ao caminho ótico da requisição r e μ é o número de comprimentos de onda suportados por cada fibra.

A formulação de Programação Linear Inteira para PIFRO é dada por:

$$\min \sum_{[i,j] \in E} \left[c_1 y_{ij} + c_2 l_{ij} y_{ij} + c_3 l_{ij} \sum_{r \in R} w_r (x_{ij}^r + x_{ji}^r) \right] \quad (2.6)$$

sujeito a:

$$\sum_{[j,i] \in E} x_{ji}^r - \sum_{[i,j] \in E} x_{ij}^r = \pi_i^r, \quad \forall r \in R, \forall i \in V \quad (2.7)$$

$$y_{ij} \geq \frac{\sum_{r \in R} [w_r (x_{ij}^r + x_{ji}^r)]}{\mu}, \quad \forall [i,j] \in E \quad (2.8)$$

$$y_{ij} \leq b_{ij}, \quad \forall [i,j] \in E \quad (2.9)$$

$$y_{ij} \in \mathbb{N}, \quad \forall [i,j] \in E \quad (2.10)$$

$$x_{ij}^r, x_{ji}^r \in \{0, 1\}, \quad \forall [i,j] \in E \quad (2.11)$$

A função objetivo, a ser minimizada, é dada pela expressão (2.6), que representa a soma de $F_{WDM}^{i,j}$ para todas as arestas $[i,j] \in E$. A restrição de conservação de fluxo para cada requisição $r \in R$ é dada pela expressão (2.7). As desigualdades (2.8), juntamente com as expressões (2.6) e (2.10), reforçam que o valor das variáveis y_{ij} estão de acordo com a expressão (2.5). A restrição (2.9) garante que não serão utilizadas mais fibras óticas que o máximo disponível. Finalmente, as restrições (2.10) e (2.11) definem o domínio das variáveis y_{ij} , x_{ij}^r e x_{ji}^r , respectivamente.

Um algoritmo *Branch-and-Cut* é implementado em dos Reis (2013) e dos Reis et al. (2019) para a solução desta formulação. Como o número de subproblemas explorados pelo *Branch-and-Cut* cresce três vezes exponencialmente com o número de nós da rede, a dimensão das instâncias que podem ser resolvidas na otimalidade com este algoritmo é limitada.

Esta questão, ou seja, a dificuldade de solucionar instâncias do PIFRO que tenham interesse real, abre o caminho para a discussão de sua complexidade, posta na seção a seguir.

2.5 Complexidade de PIFRO

Esta seção apresenta a demonstração que PIFRO é um problema da classe NP-Difícil. Esta demonstração baseia-se na prova realizada em dos Reis (2013) e dos Reis et al. (2019). A prova da complexidade de PIFRO é apresentada no Teorema 1.

Teorema 1. *PIFRO é um problema NP-Difícil.*

Demonstração. Para provar esta afirmação, mostra-se que PIFRO é tão difícil como qualquer outro problema em NP. Para este fim, mostra-se que o Problema de Cobertura de

Conjuntos (*Set-Cover* - SC) (Garey e Johnson, 1990) pode ser reduzido polinomialmente em PIFRO. Considere SC:

Entrada: $I^{SC} = \langle U, S, k \rangle$, em que U é um conjunto, $S \subseteq 2^U$ e $k \in \mathbb{N}$.

Pergunta: U possui uma cobertura $B \subseteq S$ de tamanho máximo k ?

Nota-se que B é uma k -cobertura de U se e somente se, para todo $u_i \in U$, existe um conjunto $B_j \in B$ tal que $u_i \in B_j$.

Sendo assim, para responder a pergunta de SC, define-se a função $g(\cdot) : I^{SC} \mapsto I^{FIP}$ que recebe uma instância $I^{SC} = \langle U, S, k \rangle$ de SC, a qual retorna uma instância $I^{FIP} = \langle G = (V, E), R, c_1, c_2, c_3, \mu, b_{ij}, l_{ij} \rangle$ de PIFRO tal que se o custo total da solução ótima da rede para I^{SC} é menor ou igual a k , I^{SC} responde *SIM*; caso contrário, I^{SC} responde *NÃO*. Primeiramente, o conjunto de nós é definido como $V = \{d\} \cup V^u \cup V^S$, onde:

$$\begin{aligned} V^U &= \{u_i : u_i \in U\}, \\ V^S &= \{S_j : S_j \in S\}, \end{aligned}$$

e d é o nó de destino de todos os caminhos ótimos da instância. Em seguida, o conjunto de arestas é definido como $E = E^{uS} \cup E^{Sd}$, onde:

$$\begin{aligned} E^{uS} &= \{[u_i, S_j] : (S_j \in S) \wedge (u_i \in S_j)\}, \text{ e} \\ E^{Sd} &= \{[S_j, d] : S_j \in S\}. \end{aligned}$$

Então, o conjunto de requisições por caminhos óticos é fixado para:

$$R = \{r_i = \langle u_i, d, 1 \rangle : u_i \in U\}.$$

Após, as constantes c_1 , c_2 , c_3 e μ são definidas respectivamente como:

$$c_1 = 0, \quad c_2 = 1, \quad c_3 = 0, \quad \text{e} \quad \mu = |U|.$$

Em seguida, a quantidade máxima de fibras por link é definida por:

$$b_{ij} = 1, \quad \forall [i, j] \in E.$$

Finalmente, a distância do link é definida por:

$$l_{ij} = \begin{cases} 0, & \forall [i, j] \in E^{uS} \\ 1, & \forall [i, j] \in E^{Sd}. \end{cases}$$

Um exemplo de rede ótica obtida à partir de uma instância de SC $U = \{1, 2, 3, 4, 5\}$ e $S = \{S_1 = \{1, 2\}, S_2 = \{1, 2, 3\}, S_3 = \{2, 3, 4\}, S_4 = \{3, 4, 5\}, S_5 = \{4, 5\}\}$ é mostrado na Figura 2.2. Esta transformação é feita de maneira tal que se a aresta $[S_j, d] \in E^{Sd}$ é utilizada pelo caminho ótico para a requisição $r_i \in R$, em seguida, o item $u_i \in U$ é coberto pelo conjunto $S_j \in S$ na instância SC correspondente. Nota-se que como $c_1 = 0$, $c_2 = 1$, $c_3 = 0$ e $\mu = |U|$, temos à partir da equação (2.4), que:

$$F_{WDM}^{ij}(\omega_{ij}, l_{ij}) = l_{ij} \cdot \left\lceil \frac{\omega_{ij}}{|U|} \right\rceil,$$

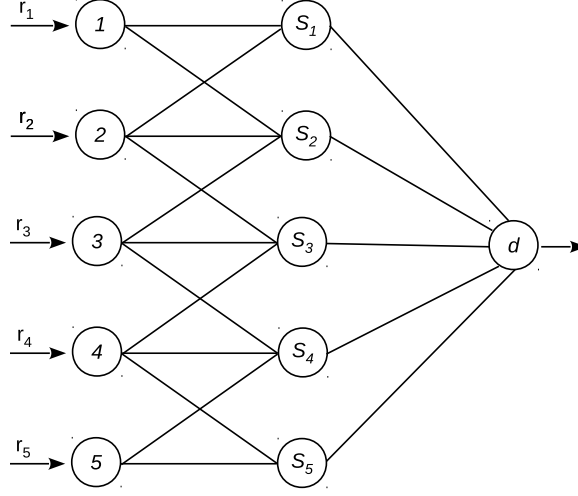


Fig. 2.2: Rede de PIFRO obtida com a transformação à partir de uma instância de SC $U = \{1, 2, 3, 4, 5\}$ e $S = \{S_1 = \{1, 2\}, S_2 = \{1, 2, 3\}, S_3 = \{2, 3, 4\}, S_4 = \{3, 4, 5\}, S_5 = \{4, 5\}\}$.

para todo $[i, j] \in E$. Assim, $F_{WDM}^{u_i S_j} = 0$, porque $l_{u_i S_j} = 0$ para todo $[u_i, S_j] \in E^{uS}$. Além de que,

$$F_{WDM}^{S_j d} = \left\lceil \frac{\omega_{S_j d}}{|U|} \right\rceil \leq 1,$$

porque $l_{S_j d} = 1$ para todo $[S_j, d] \in E^{Sd}$, e $\omega_{S_j d} \leq \sum_{r_i \in R} w_i = |R| = |U|$. Portanto, $F_{WDM}^{S_j d} = 0$,

se $\omega_{S_j d} = 0$, isto é, se a aresta $[S_j, d] \in E^{Sd}$ não é usada pelo caminho ótico, e $F_{WDM}^{S_j d} = 1$, se $\omega_{S_j d} > 0$, isto é, se esta aresta é usada por pelo menos um caminho ótico. Consequentemente, o custo da solução $P = \{P_i : r_i \in R\}$ é igual ao número de aresta em E^{Sd} usada pelo caminho ótico em P .

Dada a solução ótima $P^* = \{P_i^* : r_i \in R\}$ of I^{PIFRO} , se o custo de P^* é menor ou igual a k , pode-se fazer uma k -cobertura $B \subseteq S$ para I^{SC} com $B = \{S_j \in S : [S_j, d] \in E^{Sd} \cap E^{P^*}\}$, onde $E^{P^*} = \bigcup_{P_i^* \in P^*} P_i^*$ é o conjunto de todas as arestas usadas pelos caminhos óticos em

P^* . Uma vez que o custo de P^* é menor ou igual a k , ela usa no máximo k arestas em E^{Sd} , então $|B| \leq k$, e a resposta para I^{SC} neste caso é SIM .

Por fim, resta provar que se o custo de P^* é maior que k , da instância SC correspondente, responda $NÃO$. Se o custo de P^* é maior que k e a resposta para I^{SC} foi SIM , seria possível construir uma solução $\bar{P} = \{\bar{P}_i = \{[u_i, S_j], [S_j, d]\} : S_j \in B \text{ e } u_i \text{ é coberta por } S_j \text{ in } B\}$. Desde que foi assumido que $|B| \leq k$, $\bar{P}$ utiliza no máximo k arestas em E^{Sd} , e seu custo é menor ou igual a k . Como isso contradiz a hipótese que P^* é a solução ótima, se o custo de P^* é maior que k , da instância SC correspondente, responde $NÃO$, o que completa esta prova. $\square$

A caracterização de PIFRO como um problema NP-Difícil permite a utilização de métodos heurísticos e metaheurísticos para solucioná-lo. O Capítulo 3 apresenta uma revisão de técnicas de solução desse problema.

Capítulo 3

Estratégias de Solução: Revisão Bibliográfica

Este capítulo está organizado como segue. A Seção 3.1 apresenta uma introdução geral ao tema abordado no capítulo. A Seção 3.2 revisa as heurísticas propostas para a solução do Problema de Instalação de Fibras em Redes Óticas (PIFRO), ou seja, as heurísticas *Greedy* e PSC (*Piecewise Strongly Concave*). A Seção 3.3 descreve uma solução do PIFRO via algoritmo genético e a Seção 3.4 descreve a implementação da metaheurística *Iterated Local Search* (ILS) para a solução desse problema.

3.1 Introdução

Segundo Sörensen (2015), os algoritmos utilizados para solução de problemas de otimização combinatória (*solvers*) são classificados em exatos (Wolsey, 1998), heurísticos (Talbi, 2009) e aproximativos (Vazirani, 2001). A diferença é que os algoritmos exatos são projetados para encontrar a melhor solução possível de um problema (solução ótima). Porém, o tempo necessário para que o algoritmo exato encerre sua execução pode inviabilizar sua aplicação. Por outro lado, os algoritmos heurísticos (ou também frequentemente chamados simplesmente de heurísticas) não oferecem tal garantia, retornando, na maior parte das vezes, uma solução pior que a solução ótima. Entretanto, através de experimentos computacionais, pode-se verificar que, para determinados problemas, as soluções encontradas por heurísticas são de boa qualidade. Há uma classe intermediária entre estes, que é chamada de aproximativos, os quais oferecem uma garantia de que retornam uma solução que está a uma determinada porcentagem da solução ótima. Os algoritmos aproximativos também podem ser pensados como “heurísticas com alguma garantia” ou “não muito exatos” (Sörensen, 2015). A decisão de usar algoritmos exatos, metaheurísticas ou aproximativos pode depender da complexidade do problema, da dimensão e da estrutura das instâncias.

Devido à necessidade de garantir que a solução ótima foi encontrada, os algoritmos exatos precisam examinar todo o espaço de buscas ou domínio (ou seja, as soluções possíveis) para o problema, verificando mesmo aquelas que explicitamente não precisam ser examinadas. Diversas técnicas permitem que esta busca seja feita de forma mais eficiente, eliminando grandes conjuntos de soluções a serem pesquisadas a cada iteração do algoritmo. A espera pelo término da execução de um algoritmo exato pode ser inviável ou mesmo, dada a dimensão do problema, este pode ser intratável por este método.

De outro lado, a eliminação da necessidade de garantir o retorno da solução ótima para um problema torna o tempo de espera do término de execução de heurísticas significativamente menor, o que, ao longo do tempo, fez com que estes métodos se tornassem cada vez mais atrativos, dado o melhor aproveitamento prático. Para atingir este objetivo, heurísticas normalmente não examinam todas as soluções disponíveis no espaço de busca, terminando sua execução e retornando a melhor solução encontrada até que um critério de parada seja atingido (o qual pode ser tempo de execução, quantidade de soluções examinadas, quantidade de iterações do algoritmo, entre outros).

3.2 Heurísticas para PIFRO

Em Antonakopoulos e Zhang (2007) foram propostas e avaliadas duas heurísticas construtivas para PIFRO: Heurística *Greedy* e Heurística PSC (*Piecewise Strongly Concave*). Nestas heurísticas foi verificado que a ordem na qual as requisições são alocadas na rede ótica influencia no custo final da rede. Elas recebem, como parâmetro, as requisições $r \in R$ ordenadas aleatoriamente. Estas heurísticas são descritas a seguir.

3.2.1 Heurística *Greedy*

A heurística *Greedy*, proposta em Antonakopoulos e Zhang (2007), é uma heurística gulosa. A cada iteração, para cada requisição $r \in R$, a heurística *Greedy* calcula o custo incremental da função (2.4) em relação ao número de comprimentos de ondas, dado por:

$$C(i) = F_{WDM}(\omega_i + \alpha, l) - F_{WDM}(\omega_i, l) \quad (3.1)$$

em que α é o número de comprimentos de onda a serem adicionados nos enlaces da rede ótica. Em seguida, a heurística *Greedy* executa o Algoritmo de Dijkstra (Dijkstra, 1959) para encontrar o caminho ótico com menor custo incremental para a requisição. A heurística para quando todas as requisições tenham um caminho ótico definido.

O Algoritmo 1 exibe o pseudocódigo da heurística *Greedy*. São recebidos, como parâmetros de entrada, o grafo G , com a topologia física da rede, as requisições $r \in R$, na ordem em que serão avaliadas, o limite de comprimento de ondas por fibra μ e os custos dos equipamentos óticos. Em cada iteração do laço das linhas 1 e 6 uma requisição de R é alocada. Na linha 2, é calculado o custo de alocar cada requisição. Na linha 5, é definido o caminho ótico para a requisição através do Algoritmo de Dijkstra (Dijkstra, 1959). Após todas as requisições serem alocadas em seus respectivos caminhos óticos, a heurística é encerrada.

3.2.1.1 Busca Local

Após encontrado o caminho com menor custo incremental para todas as requisições $r \in R$, é realizado um procedimento de busca local. Este procedimento é do tipo *1-reroteamento*, em que uma requisição é removida e novamente adicionado o seu custo incremental a um novo caminho ótico com o menor custo, se possível, ou utilizando o caminho ótico já existente, caso contrário. Este processo é realizado para cada requisição $r \in R$ alocada na rede ótica, até que não seja mais possível encontrar nenhum caminho ótico com custo incremental menor ou caminho ótico diferente para nenhuma requisição.

Algorithm 1: Greedy

Entrada: $G, R, \mu, ROADM_{Cost}, OT_{Cost}, OA_{Cost}$

```

1 for elemento do conjunto de requisições  $R$  do
2   for enlace do grafo da rede ótica do
3     | Calcular o custo incremental do enlace;
4   end
5   Encontrar o caminho ótico de menor custo incremental através do algoritmo de
     | Dijkstra;
6 end

```

Algorithm 2: Busca Local

Entrada: $G, R, \mu, ROADM_{Cost}, OT_{Cost}, OA_{Cost}$

```

1 for elemento do conjunto de requisições  $R$  do
2   Remover o custo da requisição;
3   for enlace do grafo da rede do
4     | Calcular custo da adição da requisição;
5   end
6   Encontrar o caminho ótico de menor custo pelo algoritmo de Dijkstra;
7   if foi encontrado caminho ótico com custo incremental menor then
8     | Alocar a requisição no novo caminho ótico;
9     | Reinicia laço da linha 1;
10  end
11  Alocar requisição no caminho ótico original;
12 end

```

O pseudocódigo da busca local é apresentado no Algoritmo 2. São recebidos, como parâmetros de entrada, o grafo G com a topologia física da rede, as requisições $r \in R$ na ordem em que serão avaliadas, o limite de comprimento de ondas por fibra μ e os custos dos equipamentos óticos. Inicia-se uma busca local no laço da linha 1. Na linha 2, cada requisição de R é retirada do grafo que representa a rede e tenta-se encontrar um caminho ótico com menor custo (linhas 3 a 6), devido aos equipamentos que foram instalados na rede após a alocação inicial desta requisição. Se um caminho ótico com menor custo ou um caminho ótico diferente for encontrado para requisição, inicia-se novamente o processo de busca local (linha 9). Ao final da execução do algoritmo, obtém-se o custo total da inserção de todas as requisições. Caso não seja encontrado um melhor caminho ótico para a requisição retirada, a requisição será inserida no caminho ótico original (linha 12). A heurística *Greedy* encerrará sua execução quando o laço da linha 1 for executado sem que seja encontrado nenhum caminho ótico melhor ou diferente para as requisições.

3.2.2 Heurística PSC

A heurística PSC (*Piecewise Strongly Concave*), também proposta em Antonakopoulos e Zhang (2007), é semelhante à heurística *Greedy*, mas utiliza uma função de custo gulosa diferente. Enquanto a heurística *Greedy* utiliza a função F_{WDM} apresentada na Expressão (2.4), a heurística PSC utiliza uma função de custo gulosa mais sofisticada, dada

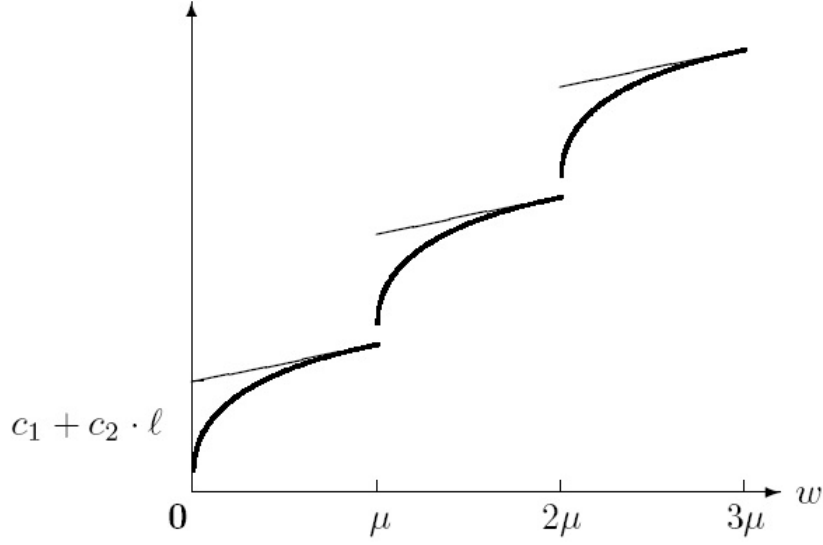


Fig. 3.1: Representação gráfica da função $F'_{WDM}(\omega, l, p, P)$ para $p = 1$ e $P = 5$.

por:

$$F'_{WDM}(\omega, l, p, P) = (c_1 + c_2 l) \left[\frac{p}{P} \left\lceil \frac{\omega}{\mu} \right\rceil + \frac{P-p}{P} f\left(\frac{\omega}{\mu}\right) \right] + c_3 \omega l \quad (3.2)$$

em que:

- p : iteração corrente do algoritmo;
- P : um valor inteiro; e
- $f(x) = \lfloor x \rfloor + \sqrt{x - \lfloor x \rfloor}$, sendo

$$x = \frac{\omega}{\mu}$$

No primeiro passo do algoritmo de busca local da heurística PSC, p é igual a zero. Neste momento, a função de custo é equivalente a $f\left(\frac{\omega}{\mu}\right)$. A cada iteração da heurística sobre o conjunto de requisições $r \in R$, o valor de p é incrementado. Desta forma, F'_{WDM} converge gradualmente para F_{WDM} , com uma aproximação côncava inferior. A velocidade da convergência é definida pelo parâmetro P . Nos experimentos em [Antonakopoulos e Zhang \(2007\)](#), foi definido o valor de P como $P = 5$.

A função $F'_{WDM}(\omega, l, p, P)$ está representada na Figura 3.1.

A função de custo gulosa utilizada na Heurística PSC privilegia a adição de novos comprimentos de onda nas fibras óticas que já estejam utilizando um número de comprimentos de onda próximos à sua capacidade, promovendo uma melhor utilização dos equipamentos óticos instalados. Com esta função de custo, é possível alterar os caminhos óticos das requisições que passam por fibras óticas que estão com menor quantidade de comprimentos de onda alocados, eliminando a necessidade da utilização desta fibra.

O Algoritmo 3 exibe o pseudocódigo da heurística PSC. O pseudocódigo da heurística PSC é semelhante ao da heurística *Greedy*. A diferença é a função utilizada para calcular

Algorithm 3: PSC

Entrada: $G, P, \mu, ROADM_{Cost}, OT_{Cost}, OA_{Cost}$

```

1 for elemento do conjunto de requisições  $R$  do
2   for enlace do grafo da rede ótica do
3     | Calcular custo da adição da requisição;
4   end
5   Encontrar o caminho ótico de menor custo incremental através do algoritmo de
     Dijkstra;
6   Executar uma busca local na solução corrente;
7   Incremente o valor de  $p$ ;
8 end

```

o custo incremental F'_{WDM} . Quando o passo p da iteração sobre o conjunto de requisições realizada pela busca local atingir o valor P , o custo para cálculo do caminho ótico é realizado pela função F_{WDM} . A linha 8 do pseudocódigo da heurística PSC incrementa em uma unidade o valor de p .

3.3 Heurística GA-PIFRO

Um Algoritmo Genético Baseado em Chaves Aleatórias para PIFRO foi proposto em Goulart et al. (2011). O uso de um Algoritmos Genéticos Baseados em Chaves Aleatórias foi motivado pelo sucesso da utilização desta metaheurística para a resolução de problemas de roteamento em redes de computadores (Buriol et al., 2005, 2007; Gonçalves e Resende, 2004, 2011; Dias et al., 2011; Noronha et al., 2011; Reis et al., 2011), assim como em diversos outros problemas de otimização combinatória (Arulselvan et al., 2007; Gonçalves et al., 2005; Lawrence e Mark, 2006; Malve e Uzsoy, 2007; Samanlioglu et al., 2008; Coco et al., 2013; de Faria Jr. et al., 2017; Cavalheiro e Lyra Filho, 2017; Silva et al., 2020; Prasetyo et al., 2018; Almeida et al., 2018; Andrade et al., 2013; Amaro Junior et al., 2017; Ruiz et al., 2015).

O algoritmo GA-PIFRO apresentado em Goulart et al. (2011) faz uso das heurísticas *Greedy* e PSC, obtendo, no entanto, resultados melhores. Os cromossomos são compostos por vetores de requisições. Cada requisição recebe um número real aleatório entre 0 e 1. Posteriormente, a população é ordenada em ordem crescente, de acordo com o valor aleatório atribuído a cada requisição. A sequência resultante da ordenação das requisições é passada como parâmetro para as heurísticas *Greedy* ou PSC e é feita uma busca local. A solução retornada pela heurística *Greedy* ou PSC é atribuída como valor de adaptabilidade do cromossomo.

Em todas as gerações, a população é dividida em dois conjuntos, denominados *TOP* e *REST*. O conjunto *TOP* da população armazena os cromossomos com as melhores soluções. Estes indivíduos são copiados, sem alterações, para as próximas gerações. O restante da população da nova geração é encontrado pela aplicação de operadores de cruzamento e mutação.

O operador de cruzamento *Parametrized Uniform Crossover* (Spears e deJong, 1991) foi utilizado em GA-PIFRO. O cruzamento é realizado entre um cromossomo da parcela *TOP* da população com um cromossomo da parcela *REST*. Cada filho recebe suas chaves

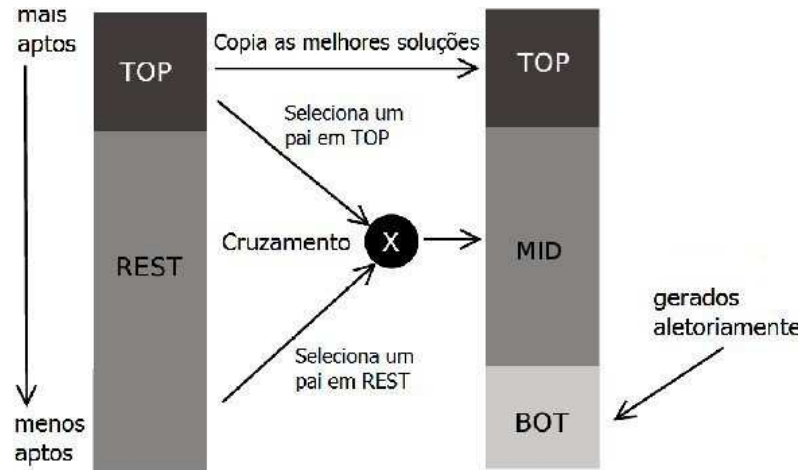


Fig. 3.2: Transição entre gerações sucessivas (Noronha et al., 2011).

ou da porção da população mais apta (ou *TOP*), com uma probabilidade de $(0,7)$; ou a chave é herdada do cromossomo pertencente à população menos apta (ou *REST*) com probabilidade de $(0,3)$. Assim, o tamanho total da população inicial é $TOP + REST$ e a quantidade de cromossomos criados a cada geração através de cruzamento é $REST - BOT$ cromossomos filhos. Cromossomos mutantes são colocados na parcela *BOT* da nova população gerada.

Não existe mutação no Algoritmo GA-PIFRO, pois, a cada geração, um número fixo de cromossomos mutantes é inserido na população. Para escapar de mínimos locais, uma parte dos cromossomos menos aptos é substituída por cromossomos gerados aleatoriamente, de forma semelhante a como é gerada a população inicial.

Em Lucena et al. (2014) são propostas alterações para a metaheurística BRKGA, associadas ao cruzamento dos cromossomos. São elas:

- (i) Definição de gênero: assim como ocorre na natureza, um cromossomo é do gênero masculino e o outro do gênero feminino. Inicialmente, cada cromossomo recebe um bit, o qual indicará seu gênero. A escolha aleatória dos cromossomos aptos ao cruzamento é realizada. O gênero do cromossomo obtido pelo cruzamento é definido de forma aleatória;
- (ii) Reprodução por multipaternidade: alguns cromossomos são selecionados para o cruzamento e geram um novo indivíduo. É realizada a escolha aleatória de alguns cromossomos e, através de uma probabilidade π_t , cada cromossomo selecionado terá a quantidade de seus alelos que irão compor o novo cromossomo.

Segundo Lucena et al. (2014), estas duas alterações no operador de cruzamento do BRKGA produziram resultados de melhor qualidade e em menor tempo para os problemas de otimização *Set Covering* e *Set Packing*.

Os tamanhos das parcelas *TOP*, *REST* e *BOT* são informados através de parâmetros ao algoritmo. A Figura 3.2 mostra a evolução da população do algoritmo genético entre duas gerações.

O pseudocódigo do algoritmo genético utilizado em Goulart et al. (2011) é descrito no Algoritmo 4. Na linha 2, os vetores de chaves são inicializados com valores aleatórios. Na linha 3, é realizada a avaliação da função de custo para cada vetor. O laço entre as linhas 5

Algorithm 4: GA-PIFRO

Entrada: G, R, p_{red}, T_{max}

```

1 for elemento do conjunto de requisições  $R$  do
2   | Gerar vetor com chaves aleatórias;
3   | Avaliar vetor de acordo com a função de custo;
4 end
5 while critério de parada não for satisfeito do
6   | Ordene a população pelo valor retornado pela função de custo;
7   | Divide em  $TOP$ ,  $MID$  e  $BOT$ ;
8   | Copiar para a próxima geração a parcela  $TOP$  da população;
9   for  $n \leftarrow 1$  até  $|MID|$  do
10    | Selecione aleatoriamente um elemento da parcela  $TOP$ ;
11    | Selecione aleatoriamente um elemento da parcela  $MID \cup BOT$ ;
12    for chave do novo cromossomo do
13      |  $p_3 \leftarrow Crossover(p_1, p_2, p_{rec})$ ;
14    end
15  end
16  | Gerar aleatoriamente parcela  $BOT$  da nova população;
17  | Avaliar cada elemento da população;
18 end

```

e 18 é executado até que o tempo T_{max} seja alcançado. Na linha 6, a população é ordenada de acordo com o valor retornado pela função de custo. Na linha 8, é realizada a reprodução da porção TOP da população, que é integralmente copiada para a próxima geração. Entre as linhas 9 e 15 é realizado o cruzamento, gerando os cromossomos da porção MID da população. Na linha 16 são gerados os novos elementos da parcela BOT , responsáveis pela diversificação da população. Na linha 17, a nova população é avaliada.

3.4 Heurísticas ILS-PIFRO

Em [dos Reis et al. \(2013\)](#) e [dos Reis et al. \(2019\)](#) são propostas duas heurísticas chamadas ILS-GREEDY e ILS-PSC para solucionar o PIFRO, ambas baseadas na metaheurística *Iterated Local Search* (ILS), proposta inicialmente em [Lourenço et al. \(2010\)](#). O ILS executa buscas no espaço de um ótimo local. Iniciando a partir de um ótimo local s , a cada iteração é feita uma perturbação em s , seguida de uma busca local na solução perturbada, produzindo, em seguida, uma nova solução ótima local chamada s' . A perturbação consiste em movimentos aleatórios em uma determinada vizinhança. Tal perturbação deve ser forte o suficiente para gerar uma solução fora da região de atração do ótimo local corrente; caso contrário, a busca local seguinte leva novamente a ele. Ao mesmo tempo, a perturbação não pode ser demasiadamente forte, pois, assim, pode ter funcionamento similar a uma busca aleatória. Se s' satisfaz algum critério de aceitação, o procedimento é repetido a partir de s' ; caso contrário, a metaheurística continua a partir de s .

A metaheurística ILS vem sendo utilizada com sucesso para solucionar diferentes problemas de otimização combinatória como *Routing and Wavelength Assignment Problem* ([Martins et al., 2012](#)), *Car Sequencing Problem* ([Aloise et al., 2008](#); [Ribeiro et al., 2008](#)),

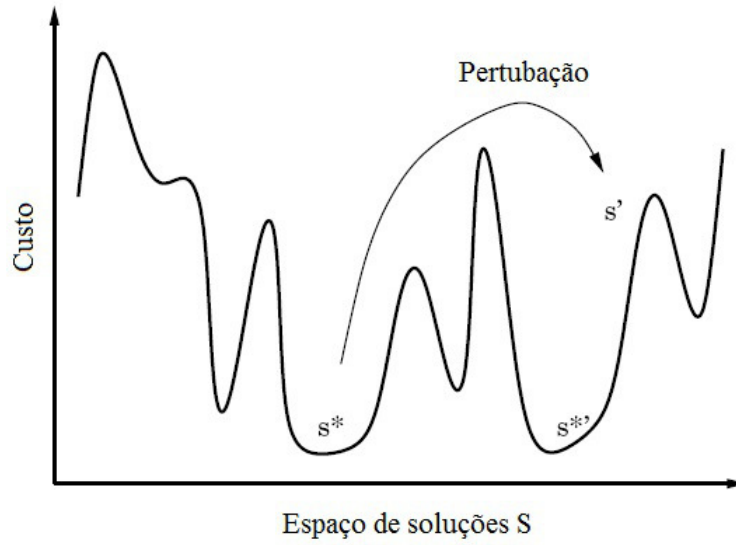


Fig. 3.3: Funcionamento da heurística ILS-PIFRO.

Referee Assignment Problem (Duarte et al., 2007), *Traveling Tournament Problem* (Costa et al., 2012), *Machine Reassignment Problem* (Lopes et al., 2015), *Vehicle Routing Problem* (Cattaruzza et al., 2014; Michallet et al., 2014; Morais et al., 2014; Silva et al., 2015), *Flowshop Problem* (Dong et al., 2013), *Bipartite Unconstrained 0-1 Quadratic Programming Problem* (Duarte et al., 2014), *Bandwidth Coloring Problem* (Lai e Lü, 2013), dentre outros.

Esta metaheurística foi utilizada para solucionar o PIFRO em dos Reis et al. (2019) e apresenta a melhor qualidade de solução para tal até então. A implementação realizada recebe o nome de ILS-PIFRO. A Figura 3.3 ilustra o funcionamento de ILS-PIFRO.

A heurística ILS-PIFRO proposta neste trabalho faz o uso da mesma busca local 1 – *reroteamento* de Antonakopoulos e Zhang (2007), em que cada caminho ótico é removido da rede e reinserido com o menor custo possível nas circunstâncias atuais. Dada uma solução s , é aplicada uma perturbação em k caminhos óticos, que serão utilizados para gerar uma solução vizinha a s . Primeiro, um conjunto K com $k = |K|$ caminhos óticos são selecionados aleatoriamente. A heurística ILS-PIFRO pode selecionar caminhos óticos aleatoriamente ou pode ser ajustada para a probabilidade de um caminho ótico selecionado ser proporcional ao número de comprimentos de onda que ele armazena. Em seguida, os k caminhos óticos são removidos de s . Isso resulta em uma redução do custo da rede quanto aos equipamentos utilizados para transportar o tráfego dos caminhos óticos em K , pois não são mais necessários. Em seguida, cada caminho ótico k é iterativamente reinserido em s , procurando um caminho ótico de menor custo sob as circunstâncias atuais.

O Algoritmo 5 mostra o pseudocódigo de ILS-PIFRO. Começando a partir da solução inicial s' fornecida pela heurísticas *Greedy* ou PSC, inicialmente, a busca local é aplicada a s' na linha 1, para produzir a solução ótima local s^* . Em seguida, o laço das linhas 2-7 é realizado, até o critério de parada ser satisfeito. Novos ótimos locais são obtidos através da aplicação de uma perturbação gulosa de tamanho k na linha 3, seguida pela busca local 1 – *reroteamento* para a solução obtida na linha 4. Em seguida, se a solução resultante é

Algorithm 5: ILS-PIFRO

Entrada: s'

```

1  $s, s^* \leftarrow \text{BuscaLocal}(1 - \text{opt}, s')$ ;
2 while critério de parada não for satisfeito do
3    $s' \leftarrow \text{Perturbação}(k - \text{reroteamento}, s)$ ;
4    $s' \leftarrow \text{BuscaLocal}(1 - \text{reroteamento}, s')$ ;
5   if  $s'$  é melhor que  $s^*$  then
6      $s^* \leftarrow s'$ ;
7   end
8   if CritérioDeAceitação( $s, s'$ ) then
9      $s \leftarrow s'$ ;
10  end
11 end
12 Retorne  $s^*$ ;

```

melhor que a solução corrente s^* , a solução corrente é atualizada na linha 5. Na sequência, a solução ótima local corrente s pode ser substituída pela solução s' , de acordo com um critério de aceitação definido na linha 6. A solução s' é aceita se o seu custo for menor que o custo de s . A melhor solução encontrada por esta heurística é retornada na linha 8.

3.4.1 Critérios de aceitação

Dois critérios de aceitação podem ser utilizado pela heurística ILS-PIFRO para determinar quando é aceita ou não uma solução corrente:

- (i) critério de aceitação Melhor (*Better*); e
- (ii) critério de aceitação Reinício (*Restart*).

Critério de aceitação Melhor

O critério de aceitação Melhor (*Better*) permite uma intensificação mais forte, aceitando somente soluções melhores (Lourenço et al., 2002). Sejam s e s' as duas solução a serem comparadas pelo critério de aceitação da heurística ILS-PIFRO. O critério de aceitação *Better* é definido por:

$$\text{Better}(s, s') = \begin{cases} s', & \text{se } \text{Custo}(s') < \text{Custo}(s); \\ s, & \text{caso contrário.} \end{cases} \quad (3.3)$$

Para tentar escapar de ótimos locais, o critério de aceitação *Better* pode ser modificado através de duas opções:

- (i) que sejam aceitas soluções com custo menor ou igual ao custo da solução corrente;
- (ii) que sejam aceitas soluções com um custo maior que o custo da solução corrente.

Para isso, um percentual β , que contém a porcentagem máxima de custo extra, deve ser ajustado.

Critério de aceitação Reinício

O critério de aceitação Reinício (*Restart*) pode ser utilizado pela metaheurística ILS-PIFRO quando a intensificação se torna ineficiente. O critério *Restart* aceita uma solução s gerada aleatoriamente como nova solução corrente e é definido por:

$$Restart(s, s') = \begin{cases} s', & \text{se } Custo(s') < Custo(s); \\ s, & \text{se } Custo(s') \geq Custo(s) \text{ e } t > t_{\max}. \end{cases} \quad (3.4)$$

Duas opções de reinício podem ser selecionadas para *Restart*:

- (i) Seja t o contador de tempo de execução da heurística ILS-PIFRO sem melhora na solução corrente e $t_{\max}$ o tempo máximo sem melhora antes de *Restart* gerar a nova solução aleatória s ; ou
- (ii) Seja x o contador de número de iterações da heurística ILS-PIFRO sem melhora na solução corrente e $x_{\max}$ a quantidade máxima de iterações sem melhora antes de *Restart* gerar a nova solução aleatória s .

Capítulo 4

Paralelismo, Placas de Processamento Gráfico e GPUs Nvidia CUDA

Este capítulo se organiza da seguinte forma. A Seção 4.1 introduz o contexto aqui tratado. A Seção 4.2 revisa as estruturas de processamento paralelo existentes. A Seção 4.3 trata das classes de arquiteturas de paralelismo. A Seção 4.4 discute a avaliação do desempenho de algoritmos paralelos. A Seção 4.5 apresenta as placas de processamento gráfico (GPU) e, finalmente, a Seção 4.6 apresenta as GPUs NVIDIA.

4.1 Introdução

Conforme visto no Capítulo 3, os resultados produzidos pelas estratégias de solução para o PIFRO são processados em computadores de forma sequencial. Cabe ressaltar que, independentemente da satisfação com o tempo necessário para encontrar as soluções, pequenas melhorias feitas em cada uma delas são bem vindas, já que cada décimo de percentual de melhoria atingido pode significar uma economia na casa de centenas de milhares de dólares. Esta melhoria pode ser obtida caso o *solver* explore de forma mais eficiente o espaço de busca do problema (domínio). Por este motivo, além de melhorar o desempenho, neste trabalho tenta-se também atingir resultados ainda melhores, variando técnicas para aprimorar a busca por soluções através de paralelismo em GPUs Nvidia CUDA.

Para compreender a atual forma de processamento paralelo das GPUs, é necessário prévio conhecimento das técnicas de processamento paralelo que as precederam assim como as atuais técnicas e sistemas concorrentes, conforme é exposto a seguir.

Em um período de 16 anos (1986 à 2002), os processadores tiveram um aumento de desempenho médio de 50% a cada ano. Desde então, o aumento de desempenho dos processadores começou a reduzir em 20% ao ano (Herlihy e Shavit, 2008). Sendo assim, diversos fabricantes decidiram que os avanços no desempenho de programas deveria ser dado por técnicas de processamento em paralelo. O processamento em paralelo ou paralelismo é uma técnica que permite que algoritmos executem instruções de forma simultânea em determinados momentos, para aumentar sua velocidade de execução (Pacheco, 2011). Segundo (Pacheco, 2011), o processamento paralelo pode ser dividido em:

- (i) Paralelismo de dados: quando o algoritmo paralelo executa uma mesma tarefa, porém simultaneamente em diversos containers de dados;

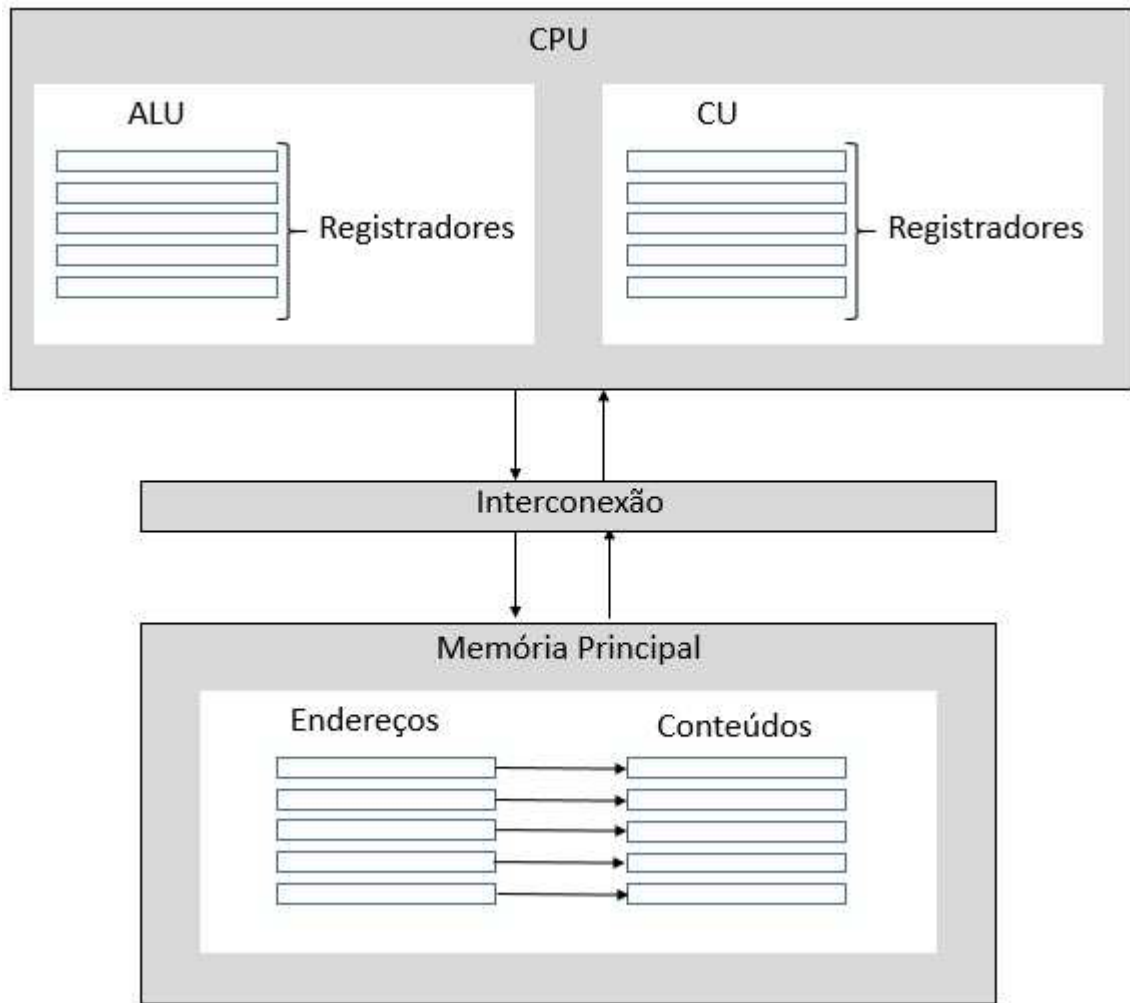


Fig. 4.1: Arquitetura de von Neumann.

- (ii) Paralelismo de tarefas: quando o algoritmo paralelo executa diversas tarefas de forma simultânea em um único container de dados.

Ainda segundo Pacheco (2011), a simples adição de mais núcleos de processamento na execução de algoritmos originalmente escritos para serem executados de forma sequencial não é capaz de aproveitar a total capacidade de aumento de desempenho que computadores dotados com diversas unidades de processamento (ou núcleos de processamento) podem proporcionar. Em Pacheco (2011) destaca-se a importância que a implementação do algoritmo a ser executado de forma paralela considere particularidades do *hardware* em que será executado, e que este seja implementado de forma a executar, nos momentos em que for possível, o processamento paralelizado.

4.2 Hardware e Software Paralelizados

Tanto *hardwares* quanto *softwares* paralelizados evoluíram a partir de *hardwares* e *softwares* que executam apenas uma instrução por vez, ou seja, fundamentam-se em *hardware* e *software* de processamento sequenciais.

A arquitetura de processamento mais difundida é a Arquitetura de von Neumann (Verhulst, 2003). Esta arquitetura mescla, através de modificações realizadas ao longo do tempo, técnicas para processamento sequencial e paralelo de algoritmos, sendo composta pelas estruturas exibidas na Figura 4.1.

As estruturas identificadas na Figura 4.1 são:

- Unidade Central de Processamento (*Central Processing Unit* - CPU) também chamada frequentemente de processador ou núcleo de processamento (*core*), a qual é capaz de executar instruções uma por vez. Segundo Verhulst (2003), uma CPU se divide em uma Unidade de Controle (*Control Unit* - CU), a qual é responsável por definir quais as instruções necessárias para o algoritmo que devem ser executadas, e uma Unidade Lógica e Aritmética (*Aritmetic and Logical Unit* - ALU), que é responsável por executar as instruções previamente definidas pela Unidade de Controle. Os dados sobre a execução do algoritmo e os dados da CPU são armazenados nos chamados registradores. Dentre os registradores, existe um tipo em especial, o qual armazena o endereço da próxima instrução a ser executada, nomeado como Contador de programa (*Program Counter*);
- Memória principal, que é uma coleção de localidades, compostas por um endereço dentro da memória e um respectivo conteúdo, capazes de armazenar dados e instruções;
- Interconexão entre a CPU e a memória principal, para armazenar, solicitar, ler, excluir ou alterar informações. Comumente chamada barramento (*bus*), é composto por um conjunto de vias paralelas de comunicação, as quais têm seu acesso controlado por algum *hardware*. Normalmente, a velocidade do processador é muito superior à velocidade da memória, e, por este motivo, o *bus* precisa determinar a velocidade do fluxo de informação que transita entre CPU e memória principal, gerando uma espécie de fila. A este problema é dado o nome de Gargalo de von Neumann (*von Neumann Botleneck*).

4.2.1 Modificações da arquitetura de von Neumann

Segundo Pacheco (2011), a arquitetura de von Neumann, proposta inicialmente na década de 1940, teve diversas modificações sugeridas ao longo do tempo, desde que o primeiro computador eletrônico digital foi criado. Muitas das modificações sugeridas têm, como principal alvo, atenuar o problema do gargalo (*bottleneck*) de von Neumann. Dentre todas elas, as que atualmente são amplamente utilizadas, são:

- Cache: consiste em uma porção de memória alocada dentro da CPU ou em algum local do computador no qual há um barramento mais veloz, aumentando a velocidade de acesso a dados ou a instruções cruciais à continuidade da execução do algoritmo. Nos computadores atuais, existem diferentes níveis de memória cache, de modo que quanto maior a prioridade de acesso aos dados ou instruções do algoritmo, mais rápido é seu barramento de acesso. A memória cache utiliza um princípio, em que é previsto que os algoritmos necessitarão dos dados ou instruções que a menos tempo foram utilizados(as), para dar-lhes a devida prioridade e selecionar o nível de cache em que devem ser armazenados;

- **Memória Virtual:** algoritmos que geram ou precisam acessar conjuntos de dados relativamente grandes correm o risco destes, devido ao seu tamanho, não possam ser armazenados na memória principal ou na memória cache da CPU. A memória virtual foi criada para tornar disponível, em uma memória secundária, um local (chamado espaço de troca) em que estes dados podem ser armazenados e acessados como se estivessem na memória principal. No entanto, como o acesso à memória secundária é sempre mais lento, o acesso à memória virtual também é consideravelmente mais lento do que o acesso à memória principal ou memória cache;
- **Paralelismo em nível de instruções (*Instruction Level Parallelism* - ILP):** caso existam múltiplas unidades funcionais da CPU disponíveis, ou ociosas, o ILP tenta melhorar o desempenho da execução do programa executando diversas instruções ao mesmo tempo. O ILP é abordado de duas formas:
 - (i) o *Pipelining* é uma técnica na qual é feita uma busca das instruções que serão necessárias para a execução do algoritmo e criada uma fila dentro da memória da CPU. As instruções aguardam o momento devido de serem executadas. Uma instrução, para ser executada, é colocada no primeiro estágio da fila e, ao concluir sua tarefa, parte para o segundo estágio, dando condições assim, para que imediatamente, a próxima instrução seja colocada em primeiro estágio para ser executada;
 - (ii) Os Problemas Múltiplos (*Multiple Issues*), replicando o algoritmo nas diversas unidades funcionais da CPU, tentam executar nestas diversas instruções simultaneamente, para aumentar a velocidade de processamento;
- **Paralelismo em nível de *thread* (*Thread-Level Parallelism* - TLP):** *Threads* são mecanismos que permitem a desenvolvedores de algoritmos delimitar partes de seus algoritmos para serem executadas simultaneamente, dividindo-os em tarefas menores que podem ou não ser independentes, e determinando o momento em que elas podem ser executadas ou bloqueadas.

4.3 Classes de arquiteturas de paralelismo

Conforme as afirmações de Pacheco (2011), antes de paralelizar um algoritmo, deve-se analisar a arquitetura de processamento paralelo em que este será executado, e implementá-lo de maneira adequada para extrair o melhor desempenho possível, de acordo com as particularidades existentes.

Segundo Pacheco (2011) o modelo para esta classificação atualmente mais aceito é a *Taxonomia de Flynn* (Flynn, 1972). Em Flynn (1972), as arquiteturas de paralelismo mais relevantes são classificadas em *Ãsnica Instrução*, *Dados Múltiplos* e *Múltiplas Instruções, Múltiplos Dados*, que são detalhadas a seguir.

4.3.1 Ãsnica instrução, dados múltiplos

A arquitetura de *hardware* de processamento paralelo classificada como *Ãsnica Instrução, Dados Múltiplos* (*Single Instruction, Multiple Data* - SIMD) aplica paralelamente uma mesma instrução de processamento a vários containeres de dados. Pode-se imaginar um

hardware paralelo SIMD como sendo dotado por única Unidade de Controle e, no entanto, com múltiplas ALUs (Flynn, 1972). O funcionamento de uma arquitetura paralela SIMD dá-se quando a Unidade de Controle faz a transmissão para execução da mesma instrução a ser processada a diversas ALUs e estas as aplicam simultaneamente ao respectivo container de dados corrente. Este modelo de arquitetura é ilustrado na Figura 4.2

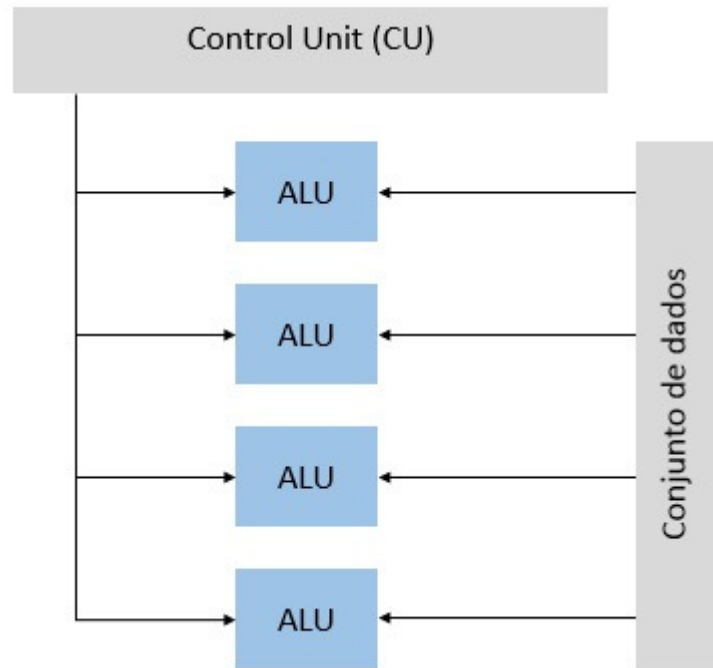


Fig. 4.2: Arquitetura *Single Instruction, Multiple Data* - SIMD.

Segundo Pacheco (2011), este tipo de arquitetura de processamento paralelo usualmente trabalha com instruções em vetores de dados. Para processamento paralelo destes vetores existem dois tipos de unidades funcionais, que são classificadas em:

- (i) Vetoriais: a instrução é executada simultaneamente em vetores que possuem operadores e operações armazenadas em seu conteúdo;
- (ii) Pipelineizadas: A instrução é executada em cada elemento do vetor que possui em seu conteúdo operadores e operações armazenadas.

4.3.2 Múltiplas instruções, dados múltiplos

Os sistemas de Múltiplas Instruções, Dados Múltiplos (*Multiple Instructions, Multiple Data* - MIMD) suportam múltiplos containeres de instruções, operando simultaneamente sobre múltiplos containeres de dados (Flynn, 1972). Em sistemas MIMD, cada uma das ALUs e Unidades de Controle, entre uma coleção de várias, pertence a uma unidade de processamento ou núcleo de processamento independente, os quais operam de forma assíncrona, cada qual a seu ritmo, a menos que o algoritmo a ser executado tenha sido escrito de forma que imponha o contrário (Pacheco, 2011).

Segundo Flynn (1972); Pacheco (2011), existem dois tipos diferentes de sistemas MIMD:

- (i) Memória compartilhada: neste tipo de sistema, cada unidade ou núcleo de processamento faz acesso a uma memória principal, que é global, ou seja, é disponibilizada

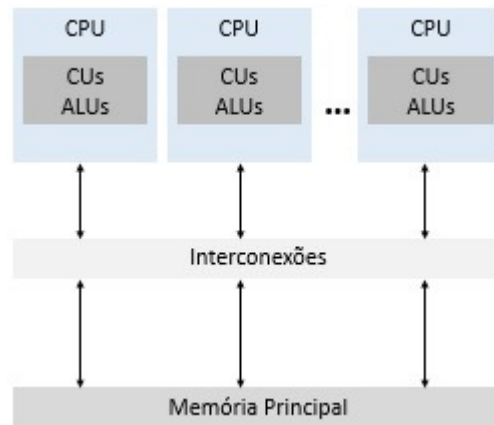


Fig. 4.3: Sistema Paralelo com Memória Compartilhada.

para todos eles, cada qual gravando e excluindo dados dos endereços dos quais necessitar. As unidades ou núcleos de processamento são conectadas à memória principal através de um conjunto de interconexões e normalmente se comunicam de forma implícita com a memória principal, podendo ou não compartilhar conjuntos de dados. Este tipo de sistema é ilustrado na Figura 4.3;

- (ii) Memória distribuída: nestes sistemas, cada unidade ou núcleo de processamento possui sua respectiva memória principal, a qual é acessada por ele de forma implícita, e somente acessada por outra unidade ou núcleo de processamento quando há um comando ou mensagem explícita para que este o faça. Todos os pares de unidade de processamento e memória disponíveis comunicam-se entre si através de uma rede de interconexões. Este tipo de sistema é largamente encontrado nos chamados *clusters*. Uma visualização deste tipo de sistema é exibida na Figura 4.4;

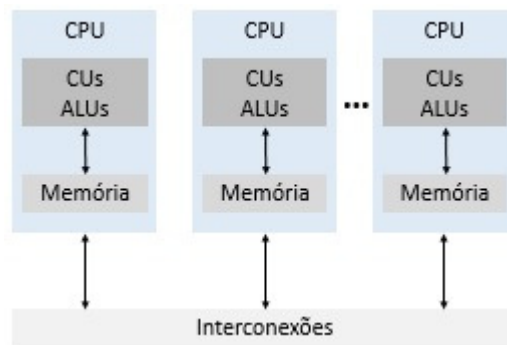


Fig. 4.4: Sistema Paralelo com Memória Distribuída

4.4 Avaliação do desempenho de algoritmos paralelos

É notório que o maior intuito de paralelizar parcialmente, ou completamente, um algoritmo é obter maior velocidade em sua execução para obtenção de um determinado resultado. Por questões diversas, seja por limitações ou vantagens de *hardware* ou devido à implementação do algoritmo, o aumento de desempenho necessário para obter um re-

sultado pode ser maior ou menor do que o esperado, à medida que se adicionam novas unidades ou núcleos de processamento para executá-lo.

Segundo [Bader et al. \(2002\)](#), atualmente são utilizadas duas unidades de contabilização do tempo de execução de um algoritmo. A primeira forma, recorrentemente chamada na literatura de *CPU TIME*, diz respeito à quantidade de ciclos que um núcleo de processamento necessita para atingir um determinado resultado ou à quantidade de instruções executadas para tal. A vantagem da utilização desta unidade de tempo é que ela descarta o tempo que uma unidade ou núcleo de processamento pode estar utilizando para outras tarefas, como abrir e fechar interconexões, atualizar informações em memória principal, atender chamados do sistema operacional, alocação ou desalocação de dados em memória virtual, sincronização dos diversos containeres de dados utilizados pelo algoritmo, bloqueio ou liberação de dados para acesso em memória, entre outros. Porém, em termos práticos, por diversas vezes, a quantidade de ciclos necessários para atingir um determinado resultados não importa, desde que o tempo cronometrado em relógio para execução do algoritmo seja reduzido, mesmo que haja super ou sub utilização do *hardware*. Neste último caso, deve-se utilizar como unidade de contabilização de tempo o tempo em relógio (cronômetro).

4.4.1 Princípio de Brent e Speedup

De acordo com o princípio de Brent, através da medida de desempenho de um algoritmo sequencial pode-se estimar o desempenho de um algoritmo paralelamente processado, por p unidades ou núcleos de processamento, com um fator de desaceleração uniforme p ([Brent, 1971](#)). Segundo [Pacheco \(2011\)](#), se, ao executar um algoritmo paralelizado, à medida em que se adicionam p núcleos de processamento, tem-se que:

$$Tempo_{paralelo} = \frac{Tempo_{serial}}{p} \quad (4.1)$$

Nesta expressão, $Tempo_{paralelo}$ indica o tempo esperado para a execução de um algoritmo processado por p núcleos de processamento e $Tempo_{serial}$ o tempo de execução do mesmo algoritmo processado por apenas um núcleo de processamento.

Constantemente, a medida de aumento de desempenho de algoritmos paralelizados tem sido dada pelo *Speedup*, o qual pode ser obtido como:

$$Speedup = \frac{Tempo_{serial}}{Tempo_{paralelo}} \quad (4.2)$$

Segundo [Bader et al. \(2002\)](#); [Pacheco \(2011\)](#), algoritmos que fazem buscas aleatórias, em um espaço de soluções de tamanho variável ou não, podem apresentar anomalias em seu *Speedup*, dado que, dependendo da disposição das soluções e da busca feita pelo algoritmo, a adição ou exclusão de núcleos de processamento gera um aumento ou redução de desempenho também aleatório. Sendo assim, estes algoritmos podem apresentar desempenho maior ou menor que o estimado pela equação 4.1. Isso não viola o princípio de Brent, dado que [Bader et al. \(2002\)](#) estima o melhor algoritmo sequencial como o melhor caso possível para os algoritmos paralelos.

Caso o *Speedup* de um algoritmo seja superior ao tempo estimado, este *Speedup* será chamado de *superlinear* e, caso contrário, será chamado de *sublinear* ([Pacheco, 2011](#)).

4.4.2 Lei de Amdahl

Gene Amdahl, na década de 1960, observou que todos os algoritmos sequenciais que são paralelizados possuem um *Speedup* limitado. Esta é a chamada *Lei de Amdahl*. Em Hill e Marty (2008); Sun e Chen (2010); Amdahl (1967), a Lei de Amdahl diz que, a partir do percentual de código paralelizado de um algoritmo, pode-se saber o máximo *Speedup* e mínimo tempo de execução que ele pode atingir. Para este cálculo, seja $perc_p$ o percentual de código paralelizado de um algoritmo e $perc_{np}$ o percentual de código serial, ambos com um valor entre 0 e 1. O tempo paralelo mínimo para execução de um algoritmo será dado por:

$$Tempo_{paralelo} = perc_p \cdot \frac{Tempo_{serial}}{p} + Tempo_{serial} \cdot perc_{serial} \quad (4.3)$$

e o máximo *Speedup* será dado por:

$$Speedup = \frac{Tempo_{serial}}{perc_p \cdot \frac{Tempo_{serial}}{p} + Tempo_{serial} \cdot perc_{serial}} \quad (4.4)$$

Por este motivo, a Lei de Amdahl permite que sejam calculados limites superiores para o *speedup* de algoritmos paralelos (Pacheco, 2011).

Algoritmos que fazem buscas aleatórias de soluções podem obter um aumento de desempenho superlinear, dado que, a maior quantidade de pontos paralelizados, disparam buscas em locais diferentes do espaço de soluções e, uma vez que sua execução finda-se ao atingir um critério de qualidade de solução, o aumento de desempenho pode ser exponencial sem violar a Lei de Amdahl.

4.5 Placas de Processamento Gráfico

A motivação inicial de aplicação de processamento paralelo em GPUs deu-se pela necessidade de aumento de desempenho no processamento de imagens. O processamento de uma imagem qualquer, de maneira não incomum, chega a milhares de megabytes de dados. Esta tarefa exige processamento massivo e, talvez recorrentemente necessário, de grandes conjuntos de dados, o que faz com que o *hardware* responsável por esta tarefa precise ser extremamente rápido e eficiente (Pacheco, 2011; Cook, 2013; Jäskeläinen et al., 2010).

Conforme dito no Capítulo 1, as GPUs atuais são dotadas de uma grande quantidade de núcleos de processamento, os quais, por sua vez, possuem diversas ALUs (Pacheco, 2011; Cook, 2013; Jäskeläinen et al., 2010). Apesar de todas as GPUs atuais fazerem-se valer de processamento paralelo SIMD, não se pode dizer que elas utilizam puramente esta arquitetura, pois cada ALU de cada *core* de uma GPU trabalha com paralelismo SIMD; no entanto, GPUs atuais possuem vários *cores* e estes podem executar instruções diferentes de maneira independente (Pacheco, 2011).

As CPUs atuais possuem memória *cache*, ao contrário das GPUs convencionais. Enquanto *threads* ociosas ficam bloqueadas aguardando por dados, outras são escalonadas para execução, o que faz a latência ser imperceptível. A Figura 4.5 ilustra as diferenças entre CPUs convencionais e GPUs.

A velocidade de processamento de dados em GPUs a tempos desperta o interesse para aplicações em que se detecta necessidade de processamento paralelo na computação cientí-

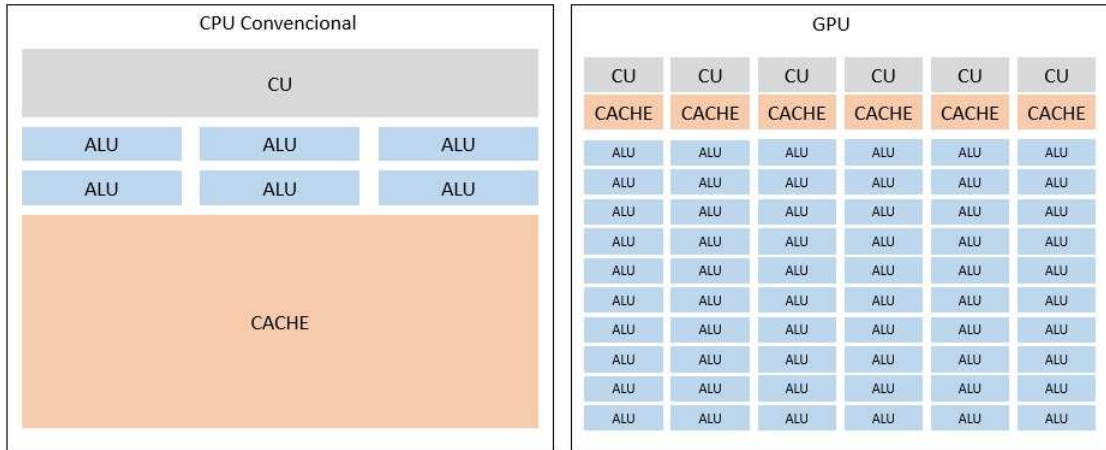


Fig. 4.5: Diferenças entre as estruturas da CPU e GPU.

fica e na indústria, o que tem acelerado o desenvolvimento de *frameworks* específicos para trabalho com este tipo de *hardware* (Cook, 2013; Jäskeläinen et al., 2010).

Classifica-se o processamento de algoritmos codificados para serem executados em GPU como heterogêneos (Milanés et al., 2013). Atuando como um co-processador da CPU principal (*host*), as tarefas são enviadas desta para a GPU (*device*), com chamadas de funções, em sintaxe específica, para iniciar um *kernel*. Um *kernel* contém a configuração para que seja criado um agrupamento de *threads*, chamado de bloco (*block*) e estas executam simultaneamente a mesma instrução. Os *blocks* em um *kernel* estarão dispostos em uma grade (*grid*). O tratamento de divergências internas em um *block* é feito por *threads* de bloqueio, que avaliam uma estrutura de decisão “se” como falsa, até que outras *threads* do mesmo *block* terminem sua execução (Milanés et al., 2013). A Figura 4.6 ilustra a arquitetura lógica dos *kernels* e blocos em uma GPU.

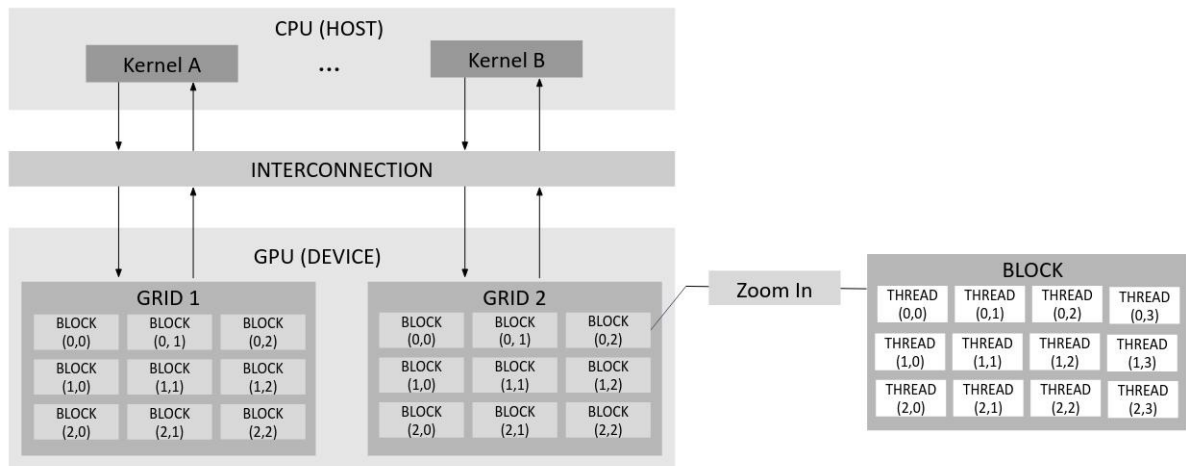


Fig. 4.6: Estrutura lógica de blocos e *threads* em GPUs.

A execução de algoritmos em um *device* precisa da interação *device-host*:

- (i) *host* aloca no *device* a quantidade de memória necessária;
- (ii) *host* copia os dados para a área de memória reservada no *device*;

- (iii) *device* inicia a configuração de execução *kernel* com a quantidade de blocos e *threads* em cada um;
- (iv) *threads* armazenam os dados gerados;
- (v) *device* envia os dados para o host;
- (vi) memória do *device* é desalocada.

Nas GPUs as diversas *threads* para processamento são agrupadas em blocos, os quais, por sua vez, são agrupados formando, assim, uma grade. Conforme pode ser observado na Figura 4.6, cada grade, bloco e *thread* é identificado por um ID atribuído pela API. Este ID é utilizado pelo programador para estabelecer as atribuições de cada *thread*, seus momentos de execução e os respectivos dados. Cada SM pode executar um conjunto de tamanho fixo de *threads* chamado *warp* por vez, os quais atualmente possuem 32 *threads*. A camada de interconexão permite que após e/ou durante a execução do *kernel* os SMs da GPU se comuniquem com o processador *host*.

A disponibilização de dados acontece em uma estrutura hierárquica de memória, na qual cada nível possui um tempo e restrição (Nvidia, 2018), relevantes para esse trabalho são:

- (i) *Private* ou *Register memory*: mais veloz e com acesso privado a cada *thread*, existente normalmente em menor quantidade nos *devices*;
- (ii) *Shared Memory*: segundo mais veloz, compartilhado entre as *threads* de um bloco e existente em segunda menor quantidade nos *devices*;
- (iii) *Global Memory*: mais lento entre os níveis, é compartilhado entre todos os blocos de uma grade, mas disponível em maior quantidade para utilização nos *devices*.

Dada a capacidade reduzida de cada um de seus núcleos de processamento isolados, quando comparado às *Central Processing Units* (CPUs) da atualidade, a execução paralela de tarefas precisa ser devidamente adaptada para GPUs, respeitando suas especificidades, até mesmo aquelas existentes entre placas de diferentes fabricantes para, então assim, poder acelerar a execução de algoritmos, como visto em Djinevski et al. (2013), Otte e Correll (2013) e Ichnowski e Alterovitz (2012).

O aumento de velocidade de execução se justifica uma vez que, dependendo da operação do algoritmo a ter sua complexidade de tempo analisada, ela será completa ou parcialmente dividida por uma constante C , tal que $C = \text{total de unidades de processamento paralelo disponíveis}$, o que, apesar de manter na maioria das vezes o grau de complexidade polinomial inalterado, na prática reduz o tempo de execução.

4.6 GPUs Nvidia CUDA©

Na vanguarda e em constante aperfeiçoamento, as GPUs Nvidia dotadas da tecnologia CUDA© têm se popularizado entre diversos segmentos como pesquisadores, jogadores, designers, engenheiros, usuários de computadores e outros, por, dentre outros vários motivos, o alto desempenho, a facilidade de uso de sua API e relação custo benefício.

Nas subseções seguintes são detalhados alguns dos pontos chave da API CUDA utilizados neste trabalho. A Subseção 4.6.1 expõe diferentes tipos de configura a execução

Algorithm 6: KernelMultiply

```

1 Kernel KernelMultiply(a, b)
   Input : arrays a e b de tamanho x, preenchidos com valores aleatórios
   Output: array a com todos os seus índices multiplicados pelos respectivos índices de b
2   private<var> tid ← getThreadId();
3   private or shared<var> bid ← getBlockId();
4   if tid < x then
5       início de região atômica
6       a[bid] ← a[bid] + a[tid] * b[bid];
7       fim de região atômica
8   end
9   return a;
10 end

```

(alocação de *threads*), paralelismo dinâmico e *cooperative groups*. A Subseção 4.6.2 exibe os *kernels* do tipo *monolithic* e *grid stride loops*. Por fim, a Subseção 4.6.3 descreve as condições de corrida e barreiras de sincronização.

4.6.1 Distribuição de *threads* por *grid*, paralelismo dinâmico e *cooperative groups*

O modo como se distribuem as *threads* disponíveis para execução de *kernels* e funções CUDA afeta o desempenho, dado que cada qual apresenta vantagens e desvantagens. Assim sendo, três diferentes formas, classificadas de acordo com a divisão de *threads*, são utilizadas.

Agrupar *threads* e seus respectivos blocos estaticamente de forma a cooperarem entre si, conforme é visto nos Algoritmos 9 e 10, oferece a vantagem de não serem necessárias alterações na grade de processamento da GPU durante a execução dos algoritmos, tarefa que, além do aguardo do tempo para readequação da grade, pode necessitar também o término do trabalho das *threads* que virão a ser redistribuídas posteriormente. A desvantagem dessa forma de processamento dá-se nas barreiras de sincronização, as quais precisam ser estabelecidas para todas as *threads* do dispositivo em parte das vezes ou na alocação posterior de *threads* desnecessárias para o resultado do algoritmo.

O paralelismo dinâmico (Nvidia, 2018; Jones, 2012; Wang e Yalamanchili, 2014; Li et al., 2015) permite que *threads* construam subgrades de processamento, de forma a processar de maneira aninhada, e com a quantidade adequada de *threads*, diferentes trechos de códigos. Essa opção reduz atrasos em barreiras de sincronização, uma vez que somente *threads* úteis serão alocadas para processamento. Caso haja necessidade de alterações no tamanho das subgrades no decorrer do algoritmo, se não existirem *threads* disponíveis imediatamente, ocorre, então, a espera pelo término de processamento antes da readequação. Dependendo de tal tempo, há possibilidade de ser mais eficiente aumentar a carga de trabalho por *thread* e reduzir as barreiras de sincronização, o que nem sempre é possível. Outra limitação pode ser a quantidade de blocos e *threads* imposta pelo *hardware* para cada subgrade. Exemplos de utilização deste tipo de chamada a *kernels* são mostrados nos Algoritmos 6 e 7.

O Algoritmo 6 recebe, como parâmetros, os arrays *a* e *b*, ambos de tamanho *x*, e tem, por finalidade, multiplicar cada um dos elementos de *a* por todos os elementos em *b*. O índice do elemento em *a* a receber o resultado das multiplicações é dado pela fun-

ção `getBlockId()` e o índice de b pela função `cgetThreadId()`.

Algorithm 7: Exemplo de Dynamic Parallel Kernel Call

```

1 Kernel DynamicParallelKernelCall( $x$ )
   Input: variable  $x$  com o mesmo tamanho dos arrays de valores aleatórios  $a$  e  $b$ 
2   private<var> tid  $\leftarrow$  getThreadId();
3   if tid <  $x$  then
4     | Call KernelMultiply<<< 1,  $x$  >>> ( $a$ ,  $b$ );
5   end
6   return  $a$ ;
7 end

```

Já o *Kernel* do Algoritmo 7, após recuperar o número da *thread* corrente, verifica se ele é menor que o tamanho dos vetores e, em caso afirmativo, a *thread* faz a chamada do *Kernel* 6 para multiplicação dos valores dos *arrays*, em que observa-se que cada *thread* cria um bloco com x *threads* para isso.

Cooperative groups (Nvidia, 2018; Harris e Perelygin, 2017) permitem unir grades ou blocos em grupos ou dividi-los em subblocos menores para serem manipulados em conjunto. Cada grupo pode ser sincronizado isoladamente; no entanto, pode ser demandada a espera pelo fim de trabalho de outros no decorrer da execução do algoritmo, caso seja necessário reajustar as dimensões. Um exemplo é exibido no Algoritmo 8.

Algorithm 8: Kernel Multiply Cooperative

```

1 Kernel KernelMultiplyCooperative( $cg, i$ )
   Input : Cooperative threads group  $cg$ , contendo o índice em  $a$  para receber o resultado,
           preenchido com matrizes de valores aleatórios  $a$  e  $b$  de tamanho  $x$ 
   Output: array  $a$  em que cada índice é multiplicado por todos os valores do array  $b$ 
2   private<var> tid  $\leftarrow$  getThreadRank( $cg$ );
3   if tid <  $x$  then
4     | begin atomic region
5     |  $a[i] \leftarrow a[i] * b[tid]$ ;
6     | end atomic region
7   end
8   barrier( $cg$ );
9   return  $a$ ;
10 end

```

O *kernel* do Algoritmo 8 armazena na variável *tid* o número de identificação da *thread* em seu respectivo grupo cg . Feito isso, caso o valor de *tid* seja menor que a quantidade de elementos em a e b , o valor da posição de índice i do vetor a é incrementado através da multiplicação de seus valores por todos os valores em b . Esta função pode ser executada simultaneamente por diversas *thread groups* e cada instância de execução ser sincronizada independentemente das outras, conforme é visto no final do algoritmo.

4.6.2 Monolithic Kernels vs Gride-Stride Loops

Processar simultaneamente diversas posições de memória é uma das tarefas mais comuns em *kernels* de GPUs. No entanto, para utilização de tal recurso, deve-se atentar que as atuais instâncias do problema ultrapassam a quantidade de *threads* normalmente

disponíveis, mesmo elas existindo em número massivo nos *devices*. Assim sendo, esta Seção mostra a forma utilizada para contornar tal limitação.

Monolithic kernels são utilizados quando há *threads* suficientes para cobrir simultaneamente todas as posições de memória que demandam processamento. Em caso contrário, pode-se fazer uso de um *grid-stride loop*, em que as posições dos vetores são distribuídas entre as *threads* disponíveis na grade e cada qual processa, a cada iteração de um *loop*, suas respectivas posições de memória (Harris, 2013). Os Algoritmos 9 e 10 exibem exemplos desses dois tipos de *kernels*.

Algorithm 9: Monolithic Kernel Example

```

1 Kernel MonolithicKernel(a,b)
   Input : arrays a e b de tamanho x preenchidos com valores aleatórios
   Output: array a o qual todos os elementos foram multiplicados pelos respectivos elementos de
           b
2   private<var> tid ← getBlockId() * getBlockSize() + getThreadId();
3   if tid < x then
4     | a[tid] ← a[tid] * b[tid];
5   end
6   return a;
7 end

```

O *monolithic kernel* do Algoritmo 9 tem o objetivo de multiplicar cada um dos respectivos elementos dos vetores armazenados em escopo global *a* e *b*, ambos de tamanho *x*. Inicialmente, cada *thread* cria uma variável privada *tid*, a qual tem a si atribuído o ID exclusivo da *thread* na grade. Para tal, através das funções `getBlockId()`, `getBlockSize()` e `getThreadId()`, é feita a multiplicação do número de identificação do bloco por seu tamanho e somado o ID da *thread* no bloco. Posteriormente, caso o número da *thread* seja menor que o tamanho dos vetores, ela multiplicará a sua respectiva posição de *a* por *b*.

Algorithm 10: Grid-Stride Loop Kernel Example

```

1 Kernel GridStrideLoopKernel(a,b)
   Input : arrays a e b de tamanho x preenchidos com valores aleatórios
   Output: array a o qual todos os índices são multiplicados pelos respectivos índices de b
2   private<var> tid ← getBlockId() * getBlockSize() + getThreadId();
3   private or shared<var> stride ← getGridSize();
4   for idx ← tid; idx < x; idx ← idx + stride do
5     | a[idx] ← a[idx] * b[idx];
6   end
7   return a;
8 end

```

O *grid-stride loop kernel* do Algoritmo 10 tem o mesmo objetivo do *kernel* do Algoritmo 9; no entanto, a sua execução é mais flexível no que tange à quantidade de *threads* disponíveis. Caso haja a mesma ou maior quantidade de *threads* necessárias para cobrir os vetores *a* e *b*, apenas uma iteração do *loop* entre as linhas 4 - 6 será executada; caso contrário, a partir do valor da variável *stride*, a qual contém as dimensões da grade de processamento disponível, o *loop* será executado quantas vezes forem necessárias para cobrir os dois vetores.

Em programação CUDA, é comum funções terem o seu valor (ou vetor) de retorno passado por referência nos parâmetros em substituição ao comando *return*, o qual é comum

à programação sequencial. Isso se dá pelo fato da possibilidade de existência de *race conditions* ao retornar um valor via uma função `device` ou `kernel`. A título de exemplo, caso N threads processassem uma função e o escopo externo a esta última esperasse um retorno, N diferentes resultados seriam apresentados, dificultando o tratamento de tais dados.

4.6.3 Race Conditions

É chamado de *race condition* a situação em que operações de escritas e leituras são efetuadas simultaneamente por várias threads em um mesmo endereço de memória compartilhada ou global (Nvidia, 2018). Três formas comuns de contorná-las, baseadas em sincronização, são:

- (i) *Operações atômicas* garantem que apenas uma thread por vez faça a leitura, modificação e retorno do valor em uma posição de memória sem a interferência de outras threads, as quais devem aguardar seu momento;
- (ii) *Sincronization barriers* permitem que threads especificadas pausem seu trabalho em um determinado ponto até que outras as alcancem; e
- (iii) *Semáforos (mutex)* são especificados pelo programador, criando regras para threads específicas sincronizarem suas operações.

É fundamental que algoritmos paralelos sejam projetados de forma a impedir *race conditions*. Por outro lado, operações de sincronização podem tornar a execução demasiadamente lenta.

Capítulo 5

Metaheurísticas Paralelas

Este capítulo se organiza da seguinte forma. A Seção 5.1 descreve as topologias de processamento paralelo para metaheurísticas. A Seção 5.2 apresenta a caracterização das metaheurísticas paralelas. As Seções 5.3, 5.4, 5.5 e 5.6 descrevem, respectivamente, as estratégias de paralelização em baixo nível, em decomposição de dados, em buscas múltiplas independentes e em buscas múltiplas cooperativas.

5.1 Topologias de Processamento Paralelo

A união de técnicas de processamento paralelo e metaheurísticas é utilizada na literatura para aprimoramento das soluções em problemas de otimização combinatória, segundo Talbi (2009), por diversos motivos, como:

- (i) Aumentar a velocidade na busca por soluções satisfatórias, através da redução do tempo de processamento necessário;
- (ii) Através da ampliação da busca no espaço de soluções, obter soluções de melhor qualidade;
- (iii) Aumentar a robustez das metaheurísticas, dando a elas mais opções de configuração e, conseqüentemente, melhores soluções, não apenas em um problema específico, mas em diversos diferentes problemas de otimização combinatória, quando aplicadas;
- (iv) Resolução de problemas de maiores dimensões e mais complexos, para os quais algoritmos sequenciais têm dificuldades em determinar uma solução.

Estes diversos objetivos são alcançados, basicamente, dividindo o processamento necessário para a metaheurística entre os diversos núcleos de processamento disponíveis. Há uma ampla literatura disponível sobre o tema, destacando-se, entre outros, os trabalhos de Crainic e Toulouse (2010); Cung et al. (2002); Verhoeven e Aarts (1995); Crainic e Hail (2005); Alba (2005).

Segundo Crainic et al. (2014), em linhas gerais, um dos maiores problemas para paralelizar metaheurísticas é a *sincronização*, que permite que diferentes partes da metaheurística paralela troquem informações entre si. A sincronização pode requerer que tarefas da metaheurística paralela sejam paralisadas por um determinado momento, acarretando, ao final, um aumento no tempo de execução necessário para que seja concluída. Os principais

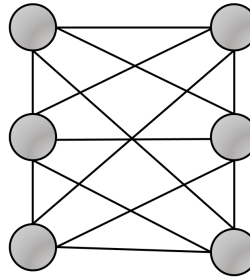


Fig. 5.1: Topologia de processamento paralelo “Completamente Conectada”.

recursos utilizados neste caso são os *semáforos* (Pacheco, 2011) e as *barreiras de sincronização* (Pacheco, 2011). Diz-se em Crainic et al. (2014) que as formas de processamento atribuídas às metaheurísticas paralelas atualmente podem ser classificadas basicamente por três topologias, as quais organizam de forma consistente a sincronização entre os núcleos de processamento e memória(s). São elas:

- (i) Completamente conectada: caracteriza-se por permitir que todos as unidades ou núcleos de processamento se comuniquem de forma direta entre si próprios. Este tipo de topologia, representado na Figura 5.1 é difícil de ser implementada pelos problemas em manter a comunicação entre os núcleos de processamento controlados à medida que a quantidade destes disponíveis aumenta;
- (ii) Estrela: nesta topologia, representada na Figura 5.2, existe um núcleo de processamento designado para controlar a comunicação entre os demais, sendo que todas as solicitações de comunicação devem ser feitas a este.

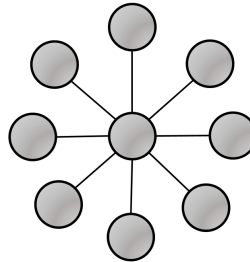


Fig. 5.2: Topologia de processamento paralelo “Estrela”.

- (iii) Anel de Processamento Unidirecional: os dados ou tarefas de processamento são enviados a cada núcleo de processamento por vez. A comunicação entre eles não é centralizada, permitindo fácil implementação. Está representada na Figura 5.3.

Inicialmente, para que seja realizada a paralelização de uma metaheurística, devem ser analisadas as características do problema envolvido, de suas instâncias, e do algoritmo que a implementa, para que possam ser identificadas as partes específicas que devem ser manipuladas por cada núcleo de processamento (Crainic, 2015). Após estas análises, pode ser aplicado o paralelismo:

- (i) nos dados dos problemas (paralelismo de dados) (Hillis e Steele, 1986), dividindo o espaço de busca (domínio) em sub-espacos (sub-domínios) e atribuindo *thread(s)* a cada, para que a metaheurística aplique o seu(s) processo(s) de busca(s) neste(s);

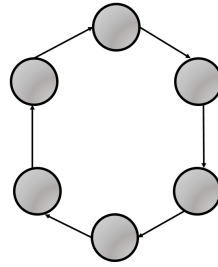


Fig. 5.3: Topologia de processamento paralelo “Anel de Processamento Unidirecional”.

- (ii) em tarefas executadas pelo algoritmo da metaheurística (paralelismo de tarefas) (Quinn, 2003), determinando quais partes de um processo deve(m) ser executados por cada *thread*.

5.2 Caracterização das Metaheurísticas Paralelas

Os trabalhos de Verhoeven e Aarts (1995); Cung et al. (2002) classificaram o paralelismo aplicado a metaheurísticas em dois tipos:

- (i) Caminhada Ãsnica (*single walk*): todos os núcleos de processamento executam iterações de um mesmo processo de busca (o qual pode ser instanciado por cada *thread* ou atuar de forma cooperativa), em um conjunto de soluções inicialmente geradas ou em uma única solução;
- (ii) Caminhada Múltipla (*multiple walk*): todos os núcleos de processamento façam iterações com diferentes processos de busca, o que ocasionalmente gera processos concorrentes. Para tanto, inicialmente devem ser gerados diferentes conjuntos de soluções ou uma única solução, para que seja(m) distribuída(s) entre os processos, e definido sobre qual cada *thread* deve iterar.

Crainic et al. (1997); Crainic e Hail (2005); Crainic (2015) propõem uma classificação mais refinada, baseando-se nos mesmos conceitos dos tipos definidos em Verhoeven e Aarts (1995); Cung et al. (2002), mas fazendo com que sejam considerados os diferentes aspectos possíveis para as buscas por soluções e comunicação. Para tanto, a classificação de metaheurísticas paralelizadas propostas em Crainic et al. (1997); Crainic e Hail (2005) considera que são necessárias três dimensões para esta classificação:

- (i) Cardinalidade do Controle de Busca;
- (ii) Controle de Busca e Comunicação; e
- (iii) Diferenciação de Busca.

A união das três dimensões para a classificação de Crainic et al. (1997); Crainic e Hail (2005) formam uma taxonomia, ilustrada na Figura 5.4. As dimensões da taxonomia são explicadas nas seções 5.2.1, 5.2.2 e 5.2.3.

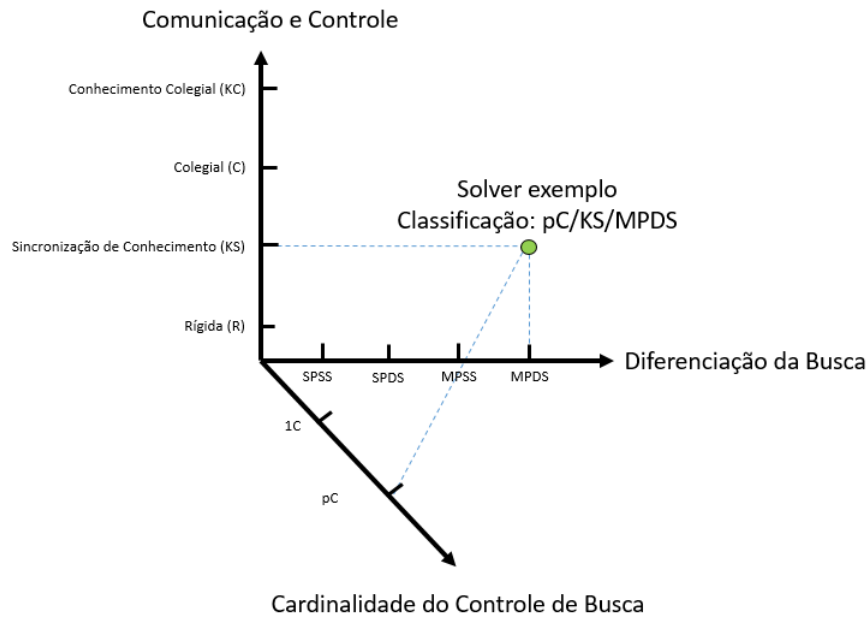


Fig. 5.4: Taxonomia para classificação de metaheurísticas paralelas (Crainic et al., 1997; Crainic e Hail, 2005)

5.2.1 Dimensão 1 - Cardinalidade do Controle de Busca

A dimensão 1 da classificação elaborada por Crainic et al. (1997) chama-se Cardinalidade do Controle de Busca (*Search Control Cardinality*) e classifica o controle global de buscas no espaço de soluções, ou seja, se a busca será feita por um único processo (*1-Control* (1C)) ou por vários processos diferentes (*p-Control* (pC)).

A classificação 1C na dimensão 1, em termos de paralelização, indica que um algoritmo delegará tarefas a outros núcleos de processamento. Um dos núcleos de processamento será responsável por distribuir e posteriormente coletar e unir o trabalho resultante das tarefas paralelamente executadas nos demais núcleos de processamento. Por este motivo, este núcleo de processamento (ou o processo que ele executa) é chamado de mestre (*master*), enquanto os demais são chamadas de escravos (*slaves*).

Ao contrário da classificação 1C, a classificação pC, em que $p > 1$, indica que diferentes formas de busca serão executadas, por diferentes núcleos de processamento. Inicialmente, cada busca é carregada em um processo contendo um método de busca que executará até que algum critério (por exemplo tempo, quantidade de iterações, etc) seja atingido, fazendo, antes disso, a sincronização dos resultados obtidos durante sua execução, quando houver determinação para isso.

5.2.2 Dimensão 2 - Controle de Busca e Comunicação

Na dimensão 2 da classificação criada por Crainic et al. (1997), chamada Controle de Busca e Comunicação (*Search Control and Communication*), exhibe-se como e se será feita a sincronização de informações entre os processos paralelos da metaheurística, para controlá-los ou guiá-los no(s) respectivo(s) espaço(s) de busca(s). Pode-se definir para que isso ocorra das seguintes formas:

- (i) síncrona: sincronização de todos os processos simultaneamente; e

- (ii) assíncrona: sincronização dos processos em momentos diferentes.

As comunicações síncrona e assíncrona são sub-divididas em:

- (i) Rígida (*Rigid* - RS): quando um processo atinge algum critério, outros processos devem ser paralisados e ser realizada a sincronização imediata de suas informações;
- (ii) Sincronização de Conhecimento (*Knowledge Sincronization* - KS): O(os) processo(s) *slaves* devem atingir algum critério ou serem requisitados pelo processo *master*, sendo então paralisados. Em seguida, o processo *master* executa a sincronização de suas informações e atribui aos processos *slaves* novas tarefas;
- (iii) Colegial (*Collegial* - C): processos independentes fazem buscas em seus respectivos sub-domínios (sub-espacos de buscas). Quando um processo encontra uma informação que se enquadra em algum critério, ele a deposita nos sub-domínios dos outros ou de alguns dos outros processos. Estes utilizam ou não a informação depositada, dependendo das condições de busca que foram determinadas;
- (iv) Conhecimento Colegial (*Knowledge Collegial* - KC): idem ao tipo *Collegial*, no entanto, antes de implantar a informação, ela primeiramente é avaliada sob sua utilidade para o sub-domínio.

5.2.3 Dimensão 3 - Diferenciação da Busca

A dimensão 3 da classificação criada por [Crainic et al. \(1997\)](#), chamada de Diferenciação da Busca (*Search Differentiation*), corresponde à classificação criada em [Du e Pardalos \(1993\)](#), conforme posto pelo próprio autor.

Esta dimensão da classificação tem como objetivo descrever a forma de inicialização das soluções utilizadas e as estratégias de busca do(s) processo(s) da metaheurística paralelizada, ou seja, se o(s) processo(s) utiliza(m) a mesma solução ou não e se este(s) utiliza(m) ou não a mesma estratégia de busca. Dada uma configuração desta dimensão, necessita-se, então, especificar se ela utiliza uma ou mais soluções (pontos) do espaço de busca, e uma ou mais estratégias de busca. Esta dimensão possui quatro configurações possíveis:

- (i) Ponto Único Estratégia Única (*Single Point, Single Strategy* - SPSS): a partir de um ponto único ou região, selecionado no espaço de busca, os processos o utilizam para iniciar sua busca;
- (ii) Ponto Único Estratégias Diferentes (do inglês - *Single Point, Different Strategies* - SPDS): várias estratégias de buscas diferentes são utilizadas pelos processos; no entanto, todas partindo do mesmo ponto ou região inicial;
- (iii) Pontos Múltiplos Estratégia Única (*Multiple Points Single Strategy* - MPSS): vários processos realizam a mesma estratégia de busca; no entanto, cada um parte de um ponto ou região diferente;
- (iv) Pontos Múltiplos Estratégias Diferentes (*Multiple Points Different Strategies* - MPDS): cada processo possui uma estratégia de busca diferente, cada qual partindo de um ponto ou região inicial diferente.

5.3 Estratégias de Paralelização em Baixo Nível

A estratégia de paralelização dos algoritmos das metaheurísticas, chamada de Estratégia de Paralelização de Baixo Nível (*Low Level Parallelization Strategy*) (Crainic et al., 1997), caracteriza-se por fazer apenas modificações internas, normalmente com o intuito de paralelizar estruturas de repetição, fazendo com que ocorra, assim, um aumento da velocidade de busca. Este tipo de estratégia não faz modificações na lógica dos algoritmos e, na maioria das vezes, metaheurísticas paralelizadas por esta estratégia são classificadas como *1C/RS/SPSS*.

Segundo Crainic (2015), em implementações de tal estratégia, frequentemente um algoritmo da classe *1C* cria um processo *master*, que é executado de forma sequencial, com o intuito de gerar uma solução ou população. Em sequência, este processo envia tarefas, que serão realizadas por processos *slaves*, para serem alocadas em diferentes unidades ou núcleos de processamento. Quando um processo *slave* atinge um determinado critério, ele comunica-se com o processo *master*, para que seja feita a sincronização no sentido *slave* $\rightarrow$ *master* ou vice versa. Neste tipo de estratégia, não há a comunicação direta entre os processos *slave*.

Na parte de busca das metaheurísticas, a paralelização, quando utiliza-se desta estratégia, é também classificada normalmente como *1C/RS/SPSS*. Isso é feito com o processo *master* agrupando soluções vizinhas e, posteriormente, as enviando a processos *slaves*. Os processos *slaves*, quando atingem um critério, enviam esta informação ao processo *master*, para que este tome uma decisão e, se necessário, comunique aos processos *slaves*. O processo *master* aguarda que o processamento de cada processo *slave* seja concluído para que a busca seja finalizada. Nos trabalhos Porto e Ribeiro (1995, 1996) técnicas de Buscas Tabu são realizadas e paralelizadas por esta estratégia para o Problema de Agendamento de Tarefas Dependentes em Processadores Heterogêneos sob Restrições Precedentes (*Task Scheduling on Heterogenous Processors Under Precedence Constraints*). Em Garcia et al. (1994), uma implementação também de Busca Tabu é realizada para o Problema de Roteamento de Veículos com Janelas de Tempo (*Vehicle Routing Problem With Time Windows - VRPTW*).

A metaheurística Colônia de Formigas (*Ant Colony*) (Dorigo, 1992) utiliza esta estratégia como uma das suas várias formas de paralelização. Para tal, o processo *master* cria tarefas para as formigas, as quais podem integrar todas uma única ou diversas colônias, e repassadas a processos *slaves*. Os processos *slaves*, utilizando-se das informações das formigas, executam uma heurística construtiva e fazem ou não uma busca local. As formigas retornam dos processos *slaves* para o processo *master*, o qual faz uma atualização nas informações da matriz de feromônios e a associa a cada formiga. Em Bullnheimer et al. (1998); Randall e Lewis (2002); Talbi et al. (1999) esta e outras estratégias de paralelização para esta metaheurística são apresentadas. Já (Doerner et al., 2004; Rahoual et al., 2002) aplicam esta estratégia de paralelização de Colônia de Formigas e diversas outras ao Problema de Roteamento de Veículos Capacitados e Cobertura de Conjuntos, respectivamente.

As metaheurísticas de Busca Dispersa (*Scatter Search*) (Glover, 1977) e Reconexão por Caminhos (*Path Relinking*) (Glover, 1997), através de um número limitado de soluções elitizadas, implementam uma estratégia evolucionária. Em Garcia-López et al. (2003); López et al. (2005, 2006) utilizam-se desta estratégia, também classificada como *1C/RS/SPSS*, para solucionar o problema de *p*-Medianas e de Seleção de Recursos, fazendo uma paralelização do processo de busca local na população, em que os processos *slaves* são responsáveis

por executar as iterações necessárias e, posteriormente, retornar a melhor solução ao processo *master*.

Segundo Crainic (2015), os algoritmos genéticos (Holland, 1975) podem ter todos os seus diversos operadores paralelizados por esta estratégia, sendo classificados como *1C/RS/SPSS*. Na literatura, segundo Crainic (2015), frequentemente é utilizada a estratégia de paralelização em baixo nível, em algoritmos genéticos, para avaliação da função objetivo. A evolução da população, segundo Crainic (2015), também é muito explorada na literatura, fazendo dos algoritmos genéticos uma metaheurística amplamente paralelizada por esta estratégia. Porém, quando é feito a partir desta, o único ganho relatado é o de velocidade.

Quando os procedimentos de avaliação de populações ou mesmo indivíduos possuem um custo dispendioso computacionalmente, e uma diminuição considerável no tempo de processamento é necessária, esta estratégia é particularmente atrativa. Por exemplo, em Chakrapani e Skorin-Kapov (1992, 1993); Taillard (1991) é utilizada esta estratégia de paralisação para Buscas Tabu para o Problema da Atribuição Quadrática (*Quadratic Assignment Problem*). Em Porto et al. (2000); Davidović e Crainic (2015) esta estratégia de paralelização é utilizada para solucionar variações do Problema de Agendamento de Tarefas (*Task Scheduling*).

Estratégias de paralelização de metaheurísticas em baixo nível podem também ser combinadas com outras estratégias de paralelização, fazendo um uso ainda mais eficiente de diversas unidades ou núcleos de processamento, quando estas estão disponíveis. Rego e Roucairol (1996) utilizaram-se de uma combinação de estratégias de paralelização de metaheurísticas para o Problema de Roteamento de Veículos (*Vehicle Routing Problem - VRP*), usando paralelismos em baixo nível na avaliação de indivíduos, combinada com uma busca local múltipla independente.

5.4 Estratégias de Paralelização em Decomposição de Dados

As Estratégias de Paralelização em Decomposição de Dados advém do conceito de dividir o espaço de busca ou domínio em sub-espacos menores para, então, aplicar metaheurísticas independentes em cada um deles. Após esta etapa, coleta-se soluções parciais, e, com elas, constrói-se ou verifica-se uma solução completa. Estas estratégias permitem que sejam aplicadas diferentes tipos de divisões do espaço de busca do problema, além de diferentes formas de comunicação entre eles, o que pode gerar enormes ganhos tanto de desempenho, quanto de qualidade de solução (Crainic, 2015).

Quando se opta pela utilização desta estratégia de paralelização de metaheurísticas, deve-se planejar a forma de divisão e comunicação entre os sub-espacos ou sub-domínios. Isso pode ser feito de duas maneiras. A primeira é definindo se os sub-espacos ou sub-domínios deverão ou não ser independentes. A segunda maneira é definindo se os sub-espacos ou sub-domínios terão regiões em comum e se soluções ocasionalmente iguais encontradas poderão ser ou não sobreescritas. A forma de troca de informações entre os sub-espacos ou sub-domínios deve ser considerada, pois, caso a divisão dos sub-espacos ou sub-domínios seja por partes do problema, estes, em um dado momento, devem construir uma solução completa do problema. Caso a opção pela divisão dos sub-espacos ou sub-domínios seja feita considerando-se o espaço de busca apenas, deve-se definir previamente

se eles irão trocar suas soluções após algum determinado critério ou não.

Esta estratégia, na maior parte das vezes, é classificada, na literatura, como $1C/RS/MPSS$ ou $1C/RS/MPDS$, e é implementada através de processos *master* e *slaves*. O processo *master* é responsável por delimitar os sub-espacos ou sub-domínios, sincronizar as informações coletadas, reconstruir soluções, modificar decomposições, determinar critérios de parada e atribuir as tarefas de busca para os processos *slaves*. Os processos *slaves*, por sua vez, podem ser independentes ou concorrentes, e executam suas buscas nos respectivos sub-espacos ou sub-domínios. Em [Fiechter \(1994\)](#); [Laganière e Mitiche \(1995\)](#); [Gendreau et al. \(2001\)](#) foi utilizada esta estratégia de paralelização em Buscas Tabu para solucionar grandes instâncias do Problema do Caixeiro Viajante, do Problema de Renderização de Imagens com Filtros Robustos e do Problema de Realocação de Ambulâncias em Tempo Real, respectivamente.

Segundo [Crainic \(2015\)](#), ao se utilizar desta estratégia, no entanto, com metaheurísticas que são classificadas como $pC/KC/MPSS$ ou $pC/KC/MPDS$, a decomposição do espaco de busca ou domínio é feita de forma colegial. Normalmente, ao se iniciar, esta metaheurística irá criar diversos processos independentes e concorrentes, os quais decidirão, por si próprios, qual o seu sub-espaco ou sub-domínio. Estes processos independentes podem ou não se comunicar, para fins de troca de informações, que devem acontecer quando algum critério for atingido por todos eles. No trabalho [Taillard \(1993\)](#), esta estratégia é utilizada com Buscas Tabu independentes, que sincronizam suas informações após um determinado número de iterações de cada um. Em [Doerner et al. \(2005\)](#), os autores criam colônias de formigas independentes, que atualizam uma matriz de feromônios global após atingir um determinado critério. Esta implementação é uma variação da metaheurística paralelizada de colônias de formigas é apresentada em [Doerner et al. \(2004\)](#).

5.5 Estratégias de Paralelização de Buscas Múltiplas Independentes

Segundo [Crainic \(2015\)](#), a estratégia de paralelização de buscas múltiplas independentes apresenta bons resultados, mesmo estando entre as primeiras estratégias de paralelização que foram disponibilizadas pela literatura.

Esta estratégia de paralelização de metaheurísticas é comumente classificção como pC/RS , tendo sua terceira dimensão de classificação em $SPSS$, $SPDS$, $MPSS$ ou $MPDS$, encontrada atualmente na literatura. Normalmente, metaheurísticas que se utilizam desta estratégia de paralelização iniciam com diversos processos, em que cada qual faz uma busca independente em todo o espaco de busca ou domínio. Após o término de execução de todas as buscas, estes processos sincronizam suas informações.

[Battiti e Tecchiolli \(1992\)](#) paralelizaram com esta estratégia Buscas Tabu e Algoritmos Genéticos para problemas de otimização combinatória em geral. [ten Eikelder et al. \(1999\)](#); [Taillard \(1994\)](#) utilizaram esta estratégia de paralelização nas buscas locais de suas heurísticas construtivas para solucionar o *Job Shop Scheduling Problem*. [Stützle \(1998\)](#) fez a paralelização da metaheurística de Colônia de Formigas utilizando esta e outras estratégias, respectivamente.

5.6 Estratégias de Paralelização de Buscas Múltiplas Cooperativas

As estratégias de paralelização de metaheurísticas baseadas em buscas múltiplas cooperativas caracterizam-se por permitirem que diferentes espaços de busca ou domínios compartilhem informações entre si. Para que ocorra a troca de informações, são necessárias técnicas de implementação que permitam que os metaheurísticas que atuam sobre um determinado espaço de busca compartilhem informações com metaheurísticas alocados em outros (Crainic, 2015). Diversos autores, como Talukdar et al. (2003); Alba (2005); Crainic (2005); Crainic e Hail (2005); Crainic (2008); Crainic e Toulouse (2007); Talbi (2006); Crainic e Toulouse (2010), utilizaram estas estratégias para solucionar complexos e diferentes problemas de otimização combinatória.

Inicialmente, as metaheurísticas definem sua atuação no respectivo espaço de busca ou domínio, que pode ser a partir de pontos diferentes, de um único ponto, a partir de diferentes populações ou de uma única. Passada esta etapa, cada uma delas começa a aplicar sua estratégia em seu espaço de buscas ou domínio, podendo ser usada a mesma forma de busca, no entanto com diferentes parâmetros, ou metaheurísticas diferentes, para cada respectivo espaço de busca. O método de compartilhamento de informações entre os diferentes espaços de busca define como as metaheurísticas interagem entre si e também como elas devem se comportar após receberem uma nova informação, sendo que informações encontradas por uma metaheurística podem ser compartilhadas globalmente com as demais ou com apenas algumas em específico. Cantú-Paz (2005) desenvolveram algoritmos genéticos paralelos com esta estratégia, Toulouse et al. (1999) desenvolveram uma estratégia busca multi níveis, Cantú-Paz (2005) também desenvolveram algoritmos genéticos em ilhas, nos quais populações diferentes são alocadas em diferentes nós de um ambiente de programação paralela de memória distribuída, com esta estratégia de cooperação, fazendo a troca direta de dados entre processos. Talukdar et al. (2003, 1998) criaram a estratégia de times assíncronos, os quais separam as soluções em times, e trocam informações em períodos irregulares. Crainic et al. (1996, 1997) elaboraram estratégias de cooperação de conhecimento colegial assíncrona. A troca indireta de informações através de um repositório de dados comuns é utilizada em Rochat e Taillard (1995); Badeau et al. (1997); Berger e Barkaoui (2004) e estratégias de troca de informações à partir de memória centralizada são utilizadas em Crainic et al. (1996, 1997, 2006); Le Bouthillier e Crainic (2005); Jin et al. (2012, 2014).

As informações a serem compartilhadas pelos espaços de busca precisam obedecer algum critério, o qual faz com que ela seja significativa e permita às metaheurísticas que atuam nos respectivos espaços de buscas que as recebam de forma mais oportuna. A informação compartilhada entre as metaheurísticas orienta, de forma mais eficiente, as buscas, fazendo assim, com que sejam encontradas, ao final da execução, soluções de melhor qualidade e em tempo computacional reduzido, muitas vezes melhor que a simples junção de informações de metaheurísticas que operam de forma individual (Crainic, 2015).

Em Crainic et al. (1996) é dito que é desejável que sejam observados uma série de problemas que podem ocorrer ao compartilhar informações entre os diferentes espaços de busca ou domínios. São eles:

- (a) Quais as informações devem ser compartilhadas?
- (b) Entre quais espaços de busca ou domínios estas informações devem ser compartilhadas?

- (c) Quando estas informações devem ser compartilhadas?
- (d) De que maneira estas informações são compartilhadas?
- (e) Qual a importância de se compartilhar estas informações?

Segundo Crainic (2015), é também importante observar que, de forma implícita na literatura, por diversas vezes, existe a possibilidade de ser necessário modificar a informação que será compartilhada entre os espaços de busca ou domínios, e se novas informações serão criadas como complemento, em caso de utilização de metaheurísticas diferentes como formas de busca.

Informações de Contexto são dados coletados rotineiramente pelas metaheurísticas durante suas buscas (Crainic, 2015). Estes dados, normalmente, geram estatísticas, as quais podem informar sobre a presença de elementos que fazem melhorias nas soluções, como, por exemplo, o quão impactante um determinado movimento de busca é para as soluções, medidas de diversidade de populações, importâncias de determinadas soluções para melhorias das buscas, dentre outros.

As trocas de informações entre os diversos espaços de buscas pode ocorrer de duas formas (Crainic, 2015):

- (i) Direta: as informações são trocadas entre os diversos espaços de busca ou processos quando algum critério é atingido por eles, ou quando uma metaheurística ou busca espalha a informação para as demais, tendo somente esta atingido um determinado critério. Segundo Crainic (2015), este segundo caso somente funciona bem quando as demais metaheurísticas ou buscas possuem algum mecanismo que armazene esta informação sem perturbar a sua execução, até que seja oportuno considerá-la;
- (ii) Indireta: as trocas de informações ocorrem através de estruturas de dados independentes, sendo armazenadas e somente utilizadas pela metaheurística ou busca que recebeu a informação de acordo com sua lógica interna de funcionamento para recuperação. Segundo Crainic (2015), as estruturas de dados utilizadas são conhecidas por vários nomes, como, por exemplo, *blackboard*, *memória*, *pool*, *data warehouse*, *reference* e *elite set*. Neste trabalho, designamos o termo *memória* para ser utilizado.

5.6.1 Cooperação Síncrona

Segundo Crainic (2015), sincronização entre metaheurísticas ou buscas é um termo designado para um determinado momento em que todas elas devem paralisar suas execuções, trocar informações entre si e aguardar até que todas estejam prontas a reiniciá-las. Uma sincronização deve ocorrer quando algum critério for atingido.

A sincronização pode ocorrer das seguintes formas:

- (i) Troca Global de Informações: neste tipo de sincronização, normalmente, um processo *master* inicia outros *solvers*, os quais são atribuídos para serem executados por processos *slaves*. Ao atingir um critério, todos os processos *slaves* são paralisados e as informações são compartilhadas entre eles através do processo *master*, o qual também atualiza as informações globais. Após feita a sincronização, os processos *slaves* são novamente iniciados e continuam a fazer buscas por soluções. Normalmente, a implementação deste tipo de sincronização pode ter suas duas primeiras dimensões

classificadas como pC/KS , com a terceira dimensão sendo $SPDS, MPSS, MPDS$. Os autores [García-López et al. \(2002\)](#) propõem um VNS para solucionar o problema das p -medianas. Metaheurísticas baseadas em busca tabu paralelizadas, utilizando este tipo de sincronização para solucionar o *Problema do Caixeiro Viajante* são propostas em [Malek et al. \(1989\)](#), para o *Problema de Roteamento de Veículos* em [Rego \(2001\)](#); [Rego e Roucairol \(1996\)](#), para o *Quadratic Assignment Problem* em [De Falco et al. \(1994b\)](#) e para o *Problema de Mapeamento de Tarefas* em [De Falco et al. \(1994a\)](#);

- (ii) Difusão: um processo *master* inicia várias metaheurísticas ou buscas, as quais são alocados em processos *slaves*. Cada processo *slave* possui uma ligação com outros processos *slaves* diretamente. Ao necessitar sincronizar suas informações, paralisa apenas o respectivo processo *slave* que deve receber a informação. Também a implementação deste tipo de sincronização pode ter suas duas primeiras dimensões classificadas como pC/KS , com a terceira dimensão sendo $SPDS, MPSS, MPDS$. A utilização desta estratégia de sincronização por ser vista em [Calégari et al. \(1997\)](#); [Tongheng e Chundi \(2002\)](#), em que se obteve bons resultados para o *Problema do Design de Redes de Rádio* com algoritmos genéticos em ilhas. O *Problema de Roteamento de Veículos* quando foi solucionado por [Alba e Dorronsoro \(2004\)](#); [Dorronsoro et al. \(2007\)](#), utilizou esta estratégia de sincronização em algoritmos genéticos. Os autores [Folino et al. \(1998\)](#); [Mühlenbein \(1989, 1992\)](#), fizeram explanações para a solução de problemas gerais de otimização combinatória com algoritmos genéticos paralelos utilizando esta estratégia. [Middendorf et al. \(2002\)](#) aplicam esta estratégia de sincronização na metaheurística de colônia de formigas, utilizando-a para multicolônias.

5.6.2 Cooperação Assíncrona

A cooperação assíncrona entre diversas metaheurísticas dá-se por processos que são iniciados de forma independente e que possuem atividades de cooperação próprias, em sua lógica interna ([Crainic, 2015](#)). Para compartilhar informações, normalmente, estas metaheurísticas buscam em outras as informações que precisam, ou recuperam, a partir de um depósito de informações, os dados que necessitam, no momento em que atingem um determinado critério. Normalmente, a implementação deste tipo de cooperação requer uma implementação de paralelismo com as duas primeiras dimensões classificadas como pC/C ou pC/KC e a terceira dimensão sendo $SPDS, MPSS, MPDS$.

Esta forma de compartilhamento de informações apresenta dois principais benefícios:

- (i) proporcionar uma forma de compartilhamento de informações entre processos, sem necessidade de paralelizar os demais; e
- (ii) aumento da capacidade de adaptação à buscas em outros locais do espaço de buscas de maneira dinâmica, pois, à medida que sua execução avança, uma metaheurística ou busca em específico muda, de acordo com algum determinado critério, o seu espaço de buscas.

Neste tipo de cooperação, quando se deseja cooperação por trocas diretas, cada metaheurística envia suas informações diretamente para as outras que atendem a um critério. Quando se deseja uma cooperação indireta, o compartilhamento de informações dá-se por

um repositório global de dados, de onde pode-se recuperar informações, quando atingirem um critério. Com a cooperação multinível, uma metaheurística faz uma busca por soluções; após, de forma hierárquica, esta solução é enviada a outras metaheurísticas ou buscas, uma a uma e de maneira linear, conforme Crainic (2015).

Sevкли e Aydin (2007) utilizam esta estratégia de cooperação na metaheurística *Variable Neighborhood Search* (VNS). Lin et al. (1994) utilizam esta estratégia para algoritmos genéticos paralelos. *Simulated-Annealing* para particionamento de grafos é utilizado juntamente com esta estratégia de sincronização em Lee e Lee (1992a,b, 1995, 1996) e para o TSP em Sanvicente-Sánchez e Frausto-Solís (2002). O VNS é utilizado com esta estratégia de cooperação em (Polacek et al., 2008) para solucionar o *Problema e Roteamento de Veículos com Janelas de Tempo* e para solucionar o *Problema das p-medianas* em Crainic et al. (2004).

Capítulo 6

Paralelização das Estratégias de Solução Sequenciais

Este capítulo está organizado da seguinte forma. A Seção 6.1 apresenta uma introdução geral ao capítulo. A Seção 6.2 descreve a estrutura de dados usada para a paralelização das estratégias de solução sequenciais. A Seção 6.3 mostra a proposta de paralelização do Algoritmo de Dijkstra. A Seção 6.4 apresenta a paralelização das estruturas construtivas *Greedy*, enquanto a Seção 6.5 mostra a paralelização em baixo nível das estruturas de busca local.

6.1 Introdução

Este Capítulo trata da paralelização em baixo nível (mais detalhes ver Seção 5.3) em GPUs CUDA das heurísticas GA-PIFRO e ILS-PIFRO, as quais, como mostrado no Capítulo 3, são as estratégias de solução que apresentam melhor desempenho e qualidade de solução para o problema até então.

O Capítulo 3 mostra que tanto GA-PIFRO quanto ILS-PIFRO criam diversas ordens de permutação para alocação das requisições $r \in R$ das instâncias e, assim, as heurísticas construtivas *Greedy* ou PSC recebem tais ordens de permutação como parâmetros e, através do algoritmo de Dijkstra (Dijkstra, 1959), alocam tais requisições uma a uma na rede ótica. A depender da ordem de permutação, o caminho ótico de menor custo para uma requisição pode vir a mudar e, assim, produzir uma rede ótica com custo diferente, ou, simplesmente, o algoritmo de Dijkstra encontrar os mesmos caminhos óticos e gerar soluções de custos iguais.

Assim sendo, mostramos, na Seção 8.3, que sempre a maior parte do tempo de execução das metaheurísticas é tomado por execuções do algoritmo de Dijkstra, tendo médias entre 93.58% e 93.88%, conforme mostra a última linha da Tabela 8.1. Portanto, dada a importância do algoritmo de Dijkstra para as estratégias de solução para PIFRO e baseando-se no trabalho de (Harish e Narayanan, 2007), o qual apresenta uma forma de paralelização do algoritmo de Dijkstra para GPUs CUDA, foram feitas as devidas adaptações para que este seja utilizado pelas estratégias de solução baseadas em GPUs CUDA para PIFRO.

A proposta de paralelização em baixo nível das estratégias de solução existentes passa por dois desafios:

- (i) na fase de relaxamento dos vértices, cada aresta do vértice relaxado precisa ser ava-

liada uma a uma, para que se encontre a menor para compor o caminho ótico, o que faz o algoritmo ser inerentemente sequencial. Reduções (Nvidia, 2018) podem ser tentadas para atenuar tal questão, porém, há o custo de aguardarem sincronizações para iniciarem ou serem finalizadas, o que acaba não gerando ganhos de desempenho significativos (Harish e Narayanan, 2007);

- (ii) a construção de soluções viáveis para o problema, usando as heurísticas construtivas existentes, depende que os caminhos óticos para as requisições sejam alocados um a um, o que também as torna métodos inerentemente sequenciais.

Assim sendo, este trabalho propõe a paralelização CUDA do algoritmo de Dijkstra, para que seja executado para a solução do PIFRO, baseando-se em (Harish e Narayanan, 2007).

Além da proposta da paralelização de algoritmo de Dijkstra, este trabalho também propõe, como formas de paralelização em baixo nível para GA-PIFRO e ILS-PIFRO, a geração simultânea de diversas ordens de permutação, o que implica na execução simultânea de várias instâncias do algoritmo de Dijkstra e, obviamente, geração simultânea de soluções viáveis diversas, podendo aumentar o desempenho de tal modalidade de paralelização.

6.2 Estruturas de dados

As características específicas de processamento em GPUs tornam necessário representar grafos e outros elementos em estruturas de dados apropriadas para melhor desempenho. Quando se adota processamento serial para a representação de grafos, frequentemente são utilizadas listas e matrizes de adjacências. Ao representar grafos em matrizes de adjacências convencionais, esta estrutura ocupa porções de memória desnecessárias, uma vez que a não existência de arestas entre pares vértices também precisa estar representada. Para verificar arestas de um vértice, por exemplo, algumas podem ser visitadas sem necessidade, desperdiçando, assim, processamento. Como vantagem, matrizes de adjacências permitem a busca da existência e/ou recuperação do peso de arestas entre vértices específicos em complexidade de tempo $\mathcal{O}(1)$. As listas de adjacências convencionais criam *arrays* do tamanho da quantidade de arestas específicas para cada vértice do grafo. Apesar da economia de memória, obriga o uso de *loops* para processar arestas, talvez desnecessárias, de um vértice em parte das ocasiões.

Portanto, por questões de eficiência no processamento e respeito às restrições de armazenamento em memória, nas GPUs os grafos são comumente representados em matrizes esparsas do tipo *Compressed sparse row (CSR)* ou *Compressed sparse column (CSC)* (Golub e Van Loan, 1996; Hildebrand, 1987), compondo uma *compact adjacency list*. As *compact adjacency lists* reúnem vantagens de matrizes e listas de adjacências, representando grafos basicamente com 3 vetores unidimensionais. O vetor Va , de tamanho $|V|$, contém, em cada posição, a quantidade de arestas do respectivo vértice, enquanto os vetores Da e Wa , ambos de tamanho $|E|$, contém, respectivamente, o vértice de destino e o peso de cada aresta. Cada índice de Va armazena o índice inicial dos vértices de destino correspondentes às arestas do n^{th} vértice em Da ; sendo assim, a quantidade de arestas do vértice n é dada por $Va[n+1] - Va[n]$. A única vantagem não oferecida por esta estrutura é a verificação da existência de arestas entre pares de vértices específicos em tempo $\mathcal{O}(1)$; porém, os algoritmos aqui apresentado não fazem uso deste recurso.

6.3 Algoritmo de Dijkstra paralelo para GPUs CUDA

O algoritmo de *shortest path* utilizado neste trabalho é uma versão paralela para GPUs CUDA do Algoritmo de Dijkstra (Dijkstra, 1959; Harish e Narayanan, 2007), visto que apresenta o melhor desempenho até o momento na literatura.

Para encontrar o caminho mínimo de um vértice de origem $s \in G$ para todos os demais vértices de um grafo, o algoritmo de Dijkstra, primeiramente, atribui um valor de distância infinito para alcançar cada um deles, excetuando-se s , ao qual atribui-se distância 0 (zero). Após isso, a partir de s , é dado início ao processo de relaxamento, o qual consiste em verificar se a distância para alcançar cada um de seus vértices adjacentes é maior que a distância até ele próprio, somado o peso de sua aresta de saída até o vértice; caso sim, a distância é atualizada. O processo de relaxamento ocorre iterativamente nos vértices adjacentes uns aos outros e termina após todos os vértices do grafo terem sido visitados, caso não tenha sido determinado um vértice de destino, ou, caso contrário, encerra-se quando o vértice de destino é alcançado. O algoritmo retorna um vetor custo de tamanho $|V|$, com a distância mínima para atingir os respectivos vértices de acordo com seus índices.

Na literatura, são encontrados diversos trabalhos, com diferentes formas de paralelização em GPUs do Algoritmo de Dijkstra, como (Martín et al., 2009), (Singh e Khare, 2013), (Harish e Narayanan, 2007) e (Ortega-Arranz et al., 2013). Em todos os trabalhos, nota-se que:

- (i) apesar de se poder relaxar mais de um vértice simultaneamente, o processo de análise de suas arestas é, obrigatoriamente, sequencial, pois o algoritmo relaxará na próxima iteração a aresta de menor distância pertencente ao vértice;
- (ii) dado que diversas arestas podem ter como destino um mesmo vértice, ocorre uma condição de corrida entre as *threads*, e, por tal motivo, é necessário fazer atualização das distâncias via operação atômica.

Nessa implementação, cada vértice a ser relaxado possui uma *thread* designada para si. Estas *threads* devem ser sincronizadas a cada iteração do processo e, assim, ao operar em grafos com ampla heterogeneidade nos graus dos vértices, *threads* originárias naqueles com graus menores podem ser forçadas a um alto tempo de espera, causando perda de desempenho do algoritmo. Para atenuar a perda de desempenho, em (Harish e Narayanan, 2007) é exibida uma implementação do Algoritmo de Dijkstra paralelizado em GPUs na qual, antes de cada iteração do processo de relaxamento, é computado um valor limite de distância dos vértices a serem relaxados, evitando relaxamento desnecessário de alguns, provocando diminuição da sobrecarga de processamento e deixando mais uniforme o tempo de sincronização das *threads*.

Há, também, adaptações do algoritmo de Dijkstra paralelizado em CUDA apresentado em Singh e Khare (2013) para operar sobre subgrafos e adaptações do algoritmo para que seja executado em diversas instâncias simultâneas, sem a necessidade de reorganização da grade de processamento das GPUs.

Os algoritmos 11, 12, 13, 14 e 15 compõem a proposta de implementação aqui introduzida, baseada em (Harish e Narayanan, 2007), que recebeu neste trabalho adaptações para operar também com subgrafos (variáveis *Rma* e *Ena*).

Nas heurísticas construtivas *Greedy* e *PSC*, a depender da quantidade de soluções viáveis geradas simultaneamente, uma instância do algoritmo de Dijkstra deve ser chamada. Na

Algorithm 11: Kernel 1: Dijkstra

Input : $Va, Sa, Da, Wa, Rma, Ena, Thr, Flag, s, t, ma, ca, vertex, tpc$
Output: array ca contendo as distâncias mais curtas de s para todos os outros vértices $v \in G$ ou apenas t para todos $candsimul$

```

1   $Thr \leftarrow \emptyset$ ;  $Flag \leftarrow false$ ;
2  for  $th \leftarrow vertex$ ;  $th < |V|$ ;  $th \leftarrow th + tpc$  do
3       $ma_{[th]} \leftarrow false$ ;
4       $ca_{[th]} \leftarrow \infty$ ;
5      if  $th = s$  then
6           $ca_{[th]} \leftarrow \emptyset$ ;
7           $Flag \leftarrow true$ ;
8           $Thr \leftarrow \infty$ ;
9      end
10 end
11 barrier();
12 while ( $Thr < \infty$ ) and ( $Flag = true$ ) do
13     barrier();
14     if ( $Flag = true$ ) then
15          $Thr \leftarrow \infty$ ;
16     end
17     for  $th \leftarrow vertex$ ; ( $th < |V|$ );  $th \leftarrow th + tpc$  do
18          $THRESHOLD(th, thr, Va, Da, Wa, Rma, Ena, ma, ca)$ ;
19     end
20     barrier();
21     for ( $th \leftarrow vertex$ ; ( $th < |V|$ );  $th \leftarrow th + tpc$ ) do
22          $RELAX(th, thr, Va, Da, Wa, Rma, Ena, ma, ca)$ ;
23     end
24     barrier();
25     for ( $th \leftarrow vertex$ ; ( $th < |V|$ );  $th \leftarrow th + tpc$ ) do
26         if ( $ma_{[vertex]} = false$ ) and ( $Rma_{[vertex]} = false$ ) and ( $t = \emptyset$ ) then
27              $Flag \leftarrow false$ ;
28         end
29     end
30     barrier();
31 end
32 return  $ca$ ;

```

maioria dos trabalhos na literatura, para tal, é usado paralelismo dinâmico, o que exige reorganização da grade de processamento e causa lentidão (Wang e Yalamanchili, 2014), (DiMarco e Taufer, 2013). Portanto, este trabalho também propõe uma adaptação para que o algoritmo possa ser executado sem a utilização de paralelismo dinâmico, evitando, assim, perda de desempenho, dada a necessidade de reorganização da grade de processamento das GPUs em questão.

A versão do Algoritmo de Dijkstra implementada, mostrada no Algoritmo 11, recebe, como parâmetros, o identificador $vertex$ (calculado como o resto da divisão do número de

identificação da *thread* pelo número de soluções candidatas simultâneas a serem geradas), um vértice de origem s e um vértice de destino t (que pode ser NULL, para localizar os menores caminhos de s para todos os demais vértices), os vetores Va (*Array* de vértices), Da (*Array* de destinos), Wa (*Array* de pesos), Rma (*Array* de vértices marcados como removidos) e Ena (*Array* de arestas marcadas como removidas). Também são recebidos os vetores de tamanho $|V|$:

- (i) ma : vetor de tipo boolean de vértices do grafo que foram visitados;
- (ii) ca : vetor de tipo numérico utilizado para armazenar o custo total para se chegar até um determinado vértice;
- (iii) tpc *threads* por candidato; quantas *threads* vão processar cada candidato.

Neste algoritmo são criadas as variáveis (compartilhadas entre as *threads*) thr (*threshold*, armazena o valor limite de distância dos vértices a serem relaxados) e $flag$ (determina o fim da execução do algoritmo). As *threads* referentes a uma determinada solução candidata permanecem executando os *kernels* de cálculo de limite e relaxamento até o vértice executar uma iteração em que o vértice de destino seja visitado. Dentro do *loop* entre as linhas 12 e 31, se faz a chamada às funções de verificação do valor limite, relaxamento e verificação de encerramento do algoritmo utilizando os *loops stride-based* para cobrir todos os vértices do grafo, nos quais, a cada iteração, um conjunto de *threads* processa uma porção de vértices.

No final da execução, se algum vértice presente no grafo não fora visitado e se um vértice de destino não estiver especificado, é atribuído à variável *Flag* um valor que informa ao Algoritmo de Dijkstra que se deve fazer mais uma iteração. Caso a primeira condição seja falsa, um vértice de destino esteja especificado e o custo para atingi-lo seja menor que infinito, o loop do Algoritmo de Dijkstra será finalizado, dado o valor da variável *Flag*.

Como forma de estruturar o código do *Kernel* 11, ele foi modularizado em funções *device*, que são descritas à seguir.

O Algoritmo 12 é uma função *device*, responsável por calcular o valor limite de distância dos vértices a serem relaxados.

Algorithm 12: DeviceFunction 1: THRESHOLD

Input : $vertex, Thr, Va, Da, Wa, Rma, Ena, ma, ca$
Output: valor threshold máximo para o vértice ser relaxado

```

1 if ( $ma_{[vertex]}=true$ ) and ( $Rma_{[vertex]}=false$ ) then
2   for  $i \leftarrow Va[vertex]; i < Va[vertex + 1]; i \leftarrow i + 1$  do
3     if ( $ma_{[Da[i]]} = false$ ) and ( $Rma_{[Da[i]]} = false$ ) and ( $Ena_{[i]} = true$ ) then
4       begin atomic region
5         if  $Thr > (ca_{[vertex]} + Wa[i])$  then
6            $Thr \leftarrow ca_{[vertex]} + Wa[i];$ 
7         end
8       end atomic region
9     end
10  end
11 end
12 return  $Thr$ ;

```

A função **THRESHOLD** recebe, via parâmetros, os vetores Va , Da , Wa , Ra , Ena , ma , ca previamente explicados; a variável th , que representa o vértice a ser analisado, advinda da variável que controla a quantidade de execuções do *stride-loop* pelo qual a função foi chamada; e a variável Thr , a qual é compartilhada entre todas as *threads* referentes a uma solução candidata e, ao final da execução da função, conterá o valor limite de distância dos vértices a serem relaxados. A primeira estrutura de decisão (**if**) verifica se o vértice candidato a estabelecer o valor limite já foi visitado e se não foi marcado como removido (linha 1). Sendo satisfeita esta condição, o *loop* de verificação de cada aresta do vértice (execução sequencial dentro de uma *thread*) é executado em paralelo entre todas as *threads* (várias *threads* analisam vértices simultaneamente) (linha 2). Cada aresta ($Ena[i]$) é analisada. Caso ela e seu vértice de destino ($Rma[Da[i]]$) não tenham sido removidos do grafo e o vértice de destino não tenha sido previamente visitado ($ma[Da[i]]$) (linha 3), então, caso o valor de *threshold* (Thr) for maior que a soma da menor distância até o vértice analisado mais o peso de sua menor aresta, atualiza-se seu valor atômicaamente pela soma de menor distancia¹. Por fim, a função retorna o valor limite na variável $Thr(threshold)$.

Uma vez calculado pelo Algoritmo 12 o limite máximo de distância dos vértices a serem relaxados, a função *device* do Algoritmo 13 é chamada para executar o relaxamento.

Algorithm 13: DeviceFunction 2: RELAX

Input : $vertex, Thr, Va, Da, Wa, Rma, Ena, ma, ca$
Output: array ca contendo o custo atualizado para alcançar o vértice e array ma contendo o vértice resolvido e não resolvido

```

1 if ( $ma_{[vertex]} = \text{true}$ ) and ( $Rma_{[Da[vertex]]} = \text{false}$ ) and ( $ca_{[vertex]} < Thr$ ) then
2    $ma_{[vertex]} \leftarrow \text{true};$ 
3   if ( $ma_{[Da[vertex]]} = \text{false}$ ) and ( $Rma_{[vertex]} = \text{false}$ ) and ( $Ena_{[vertex]} = \text{true}$ ) then
4     begin atomic region
5       if ( $ca_{[Da[vertex]]} > (ca_{[vertex]} + Wa_{[vertex]})$ ) then
6          $ca_{[Da[vertex]]} \leftarrow ca_{[vertex]} + Wa_{[vertex]};$ 
7       end
8     end atomic region
9   end
10  return  $ca, ma;$ 
11 end

```

A função **RELAX** verifica se o vértice a ser relaxado não foi visitado, não foi removido do grafo e o caminho mínimo atual até ele é menor que o valor limite estipulado em Thr . Caso essas condições sejam satisfeitas, o vértice é marcado como visitado e faz-se o relaxamento de suas arestas sequencialmente. Estando a aresta e seu vértice de destino presentes no grafo, atualiza-se atômicaamente a distância até ele, caso seja maior que a distância até o vértice relaxado mais o peso da aresta analisada. A função retorna os vetores ca e ma atualizados. Ao fim de sua execução, o Algoritmo de Dijkstra retorna um vetor de tamanho $|V|$, contendo a distância total do *shortest path* até cada respectivo vértice para cada instância solicitada. Para extrair os caminhos completos até os vértices, a função 14 deve ser executada para cada vértice.

¹Informações detalhadas sobre operações atômicas em CUDA são vistos em (Nvidia, 2018)

Algorithm 14: DeviceFunction 5: ExtractPreviousEdges

Input : $Sa, Da, Wa, ca, prev$
Output: array $prev$ contendo o índice da última borda de seu caminho mais curto

```

1 private<var> tid  $\leftarrow$  getThreadId();
2 private or shared<var> stride  $\leftarrow$  getStrideSize();
3 shared or global<array of size  $|V|$ > prev;
4 for th  $\leftarrow$  tid; th <  $|E|$ ; th  $\leftarrow$  th + stride do
5   | if ( $Wa[th] + ca[Da[th]] = ca[Da[th]]$ ) and ( $ca[Da[th]] < \infty$ ) then
6   |   |  $prev[Da[th]] \leftarrow th$ ;
7   | end
8 end
9 barrier();
10 return prev;

```

A função `ExtractPreviousEdges`, mostrada no Algoritmo 14, recebe, como um dos parâmetros, o vetor $prev$, de tamanho $|V|$, o qual é retornado ao final de sua execução contendo, em cada uma de suas posições, o índice da última aresta que compõe o *shortest path* para alcançar o respectivo vértice. A função compara o peso de cada aresta $e \in E$, somado ao peso final do *shortest path* do seu vértice de origem. Caso seja verdadeira, o índice referente à aresta é armazenado na posição equivalente a seu vértice de destino em $prev$.

Após obter, via função `ExtractPreviousEdges`, o vetor $prev$, um caminho de s até t pode ser recuperado através da função `ExtractPath`, introduzida no Algoritmo 15.

Para extrair o *shortest path* de s até t a partir do array $prev$, previamente gerado pela função `ExtractPreviousEdges`, a função `ExtractPath`, sequencialmente, após verificar se no array $prev$ o vértice de t foi alcançado pelo Algoritmo de Dijkstra, executa um *loop* que compõe o caminho com o índice de suas arestas no sentido $t \rightarrow s$. O caminho é armazenado no array $path$, recebido por parâmetro, e é retornado ao final, juntamente com as variáveis $cost$ e $length$, obtidas da mesma forma e que carregam as informações de custo e quantidade de arestas presentes no array $path$. Uma vez que a descoberta de uma aresta precisa ser precedida de outra, a função é sequencial; porém, várias instâncias podem ser executadas em paralelo.

Em conclusão, a paralelização do Algoritmo de Dijkstra consiste na execução simultânea da busca por valores menores ou iguais que o *threshold value* e no relaxamento para cada vértice. No relaxamento, a verificação sequencial das arestas dos vértices acarreta perda de desempenho. Também observa-se a necessidade da existência de barreiras de sincronização após a execução das funções. Na computação de *shortest paths* dos vértices podem ser paralelizadas múltiplas buscas das arestas finais dos *shortest path* dos vértices.

Algorithm 15: DeviceFunction 6: ExtractPath

Input : $s, t, Sa, Wa, ca, prev$
Output: array $path$ contendo o índice de arestas que compõe o caminho mais curto de s a t , a variável $cost$ contendo o custo total do caminho e a variável $length$ contendo o total de arestas em $path$

```

1 private<var>  $cost \leftarrow 0, length \leftarrow 0;$ 
2 private<array of size  $|V|$ >  $path \leftarrow \emptyset;$ 
3  $i \leftarrow 1;$ 
4  $next \leftarrow t;$ 
5 if  $ca_{[t]} < \infty$  then
6   while true do
7      $path_{[i]} \leftarrow prev_{[next]};$ 
8      $cost \leftarrow cost + Wa[i];$ 
9     if  $(Sa[path_{[i]}] = s)$  then
10       $\text{break};$ 
11     else
12       $next = Sa[path_{[i]}];$ 
13       $i \leftarrow i + 1;$ 
14     end
15   end
16    $length \leftarrow i;$ 
17 end
18 return  $path, cost, length;$ 

```

6.4 Paralelização das heurísticas construtivas

O funcionamento (e aumento de desempenho) da heurística construtiva *Greedy*, paralelizada via estratégia de baixo nível em CUDA, é baseada na obtenção de *lightpaths* via algoritmo de Dijkstra paralelo. A sua lógica de funcionamento é semelhante à sua versão sequencial, fato que ocorre com frequência neste tipo de paralelização, conforme é citado na Seção 5.3. O pseudocódigo do Algoritmo 16 detalha os pontos de sincronização e de utilização de paralelismo via *grid stride loops* utilizados pela heurística e nomeia como *CUDA Parallel Greedy Shortest Path Based*.

Observa-se no Algoritmo 16 que, inicialmente, a heurística *CUDA Parallel Greedy Shortest Path Based* recebe como parâmetros o grafo G , o conjunto de requisições R e os valores das constantes μ , ROADMCost, OTCost, OACost. O parâmetro final *permutation* é um vetor de tamanho $|R|$, no qual cada índice contém o número da requisição $r \in R$ a ser alocada na rede. A cada requisição r a ser alocada na rede um *stride loop* é acionado, para que seja calculado de forma paralela o custo incremental de alocação da requisição em cada aresta de G . Um vez realizados tais cálculos, o Algoritmo de Dijkstra paralelo é chamado e calcula, de acordo com o custo incremental, o caminho de menor custo para a requisição.

A paralelização em baixo nível da heurística PSC é dada de forma praticamente idêntica à paralelização da heurística *Greedy*, modificando-se apenas a forma de cálculo do custo incremental, conforme é visto na Seção 3.2.2 e incrementando o valor de p a cada iteração. Neste trabalho a paralelização da heurística PSC é nomeada por *CUDA Parallel PSC Shortest Path Based*.

Algorithm 16: DeviceFunction: *CUDA Parallel Greedy Shortest Path Based*

Input : $G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}, \text{permutation}$
Output: um conjunto X contendo os K -Shortest Paths de s a $t \forall r \in R$

```

1 shared or private<var>  $i \leftarrow 0, \text{tid} \leftarrow \text{getThreadId}(), \text{stride} \leftarrow \text{getGridSize}()$ ;
2 for ( $i \leftarrow 0; i \leq |R|; i \leftarrow i + 1$ ) do
3   for ( $\text{th} \leftarrow \text{tid}; \text{th} < |E|; \text{th} \leftarrow \text{stride}$ ) do
4     | calculate the cost to allocate the wavelengths in the  $\text{th}$  of  $G$ ;
5   end
6   barrier();
7   Chamada do Parallel Dijkstra para encontrar o caminho com o menor custo para
   |  $r_i$  em  $G$  e alocar os comprimentos de onda;
8 end
9 return  $G$ ;

```

Quanto à classificação tridimensional baseada na taxonomia de [Crainic et al. \(1997\)](#); [Crainic e Hail \(2005\)](#), da paralelização em baixo nível para CUDA, da heurística *Greedy*, têm-se que:

- (i) faz sua busca através de único processo, portanto tem a classificação na dimensão 1 em $1C$;
- (ii) realiza a sincronização das informações pertinentes a cada execução do algoritmo de Dijkstra e, portanto, tem classificação na dimensão 2 em RS ;
- (iii) a heurística utiliza único ponto do espaço de buscar para iniciar e, portanto, tem classificação da dimensão 3 em $SPSS$.

Portanto, a classificação baseada na taxonomia tridimensional de [Crainic et al. \(1997\)](#); [Crainic e Hail \(2005\)](#) da heurística *Greedy* paralelizada em baixo nível para CUDA é $1C/RS/SPSS$.

6.5 Paralelização em baixo nível da busca local

Assim como a paralelização em baixo nível das heurísticas construtivas, a paralelização em baixo nível da busca local também herda a lógica de funcionamento da busca local sequencial. O aumento de desempenho também advém da paralelização do Algoritmo de Dijkstra e da possibilidade de paralelizar as etapas de remoção e alocação de requisições na rede. O Algoritmo 17 exhibe as especificidades como pontos de sincronização e utilização de *grid stride loops* necessários. Este trabalho nomeia paralelização em baixo nível da busca local existente para PIFRO por *CUDA Parallel Local Search Shortest Path Based*.

Nota-se, no Algoritmo 17, que as operações de remoção das requisições, cálculo do custo de alocação da requisição nos enlaces e chamadas ao Algoritmo de Dijkstra, apesar de serem paralelas, necessitam ocorrer em etapas a cada iteração e, por isso, são impostas barreiras de sincronização entre cada etapa.

Quanto à classificação tridimensional baseada na taxonomia de [Crainic et al. \(1997\)](#); [Crainic e Hail \(2005\)](#), da paralelização em baixo nível para CUDA, da busca local para PIFRO, têm-se que:

Algorithm 17: DeviceFunction: *CUDA Parallel Local Search Shortest Path Based*

Input : $G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}, \text{permutation}$
Output: um conjunto X contendo os K -Shortest Paths de s a $t \forall r \in R$

```

1 shared or private<var>  $i \leftarrow 0, \text{tid} \leftarrow \text{getThreadId}(), \text{stride} \leftarrow \text{getGridSize}()$ ;
2 for ( $i \leftarrow 0; i \leq |R|; i \leftarrow i + 1$ ) do
3   | in parallel remove the cost of  $r_i$  from  $G$ ;
4   | barrier();
5   | for ( $\text{th} \leftarrow \text{tid}; \text{th} < |E|; \text{th} \leftarrow \text{stride}$ ) do
6   |   | calcular o custo para alocar os comprimentos de onda em  $\text{th}$  de  $G$ ;
7   | end
8   | barrier();
9   | Chamada do Parallel Dijkstra para encontrar o caminho com o menor custo para
    |    $r_i$  em  $G$  e aloque os comprimentos de onda;
10  | barrier();
11 end
12 return  $G$ ;

```

- (i) faz sua busca através de único processo, portanto tem a classificação na dimensão 1 em $1C$;
- (ii) realiza a sincronização das informações pertinentes a cada execução do algoritmo de Dijkstra, remoção e adição de requisições na rede ótica e, portanto, tem classificação na dimensão 2 em RS ;
- (iii) a busca local utiliza único ponto do espaço de busca para iniciar, portanto, tem classificação da dimensão 3 em $SPSS$.

Portanto a classificação baseada na taxonomia tridimensional de [Crainic et al. \(1997\)](#); [Crainic e Hail \(2005\)](#) da busca local paralelizada em baixo nível para CUDA é $1C/RS/SPSS$, assim como a paralelização em baixo nível das heurísticas construtivas.

Capítulo 7

Estratégias de Solução Paralelas

Este capítulo apresenta estratégias de solução CUDA para o Problema de Instalação de Fibras em Redes Óticas (PIFRO). A Seção 7.1 a seguir apresenta uma introdução geral ao que é abordado. A Seção 7.2 apresenta topologias de processamento paralelo hierárquico. A Seção 7.3 apresenta novas estratégias de solução para o problema baseadas na geração de caminhos a partir de um armazém destes criado em uma etapa de pré-processamento. A Seção 7.4 descreve a paralelização CUDA do Algoritmo de Yen. A Seção 7.5 introduz a paralelização CUDA do Algoritmo Generalização de Yen e do Algoritmo MPS. As Seções 7.6, 7.7 e 7.8 descrevem as propostas de paralelização CUDA das heurísticas construtivas e de busca local, da metaheurística *Biased Random Key Genetic Algorithm* (BRKGA) e da metaheurística *Iterated Local Search* (ILS). Por fim, a Seção 7.9 apresenta uma análise de dominância do espaço de buscas com soluções geradas via *Paths Pool*.

7.1 Introdução

Segundo Sörensen (2015); Crainic et al. (2014); Crainic (2015), é importante que sejam implementadas maiores quantidades de estratégias de paralelismo de um mesmo *solver*, para um determinado problema. Estes trabalhos explicam ainda a importância de que isso ocorra, pois cada qual possui suas particularidades, o que influencia diretamente na qualidade final das soluções e o desempenho (velocidade) atingida pelo *solver*. Sörensen (2015); Crainic et al. (2014); Crainic (2015) asseguram ainda que a literatura atual pede que sejam elaboradas novas formas de paralelismo.

Conforme visto nas seções 6.4 e 6.5, a paralelização CUDA em baixo nível das estratégias de solução existentes para PIFRO tem grande dependência do Algoritmo de Dijkstra. De acordo com os resultados dos experimentos exibidos na Seção 8.3, em torno de 93,58% e 93,88% do tempo demandado para execução destas estratégias é tomado por sucessivas execuções deste algoritmo.

Embora existam diferentes formas de paralelização do Algoritmo de Dijkstra em CUDA, como, por exemplo, as mostradas em Harish e Narayanan (2007), Martín et al. (2009) e Crauser et al. (1998), a utilização deste algoritmo para solucionar subproblemas pode criar um gargalo (*bottleneck*), dados seus pontos inerentemente sequenciais. Assim sendo, minimizar a dependência das estratégias de solução para PIFRO com relação ao Algoritmo de Dijkstra, bem como aprimorar sua forma de utilização, aumenta o desempenho das estratégias de solução.

Neste capítulo propomos estratégias de solução paralelizadas em CUDA para o problema, porém, abordando topologia de processamento hierárquico (Talbi et al., 2008), em busca principalmente do ganho de desempenho através da redução de dependência do Algoritmo de Dijkstra. Conforme é mostrado nas seções seguintes, as estratégias de solução propostas foram projetadas já em formato paralelizado em GPUs CUDA, levando em consideração todas as particulares para a arquitetura e, além disso, que possam ser executadas em quaisquer plataformas CUDA existentes até o momento. É possível, ainda, para tais estratégias de solução, garantir, através da análise de dominação, alguma qualidade para as soluções apresentadas (Glover e Punnen, 1997; Gutin e Yeo, 2002; Orlin e Sharma, 2004).

O formato proposta para a implementação destas estratégias de solução paralelizadas em topologia hierárquica é o *Problem dependent intra-algorithm parallelization* (visto em detalhes na Seção 7.2), o qual executa tarefas (ou algoritmos) uma a uma para compor uma estratégias de solução, porém com cada tarefa (ou algoritmo) paralelizado. Para as estratégias propostas, é necessária a execução de três etapas:

- (i) **Etapa 1:** Criar um armazém com caminhos entre os vértices de origem e destino para cada requisição $r \in R$ da instância de PIFRO;
- (ii) **Etapa 2:** Construir soluções viáveis para a instância através da combinação de um caminho advindo do armazém de caminhos para cada requisição $r \in R$ da instância de PIFRO e;
- (iii) **Etapa 3:** Opcionalmente, tentar melhorar a solução viável trocando os caminhos para determinadas requisições r da instância, utilizando, para isso, caminhos advindos do armazém ou de algoritmos de *shortest path*.

Cada etapa necessária para a implementação das heurísticas paralelizadas em topologia hierárquica é detalhada nas seções seguintes.

7.2 Topologias de Processamento Paralelo Hierárquica

Crainic et al. (2014) afirmam que as topologias de processamento paralelo mais frequentemente utilizadas são *Completamente Conectada*, *Estrela* e *Anel de Processamento Unidirecional*. Em Talbi et al. (2008) são classificadas metaheurísticas paralelas hierárquicas. Segundo Talbi et al. (2008), as metaheurísticas paralelas hierárquicas são adequadas em situações em que existem um grande número de núcleos de processamento disponíveis, porém, a interconexão entre eles é lenta, como, por exemplo, em *clusters* em que é utilizado um ambiente de processamento paralelo de memória distribuída. Sendo assim, neste tipo de metaheurística, são delegadas tarefas para os núcleos de processamento existentes e, após algum critério ser atingido, as soluções são avaliadas, unidas, ou retornadas.

Segundo Talbi et al. (2008), as metaheurísticas paralelas hierárquicas podem ser classificadas em:

- (i) *Self-contained parallel cooperation*: este grupo de metaheurísticas faz com que cada núcleo de processamento execute um algoritmo independente, usando uma sub-população diferente, o que faz com que elas sejam frequentemente chamadas na literatura de “ilhas”. Estes algoritmos podem cooperar trocando soluções quando algum critério seja atingido;

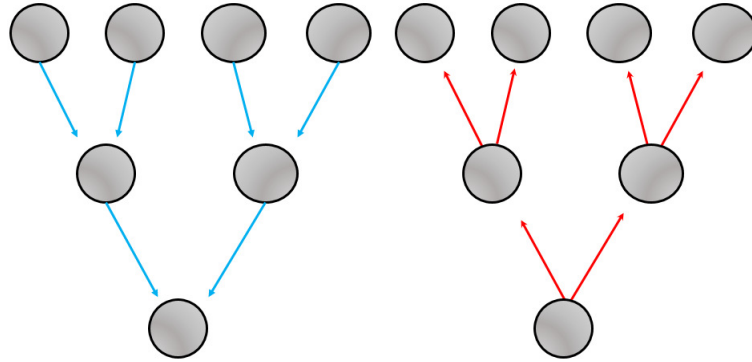


Fig. 7.1: Topologia de processamento paralelo “Hierárquica”.

- (ii) *Problem independent intra-algorithm parallelization*: dado que a maioria das metaheurísticas são métodos iterativos, este modelo aproveita-se desta característica para paralelizar uma única iteração do algoritmo;
- (iii) *Problem dependent intra-algorithm parallelization*: as operações necessárias da metaheurística são executadas em paralelo. Em geral, o maior interesse deste modelo é a paralelização da avaliação de uma única solução. Este modelo é paralelizado com base em particionamento de dados ou particionamento de tarefas.

À medida em que tais metaheurísticas são executadas e critérios são atingidos, as *threads* migram para outros processos ou espaços de buscas, levando consigo suas informações relevantes. A cada vez que os critérios são atingidos, menos espaços de buscas serão explorados por tais *threads*, o que faz que sejam aproveitadas de maneira mais eficiente, abandonando espaços de buscas que contribuem menos para o resultado desejado e auxiliando no processamento de espaços de buscas aparentemente mais promissores. Ainda segundo Talbi et al. (2008), a topologia hierárquica pode também funcionar em sentido contrário, aumentando espaços de buscas explorados e distribuindo suas *threads*.

A ilustração para esta topologia pode ser visualizada na Figura 7.1. Nesta figura, cada círculo representa um espaço de busca ou processo. As setas indicam a migração das *threads* entre eles, que pode ser no sentido de serem concentradas em espaço(s) de busca ou processo(s), após algum critério ser atingido, ou serem distribuídas entre vários destes, após critérios serem alcançados.

7.3 Paths Pool: Armazém de Caminhos entre Vértices

Uma vez exibidos os gargalos que comprometem a paralelização do algoritmo de *shortest path*, que possui melhor desempenho para GPUs–Dijkstra (Seção 6.3), e a alta dependência das estratégias de solução com melhores resultados para PIFRO até então (Capítulo 3), esta tese propõe que novas estratégias de solução para o problema baseiem-se em obter caminhos a partir de um armazém destes criado em uma etapa de pré-processamento.

Para tal, propomos diferentes formas de paralelização em GPUs CUDA de algoritmos capazes de obterem diversos caminhos entre pares de vértices de grafos. Tal motivação para escolha destes algoritmos advém do fato das requisições r das instâncias de PIFRO necessitarem destes diretamente.

A construção do *Paths Pool* é modelada fazendo com que cada requisição r de uma instância execute um algoritmo para solução do problema *K-Shortest Loopless Paths* (Hoffman e Pavley, 1959), o qual extrai, de um grafo, caminhos entre pares de vértices $\langle i, j \rangle$, sem que hajam repetições, ordenados por custos e disponibilizando-os em memória para as heurísticas, sendo a execução deste algoritmo, portanto, a **Etapa 1** das estratégias de solução paralelizadas CUDA na topologia de processamento hierárquica para PIFRO. Logo na entrada da execução das estratégias de solução, deve ser informado, ao algoritmo de *K-Shortest Loopless Paths*, a quantidade de caminhos para requisição, para, em seguida, as heurísticas criarem soluções viáveis para as instâncias através dos caminhos (na forma de um para cada requisição r).

Dentre os algoritmos encontrados na literatura para solução do problema *k-Shortest Loopless Paths*, o mais comumente utilizado é o Algoritmo de Yen, introduzido em Yen (1971). O Algoritmo de Yen faz ampla utilização do Algoritmo de *Shortest Path* e, por tal motivo, gera desconfiança no que tange a seu desempenho em CUDA e, por isso, desperta interesse que o desempenho de sua versão paralelizada para CUDA seja comparada com o mesmo formato de paralelização de outros algoritmos para o problema. O Algoritmo Yen Generalizado e o Algoritmo MPS, ambos propostos em Martins et al. (1999), fazem a utilização do Algoritmo de *Shortest Path* uma única vez no decorrer de suas execuções, mas, em contrapartida, criam estruturas de memória que armazenam dados pertinentes advindos deste algoritmo. Por este motivo, para a criação do *Paths Pool*, este trabalho, além de mostrar formas inéditas na literatura da paralelização CUDA destes algoritmos, compara, no Capítulo 8, os desempenhos das propostas aqui apresentadas.

7.3.1 O subproblema *K-Shortest Paths*

Popular há décadas, o problema *K-Shortest Loopless Paths* (KSLP) (Martins et al., 1999; Feng, 2014; Hershberger et al., 2007; Yen, 1971) consiste em encontrar, em um grafo $G(V, E)$, no qual V representa o conjunto de vértices e E o conjunto de arestas que os conecta, um número de K caminhos, tal que $K > 1$, sem repetições de vértices entre uma origem $s \in V$ e um destino $t \in V$, classificados em ordem crescente de custo (ou distância total). Este problema foi proposto em 1959 por Hoffman e Pavley (1959), e o interesse no seu estudo reside em suas aplicações reais, como roteamento em redes, rastreamento de múltiplos objetos, planejamento de movimentação de robôs, planejamento de *layout* de circuitos, planejamento de tráfego, bancos de dados, dentre outras aplicações, como pode ser visto em Eppstein (1998); Berclaz et al. (2011); Chen e Prasanna (2016); Feng (2014); Hershberger et al. (2007); Hoffman e Pavley (1959); Srivastava et al. (2015); Wen et al. (2013); Yen (1971); Singh e Singh (2015).

As primeiras soluções obtidas para o KSLP, como em Clarke et al. (1963); Bock et al. (1957); Pollack (1961); Sakarovitch (1968), apresentam complexidade exponencial de tempo. Em 1971, Yen (1971) propõe um algoritmo, que aqui será denominado Algoritmo de Yen, capaz de solucionar o KSLP em complexidade de tempo polinomial ($\mathcal{O}(Kn(n \log n + m))$). Este algoritmo atua reduzindo o KSLP para $\mathcal{O}(K)$ instâncias do *Replacement Paths Problem* (Kare, 2016).

Baseado no Algoritmo de Yen, Katoh et al. (1982) propõem um algoritmo, chamado Algoritmo KIM (acrônimo formado pelas iniciais dos autores Katoh, Ibaraki e Mine). O Algoritmo KIM apresenta uma redução na complexidade de tempo para $\mathcal{O}(k(m + n \log n))$ em grafos não direcionados.

Por sua vez, através de alterações feitas no Algoritmo de Yen e técnicas advindas do Algoritmo de KIM, (Martins et al., 1997, 1999) desenvolveram os algoritmos denominados Algoritmo Generalização de Yen e Algoritmo MPS (acrônimo formado pelas iniciais dos autores Martins, Pascoal e Santos). Ambos operam através de um atalho na obtenção de caminhos candidatos em suas iterações. Para obter tal atalho, após uma etapa inicial de pré-processamento, na qual calculam-se os *shortest paths* a partir do vértice de destino $t \in G$ para todos os demais vértices $v \in G$, aproveita-se tais caminhos na obtenção de novos, reduzindo, assim, a complexidade de tempo da solução do problema para $O(m \log n + kn)$ no pior caso.

Mesmo existindo uma vasta gama de estudos disponíveis, ainda é desafiador obter bom desempenho na resolução de instâncias de grandes dimensões do KSLP. É importante destacar que o termo “*instâncias de grandes dimensões do KSLP*” implica, na atualidade, em instâncias de grafos nas quais $|V|$ e $|E|$ são da ordem de centenas de milhões e, por tal motivo, técnicas diversas de paralelização de estratégias de solução para o KSLP têm sido objeto de grande interesse (Ruppert, 2000; Guerriero e Musmanno, 2000; Kumar e Ghosh, 1994; Nuutila, 1995; Eppstein, 1998).

Dado que os Algoritmos de Yen, Generalização de Yen e MPS são capazes de operar em grafos direcionados e não direcionados, continuam servindo como base para o desenvolvimento de outras estratégias de solução para KSLP (como em Martins et al. (1999); Feng (2014); Katoh et al. (1982)) e possuem pontos de paralelização que requerem estratégias diferentes para tal, foram, então, selecionados nesta tese para serem paralelizados em CUDA e, assim, criarem o *Paths Pool* (armazém de caminhos ou *lightpaths*), necessário para novas estratégias de solução paralelas que operam em topologia de processamento hierárquico em CUDA.

As técnicas aqui propostas na paralelização CUDA do algoritmo de Yen, contudo, divergem daquelas mostradas em Singh e Singh (2015), no qual uma estratégia de paralelização CUDA do algoritmo de Yen, apesar de atingir um *speedup* de até $6,5\times$ em relação à versão serial, tem os experimentos restritos a instâncias que possuem dimensões de $|V|$ e $|E|$ adequadas para processamento de todo o grafo em apenas uma iteração pelas grades de processamento da GPU.

Além de colaborarem nas novas estratégias de solução paralelizadas em CUDA para PIFRO, as estratégias de paralelização dos algoritmos para KSLP apresentadas nesta tese têm, como contribuições principais para a literatura do KSLP, os seguintes aspectos:

- (i) Propostas inéditas de estratégias de paralelização dos Algoritmos de Yen, Generalização de Yen e MPS em GPUs Nvidia CUDA;
- (ii) Criação de um mecanismo de busca simultânea por mais de um caminho candidato nestes algoritmos;
- (iii) Representação de subgrafos sem a necessidade de reorganização das estruturas de memórias das GPUs durante a execução;
- (iv) Desenvolvimento de mecanismo para a chamada de diversas instâncias de funções *device*, necessárias aos algoritmos, sem a necessidade de reorganização da grade de processamento das GPUs;
- (v) Execução de experimentos e exibição dos respectivos resultados para diferentes tamanhos de K em instâncias de grafos de grande porte no Capítulo 8.

Neste trabalho, para as propostas de implementação em CUDA dos Algoritmos Yen, Generalização de YEN e MPS, as referências a elementos específicos de *arrays* unidimensionais são dadas por $Array[i]$ ou $Array[i]$, ou seja, o i -ésimo elemento de $Array$. Apesar de, na prática, serem implementados em *arrays* unidimensionais contínuos, *arrays* aninhados (ou *array sets*) têm seus elementos referenciados por $Array[i,y]$, ou seja, o y -ésimo elemento do i -ésimo *array* armazenado em $Array$. Além disso, também são usadas as seguintes expressões:

- i : vértice existente em um caminho tal que $i \in V$;
- p : conjunto de *arrays* contendo caminhos ou partes de caminhos diversos;
- $p_{\langle i,j \rangle}$: *array* contendo as arestas com o path de i até j armazenado em p ;
- T^k : *array* aninhado contendo caminhos candidatos a caminho mais curto na k -ésima iteração;
- X : *array* aninhado com os K -caminhos mais curtos exclusivos e ordenados;
- $C^{[index]}$, $L^{[index]}$ e $A^{[index]}$: *arrays set*, em que cada índice contém, respectivamente, o custo, a quantidade de arestas e a disponibilidade do respectivo caminho em um *array set*.

7.4 O Algoritmo de Yen original e mapa de pontos de paralelização

A Figura 7.2 apresenta, em *flowchart*, a versão original do Algoritmo de Yen, introduzido em Yen (1971), e aponta os seus pontos de paralelização explorados neste projeto de tese.

A Tabela 7.1 mostra a notação adotada para o *flowchart* mostrado na Figura 7.2. Conforme mostra esta figura, o Algoritmo de Yen, ao ser iniciado, para encontrar o primeiro dos k shortest paths (ou seja, *path* quando $k = 1$), aplica um Algoritmo de *Shortest Path* entre os vértices s e t no grafo G e armazena o resultado em $X_{[k]}$. Para cada k^{th} -*Shortest Path* a ser encontrado, o $X_{[k-1]}$ caminho é decomposto vértice a vértice a cada iteração $v : v = 1, 2, \dots$, tamanho de $X_{[k-1]} - 1$.

Root Path é o subcaminho formado pelos v^{th} primeiros vértices de $X_{[k-1]}$. Um subgrafo é criado retirando-se de G todos os vértices v presentes no *Root Path*, assim como as arestas que conectam os vértices de cada um dos caminhos presentes em X que tenham todos os seus primeiros v vértices presentes no *Root Path*. Este procedimento se denomina processo de decomposição. O caminho que parte do último vértice do *Root Path* até t , denominado *Spur Path*, é encontrado pelo Algoritmo de *Shortest Path* no subgrafo (i.e., do último nó do *Root Path* até t).

Sendo encontrado um *Spur Path* no subgrafo, um *Candidate Path* é produzido através de $Root Path \cup Spur Path$. Caso não esteja previamente armazenado um caminho candidato idêntico, este *Candidate Path* gerado é armazenado em T^k . Antes de dar início à iteração seguinte, o algoritmo restaura todas as arestas e vértices de G . A cada iteração do processo de decomposição, o algoritmo verifica se a quantidade de caminhos armazenados em T^k , somados aos caminhos armazenados em X , atingiram a quantidade K . Nesse caso,

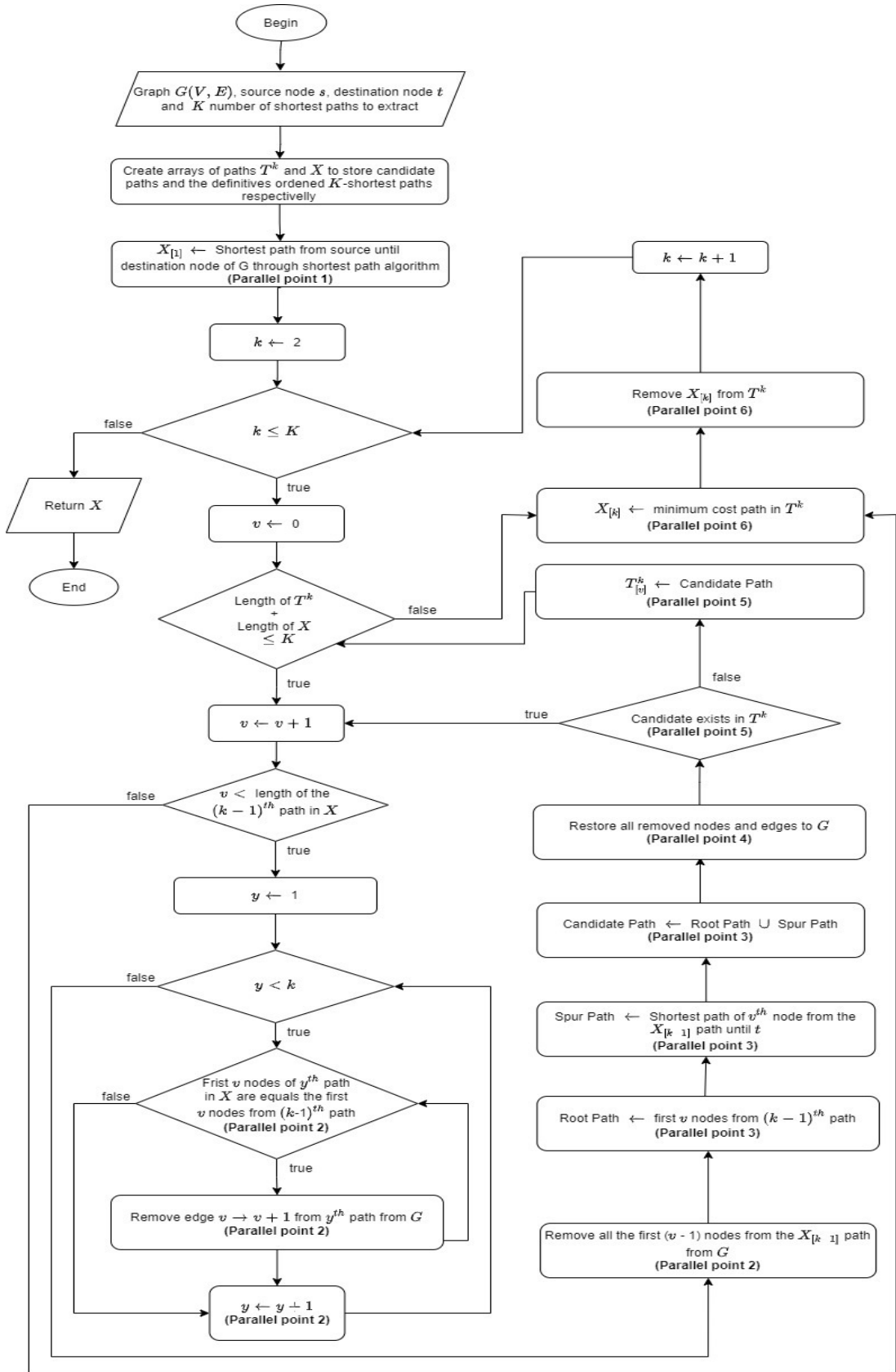


Fig. 7.2: Flowchart para Algoritmo de Yen.

Tabela 7.1: Notação para Figura 7.2.

s :	vértice de origem de um caminho
t :	vértice de destino de um caminho
K :	número K de <i>shortest paths</i> a serem extraídos
T^* :	arrays de <i>shortest paths</i> a partir de cada nós em G até t
T^k :	arrays de candidatas a k^{th} <i>shortest path</i>
T_v^k :	um caminho candidato v encontrado na iteração k
X :	Conjunto das K <i>shortest paths</i> ordenadas em ordem crescente de custo ou distância
$X_{[1]}$:	<i>shortest path</i> em X (ou seja, para $k = 1$)
k :	k^{th} <i>shortest path</i> dentre os candidatos da iteração k
v :	vértice da k^{th} <i>shortest path</i> em X
y :	iterador sobre os caminhos em X
<i>Root Path</i> :	Caminho com os primeiros v nós do caminho em X da iteração anterior ($k - 1$)
<i>Spur Path</i> :	Caminho a partir do nó v do <i>Root Path</i> para t

o processo iterativo de decomposição é encerrado, e, um a um, cada caminho presente em T^k é transferido para X , ordenado por custo (ou distância). Por outro lado, se a soma dos caminhos em X e T^k for menor que K , apenas é transferido para X o *shortest path* armazenado em T^k , incrementa-se k e reinicia-se a decomposição do $X_{[k-1]}$.

Os itens a seguir detalham os pontos de paralelização do Algoritmo de Yen, explorados nesta tese, bem como indicam seus respectivos *kernels*:

- (i) O *kernel* 18 inicializa variáveis e gerencia a execução dos demais *kernels* responsáveis pela geração dos K -*shortest paths* e soluções candidatas;
- (ii) Apontada no *flowchart* da Figura 7.2 como **parallel point 1**, uma versão paralela CUDA do Algoritmo de Dijkstra, baseada em Singh e Khare (2013), é utilizada para obtenção do primeiro K -*shortest path* e também na busca por *spur paths* nos subgrafos das soluções candidatas. O pseudocódigo é apresentado nos Algoritmos 11, 12, 13, 14 e 15;
- (iii) A quantidade massiva de *threads* das GPUs é capaz de gerar em paralelo diversas soluções candidatas. Para tal, as *threads* são agrupadas e alocadas para cada vértice disponível no processo iterativo de decomposição da $X_{[k-1]}$ *shortest path*, de acordo com o *kernel* 20. Mesmo não estando presente visualmente no *flowchart* da Figura 7.2, este *kernel* é o primeiro passo executado para o **parallel point 2**;
- (iv) Para cada solução candidata a ser gerada em paralelo, uma versão diferente de G , ou seja, um subgrafo, precisa ser criado. Isso se dá pelo fato de cada nó do processo iterativo de decomposição de $X_{[k-1]}$ necessitar remover nós e arestas diferentes entre si. O *kernel* 21 é responsável pela elaboração de cada subgrafo. As tarefas realizadas pelo *kernel* 21 são apontadas no *flowchart* da Figura 7.2 como **parallel point 2**;
- (v) Uma vez alocadas as *threads* para cada um dos múltiplos candidatos a serem gerados em paralelo e estabelecidos seus respectivos subgrafos, o *kernel* 19 gera cada *root*

path e a une a *spur path*, finalizando a solução candidata de cada grupo de *threads* designado para tal. Estes passos são apontados no *flowchart* da Figura 7.2 como **parallel point 3**;

- (vi) Após geradas as soluções candidatas, os subgrafos são restaurados, ou seja, recebem de volta seus nós e arestas através do *kernel* 22. As operações executadas pelo *kernel* 22 são apontadas no *flowchart* da Figura 7.2 como **parallel point 4**;
- (vii) Como última etapa do processo iterativo de decomposição de $X_{[k-1]}$, o *kernel* 18 verifica se cada *candidate path* já não existe em T^k e, acaso não exista, a armazena em T^k . Apesar do processo de verificar a repetição de cada caminho precisar ser realizado um a um, sequencialmente, cada verificação é paralelizada. Esta etapa é exibida no *flowchart* da Figura 7.2 como **parallel point 5**;
- (viii) Após a descoberta do *shortest path* presente em T^k via redução, este *path* é transferida em paralelo para $X_{[k]}$ pelo *kernel* 18 de forma paralela e removida de T^k . Tal operação é exibida no *flowchart* da Figura 7.2 como **parallel point 6**.

7.4.1 Paralelização CUDA do Algoritmo de Yen

Uma proposta de paralelização em CUDA para o Algoritmo de Yen é introduzida nos Algoritmos 18, 19, 21 e 22, sendo detalhada em seguida. O algoritmo sequencial associado está descrito no *flowchart* mostrado na Figura 7.2.

Na proposta de paralelização do Algoritmo 18, primeiramente é obtido o *shortest path* de s até t no grafo G e suas respectivas informações armazenadas em X (veja Section 6.3). Uma vez que a função **ExtractPath**, mostrada no Algoritmo 15, possui caráter sequencial, é executada apenas pela *thread* 0 (escolhida de forma arbitrária). O processo iterativo de decomposição de caminhos é realizado pela função **YenGenerateCandidates**, mostrada no Algoritmo 19, e os caminhos candidatos, obtidos através dela, armazenados no *array set* T^k , juntamente com suas respectivas informações, em L^{T^k} e A^{T^k} .

A função **YenGenerateCandidates** pode gerar um ou vários caminhos candidatos simultaneamente. Isso é feito distribuindo-se as *threads* da grade de processamento, de forma que cada conjunto atue em um vértice do caminho a ser decomposto. A motivação para isso é a alta dependência do Algoritmo de Dijkstra, o qual, conforme dito anteriormente na Seção 6.3, em nem todos os pontos necessita ou pode ser executado em paralelo por toda a grade. No entanto, em instâncias de G em que $|V|$ e $|E|$ são maiores que a quantidade de *threads* disponíveis, a sobrecarga de trabalho sobre cada *thread* em particular aumenta.

Terminado o processo iterativo de decomposição de caminhos, uma redução¹ é executada, de forma a localizar o *shortest candidate path* disponível em todo o conjunto T^k pela análise conjunta com C^{T^k} e A^{T^k} , armazenando o índice na variável *sht*. Dado o índice em *sht*, o respectivo caminho é transferido de $T_{[sht]}^k$ para $X_{[k]}$ em paralelo e marcado como indisponível em $A_{[sht]}^{T^k}$. Por questões de desempenho, evitando ser necessário toda a readequação da estrutura de memória da GPU, o que implicaria também em sincronização de todas as *threads* previamente a cada k iteração, estes caminhos permanecem na memória da GPU durante toda a execução do algoritmo. Esta forma de armazenamento é opcional

¹Reduções permitem uma varredura de forma mais eficiente de arrays utilizando para isso múltiplas *threads* e estão bem definidas na literatura CUDA em trabalhos como Che et al. (2008); Harris (2007).

Algorithm 18: Kernel Yen

```

1 Kernel KernelYen( $K, tmppc, Va, Sa, Da, Wa, Ra, s, t, tpc, smc$ )
   Input :  $K, tmppc, Va, Sa, Da, Wa, Ra, s, t, tpc, smc$ 
   Output: array set  $X$  contendo  $K$  caminhos mais curtos de  $s$  a  $t$ 
2 shared<var> tid  $\leftarrow$  getThreadID(), stride  $\leftarrow$  getStride-
   Size(), tmppc, count  $\leftarrow$  0, sht  $\leftarrow$   $\emptyset$ ;
3 shared<array de tamanho  $|V|$ >  $Thr, Flag$ ;
4 shared<array de tamanho  $|V| * |V|$ >  $ma, ca, prev, Rma$ ;
5 shared<array de tamanho  $|V| * |E|$ >  $Ena$ ;
6  $ca \leftarrow$  Dijkstra( $Va, Sa, Da, Wa, Rma, Ena, Thr, Flag, s, t, ma, ca, 1, vertex,$ 
    $tpc$ );
7 barrier();
8 if  $ca[t] \neq \infty$  then
9    $prev \leftarrow$  ExtractPreviousEdges( $Sa, Da, Wa, ca, prev$ );
10  barrier();
11  if tid = 0 then
12     $X[1], C_{[1]}^X, L_{[1]}^X \leftarrow$  ExtractPath ( $s, t, Sa, Wa, ca, prev$ );
13    count  $\leftarrow$  count + 1;
14  end
15  barrier();
16  for  $k \leftarrow 2; k < K; k \leftarrow k + 1$  do
17    barrier();
18    if count <  $K$  then
19       $T^k, C^{T^k}, L^{T^k}, A^{T^k}, \text{count} \leftarrow$  YenGenerateCandidates( $tmppc, count, k,$ 
         $Sa, Da, Wa, Ra, X, C, L, A, t$ );
20    end
21    barrier();
22    sht  $\leftarrow$  Executar redução paralela para encontrar o índice de caminho
      mais curto e disponível de  $T^k$ . Isto é feito através de  $C^{T^k}$  e  $A^{T^k}$ ;
23    barrier();
24    for th  $\leftarrow$  tid; th <  $L_{[sht]}^{T^k}$ ; th  $\leftarrow$  th + stride do
25      if th <  $L_{[sht]}^{T^k}$  then
26         $X_{[k,th]} \leftarrow T_{[sht,th]}^k$ ;
27      end
28    end
29    if tid = 0 then
30       $A_{[sht]}^{T^k} \leftarrow$  false;  $L_{[k]}^X \leftarrow L_{[sht]}^{T^k}$ ;
31    end
32  end
33 end
34 return  $X$ ;
35 end

```

Algorithm 19: DeviceFunction YenGenerateCandidates

```

1 Kernel YenGenerateCandidates(tmppc, count, k, Sa, Da, Wa, Ra, X, C, L, A, t)
   Input : tmppc, count, k, Sa, Da, Wa, Ra, X, C, L, A, t
   Output: conjunto  $T^k$  contendo candidatos atualizados para  $k$  shortest paths de  $s$  a  $t$ 
2 shared<var> tpc  $\leftarrow$  0;
3 shared or private<var> stride  $\leftarrow$  getStrideSize();
4 shared<array set contendo  $L_{[k-1]}^X$  arrays de tamanho  $|V| > Rma$ , ca, prev>;
5 shared<array set contendo  $L_{[k-1]}^X$  de tamanho  $|E| > Ena$ >;
6 private<var> tid  $\leftarrow$  getThreadId();
7 if tid = 0 then
8   | tpc  $\leftarrow$  getThreadsSetsSize(tmppc,  $L_{[k-1]}^X$ );
9 end
10 barrier();
11 for th  $\leftarrow$  tid; ( $\lceil \frac{th}{tpc} \rceil < L_{[k-1]}^X$ ) and (count  $\leq$  K); th  $\leftarrow$  th + stride do
12   | cn  $\leftarrow$   $\lceil \frac{th}{tpc} \rceil$ ;
13   | vertex  $\leftarrow$  Sa[X[k-1,cn]];
14   | Rma[cn], Ena[cn]  $\leftarrow$  YenSubgraphSetup(modulus( $\frac{tid}{tpc}$ ), tpc, k, cn, X, L, Sa, Da, Ra);
15   | barrier();
16   | ca[cn]  $\leftarrow$  Dijkstra(Va, Sa, Da, Wa, Rma, Ena, Thr, Flag, s, t, ma, ca, cn, vertex, tpc);
17   | barrier();
18   | if ca[cn,t]  $\neq \infty$  then
19     | prev[cn]  $\leftarrow$  ExtractPreviousEdges(Sa, Da, Wa, ca[cn]); barrier();
20     | p[cn], C[cn]p, L[cn]p  $\leftarrow$  ExtractPath(vertex, t, Sa, Wa, ca[cn], prev[cn]);
21     | barrier();
22     | for thc  $\leftarrow$  tid; thc < (cn +  $L_{[cn]}^p$ ); thc  $\leftarrow$  thc + tpc do
23       | if thc < cn then
24         | | T[k,cn,thc]  $\leftarrow$  X[k-1,thc];
25       | else
26         | | T[k,cn,thc]  $\leftarrow$  p[cn,(thc-cn)];
27       | end
28       | if modulus( $\frac{thc}{tpc}$ ) = 0 then
29         | | A[k,cn,thc]  $\leftarrow$  true;
30         | | count  $\leftarrow$  count + 1;
31       | end
32     | end
33   | end
34   | Rma[cn], Ena[cn]  $\leftarrow$  YenRestoreGraph(modulus( $\frac{tid}{tpc}$ ), tpc,  $|V|$ ,  $|E|$ , Rma[cn], Ena[cn]);
35 end
36 barrier();
37 return  $T^k$ ,  $C^{T^k}$ ,  $L^{T^k}$ ,  $A^{T^k}$ , count;
38 end

```

e, portanto, pode ser modificada. Ao final, retorna-se o conjunto X , que contém o resultado almejado.

Uma vez que o Algoritmo 18 fornece o número ordinal do vértice a ser decomposto, a função YenGenerateCandidates armazena o vértice no *array set* v_k , estabelece o *root path* e, a cada iteração em que são solicitados caminhos candidatos, são gerados subgrafos, que

são criados pela função `YenSubgraphSetup`, mostrada no Algoritmo 21. Esta função, por sua vez, modifica em paralelo, para verdadeiro ou falso, os valores nos *array sets* *Rma* e *Ena*. Caso a função esteja trabalhando na modalidade de decompor vários vértices de uma só vez, para cada grupo de *threads* é criado um subgrafo, para, posteriormente, executar o Algoritmo de Dijkstra e obter a *spur path*. A união do *root path* e o *spur path* produz um caminho possível candidato, que somente é armazenado e retornado em T^k , caso não ocorra repetição de nenhum vértice.

Algorithm 20: DeviceFunction `getThreadsSetsSize`

```

1 Kernel getThreadsSetsSize(tmppc, nsets)
   Input : tmppc, nsets
   Output: variável tps contendo quantos tópicos estão disponíveis por candidato
2 private<var> grid  $\leftarrow$  getGridSize();
3 if tmpps  $\neq \emptyset$  then
4   if grid  $\geq (nsets * |V|)$  then
5     tps  $\leftarrow \lfloor \frac{grid}{nsets * |V|} \rfloor$ ;
6   else if  $\lfloor \frac{grid}{nsets} \rfloor > (grid * tmpps)$  then
7     tps  $\leftarrow \lfloor \frac{grid}{nsets} \rfloor$ ;
8   else
9     tps  $\leftarrow \lceil grid * tmppc \rceil$ ;
10  else if  $\frac{grid}{nsets} > 1$  then
11    if  $\frac{grid}{nsets} > (nsets * |V|)$  then
12      tps  $\leftarrow |V|$ ;
13    else
14      tps  $\leftarrow \lceil \frac{grid}{nsets} \rceil$ ;
15  return tps;
16 end

```

O tamanho de cada grupo de *threads* que analisam simultaneamente cada vértice do caminho decomposto é obtido pela função `getThreadsSetsSize`, mostrada no Algoritmo 20. Ela é utilizada previamente a cada chamada da função de geração de decomposição de caminhos e armazena o resultado na variável *tps*. O valor a ser alocado em *tps* é dado de acordo com a quantidade de *threads* disponíveis e a quantidade de vértices a serem decompostos no caminho. A função `getThreadsSetsSize` recebe, via parâmetro, o percentual mínimo de *threads* a serem utilizadas em cada vértice. Optando-se por utilizar todas as *threads* de uma só vez sucessivamente em cada vértice, o algoritmo opera no formato original proposto por seu autor (Yen, 1971). No entanto, ressalta-se que um vértice necessita no máximo de $|V|$ *threads* para ser processado, dada a limitação imposta pelo Algoritmo de Dijkstra. Uma quantidade excessivamente maior pode causar perda de desempenho, uma vez que poderá haver inutilização do trabalho de *threads*, que apenas aguardarão o fim do processamento de outras em uma barreira de sincronização, ou, ainda, podem ser disparadas após o trabalho útil já ter sido concluído por outras, aumentando o tempo de espera

por sincronização. Por outro lado, um número muito pequeno de *threads* pode causar um aumento desnecessário no tempo de processamento de arestas dos vértices também pelo Algoritmo de Dijkstra. Assim sendo, caso não existam *threads* suficientes, pois podem ser alocadas até pelo menos uma *thread* por vértice ou divididas até uma quantidade percentual mínima, especificada na variável *tmppc*, é feito um *stride loop* para que o caminho seja completamente coberto².

A cada iteração em que é solicitado um caminho candidato, é gerado um subgrafo, que é criado através da função **YenSubgraphSetup** mostrada no Algoritmo 21, que, por sua vez, modifica em paralelo, para verdadeiro ou falso, os valores nos vetores *Rma* e *Ena*, de acordo com as especificações advindas do Algoritmo de Yen. Feito isso, executa-se novamente em paralelo o Algoritmo de Dijkstra, e extrai-se o *path* encontrado no subgrafo do i^{th} vértice do *shortest path* $(k-1)^{th}$ até t . Um caminho candidato é gerado, unindo-se, em paralelo, os i^{th} vértices do $(k-1)^{th}$ *path* e o caminho localizado do i^{th} vértice do *path* $(k-1)^{th}$ até t sendo armazenado e retornado em T .

Ao final da execução da função **YenGenerateCandidates**, mostrada no Algoritmo 19, o grafo G tem adicionadas novamente todas as suas arestas e vértices originais. Isso é realizado pela função **YenRestoreGraph**, mostrada no Algoritmo 22, que trabalha de forma trivial, apenas alterando todos os valores de *Rma* e *Ena* novamente para *true*.

²Para que um *warp* não execute operações divergentes e por tal motivo o algoritmo não tenha problemas de sincronização de *threads* ao ser executado, é conveniente que o valor da variável *tps*, retornada pela função **device** 20, seja sempre um valor divisível pelo tamanho de um *warp* completo. Por exemplo, $tps \leftarrow tps + \text{modulus}(tps, \text{WARP_SIZE})$, sendo o *WARP_SIZE* um valor constante e que contém a quantidade de *threads* de um *warp* – atualmente 32 *threads*.

Algorithm 21: DeviceFunction YenSubgraphSetup

```

1 Kernel YenSubgraphSetup(tid, stride, k, spurnode, X, L, Sa, Da, Ra)
   Input : tid, stride, k, spurnode, X, L, Sa, Da, Ra
   Output: Arrays Rma e Ena contendo os vértices ativos e inativos e arestas de G
2 shared<array de tamanho  $|V|$ > Rma;
3 shared<array de tamanho  $|E|$ > Ena;
4 shared<var> flag  $\leftarrow$  false;
5 for i  $\leftarrow$  0; i < k; i  $\leftarrow$  i + 1 do
6     flag  $\leftarrow$  false;
7     barrier();
8     for th  $\leftarrow$  tid; (th < spurnode) and (th <  $L_{[i]}^X$ ) and (flag = false); th  $\leftarrow$  th +
       stride do
9         for  $X_{[k-1,th]} \neq X_{[i,th]}$  do
10            flag  $\leftarrow$  true;
11        end
12    end
13    barrier();
14    if (flag = false) and (tid = 0) then
15        Ena[ $X_{[i,spurnode]}$ ]  $\leftarrow$  false;
16        Ena[ $Ra[X_{[i,spurnode]}]$ ]  $\leftarrow$  false;
17    end
18 end
19 for th  $\leftarrow$  tid; th <  $L_{[k-1]}^X$ ; th  $\leftarrow$  th + stride do
20     for i  $\leftarrow$  0; i < spurnode; i  $\leftarrow$  1 do
21         if ( $Sa[th] = Sa[X_{[k-1,i]}]$ ) or ( $Sa[th] = Da[X_{[k-1,i]}]$ ) and
           ( $Sa[X_{[k-1,i]}] \neq X_{[k-1,spurnode]}$ ) then
22             Rma[ $Sa[th]$ ]  $\leftarrow$  false;
23         end
24         if ( $Da[th] = Sa[X_{[k-1,i]}]$ ) or ( $Da[th] = Da[X_{[k-1,i]}]$ ) and
           ( $Da[X_{[k-1,i]}] \neq X_{[k-1,spurnode]}$ ) then
25             Rma[ $Da[th]$ ]  $\leftarrow$  false;
26         end
27     end
28 end
29 barrier();
30 return Rma, Ena;
31 end

```

Algorithm 22: DeviceFunction YenRestoreGraph

```

1 Kernel YenRestoreGraph(tid, stride,  $|V|$ ,  $|E|$ , Rma, Ena)
   Input : tid, stride,  $|V|$ ,  $|E|$ , Rma, Ena
   Output: Arrays Rma e Ena contendo todos os vértices e arestas de G ativos
2   for th  $\leftarrow$  tid; th <  $|V|$ ; th  $\leftarrow$  th + stride do
3     Rma[th]  $\leftarrow$  true;
4   end
5   for th  $\leftarrow$  tid; th <  $|E|$ ; th  $\leftarrow$  th + stride do
6     Ena[th]  $\leftarrow$  true;
7   end
8   barrier();
9   return Rma, Ena;
10 end

```

7.5 Paralelização CUDA do Algoritmo MPS

Através de melhorias realizadas no Algoritmo de Yen, [Martins et al. \(1997, 1999\)](#) introduzem dois algoritmos. O primeiro é uma generalização do Algoritmo de Yen, que, posteriormente, recebeu aditivos que reorganizam G para acelerar a obtenção de caminhos candidatos, resultado, assim, no segundo algoritmo, denominado então Algoritmo MPS. O Algoritmo MPS possui complexidade de tempo $O(m \log n + Kn)$ quando analisado o pior caso e possui *speed up* maior que o Algoritmo de Yen, como pode ser visto em ([Hershberger et al., 2007](#); [Feng, 2014](#)).

Propostos originalmente em ([Martins et al., 1999](#)), a generalização do Algoritmo de Yen e o Algoritmo MPS apresentam características que os colocam como bons candidatos a estratégias de solução paralela baseada em CUDA para o subproblema *K-Shortest Paths* (KSLP). Tais características advêm do fato de realizarem inicialmente um pré-processamento, o qual estabelece os *shortest paths* de todos os vértices em G para o vértice de destino t , sendo então aproveitados posteriormente como *spur paths* (ou seja, um caminho de um nó v até t em G) para serem unidos aos *root path* (ou seja, caminho entre os primeiros v nós do *shortest path* corrente) e compor um caminho candidato, evitando a utilização recorrente do Algoritmo de *Shortest Path* (como é feito no Algoritmo de Yen), o qual é inerentemente sequencial em algumas etapas, compensando o tempo extra demandado na fase de pré-processamento.

As diferenças entre a generalização do Algoritmo de Yen e o Algoritmo MPS residem na adição de uma etapa na fase de pré-processamento do segundo, a qual leva a facilitação no processamento na busca por soluções candidatas, dado o fato de uma classificação prévia das arestas de cada vértice de G . Para tal, após obter-se os caminhos de menor custo entre cada vértice de G até o vértice de destino, as arestas de G são classificadas de acordo com o custo reduzido, organizando G em um formato chamado de *Forward Sorted Star Form*. O formato *Forward Sorted Star Form*, bem como o custo, são vistos em detalhes nas seções [7.5.3.2](#) e [7.5.3.1](#).

7.5.1 Algoritmo MPS original e mapa de pontos de paralelização

O *flowchart* da Figura [7.3](#) apresenta a versão original do Algoritmo MPS e aponta os seus pontos de paralelização explorados nesta tese. Não há necessidade de exibir neste mesmo texto um *flowchart* da generalização do Algoritmo de Yen, uma vez que este apenas elimina a etapa de organização de G em *Forward Sorted Star Form*.

Como vê-se na Figura [7.3](#), ao ser iniciado, o Algoritmo MPS gera, na fase de pré-processamento, a árvore de caminhos T_t^* , utilizando um algoritmo de *single source shortest path*, tendo o vértice t como origem, todos os demais vértices como destino, e, posteriormente, invertendo a direção das arestas de cada caminho.

Após a fase de pré-processamento, o *shortest path* de s a t é extraído de T_t^* e armazenado em X , passando a ser, assim, o primeiro dos K *shortest paths* a serem encontrados. Inicia-se, então, de forma iterativa, o processo de geração de caminhos candidatos. A cada iteração é designado um vértice v_i a partir do $(k - 1)^{th}$ caminho disponível em X , o qual passa também a ser o caminho corrente a ser decomposto, servindo de base para a formação de outros candidatos a serem armazenados em T^k . Um *root path* é composto pelas primeiras v_i arestas do caminho decomposto corrente. O *spur path* é obtido após a análise, através de uma redução de todas as arestas que partem do vértice v_i e encontrada a mais curta

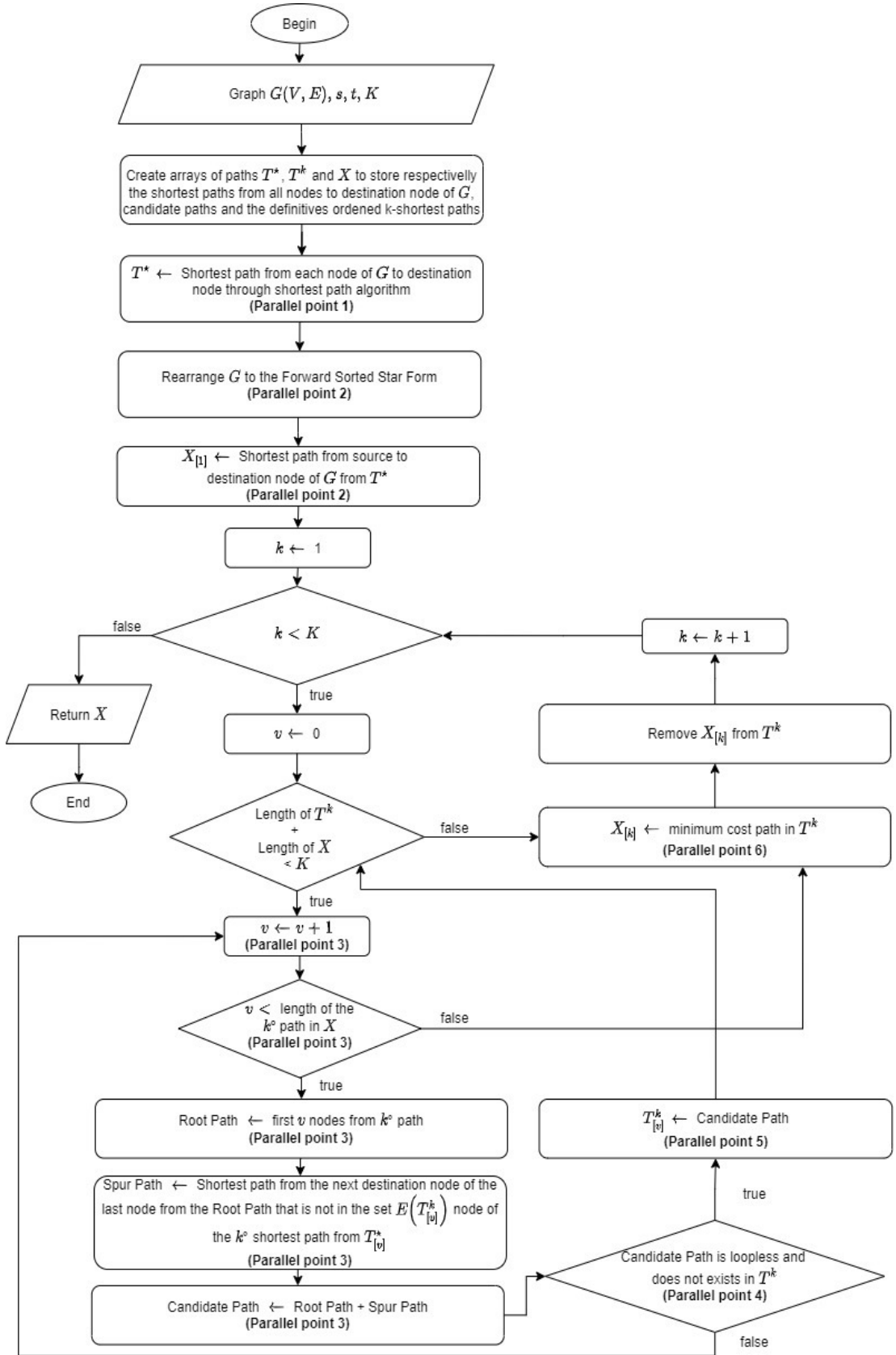


Fig. 7.3: Flowchart do Algoritmo MPS.

não utilizada nos caminhos em T^k . Uma vez sabido o vértice de destino de tal aresta, o algoritmo faz a extração da *spur path* dele até t a partir de T^* . A união do *root path* e *spur path* compõe o novo caminho candidato. Dado que não há garantias de que seja um *loopless path*, é, então, verificado, antes de ser armazenado em T^k . O processo se repete até que todos os vértices de v_i até t do $(k-1)^{th}$ caminho sejam analisados. Antes de iniciar a próxima k iteração, o *shortest path* disponível em T^k é encontrado via redução, a qual faz isso analisando os *arrays sets* C^{T_k} e A^{T_k} e encontra seu índice, para que, em seguida, seja transferido para X e marcado como indisponível em $A_{[shd]}^{T_k}$. O processo de geração de caminhos candidatos encerra-se após K iterações ou quando a soma dos caminhos em X e T^k atinge o valor de K . A geração de soluções candidatas proposta aqui é capaz de decompor um ou vários vértices simultaneamente $(k-1)^{th}$ caminho, sendo apresentada no Algoritmo 24.

Os itens presentes na lista a seguir detalham os pontos de paralelização do Algoritmo MPS explorados nesta tese, bem como indicam seus respectivos kernels:

- (i) Os *kernels* 23 (o qual trata da paralelização da generalização do Algoritmo de Yen) ou 28 (que trata a paralelização do Algoritmo MPS) inicializam variáveis e criam/modificam estruturas de dados bem como realizam o pré-processamento necessário a cada iteração de geração de soluções candidatas para se obtenham os *K-Shortest Paths*. Estes *Kernels* também são responsáveis pela chamada de outros *kernels* necessários ao algoritmo;
- (ii) Apontada no *flowchart* da Figura 7.3 como **parallel point 1**, uma versão paralela CUDA do Algoritmo de Dijkstra baseada em Singh e Khare (2013) é utilizada para obtenção da árvore de caminhos T^* , bem como a primeira das *k-shortest paths*. O pseudocódigo é apresentado nos Algoritmos 11, 12, 13, 14 e 15;
- (iii) Mostrado no *flowchart* da Figura 7.3 como **parallel point 2**, a reorganização de G em *Forward Sorted Star Form* em paralelo reúne uma série de observações vistas nas Seções 7.5.3.2 e 7.5.3.1, além do Algoritmo 27;
- (iv) A quantidade massiva de *threads* das GPUs é capaz de gerar em paralelo diversas soluções candidatas. Para tal, as *threads* são agrupadas e alocadas para cada vértice disponível no processo iterativo de decomposição da $X_{[k-1]}$ *shortest path*, de acordo com o *kernel* 25. Mesmo não estando presente visualmente no *flowchart* da Figura 7.3, este *kernel* é o primeiro passo executado para o **parallel point 3**;
- (v) Uma vez alocadas as *threads* para cada um dos múltiplos candidatos a serem gerados em paralelo e estabelecidos seus respectivos subgrafos, os *kernels* 24 ou 29 geram cada *root path* e os unem aos *spur paths*, finalizando a solução candidata de cada grupo de *threads* designado para tal. Estes passos são apontados no *flowchart* da Figura 7.3 como **parallel point 3**;
- (vi) Como última etapa do processo iterativo de decomposição de $X_{[k-1]}$, os *kernels* 23 e 28 verificam se cada *candidate path* não existe em T^k e, acaso não existam, as armazenam em T^k . Apesar do processo de verificar a repetição de cada caminho precisar ser realizado um a um, sequencialmente, cada verificação é paralelizada. Soluções candidatas inexistentes são armazenadas em T^k para posterior classificação

e cópia para X . Estas etapas são exibidas no *flowchart* da Figura 7.3 como **parallel point 4 e 5**;

- (vii) Após a descoberta do *shortest path* presente em T^k via redução, tal *path* é transferido em paralelo para $X_{[k]}$ pelos *kernels* 23 ou 28 de forma paralela e removida de T^k . Tal operação é exibida no *flowchart* da Figura 7.3 como **parallel point 6**;

7.5.2 Paralelização CUDA da generalização do Algoritmo de Yen

Uma proposta de paralelização em CUDA para a generalização do Algoritmo de Yen é exibida nos algoritmos 23, 24 e 25, sendo detalhado em seguida.

O Algoritmo 23 gera, na fase de pré-processamento, a árvore de caminhos T_t^* , utilizando um Algoritmo de *single source shortest path*, tendo o vértice de destino t como origem, todos os demais vértices como destino e, posteriormente, invertendo a ordem das arestas de cada caminho.

Após a fase de pré-processamento, o *shortest path* de s a t é extraído de T_t^* e armazenado em X , passando a ser, assim, o primeiro dos K *shortest paths* a serem encontrados. Inicia-se, então, de forma iterativa, o processo de geração de caminhos candidatos. A cada iteração é designado um vértice v_i a partir do $(k-1)^{th}$ caminho disponível em X , o qual passa também a ser o caminho corrente a ser decomposto, servindo de base para formação de outros candidatos a serem armazenados em T^k . Um *root path* é composto pelos vértices de s até v_i do caminho decomposto corrente. O *spur path* é obtido após a análise, através de uma redução de todas as arestas que partem do vértice v_i e encontrada a mais curta não utilizada nos caminhos em T^k . Uma vez sabido o vértice de destino de tal aresta, o algoritmo faz a extração da *spur path* dele até t a partir de T^* . A união do *root path* e *spur path* compõe o novo caminho candidato. Dado que não há garantias de que seja um *loopless path*, é, então, verificado, antes de ser armazenado em T^k . O processo se repete até que todos os vértices de v_i até t do $(k-1)^{th}$ caminho sejam analisados. Antes de iniciar a próxima k iteração, o *shortest path* disponível em T^k é encontrado via redução, a qual faz isso analisando os *arrays sets* C^{T_k} e A^{T_k} e encontra seu índice, para que, em seguida, seja transferido para X e marcado como indisponível em $A_{[sh]}^{T_k}$. O processo de geração de caminhos candidatos encerra-se após K iterações ou quando a soma dos caminhos em X e T^k atinge o valor de K . A geração de soluções candidatas proposta aqui é capaz de decompor um ou vários vértices simultaneamente $(k-1)^{th}$ caminho, sendo apresentada no Algoritmo 24.

Na função 24 também são inicialmente divididas e alocadas as *threads* para vértices do caminho decomposto, de acordo com os parâmetros oferecidos para a função 25. O trabalho de cada conjunto de *threads* é combinar o *root path* corrente e o *spur path* respectivo de T^* , após selecionada a menor aresta que parte de v_i . Alocar todas as *threads* para a geração de um único caminho candidato aumenta o desempenho em grandes instâncias de G , ou seja, nas quais os vértices possuem grandes quantidades de arestas e os possíveis caminhos também são extensos, processando simultaneamente maior quantidade destes. Por outro lado, do ponto de vista prático, mesmo em instâncias com grandes valores de $|V|$ e $|E|$ nem sempre a quantidade de arestas, assim como nem todo caminho, supera o tamanho da grade, o que pode causar desnecessariamente o ócio de *threads* em uma barreira de sincronização após a obtenção de cada candidato.

O tamanho de cada grupo de *threads* que analisam simultaneamente cada vértice do

Algorithm 23: DeviceFunction: Yen Generalization

Input : $K, Sa, Da, Wa, Ra, X, L, C, A, s, t$
Output: um conjunto X contendo os K -Shortest Paths de s a t em G

```

1  private<var>  $tid \leftarrow \text{getThreadId}()$ ;
2  private<var>  $k \leftarrow 1, flag \leftarrow false, stride \leftarrow \text{getStrideSize}()$ ;
3  shared<arrays de tamanho  $|V| > ca, prev$ ;
4   $ca \leftarrow \text{Dijkstra}(Va, Sa, Da, Wa, \emptyset, \emptyset, t, \emptyset)$ ;
5  if  $ca[t] \neq \infty$  then
6       $prev \leftarrow \text{ExtractPreviousEdges}(Sa, Da, Wa, ca)$ ;
7      barrier();
8      for  $th \leftarrow tid; th < |V|; th \leftarrow th + stride$  do
9           $T_{[th,t]}^*, C_{[th,t]}^{T^*}, L_{[th,t]}^{T^*} \leftarrow \text{ExtractPath}(th, t, Sa, Wa, ca, prev)$ ;
10     end
11 end
12 barrier();
13 if  $tid = 0$  then
14      $L_{[k]}^X \leftarrow L_{[s,t]}^{T^*}$ ;
15      $C_{[k]}^X \leftarrow C_{[s,t]}^{T^*}$ ;
16      $L_{[k]}^{T^k} \leftarrow L_{[s,t]}^{T^*}$ ;
17      $C_{[k]}^{T^k} \leftarrow C_{[s,t]}^{T^*}$ ;
18 end
19 for  $th \leftarrow tid; th < L_{[s,t]}^{T^*}; th \leftarrow th + stride$  do
20      $X_{[k,th]} \leftarrow T_{[s,t,th]}^*$ ;
21      $T_{[k,th]}^k \leftarrow T_{[s,t,th]}^*$ ;
22 end
23 while  $(k < K)$  and  $(flag = false)$  do
24      $flag \leftarrow true$ ;
25     barrier();
26     Call  $T^k, C^{T^k}, L^{T^k}, A^{T^k}, count \leftarrow \text{Yen Generalization Generate Candidates}(tmppc, k, Va, Sa,$ 
        $Da, Wa, Ra, X, L, C, A, s, t)$ ;
27     barrier();
28     if  $tid = 0$  then
29          $k \leftarrow k + 1$ ;
30     end
31      $sht \leftarrow \text{redução para encontrar o shortest path disponível em } T_{[k]} \text{ através da análise de } A^{T^k} \text{ e}$ 
        $C^{T^k}$ ;
32     barrier();
33     if  $tid = 0$  then
34          $L_{[k]}^X \leftarrow L_{[sht]}^{T^k}$ ;
35     end
36     for  $th \leftarrow tid; th < L_{[sht]}^{T^k}; th \leftarrow th + strideSize$  do
37          $X_{[k,th]} \leftarrow T_{[sht,th]}^k$ ;
38     end
39 end
40 return  $X$ ;

```

caminho decomposto é obtida através da função 25. Ela é utilizada previamente a cada chamada da função de geração de decomposição de caminhos e armazena o resultado na variável tps . O valor a ser alocado em tps é dado de acordo com a quantidade de *threads*

Algorithm 24: DeviceFunction: Generate Candidates

```

Input :  $tmppc, K, Sa, Da, Wa, Ra, X, L, C, A, s, t$ 
Output: um conjunto  $X$  contendo os  $K$ -Shortest Paths de  $s$  a  $t$  em  $G$ 
1 private<var> tid  $\leftarrow$  getThreadId();
2 shared<var> tpc  $\leftarrow$  0, smc  $\leftarrow$  0, stride  $\leftarrow \frac{L_{[k-1]}^X}{tpc}$ , gridSize  $\leftarrow$  getGridSize();
3 shared<array set of size  $|V| * L_{[k]}^X - k$ > loopless  $\leftarrow$  true;
4 if tid = 0 then
5   | tpc  $\leftarrow$  getThreadsSetsSize( $tmppc, L_{[k]}^X - k$ );
6   | smc  $\leftarrow \lfloor \frac{gridSize}{tpc} \rfloor$ ;
7 end
8 barrier();
9 private<var> cn  $\leftarrow \lceil \frac{th}{tpc} \rceil$ ;
10 for th  $\leftarrow \lceil \frac{tid}{tpc} \rceil$ ; th  $\leq L_{[k]}^X - k$ ; th  $\leftarrow$  th + smc do
11   | private<var> vertex  $\leftarrow Sa[X_{[k,cn]}]$ ;
12   | edge  $\leftarrow$  usando apenas threads alocados para  $cn$ , execute a redução para encontrar o índice da aresta mais
      | curta enraizada em  $\langle vertex, j \rangle$ , cujo custo é minimizado em  $Wa_{\langle vertex, j \rangle} + C_{[j]}^{T^*}$  e não está em  $A_{[cn]}^{T^k}$ ;
13   | barrier();
14   | for thc  $\leftarrow$  modulus( $\frac{tid}{tpc}$ ); thc  $\leq k + L_{[Da[edge]]}^{T^*}$  and loopless[cn] = true; thc  $\leftarrow$  thc + tpc do
15     | if (thc  $\leq k$ ) and (loopless[cn] = true) then
16       | |  $q_{[k,cn,thc]} \leftarrow X_{[k,thc]}$ ;
17       | |  $count_{[k,cn,Sa[X_{[k,thc]}]]} \leftarrow count_{[k,cn,Sa[X_{[k,thc]}]]} + 1$ ;
18       | | if  $count_{[k,cn,Sa[X_{[k,thc]}]]} > 1$  then
19       | | |  $< loopless[cn] \leftarrow$  false; $>$ 
20       | | end
21       | | else if (thc - k)  $< L_{[Da[edge]]}^{T^*}$  then
22       | | |  $q_{[k,cn,thc]} \leftarrow T_{[Da[edge],(thc-k)]}^{T^*}$ ;
23       | | |  $count_{[k,cn,Sa[T_{[Da[edge],(thc-k)]}^{T^*}]]} \leftarrow count_{[k,cn,Sa[T_{[Da[edge],(thc-k)]}^{T^*}]]} + 1$ ;
24       | | | if  $count_{[k,cn,Sa[T_{[Da[edge],(thc-k)]}^{T^*}]]} > 1$  then
25       | | | |  $< loopless[cn] \leftarrow$  false; $>$ 
26       | | | end
27     | end
28     | barrier();
29     | if (loopless[cn] = true) and ( $q_{[k,cn]}$  not in  $T^k$ ) then
30       | | if modulus( $\frac{tid}{tpc}$ ) = 0 then
31       | | |  $L_{[(k+1),cn]}^{T^k} \leftarrow k + L_{[Da[edge]]}^{T^*}$ ;
32       | | | count  $\leftarrow$  count + 1;
33       | | end
34       | | barrier();
35       | | for thc  $\leftarrow$  modulus( $\frac{tid}{tpc}$ ); (th  $< L_{[(k+1),cn]}^{T^k}$ ) and (loopless[cn] = true; th  $\leftarrow$  th + tpc) do
36       | | |  $T_{[(k+1),cn,thc]}^k \leftarrow q_{[k,cn,thc]}$ ;
37       | | end
38     | end
39     | cn  $\leftarrow$  cn + smc;
40 end
41 return  $T^k, C^{T^k}, L^{T^k}, A^{T^k}, count$ ;

```

disponíveis e a quantidade de vértices a serem decompostos no caminho. A função 25 recebe, via parâmetro, o percentual mínimo de *threads* a serem utilizadas em cada vértice. Optando-se por utilizar todas as *threads* de uma só vez sucessivamente em cada vértice, o algoritmo opera no formato original proposto por seu autor (Yen, 1971). No entanto, ressalta-se que um vértice, por necessitar de no máximo $|V|$ *threads* para ser processado, dado a limitação imposta pelo Algoritmo de Dijkstra, uma quantidade excessivamente

Algorithm 25: Kernel 7: getThreadsSetsSize

Input : $tmpps, nsets$
Output: variável tps contendo a quantidade de threads disponíveis por candidato

```

1 private<var> grid  $\leftarrow$  gridSize();
2 if  $tmpps \neq \emptyset$  then
3   if  $grid \geq (nsets * |V|)$  then
4      $tps \leftarrow \lfloor \frac{grid}{nsets * |V|} \rfloor$ ;
5   else if  $\lfloor \frac{grid}{nsets} \rfloor > (grid * tmpps)$  then
6      $tps \leftarrow \lfloor \frac{grid}{nsets} \rfloor$ ;
7   else
8      $tps \leftarrow \lceil grid * tmpps \rceil$ ;
9 else if  $\frac{grid}{nsets} > 1$  then
10   if  $\frac{grid}{nsets} > nsets * |V|$  then
11      $tps \leftarrow |V|$ ;
12   else
13      $tps \leftarrow \lceil \frac{grid}{nsets} \rceil$ ;
14 return  $tps$ ;
```

maior pode causar perda de desempenho, uma vez que poderá haver inutilização do trabalho de *threads* que apenas aguardarão o fim do processamento de outras em uma barreira de sincronização, ou, ainda, podem ser disparadas após o trabalho útil já ter sido concluído por outras, aumentando o tempo de espera por sincronização. Por outro lado, um número muito pequeno de *threads* pode causar um desnecessário aumento no tempo de processamento de arestas dos vértices também pelo Algoritmo de Dijkstra. Assim sendo, caso não existam *threads* suficientes, podem ser alocadas até pelo menos uma *thread* por vértice ou divididas, até uma quantidade percentual mínima especificada na variável *tmppc* e feito um *stride loop*, para que o caminho seja completamente coberto.

7.5.3 Otimização da estrutura de dados

A cada iteração em um vértice v_i de um caminho decomposto na geração de candidatos do Algoritmo 24, uma redução faz-se necessária para localizar a menor aresta do vértice. O funcionamento pode ser melhorado fazendo-se uma reorganização na disposição das arestas de G .

7.5.3.1 Reduced Cost

Localizar a mais curta aresta de um vértice não garante que seu caminho até t seja o menor entre elas. Por tal motivo, é mais eficiente que sejam organizadas de acordo com seu *reduced cost*.

O *reduced cost* de uma aresta $\langle i, j \rangle$ é dado por $\bar{c}^{\langle i, j \rangle} = \pi(j) - \pi(i) + Wa[\langle i, j \rangle]$, no

qual $\pi(j)$ e $\pi(i)$ representam o custo do respectivo caminho do vértice até t em T_t^* . Uma vez calculado o custo reduzido das arestas, elas devem ser dispostas em G em ordem crescente nos seus respectivos vértices. Tal técnica foi utilizada pela primeira vez em Eppstein (1998) e repetida com sucesso em Martins et al. (1999), mostrando um aumento de desempenho que supera o tempo demandado para a reorganização.

7.5.3.2 Formato *Forward Sorted Star Form*

O formato *Forward Sorted Star Form* é o grafo G com todas as suas arestas ordenadas de maneira crescente em seus respectivos vértices, de acordo com o seu custo reduzido (Martins et al., 1999). Dado que cada vértice possui um conjunto de arestas a serem ordenadas de forma independente dos demais, a sua ordenação pode utilizar estratégias de solução para o problema *Segmented Sort*. Isso permite que, na fase de elaboração de caminhos candidatos, possam ser evitadas diversas execuções de reduções na busca da menor aresta de v_i que, uma vez ordenadas, asseguram que sempre a primeira disponível é a aresta correta a ser utilizada. De fato, deve ser ponderado se o custo da reorganização de G em sua *forward sorted star form* é compensado posteriormente de modo a superar o tempo demandado pelas reduções.

Benchmarks para o ordenação com utilização de paralelismo dinâmico são demonstrados em (Bergstrom e Reppy, 2012). Utilizando a função `radix sort` da biblioteca Thrust (Nvidia, 2018), os autores discutem também o problema *Segmented Sorting*, mas não apresentam os resultados de testes de desempenho para o problema.

Baseado no Algoritmo Merge Sort, os mesmos criadores da *Modern GPU* (MGPU) *library* (Baxter, 2013) apresentam, em (Baxter), um algoritmo para *Segmented Sort*. Apesar da falta de informações detalhadas, a página mostra resultados de desempenho com diferentes entradas.

Desenvolvida por pesquisadores da NVIDIA, a biblioteca CUB (Research) possui um algoritmo para *Segmented Sort*. Tal algoritmo trabalha executando, de forma concorrente, diversas chamadas de uma versão paralela do `radix sort` para cada segmento do *array*. Apesar de bons resultados em algumas situações, algumas entradas causam perda de desempenho. Não há estudos disponíveis a respeito da metodologia utilizada, o que prejudica concluir qual é o motivo.

CUDA também disponibiliza diversos exemplos de algoritmos de ordenação que podem ser chamados de forma concorrente para cada segmento do *array*, sendo necessária, então, a reorganização da grade de processamento a cada iteração, implicando em alta quantidade de barreiras de sincronização que deixam *threads* em ócio. Em Nvidia (2018) há exemplos de utilização de Bitonic Sort, Merge Sort, Odd Even Merge Sort e Quick Sort. Um destaque a ser feito sobre a implementação do Bitonic Sort é sua capacidade de ordenar segmentos de um *array*, desde que sejam fornecidos o *array* com os elementos a serem ordenados, o número de segmentos e o tamanho de cada um.

Estratégias de solução paralelas para o *Segmented Sort* são vistas em Hou et al. (2017); Schmid et al. (2016). Em Hou et al. (2017) é utilizado um método que aproveita os registradores dos SMs; no entanto, a entrada acaba tendo seu tamanho limitado pela quantidade de memória local disponível, o que inviabiliza seu uso para grandes instâncias do problema tratado neste trabalho. Em Schmid et al. (2016) é apresentada uma estratégia na qual todos os segmentos do *array* podem ser ordenados de uma vez, desde que anteriormente cada valor do *array* seja processado para mantê-lo dentro do respectivo segmento após a

Algorithm 26: DeviceFunction: *MPS Forward Sorted Star Form Nested Sort Call*

Input : G, K, s, t
Output: um conjunto X contendo os K -Shortest Paths de s a t em G

```

1 shared<var> stride  $\leftarrow$  gridSize();
2 for  $th \leftarrow tid; th < |V|; th \leftarrow th + stride$  do
3   |  $\overline{C}_{[th]}^{Ea} \leftarrow C_{[Da[th]]}^{T*} - C_{[Ea[th]]}^{T*} + C_{[th]}^{Wa};$ 
4 end
5 barrier();
6 shared  $tpv \leftarrow$  getThreadsSetsSize( $tmppc, |E|$ );
7 stride  $\leftarrow$  getMaximumAvailableBlocksOfSize( $tpv$ );
8 for  $th \leftarrow tid; th < |V|; th \leftarrow th + stride$  do
9   | Call< < < 1,  $tpv$  > > o algoritmo de ordenação paralela escolhido para
   |   ordenar o segmento de arestas correspondente do vértice  $th$  no array  $Ea$  de
   |   acordo com os valores armazenados em  $\overline{C}^{Ea}$ .
10 end
11 barrier();
```

execução do algoritmo.

Escolhemos aqui testar duas formas de organizar o grafo G em *Sorted Forward Star Form*:

- (i) ordenação de segmentos por execução do algoritmo de ordenação por paralelismo dinâmico, sendo demonstrado no Algoritmo 26; e
- (ii) estratégia de Schmid et al. (2016) apresentada no Algoritmo 27 e exemplificado na Figura 7.4.

7.5.3.3 Dynamic Parallelism Based Strategy

O Algoritmo 26 inicialmente calcula o *reduced cost* de cada aresta e armazena-o na respectiva posição no *array set* $\overline{C}_{[th]}^{Ea}$. Posteriormente, através da função 25, a variável tpv passa a conter o número de *threads* que estarão disponíveis para o processamento de cada vértice, e a variável $stride$ passa a conter o número máximo de blocos simultâneos que podem ser executados. A cada iteração do laço que varre todos os vértices para ordenar suas arestas, um algoritmo de ordenação paralelo é executado.

7.5.3.4 Graph Edges Sort Through Parallel Fix Strategy

A *Parallel Fix Strategy*, desenvolvida em Schmid et al. (2016), consiste em pré-processar todo o *array* que representa os segmentos a serem ordenados, de forma tal que somente uma execução do algoritmo de ordenação seja necessária, podendo utilizar toda a grade disponível sem necessidade de sua reorganização e, após isso, caso necessário, pós processá-lo novamente, devolvendo o valor original a cada elemento. O Algoritmo 27 exibe de forma detalhada todo o processo.

Conforme visto no Algoritmo 27, o vetor Da é dividido em segmentos. Cada um deles representa as arestas do respectivo vértice, referindo-se, para tal divisão, ao vetor Va . Após

Algorithm 27: DeviceFunction: *MPS Forward Sorted Star Form Parallel Fix Strategy*

Input : G, K, s, t
Output: um conjunto X contendo os K -Shortest Paths de s a t em G

```

1 shared<var> msb  $\leftarrow \emptyset$ ;
2 for th  $\leftarrow$  tid; th <  $|E|$ ; th  $\leftarrow$  th + strideSize do
3   |  $\overline{C}_{[th]}^{Ea} \leftarrow C_{[Da[th]]}^{T*} - C_{[Ea[th]]}^{T*} + C_{[th]}^{Wa}$ ;
4 end
5 barrier();
6 msb  $\leftarrow$  redução para encontrar o maior valor em  $\overline{C}^{Ea}$ ;
7 barrier();
8 if tid = 0 then
9   | msb  $\leftarrow \log_2 msb$ ;
10 end
11 barrier();
12 for th  $\leftarrow$  tid; th <  $|E|$ ; th  $\leftarrow$  th + strideSize do
13   | fix  $\leftarrow Ea_{[th]} + shiftLeft(Sa_{[th]}, msb)$ ;
14   |  $\overline{C}_{[th]}^{Ea} \leftarrow \overline{C}_{[th]}^{Ea} + \text{fix}$ ;
15 end
16 barrier();
17 Call o algoritmo de ordenação paralela escolhido para ordenar o array  $Ea$  de acordo
    com os valores armazenados em  $\overline{C}^{Ea}$ . Use todas as threads disponíveis.

```

Vertex = [Binary]	1 = [1]			2 = [10]				3 = [11]		4 = [100]					...
Da	2	3	4	1	4	5	6	1	4	1	2	3	5	7	...
$\overline{C}_{[cl,p]}^{Ea}$ (divided by 1000 and rounded up)	21	20	48	21	50	19	20	20	22	48	50	22	36	52 (highest value)	...
Binary of $\overline{C}_{[cl,p]}^{Ea}$	10101	10100	110000	10101	110010	10011	10100	10100	10110	110000	110010	10110	100100	110100	...
1 st - MSB Index	$\lceil \log_2^{21} \rceil \rightarrow 6$														
2 nd - Pre sort	1010101	1010100	1110000	10010101	10110010	10010011	10010100	11010100	11010110	100110000	100110010	100010110	100100100	100110100	...
3 rd - Sorted	1010100	1010101	1110000	10010011	10010100	10010101	10110010	11010100	11010110	100010110	100100100	100110000	100110010	100110100	...
4 th - Post sort	10100	10101	110000	10011	10100	10101	110010	10100	10110	10110	100100	110000	110010	110100	...
Da	3	2	4	5	6	1	4	1	4	3	5	1	2	7	...

Fig. 7.4: *Segmented Sorting Process* usando *Parallel Fix Strategy*.

calculado o custo reduzido das arestas, eles podem ser divididos por uma constante, para evitar que, na fase de pré-processamento, ocorra *overflow*, dado o limite do tipo de dado selecionado para cada valor (por exemplo, `int`, `float`, `long`, etc). O primeiro passo da **Parallel Fix Strategy** é localizar o *most significant bit* dentre os valores de *reduced cost* das arestas. Para tal, executa-se uma redução em busca do maior valor e calcula-se o $\log_2$ sobre ele, posteriormente armazenando o resultado em *msb*. O segundo passo é, então, adicionar, após o *msb*, em cada valor de custo reduzido de cada uma das arestas, os bits correspondentes ao número do segmento somado ao resultado da função $\text{ShifLeft}(\text{msb})$, responsável por movimentar bits à esquerda de acordo com o valor de *msb*, e executar

o algoritmo de ordenação paralelo sobre todo o vetor de uma só vez, movendo o vértice de destino do vetor Da para a posição correspondente, a qual é movida o seu *reduced cost*. Opcionalmente, pode ser executado um pós processamento no vetor de *reduced costs*, subtraindo o valor adicionado anteriormente a cada posição, o que é dispensável para o MPS. Um exemplo de aplicação da *Parallel Fix Strategy* é mostrado na Figura 7.4, a qual representa o grafo da Figura 6.1, antes e após a aplicação do Algoritmo 27.

7.5.4 Algoritmo MPS Paralelo

Feitas as modificações na estrutura de G , Martins et al. (1999) propõem o Algoritmo MPS, baseado no Algoritmo 23, que foi proposto no mesmo artigo. A presente tese apresenta, no Algoritmo 28, uma versão paralela do Algoritmo MPS.

Como pode-se observar, o Algoritmo 28 assemelha-se ao Algoritmo 23, excetuando-se a linha 23, na qual o algoritmo faz a chamada de uma das funções propostas 26 ou 27, descritas anteriormente. Como dito, uma vez reorganizado em *Forward Sorted Star Form*, não é mais necessária a execução de reduções para encontrar a menor aresta de v_i , conforme proposto na função de geração de caminhos candidatos 29.

Da mesma forma, a função de geração de caminhos candidatos do MPS paralelo tem apenas a linha 13 da função de geração de candidatos da generalização de Yen paralela modificada. Tal modificação exclui totalmente a necessidade de reduções, tornando mais ágil a busca pela aresta de menor custo. No entanto, deve obter tamanho desempenho a ponto de compensar a fase de processamento para deixar G em sua *Forward Sorted Star Form*.

Algorithm 28: DeviceFunction: MPS

Input : $K, Sa, Da, Wa, Ra, X, L, C, A, s, t$
Output: um conjunto X contendo os K -Shortest Paths de s a t em G

```

1  shared<var>  $k \leftarrow 1, flag \leftarrow false, tid \leftarrow \text{getThreadId}(), stride \leftarrow \text{getStrideSize}()$ ;
2  shared<arrays of size  $|V|$ >  $ca, prev$ ;
3   $ca \leftarrow \text{Dijkstra}(Va, Sa, Da, Wa, \emptyset, \emptyset, t, \emptyset)$ ;
4  if  $ca[t] \neq \infty$  then
5       $prev \leftarrow \text{ExtractPreviousEdges}(Sa, Da, Wa, ca)$ ;
6      barrier();
7      for  $th \leftarrow tid; th < |V|; th \leftarrow th + stride$  do
8           $T_{[th,t]}^*, C_{[th,t]}^{T^*}, L_{[th,t]}^{T^*} \leftarrow \text{ExtractPath}(th, t, Sa, Wa, ca, prev)$ ;
9      end
10 end
11 barrier();
12 if  $tid = 0$  then
13      $L_{[k]}^X \leftarrow L_{[s,t]}^{T^*}$ ;
14      $C_{[k]}^X \leftarrow C_{[s,t]}^{T^*}$ ;
15      $L_{[k]}^{T^k} \leftarrow L_{[s,t]}^{T^*}$ ;
16      $C_{[k]}^{T^k} \leftarrow C_{[s,t]}^{T^*}$ ;
17 end
18 for  $th \leftarrow tid; th < L_{[s,t]}^{T^*}; th \leftarrow th + stride$  do
19      $X_{[k,th]} \leftarrow T_{[s,t,th]}^*$ ;
20      $T_{[k,th]}^k \leftarrow T_{[s,t,th]}^*$ ;
21 end
22 barrier();
23 Chamar uma função MPS Forward Sorted Star Form para reorganizar  $G$  (arrays  $Sa, Da, Wa, Ra$ );
24 barrier();
25 while ( $k < K$ ) and ( $flag = false$ ) do
26      $flag \leftarrow true$ ;
27     barrier();
28     Call  $T^k, C^{T^k}, L^{T^k}, A^{T^k}, count \leftarrow \text{MPS Generate Candidates}(tmppc, k, Va, Sa, Da, Wa, Ra, X, L, C, A, s, t)$ ;
29     barrier();
30     if  $tid = 0$  then
31          $k \leftarrow k + 1$ ;
32     end
33      $sht \leftarrow$  redução para encontrar o caminho mais curto disponível em  $T_{[k]}$  através da análise de  $A^{T^k}$  e  $C^{T^k}$ ;
34     barrier();
35     if  $tid = 0$  then
36          $L_{[k]}^X \leftarrow L_{[sht]}^{T^k}$ ;
37     end
38     for  $th \leftarrow tid; th < L_{[sht]}^{T^k}; th \leftarrow th + strideSize$  do
39          $X_{[k,th]} \leftarrow T_{[sht,th]}^k$ ;
40     end
41 end
42 return  $X$ ;

```

Algorithm 29: DeviceFunction: MPS Generate Candidates

Input : $tmppc, k, Va, Sa, Da, Wa, Ra, X, L, C, A, s, t$
Output: um conjunto X contendo os K -Shortest Paths de s a t em G

```

1  shared or private<var> tpc  $\leftarrow 0$ , smc  $\leftarrow 0$ , tid  $\leftarrow \text{getThreadId}()$ , stride  $\leftarrow \frac{L_{[k-1]}^X}{tpc}$ , gridSize  $\leftarrow \text{getGridSize}()$ ;
2  shared<array set of size  $|V| * L_{[k]}^X - k$ > loopless  $\leftarrow \text{true}$ ;
3  if tid = 0 then
4      tpc  $\leftarrow \text{getThreadsSetsSize}(tmppc, L_{[k]}^X - k)$ ;
5      smc  $\leftarrow \lfloor \frac{\text{gridSize}}{tpc} \rfloor$ ;
6  end
7  barrier();
8  for th  $\leftarrow \lfloor \frac{tid}{tpc} \rfloor$ ; th  $\leq L_{[k]}^X - k$ ; th  $\leftarrow th + smc$  do
9      private<var> cn  $\leftarrow \lfloor \frac{th}{tpc} \rfloor$ ;
10     private<var> vertex  $\leftarrow Sa[X_{[k,cn]}]$ ;
11     if  $A_{[cn]}^{T^k} < Va[vertex]$  then
12         edge  $\leftarrow A_{[cn]}^{T^k} + 1$ ;
13          $A_{[cn]}^{T^k} \leftarrow A_{[cn]}^{T^k} + 1$ ;
14     else
15         next th;
16     end
17     barrier();
18     for thc  $\leftarrow \text{modulus}(\frac{tid}{tpc})$ ; thc  $\leq k + L_{[Da[edge]]}^{T^*}$  and (loopless[cn] = false); thc  $\leftarrow thc + tpc$  do
19         if (thc  $\leq k$ ) and (loopless[cn] = false) then
20              $q_{[k,cn,thc]} \leftarrow X_{[k,thc]}$ ;
21              $count_{[k,cn,Sa[X_{[k,thc]}]]} \leftarrow count_{[k,cn,Sa[X_{[k,thc]}]]} + 1$ ;
22             if  $count_{[k,cn,Sa[X_{[k,thc]}]]} > 1$  then
23                 begin atomic region
24                     loopless[cn]  $\leftarrow \text{false}$ ;
25                 end atomic region
26             end
27         else if (thc - k)  $< L_{[Da[edge]]}^{T^*}$  then
28              $q_{[k,cn,thc]} \leftarrow T_{[Da[edge],(thc-k)]}^{T^*}$ ;
29              $count_{[k,cn,Sa[T_{[Da[edge],(thc-k)]}^{T^*}]]} \leftarrow count_{[k,cn,Sa[T_{[Da[edge],(thc-k)]}^{T^*}]]} + 1$ ;
30             if  $count_{[k,cn,Sa[T_{[Da[edge],(thc-k)]}^{T^*}]]} > 1$  then
31                 begin atomic region
32                     loopless[cn]  $\leftarrow \text{false}$ ;
33                 end atomic region
34             end
35         end
36     barrier();
37     if loopless[cn] = true and ( $q_{[k,cn]}$  not in  $T^k$ ) then
38         if  $\text{modulus}(\frac{tid}{tpc}) = 0$  then
39              $L_{[(k+1),cn]}^{T^k} \leftarrow k + L_{[Da[edge]]}^{T^*}$ ;
40             count  $\leftarrow \text{count} + 1$ ;
41         end
42         barrier();
43         for thc  $\leftarrow \text{modulus}(\frac{tid}{tpc})$ ; (thc  $< L_{[(k+1),cn]}^{T^k}$ ) and (loopless[cn] = true); thc  $\leftarrow thc + tpc$  do
44              $T_{[(k+1),cn,thc]}^k \leftarrow q_{[k,cn,thc]}$ ;
45         end
46     end
47     cn  $\leftarrow cn + smc$ ;
48 end
49 return  $T^k, C^{T^k}, L^{T^k}, A^{T^k}, count$ ;

```

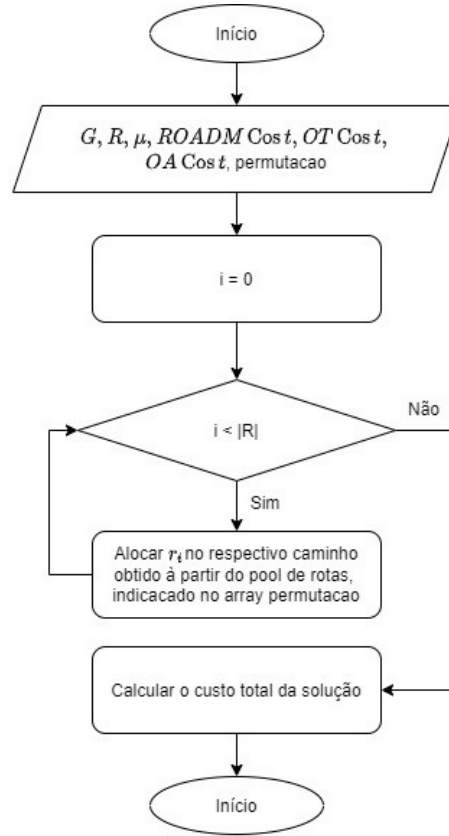


Fig. 7.5: Funcionamento da heurística CUDA *Greedy* utilizando o *Paths Pool*.

7.6 Paralelização CUDA das Estratégias de Solução

Esta seção propõe novas estratégias e topologias de processamento paralelo baseado em CUDA para as heurísticas construtivas, de busca local, GA-PIFRO e ILS-PIFRO. As propostas da estratégia de solução consistem em implementar as estratégias de solução para fazerem o processamento em topologia hierárquica, combinando os *lighpaths* do *Paths Pool* entre si e, assim, gerarem soluções viáveis de modo a retornar a de melhor qualidade ao final. Se, por um lado, a criação do *Paths Pool* demanda tempo, por outro, tal modificação minimiza a utilização do Algoritmo de Dijkstra, aproveitando de forma mais eficiente o paralelismo da arquitetura CUDA.

Para todas as estratégias apresentadas a seguir, requiere-se que o algoritmo de *K-Shortest Loopless Paths* selecionado tenha sido executado e estabelecido os caminhos para cada requisição $r \in R$, armazenando-os no *Paths Pool*.

7.6.1 Heurísticas Construtivas CUDA baseadas no *Paths Pool*

A versão paralelizada em CUDA das heurísticas *Greedy* e PSC utilizando o *Paths Pool* é mostrada no fluxograma da Figura 7.5. Observa-se neste fluxograma que o funcionamento das heurísticas *Greedy* e PSC paralelizadas em CUDA e baseadas no *Paths Pool* eliminam a execução do Algoritmo de Dijkstra a cada iteração, sendo este executado apenas em seu início, por um algoritmo de *K-Shortest Loopless Paths*. A utilização do Algoritmo de Dijkstra nas iterações é substituída por *lighpaths*, que são extraídos do *Paths Pool*. Esta

Algorithm 30: DeviceFunction: *Parallel Greedy Paths Pool Based*

Input : $G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}, \text{permutation}$

```

1 shared or private<var>  $i \leftarrow 0, \text{tid} \leftarrow \text{getThreadId}(), \text{stride} \leftarrow \text{getGridSize}()$ ;
2 for ( $i \leftarrow 0; i \leq |R|; i \leftarrow i + 1$ ) do
3   for ( $\text{th} \leftarrow \text{tid}; \text{th} \leq |X_{[\text{permutacao}[i]]}^{[r_i]}|; \text{th} \leftarrow \text{stride}$ ) do
4      $\text{allocate in the fiber } |X_{[\text{permutacao}[i], \text{th}]}^{[r_i]}| \text{ of } G \text{ the number of wavelengths in } r_{[i]}$ 
5   end
6 end
7 return  $G$ ;

```

é uma operação completamente paralelizada e que, por isso, reduz a queda de desempenho causada pelo gargalo do Algoritmo de Dijkstra em sua etapa de relaxamento de vértices. Tais questões são observadas no Algoritmo 30.

Os parâmetros iniciais do Algoritmo 30 são idênticos aos do Algoritmo 16, porém, o vetor *permutation* passa a ter, em cada um de seus respectivos índices, o número do *lightpath* a ser extraído do *Paths Pool* para a respectiva requisição $r \in R$. Uma solução viável é dada pela combinação de um *lightpath* para cada uma das requisições.

Assim como na versão paralela baseada em Algoritmo de *Shortest Path*, a paralelização da heurística PSC baseada em *Paths Pool* é dada de forma praticamente idêntica à heurística *Greedy*, modificando-se apenas a forma de cálculo do custo incremento, conforme é visto na Seção 3.2.2, e incrementando o valor de p a cada iteração.

7.6.2 Busca Local Baseada em *Paths Pool*

A versão da busca local paralelizada em CUDA, baseada no *Paths Pool*, é exibida no fluxograma da Figura 7.6. Nota-se que as operações de remoção da requisição e alocação de um *lightpath* para a requisição a partir do *Paths Pool* continuam na sequência. É realizada uma iteração para cada requisição $r \in R$, na qual, por sua vez, faz uma iteração para cada caminho presente no *Paths Pool* para tal. Caso o custo da rede seja reduzido durante tais iterações, o algoritmo é finalizado. Um aprimoramento é visto no algoritmo para o caso do *Paths Pool* não conter todos os *lightpaths* possíveis para a requisição, no qual é executado o Algoritmo de Dijkstra tentando encontrar um *lightpath* de menor custo. Os detalhes para a busca local baseada no *Paths Pool* são exibidos no Algoritmo 31.

Nota-se no Algoritmo 31 que não há necessidade de barreira de sincronização entre as etapas de remoção da requisição da rede e sua alocação em um *lightpath* diferente, aumentando o desempenho de execução.

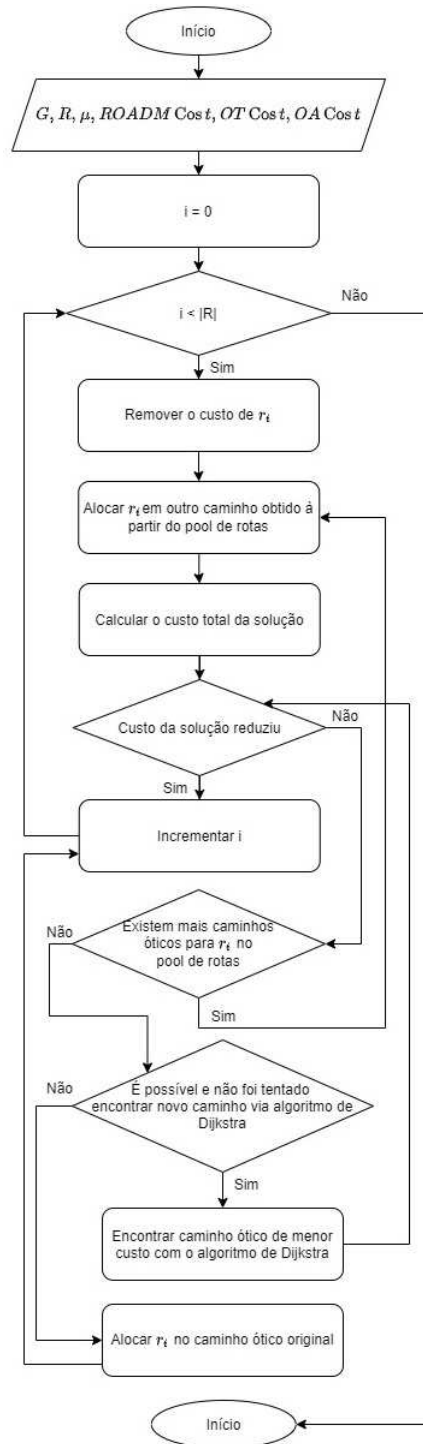


Fig. 7.6: Funcionamento da busca local baseada no Algoritmo de Dijkstra.

Algorithm 31: DeviceFunction: *Parallel Local Search Paths Pool Based*

Input : G , R , μ , ROADMCost, OTCost, OACost, permutation

```

1 shared or private<var>  $i \leftarrow 0$ ,  $tid \leftarrow \text{getThreadId}()$ ,  $stride \leftarrow \text{getGridSize}()$ ;
2 for ( $i \leftarrow 0$ ;  $i \leq |R|$ ;  $i \leftarrow i + 1$ ) do
3   for ( $y \leftarrow 0$ ;  $y \leq |X^{[r_i]}|$ ;  $y \leftarrow y + 1$ ) do
4     in parallel remove the cost of  $r_i$  from  $G$ ;
5     for ( $th \leftarrow tid$ ;  $th < |X^{[r_i]}_{[y]}|$ ;  $th \leftarrow stride$ ) do
6       alocar os wavelengths of  $r_i$  in the edge  $X^{[r_i]}_{[y,th]}$  of  $G$ ;
7     end
8     barrier();
9     if (cost of  $G$  has been improved) then
10      break;
11    end
12    else if ((cost of  $G$  has not been improved) and ( $y = |X^{[r_i]}|$ )) then
13      call o Algoritmo Parallel Dijkstra para encontrar o caminho com o menor
        custo para  $r_i$  in  $G$  e alocar os comprimentos de onda;
14      if (custo de  $G$  tiver sido melhorado) then
15        break;
16      end
17    end
18 end
19 barrier();
20 return  $G$ ;

```

7.7 Metaheurística Paralela CUDA GA-PIFRO

Conforme visto na Seção 3.3, a Heurística GA-PIFRO é baseada na metaheurística Algoritmo Genético Baseado em Chaves Aleatórias (*Biased Random Key Genetic Algorithm - BRKGA*) (Gonçalves e Resende, 2011). Recentemente, diversas estratégias de paralelização desta metaheurística têm sido propostas.

Um dos trabalhos fundamentais nesse sentido é Gonçalves e Resende (2011). Ele apresenta duas estratégias para a paralelização do BRKGA em arquitetura de memória compartilhada:

- (i) **Estratégia Monopopulacional:** esta estratégia é classificada pela taxonomia de Crainic et al. (1997) como *1C/RS/SPSS*. Um processo **master** cria uma população. O cruzamento entre indivíduos é realizado de forma paralela por processos **slaves**, acelerando, assim, a evolução da população. A evolução entre as gerações faz com que a topologia de processamento desta estratégia seja classificada em Anel de Processamento Unidirecional. Com esta estratégia, pode-se observar que é executada uma paralelização em baixo nível, usando uma única região do espaço de busca para a metaheurística;
- (ii) **Estratégia Multipopulacional:** um processo **master** cria diversas populações diferentes, cada qual com suas parcelas TOP e MID. Processos **slaves** são criados e estes, simultaneamente, fazem o cruzamento da quantidade determinada de cromossomos, de modo que todos os processos **slaves** atuam em uma mesma população por vez. Gonçalves e Resende (2011) mostra que pode ser configurado um momento em que as populações trocam seus melhores cromossomos, de forma classificada pela taxonomia de Crainic et al. (1997) como *KS*, através do processo **master**. A esta operação de sincronização dá-se o nome de *Exchange Elite*, e é realizada a cada X_{NUMBER} gerações. Desta forma, esta estratégia pode ser classificada como *1C/KS/MPSS*, caso utilize *Exchange Elite* ou *1C/MPSS*, caso contrário. Como o processamento das gerações ocorre um por vez, a topologia de processamento desta estratégia é classificada também como Anel de Processamento Unidirecional. Nota-se nesta estratégia a decomposição do espaço de busca, o qual é dividido em sub-domínios, um para cada população. No entanto, pode-se caracterizar uma busca múltipla cooperativa ou independente, devido ao parâmetro *ExchangeElite*. Esta estratégia permite somente uma mesma configuração para todas as populações.

Gonçalves e Resende (2012) mostra que a sincronização causa uma melhora na qualidade final das soluções quando o BRKGA é aplicado com estratégia multipopulacional, paralelizado em ambiente de memória compartilhada, para o Problema de Carregamento de Containeres. A mesma forma de paralelização do BRKGA e para a mesma arquitetura é aplicada com sucesso em de Faria Jr. et al. (2019) para o Problema de Reconfiguração da Rede de Distribuição de energia elétrica e por Amaro Junior et al. (2017) para o Problema de Embalagem de Tiras Irregulares. A justificativa para a utilização de múltiplas populações do BRKGA nestas arquiteturas dá-se pela distribuição dos núcleos de processamento e como forma de evitar sincronizações longas entre os resultados obtidos pelas *threads*.

O BRKGA também encontra-se paralelizado de forma híbrida, para arquiteturas CUDA, nos trabalhos de Expósito Piñol (2012); Gifre Renom (2012); Alves et al. (2021). Em Gifre Renom (2012), uma versão paralelizada em CUDA do BRKGA é utilizada com sucesso

para a solução do Problema de Otimização de Redes Multinível. Em [Expósito Piñol \(2012\)](#) e [Alves et al. \(2021\)](#), os autores desenvolveram *frameworks* para programadores utilizarem o BRKGA Paralelizado em CUDA e os disponibilizaram. Tanto a estratégia monopopulacional quanto a multipopulacional são disponibilizadas nestes *frameworks* e devem ser utilizadas de acordo com a sincronização de resultados imposta pelo problema a ser tratado.

Conforme visto no Capítulo 8, as heurísticas construtivas para PIFRO baseadas no Algoritmo de Dijkstra, apesar de criarem soluções viáveis diversas, através de diferentes entradas do vetor com a ordem de permutação das requisições r a serem instaladas na rede, nem sempre terão caminhos diferentes para cada um. Diferentemente de tais heurísticas construtivas, as heurísticas construtivas para PIFRO, quando baseadas no *paths pool*, podem ser configuradas para criarem soluções viáveis com caminhos diferentes entre si, dado que eles foram previamente estabelecidos.

7.7.1 Implementações Propostas para CUDA-GA-PIFRO

Adotamos neste trabalho três estratégias diferentes para CUDA-GA-PIFRO, ao usar diferentes heurísticas construtivas, sendo:

- (i) CUDA GA-PIFRO baseado em paralelização de baixo nível monopopulacional: as *threads* são divididas de forma a gerarem um número maior de cromossomos na população, pois, como há um ponto inerentemente sequencial na construção de soluções viáveis, criar um número maior de soluções viáveis em paralelo atribui maior volume de trabalho útil a cada *thread* disponível na grade da GPU;
- (ii) CUDA GA-PIFRO baseado em paralelização de baixo nível multipopulacional: as *threads* são divididas de forma a gerarem diversas populações. A diferença de desempenho nesta estratégia deve-se ao fato de populações necessitarem sincronizar soluções entre seus cromossomos e não entre cromossomos de diversas populações;
- (iii) CUDA GA-PIFRO baseado em paralelização hierárquica com *Paths Pool*: sincronizações para instalação de requisições r nas redes óticas são desnecessárias, pois, uma vez já estando estabelecidos, pelo menos em partes, os *lightpaths* possíveis, sua instalação pode ser feita em paralelo.

O Algoritmo CUDA GA-PIFRO é apresentado nos pseudocódigos dos algoritmos [32](#), [33](#) e [34](#).

O Algoritmo [32](#) exibe a forma como é gerada a população inicial do Algoritmo CUDA GA-PIFRO, através das heurísticas construtivas *Greedy* ou *PSC*, paralelizadas em baixo nível para CUDA. Inicialmente, o algoritmo recebe como parâmetros:

- (i) Um *array cromossomes*, o qual é retornado ao final da execução do algoritmo, contendo um conjunto de soluções viáveis iniciais para o Algoritmo CUDA GA-PIFRO;
- (ii) O tamanho do conjunto de requisições R ; e
- (iii) o tamanho da população a ser criada.

O *loop* entre as linhas [2](#) e [8](#) faz com que, para cada indivíduo a ser criado na população inicial, uma ordem de permutação aleatória das requisições R seja criada. Em seguida, ainda neste *loop*, as heurísticas construtivas *Greedy* ou *PSC*, paralelizadas em baixo nível

Algorithm 32: DeviceFunction: Initial_Population_Shortest_Path_Based

Input : cromossomes, $|R|$, size of population
Output: um array de cromossomos, contendo a população inicial de soluções

```

1 shared or private<var> tid  $\leftarrow$  getThreadId(), stride  $\leftarrow$  getGridSize();
2 for ( $y \leftarrow 0$ ;  $y < \text{size of population}$ ;  $y \leftarrow y + 1$ ) do
3   for ( $th \leftarrow tid$ ;  $th \leq |R|$ ;  $th \leftarrow th + \text{stride}$ ) do
4     permutation[th]  $\leftarrow$  Random(0,  $|R|$ );
5   end
6   barrier();
7   cromossomes[y]  $\leftarrow$  Parallel Greedy or PSC Shortest Paths Algorithm Based(G,
    R,  $\mu$ , ROADMCost, OTCost, OACost, permutation);
8 end
9 barrier();
10 return cromossomes;

```

Algorithm 33: DeviceFunction: Initial_Population_Paths_Pool_Based

Input : cromossomes, $|R|$, size of population
Output: um array de cromossomos, contendo a população inicial de soluções

```

1 shared or private<var> tid  $\leftarrow$  getThreadId(), stride  $\leftarrow$  getGridSize();
2 for ( $y \leftarrow 0$ ;  $y < \text{size of population}$ ;  $y \leftarrow y + 1$ ) do
3   for ( $th \leftarrow tid$ ;  $th \leq |R|$ ;  $th \leftarrow th + \text{stride}$ ) do
4     permutation[th]  $\leftarrow$  Random(0,  $|X^{[r_{th}]|}$ ); warranty each permutation be
      exclusive;
5   end
6   barrier();
7   cromossome[y]  $\leftarrow$  Parallel Greedy or PSC Paths Pool Based(G, R,  $\mu$ ,
    ROADMCost, OTCost, OACost, permutation);
8 end
9 barrier();
10 return permutation;

```

para CUDA, constroem uma solução viável para cada ordem de permutação criada e a armazena no *array cromossomes*. Ao final da execução do *loop*, o *array cromossomes* é retornado.

Por sua vez, o Algoritmo 33 exibe a forma como é gerada a população inicial do Algoritmo CUDA-GA-PIFRO, através das heurísticas construtivas, paralelizadas através do *Paths Pool*, *Greedy* ou *PSC* para CUDA. Inicialmente, assim como o Algoritmo 32, são recebidos, como parâmetros, o *array cromossomes*, o tamanho do conjunto de requisições R e o tamanho da população a ser criada. O *loop* entre as linhas 2 e 8 faz com que, para cada indivíduo a ser criado na população inicial, uma ordem de permutação aleatória, contendo, em cada um de seus índices, o número do caminho de $X^{[r_{th}]}$ a ser alocado na rede ótica, seja criada. Em seguida, ainda neste *loop*, as heurísticas construtivas paralelizadas *Greedy* ou *PSC*, baseadas em *Paths Pool* para CUDA, constroem uma solução viável para cada ordem de permutação criada e a armazena no *array cromossomes*. Ao final da execução do *loop*, o *array cromossomes* é retornado.

Algorithm 34: DeviceFunction: CUDA GA-FIP

Input : $G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}$

```

1 shared or private<var>  $i \leftarrow 0$ ,  $\text{tid} \leftarrow \text{getThreadId}()$ ,  $\text{stride} \leftarrow \text{getGridSize}()$ ,
   $\text{top-cromossome} \leftarrow \infty$ ;
2 Call Um dos kernels de geração de população inicial (algorithms 32 ou 33; while
  (critério de parada não for alcançado) do
3   em paralelo classificar a população pelos custos dos cromossomos;
4   barrier();
5   if ((custo do cromossome[0] < custo do top-cromossome) and (tid = 0)) then
6     |  $\text{top-cromossome} \leftarrow \text{cromossome}[0]$ ;
7   end
8   em paralelo, divida a população em conjuntos  $TOP$ ,  $MID$  e  $BOT$ ;
9   em paralelo copie a parcela  $TOP$  para a próxima geração;
10  barrier();
11  for ( $\text{th} \leftarrow \text{tid}$ ;  $\text{th} < |MID|$ ;  $\text{th} \leftarrow \text{stride}$ ) do
12    | selecione um elemento aleatório do conjunto  $TOP$ ;
13    | selecione um elemento aleatório do conjunto  $MID \cup BOT$ ;
14    | for ( $y \leftarrow 0$ ;  $y < |R|$ ;  $y \leftarrow y + 1$ ) do
15      |  $p_3 \leftarrow \text{Crossover}(p_1, p_2, p_{rec})$ ;
16    | end
17  end
18  barrier();
19  em paralelo, gere aleatoriamente o conjunto  $BOT$  da população;
20  em paralelo avalie cada elemento da população;
21  barrier();
22 end
23 return  $\text{top-cromossome}$ ;

```

O Algoritmo 34 recebe, como parâmetros, o grafo G , o conjunto de requisições R e os valores das constantes μ , ROADMCost , OTCost e OACost , explicadas anteriormente. Na linha 2 do Algoritmo CUDA-GA-PIFRO, são chamados os *kernels* que geram populações iniciais, de acordo com o desejo de trabalhar com *Paths Pool* ou paralelização em baixo nível. Após tal população ser gerada, o algoritmo, a cada iteração, faz o cruzamento de forma paralelizada em baixo nível dos cromossomos e os divide nas parcelas TOP , MID e BOT . Os cromossomos aptos são copiados de forma paralela para a próxima geração, descartando-se os demais. Caso deseje-se utilizar mais de uma população para o Algoritmo CUDA GA-PIFRO, várias instâncias do Algoritmo 34 devem ser executadas simultaneamente. Destaca-se a vantagem desta estratégia para a execução baseada nas heurísticas construtivas *Greedy* ou PSC paralelizadas em baixo nível, pois, assim, o ponto inerentemente sequencial do Algoritmo de Dijkstra pode utilizar de forma mais ampla o trabalho útil de diversas *threads* simultâneas.

7.8 Metaheurística *Iterated Local Search* Paralela

Conforme já exposto na Seção 3.4, a heurística ILS-PIFRO baseia-se na metaheurística *Iterated Local Search* (ILS) (Lourenço et al., 2010). O trabalho dos Reis et al. (2019) mostra que os resultados desta heurística são melhores que os obtidos pela heurística GA-PIFRO, apresentada na Seção 3.3. Assim, é de forte interesse o estudo das possibilidades de paralelização dessa heurística.

Diversas estratégias de paralelização do ILS, bem como de técnicas diversas de otimização de busca local, são encontradas na literatura para solucionar problemas de otimização combinatória. Em Johnson e Mcgeoch (1997) são definidas três estratégias de paralelização de otimização de busca local para solucionar o Problema do Caixeiro Viajante (*Traveling Salesman Problem* - TSP), utilizando buscas *k-opt*. A primeira estratégia é classificada como *pC/RS/MPSS*. Funciona através de particionamento geométrico, dividindo o conjunto de cidades em sub-grupos, que são enviados para diferentes núcleos de processamento, os quais executam uma heurística construtiva seguida de uma busca local. Após isso, segundo Johnson e Mcgeoch (1997), as rotas são reconectadas, no entanto, de forma não muito eficiente. A topologia de processamento paralelo desta estratégia é estrela. A segunda estratégia é classificada como *pC/KS/MPSS* e executa o particionamento geométrico baseado em turnês para dividir rotas em soluções parciais, que incluem uma parte das bordas da solução corrente. A segunda estratégia também utiliza a topologia de processamento paralelo estrela. A terceira abordagem trata uma paralelização de construção e exploração de vizinhança utilizando-se de estratégia de paralelização de baixo nível, sendo, então, classificada como *1C/RS/SPSS*, e sua topologia de processamento paralelo é anel de processamento unidirecional. Em Hong et al. (1997), técnicas de otimização de busca local são paralelizadas de forma a construir múltiplas soluções simultaneamente para o TSP, sendo uma em cada núcleo de processamento disponível. Esta estratégia é classificada em *pC/KS/MPSS* e sua topologia de processamento paralelo é estrela. Martin e Otto (1996) propuseram uma implementação paralelizada em uma forma semelhante a de Hong et al. (1997) para o mesmo problema; no entanto, a melhor dentre todas as soluções encontradas substitui as demais em um intervalo irregular de tempo, sendo então classificada em *pC/C/MPSS* e sua topologia de processamento paralelo é completamente conectada.

Luong et al. (2010b) propõem uma metodologia de paralelização do ILS em GPU para solucionar o TSP. O *host* é responsável por todas as buscas locais serem executadas nas diversas construções geradas simultaneamente no *device*, classificando esta heurística como *pC/RS/MPSS*, tendo sua topologia de processamento paralelo estrela. No trabalho de Delévacq et al. (2012), o TSP é solucionado através de um ILS paralelizado em GPU, em duas estratégias. A primeira estratégia executa um paralelismo de baixo nível para acelerar o processo de busca, sendo classificada como *1C/RS/SPSS*, com topologia de processamento paralelo estrela. A segunda estratégia baseia-se em associar cada solução a um bloco no *device* e executar uma busca local. A troca de informações entre as múltiplas soluções é realizada através da memória principal para sincronização. Esta estratégia pode ser classificada como *pC/C/MPSS* e sua topologia de processamento paralelo é estrela. Fosin et al. (2013) propõem buscas locais *2-opt* e *3-opt* em um ILS paralelizado. Estas estruturas de busca local paralelizadas são apresentadas em Johnson e Mcgeoch (1997). As buscas locais são realizadas em um *device* GPU, para tentar melhorar as rotas paralelamente, enquanto o restante da computação é executada no processador *host*. Esta estratégia pode ser classificada como *pC/RS/MPSS* e sua topologia de processamento

paralelo é estrela.

Rocki e Suda (2012) também apresentam um estudo a respeito de solução paralela usando ILS para o TSP. Neste trabalho, inicialmente, um processo **master** dá início a diversos processos **slaves**, em que cada um recebe um diferente ponto inicial no espaço de buscas ou domínio para iniciar uma busca. A cada iteração, a melhor solução global corrente é transmitida aos processos **slaves**. Esta estratégia é classificada como $pC/RS/MPSS$. Rocki e Suda (2012) propõem ainda uma segunda estratégia de paralelização da metaheurística. Nesta segunda estratégia, diferentemente da primeira, quando um dos processos **slaves** encontra a melhor solução global, ela é copiada para os demais, que passam a realizar suas buscas a partir dela. Esta segunda estratégia é classificada como $pC/KS/SPDS$. Nota-se nesta segunda estratégia a utilização de cooperação síncrona entre buscas múltiplas independentes, utilizando a troca global de informações. Rocki e Suda (2012) utilizaram ainda uma estratégia simular à segunda estratégia, a qual é implementada em um ambiente de memória distribuída. As três implementações de Rocki e Suda (2012) utilizam-se da topologia de processamento paralelo estrela e foram realizadas em ambientes de memória compartilhada, distribuída e heterogêneos, utilizando GPU.

Também para solucionar o TSP, em Lei e Xu (2014) a metaheurística ILS é paralelizada. Neste caso, aplica-se múltiplas buscas independentes, que se sincronizam ao atingir um determinado critério ou retornam a melhor dentre todas as buscas ao final da execução da metaheurística. Respectivamente, podemos classificar estas estratégias de paralelização como $pC/RS/MPDS$ e $pC/KS/SPDS$. Pode-se, ainda, afirmar que as estratégias de sincronização adotadas por cada estratégia são, respectivamente, buscas múltiplas independentes e cooperação síncrona. As duas estratégias utilizam-se de topologia de processamento paralelo estrela.

Zhou et al. (2016) também apresenta uma estratégia de paralelização do ILS para solucionar o TSP. Para tal, um processo **master** seleciona um ponto no espaço de busca, que é enviado para a GPU (*device*), em que processos **slaves** paralelamente avaliam soluções vizinhas no espaço de busca e retornam a melhor solução encontrada para o processo **master**, quando um determinado critério é alcançado. Caso o critério de parada ainda não tenha sido atingido, o processo **master** repete as etapas anteriores. Esta estratégia pode ser classificada como $pC/RS/MPSS$.

Bäumelt et al. (2016) paraleliza o ILS em ambientes heterogêneos com GPU e obtém resultados para o TSP. Neste trabalho, são paralelizadas, em GPU, a avaliação em paralelo de diversos movimentos de busca em uma primeira estratégia, sendo então classificada como $1C/RS/SPSS$, usando conceitos de paralelismo em baixo nível. Em uma segunda estratégia, são paralelizadas as buscas locais executadas pelo ILS, sendo executada uma busca diferente em cada bloco de *threads* na GPU, o que a classifica como $pC/RS/MPDS$. A cada iteração, a melhor solução global é repassada aos processos *slaves*.

Em Munguía et al. (2017), o ILS é paralelizado em ambientes de memória compartilhada e distribuída para o Problema de Fluxo Multiproduto em Redes (*Multicommodity Network Flow Problem*). Para isso, Munguía et al. (2017) desassociam a sequência de buscas locais, deixando cada sequência independente, de modo que todas sejam exploradas em paralelo. Quando as sequências encontram alguma melhoria, são recombinadas em uma única solução. Esta estratégia é classificada como $pC/RS/MPSS$ e sua topologia de processamento paralelo é estrela.

Em Luong et al. (2010a), para solucionar o *Quadratic 3-Dimensional Assignment Problem*, uma heurística de Busca Tabu é incorporada a um ILS paralelizado em GPU, para

gerar múltiplas soluções paralelamente, sendo então classificada como *pC/RS/MPSS*.

Juan et al. (2012) paralelizam o ILS para solucionar o *Permutation Flow-Shop Problem* para ambientes de memória compartilhada e heterogêneos com GPU. Neste trabalho, o processo *master* cria cópias da instância carregada em memória, usando uma semente diferente para sortear a ordem de permutação das tarefas, as quais serão alocadas por processos *slaves* e aplicado o paralelismo em baixo nível. Esta estratégia é classificada como *pC/RS/MPSS*.

Nota-se que a aplicação do ILS em forma paralela tem sido um importante foco de pesquisa para o estudo da solução de problemas diversos de Otimização Combinatória, em especial quando se busca a minimização do custo computacional da solução de instâncias de grande porte. Deve-se destacar o largo uso de implementações usando GPU.

7.8.1 Implementações Propostas para CUDA ILS-PIFRO

Para CUDA ILS-PIFRO, apresentamos três propostas de implementação. As propostas de implementação diferenciam-se de acordo com as heurísticas construtivas e/ou critérios que permitem ou não repetição da ordem de permutação a ser utilizada, sendo:

- (i) CUDA ILS-PIFRO baseado em paralelização de baixo nível: as *threads* são divididas, de forma a gerarem uma ordem de permutação que é enviada para uma heurística construtiva para obter uma solução viável. Nesta modalidade de paralelização, após se obter uma solução viável, a ordem de permutação que a gerou passa por uma perturbação, que altera a ordem das requisições r a serem alocadas na rede ótica. O pseudocódigo desta proposta de implementação é mostrado no Algoritmo 35;
- (ii) CUDA ILS-PIFRO baseado em paralelização hierárquica com *Paths Pool*: as *threads* são divididas de forma a gerarem diversas permutações, as quais, posteriormente, são enviadas para heurísticas construtivas baseadas em *Paths Pool*;
- (iii) CUDA ILS-PIFRO baseado em paralelização hierárquica com *Paths Pool* e exclusividade de ordem de permutação: difere-se da proposta de paralelização (ii) pelo fato de cada ordem de permutação gerada verificar previamente se já foi ou não enviada a uma heurística construtiva (somente permutações exclusivas são permitidas a cada iteração).

O Algoritmo 35 apresenta o procedimento CUDA ILS-PIFRO, baseado em baixo nível. A lógica de funcionamento é a mesma de ILS-PIFRO, conforme é exibido no Algoritmo 3.4. Basicamente, paraleliza-se a formação da ordem de permutação, conforme visto entre as linhas 2 e 4, e, após a sincronização das *threads* disponíveis, uma heurística construtiva também paralelizada em baixo nível para CUDA é chamada, para criar, assim, um solução viável, conforme visto na linha 6. Nas linhas seguintes, assim como na versão sequencial de ILS-PIFRO, a ordem de permutação gerada sofre uma perturbação, a cada iteração uma nova solução viável é criada, e a melhor dentre todas é retornada ao final da execução do algoritmo.

Já o Algoritmo 36 exhibe as formas de paralelização CUDA para ILS-PIFRO descritas nos itens (ii) e (iii). Conforme pode ser observado entre as linhas 2 e 4, assim como a forma de paralelização CUDA em baixo nível, essa modalidade também paraleliza a criação da ordem de permutação, diferenciando-se da primeira por cada índice do vetor

Algorithm 35: DeviceFunction: CUDA ILS-PIFRO

Input : $G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}$

```

1 shared or private<var> i  $\leftarrow$  0, tid  $\leftarrow$  getThreadId(), stride  $\leftarrow$  getGridSize();
2 for (th  $\leftarrow$  tid; th <  $|R|$ ; th  $\leftarrow$  stride) do
3   | permutation[th]  $\leftarrow$  Random(0,  $|R|$ );
4 end
5 barrier();
6  $s \leftarrow \text{Parallel Greedy}(G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}, \text{permutation})$ ;
7 barrier();
8  $s^* \leftarrow \text{ParallelLocalSearch}(s)$ ;
9 while (critério de parada não for alcançado) do
10  |  $s' \leftarrow \text{ParallelPertubation}(k\text{-rerouting}, s)$ ;
11  |  $s' \leftarrow \text{ParallelLocalSearch}(s')$ ;
12  | barrier();
13  | if (cost of  $s$  < cost of  $s^*$  and (tid = 0)) then
14  |   |  $s^* \leftarrow s'$ ;
15  | end
16 end
17 return  $s^*$ ;

```

Algorithm 36: DeviceFunction: CUDA ILS-PIFRO

Input : $G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}$

```

1 shared or private<var> i  $\leftarrow$  0, tid  $\leftarrow$  getThreadId(), stride  $\leftarrow$  getGridSize();
2 for (th  $\leftarrow$  tid; th <  $|R|$ ; th  $\leftarrow$  stride) do
3   | permutation[th]  $\leftarrow$  Random(0,  $|X^{[r_{th}]|}$ );
4 end
5 for (th  $\leftarrow$  tid; th <  $|R|$ ; th  $\leftarrow$  stride) do
6   | Verificar se a ordem de permutação gerada já não consta em um lista de
   |   candidatos restritos;
7 end
8 barrier();
9  $s \leftarrow \text{Parallel Greedy}(G, R, \mu, \text{ROADMCost}, \text{OTCost}, \text{OACost}, \text{permutation})$ ;
10 barrier();
11  $s^* \leftarrow \text{ParallelLocalSearch}(s)$ ;
12 while (stop criteria does not reached) do
13  |  $s' \leftarrow \text{ParallelPertubation}(k\text{-rerouting}, s)$ ;
14  |  $s' \leftarrow \text{ParallelLocalSearch}(s')$ ;
15  | barrier();
16  | if (cost of  $s$  < cost of  $s^*$  and (tid = 0)) then
17  |   |  $s^* \leftarrow s'$ ;
18  | end
19 end
20 return  $s^*$ ;

```

de permutação corresponder a uma requisição r e o valor em cada uma representar o número do k -Shortest Path previamente armazenado no *Paths Pool* que deve ser utilizado para a respectiva requisição. De forma opcional, a exclusividade da ordem de permutação criada pode ser verificada conforme exibe-se entre as linhas 5 e 7. Em seguida, após as *threads* disponíveis serem sincronizadas, uma heurística construtiva paralelizada em CUDA e baseada em *Paths Pool* é chamada na linha 9 para gerar a solução viável. Nas linhas seguintes, assim como na versão sequencial de ILS-PIFRO, a ordem de permutação gerada sofre uma perturbação, a cada iteração uma nova solução viável é criada, e a melhor dentre todas é retornada ao final da execução do algoritmo.

7.9 Análise de dominância do espaço de buscas com soluções geradas via *Paths Pool*

Ao criar estratégias de solução para PIFRO baseadas na utilização do *Paths Pool*, pode-se calcular a quantidade de soluções exploradas no espaço de busca. Apesar deste trabalho não provar a aproximação de suas estratégias de solução para PIFRO em percentuais de acordo com a solução ótima para as instâncias, é possível mostrar sua aproximação por taxa de dominância. A taxa de dominância consiste em apresentar, ao final da execução de uma estratégia, o percentual de soluções disponíveis no espaço de busca que foram explorados por ela, conforme é explicado no teorema à seguir:

Teorema 2. *É possível determinar a taxa de dominância de uma estratégia de solução para as instâncias de PIFRO.*

Demonstração. Estratégias de solução para PIFRO são métodos que geram combinações entre um caminho para cada requisição $r \in R$ de uma instância. Uma solução viável para uma instância de PIFRO precisa conter pelo menos um caminho entre os vértices de origem e destino para cada requisição. Sendo assim, o *lower bound* de soluções viáveis para uma instância é 1.

O *upper bound* de soluções viáveis é dado através de análise combinatória, pelo princípio fundamental da contagem (Chartrand et al., 2018)(Rosen, 2010), o qual indica o limite superior de soluções para uma instância. Para tanto, sabendo-se que a quantidade máxima de caminhos para os vértices de origem e destino de cada requisição de uma instância de PIFRO, em seu grafo $G(V, E)$, é $2^{|E|}$ no pior caso, seja então:

- X : o conjunto de caminhos para cada requisição $r \in R$;
- X^{r_i} : o subconjunto de X que contém os caminhos para a requisição $r_i \in R$;
- $X_{[p]}^{r_i}$: a indicação do caminho de índice p da requisição r_i armazenado em X ;

Portanto, pelo princípio fundamental da contagem, o *upper bound* de soluções viáveis para uma instância de PIFRO é $|r_1|x|r_2|x \dots x|r_{|R|}|$. A ilustração da Figura 7.7 exibe as possibilidades de combinações de caminhos para uma instância de PIFRO.

Do lado esquerdo da Figura 7.7 é visto o conjunto X , que tem, à direita, o detalhamento dos caminhos para as requisições r_n de uma instância de PIFRO, cada qual em sua respectiva tabela. Estas tabelas contém, na primeira coluna, o número da respectiva requisição; na segunda coluna, o índice do caminho, e nas colunas seguintes os vértices ou

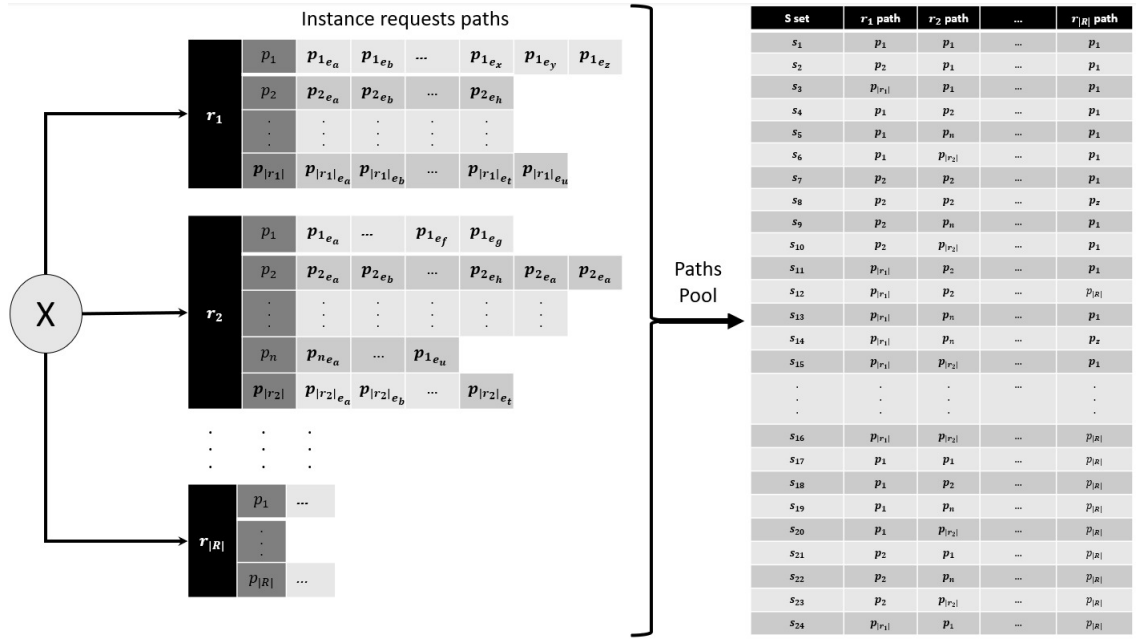


Fig. 7.7: Ilustração do conjunto de soluções viáveis possíveis de acordo com os caminhos para cada $r \in R$

índices das arestas que compõem o respectivo caminho. A Tabela *Paths Pool*, à direita das tabelas que detalham os caminhos de cada requisição na Figura 7.7, mostra o maior tamanho possível de espaço de busca para soluções de uma instância de PIFRO, soluções estas que, por sua vez, são armazenadas no conjunto S . Nesta tabela, a primeira coluna indica o índice da solução; as colunas seguintes indicam os índices dos caminhos selecionados para cada respectiva requisição. Portanto, é possível criar estratégias de solução para o problema capazes de determinar qual o percentual do espaço de busca será vasculhado e, assim, determinar sua taxa de dominação. $\square$

Capítulo 8

Experimentos Computacionais

Este Capítulo apresenta experimentos realizados para a resolução do Problema de Instalação de Fibras em Redes Óticas (PIFRO) utilizando instâncias baseadas nas bibliotecas *DIMACS* (Rossi e Ahmed, 2015), *SNAP* (Leskovec e Krevl, 2021) e *SNDlib* (Orlowski et al., 2007b). Inicialmente, avaliou-se a quantidade de execuções do Algoritmo de Dijkstra em um tempo de 10 minutos nas versões sequenciais das heurísticas *Greedy* e PSC. Em um segundo momento, foram avaliadas os desempenhos das versões paralelizadas em CUDA dos algoritmos de Yen, Generalização de Yen e MPS.

O Capítulo está, portanto, organizado como segue. A Seção 8.1 descreve a configuração dos parâmetros dos experimentos realizados. A Seção 8.2 descreve as instâncias de dados utilizadas para alimentar os algoritmos propostos nesta tese. A Seção 8.2.4 analisa a utilização do Algoritmo de Dijkstra nas estratégias de solução sequenciais GA-PIFRO e ILS-PIFRO, mostradas nas Seções 3.3 e 3.4. As Seções 8.4 e 8.5 mostram os resultados dos experimentos associados às heurísticas paralelizadas CUDA-Yen, CUDA-Yen-*Generalization* e CUDA-MPS. As Seções 8.6 e 8.7 avaliam o desempenho das heurísticas paralelas propostas e o desempenho das melhores heurísticas paralelas dentre as diferentes versões. Já as Seções 8.8 e 8.9 avaliam a qualidade das soluções geradas e a qualidade das soluções geradas pelas melhores heurísticas paralelas.

8.1 Configuração dos Parâmetros

As versões sequenciais das heurísticas *Greedy*, PSC, GA-FIP, ILS-*Greedy* e ILS-PSC foram codificadas na linguagem C++ e compiladas com o compilador *GNU GCC* versão 4.4.3. Como em Goulart et al. (2011), o tamanho das parcelas TOP, REST e BOT do Algoritmo GA-PIFRO foi definido como $0,25|R|$; $0,70|R|$; e $0,05|R|$, respectivamente. Experimentos preliminares foram configurados com o valor de k para $k = 0, 3|R|$ para o procedimento de perturbação k -opt dos Algoritmos ILS-PIFRO-*Greedy* e ILS-PIFRO-PSC. As heurísticas e algoritmos paralelizados em CUDA (*Greedy*, PSC, CUDA-GA-PIFRO, CUDA-ILS-PIFRO, CUDA-Yen, CUDA-Yen-*Generalization* e CUDA-MPS) foram compilados sob o CUDA Toolkit 11.4 Update 2.

Como sugerido por Antonakopoulos e Zhang (2007), os valores dos equipamentos óticos foram configurados como:

- (i) $C_{ROADM} = 1000$;
- (ii) $C_{OA} = 0,5C_{ROADM}$;

(iii) $C_{OT} = 0,1C_{ROADM}$.

A quantidade de comprimentos de onda suportados por cada fibra ótica foi configurada para $\mu = 100$ e a quantidade máxima de fibras por aresta foi definida como $b_{ij} = 8$, os quais são os valores típicos para estes parâmetros.

Todos os experimentos foram executados em computador com processador Intel (R) Core (TM) i7 CPU, 32GB de RAM, drive NVMe e GPU Nvidia CUDA GTX 1060.

8.2 Instâncias

Esta seção descreve as instâncias de dados utilizadas para alimentar os algoritmos propostos neste trabalho. Foram utilizados três acervos conhecidos da literatura:

- (i) *SNDlib* (*Survivable Network Design Library*) (Orlowski et al., 2007b,a, 2010);
- (ii) *Stanford Network Analysis Project* (Leskovec e Krevl, 2021) (SNAP); e
- (iii) *Discrete Mathematics and Theoretical Computer Science* (DIMACS) (Rossi e Ahmed, 2015).

Estes conjuntos são descritos a seguir.

8.2.1 Instâncias SNDLib

As instâncias presentes na biblioteca *SNDlib* são originárias de problemas de projeto de redes. Um subconjunto destas instâncias foi obtido junto às indústrias e fabricantes de equipamentos. Outro subconjunto destas instâncias são referências em projetos de pesquisas internacionais de redes envolvendo parcerias entre empresas e instituições de ensino. O último subconjunto destas instâncias não tem sua origem divulgada devido a termos de privacidade. Algumas instâncias reais da *SNDlib* sofreram alterações nos seus dados, como o custo e o número de *links*, com o objetivo de garantir a privacidade das empresas. Porém, segundo Orlowski et al. (2007b), as alterações destes dados não descaracterizam estas instâncias.

Quanto à origem, os conjuntos de instâncias *dfn-bwin*, *dfn-gwin*, *pdh* e *polska* foram obtidos através de parcerias entre instituições de ensino, fabricantes de equipamentos e operadoras de rede. As instâncias *geant*, *nobel-us* e *nobel-germany* são referências em grandes projetos de pesquisas na Europa, tendo, como participantes, grandes operadoras de telecomunicação e fabricantes de equipamentos (Orlowski et al., 2007a). O grupo final de instâncias foi utilizado por outros autores em problemas de otimização e disponibilizado para a biblioteca *SNDlib*. Algumas destas instâncias, como *abilene*, *atlanta* e *di-yuan*, foram alteradas através de, por exemplo, aumento do número de *links*, para torná-las mais desafiadoras.

Todas as instâncias disponíveis no site *SNDlib* estão disponibilizadas no formato *Graph Modelling Language* -(GML). O formato *GML* foi criado com o objetivo de se tornar o padrão para a utilização em vários programas editores de grafos. Dentre as opções de programas gratuitos para a edição de grafos com suporte ao formato GML, os autores do projeto *SNDlib* recomendam o *yEd* (yWorks, 2016). Através destes programas, é possível obter informações visuais sobre os grafos e criar grafos através de interface gráfica.

As instâncias disponíveis no *site SNDlib* são utilizadas em problemas de redes genéricos e contém informações como (i) custo; (ii) capacidade dos *links*; (iii) se os *links* são direcionados ou não direcionados, dentre outros dados. O nome de cada instância é formado pelo nome da rede, adicionado a parâmetros que identificam o seu modelo. São exemplos de nomes: *nobel-germany-D-B-M-N-S-A-N-U* e *abilene-D-B-L-N-S-A-N-N*. As letras finais do nome representam informações de configuração do modelo, de acordo com a posição em que ocupam. As informações representadas pelas letras correspondem às seguintes configurações, de acordo com a ordem: modelo da demanda; modelo do *link*; capacidade do *link*; modelo de carga; modelo de rotas; modelo dos caminhos; limite de saltos; e modelo de proteção. Como exemplo, a primeira letra (D) do nome da instância *nobel-germany-D-B-M-N-S-A-N-U* indica que o modelo de demandas utilizado pela instância é **D**irecionado. A próxima letra, (B), indica que o modelo do *link* é **B**idirecional. A definição das outras configurações da rede são descritas a seguir.

Quanto ao modelo, as instâncias são classificadas em:

- modelo direcionado (D): em que cada demanda é roteada de uma origem até um destino com uma matriz de tráfego assimétrica;
- modelo não direcionado (U): em que cada demanda é roteada sem direção entre os nós, sendo a sua matriz de tráfego triangular.

O modelo dos *links* podem ser:

- direcionados (D): em que as capacidades podem ser usadas no sentido do nó de origem até o de destino, sendo o fluxo limitado pela capacidade do enlace. Para este tipo de *link*, o modelo de demanda deve ser direcionado (D);
- não direcionados (U): com a capacidade utilizada em ambas as direções, de modo que a soma do fluxo não devem exceder a capacidade do *link*;
- bidirecionais (B): podem ser usados em ambas as direções e a capacidade de fluxo em cada direção não deve exceder a capacidade do *link*. O modelo de demanda deve ser direcionado.

Quanto a capacidade dos *links*, pode-se ter:

- capacidade linear (L): suporta qualquer valor de capacidade não negativa, fracionária ou inteira;
- capacidade de *link* modular (M): suporta qualquer valor inteiro não negativo da base de capacidades especificadas para o *link*;
- capacidade de *link* explícita (E): utiliza uma lista explícita de capacidades.

Os links podem ter um modelo de carga:

- fixa (Y);
- variável (N).

As rotas podem seguir um modelo:

- contínuo (C): em que qualquer fração da demanda pode ser transportada em qualquer caminho possível;
- inteiro (I): em que apenas múltiplos inteiros das unidades de demanda podem ser utilizados;
- caminho simples: em que cada demanda deve ser roteada sobre um caminho único (S).

Quanto aos caminhos, pode-se considerar:

- todos os caminhos podem ser utilizados;
- verificar o limite de nós utilizados para compor o caminho (A); e
- utilizar lista explícita para cada demanda, restringindo os caminhos possíveis (E).

Quanto a estrutura de demanda:

- Para cada demanda, pode ser considerado um limite no número de nós a serem utilizados (Y);
- o limite do número de nós utilizados pode ser ignorado (N).

Os caminhos podem utilizar:

- proteção na rotas (Y); ou
- serem caminhos sem proteção(N).

Quanto ao tipo de proteção:

- proteção um mais um: em que, para cada demanda, devem existir duas rotas disjuntas em enlaces (P);
- proteção com caminho compartilhado, em que, para um conjunto de demandas, existe um caminho de proteção (S).

A título de exemplo, a topologia da instância *abilene* é apresentada na Figura 8.1. A Figura 8.2 apresenta um exemplo da interface gráfica disponível no *site* do projeto [Orlowski et al. \(2007b\)](#), que é utilizada para a pesquisa das instâncias disponíveis.

Os parâmetros utilizados para realizar a pesquisa no *site* *SNDlib*, que retornaram as instâncias utilizadas nos experimentos deste trabalho são, conforme apresentado na Figura 8.2:

- (i) modelo de demanda direcionado;
- (ii) modelo de *link* bidirecional;
- (iii) sem capacidade de carga nos *links*;
- (iv) modelo de rotas com caminhos simples;
- (v) qualquer modelo de caminho;
- (vi) sem limites no número de saltos; e
- (vii) modelo de tráfego sem proteção.

- (v) Redes de colaboração, sendo os nós cientistas e as arestas colaborações;
- (vi) Grafos representando a *web*, com nós sendo páginas e arestas *hyperlinks*;
- (vii) Rede de produtos geralmente comprados juntos no site de vendas *Amazon* ©;
- (viii) Redes da internet, sendo nós computadores e arestas comunicações;
- (ix) Estradas de estados, como o da Califórnia (EUA), sendo as interseções os nós;
- (x) Grafos temporais da internet como um todo;
- (xi) Redes com pesos positivos e negativos representando oposição;
- (xii) Redes sociais com dados geográficos;
- (xiii) Rede de dados da *Wikipedia*;
- (xiv) Redes temporais de várias redes sociais;
- (xv) Dados do *Twitter* e *Memetracker*;
- (xvi) Dados de comunidades online como o *Reddit*;
- (xvii) Dados de sites que possuem sistema de *review* como a *Amazon*;
- (xviii) Dados de interação física entre pessoas;

8.2.3 Instâncias DIMACS

O terceiro grupo de instâncias utilizado nos experimentos foi o *9th DIMACS Implementation Challenge - Shortest Paths* (Demetrescu et al., 2009). Este grupo está incluído no repositório geral Rossi e Ahmed (2015), que possui informações geográficas de várias estradas dos Estados Unidos e se subdivide em dois tipos de grafos:

- (i) Distâncias físicas entre os nós do grafo;
- (ii) Tempo médio de viagem gasto entre os trajetos;

8.2.4 Adaptações realizadas nos grupos de instâncias

Como os arquivos de cada conjunto de instâncias das bibliotecas aqui referenciadas possuem informações adicionais e dados que não são necessários para execução de testes para o problema, um *software*, desenvolvido em linguagem C++, foi utilizado para formatá-las com apenas as informações necessárias para sua utilização. Este *software*, disponível em PIFRO (2021), recebe, como entrada, o arquivo de dados de uma instância da *SNDlib* e fornece, como saída, um arquivo com o mesmo nome e extensão *.net*. Nas instâncias, a posição de cada vértice é fornecida em função de sua latitude e longitude. O *software* utilizado na transformação das instâncias calcula a distância em quilômetros entre os vértices do grafo a ser analisado.

8.3 Experimento I: Utilização do Algoritmo de Dijkstra nas estratégias de solução sequenciais

O Experimento I tem como objetivo apurar a média do percentual de tempo, pelo Algoritmo de Dijkstra, de cada estratégia de solução sequencial de PIFRO. A Tabela 8.1 mostra o percentual de tempo utilizado pelo Algoritmo de Dijkstra na execução das metaheurísticas GA-PIFRO e ILS-PIFRO após 10 minutos de execução. Como instâncias, foram utilizadas as disponíveis no acervo da biblioteca SNDLib. Tal escolha por estas instâncias para este experimento advém do fato dos trabalhos sobre PIFRO, como [Goulart et al. \(2011\)](#), [dos Reis et al. \(2013\)](#) e [dos Reis et al. \(2019\)](#), as utilizarem para sua realização.

Tabela 8.1: Percentual de tempo utilizado para execução do Algoritmo de Dijkstra pelas estratégias de solução para PIFRO

Instância	V	E	R	BRKGA-GREEDY		BRKGA-PSC		IGS-GREEDY (30%)		IGS-PSC (30%)	
				Avg Iter	% Dijkstra	Avg Iter	% Dijkstra	Avg Iter	% Dijkstra	Avg Iter	% Dijkstra
dfn-bwin	10	45	45	396810	94,71%	493335	92,61%	84497	94,57%	45482	94,48%
dfn-gwin	11	47	55	523600	93,75%	349360	93,56%	67436	95,34%	52623	94,85%
di-yuan	11	42	22	282062	93,01%	319638	92,04%	131837	93,61%	118371	93,58%
pdh	11	34	24	482640	95,07%	496248	95,58%	157911	92,31%	127270	92,95%
abilene	12	15	66	573606	92,40%	511698	92,47%	137654	94,88%	147656	93,53%
polska	12	18	66	562782	94,01%	856416	92,56%	184766	93,80%	159430	94,17%
nobel-germany	14	21	121	523688	93,40%	405350	93,49%	98332	94,57%	90501	92,78%
nobel-us	14	21	91	648648	92,89%	610428	93,57%	160323	94,80%	148995	94,48%
atlanta	15	22	105	603960	94,75%	283815	94,09%	115087	92,87%	94436	94,24%
newyork	16	49	120	313320	95,73%	202560	92,11%	66081	93,06%	47803	94,21%
geant	22	36	231	242088	94,23%	258720	94,11%	40757	92,94%	37316	93,29%
france	25	45	300	222300	92,60%	133800	95,48%	26046	92,06%	23468	94,13%
norway	27	51	351	177255	93,30%	157599	93,48%	20577	92,65%	17324	95,04%
sun	27	102	65	39455	94,49%	47970	93,84%	24416	95,20%	21245	93,23%
nobel-eu	28	41	378	195804	93,89%	133056	93,02%	20749	94,09%	19212	94,25%
cost266	37	57	666	113886	92,55%	59940	92,96%	8624	95,47%	7588	94,16%
janos-us-ca	39	122	741	36309	95,47%	25935	93,63%	2232	92,76%	2052	95,28%
pioro40	40	89	780	77220	92,81%	72540	93,33%	5485	92,51%	4895	93,54%
germany50	50	88	662	56932	93,61%	52298	94,05%	6092	92,42%	5759	92,94%
zib54	54	81	626	51958	95,76%	51958	94,28%	6404	94,05%	5671	93,18%
ta2	65	108	807	34701	92,96%	32280	95,00%	3984	93,81%	3413	93,88%
Média →					93,88%		93,58%		93,70%		93,91%

A Tabela 8.1 exibe, respectivamente, nas colunas 1, 2, 3 e 4, o nome da instância, a quantidade de vértices e arestas do grafo que representa a rede ótica e a quantidade de requisições por caminhos óticos. Em seguida, a cada par de colunas, são exibidos, também respectivamente, a quantidade de soluções viáveis geradas pelas metaheurísticas GA-PIFRO e ILS-PIFRO e o percentual total do tempo gasto por elas em execuções do Algoritmo de Dijkstra. Cabe ressaltar que, dentre a quantidade de soluções viáveis, são consideradas também as soluções repetidas.

Como nota-se na Tabela 8.1, sempre a maior parte do tempo de execução das metaheurística é tomado por execuções do Algoritmo de Dijkstra, tendo médias entre 93,58% e 93,88%, conforme é mostrado na última linha da Tabela 8.1.

8.4 Experimento II: Apuração do desempenho do Algoritmo CUDA-Yen

Para comprovar o desempenho da proposta de paralelização do Algoritmo CUDA-Yen aqui apresentada, foram realizados experimentos comparativos com diferentes configurações de múltiplas gerações de caminhos candidatos. A configuração de *TMPPC* assemelha-se, em boa parte, com a implementação proposta em [Singh e Singh \(2015\)](#), sofrendo, contudo, adaptações para comportar grandes instâncias.

Como instâncias de dados utilizadas para alimentar os algoritmos, foram utilizados os acervos:

- (i) *Stanford Network Analysis Project* (SNAP) ([Leskovec e Krevl, 2021](#));
- (ii) *Discrete Mathematics and Theoretical Computer Science* (DIMACS) ([Rossi e Ahmed, 2015](#)).

Tais instâncias foram selecionadas por terem quantidades maiores de vértices e arestas em seus grafos que as instâncias *SNDLib*, sendo mais adequadas para tal experimento. O critério para a seleção de instâncias de ambos os grupos foi ter disponibilidade de caminhos para $K = 10$, $K = 100$ e $K = 1000$ entre dois vértices $i - > j$ selecionados aleatoriamente. A grade de processamento é configurada de forma que, quando uma instância de grafo excede o tamanho máximo de um bloco, são utilizados *Cooperative Groups* para que alcance ou chegue o mais próximo possível de atingir tal. Os algoritmos tiveram sua execução permitida de acordo com o tempo necessário para concluir o processamento dos *K-Shortest Loopless Paths* solicitados.

As Tabelas 8.2, 8.3 e 8.4 exibem os resultados de tempo de execução para os valores de K para 10, 100 e 1000, respectivamente. Nas três tabelas, a coluna 1 exibe o nome do grupo de instâncias; a coluna 2, o nome da instância, a coluna 3, o número de vértices do grafo; e a coluna 4 o número de arestas. As colunas agrupadas sob o título *Thread Minimum Percentage per Candidate Speedup (TMPPC)* exibem, respectivamente, os tempos de execução do algoritmo CUDA-Yen para atingir os K caminhos indicados com *TMPPC* configurado em 100%; 50%; 33%; 25%; 12,5%; e 6,25%. A última linha de cada tabela indica a quantidade de instâncias na qual a configuração de *TMPPC* foi a mais veloz.

Conforme visto nas Tabelas 8.2, 8.3 e 8.4, apenas em uma instância e para $K = 100$ uma configuração diferente de *TMPPC*, que não 25% ou 12,5%, é superior. Em um total de 450 execuções (resultado obtido através da multiplicação do número de instâncias $\times$ número de configurações $\times$ três diferentes valores de K) que resulta em 75 comparações entre configurações de *TMPPC*, a configuração *TMPPC* = 25% é superior em 80% das comparações; *TMPPC* = 50% é superior em 16%; e *TMPPC* = 6,25% é superior em 0,013% das comparações. O gráfico da Figura 8.3 exibe o *speedup* médio para cada configuração de *TMPPC* comparado sempre com *TMPPC* em 100%.

Nota-se, pela Figura 8.3, que, além de ocorrer em maior quantidade de comparações, o desempenho da configuração para *TMPPC* em 25% apresenta, em média, um *speedup* maior quando comparado à configuração de *TMPPC* = 100%. Apesar de apresentarem *speedup* médio maior ou próximo a *TMPPC* = 12,5%, as demais configurações para *TMPPC* raramente são as mais velozes, quando comparada às demais utilizadas.

Tabela 8.2: Tempo de execução para atingir *K-Shortest Paths* para $K = 10$

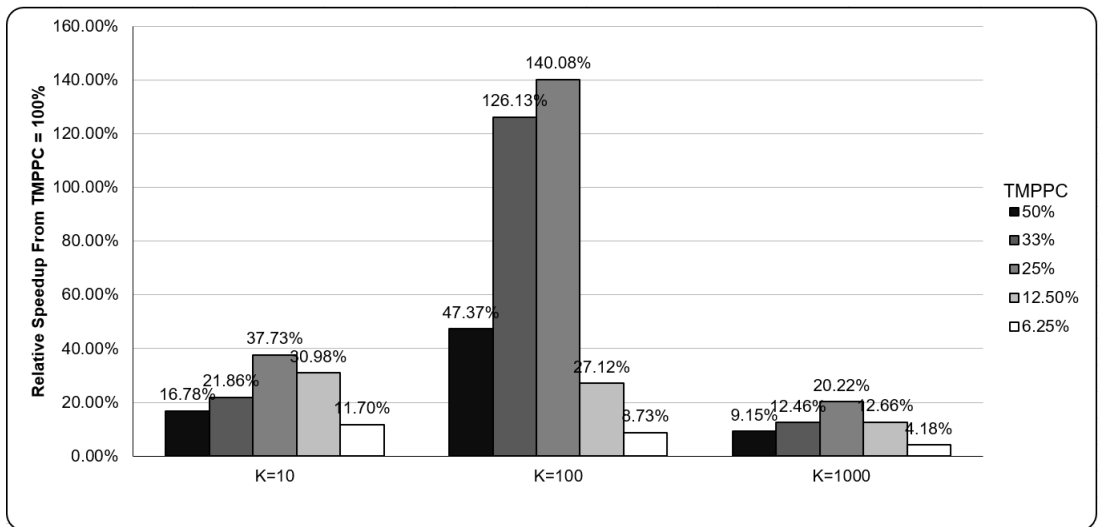
Grupo	Instância	V	E	Porcentagem mínima de <i>Thread</i> por <i>speedup</i> do candidato					
				100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	0,57	0,49	0,47	0,46	0,46	0,59
	ego-Facebook	4039	88234	1,78	1,55	1,48	1,43	1,44	1,79
	p2p-Gnutella04	10876	39994	2,17	1,89	1,81	1,69	1,71	1,98
	Oregon-2	10900	31180	1,70	1,48	1,42	1,32	1,36	1,81
	ego-Gplus	107614	13673453	7357,27	6397,63	6131,06	5709,46	5723,31	6688,43
	amazon0302	262111	1234877	1618,37	1407,28	1348,65	1252,90	1254,55	1471,25
	roadNet-CA	1965206	2766607	27184,76	23638,92	22653,97	21835,15	21873,46	24713,42
	com-Orkut	3072441	117185083	1800221,27	1565409,80	1500184,39	1385960,86	1415520,36	1636564,79
	com-LiveJournal	3997962	34681189	693270,38	602843,81	577725,32	556843,68	567418,90	630245,80
	soc-LiveJournal1	4847571	68993773	1672261,07	1454140,06	1393550,89	1343181,58	1296326,41	1520237,33
	twitter7	17069982	476553560	40673803,46	35368524,74	33894836,21	30669721,65	31530080,20	36976184,96
	com-Friendster	65608366	1806067135	602465568,07	515187450,49	493721306,72	435875958,29	459275634,16	538605061,88
	johnson32-2-4	496	108000	0,27	0,23	0,22	0,22	0,21	0,24
	C-1000-9	1000	450000	2,25	1,96	1,88	1,74	1,74	2,05
DIMACS	sant1000	1000	251000	1,26	1,09	1,05	0,97	0,97	1,14
	mann-a45	1000	533000	2,67	2,32	2,22	2,04	2,07	2,42
	p-hat1500-3	2000	847000	8,47	7,37	7,06	6,80	6,57	7,70
	keller6	3000	5000000	75,00	65,22	62,50	54,24	58,14	68,18
	C4000-5	4000	4000000	80,00	69,57	66,67	64,26	66,02	72,73
	hamming10-2	519000	1000	2,60	2,26	2,16	2,08	2,16	2,36
	NW	1207945	2840208	17154,08	14916,59	14295,06	13178,37	13297,73	15594,61
	NE	1524453	3897636	29708,81	25833,75	24757,35	23862,50	23930,09	27008,01
	CAL	1890815	4657742	44034,64	38290,99	36695,54	35369,19	36135,38	40031,49
	CTR	14081816	34292496	2414503,09	2099567,91	2012085,91	1839359,92	1871707,83	2195002,81
	USA	23947347	58333344	6984644,15	6073603,61	5820536,79	5110155,95	5414452,83	6349676,50
	Total de Melhores Tempos =>			0	0	0	20	5	0

Tabela 8.3: Tempo de execução para atingir *K-Shortest Paths* para $K = 100$

Grupo	Instância	V	E	Porcentagem mínima de <i>Thread</i> por <i>speedup</i> do candidato					
				100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	1,70	1,98	1,42	1,83	1,32	1,03
	ego-Facebook	4039	88234	5,35	3,10	5,94	2,76	5,72	3,24
	p2p-Gnutella04	10876	39994	4,35	3,78	5,44	3,95	5,06	6,99
	Oregon-2	10900	31180	6,80	5,77	5,66	5,46	4,43	4,63
	ego-Gplus	107614	13673453	22071,82	25590,52	18393,19	11818,92	17109,94	13376,86
	amazon0302	262111	1234877	6473,50	5629,13	5394,58	4299,60	4563,66	4413,75
	roadNet-CA	1965206	2766607	81554,29	90615,88	47277,85	43670,30	84293,84	74140,26
	com-Orkut	3072441	117185083	5400663,80	4696229,39	3000368,78	2891921,72	5582081,45	6546259,16
	com-LiveJournal	3997962	34681189	2079811,14	1808531,42	1710901,26	1213687,36	1612256,69	1760491,60
	soc-LiveJournal1	4847571	68993773	6689044,26	4362420,17	5574203,55	3929544,74	3888979,22	4560712,00
	twitter7	17069982	476553560	130678886,61	141474098,98	101684508,64	81347606,91	126120320,79	140928554,88
	com-Friendster	65608366	1806067135	2369862272,27	1545562351,48	987442613,45	951751916,58	1837102536,65	2154420247,52
	johnson32-2-4	496	108000	0,80	0,70	0,67	0,49	0,83	0,65
	C-1000-9	1000	450000	9,00	3,91	3,75	3,61	6,98	4,09
DIMACS	sant1000	1000	251000	5,02	4,37	4,18	3,02	3,95	4,12
	mann-a45	1000	533000	10,66	4,63	6,66	4,85	4,13	6,42
	p-hat1500-3	2000	847000	28,23	20,41	16,94	14,73	19,70	30,80
	keller6	3000	5000000	225,00	260,87	187,50	180,72	232,56	204,55
	C4000-5	4000	4000000	320,00	278,26	266,67	186,05	175,45	192,77
	hamming10-2	519000	1000	5,19	4,51	6,49	6,25	4,02	9,44
	NW	1207945	2840208	51462,23	46783,84	42885,19	39833,17	41335,12	53190,93
	NE	1524453	3897636	118835,26	77501,26	99029,38	57725,00	59090,27	64016,03
	CAL	1890815	4657742	132103,93	141476,76	102406,14	73391,07	76581,99	80062,99
	CTR	14081816	34292496	7243509,28	4599135,82	4448343,65	3878719,83	4390005,63	7186831,30
	USA	23947347	58333344	13969288,30	18220810,83	17461610,38	11220311,89	16243358,49	16999353,00
	Total de Melhores Tempos =>			0	0	0	19	4	1

Tabela 8.4: Tempo de execução para atingir *K-Shortest Paths* para $K = 1000$

Grupo	Instância	V	E	Porcentagem mínima de <i>Thread</i> por <i>speedup</i> do candidato					
				100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	15,91	8,89	11,36	10,57	10,96	12,40
	ego-Facebook	4039	88234	24,95	24,79	24,73	24,06	26,15	26,88
	p2p-Gnutella04	10876	39994	69,60	45,39	43,50	23,60	27,95	47,45
	Oregon-2	10900	31180	27,19	25,69	24,49	22,68	21,62	21,72
	ego-Gplus	107614	13673453	88287,30	85543,13	85097,00	84146,39	844098,61	910522,36
	amazon0302	262111	1234877	38840,98	33774,77	32367,48	30398,18	32145,72	36597,49
	roadNet-CA	1965206	2766607	652434,32	621889,89	571847,63	568538,18	565763,04	591975,80
	com-Orkut	3072441	117185083	25203097,75	23569835,16	21002581,46	20003060,59	23492488,71	25277554,94
	com-LiveJournal	3997962	34681189	11092326,06	11029719,96	1100103,79	1082124,13	11098702,37	11125899,17
	soc-LiveJournal1	4847571	68993773	20067132,79	19266240,92	16296814,21	15490905,26	15533875,34	160283322,66
	twitter7	17069982	476553560	1138866496,77	742739019,63	588054989,73	576738034,56	597541443,57	666499884,16
	com-Friendster	65608366	1806067135	8294517952,96	7882249405,93	7799081815,15	7442102299,91	7444788317,39	7892652148,13
DIMACS	johnson32-2-4	496	108000	6,43	5,59	2,68	2,58	6,64	4,38
	C-1000-9	1000	450000	54,00	46,96	45,00	25,30	28,64	33,87
	sant1000	1000	251000	15,06	15,04	15,00	14,44	14,92	15,69
	mam-a45	1000	533000	74,62	64,89	61,09	59,95	59,38	61,61
	p-hat1500-3	2000	847000	203,28	176,77	148,23	138,49	139,58	147,60
	keller6	3000	5000000	1050,00	1946,09	1838,91	1737,71	1739,23	1795,55
	C4000-5	4000	4000000	960,00	932,87	912,33	901,08	903,37	923,73
	hamming10-2	519000	1000	72,66	71,59	69,20	65,36	66,28	70,31
	NW	1207945	2840208	480314,11	463248,33	450261,76	442897,23	469252,39	490703,05
	NE	1524453	3897636	534758,66	525007,53	522235,05	520700,04	520941,60	526144,24
	CAL	1890815	4657742	792623,56	788983,84	760519,63	724430,29	745790,68	780503,89
	CTR	14081816	34292496	38632049,51	36194814,90	36145030,94	36044637,96	361894650,40	389460078,76
	USA	23947347	58333344	223508612,87	219324864,99	209692883,04	20002495,13	201031245,30	214294177,04
	Total de Melhores Tempos =>			0	0	0	21	3	0

**Fig. 8.3:** Aceleração média obtida por cada configuração TMPPC vs 100%

8.5 Experimento III: Desempenho dos Algoritmos CUDA-Yen-Generalization e CUDA-MPS

Para avaliar o desempenho da proposta de paralelização dos algoritmos CUDA-Yen-Generalization e CUDA-MPS, foram realizados experimentos comparativos com diferentes configurações de múltiplas gerações de caminhos candidatos e comparados ao desempenho da configuração mais veloz vista em (dos Reis et al., 2021).

Assim como no Experimento II, que avalia o desempenho do Algoritmo CUDA-Yen, as instâncias de dados utilizadas para alimentar os algoritmos deste experimento foram utilizados dos acervos:

- (i) *Stanford Network Analysis Project* (SNAP) (Leskovec e Krevl, 2021);
- (ii) *Discrete Mathematics and Theoretical Computer Science* (DIMACS) (Rossi e Ahmed, 2015).

Também de forma idêntica ao Experimento I, o critério para a seleção de instâncias de ambos os grupos foi haver disponibilidade de caminhos para $K = 10$, $K = 100$ e $K = 1000$ entre dois vértices $i \rightarrow j$ selecionados aleatoriamente.

As Tabelas 8.5, 8.6, 8.7, 8.8, 8.9 e 8.10 exibem os resultados de tempo de execução, respectivamente, para os valores de $K = 10$, $K = 100$ e $K = 1000$. Em cada uma destas tabelas, as quatro primeiras colunas definem as instâncias utilizadas nos testes. A primeira coluna mostra o nome do grupo de instâncias; a segunda coluna, o nome da instância; a terceira coluna, o número de vértices $|V|$ do grafo; e a quarta coluna mostra o número $|E|$ de arestas. O tempo de execução do Algoritmo CUDA-Yen é expresso em milisegundos (dos Reis et al., 2021) na quinta coluna. As colunas agrupadas sob o título *Thread Minimum Percentage per Candidate Speedup (TMPPC)*, ou seja, da sexta à décima-primeira colunas, exibem, respectivamente, os tempos de execução do Algoritmo CUDA-Yen-Generalization ou do Algoritmo CUDA-MPS para atingir os K caminhos indicados com *TMPPC* configurado em 100%, 50%, 33%, 25%, 12,5% e 6,25%. Estas colunas apresentam os valores percentuais de *speedup* atingido para as diferentes configurações dos algoritmos. As linhas finais de cada tabela indicam a quantidade de instâncias na qual a respectiva configuração de *TMPPC* foi a mais veloz.

Conforme visto nas Tabelas 8.5, 8.6, 8.7, 8.8, 8.9 e 8.10, em um total de 450 execuções de cada algoritmo (resultado obtido através da multiplicação do número de instâncias $\times$ número de configurações $\times$ três diferentes valores de K), que resultam em 75 diferentes configurações de *TMPPC* para cada um, a configuração *TMPPC*= 50% foi superior em 6 casos para o Algoritmo CUDA-Yen-Generalization. Nas outras 69 configurações, o *TMPPC*= 33% para o Algoritmo CUDA-Yen-Generalization. Para o Algoritmo CUDA-MPS, a configuração de *TMPPC*= 25% é superior em 72 configurações e o *TMPPC*= 33% é superior em 3 casos. Nota-se que, apesar do Algoritmo CUDA-Yen-Generalization apresentar maior desempenho que o Algoritmo CUDA-Yen de dos Reis et al. (2021), o fato de possuir previamente o grafo organizado em *Sorted Forward Star Form* dá ao Algoritmo CUDA-MPS maior velocidade, pois permite diversificar em maior quantidade numérica as *threads* disponíveis para geração simultânea de caminhos candidatos, obtendo, assim, ainda melhor desempenho de execução, quando comparado ao Algoritmo CUDA-Yen-Generalization. Observe ainda que o tempo gasto para organizar o grafo *Sorted*

Tabela 8.5: Avaliação do Algoritmo CUDA-Yen-Generalization: Tempo de execução para atingir K -Shortest Paths para $K = 10$

Grupo	Instance	V	E	CUDA-Yen	Porcentagem Mínima de Thread por speedup do candidato					
					100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	0.46	5.05	14.10	13.99	10.00	-5.18	-21.86
	ego-Facebook	4039	88234	1.43	8.65	9.75	18.64	8.01	-7.63	-19.06
	p2p-Gnutella04	10876	39994	1.69	10.94	12.77	14.81	8.48	-7.83	-19.29
	Oregon-2	10900	31180	1.32	5.66	11.70	14.64	5.16	-3.36	-18.04
	ego-Gplus	107614	13673453	5709.46	6.50	9.77	16.74	9.51	-6.30	-21.18
	amazon0302	262111	1234877	1252.90	6.64	12.26	15.48	8.59	-4.62	-20.36
	roadNet-CA	1965206	2766607	21835.15	5.56	12.63	17.61	6.52	-7.54	-19.44
	com-Orkut	3072441	117185083	1385960.86	9.09	9.97	18.58	8.63	-7.47	-18.10
	com-LiveJournal	3997962	34681189	556843.68	9.92	11.65	17.72	7.14	-5.50	-18.37
	soc-LiveJournal1	4847571	68993773	1343181.58	5.81	11.39	14.05	5.61	-5.20	-22.69
DIMACS	twitter7	17069982	476553560	30669721.65	7.50	10.22	13.41	5.88	-5.76	-19.72
	com-Friendster	65608366	1806067135	435875958.3	6.31	10.17	18.64	5.71	-6.07	-20.25
	johnson32-2-4	496	108000	0.22	8.56	12.23	14.11	7.27	-7.95	-21.01
	C-1000-9	1000	450000	1.74	6.87	10.78	18.78	8.79	-3.72	-21.04
	sant1000	1000	251000	0.97	5.90	9.68	13.15	6.63	-5.92	-19.72
	mann-a45	1000	533000	2.04	7.91	10.76	16.37	7.45	-6.97	-20.56
	p-hat1500-3	2000	847000	6.80	10.96	9.53	14.55	8.54	-6.47	-18.41
	keller6	3000	5000000	54.24	6.80	14.45	14.44	5.22	-7.82	-20.60
	C4000-5	4000	4000000	64.26	9.88	10.56	13.57	6.08	-4.17	-22.68
	hamming10-2	519000	1000	2.08	5.07	11.96	15.35	6.58	-5.49	-19.82
	NW	1207945	2840208	13178.37	6.40	12.40	17.29	7.42	-7.23	-19.40
	NE	1524453	3897636	23862.50	10.96	10.10	14.46	6.21	-5.57	-19.57
	CAL	1890815	4657742	35369.19	10.97	12.99	13.14	9.06	-5.12	-21.12
	CTR	14081816	34292496	1839359.92	8.78	11.25	18.80	8.82	-3.50	-20.91
	USA	23947347	58333344	5110155.95	10.89	14.65	15.27	8.72	-5.62	-19.21
Total de Melhores Tempos Parciais $\Rightarrow$					0	2	23	0	0	0

Tabela 8.6: Avaliação do Algoritmo CUDA-MPS: Tempo de execução para atingir K -Shortest Paths para $K = 10$

Grupo	Instance	V	E	CUDA-Yen	Porcentagem Mínima de Thread por speedup do candidato					
					100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	0.46	8.09	16.13	20.96	22.07	2.49	-8.66
	ego-Facebook	4039	88234	1.43	10.14	15.5	19.73	21.97	6.6	-12.91
	p2p-Gnutella04	10876	39994	1.69	11.75	12.85	19.23	25.76	-10.24	-20.09
	Oregon-2	10900	31180	1.32	10.05	12.86	18.44	25.79	-5.93	-13.64
	ego-Gplus	107614	13673453	5709.46	12.12	12.04	17.81	24.73	-3.8	-20.4
	amazon0302	262111	1234877	1252.90	12.73	14.13	20.72	22.19	1.67	-22.06
	roadNet-CA	1965206	2766607	21835.15	10.6	13.2	17.44	19.11	4.67	-14.92
	com-Orkut	3072441	117185083	1385960.86	13.9	15.78	21.62	21.74	-9.33	-9.25
	com-LiveJournal	3997962	34681189	556843.68	13.22	15.48	16.69	25.68	-6.72	-15.04
	soc-LiveJournal1	4847571	68993773	1343181.58	10.21	12.72	17.65	19.69	-6.41	-10.06
DIMACS	twitter7	17069982	476553560	30669721.65	11.26	12.85	20.99	23.39	-6.81	-18.78
	com-Friendster	65608366	1806067135	435875958.3	9.61	17.04	20.61	25.12	4.11	-21.04
	johnson32-2-4	496	108000	0.22	10.88	15.88	20.16	21.93	7.35	-11.5
	C-1000-9	1000	450000	1.74	8.01	14.65	19.02	20.9	-1.27	-18.69
	sant1000	1000	251000	0.97	12.79	12.56	19.55	24.34	-0.43	-11.1
	mann-a45	1000	533000	2.04	8.69	14.86	16.84	19.57	-5.79	-14.95
	p-hat1500-3	2000	847000	6.80	12.48	16.33	21.03	20.95	-7.26	-16.76
	keller6	3000	5000000	54.24	13.46	12.23	18.24	25.37	7.43	-20.21
	C4000-5	4000	4000000	64.26	9.4	15.23	16.1	22.27	6.15	-12.48
	hamming10-2	519000	1000	2.08	13.02	12.74	17.37	25.47	5.79	-13.24
	NW	1207945	2840208	13178.37	12.09	16.71	16.32	25.16	-10.04	-21.74
	NE	1524453	3897636	23862.50	10.13	17.27	17.36	22.55	7.04	-21.06
	CAL	1890815	4657742	35369.19	8.68	14.98	16.76	23.69	-3.31	-13.26
	CTR	14081816	34292496	1839359.92	8.68	15.75	16.42	22.55	3.8	-10.49
	USA	23947347	58333344	5110155.95	11.73	16.93	18	21.1	4.76	-10.81
Total de Melhores Tempos Parciais $\Rightarrow$					0	0	1	24	0	0

Forward Star Form é compensado ao dispensar o *loop* utilizado pelo Algoritmo CUDA-Yen-Generalization para localizar a aresta de menor custo incremental de um vértice.

O gráfico da Figura 8.4 exibe o *speedup* médio para cada configuração de *TMPPC* comparado ao Algoritmo CUDA-Yen de dos Reis et al. (2021). Nota-se por este gráfico que, além de acontecer em maior quantidade de comparações, o desempenho da configuração para *TMPPC* em 25% apresenta, em média, um *speedup* maior dentre todos os demais

Tabela 8.7: Avaliação do Algoritmo CUDA-Yen-Generalization: Tempo de execução para atingir K -Shortest Paths para $K = 100$

Grupo	Instance	V	E	CUDA-Yen	Porcentagem Mínima de Thread por speedup do candidato					
					100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	1.83	5.06	12.32	15.41	9.77	-7.79	-22.87
	ego-Facebook	4039	88234	2.83	7.15	11.68	14.42	9.81	-7.32	-21.71
	p2p-Gnutella04	10876	39994	3.83	8.10	10.60	13.12	5.51	-3.06	-19.68
	Oregon-2	10900	31180	4.83	10.62	10.50	13.27	5.53	-5.40	-20.19
	ego-Gplus	107614	13673453	5.83	8.87	9.33	18.15	8.41	-5.02	-22.68
	amazon0302	262111	1234877	6.83	7.61	14.75	16.01	9.60	-7.32	-19.91
	roadNet-CA	1965206	2766607	7.83	10.03	14.08	13.42	8.53	-4.23	-22.44
	com-Orkut	3072441	117185083	8.83	9.30	13.49	18.91	9.48	-6.17	-21.45
	com-LiveJournal	3997962	34681189	9.83	10.00	14.77	14.44	8.41	-4.02	-22.47
	soc-LiveJournal1	4847571	68993773	10.83	10.20	14.27	15.18	6.21	-6.87	-18.03
	twitter7	17069982	476553560	11.83	10.92	13.98	14.00	8.56	-3.01	-18.68
DIMACS	com-Friendster	65608366	1806067135	12.83	8.45	12.25	14.20	9.43	-7.65	-22.46
	johnson32-2-4	496	108000	0.49	5.63	13.07	13.55	8.53	-7.48	-20.98
	C-1000-9	1000	450000	3.61	5.51	14.15	13.46	8.98	-7.08	-22.69
	sant1000	1000	251000	3.02	6.36	10.80	15.78	5.11	-6.50	-21.50
	mann-a45	1000	533000	4.85	7.03	10.46	15.27	6.62	-6.79	-20.75
	p-hat1500-3	2000	847000	14.73	7.50	10.42	17.71	7.53	-5.03	-22.52
	keller6	3000	5000000	180.72	6.59	13.18	15.68	6.24	-7.63	-20.71
	C4000-5	4000	4000000	186.05	5.22	12.36	16.97	8.64	-6.17	-22.80
	hamming10-2	519000	1000	6.25	5.11	9.69	15.55	9.41	-7.94	-19.84
	NW	1207945	2840208	39833.17	7.95	12.05	17.49	7.76	-4.27	-19.26
	NE	1524453	3897636	57725	6.54	12.55	17.50	8.12	-6.07	-20.12
	CAL	1890815	4657742	73391.07	9.02	9.23	18.35	9.52	-4.17	-19.10
	CTR	14081816	34292496	3878719.83	10.02	12.59	15.07	6.22	-6.75	-21.34
	USA	23947347	58333344	11220311.89	5.91	12.47	18.28	8.86	-4.88	-20.10
Total de Melhores Tempos Parciais $\Rightarrow$					0	3	23	0	0	0

Tabela 8.8: Avaliação do Algoritmo CUDA-MPS: Tempo de execução para atingir K -Shortest Paths para $K = 100$

Grupo	Instance	V	E	CUDA-Yen	Porcentagem Mínima de Thread por speedup do candidato					
					100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	1.83	10.45	14.08	20	21.13	2.45	-22.37
	ego-Facebook	4039	88234	2.83	9.37	15.43	17.49	19.01	4.13	-9.71
	p2p-Gnutella04	10876	39994	3.83	12.84	15.58	18.38	21.08	-6.87	-18.97
	Oregon-2	10900	31180	4.83	10.72	16.89	16.43	20.23	-9.68	-21.11
	ego-Gplus	107614	13673453	5.83	10.04	16.01	16.24	20.57	-2.38	-12.55
	amazon0302	262111	1234877	6.83	11.18	12.19	21.36	19.11	-10.53	-13.49
	roadNet-CA	1965206	2766607	7.83	12.78	14.94	19.5	23.39	6.99	-17.73
	com-Orkut	3072441	117185083	8.83	12.83	12.74	20.28	22.08	-6.99	-20.35
	com-LiveJournal	3997962	34681189	9.83	9.46	15.08	17.53	20.83	5.87	-20.7
	soc-LiveJournal1	4847571	68993773	10.83	9.55	14.27	17.24	25.37	2.32	-11.96
	twitter7	17069982	476553560	11.83	13.74	14.27	19.54	21.51	-4.38	-14.17
DIMACS	com-Friendster	65608366	1806067135	12.83	10.51	16.93	20.08	23.93	-0.15	-9.39
	johnson32-2-4	496	108000	0.49	8.78	16.93	19.18	22.8	-9.17	-12.36
	C-1000-9	1000	450000	3.61	8.9	14.83	20.62	22.74	-0.47	-18.67
	sant1000	1000	251000	3.02	11.46	13.46	23.84	23.73	-10.98	-8.93
	mann-a45	1000	533000	4.85	13.53	13.2	16.96	22.66	-4.61	-10.88
	p-hat1500-3	2000	847000	14.73	11.13	13.34	19.77	20	5.99	-17.06
	keller6	3000	5000000	180.72	10.27	16.63	20.73	22.94	-4.56	-11.4
	C4000-5	4000	4000000	186.05	10.6	15.97	20.68	24.65	7.11	-18.12
	hamming10-2	519000	1000	6.25	9.45	16.75	19.37	21.8	-4.4	-20.08
	NW	1207945	2840208	39833.17	10.85	16.54	16.85	24.65	-8.51	-13.02
	NE	1524453	3897636	57725	10.47	14.98	21.77	25.94	-1.87	-17.13
	CAL	1890815	4657742	73391.07	12.81	12.37	21.3	25.38	-5.06	-11.08
	CTR	14081816	34292496	3878719.83	13.8	12.94	21.03	24.25	-8.03	-19.68
	USA	23947347	58333344	11220311.89	12.31	12.13	17.71	24.61	-3.45	-14.48
Total de Melhores Tempos Parciais $\Rightarrow$					0	0	2	23	0	0

apurados e isso ocorre para o Algoritmo CUDA-MPS.

Tabela 8.9: Avaliação do Algoritmo CUDA-Yen-Generalization: Tempo de execução para atingir K -Shortest Paths para $K = 1000$

Grupo	Instance	V	E	CUDA-Yen	Porcentagem Mínima de Thread por speedup do candidato					
					100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	10.57	8.88	12.71	14.88	9.27	-4.78	-21.73
	ego-Facebook	4039	88234	24.06	5.31	13.95	17.25	9.63	-3.86	-19.67
	p2p-Gnutella04	10876	39994	23.6	7.71	13.88	14.96	6.64	-5.02	-22.14
	Oregon-2	10900	31180	22.68	5.77	11.06	14.20	9.47	-5.49	-21.80
	ego-Gplus	107614	13673453	84146.39	6.84	13.95	17.55	6.75	-3.40	-21.03
	amazon0302	262111	1234877	30398.18	8.69	9.60	15.46	8.55	-6.28	-22.43
	roadNet-CA	1965206	2766607	568538.18	7.60	10.73	16.31	6.80	-3.90	-19.75
	com-Orkut	3072441	117185083	20003060.59	5.10	13.32	17.09	5.14	-6.68	-18.20
	com-LiveJournal	3997962	34681189	1082124.13	9.84	13.87	13.82	5.16	-5.19	-19.59
	soc-LiveJournal1	4847571	68993773	15490905.26	9.58	10.37	17.14	8.70	-5.20	-22.42
	twitter7	17069982	476553560	576738034.60	7.22	10.88	18.42	7.96	-6.76	-20.99
	com-Friendster	65608366	1806067135	7442102300	7.70	10.35	17.73	5.82	-4.19	-21.68
DIMACS	johnson32-2-4	496	108000	2.58	8.27	10.83	14.37	7.94	-3.93	-20.45
	C-1000-9	1000	450000	25.3	8.14	11.83	16.23	6.25	-7.97	-19.30
	sant1000	1000	251000	14.44	8.91	14.05	16.30	8.32	-7.67	-21.98
	mann-a45	1000	533000	59.95	10.78	10.67	18.53	7.83	-3.01	-22.46
	p-hat1500-3	2000	847000	138.49	5.61	13.10	15.84	8.98	-4.63	-22.86
	keller6	3000	5000000	1737.71	6.28	13.94	18.67	5.27	-6.76	-22.36
	C4000-5	4000	4000000	901.08	5.95	10.73	15.27	8.04	-6.15	-20.39
	hamming10-2	519000	1000	65.36	5.73	10.70	15.11	9.87	-4.39	-21.78
	NW	1207945	2840208	442897.23	10.15	13.14	18.31	9.14	-6.48	-19.02
	NE	1524453	3897636	520700.04	7.39	10.09	13.02	5.02	-6.73	-21.88
	CAL	1890815	4657742	724430.29	10.64	11.52	16.31	5.64	-6.38	-20.45
	CTR	14081816	34292496	36044637.96	10.42	9.16	13.86	6.97	-4.14	-19.98
	USA	23947347	58333344	20002495.13	5.53	13.85	17.60	5.39	-5.29	-20.55
	Total de Melhores Tempos Parciais $\Rightarrow$				0	1	24	0	0	0

Tabela 8.10: Avaliação do Algoritmo CUDA-MPS: Tempo de execução para atingir K -Shortest Paths para $K = 1000$

Grupo	Instance	V	E	CUDA-Yen	Porcentagem Mínima de Thread por speedup do candidato					
					100%	50%	33%	25%	12.5%	6.25%
SNAP	CollegeMsg	1899	59835	10.57	12.3	14.3	17.74	23.15	-8.55	-8.67
	ego-Facebook	4039	88234	24.06	8.25	13.44	17.2	19.13	2.8	-9.74
	p2p-Gnutella04	10876	39994	23.6	8.08	15.89	17.29	22.33	2.73	-8.04
	Oregon-2	10900	31180	22.68	12.88	13.25	20.66	22.01	-10	-15.24
	ego-Gplus	107614	13673453	84146.39	8.25	13.31	19.67	24.41	0.45	-10.82
	amazon0302	262111	1234877	30398.18	11.36	14.47	20.31	21.51	-6.08	-15.16
	roadNet-CA	1965206	2766607	568538.18	13.71	12.57	16.28	25.79	-9.56	-13.98
	com-Orkut	3072441	117185083	20003060.59	8.09	15.76	17.59	22.96	-4.68	-19.7
	com-LiveJournal	3997962	34681189	1082124.13	9	14.35	19.03	19.56	-1.54	-10.48
	soc-LiveJournal1	4847571	68993773	15490905.26	13.09	14.08	18.42	19.03	6.74	-13.25
	twitter7	17069982	476553560	576738034.60	12.14	15.38	17.32	19.88	-4.68	-21.84
	com-Friendster	65608366	1806067135	7442102300	9.03	13.58	20.52	19.67	-1.69	-19.05
DIMACS	johnson32-2-4	496	108000	2.58	12.33	13.33	17.63	19.16	-3.99	-21.17
	C-1000-9	1000	450000	25.3	8.88	12.08	21.49	23.62	3.11	-17.59
	sant1000	1000	251000	14.44	13.95	17.63	17.67	24.33	1.47	-21.29
	mann-a45	1000	533000	59.95	10.15	13.84	18.41	24.9	-8.59	-8.97
	p-hat1500-3	2000	847000	138.49	11.43	17.95	20.44	24.93	6.62	-9.53
	keller6	3000	5000000	1737.71	10.78	16.25	17.84	23.91	0.28	-11.39
	C4000-5	4000	4000000	901.08	10.05	13.69	20.07	25.93	-5.61	-21.87
	hamming10-2	519000	1000	65.36	13.34	15.54	21.12	23.95	3.44	-22.43
	NW	1207945	2840208	442897.23	10.87	13.45	19.08	25.94	-5.03	-22.53
	NE	1524453	3897636	520700.04	12.37	12.78	20.49	21.01	5.24	-18.71
	CAL	1890815	4657742	724430.29	12.76	13.76	17.16	20.94	7.95	-19.21
	CTR	14081816	34292496	36044637.96	13.53	15.92	19.48	21.58	-7.34	-19.72
	USA	23947347	58333344	20002495.13	13.49	17.1	21.27	24.57	-0.3	-14.57
	Total de Melhores Tempos Parciais $\Rightarrow$				0	0	2	23	0	0

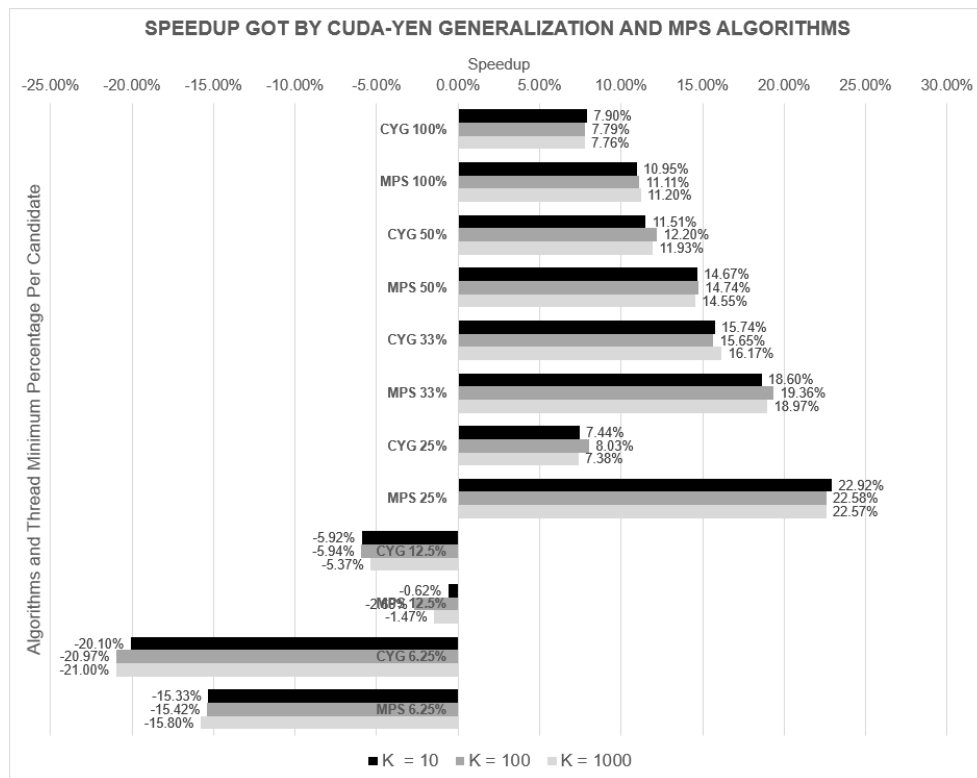


Fig. 8.4: Aceleração média obtida por cada configuração TMPPC.

8.6 Avaliação do desempenho das heurísticas paralelas para PIFRO

As comparações dos desempenhos entre as heurísticas paralelas para PIFRO, ou seja, entre as heurísticas *Parallel Multistart PSC*, *Parallel Multistart Greedy* e *Multistart Paths Pool*, têm seus resultados exibidos nessa seção. Para tanto, foi estipulado um valor alvo, o qual consiste em 50% do valor da melhor qualidade de solução encontrado para a instância, independentemente da estratégia de solução utilizada e, então, a heurística tem seu tempo de execução liberado até atingir este valor.

Os resultados são apresentados nas Subseção 8.6.1, para o Algoritmo CUDA-Multistart PIFRO; Subseção 8.6.2, para o Algoritmo CUDA GA-PIFRO; e Subseção 8.6.3, para o Algoritmo CUDA ILS-PIFRO.

8.6.1 Experimento IV: Desempenho do Algoritmo CUDA-*Multistart PIFRO*

A Tabela 8.11 apresenta os resultados dos experimentos para avaliação do desempenho obtido pelo Algoritmo CUDA-*Multistart PIFRO*, usando as três versões diferentes deste algoritmo, implementadas através de estratégias de paralelização de baixo nível das heurísticas *Parallel Multistart PSC*, *Parallel Multistart Greedy* e *Multistart Paths Pool*.

Tabela 8.11: Desempenho obtido entre as diferentes versões de CUDA-*Multistart PIFRO*

Instâncias		Paralelização Baixo Nível		
Grupo	Nome	<i>Parallel Multistart PSC</i>	<i>Parallel Multistart Greedy</i>	<i>Multistart Paths Pool</i>
SNAP	CollegeMsg	227.25	-1.17%	-2.92%
	ego-Facebook	1781.89	-1.30%	-1.46%
	p2p-Gnutella04	869.95	-1.65%	-1.48%
	Oregon-2	1699.31	-0.76%	-1.99%
	ego-Gplus	5885819.88	-0.22%	-2.12%
	amazon0302	971024.54	-0.58%	-1.75%
	roadNet-CA	10873905.35	-1.81%	-2.26%
	com-Orkut	720088507.20	-2.75%	-1.53%
	com-LiveJournal	277308151.47	-1.81%	-2.56%
	soc-LiveJournal1	1672261065.88	-1.58%	-1.69%
	twitter7	16269521382.47	-1.54%	-2.12%
	com-Friendster	592465568068.26	-1.29%	-2.80%
DIMACS	johnson32-2-4	107.14	-0.83%	-2.01%
	C-1000-9	1800.00	-0.37%	-1.59%
	sant1000	1004.00	-2.43%	-2.64%
	mann-a45	2132.00	-0.63%	-1.61%
	p-hat1500-3	8470.00	-0.89%	-1.12%
	keller6	60000.00	-1.36%	-1.44%
	C4000-5	32000.00	-2.27%	-2.60%
	hamming10-2	2076.00	-1.09%	-2.82%
	NW	13723260.21	-2.24%	-2.29%
	NE	17825288.68	-1.28%	-2.73%
	CAL	35227713.76	-0.48%	-1.03%
	CTR	965801237.71	-1.73%	-2.34%
	USA	6984644152191840.00	-0.29%	-1.55%
Média de melhoria do desempenho			-1.11%	-2.02%

Tabela 8.12: Desempenhos obtidos pela Heurística CUDA GA-PIFRO - *Greedy*

Instâncias		CUDA-GA-PIFRO			
Grupo	Nome	<i>Greedy</i> -Paralelização de Baixo Nível			
		Populações			
		1	2	4	8
SNAP	CollegeMsg	-7.92%	-0.80%	-1.26%	-3.80%
	ego-Facebook	-6.24%	-4.45%	-4.41%	-4.11%
	p2p-Gnutella04	-6.21%	-0.65%	-4.34%	-0.19%
	Oregon-2	-7.43%	-0.79%	-2.38%	-4.11%
	ego-Gplus	-7.95%	-1.01%	-1.70%	-4.01%
	amazon0302	-7.91%	-4.29%	-3.92%	-2.62%
	roadNet-CA	-6.69%	-4.16%	-0.46%	-3.09%
	com-Orkut	-7.04%	-3.29%	-1.88%	-1.45%
	com-LiveJournal	-6.50%	-2.28%	-3.15%	-4.16%
	soc-LiveJournal1	-7.04%	-0.27%	-3.48%	-1.12%
	twitter7	-7.88%	-3.06%	-0.93%	-4.41%
	com-Friendster	-6.68%	-2.93%	-0.71%	-1.80%
DIMACS	johnson32-2-4	-7.49%	-0.41%	-0.87%	-1.75%
	C-1000-9	-6.00%	-2.88%	-1.14%	-1.88%
	sant1000	-7.45%	-0.55%	-4.71%	-2.76%
	mann-a45	-7.91%	-1.46%	-2.09%	-3.14%
	p-hat1500-3	-5.96%	-4.77%	-3.90%	-0.98%
	keller6	-7.04%	-4.54%	-4.55%	-4.08%
	C4000-5	-7.23%	-1.37%	-1.48%	-4.30%
	hamming10-2	-7.22%	-2.81%	-1.79%	-0.94%
	NW	-6.80%	-3.62%	-4.16%	-3.16%
	NE	-7.94%	-1.89%	-0.70%	-2.33%
	CAL	-6.80%	-0.43%	-1.38%	-4.75%
	CTR	-6.77%	-2.12%	-1.17%	-2.47%
	USA	-7.41%	-3.30%	-1.65%	-1.94%
Média de melhoria do desempenho		-7.10%	-2.32%	-2.33%	-2.77%

Na Tabela 8.11, a primeira coluna indica o grupo de instâncias a que pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As três colunas seguintes mostram, respectivamente, o tempo de execução do algoritmo em milissegundos para a heurística paralelizada em baixo nível *Parallel Multistart PSC*, que obteve, em média, o pior desempenho e o percentual de redução de tempo de execução para as estratégias de paralelização de baixo nível *Paralelização Multistart Greedy* e *Multistart Paths Pool*.

Nota-se que as heurísticas *Parallel Multistart Greedy* e *Multistart Paths Pools* tiveram desempenho superior à heurística *Parallel Multistart PSC* em todas as 25 instâncias disponíveis. No entanto, em 23 delas (ou seja, em 92% das instâncias), a heurística *Multistart Paths Pool* tem melhor desempenho dentre todas, atingindo em média 2,02% de melhoria, enquanto a heurística *Parallel Multistart Greedy* apresenta melhoria de desempenho médio de 1.11%.

8.6.2 Experimento V: Desempenho do Algoritmo CUDA GA-PIFRO

As Tabelas 8.12, 8.13 e 8.14 apresentam os resultados dos experimentos para avaliação de desempenho do Algoritmo CUDA GA-PIFRO, obtidos a partir das três versões diferentes desse algoritmo, usando as heurísticas *Parallel PSC*, *Parallel Greedy* e *Parallel Paths Pool*.

A primeira coluna das Tabelas 8.12, 8.13 e 8.14 indica o grupo de instâncias a que pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As quatro colunas restantes mostram o tempo de exe-

Tabela 8.13: Desempenhos obtidos pela Heurística CUDA GA-PIFRO - PSC

Instâncias		CUDA-GA-PIFRO			
Grupo	Nome	PSC - Paralelização de Baixo Nível			
		Populações			
		1	2	4	8
SNAP	CollegeMsg	-7.43%	-4.44%	-3.79%	-2.72%
	ego-Facebook	-7.52%	-2.89%	-2.04%	-2.78%
	p2p-Gnutella04	-7.97%	-4.50%	-4.41%	-3.36%
	Oregon-2	-6.70%	-3.37%	-4.38%	-3.21%
	ego-Gplus	-7.42%	-2.42%	-3.17%	-2.09%
	amazon0302	-5.88%	-3.30%	-4.04%	-3.10%
	roadNet-CA	-7.16%	-2.81%	-3.30%	-3.06%
	com-Orkut	-5.23%	-5.01%	-3.23%	-2.10%
	com-LiveJournal	-5.80%	-3.89%	-2.94%	-2.28%
	soc-LiveJournal1	-6.41%	-2.16%	-3.24%	-2.98%
	twitter7	-5.87%	-3.59%	-2.92%	-2.64%
	com-Friendster	-5.43%	-4.79%	-3.99%	-2.08%
DIMACS	johnson32-2-4	-6.07%	-4.92%	-3.49%	-2.98%
	C-1000-9	-5.60%	-5.57%	-4.76%	-3.74%
	sant1000	-7.85%	-2.72%	-2.24%	-3.08%
	mann-a45	-5.43%	-5.70%	-2.88%	-2.97%
	p-hat1500-3	-7.28%	-2.01%	-3.93%	-3.41%
	keller6	-5.22%	-4.25%	-2.58%	-2.78%
	C4000-5	-6.59%	-3.97%	-4.80%	-3.42%
	hamming10-2	-7.68%	-2.20%	-4.23%	-3.66%
	NW	-7.69%	-4.46%	-3.55%	-3.18%
	NE	-6.49%	-5.64%	-3.11%	-2.80%
	CAL	-7.86%	-3.75%	-4.12%	-3.89%
	CTR	-7.27%	-3.80%	-3.79%	-2.05%
	USA	-5.98%	-3.30%	-4.71%	-3.93%
Média de melhoria do desempenho		-6.63%	-3.82%	-3.59%	-2.97%

ção dos algoritmos paralelizados CUDA-GA-PIFRO-Greedy, CUDA-GA-PIFRO-PSC, e CUDA-GA-PIFRO-Paths-Pool em milissegundos em relação ao tempo de execução obtido pela heurística *Parallel Multistart PSC*, mostrado na terceira coluna da Tabela 8.11. Esse formato de comparação permite avaliar o comportamento dentre todas elas, a partir de uma referência.

Os resultados de cada uma das heurísticas, agrupados nas quatro colunas, representam os números de populações aplicadas no Algoritmo. Nota-se que, para todos os casos, as heurísticas foram testadas para 1, 2, 4 e 8 populações. A Tabela 8.12 mostra os resultados para a heurística CUDA-GA-PIFRO-Greedy. A Tabela 8.13 apresenta os resultados para a heurística CUDA-GA-PIFRO-PSC e a Tabela 8.14 descreve os resultados para a heurística CUDA-GA-PIFRO-Paths-Pool.

Os resultados mostram que, das 25 instâncias disponíveis, a heurística CUDA-GA-PIFRO *Paths Pools*, sendo configurada para utilizar apenas uma população, obteve o melhor desempenho em 20 das instâncias (80%), enquanto a mesma heurística configurada para utilizar duas populações obteve o melhor desempenho em 5 instâncias (20%). Para o grupo de instâncias SNAP, dentre as 12 instâncias avaliadas, a heurística CUDA-GA-PIFRO *Paths Pools* obteve o melhor desempenho em 8 para uma população e em 4 para duas populações. Para o grupo de instâncias, dentre as 13 instâncias avaliadas, essa heurística obteve o melhor desempenho em 12 instâncias para uma população e em uma para duas populações. No entanto, em todas as formas de configuração testadas para a heurística CUDA-GA-PIFRO, o desempenho é sempre superior às heurísticas baseadas em *Multistart*.

Tabela 8.14: Desempenhos obtidos pela Heurística CUDA GA-PIFRO - Hierárquica com *Paths Pool*

Instâncias		CUDA-GA-PIFRO			
Grupo	Nome	Paralelização Hierárquica com <i>Paths Pool</i>			
		Populações			
		1	2	4	8
SNAP	CollegeMsg	-13.22%	-12.65%	-10.19%	-8.43%
	ego-Facebook	-14.51%	-15.40%	-11.86%	-9.96%
	p2p-Gnutella04	-14.24%	-11.63%	-10.03%	-9.61%
	Oregon-2	-15.07%	-16.17%	-11.86%	-8.92%
	ego-Gplus	-14.57%	-12.97%	-10.47%	-7.64%
	amazon0302	-13.34%	-17.90%	-11.19%	-7.41%
	roadNet-CA	-17.36%	-11.20%	-10.46%	-7.56%
	com-Orkut	-20.80%	-15.09%	-10.60%	-8.77%
	com-LiveJournal	-13.77%	-16.86%	-11.51%	-7.47%
	soc-LiveJournal1	-13.77%	-11.66%	-10.17%	-7.05%
	twitter7	-19.10%	-16.39%	-11.72%	-8.53%
	com-Friendster	-17.63%	-15.74%	-10.44%	-9.11%
DIMACS	johnson32-2-4	-18.00%	-15.29%	-11.93%	-9.92%
	C-1000-9	-15.90%	-12.67%	-10.28%	-8.77%
	sant1000	-16.61%	-13.76%	-10.50%	-8.35%
	mann-a45	-14.28%	-12.58%	-10.83%	-9.93%
	p-hat1500-3	-14.49%	-15.96%	-11.39%	-8.53%
	keller6	-14.36%	-11.37%	-11.81%	-7.13%
	C4000-5	-17.97%	-16.23%	-10.46%	-9.45%
	hamming10-2	-19.98%	-13.75%	-11.79%	-8.87%
	NW	-14.31%	-11.58%	-10.02%	-9.63%
	NE	-19.33%	-14.15%	-10.33%	-7.70%
	CAL	-19.22%	-11.24%	-10.48%	-8.05%
	CTR	-13.90%	-12.86%	-11.19%	-8.76%
	USA	-20.19%	-11.30%	-10.16%	-8.46%
Média de melhoria do desempenho		-16.24%	-13.86%	-10.87%	-8.56%

A heurística CUDA-GA-PIFRO-*Greedy* obteve desempenho médio de 7,10% com uma população; 2,32% para duas populações; 2,33% para 4 populações; e 2,77% para 8 populações. Já CUDA-GA-PIFRO-PSC obteve melhoria de desempenho de 6,63% quando configurada para utilizar uma população; 3,82% para 2 populações, 3,59% para 4 populações; e 2,97% para 8 populações.

8.6.3 Experimento VI: Desempenho do Algoritmo CUDA ILS-PIFRO

As Tabelas 8.15, 8.16 e 8.17 apresentam os resultados dos experimentos para avaliação do desempenho obtido usando as três versões diferentes da heurística CUDA ILS-PIFRO, quais sejam, CUDA ILS-PIFRO-*Greedy*, CUDA ILS-PIFRO-PSC e CUDA ILS-PIFRO-*Paths Pool*. A Tabela 8.15 mostra os resultados para a heurística CUDA ILS-PIFRO-*Greedy*; a Tabela 8.16 mostra os resultados para a heurística CUDA ILS-PIFRO-PSC e a Tabela 8.17 mostra os resultados para a heurística CUDA ILS-PIFRO-*Paths Pool*.

A primeira coluna das tabelas indica os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As colunas seguintes mostram o tempo de execução do algoritmo em milissegundos para a heurística paralelizada em baixo nível CUDA-ILS-PIFRO-*Greedy*, CUDA-ILS-PIFRO-PSC e CUDA-ILS-PIFRO-*Paths Pool*, respectivamente. Os resultados de cada uma das heurísticas estão agrupados em três colunas, as quais representam os percentuais de perturbação aplicados a cada modalidade da heurística. Para todos os

Tabela 8.15: Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO-*Greedy*

Instâncias		CUDA-ILS-PIFRO		
Grupo	Nome	<i>Greedy</i> - paralelização de baixo nível % Perturbação		
		10%	30%	50%
SNAP	CollegeMsg	-10,86%	-10,94%	-10,75%
	ego-Facebook	-14,82%	-18,31%	-10,00%
	p2p-Gnutella04	-14,58%	-12,36%	-14,59%
	Oregon-2	-10,13%	-15,69%	-14,36%
	ego-Gplus	-10,70%	-11,02%	-11,65%
	amazon0302	-15,78%	-20,17%	-10,37%
	roadNet-CA	-10,05%	-19,80%	-14,23%
	com-Orkut	-15,61%	-11,32%	-11,28%
	com-LiveJournal	-14,17%	-14,22%	-15,63%
	soc-LiveJournal1	-9,58%	-8,55%	-14,76%
	twitter7	-9,82%	-13,61%	-10,72%
	com-Friendster	-11,02%	-14,67%	-10,59%
DIMACS	johnson32-2-4	-12,82%	-18,57%	-10,27%
	C-1000-9	-9,86%	-9,96%	-15,79%
	sant1000	-14,14%	-13,48%	-10,67%
	mann-a45	-10,16%	-16,64%	-10,19%
	p-hat1500-3	-13,43%	-17,95%	-10,60%
	keller6	-10,15%	-17,98%	-9,12%
	C4000-5	-9,59%	-19,88%	-12,24%
	hamming10-2	-10,53%	-20,52%	-10,59%
	NW	-15,03%	-18,63%	-11,98%
	NE	-9,32%	-20,26%	-11,21%
	CAL	-9,88%	-19,69%	-9,01%
	CTR	-9,43%	-16,26%	-10,82%
	USA	-15,92%	-13,84%	-13,10%
Média de melhoria do desempenho		-11,89%	-15,77%	-11,78%

casos, as heurísticas foram testadas com percentuais de perturbação de 10%, 30% e 50%. A comparação dos desempenhos das heurísticas é feita em relação ao tempo obtido pela heurística *Parallel Multistart PSC*.

As Tabelas 8.15, 8.16 e 8.17 mostram que, das 25 instâncias disponíveis, a heurística CUDA-ILS-PIFRO *Paths Pools*, sendo configurada para um percentual de perturbação de 30%, obteve o melhor desempenho em 20 das instâncias (80%), enquanto a heurística CUDA-ILS-PIFRO-PSC, configurada para um percentual de perturbação também de 30%, obteve o melhor desempenho em três instâncias (12%) e a heurística CUDA-ILS-PIFRO-*Greedy*, configurada com um percentual de perturbação de 30%, obteve o melhor desempenho em apenas uma instância (4%). No entanto, em todas as formas de configuração testadas para CUDA-ILS-PIFRO, o desempenho médio é sempre superior às heurísticas baseadas em *Multistart* e, utilizando percentual de perturbação de 30%, apresenta desempenho superior a CUDA-GA-PIFRO (26% em média de melhoria comparando-se a melhor configuração de cada uma das heurísticas individualmente) e, em quaisquer que sejam suas configurações tendo percentuais próximos aos de melhor desempenho desta heurística, quando utiliza os demais percentuais de perturbação testados.

Tabela 8.16: Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO-PSC

Instâncias		CUDA-ILS-PIFRO		
Grupo	Nome	PSC - paralelização de baixo nível % Perturbação		
		10%	30%	50%
SNAP	CollegeMsg	-10,58%	-16,38%	-11,15%
	ego-Facebook	-15,06%	-12,80%	-14,41%
	p2p-Gnutella04	-13,60%	-20,11%	-15,22%
	Oregon-2	-14,94%	-11,89%	-13,12%
	ego-Gplus	-12,98%	-16,96%	-14,67%
	amazon0302	-10,54%	-10,68%	-9,28%
	roadNet-CA	-9,24%	-18,49%	-13,30%
	com-Orkut	-11,80%	-17,38%	-9,92%
	com-LiveJournal	-9,10%	-19,81%	-11,32%
	soc-LiveJournal1	-11,71%	-10,10%	-15,12%
	twitter7	-10,15%	-11,40%	-12,58%
	com-Friendster	-12,42%	-11,35%	-10,71%
	johnson32-2-4	-13,42%	-18,78%	-14,06%
	C-1000-9	-15,99%	-10,66%	-10,83%
Dimacs	sant1000	-10,04%	-13,98%	-11,34%
	mann-a45	-11,46%	-13,99%	-14,37%
	p-hat1500-3	-9,05%	-12,89%	-10,20%
	keller6	-10,43%	-18,62%	-9,91%
	C4000-5	-14,85%	-15,51%	-15,98%
	hamming10-2	-10,58%	-15,39%	-11,19%
	NW	-14,77%	-15,33%	-12,40%
	NE	-10,62%	-17,59%	-10,35%
	CAL	-11,10%	-18,11%	-10,15%
	CTR	-15,31%	-17,70%	-9,65%
	USA	-15,05%	-9,37%	-9,89%
Média de melhoria do desempenho		-12,19%	-15,01%	-12,04%

Tabela 8.17: Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO-*Paths Pool*

Instâncias		CUDA-ILS-PIFRO		
Grupo	Nome	Paralelização hierárquica com <i>Paths Pool</i> % Perturbação		
		10%	30%	50%
SNAP	CollegeMsg	-10,87%	-23,11%	-11,71%
	ego-Facebook	-10,90%	-27,95%	-13,86%
	p2p-Gnutella04	-11,10%	-19,95%	-12,27%
	Oregon-2	-15,01%	-19,96%	-11,33%
	ego-Gplus	-15,76%	-20,18%	-15,01%
	amazon0302	-14,83%	-21,34%	-13,14%
	roadNet-CA	-9,54%	-15,12%	-11,40%
	com-Orkut	-14,81%	-25,03%	-14,80%
	com-LiveJournal	-13,38%	-15,34%	-11,94%
	soc-LiveJournal1	-14,69%	-22,39%	-14,52%
	twitter7	-11,33%	-19,85%	-14,81%
Dimacs	com-Friendster	-12,95%	-25,73%	-13,84%
	johnson32-2-4	-12,41%	-22,64%	-10,94%
	C-1000-9	-15,15%	-19,03%	-15,33%
	sant1000	-9,15%	-21,83%	-9,88%
	mann-a45	-14,98%	-17,31%	-10,08%
	p-hat1500-3	-14,42%	-17,80%	-13,43%
	keller6	-12,19%	-27,92%	-10,61%
	C4000-5	-15,63%	-16,70%	-14,35%
	hamming10-2	-13,28%	-23,01%	-13,50%
	NW	-11,95%	-26,34%	-9,91%
	NE	-9,46%	-25,60%	-13,18%
	CAL	-13,92%	-27,71%	-15,66%
	CTR	-13,28%	-21,17%	-13,61%
	USA	-12,43%	-21,36%	-12,43%
Média de melhoria do desempenho		-12,94%	-21,78%	-12,86%

8.7 Avaliação do desempenho das melhores heurísticas paralelas para PIFRO

Nas seções 8.7.1, 8.6.2 e 8.6.3 são detalhadas, respectivamente, os desempenhos das heurísticas CUDA *Multistart*-PIFRO, CUDA GA-PIFRO e CUDA ILS-PIFRO nas configurações que obtiveram melhores resultados, ou seja:

- CUDA *Multistart*-PIFRO *Paths Pool*;
- CUDA GA-PIFRO *Paths Pool* Monopopulacional; e
- CUDA ILS-PIFRO *Paths Pool* com percentual de perturbação de solução fixado em 30%.

Os resultados dos experimentos detalhados para estas heurísticas são exibidos nas Tabelas 8.18, 8.19 e 8.20.

8.7.1 Experimento VII: Melhor desempenho da heurística CUDA *Multistart*-PIFRO

A Tabela 8.18 detalha os resultados dos experimentos realizados para a heurística CUDA-*MULTSTART*-PIFRO *Paths Pool*, ou seja, na configuração na qual esta heurística apresentou o melhor desempenho.

Tabela 8.18: Detalhamento do desempenho da versão mais eficiente da heurística CUDA *Multistart*-PIFRO

Grupo	Instância	CUDA Multistart-PIFRO <i>Paths Pool</i> - Desempenho				
		Média	Mínimo	Máximo	Desvio Padrão	Variância
SNAP	CollegeMsg	233,89	227,89	246,89	9,71	4,15%
	ego-Facebook	1807,93	1738,93	1933,93	98,88	5,47%
	p2p-Gnutella04	882,84	847,84	937,84	45,37	5,14%
	Oregon-2	1733,08	1619,08	1857,08	119,04	6,87%
	ego-Gplus	6010429,46	5881040,46	6542295,46	350447,60	5,83%
	amazon0302	988013,93	889296,93	1076213,93	93507,80	9,46%
	roadNet-CA	11071068,80	10524164,80	11365711,80	427028,46	3,86%
	com-Orkut	731081846,08	705227968,08	751929601,08	23395492,52	3,20%
	com-LiveJournal	282334135,24	275356564,24	307050271,24	16653652,95	5,90%
	soc-LiveJournal1	1700527265,07	1613685435,07	1834698084,07	111347730,95	6,55%
	twitter7	16519878735,58	15982271082,58	17744109795,58	902943273,18	5,47%
	com-Friendster	600103793351,14	565142862486,14	654170505626,14	44854203270,59	7,47%
DIMACS	johnson32-2-4	109,29	101,29	117,29	8,00	7,32%
	C-1000-9	1828,68	1761,68	1994,68	119,95	6,56%
	sant1000	1028,43	969,43	1080,43	55,54	5,40%
	mann-a45	2166,39	1961,39	2317,39	178,68	8,25%
	p-hat1500-3	8565,04	8039,04	9150,04	555,76	6,49%
	keller6	60864,86	55769,86	65995,86	5113,01	8,40%
	C4000-5	32831,54	30963,54	33764,54	1426,27	4,34%
	hamming10-2	2098,69	1978,69	2194,69	108,22	5,16%
	NW	14037614,74	12972828,74	14231388,74	677654,15	4,83%
	NE	18312520,76	16518565,76	19155786,76	1346867,15	7,35%
	CAL	35590181,27	33556359,27	36488258,27	1502165,44	4,22%
	CTR	982524245,73	969625790,73	1017559090,73	24803937,03	2,52%
	USA	7092717276044925,00	6589315389409492,00	7499379180061896,00	455888046692650,94	6,43%
	Médias até o valor alvo				18237359377820,69	5,87%

A primeira coluna da Tabela 8.18 indica os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o

nome de cada instância. As cinco colunas seguintes mostram, respectivamente, a média do tempo das execuções até atingir o valor alvo, o menor tempo dentre as execuções testadas, o maior tempo entre as execuções, o desvio padrão do tempo de execução apurado entre todas as execuções e a variância no tempo de execução apurado entre todas as execuções.

Nota-se, na Tabela 8.18, que a variância média do tempo de execução para atingir o valor alvo das instância é de 5,87%, sendo a maior variação de 9,48% (instância **amazon302**) e a menor de 2,52% (instância **CTR**).

8.7.2 Experimento VIII: Melhor desempenho da heurística CUDA GA-PIFRO

A Tabela 8.19 detalha os resultados experimentos da heurística CUDA-GA-PIFRO *Paths Pool* Monopopulacional, ou seja, na configuração na qual a heurística CUDA-GA-PIFRO apresentou o melhor desempenho.

Tabela 8.19: Detalhamento da performance da versão mais eficiente da heurística CUDA GA-PIFRO

Grupo	Instância	CUDA GA-FIP <i>Paths Pool</i> - Desempenho				
		Média	Mínimo	Máximo	Desvio Padrão	Variância
SNAP	CollegeMsg	257,29	248,29	279,29	15,95	6,20%
	ego-Facebook	2040,49	1888,49	2069,49	97,22	4,76%
	p2p-Gnutella04	993,87	982,87	1066,87	45,65	4,59%
	Oregon-2	1955,37	1827,37	2129,37	151,58	7,75%
	ego-Gplus	6743129,02	6626650,02	7287188,02	352580,49	5,23%
	amazon0302	1100586,73	1075577,73	1174834,73	51623,91	4,69%
	roadNet-CA	12762072,94	11754451,94	13763295,94	1004423,70	7,87%
	com-Orkut	869882276,63	822313289,63	912230294,63	44983758,07	5,17%
	com-LiveJournal	315496914,66	295331034,66	327608493,66	16305354,04	5,17%
	soc-LiveJournal1	1902515061,93	1729352789,93	1995371913,93	135014654,60	7,10%
	twitter7	19377641764,78	18010694771,78	19818556942,78	942630707,58	4,86%
	com-Friendster	696897547740,70	663113332053,70	765754970553,70	52310011565,66	7,51%
DIMACS	johnson32-2-4	126,42	121,42	136,42	7,64	6,04%
	C-1000-9	2086,20	1884,20	2235,20	176,17	8,44%
	sant1000	1170,79	1065,79	1205,79	72,86	6,22%
	mann-a45	2436,44	2375,44	2668,44	154,59	6,35%
	p-hat1500-3	9697,58	9455,58	10462,58	525,65	5,42%
	keller6	68616,18	67157,18	70036,18	1439,54	2,10%
	C4000-5	37749,55	34293,55	39795,55	2780,95	7,37%
	hamming10-2	2490,70	2257,70	2553,70	155,92	6,26%
	NW	15687119,21	15236278,21	17015815,21	925152,47	5,90%
	NE	21271778,42	20112547,42	21718815,42	829031,11	3,90%
	CAL	41997399,23	40293271,23	43906504,23	1807585,26	4,30%
	CTR	1100017183,47	1007110946,47	1170575663,47	81986565,64	7,45%
	USA	8394896249954088,00	7885294890315811,00	9049199193675188,00	583449385877609,50	6,95%
	Médias até o valor alvo				23340116871449,43	5,90%

Na primeira coluna da Tabela 8.19 são indicados os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As cinco colunas seguintes mostram, respectivamente, a média do tempo das execuções até atingir o valor alvo, o menor tempo dentre as execuções testadas, o maior tempo entre as execuções, o desvio padrão do tempo de execução apurado dentre todas as execuções e a variância no tempo de execução apurado entre todas as execuções.

Nota-se, nessa tabela, que a variância média do tempo de execução para atingir o valor alvo das instância é de 5,90%, sendo a maior variação de 8,44% (instância **C-1000-9**) e a menor 2,10% (instância **keller6**).

8.7.3 Experimento IX: Melhor desempenho da heurística CUDA ILS-PIFRO

Nesta seção, através da tabela 8.20 são detalhados os resultados dos experimentos da heurística CUDA-ILS-PIFRO Paths Pool, com percentual de perturbação de soluções configurado para 30%, ou seja, na configuração na qual a heurística apresentou a melhor performance.

Tabela 8.20: Detalhamento do desempenho da versão mais eficiente da heurística CUDA ILS-PIFRO

Grupo	Instância	CUDA ILS-FIP <i>Paths Pool</i> - desempenho				
		Média	Mínimo	Máximo	Desvio Padrão	Variância
SNAP	CollegeMsg	279,77	265,77	301,77	18,15	6,49%
	ego-Facebook	2279,96	2116,96	2392,96	138,75	6,09%
	p2p-Gnutella04	1043,54	977,54	1084,54	53,98	5,17%
	Oregon-2	2038,57	1967,57	2083,57	58,48	2,87%
	ego-Gplus	7073669,46	6705964,46	7147522,46	236514,52	3,34%
	amazon0302	1178267,98	1069860,98	1246923,98	89272,08	7,58%
	roadNet-CA	12518410,01	11911063,01	13639041,01	876602,55	7,00%
	com-Orkut	900306766,47	842148268,47	937949736,47	48265455,82	5,36%
	com-LiveJournal	319858056,94	302729528,94	351288081,94	24627782,46	7,70%
	soc-LiveJournal1	2046633805,49	2010662780,49	2136657033,49	64900755,01	3,17%
	twitter7	19499040687,89	18692858537,89	19981185219,89	650919688,89	3,34%
	com-Friendster	744899988377,71	680741580069,71	808914811314,71	64086629026,73	8,60%
DIMACS	johnson32-2-4	131,39	123,39	139,39	8,00	6,09%
	C-1000-9	2142,59	2014,59	2177,59	85,81	4,00%
	sant1000	1223,22	1125,22	1237,22	61,02	4,99%
	manu-a45	2500,98	2295,98	2705,98	205,00	8,20%
	p-hat1500-3	9977,71	9657,71	10319,71	331,06	3,32%
	keller6	76749,96	74610,96	80524,96	2994,48	3,90%
	C4000-5	37343,47	35616,47	39787,47	2095,75	5,61%
	hamming10-2	2553,66	2476,66	2796,66	167,02	6,54%
	NW	17338578,38	15768676,38	17558484,38	976077,49	5,63%
	NE	22389160,03	20758153,03	24525968,03	1889557,36	8,44%
	CAL	44989360,27	44323656,27	47290968,27	1557000,41	3,46%
	CTR	1170264406,54	1148739571,54	1186901839,54	19133223,28	1,63%
	USA	8476875604376562,00	8046477152845204,00	9147032202801240,00	554613095090691,60	6,54%
Médias até o valor alvo					22187119807914,63	5,40%

Na primeira coluna da Tabela 8.20 são indicados os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As cinco colunas seguintes mostram, respectivamente, a média do tempo das execuções até atingir o valor alvo, o menor tempo dentre as execuções testadas, o maior tempo entre as execuções, o desvio padrão do tempo de execução apurado entre todas as execuções e a variância no tempo de execução apurado entre todas as execuções.

Nota-se na Tabela 8.20 que a variância média do tempo de execução para atingir o valor alvo das instâncias é de 5,40%, sendo a maior variação de 8,60% (instância *com-Friendster*) e a menor 1,63% (instância *CTR*).

8.7.4 Experimento X: Gráficos do desempenho das heurísticas paralelizadas CUDA PIFRO

Nesta seção são detalhadas, respectivamente, os desempenhos das heurísticas CUDA *Multistart*-PIFRO, CUDA GA-PIFRO e CUDA ILS-PIFRO nas configurações que obtiveram melhores resultados, ou seja:

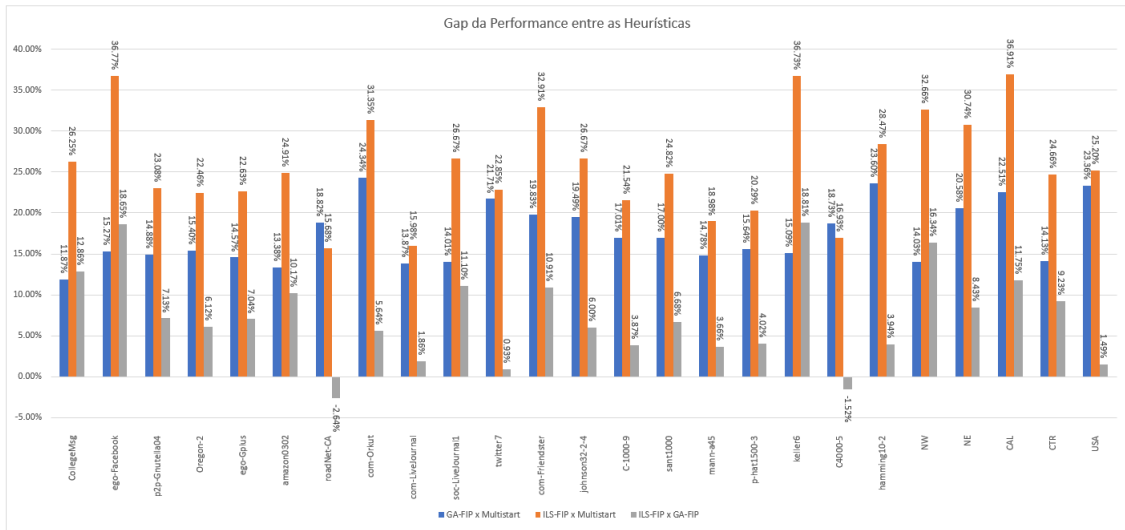


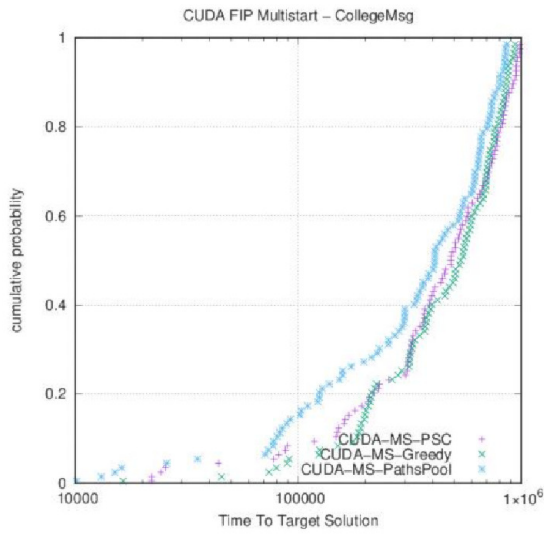
Fig. 8.5: Gap entre o tempo demandado pelas heurísticas para atingir o valor.

- CUDA *Multistart*-PIFRO *Paths Pool*;
- CUDA GA-PIFRO *Paths Pool* Monopopulacional; e
- CUDA ILS-PIFRO *Paths Pool* com percentual de perturbação de solução fixado em 30%.

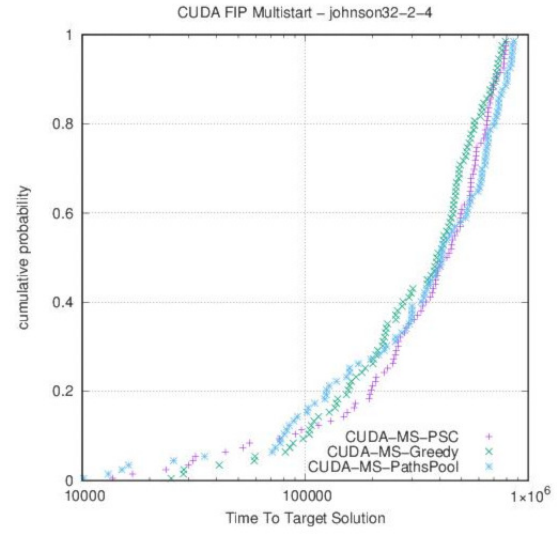
O gráfico da Figura 8.5 mostra o *gap* da comparação de desempenho entre as heurísticas paralelas CUDA para PIFRO. No eixo horizontal do gráfico são listadas as instâncias analisadas do problema, enquanto no eixo vertical é mostrada a escala do *gap* entre as heurísticas (em percentuais). Conforme visto na legenda do gráfico, nas barras azuis são mostrados os valores de *gap* entre as heurísticas CUDA GA-PIFRO e CUDA *Multistart* PIFRO; nas barras laranja, é mostrado o *gap* entre as heurísticas CUDA ILS PIFRO e CUDA *Multistart* PIFRO; e, nas barras cinza, é mostrado o *gap* entre as heurísticas CUDA ILS PIFRO e CUDA GA PIFRO. O gráfico mostra que as heurísticas CUDA GA-PIFRO e CUDA ILS-PIFRO são sempre superiores à heurística CUDA *Multistart* PIFRO em no mínimo 11,87%. Essa situação foi encontrada na instância **CollegeMsg** no *gap* entre as heurísticas CUDA GA-PIFRO e CUDA *Multistart* PIFRO e de até 36,91%, quando observada a instância **CAL** no *gap* entre as heurísticas CUDA ILS-PIFRO e CUDA *Multistart* PIFRO. Nota-se ainda que a heurística CUDA ILS-PIFRO sempre apresenta desempenho superior à heurística CUDA GA-PIFRO, exceto em duas instâncias (**roadNet-CA** e **C4000-5**, quando a heurística GA-PIFRO tem valor de *gap*, respectivamente, 2,64% e 1,52% superior). Portanto, a heurística CUDA ILS-PIFRO tem desempenho superior em 92% das instâncias avaliadas. Ressalta-se ainda que os valores de *gap* de desempenho entre as heurísticas CUDA ILS-PIFRO e CUDA GA-PIFRO estão entre 0.93% e 18.81% nas 23 instâncias nas quais foi superior.

Os gráficos seguintes exibem os *TTT-Plot* das instâncias **CollegeMSG** e **Johnson32-4**, em um total de 100 execuções, com tempo de execução liberado até atingirem o valor alvo de 50% do melhor resultado encontrado pela heurística CUDA ILS-PIFRO (heurística que registrou os resultados com melhor desempenho) para as instâncias.

Os gráficos 8.6(a) e 8.6(b) mostram o *TTT-Plot* das diferentes versões de CUDA *Multistart*-PIFRO para as instâncias **CollegeMSG** e **Johnson32-4**.



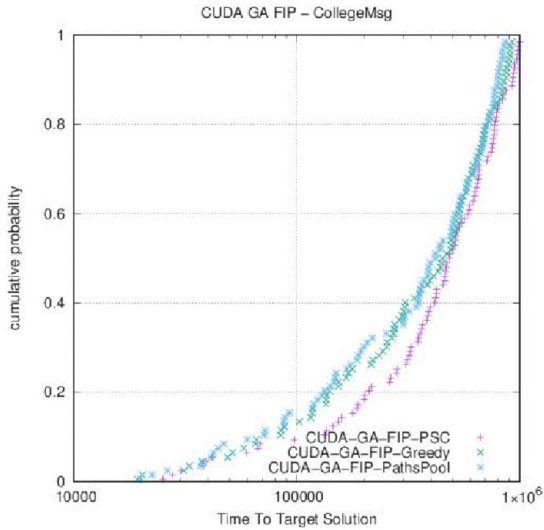
(a) TTT-Plot da instância CollegeMsg para atingir o valor alvo através da heurística Multistart PIFRO.



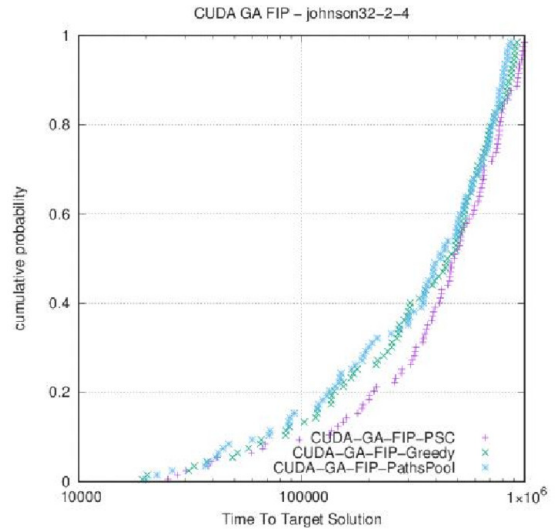
(b) TTT-Plot da instância johnson32-2-4 para atingir o valor alvo através da heurística Multistart PIFRO.

Fig. 8.6: TTT-Plot das diferentes versões de CUDA Multistart PIFRO para as instâncias CollegeMSG e Johnson32-4.

Os gráficos 8.7(a) e 8.7(b) mostram o *TTT-Plot* das diferentes versões de CUDA GA-PIFRO para as instâncias CollegeMSG e Johnson32-4.



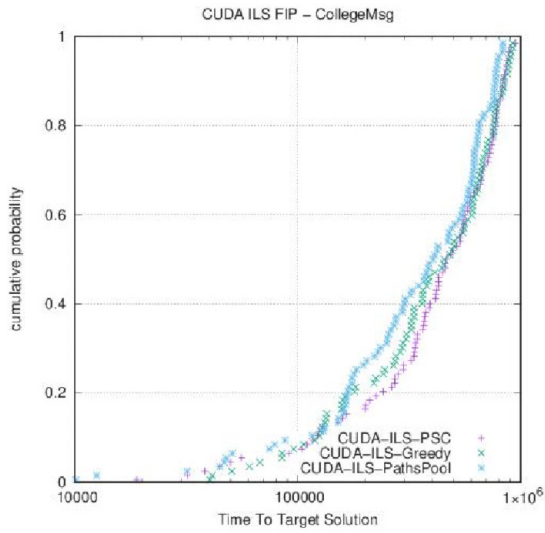
(a) TTT-Plot da instância CollegeMsg para atingir o valor alvo através da heurística CUDA-GA-PIFRO.



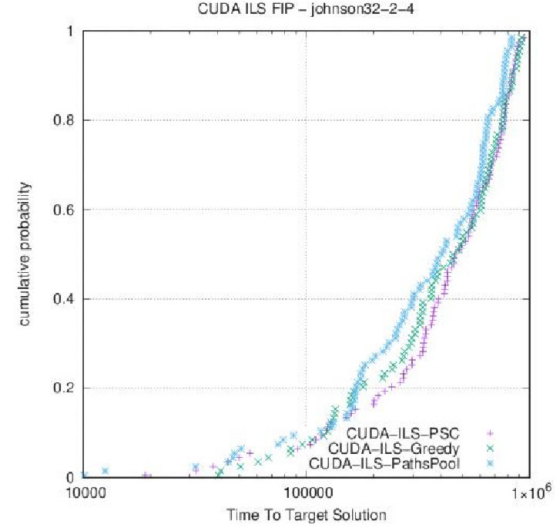
(b) TTT-Plot da instância johnson32-2-4 para atingir o valor alvo através da heurística CUDA-GA-PIFRO.

Fig. 8.7: TTT-Plot das diferentes versões de CUDA-GA PIFRO para as instâncias CollegeMSG e Johnson32-4.

Os gráficos 8.8(a) e 8.8(b) mostram o *TTT-Plot* das diferentes versões de CUDA ILS-PIFRO para as instâncias CollegeMSG e Johnson32-4.



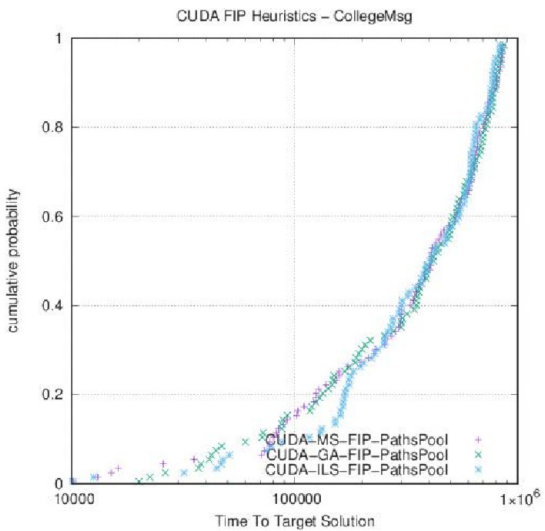
(a) TTT-Plot da instância CollegeMsg para atingir o valor alvo através da heurística CUDA-ILS-PIFRO.



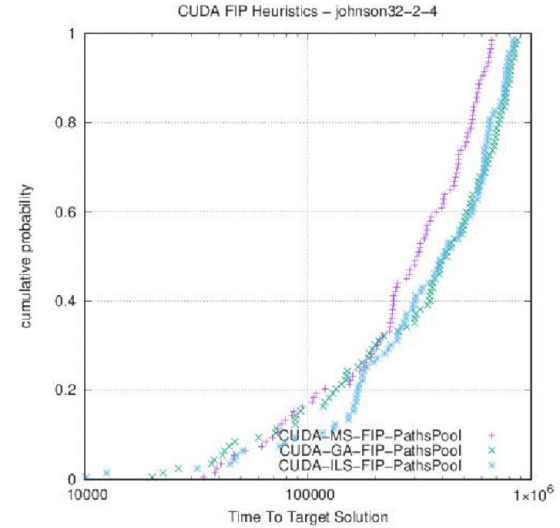
(b) TTT-Plot da instância johnson32-2-4 para atingir o valor alvo através da heurística CUDA-ILS-PIFRO.

Fig. 8.8: TTT-Plot das diferentes versões de CUDA-ILS PIFRO para as instâncias CollegeMSG e Johnson32-4.

Os gráficos 8.9(a) e 8.9(b) mostram o *TTT-Plot* das diferentes versões da heurísticas CUDA para as instâncias CollegeMSG e Johnson32-4:



(a) TTT-Plot da instância CollegeMsg para atingir o valor alvo através das heurísticas CUDA para PIFRO.



(b) TTT-Plot da instância johnson32-2-4 para atingir o valor alvo através das heurísticas CUDA para PIFRO.

Fig. 8.9: TTT-Plot das diferentes versões da heurísticas CUDA para PIFRO para as instâncias CollegeMSG e Johnson32-4.

8.8 Apuração da qualidade de solução das heurísticas paralelas para PIFRO

A comparação da qualidade final das soluções encontradas entre as heurísticas paralelas para PIFRO tem seus resultados exibidos nessa seção. Para tanto, foi estipulada a geração de 100.000 soluções como critério de parada para as heurísticas, independentemente da estratégia de solução utilizada e, então, a heurística tem seu tempo de execução liberado.

A comparação da qualidade final das soluções obtidas entre as heurísticas *Parallel Multistart* PSC, *Parallel Multistart Greedy* e *Multistart Paths Pool* tem seus resultados exibidos nas Subseções 8.8.1, 8.8.2 e 8.8.3.

8.8.1 Experimento XI: Melhoria da qualidade de solução da heurística CUDA *Multistart*-PIFRO

A Tabela 8.21 apresenta os resultados dos experimentos para avaliação da qualidade de soluções obtidas usando as três versões diferentes da heurística CUDA *Multistart* PIFRO, implementadas através de estratégias de paralelização de baixo nível.

Tabela 8.21: Desempenho obtido entre as diferentes versões da heurística CUDA-*Multistart* PIFRO

Instâncias		Paralelização Baixo Nível		
Grupo	Nome	<i>Parallel Multistart PSC</i>	<i>Parallel Multistart Greedy</i>	<i>Multistart Paths Pool</i>
SNAP	CollegeMsg	227.25	-1.15%	-1.95%
	ego-Facebook	1781.89	-1.79%	-1.88%
	p2p-Gnutella04	869.95	-2.13%	-2.81%
	Oregon-2	1699.31	-2.26%	-1.10%
	ego-Gplus	5885819.88	-2.28%	-2.90%
	amazon0302	971024.54	-0.35%	-2.19%
	roadNet-CA	10873905.35	-1.69%	-1.92%
	com-Orkut	720088507.20	-1.20%	-1.09%
	com-LiveJournal	277308151.47	-2.37%	-2.42%
	soc-LiveJournalI	1672261065.88	-2.81%	-1.89%
	twitter7	16269521382.47	-2.00%	-2.90%
	com-Friendster	592465568068.26	-1.05%	-2.88%
DIMACS	johnson32-2-4	107.14	-0.36%	-1.06%
	C-1000-9	1800.00	-0.27%	-2.22%
	sant1000	1004.00	-0.38%	-2.37%
	mann-a45	2132.00	-1.56%	-2.46%
	p-hat1500-3	8470.00	-0.45%	-2.19%
	keller6	60000.00	-2.67%	-1.32%
	C4000-5	32000.00	-1.55%	-2.31%
	hamming10-2	2076.00	-0.24%	-2.22%
	NW	13723260.21	-1.84%	-1.07%
	NE	17825288.68	-2.81%	-2.37%
	CAL	35227713.76	-1.15%	-1.17%
	CTR	965801237.71	-0.42%	-2.38%
	USA	6984644152191840.00	-1.48%	-2.54%
Média de melhoria do desempenho			-1.45%	-2.06%

Como pode ser visto na Tabela 8.21, a primeira coluna indica o grupo de instâncias a que pertence os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As três colunas seguintes mostram, respectivamente, a qualidade de solução final encontrada para a heurística paralelizada em baixo nível *Parallel Multistart* PSC, a qual obteve, em média, os piores valores e o percentual de redução de custo final para a solução das instâncias em cada execução para as estratégias de paralelização de baixo nível *Paralelização Multistart Greedy* e *Multistart Paths Pool*.

Analisando-se esta tabela, nota-se que as heurísticas *Parallel Multistart Greedy* e *Multistart Paths Pools* tiveram desempenho superior à heurística *Parallel Multistart PSC* em todas as 25 instâncias disponíveis. Porém, em 19 delas (ou seja, em 76% das instâncias), a heurística *Multistart Paths Pool* apresenta melhor qualidade de solução, atingindo em média 2,06% de redução no custo das soluções, enquanto a heurística *Parallel Multistart Greedy* apresenta melhoria de desempenho médio de 1,45%, com os melhores resultados atingidos em 6 das instâncias (24%).

8.8.2 Experimento XII: Melhoria da qualidade de solução da heurística CUDA GA-PIFRO

As Tabelas 8.22, 8.23 e 8.22 apresentam os resultados dos experimentos para avaliação da qualidade das soluções obtidas usando as três versões diferentes da heurística CUDA GA-PIFRO.

Tabela 8.22: Desempenhos obtidos entre as diferentes versões da heurística CUDA GA-PIFRO-*Greedy*

Instâncias		CUDA-GA-PIFRO - <i>Greedy</i>			
Grupo	Nome	<i>Greedy</i> -Paralelização de Baixo Nível			
		Populações			
		1	2	4	8
SNAP	CollegeMsg	-1.09%	-2.30%	2.19%	1.44%
	ego-Facebook	-2.18%	-2.45%	-0.76%	3.19%
	p2p-Gnutella04	-1.47%	-1.75%	2.24%	3.43%
	Oregon-2	-3.85%	-1.03%	-2.93%	-0.69%
	ego-Gplus	-2.35%	-1.21%	-2.40%	4.10%
	amazon0302	-2.23%	-2.54%	-0.71%	3.27%
	roadNet-CA	-3.27%	-2.66%	-0.51%	-3.75%
	com-Orkut	-3.61%	-1.66%	1.66%	0.77%
	com-LiveJournal	-2.24%	-1.95%	2.48%	3.03%
	soc-LiveJournal1	-2.60%	-1.44%	-1.96%	3.58%
	twitter7	-2.98%	-1.50%	-2.06%	3.44%
	com-Friendster	-1.77%	-1.03%	-0.92%	-0.31%
DIMACS	johnson32-2-4	-3.57%	-2.77%	2.98%	-2.76%
	C-1000-9	-3.52%	-2.11%	2.42%	2.40%
	sant1000	-1.54%	-2.01%	1.11%	2.73%
	mann-a45	-2.14%	-1.26%	0.07%	2.36%
	p-hat1500-3	-1.56%	-2.37%	-1.18%	-3.68%
	keller6	-1.44%	-2.73%	-1.01%	4.01%
	C4000-5	-1.06%	-2.45%	0.90%	-2.29%
	hamming10-2	-3.20%	-2.31%	0.88%	2.32%
	NW	-3.85%	-1.87%	0.20%	2.98%
	NE	-2.72%	-2.73%	2.70%	2.41%
	CAL	-2.54%	-2.74%	0.51%	2.51%
	CTR	-3.27%	-2.04%	0.30%	-0.81%
	USA	-1.20%	-1.08%	2.56%	-2.79%
Média de melhoria da qualidade de solução		-2.45%	-2.00%	0.35%	1.24%

A primeira coluna destas tabelas indica o grupo de instâncias a que pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As quatro colunas seguintes em cada uma das tabelas mostram, respectivamente, o custo das soluções encontradas para cada uma das instâncias com as heurísticas paralelizadas em baixo nível CUDA-GA-PIFRO-*Greedy*, CUDA-GA-PIFRO-PSC e CUDA-GA-PIFRO-*Paths-Pool*, tendo em vista diferentes números de populações aplicadas no algoritmo. Nota-se que, para todos os casos, as heurísticas foram testadas para 1, 2, 4 e 8 populações. A comparação das qualidades das soluções das heurísticas é feita em relação aos respectivos resultados obtidos pela heurística *Parallel Multistart PSC*,

Tabela 8.23: Desempenhos obtidos entre as diferentes versões de CUDA GA-PIFRO-PSC

Instâncias		CUDA-GA-PIFRO-PSC			
Grupo	Nome	PSC-Paralelização de Baixo Nível			
		Populações			
		1	2	4	8
SNAP	CollegeMsg	-1.57%	-2.53%	2.48%	4.52%
	ego-Facebook	-3.74%	-1.40%	-2.69%	-2.56%
	p2p-Gnutella04	-3.92%	-2.47%	1.87%	-2.91%
	Oregon-2	-3.85%	-1.97%	2.53%	1.57%
	ego-Gplus	-3.34%	-2.57%	2.12%	-0.18%
	amazon0302	-3.83%	-2.08%	2.43%	3.67%
	roadNet-CA	-3.34%	-2.27%	1.13%	-1.09%
	com-Orkut	-2.29%	-2.88%	2.43%	0.51%
	com-LiveJournal	-2.98%	-2.33%	1.09%	3.83%
	soc-LiveJournal1	-2.96%	-2.61%	-0.35%	2.88%
	twitter7	-1.24%	-2.84%	1.17%	-3.46%
	com-Friendster	-1.60%	-1.43%	2.59%	-2.70%
DIMACS	johnson32-2-4	-3.80%	-1.54%	2.20%	-0.24%
	C-1000-9	-1.43%	-2.62%	2.80%	-0.50%
	sant1000	-2.19%	-2.06%	-2.49%	2.48%
	mann-a45	-1.10%	-2.46%	1.32%	2.26%
	p-hat1500-3	-1.73%	-2.82%	-1.14%	-3.44%
	keller6	-1.83%	-1.07%	1.01%	2.96%
	C4000-5	-3.45%	-1.10%	-0.29%	2.45%
	hamming10-2	-2.40%	-2.31%	2.41%	0.52%
	NW	-1.25%	-1.99%	1.07%	4.77%
	NE	-1.39%	-2.73%	-2.20%	-2.28%
	CAL	-2.47%	-1.26%	0.81%	3.89%
	CTR	-1.80%	-2.40%	-2.52%	4.41%
	USA	-3.31%	-2.58%	1.39%	3.71%
Média de melhoria da qualidade de solução		-2.51%	-2.17%	0.85%	1.00%

Tabela 8.24: Desempenhos obtidos entre as diferentes versões da heurística CUDA GA-PIFRO-*Paths Pool*

Instâncias		CUDA-GA-PIFRO- <i>Paths Pool</i>			
Grupo	Nome	Paralelização	Hierárquica	Com	
		<i>Paths Pool</i>			
		Populações			
		1	2	4	8
SNAP	CollegeMsg	-5.04%	-3.94%	-1.96%	-2.02%
	ego-Facebook	-4.30%	-3.84%	-2.83%	-1.00%
	p2p-Gnutella04	-5.15%	-3.17%	-3.12%	-2.91%
	Oregon-2	-5.64%	-3.28%	-2.74%	-2.91%
	ego-Gplus	-5.99%	-2.63%	-1.76%	-1.60%
	amazon0302	-3.53%	-3.51%	-3.31%	-1.95%
	roadNet-CA	-4.40%	-3.79%	-3.63%	-2.17%
	com-Orkut	-3.78%	-3.27%	-1.84%	-2.34%
	com-LiveJournal	-5.05%	-2.64%	-2.97%	-1.23%
	soc-LiveJournal1	-3.80%	-3.65%	-1.34%	-1.65%
	twitter7	-4.56%	-2.97%	-3.07%	-1.57%
	com-Friendster	-4.08%	-2.00%	-1.78%	-2.92%
DIMACS	johnson32-2-4	-5.11%	-2.95%	-2.74%	-1.03%
	C-1000-9	-4.82%	-2.05%	-3.02%	-2.42%
	sant1000	-3.46%	-2.65%	-2.07%	-2.43%
	mann-a45	-3.34%	-2.28%	-1.67%	-1.03%
	p-hat1500-3	-5.96%	-3.17%	-1.66%	-1.10%
	keller6	-4.11%	-2.45%	-3.81%	-1.17%
	C4000-5	-5.44%	-2.82%	-2.61%	-1.41%
	hamming10-2	-3.75%	-2.20%	-3.50%	-1.14%
	NW	-5.07%	-3.14%	-1.92%	-1.45%
	NE	-3.86%	-2.28%	-1.04%	-1.68%
	CAL	-4.82%	-2.54%	-2.02%	-2.65%
	CTR	-3.14%	-3.13%	-1.29%	-2.12%
	USA	-3.49%	-3.64%	-2.19%	-1.14%
Média de melhoria da qualidade de solução		-4.47%	-2.96%	-2.40%	-1.80%

assim como é feito para todas as demais heurísticas apresentadas neste trabalho, o que permite a comparação da evolução de performance entre todas elas.

É interessante observar, frente aos resultados das Tabelas 8.22, 8.23 e 8.22, que das 25 instâncias disponíveis, a heurística CUDA-GA-PIFRO *Paths Pools*, sendo configurada para utilizar apenas uma população, obteve o melhor desempenho em 22 das instâncias (88%), enquanto as heurísticas CUDA-GA-PIFRO-PSC, CUDA-GA-PIFRO-*Greedy* e CUDA-GA-PIFRO *Paths Pools*, configuradas para gerar indivíduos em duas populações, obtiveram a melhor qualidade de solução para uma instância cada (4%). Nota-se também que a heurística CUDA-GA-PIFRO *Paths Pools* monopopulacional, que é a melhor configuração para a heurística, tem 146% em média de melhoria na qualidade das soluções obtidas quando comparada a CUDA-Multistart-PIFRO *Paths Pool*.

8.8.3 Experimento XIII: Melhoria da qualidade de solução da heurística CUDA ILS-PIFRO

As Tabelas 8.25, 8.26 e 8.27 apresentam os resultados dos experimentos para avaliação da qualidade das soluções obtidas usando as duas versões diferentes da heurística CUDA ILS-PIFRO.

Tabela 8.25: Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO-*Greedy*

Instâncias		CUDA-ILS-PIFRO- <i>Greedy</i>		
Grupo	Nome	<i>Greedy</i> - paralelização de baixo nível		
		% Perturbação		
		10	30	50
SNAP	CollegeMsg	-0.87%	-8.32%	-4.82%
	ego-Facebook	-2.24%	-11.73%	-3.17%
	p2p-Gnutella04	-0.42%	-8.76%	-5.68%
	Oregon-2	-0.01%	-11.15%	-6.77%
	ego-Gplus	-0.02%	-9.47%	-3.77%
	amazon0302	-1.99%	-8.31%	-4.92%
	roadNet-CA	-0.30%	-11.92%	-6.55%
	com-Orkut	-0.62%	-11.76%	-4.31%
	com-LiveJournal	-0.39%	-10.13%	-6.96%
	soc-LiveJournal1	-2.04%	-10.53%	-3.57%
DIMACS	twitter7	-1.33%	-8.49%	-3.47%
	com-Friendster	-1.03%	-10.91%	-6.01%
	johnson32-2-4	-2.67%	-11.94%	-4.33%
	C-1000-9	-1.80%	-9.04%	-7.57%
	sant1000	-1.00%	-7.29%	-5.25%
	mann-a45	-2.52%	-7.96%	-5.04%
	p-hat1500-3	-0.20%	-8.73%	-7.20%
	keller6	-2.94%	-11.69%	-4.09%
	C4000-5	-1.24%	-9.10%	-6.56%
	hamming10-2	-1.31%	-10.88%	-7.57%
	NW	-1.12%	-10.24%	-7.02%
	NE	-2.42%	-8.78%	-3.40%
	CAL	-0.86%	-9.41%	-4.28%
	CTR	-2.15%	-11.68%	-7.73%
	USA	-2.90%	-8.05%	-5.28%
Média de melhoria da qualidade de solução		-1.38%	-9.21%	-5.41%

Nas primeiras colunas das Tabelas 8.25, 8.26 e 8.27 são indicados os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS.

Tabela 8.26: Desempenhos obtidos entre as diferentes versões de CUDA ILS-PIFRO-PSC

Instâncias		CUDA-ILS-PIFRO-PSC		
Grupo	Nome	PSC - paralelização de baixo nível		
		% Perturbação		
		10	30	50
SNAP	CollegeMsg	-0.85%	-6.64%	-5.88%
	ego-Facebook	-2.67%	-6.39%	-3.38%
	p2p-Gnutella04	-2.59%	-3.86%	-4.08%
	Oregon-2	-2.49%	-3.93%	-4.26%
	ego-Gplus	-2.64%	-7.77%	-4.85%
	amazon0302	-1.08%	-7.04%	-4.25%
	roadNet-CA	-2.87%	-4.90%	-4.43%
	com-Orkut	-0.77%	-5.21%	-3.78%
	com-LiveJournal	-2.23%	-7.18%	-4.17%
	soc-LiveJournal1	-0.68%	-3.20%	-3.08%
	twitter7	-2.02%	-8.19%	-5.88%
	com-Friendster	-2.97%	-3.02%	-4.52%
DIMACS	johnson32-2-4	-1.33%	-7.91%	-5.67%
	C-1000-9	-2.47%	-6.82%	-5.31%
	sant1000	-2.48%	-6.19%	-5.88%
	mann-a45	-0.74%	-5.17%	-3.52%
	p-hat1500-3	-2.06%	-8.80%	-3.60%
	keller6	-2.53%	-7.77%	-5.43%
	C4000-5	-2.15%	-7.29%	-5.34%
	hamming10-2	-2.83%	-5.52%	-4.82%
	NW	-1.22%	-5.91%	-3.41%
	NE	-1.66%	-4.74%	-4.93%
	CAL	-2.02%	-3.47%	-3.82%
	CTR	-2.87%	-5.73%	-3.43%
	USA	-1.67%	-4.51%	-4.02%
Média de melhoria da qualidade de solução		-2.00%	-5.89%	-4.47%

A segunda coluna indica o nome de cada instância. As três colunas seguintes em cada uma das tabelas mostram, respectivamente, a qualidade das soluções obtidas para as heurísticas paralelizadas em baixo nível CUDA-ILS-PIFRO-*Greedy*, CUDA-ILS-PIFRO-PSC, e CUDA-ILS-PIFRO-*Paths-Pool*, representando os percentuais de perturbação aplicados a cada modalidade da heurística. Nota-se que, para todos os casos, as heurísticas foram testadas com percentuais de perturbação de 10%, 30% e 50%. A comparação dos desempenhos das heurísticas é feita em relação aos respectivos resultados quanto aos valores de tempo obtidos pela heurística *Parallel Multistart PSC*, assim como é feito para todas as demais heurísticas apresentadas neste trabalho.

Nota-se que, das 25 instâncias disponíveis, a heurística CUDA-ILS-PIFRO *Paths Pools*, sendo configurada para um percentual de perturbação de 30%, obteve a melhor qualidade de soluções em 10 das instâncias (40%); configurada para percentual de perturbação em 50%, obteve a melhor qualidade de solução em 5 das instâncias (20%); a heurística CUDA-ILS-PIFRO-PSC, configurada para um percentual de perturbação também de 30%, obteve a melhor qualidade de solução em uma instância (4%); e a heurística CUDA-ILS-PIFRO-*Greedy*, configurada com um percentual de perturbação de 30%, obteve a melhor qualidade de solução em 9 instâncias (36%). No entanto, cabe frisar que a qualidade de solução média de CUDA-ILS-PIFRO é sempre superior às heurísticas baseadas em *Multistart* e, utilizando percentual de perturbação de 30%, apresenta desempenho superior à heurística CUDA-GA-PIFRO (48% em média de melhoria comparando-se a melhor configuração de

Tabela 8.27: Desempenhos obtidos entre as diferentes versões de CUDA
ILS-PIFRO-*Paths Pool*

Instâncias		CUDA-ILS-PIFRO- <i>Paths Pool</i>		
Grupo	Nome	Paralelização hierárquica com <i>Paths Pool</i>		
		% Perturbação		
		10	30	50
SNAP	CollegeMsg	-4.35%	-7.47%	-7.42%
	ego-Facebook	-3.76%	-11.07%	-6.12%
	p2p-Gnutella04	-4.15%	-8.65%	-9.49%
	Oregon-2	-3.94%	-9.26%	-6.10%
	ego-Gplus	-3.45%	-7.41%	-10.27%
	amazon0302	-4.65%	-10.41%	-9.66%
	roadNet-CA	-4.20%	-11.05%	-8.46%
	com-Orkut	-3.43%	-8.75%	-6.47%
	com-LiveJournal	-4.50%	-11.17%	-7.43%
	soc-LiveJournal1	-4.26%	-10.91%	-10.76%
	twitter7	-4.48%	-8.10%	-9.04%
	com-Friendster	-4.17%	-9.23%	-8.58%
DIMACS	johnson32-2-4	-4.57%	-10.39%	-6.04%
	C-1000-9	-3.01%	-10.05%	-7.64%
	sant1000	-4.07%	-11.97%	-8.43%
	mann-a45	-3.51%	-10.32%	-6.52%
	p-hat1500-3	-4.13%	-8.26%	-6.71%
	keller6	-3.14%	-7.47%	-8.41%
	C4000-5	-4.99%	-8.17%	-10.53%
	hamming10-2	-3.15%	-9.18%	-10.53%
	NW	-3.41%	-11.61%	-8.11%
	NE	-4.15%	-8.39%	-6.03%
	CAL	-4.57%	-7.65%	-9.51%
	CTR	-3.75%	-9.97%	-9.85%
	USA	-3.38%	-10.70%	-9.03%
Média de melhoria da qualidade de solução		-2.00%	-5.89%	-4.47%

cada uma das heurísticas individualmente) e, em quaisquer que sejam suas configurações tendo percentuais próximos aos de melhor desempenho desta heurística quando utiliza percentuais dos demais percentuais de perturbação testados.

8.9 Análise da qualidade de solução das melhores heurísticas paralelas para PIFRO

Nas seções 8.9.1, 8.9.2 e 8.9.3 são detalhadas, respectivamente, os desempenhos das heurísticas CUDA *Multistart*-PIFRO, CUDA GA-PIFRO e CUDA ILS-PIFRO nas configurações que obtiveram melhores resultados, ou seja:

- CUDA *Multistart*-PIFRO *Paths Pool*;
- CUDA GA-PIFRO *Paths Pool* Monopopulacional; e
- CUDA ILS-PIFRO *Paths Pool* com percentual de perturbação de solução fixado em 30%;

Os resultados dos experimentos detalhados são exibidos nas Tabelas 8.28, 8.29 e 8.30

8.9.1 Experimento XIV: Detalhamento da versão de melhor qualidade de solução da heurística CUDA *Multistart*-PIFRO

A Tabela 8.28 detalha os resultados dos experimentos da heurística CUDA-*Multistart*-PIFRO *Paths Pool*, ou seja, na configuração na qual a heurística apresentou o melhor desempenho.

Tabela 8.28: Detalhamento da qualidade das soluções da versão mais eficiente da heurística CUDA *Multistart*-PIFRO

Grupo	Instância	CUDA <i>Multistart</i> -FIP <i>Paths Pool</i> - Desempenho				
		Média	Mínimo	Máximo	Desvio Padrão	Variância
SNAP	CollegeMsg	231,69	225,69	235,69	5,03	2,17%
	ego-Facebook	1815,41	1774,41	1838,41	32,42	1,79%
	p2p-Gnutella04	894,37	872,37	917,37	22,50	2,52%
	Oregon-2	1717,94	1688,94	1739,94	25,58	1,49%
	ego-Gplus	6056428,15	5875797,15	6127297,15	129680,52	2,14%
	amazon0302	992323,49	966611,49	1021584,49	27505,59	2,77%
	roadNet-CA	11083006,04	10838393,04	11381877,04	272193,02	2,46%
	com-Orkut	727930582,93	713400835,93	742667491,93	14633450,20	2,01%
	com-LiveJournal	284014911,61	276974271,61	289124353,61	6100566,94	2,15%
	soc-LiveJournal1	1703804136,27	1661801712,27	1751140190,27	44695766,54	2,62%
	twitter7	16741473045,28	16503157774,28	17182162995,28	344492353,71	2,06%
	com-Friendster	609546607018,60	598397204820,60	623054693887,60	12347532461,47	2,03%
DIMACS	johnson32-2-4	108,27	106,27	111,27	2,52	2,32%
	C-1000-9	1840,00	1809,00	1883,00	37,16	2,02%
	sant1000	1027,79	998,79	1045,79	23,71	2,31%
	mann-a45	2184,53	2120,53	2216,53	48,88	2,24%
	p-hat1500-3	8655,08	8402,08	8844,08	221,77	2,56%
	keller6	60794,60	59762,60	62079,60	1160,80	1,91%
	C4000-5	32739,50	32029,50	33294,50	634,08	1,94%
	hamming10-2	2122,08	2086,08	2180,08	47,43	2,23%
	NW	13870665,82	13525611,82	14268825,82	371923,09	2,68%
	NE	18248291,63	17799102,63	18665257,63	433177,39	2,37%
	CAL	35640064,96	35140919,96	36351149,96	608200,11	1,71%
	CTR	988754866,13	972245734,13	1015667551,13	21917642,23	2,22%
	USA	7162074591459973,00	7033983411843706,00	7310920952016594,00	138598334828530,78	1,94%
	Médias até o valor alvo				5544444641828,54	2,19%

Na primeira coluna da Tabela 8.28 são indicados os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As cinco colunas seguintes mostram, respectivamente, a média da qualidade das soluções (custo) das execuções até atingir o valor alvo, o menor custo (melhor qualidade) obtido entre as execuções testadas, o maior custo (pior qualidade) entre as execuções, o desvio padrão do custo (ou qualidade das soluções) apurado entre todas as execuções e a variância no custo (qualidade) apurado entre todas as execuções.

Nota-se que a variância média do custo das soluções das instâncias é de 2.77%, sendo a maior variação de 2.77% (instância **amazon0302**) e a menor 1.49% (instância **Oregon-2**).

8.9.2 Experimento XV: Detalhamento da versão de melhor qualidade de solução da heurística CUDA GA-PIFRO

A Tabela 8.29 detalha os resultados dos experimentos da heurística CUDA-GA-PIFRO *Paths Pool* Monopopulacional, ou seja, na configuração na qual a heurística apresentou o melhor desempenho.

Na primeira coluna da Tabela 8.29 são indicados os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna

Tabela 8.29: Detalhamento da qualidade das soluções da versão mais eficiente da heurística CUDA GA-PIFRO

Grupo	Instância	CUDA GA-FIP <i>Paths Pool</i> - Desempenho				
		Média	Mínimo	Máximo	Desvio Padrão	Variância
SNAP	CollegeMsg	238,71	234,71	242,71	4,00	1,68%
	ego-Facebook	1858,45	1838,45	1878,45	20,00	1,08%
	p2p-Gnutella04	914,75	900,75	925,75	12,53	1,37%
	Oregon-2	1795,11	1769,11	1820,11	25,50	1,42%
	ego-Gplus	6238260,21	6163664,21	6362049,21	100203,87	1,61%
	amazon0302	1005267,98	985446,98	1025234,98	19894,04	1,98%
	roadNet-CA	11352414,77	11208357,77	11542940,77	167828,46	1,48%
	com-Orkut	747331605,09	732985129,09	756518349,09	11860509,28	1,59%
	com-LiveJournal	291314015,26	287484463,26	296003513,26	4266752,73	1,46%
	soc-LiveJournal1	1735766028,77	1710063772,77	1765562436,77	27774489,49	1,60%
	twitter7	17011543247,73	16808074576,73	17226314487,73	209145407,46	1,23%
	com-Friendster	616623345642,46	607704831308,46	628343640332,46	10351051826,39	1,68%
DIMACS	johnson32-2-4	112,61	110,61	114,61	2,00	1,78%
	C-1000-9	1886,72	1849,72	1909,72	30,27	1,60%
	sant1000	1038,73	1026,73	1058,73	16,17	1,56%
	mann-a45	2203,28	2174,28	2244,28	35,17	1,60%
	p-hat1500-3	8974,59	8832,59	9144,59	156,21	1,74%
	keller6	62467,33	61601,33	63290,33	844,59	1,35%
	C4000-5	33741,42	33384,42	34163,42	389,95	1,16%
	hamming10-2	2153,88	2123,88	2176,88	26,58	1,23%
	NW	14418373,76	14168836,76	14628434,76	230081,38	1,60%
	NE	18514050,21	18268305,21	18741303,21	236559,24	1,28%
	CAL	36926898,97	36369118,97	37537425,97	584351,92	1,58%
	CTR	996110399,57	984622930,57	1010755865,57	13098230,86	1,31%
	USA	7228348935224954,00	7148874154861334,00	7343661579149012,00	97941638424140,22	1,35%
	Médias até o valor alvo				3918090278473,53	1,49%

indica o nome de cada instância. As cinco colunas seguintes mostram, respectivamente, a média da qualidade das soluções (custo) das execuções até atingir o valor alvo, o menor custo (melhor qualidade) obtido entre as execuções testadas, o maior custo (pior qualidade) entre as execuções, o desvio padrão do custo (ou qualidade das soluções) apurado entre todas as execuções e a variância no custo (qualidade) apurado entre todas as execuções.

Nota-se na Tabela 8.29 que a variância média do custo das soluções das instâncias é de 1,49%. A maior variação encontrada é de 1,98% (instância **amazon0302**) e a menor 1,08% (instância **ego-Facebook**).

8.9.3 Experimento XVI: Detalhamento da versão de melhor qualidade de solução da heurística CUDA ILS-PIFRO

A Tabela 8.30 detalha os resultados dos experimentos da heurística CUDA-ILS-PIFRO *Paths Pool* Monopopulacional, ou seja, na configuração na qual a heurística apresentou o melhor desempenho.

Na primeira coluna da Tabela 8.30 são indicados os grupos de instâncias aos quais pertencem os problemas avaliados, com duas opções: SNAP e DIMACS. A segunda coluna indica o nome de cada instância. As cinco colunas seguintes mostram, respectivamente, a média da qualidade das soluções (custo) das execuções até atingir o valor alvo, o menor custo (melhor qualidade) obtido entre as execuções testadas, o maior custo (pior qualidade) entre as execuções, o desvio padrão do custo (ou qualidade das soluções) apurado entre todas as execuções e a variância no custo (qualidade) apurado entre todas as execuções.

A partir da Tabela 8.30, observa-se que a variância média do custo das soluções das instâncias é de 1,45%, sendo a maior variação de 1,82% (instância **CTR**) e a menor 1,16%

Tabela 8.30: Detalhamento do desempenho da versão mais eficiente da heurística CUDA ILS-PIFRO

Grupo	Instância	CUDA ILS-FIP <i>Paths Pool</i> - desempenho				
		Média	Mínimo	Máximo	Desvio Padrão	Variância
SNAP	CollegeMsg	244,22	241,22	247,22	3,00	1,23%
	ego-Facebook	1858,45	1826,45	1883,45	28,57	1,54%
	p2p-Gnutella04	914,75	903,75	931,75	14,11	1,54%
	Oregon-2	1795,11	1773,11	1815,11	21,01	1,17%
	ego-Gplus	6238260,21	6160170,21	6306684,21	73310,12	1,18%
	amazon0302	1005267,98	986044,98	1015372,98	14898,36	1,48%
	roadNet-CA	11352414,77	11192618,77	11487923,77	147818,86	1,30%
	com-Orkut	747331605,09	737496764,09	757309533,09	9906470,61	1,33%
	com-LiveJournal	291314015,26	285666530,26	295115692,26	4754532,88	1,63%
	soc-LiveJournal1	1735766028,77	1704770213,77	1756292698,77	25937910,44	1,49%
	twitter7	17011543247,73	16791927452,73	17185426523,73	197191958,07	1,16%
	com-Friendster	616623345642,46	609105981536,46	623392155990,46	7146354981,85	1,16%
DIMACS	johnson32-2-4	112,61	110,61	114,61	2,00	1,78%
	C-1000-9	1886,72	1849,72	1912,72	31,66	1,68%
	sant1000	1038,73	1025,73	1057,73	16,09	1,55%
	mann-a45	2203,28	2168,28	2232,28	32,05	1,45%
	p-hat1500-3	8974,59	8870,59	9080,59	105,00	1,17%
	keller6	62467,33	61748,33	63377,33	816,36	1,31%
	C4000-5	33741,42	33291,42	34258,42	483,89	1,43%
	hamming10-2	2153,88	2127,88	2196,88	34,85	1,62%
	NW	14418373,76	14241210,76	14643736,76	201743,40	1,40%
	NE	18514050,21	18261248,21	18847527,21	294063,15	1,59%
	CAL	36926898,97	36379926,97	37504297,97	562254,11	1,52%
	CTR	996110399,57	977304505,57	1013482411,57	18093688,28	1,82%
	USA	7228348935224954,00	7103419722453745,00	7352450308044489,00	124515522124082,22	1,72%
	Médias até o valor alvo				4980917026372,04	1,45%

(instância `twitter7`).

8.9.4 Experimento XVII: Detalhamento gráfico das qualidades de soluções das heurística paralelizadas de PIFRO

Nesta seção são detalhadas, respectivamente, as qualidade de soluções das heurísticas CUDA Multistart-PIFRO, CUDA GA-PIFRO e CUDA ILS-PIFRO nas configurações que obtiveram melhores resultados, ou seja:

- CUDA *Multistart*-PIFRO *Paths Pool*;
- CUDA GA-PIFRO *Paths Pool* Monopopulacional e;
- CUDA ILS-PIFRO *Paths Pool* com percentual de perturbação de solução fixado em 30%.

O gráfico da Figura 8.10 mostra o *gap* da qualidade das soluções entre as heurísticas paralelas para PIFRO. No eixo horizontal deste gráfico são listadas as instâncias analisadas do problema, enquanto no eixo vertical é mostrada a escala do *gap* entre as heurísticas (em percentuais). Conforme visto na legenda do gráfico, nas barras azuis são mostrados os *gaps* entre a heurística CUDA GA-PIFRO e *Multistart* PIFRO; nas barras laranja, o *gap* entre CUDA ILS PIFRO e CUDA *Multistart* PIFRO e, nas barras cinza, o *gap* entre CUDA ILS PIFRO e CUDA GA PIFRO. O gráfico mostra que as heurísticas CUDA GA-PIFRO e CUDA ILS-PIFRO são sempre superiores à heurística CUDA *Multistart* PIFRO em, no mínimo, 0,79%, situação vista na instância CTR nos valor de *gap* entre as heurísticas CUDA GA-PIFRO e CUDA *Multistart* PIFRO e de até 11,92%, quando observada a instância NW

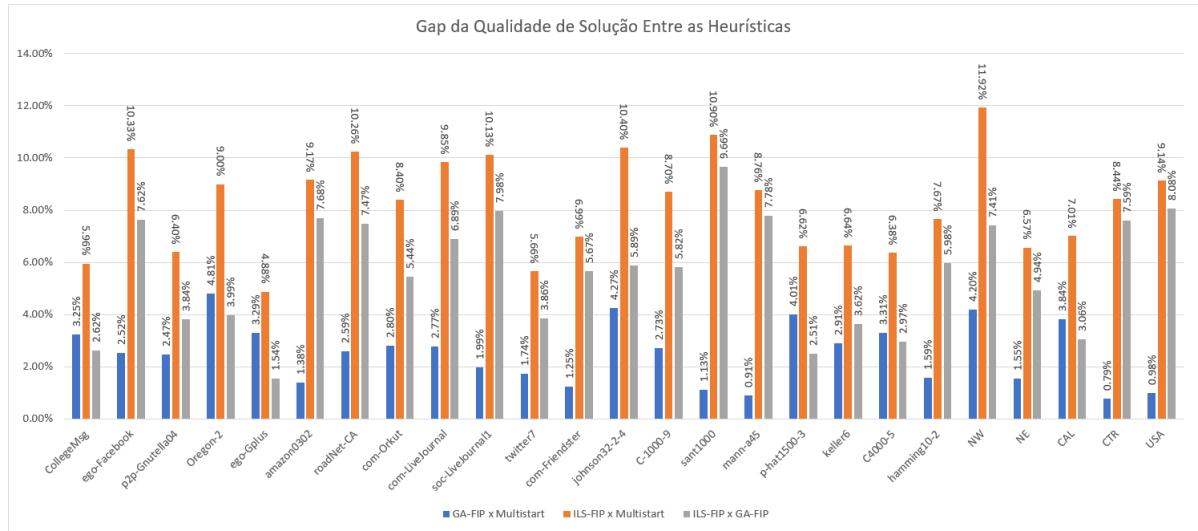


Fig. 8.10: *Gap* entre a qualidade das soluções apresentadas pelas heurísticas ao atingir o critério de parada.

no *gap* entre a heurística CUDA ILS-PIFRO e CUDA *Multistart* PIFRO. Nota-se ainda que a heurística CUDA ILS-PIFRO sempre apresenta desempenho superior à heurística CUDA GA-PIFRO, portanto, a heurística CUDA ILS-PIFRO tem desempenho superior em 100% das instâncias avaliadas. Ressalta-se ainda que o *gap* de desempenho entre as heurísticas CUDA ILS-PIFRO e CUDA GA-PIFRO está entre 1,54% e 9,66%.

Capítulo 9

Conclusões

Este capítulo apresenta as conclusões fundamentais derivadas desta tese de doutorado, destacando-se as principais contribuições e recomendações para pesquisas futuras.

9.1 Principais Contribuições

Esta tese apresentou contribuições significativas no contexto do Problema de Instalação de Fibras em Redes Óticas (PIFRO), ao introduzir novas estratégias de solução baseadas no paralelismo em GPUs Nvidia CUDA.

As principais contribuições deste trabalho são o desenvolvimento de estratégias de solução capazes de superar as abordagens existentes na literatura, tanto em termos de desempenho quanto de qualidade de solução, abrangendo também instâncias de maiores dimensões do problema, que não tinham sido analisadas até o momento. Essas estratégias de solução são um marco e servirão de referência para futuras pesquisas em problemas de otimização combinatória com características semelhantes ao PIFRO.

Para atingir os resultados mencionados, este trabalho seguiu as seguintes etapas:

- (i) Modelagem, projeto e implementação das estratégias de solução sequenciais existentes em versões paralelas para GPUs Nvidia CUDA, resultando nas heurísticas CUDA-PIFRO-*Greedy*, CUDA-PIFRO-PSC, CUDA-PIFRO-*Multistart*, CUDA-GA-PIFRO e CUDA-ILS-PIFRO;
- (ii) Identificação e superação dos gargalos na paralelização das estratégias de solução originalmente sequenciais do PIFRO;
- (iii) Modelagem, projeto e implementação das estratégias de solução nativamente paralelas para GPUs Nvidia CUDA, utilizando a metodologia de *Paths Pool*. Nessa abordagem, são calculados caminhos previamente para, posteriormente, serem combinados pelas heurísticas CUDA-PIFRO-*Greedy Paths Pools Based*, CUDA-PIFRO-PSC *Paths Pools Based*, CUDA-PIFRO-*Multistart Paths Pools Based*, CUDA-GA-PIFRO *Paths Pools Based* ou CUDA-ILS-PIFRO *Paths Pools Based*. Essa implementação eliminou a dependência excessiva do Algoritmo de Dijkstra, que é intrinsecamente sequencial;
- (iv) Desenvolvimento de versões paralelas dos algoritmos-chave do problema *K-Shortest Paths*, como o Algoritmo de Yen, o Algoritmo de Generalização de YEN e o Algoritmo

MPS. A versão baseada no Algoritmo MPS demonstrou os melhores resultados dentre todas as versões propostas;

- (v) Este trabalho também apresentou evidências de que as dimensões do espaço de busca pesquisado pelas heurísticas e futuras estratégias de solução, utilizando o *Paths Pool*, podem ser determinadas e garantir que a solução encontrada seja a melhor dentro do espaço determinado.

Parte significativa dessa tese está posta nos experimentos computacionais realizados para validar e avaliar as heurísticas propostas. O Experimento I mostra quão dependentes são as heurísticas até então existentes para PIFRO em relação ao Algoritmo de Dijkstra. Os Experimentos II e III mostram e comparam os desempenhos das versões paralelizadas CUDA propostas para os algoritmos de Yen, Generalização de Yen e MPS. Os Experimentos de IV a X exibem e comparam os desempenhos das heurísticas paralelizadas para CUDA propostos nessa tese, quais sejam, CUDA-*Multistart*-PIFRO, CUDA-GA-PIFRO e CUDA-ILS-PIFRO. Os Experimentos de XI a XVII exibem e comparam as qualidades das soluções retornadas por estas heurísticas paralelizadas. Um resumo conclusivo destes experimentos é exposta a seguir.

Experimento I

Através das Instâncias SNDLib e as heurísticas sequenciais para o PIFRO, faz-se a contagem de quantas execuções do algoritmo de Dijkstra (que é intrinsecamente sequencial) são realizadas por cada heurística dentro de 10 minutos de execução. Conclui-se que, nas execuções das heurísticas, em média, mais de 93% do tempo é utilizado para execuções de tal algoritmo.

Experimento II

Para criação do *Paths Pool* e, assim, poder projetar heurísticas nativamente paralelas para o PIFRO, reduzindo-se significativamente a utilização do algoritmo de Dijkstra, optou-se por fazê-lo via caminhos advindos para as instâncias através dos algoritmos para *K-Shortest-Paths*. O algoritmo de Yen é o mais utilizado na literatura, tornando sua paralelização imprescindível. O algoritmo de Yen executa, a cada iteração, por diversas vezes, o Algoritmo de Dijkstra. Assim, sendo, propusemos duas versões de paralelização CUDA do algoritmo de Yen. A primeira versão utiliza uma versão paralela do Algoritmo de Dijkstra, sendo uma paralelização mais intuitiva, associada à bibliografia atual. A segunda versão executa simultaneamente o Algoritmo de Dijkstra paralelo em diversas instâncias de iteração do *loop*. Essa versão é uma proposição original dessa tese, segundo o conhecimento do autor, e tem características paralelas não-intuitivas, sendo possível e viável graças à peculiaridades da tecnologia Nvidia CUDA. Essa segunda versão, com múltiplas instâncias de execução do Algoritmo de Dijkstra, apresenta melhor desempenho.

Experimento III

Bem como o ocorrido para o caso do Algoritmo de Yen para a elaboração do *Paths Pool*, essa tese propôs de forma original a paralelização dos Algoritmos Generalização de Yen e MPS. A seleção destes algoritmos para paralelização CUDA advém de observações de características de sua inicialização. Nessa etapa, eles geram árvores de caminhos através

de execuções do Algoritmo de Dijkstra para, posteriormente, eliminar a sua necessidade de execução em seus *loops* internos. A diferença principal entre o Algoritmo Generalização de Yen e o Algoritmo MPS dá-se na ordenação da árvore inicial de caminhos candidatos, ordenação essa denominada *Forward Sorted Star Form*, que reduz posteriormente a quantidade de testes para ordenação dos caminhos candidatos (tarefa intrinsecamente sequencial). Para a etapa de criação da *Forward Sorted Star Form*, é preciso tratar o problema *Segmented Sort*. Propusemos, neste trabalho que, para tal, fossem utilizados a *Dinamic Parallelism Based Strategy* ou a *Graph Edges Sort Through Parallel Fix Strategy* e, após experimentos, concluímos que a *Graph Edges Sort Through Parallel Fix Strategy* apresenta melhor desempenho. Na comparação de desempenho entre os algoritmos CUDA Yen Generalization e CUDA MPS, concluímos que CUDA MPS apresenta desempenho superior.

Experimentos IV, VII e XI

Fazem a comparação entre as heurísticas *Parallel Multistart PSC*, *Parallel Multistart Greedy* e *Parallel Multistart Paths Pool*, paralelizadas em baixo nível para o PIFRO. Deve-se ressaltar que as heurísticas *Parallel Multistart PSC* e *Parallel Multistart Greedy* são paralelizações em baixo nível (ou intuitivas) das heurísticas construtivas PSC e *Greedy* em procedimento de *Multistart* e, portanto, possuem alta dependência do Algoritmo de Dijkstra, enquanto que a heurística *Parallel Multistart Paths Pool* é um procedimento guloso de busca de caminhos criados em um *Paths Pool* (os experimentos consideram como seu tempo de execução a soma dos tempos necessários para criar o *Paths Pool* via Algoritmo CUDA-MPS e o tempo de execução do procedimento guloso de seleção dos caminhos que irão compor uma solução viável). A heurística CUDA-PIFRO-*Multistart* gera, aleatoriamente, a ordem de permutação dos caminhos $\langle i, j \rangle$ a serem inseridos na solução, advindos da instância selecionada. Conclui-se, então, que, seja qual for o grupo de instâncias, o melhor desempenho, bem como a melhor qualidade de solução, advém da heurística *Parallel Multistart Paths Pool*.

Experimentos V, VIII e XII

Fazem, respectivamente, a comparação de desempenho e qualidade de solução advindas das versões paralelizadas para CUDA dos algoritmos GA-PIFRO. As versões testadas advém das paralelizações da metaheurística BRKGA (*Based Random Key Genetic Algorithm*) existentes na literatura, porém, até então não implementadas e testadas em ambientes para GPUs. A heurística CUDA-GA-PIFRO cria cromossomos que repassam uma ordem de permutação dos caminhos óticos a serem traçados e formarem, assim, as redes solicitadas pelas instâncias, para heurísticas construtivas criarem soluções viáveis. Os experimentos compararam o desempenho e a qualidade de soluções advindas de CUDA-GA-PIFRO, utilizando, como heurísticas construtivas, as heurísticas PSC, *Greedy* e *Multistart Paths Pool*. Conclui-se que tanto o desempenho quanto a qualidade de solução de CUDA-GA-PIFRO, quando utiliza a construção de soluções viáveis via *Paths Pool*, é superior às duas outras versões propostas.

Experimentos VI, IX e XIII

Fazem, respectivamente, as comparações de desempenho e qualidade das soluções advindas das versões paralelizadas para CUDA dos algoritmos ILS-PIFRO. As versões testa-

das advém das paralelizações da metaheurística *Iterated Local Search* (ILS) existentes na literatura. A heurística ILS cria ordens de permutação que são modificadas a cada iteração, realizando a desconstrução de uma solução viável, aplicando uma perturbação em um percentual de seus caminhos e uma reconstrução através de caminhos diferentes a serem inseridos na busca por uma nova solução viável, que, acaso seja de melhor qualidade que a melhor solução corrente, passa a sofrer perturbações e reconstruções na busca de melhoria de qualidade. Para a perturbação, são retirados caminhos solicitados pela instância da solução e reinseridos novos através da execução do Algoritmo de Dijkstra paralelo pelas heurísticas construtivas *Greedy* ou PSC. Já a heurística CUDA-ILS-PIFRO *Paths Pool Based*, em seu processo de desconstrução, remove o caminho ótico selecionado para $\langle i, j \rangle$ e insere um caminho ótico diferente, a partir da escolha de um caminho previamente traçado no *Paths Pool*. Conclui-se que, tanto para desempenho quanto para a qualidade de solução, a heurística CUDA-ILS-PIFRO *Paths Pool Based* apresenta os melhores resultados

Experimentos XIV e XV

Comparam os experimentos anteriores e concluem que a heurística CUDA-ILS-PIFRO *Paths Pool Based* apresenta os melhores resultados de desempenho e qualidade de solução dentre todas as heurísticas testadas.

9.2 Trabalhos publicados

Esta tese já produziu os trabalhos:

- (i) RAIRO (2019) - *On the Problem of Minimizing the Cost with Optical Devices in WDM Optical Networks: complexity analysis, mathematical formulation and computational heuristics for solving it*;
- (ii) ENIAC (2020) - *Proposta de paralelização em GPUs CUDA do algoritmo MPS para resolução do problema de encontrar os K-Caminhos Mais Curtos Sem Repetições de Vértices em grafos direcionados e não direcionados*;
- (iii) *K-Shortest Loopless Paths on direct and undirected graphs through YEN CUDA based parallel versions*. Submetido a periódico acadêmico.

9.3 Recomendações para Pesquisas Futuras

Com base nos resultados obtidos, surgem recomendações para pesquisas futuras, que podem ampliar ainda mais o conhecimento nesta área:

- (i) Avaliar a metodologia de *Paths Pool* em outros problemas de otimização que envolvam a pesquisa por caminhos, seja em ambientes paralelos convencionais ou em ambientes paralelos baseados na tecnologia Nvidia CUDA;
- (ii) Projetar e implementar versões paralelas de outros algoritmos relevantes para a solução do problema K-Shortest-Paths, levando em consideração as particularidades do ambiente Nvidia CUDA.

Referências Bibliográficas

- Alba, Enrique. (2005). *Parallel Metaheuristics: A New Class of Algorithms*. John Wiley & Sons, NJ, USA.
- Alba, Enrique e Dorronsoro, Bernabé. (2004). Solving the vehicle routing problem by using cellular genetic algorithms. *European Conference on Evolutionary Computation in Combinatorial Optimization*, p. 11–20. Springer, (2004).
- Almeida, Bernardo F.; Correia, Isabel e Saldanha-daGama, Francisco. (2018). A biased random-key genetic algorithm for the project scheduling problem with flexible resources. *TOP*, v. 26, p. 283–308. doi: 10.1007/s11750-018-0472-9.
- Aloise, D. J.; Ribeiro, C. C.; Noronha, T. F.; Rocha, C. e Urrutia, S. (2008). A hybrid heuristic for a multi-objective real-life car sequencing problem with painting and assembly line constraints. *European Journal of Operational Research*, v. 191, p. 981–992.
- Alves, Derek; Oliveira, Davi; Andrade, Ermeson e Nogueira, Bruno. (2021). GPU-BRKGA: A GPU accelerated library for optimization using the biased random-key genetic algorithm. *IEEE Latin America Transactions*, v. 100.
- Amaro Junior, Bonfim; Pinheiro, Plácido Rogério e Coelho, Pedro Veras. (2017). A parallel biased random-key genetic algorithm with multiple populations applied to irregular strip packing problems. *Mathematical Problems in Engineering*, v. 2017. doi: 10.1155/2017/1670709.
- Amdahl, Gene M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), p. 483–485, New York, NY, USA. ACM.
- Andrade, Carlos E.; Miyazawa, Flávio K. e Resende, Mauricio G.C. (2013). Evolutionary algorithm for the k-interconnected multi-depot multi-traveling salesmen problem. *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO '13*, p. 463–470, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/2463372.2463434.
- Antonakopoulos, S. e Zhang, L. (2007). Heuristics for fiber installation in optical network optimization. *Proceedings of the 2007 IEEE Global Telecommunications Conference*, Washington.
- Arulselvan, A.; Commander, C. e Pardalos, P. (2007). A random keys based genetic algorithm for the target visitation problem. *Advances in Cooperative Control and Optimization*, volume 369, p. 389–397. Springer.

- Badeau, Philippe; Guertin, François; Gendreau, Michel; Potvin, Jean-Yves e Taillard, Eric. (1997). A parallel tabu search heuristic for the vehicle routing problem with time windows. *Transportation Research Part C: Emerging Technologies*, v. 5, n. 2, p. 109–122.
- Bader, David A.; Moret, Bernard M. E. e Sanders, Peter. (2002). Algorithm engineering for parallel computation. Fleischer, Rudolf; Moret, Bernard e Schmidt, Erik Meineche, editors, *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, p. 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Battiti, Roberto e Tecchiolli, Giampietro. (1992). Parallel biased search for combinatorial optimization: genetic algorithms and tabu. *Microprocessors and Microsystems*, v. 16, n. 7, p. 351–367.
- Bäumelt, Zdeněk; Dvořák, Jan; Šucha, Přemysl e Hanzálek, Zdeněk. (2016). A novel approach for nurse rostering based on a parallel algorithm. *European Journal of Operational Research*, v. 251, n. 2, p. 624–639.
- Baxter, Sean. Segmented sort and locality sort - modern gpu. <https://moderngpu.github.io/segssort.html>, (). (Accessed on 11/05/2021).
- Baxter, Sean. Intro - modern gpu. <https://moderngpu.github.io/intro.html>, (2013). (Acesso em 11/05/2021).
- Berclaz, Jerome; Fleuret, Francois; Turetken, Engin e Fua, Pascal. (2011). Multiple object tracking using k -shortest paths optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 33, n. 9, p. 1806–1819.
- Berger, Jean e Barkaoui, Mohamed. (2004). A parallel hybrid genetic algorithm for the vehicle routing problem with time windows. *Computers & Operations Research*, v. 31, n. 12, p. 2037–2053.
- Bergstrom, Lars e Reppy, John. (2012). Nested data-parallelism on the gpu. *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, p. 247–258, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/2364527.2364563.
- Bock, Frederick; Kantner, Harold e Haynes, John. (1957). An algorithm (the r -th best path algorithm) for finding and ranking paths through a network. Research report,, Armour Research Foundation of Illinois Institute of Technology, Chicago, Illinois, USA.
- Born, Max e Wolf, Emil. (1999). *Principles of Optics*. Cambridge University Press, 7^a edição.
- Brent, Richard P. (1971). An algorithm with guaranteed convergence for finding a zero of a function. *The Computer Journal*, v. 14, n. 4, p. 422–425.
- Bullnheimer, Bernd; Kotsis, Gabriele e Strauß, Christine. (1998). Parallelization strategies for the ant system. De Leone, Renato; Murli, Almerico; Pardalos, Panos M. e Toraldo, Gerardo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, volume 24, p. 87–100. Springer US, Dordrecht,.

- Buriol, L. S.; Resende, M. G. C.; Ribeiro, C. C. e Thorup, M. (2005). A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing. *Networks*, v. 46, p. 36–56.
- Buriol, L. S.; Resende, M. G. C. e Thorup, M. (2007). Survivable IP network design with OSPF routing. *Networks*, v. 49, p. 51–64.
- Calégari, Patrice; Guidec, Frédéric; Kuonen, Pierre e Kobler, Daniel. (1997). Parallel island-based genetic algorithm for radio network design. *Journal of Parallel and Distributed Computing*, v. 47, n. 1, p. 86–90.
- Cantú-Paz, Erick. (2005). Theory of parallel genetic algorithms. Alba, Enrique, editor, *Parallel Metaheuristics: A New Class of Algorithms*, p. 423–445. John Wiley & Sons, Inc.
- Cattaruzza, D.; Absi, N.; Feillet, D. e Vigo, D. (2014). An iterated local search for the multi-commodity multi-trip vehicle routing problem with time windows. *Computers & Operations Research*, v. 51, n. 0, p. 257–267.
- Cavalheiro, Ellen Marianne Bernal e Lyra Filho, Christiano. Set(2017). Otimização de Redes de Distribuição de Energia Elétrica por BRKGA. *XLIX Simpósio Brasileiro de Pesquisa Operacional (SBPO 2017)*, Blumenau, SC, Brasil. URL <http://www.din.uem.br/~ademir/sbpo/sbpo2017/pdf/168829.pdf>.
- Chakrapani, Jaishankar e Skorin-Kapov, Jadranka. (1992). A connectionist approach to the quadratic assignment problem. *Computers & Operations Research*, v. 19, n. 3-4, p. 287–295.
- Chakrapani, Jaishankar e Skorin-Kapov, Jadranka. (1993). Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, v. 41, n. 4, p. 327–341.
- Chartrand, Gary; Polimeni, Albert D. e Zhang, Ping. (2018). *Mathematical Proofs: A Transition to Advanced Mathematics*. Pearson, fourth edição. ISBN 978-0134746753. URL <https://www.pearson.com/us/higher-education/program/PGM1763590.html>.
- Che, Shuai; Boyer, Michael; Meng, Jiayuan; Tarjan, David; Sheaffer, Jeremy W e Skadron, Kevin. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, v. 68, n. 10, p. 1370–1380.
- Chen, Ren e Prasanna, Viktor K. (2016). Accelerating equi-join on a CPU-FPGA heterogeneous platform. *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 212–219. IEEE, (2016).
- Clarke, S.; Krikorian, A. e Rausen, J. (1963). Computing the N best loopless paths in a network. *Journal of the Society for Industrial and Applied Mathematics*, v. 11, n. 4, p. 1096–1102.

- Coco, Amadeu Almeida; deNoronha, Thiago Ferreira e Santos, Andréa Cynthia. Set(2013). Algoritmo baseado em BRKGA para o Problema de Árvore Geradora Mínima Robusta. *XLV Simpósio Brasileiro de Pesquisa Operacional (SBPO 2013)*, Natal, RN, Brasil. URL <https://hal.archives-ouvertes.fr/hal-02304057>.
- Cook, Shane. (2013). *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edição.
- Costa, F. N.; Urrutia, S. e Ribeiro, C. C. (2012). An ILS heuristic for the traveling tournament problem with predefined venues. *Annals of Operations Research*, v. 194, n. 1, p. 137–150.
- Crainic, T. G. (2005). Parallel computation, co-operation, tabu search. Alidaee, B. e In, Rego C., editors, *Metaheuristic Optimization Via Memory and Evolution: Tabu Search and Scatter Search*, p. 283–302. Kluwer Academic Publishers,, Norwell, MA,.
- Crainic, T. G. (2008). Parallel solution methods for vehicle routing problems. Golden, Bruce; Raghavan, S. e Wasil, Edward, editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, p. 171–198. Springer, New York.
- Crainic, T. G.; Davidović, T. e Ramljak, D. (2014). Designing parallel meta-heuristic methods. Belic, A. e Milutinovic, M. InV. Despotovic-Zrakic, editors, *High Performance and Cloud Computing in Scientific Research and Education*,, p. 260–280. IGI Global,.
- Crainic, T. G. e Hail, N. (2005). Parallel metaheuristic applications. *Parallel Metaheuristics: A New Class of Algorithms*, v. 47, p. 447.
- Crainic, Teodor Gabriel. (2015). Parallel meta-heuristic search. Publication CIRRELT 2015-42, Centre Interuniversitaire de Recherche sur les Réseaux d'Entreprise, la Logistique et le Transport, Université de Montréal, Montréal, QC, Canada.
- Crainic, Teodor Gabriel; Di Chiara, Beatrice; Nonato, Maddalena e Tarricone, Luciano. (2006). Tackling electromog in completely configured 3g networks by parallel cooperative meta-heuristics. *IEEE Wireless Communications*, v. 13, n. 6, p. 34–41.
- Crainic, Teodor Gabriel; Gendreau, Michel; Hansen, Pierre e Mladenović, Nenad. (2004). Cooperative parallel variable neighborhood search for the p-median. *Journal of Heuristics*, v. 10, n. 3, p. 293–314.
- Crainic, Teodor Gabriel e Toulouse, Michel. (2007). Explicit and emergent cooperation schemes for search algorithms. *International Conference on Learning and Intelligent Optimization*, p. 95–109. Springer, (2007).
- Crainic, Teodor Gabriel e Toulouse, Michel. (2010). Parallel meta-heuristics. *Handbook of metaheuristics*, p. 497–541. Springer.
- Crainic, Teodor Gabriel; Toulouse, Michel e Gendreau, Michel. (1996). Parallel asynchronous tabu search for multicommodity location-allocation with balancing requirements. *Annals of Operations Research*, v. 63, n. 2, p. 277–299.
- Crainic, Teodor Gabriel; Toulouse, Michel e Gendreau, Michel. (1997). Towards a taxonomy of parallel tabu search heuristics. *INFORMS Journal on Computing*, v. 9, n. 1, p. 61–72.

- Crauser, Andreas; Mehlhorn, Kurt; Meyer, Ulrich e Sanders, Peter. (1998). A parallelization of Dijkstra's shortest path algorithm. Brim, Luboš; Gruska, Jozef e Zlatuška, Jiří, editors, *International Symposium on Mathematical Foundations of Computer Science*, p. 722–731. Springer, (1998).
- Cung, Van-Dat; Martins, Simone L; Ribeiro, Celso C e Roucairol, Catherine. (2002). Strategies for the parallel implementation of metaheuristics. *Essays and surveys in metaheuristics*, p. 263–308. Springer.
- Davidović, Tatjana e Crainic, Theodor. (2015). Parallel local search to schedule communicating tasks on identical processors. *Parallel Computing*, v. 48, p. 1–14.
- De Falco, Ivan; Del Balio, Renato e Tarantino, Ernesto. (1994)a. Solving the mapping problem by parallel tabu search. *APPLIED INFORMATICS-PROCEEDINGS*, p. 264–264. IASTED, (1994)a.
- De Falco, Ivan; Del Balio, Renato; Tarantino, Ernesto e Vaccaro, Roberto. (1994)b. Improving search by incorporating evolution principles in parallel tabu search. *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, p. 823–828. IEEE, (1994)b.
- de Faria Jr., H.; Resende, Mauricio G. C. e Ernst, D. (2017). A biased random key genetic algorithm applied to the electric distribution network reconfiguration problem. *Journal of Heuristics*, v. 23, n. 6, p. 533–550.
- de Faria Jr., Haroldo; Lunardi, Willian Tessaro e Voos, Holger. (2019). A parallel multi-population biased random-key genetic algorithm for electric distribution network reconfiguration. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, p. 281–282, New York, NY, USA. Association for Computing Machinery. doi: 10.1145/3319619.3321950.
- Delévacq, Audrey; Delisle, Pierre e Krajecki, Michaël. (2012). Parallel gpu implementation of iterated local search for the travelling salesman problem. Hamadi, Youssef e Schoenauer, Marc, editors, *Learning and Intelligent Optimization: 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers*, p. 372–377. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-34413-8. doi: 10.1007/978-3-642-34413-8_30. URL http://dx.doi.org/10.1007/978-3-642-34413-8_30.
- Demetrescu, Camil; Goldberg, Andrew V. e Johnson, David S., editors. (2009). *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of DIMACS book. American Mathematical Society.
- Dias, L. G. S.; Goulart, N.; Noronha, T. F. e de Souza, S. R. (2011). Algoritmo genético para o problema de instalação de fibras em redes óticas. *Anais do XLIII Simpósio Brasileiro de Pesquisa Operacional*, (2011).
- Dijkstra, Edsger Wybe. (1959). A note on two problems in connection with graphs. *Numerische Mathematik*, v. 1, p. 269:271.

- DiMarco, Jeffrey e Taufer, Michela. (2013). Performance impact of dynamic parallelism on different clustering algorithms. Kelmelis, Eric J., editor, *Modeling and Simulation for Defense Systems and Applications VIII*, volume 8752, p. 97–104. International Society for Optics and Photonics, SPIE, (2013). doi: 10.1117/12.2018069.
- Djinevski, Leonid; Ristov, Sasko e Gusev, Marjan. (2013). Superlinear speedup for matrix multiplication in GPU devices. Markovski, Smile e Gusev, Marjan, editors, *ICT Innovations 2012*, p. 285–294, Berlin, Heidelberg. Springer.
- Doerner, Karl F.; Hartl, Richard F.; Kiechle, Guenter; Lucka, Maria e Reimann, Marc. (2004). Parallel ant systems for the capacitated vehicle routing problem. Gottlieb, Jens e Raidl, Günther R., editors, *Proceedings of the 4th. Evolutionary Computation in Combinatorial Optimization (EvoCOP 2004)*, p. 72–83, Coimbra, Portugal. Springer Berlin Heidelberg.
- Doerner, Karl F; Hartl, Richard F e Lucka, Maria. (2005). A parallel version of the d-ant algorithm for the vehicle routing problem. *Parallel Numerics*, v. 5, p. 109–118.
- Dong, X.; Chen, P.; Huang, H. e Nowak, M. (2013). A multi-restart iterated local search algorithm for the permutation flow shop problem minimizing total flow time. *Computers & Operations Research*, v. 40, n. 2, p. 627–632.
- Dorigo, Marco. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, (1992).
- Dorronsoró, Bernabé; Arias, Daniel; Luna, Francisco; Nebro, Antonio J e Alba, Enrique. (2007). A grid-based hybrid cellular genetic algorithm for very large scale instances of the cvrp. *High Performance Computing & Simulation Conference (HPCS)*, p. 759–765. IEEE Press Piscataway, NJ, (2007).
- dos Reis, Daniel Morais. Estudo sobre heurísticas para o problema de instalação de fibras em redes óticas. Dissertação de Mestrado, Centro Federal de Educação Tecnológica de Minas Gerais - CEFET MG, Agosto(2013).
- dos Reis, Daniel Morais; da Silva, Natã Goulart; Noronha, Thiago F. e de Souza, Sérgio Ricardo. (2019). On the problem of minimizing the cost with optical devices in wavelength division multiplexing optical networks: complexity analysis, mathematical formulation and improved heuristics. *RAIRO – Operations Research*, v. 53, n. 5, p. 1513–1528. doi: 10.1051/ro/2018072. URL <https://doi.org/10.1051/ro/2018072>.
- dos Reis, Daniel Morais; Espíndola, Álvaro Martins; de Almeida, Marcelo Henrique Ribeiro; Barrientos, Anolan Yamilé Milanés e de Souza, Sérgio Ricardo. A parallel CUDA implementation of yen’s algorithm for solving the K-Shortest Loopless Paths problem on direct and undirected graphs. Submetido, (2021).
- dos Reis, Daniel Morais; Noronha, Thiago F. e de Souza, Sérgio R. (2013). Iterated local search for fiber installation in optical network optimization. *Electronic Notes in Discrete Mathematics*, v. 41, p. 277 – 284.
- Du, Eds D-Z e Pardalos, PM. (1993). Tabu search: Applications and prospects. *Network Optimization Problems: Algorithms, Applications and Complexity*, v. 2, p. 333.

- Duarte, A.; Laguna, M.; Martí, R. e Sánchez-Oro, J. (2014). Optimization procedures for the bipartite unconstrained 0-1 quadratic programming problem. *Computers & Operations Research*, v. 51, n. 0, p. 123–129.
- Duarte, A. R.; Ribeiro, C. C. e Urrutia, S. (2007). A hybrid ILS heuristic to the referee assignment problem with an embedded MIP strategy. *Proceedings of the 4th International Workshop on Hybrid Metaheuristics*, p. 82–95. Springer, (2007).
- Dutta, A.; Dutta, N. e M., Fujiwara. (2004). *WDM Technologies: Optical Networks*. Elsevir.
- Eppstein, David. (1998). Finding the k shortest paths. *SIAM Journal on Computing*, v. 28, n. 2, p. 652–673.
- Expósito Piñol, Luis. (2012). Framework de desarrollo rápido de algoritmos paralelos. Master thesis, Facultat d’Informàtica de Barcelona, Universitat Politècnica de Catalunya.
- Feng, Gang. (2014). Improving space efficiency with path length prediction for finding k shortest simple paths. *IEEE Transactions on Computers*, v. 63, n. 10, p. 2459–2472.
- Fiechter, C-N. (1994). A parallel tabu search algorithm for large traveling salesman problems. *Discrete Applied Mathematics*, v. 51, n. 3, p. 243–267.
- Flynn, Michael J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, v. 21, n. 9, p. 948–960.
- Folino, Gianluigi; Pizzuti, Clara e Spezzano, Giandomenico. (1998). Solving the satisfiability problem by a parallel cellular genetic algorithm. *Proceedings of the 24th Euromicro Conference (1998)*, volume 2, p. 715–722, (1998).
- Fosin, Juraj; Davidović, Davor e Carić, Tonči. (2013). A gpu implementation of local search operators for symmetric travelling salesman problem. *PROMET-Traffic&Transportation*, v. 25, n. 3, p. 225–234.
- Galinier, P. e Hertz, A. (2006). A survey of local search methods for graph coloring. *Computers and Operations Research*, v. 33, p. 2547–2562.
- Garcia, Bruno-Laurent; Potvin, Jean-Yves e Rousseau, Jean-Marc. (1994). A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research*, v. 21, n. 9, p. 1025–1033.
- García-López, Félix; Melián-Batista, Belén; Moreno-Pérez, José A e Moreno-Vega, J Marcos. (2002). The parallel variable neighborhood search for the p-median problem. *Journal of Heuristics*, v. 8, n. 3, p. 375–388.
- Garcia-López, Félix; Melián-Batista, Belén; Moreno-Pérez, José A e Moreno-Vega, J Marcos. (2003). Parallelization of the scatter search for the p-median problem. *Parallel Computing*, v. 29, n. 5, p. 575–589.
- Garey, M. R. e Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

- Gendreau, Michel; Laporte, Gilbert e Semet, Frédéric. (2001). A dynamic model and parallel tabu search heuristic for real-time ambulance relocation. *Parallel computing*, v. 27, n. 12, p. 1641–1653.
- Gifre Renom, Lluís. (2012). Implementación de aplicación masivamente paralela usando CUDA. Master thesis, Facultat d'Informàtica de Barcelona, Universitat Politècnica de Catalunya.
- Glover, F e Punnen, A P. (1997). The travelling salesman problem: new solvable cases and linkages with the development of approximation algorithms. *Journal of the Operational Research Society*, v. 48, n. 5, p. 502–510. doi: 10.1057/palgrave.jors.2600392. URL <https://doi.org/10.1057/palgrave.jors.2600392>.
- Glover, Fred. (1977). Heuristics for integer programming using surrogate constraints. *Decision Sciences*, v. 8, n. 1, p. 156–166.
- Glover, Fred. (1997). Tabu search and adaptive memory programming—advances, applications and challenges. *Interfaces in computer science and operations research*, p. 1–75. Springer.
- Golub, Gene H. e Van Loan, Charles F. (1996). *Matrix Computations*. Johns Hopkins University Press, USA, 3rd edição. ISBN 0801854148.
- Gonçalves, José Fernando e Resende, Mauricio GC. (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, v. 17, n. 5, p. 487–525.
- Gonçalves, J. F.; Mendes, J. J. M. e Resende, M. G. C. (2005). A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research*, v. 167, p. 77–95.
- Gonçalves, J. F. e Resende, M. G. C. (2004). An evolutionary algorithm for manufacturing cell formation. *Computers and Industrial Engineering*, v. 47, p. 247–273.
- Gonçalves, J. F. e Resende, M. G. C. (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, v. 17, p. 487–525.
- Gonçalves, José Fernando e Resende, Mauricio G. C. (2012). A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers & Operations Research*, v. 39, n. 2, p. 179 – 190.
- Goulart, N.; Dias, L. G. S.; de Souza, S. R. e Noronha, T. F. (2011). Biased random-key genetic algorithm for fiber installation in optical network optimization. *Proceedings of the 2011 IEEE Congress on Evolutionary Computation*, p. 2267–2271, (2011).
- Guerriero, Francesca e Musmanno, Roberto. (2000). Parallel asynchronous algorithms for the k shortest paths problem. *Journal of Optimization Theory and Applications*, v. 104, n. 1, p. 91–108. doi: 10.1023/A:1004676705907.
- Gutin, G. e Yeo, A. (2002). Anti-matroids. *Oper. Res. Lett.*, v. 30, p. 97–99.

- Harish, Pawan e Narayanan, P. J. (2007). Accelerating large graph algorithms on the GPU using CUDA. Aluru, Srinivas; Parashar, Manish; Badrinath, Ramamurthy e Prasanna, Viktor K., editors, *International conference on High Performance Computing – HiPC 2007*, p. 197–208. Springer, (2007).
- Harris, Mark. Optimizing parallel reduction in CUDA, (2007). URL http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf. (Accessed on 27/05/2021).
- Harris, Mark. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops, (2013). URL <https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/>.
- Harris, Mark. Unified Memory for CUDA Beginners | NVIDIA Developer Blog. <https://devblogs.nvidia.com/unified-memory-cuda-beginners/>, 07(2017). (Accessed on 05/29/2021).
- Harris, Mark e Perelygin, Kyrylo. Cooperative groups: Flexible CUDA thread programming | Nvidia Developer Blog. <https://developer.nvidia.com/blog/cooperative-groups>, (2017). (Accessed on 27/05/2021).
- Herlihy, M. e Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann, Boston.
- Hershberger, John; Maxel, Matthew e Suri, Subhash. (2007). Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)*, v. 3, n. 4, p. 45.
- Hertz, A. e deWerra, D. (1987). Using tabu search techniques for graph coloring. *Computing*, v. 39, p. 345–351.
- Hildebrand, Begnaud Francis. (1987). *Introduction to Numerical Analysis*. Dover Publications, Inc., USA, 2nd edição. ISBN 0486653633.
- Hill, Mark D. e Marty, Michael R. (2008). Amdahl's law in the multicore era. *Computer*, v. 41, n. 7, p. 33–38.
- Hillis, W. Daniel e Steele, Guy L.Jr. December(1986). Data parallel algorithms. *Commun. ACM*, v. 29, n. 12, p. 1170–1183. ISSN 0001-0782. doi: 10.1145/7902.7903.
- Hoffman, Walter e Pavley, Richard. (1959). A method for the solution of the Nth best path problem. *Journal of the ACM (JACM)*, v. 6, n. 4, p. 506–514.
- Holland, John H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press.
- Hong, Inki; Kahng, Andrew B. e Moon, Byung-Ro. (1997). Improved large-step markov chain variants for the symmetric TSP. *Journal of Heuristics*, v. 3, n. 1, p. 63–81. ISSN 1572-9397. doi: 10.1023/A:1009624916728. URL <http://dx.doi.org/10.1023/A:1009624916728>.

- Hou, Kaixi; Liu, Weifeng; Wang, Hao e Feng, Wu-chun. (2017). Fast segmented sort on gpus. *Proceedings of the International Conference on Supercomputing*, p. 12. ACM, (2017).
- Hyttiä, E. e Virtamo, J. (1998). Wavelength assignment and routing in WDM networks. *Fourteenth Nordic Teletraffic Seminar*, p. 31–40, Copenhagen.
- Ichnowski, Jeffrey e Alterovitz, Ron. (2012). Parallel sampling-based motion planning with superlinear speedup. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*, p. 1206–1212, (2012). doi: 10.1109/IROS.2012.6386194.
- Jäskeläinen, P. O.; de La Lama, C. S.; Huerta, P. e Takala, J. H. (2010). Opencl-based design methodology for application-specific processors. *Proceedings of the 2010 International Conference on Embedded Computer Systems (SAMOS)*, p. 223–230, (2010).
- Jin, Jianyong; Crainic, Teodor Gabriel e Løkketangen, Arne. (2012). A parallel multi-neighborhood cooperative tabu search for capacitated vehicle routing problems. *European Journal of Operational Research*, v. 222, n. 3, p. 441–451.
- Jin, Jianyong; Crainic, Teodor Gabriel e Løkketangen, Arne. (2014). A cooperative parallel metaheuristic for the capacitated vehicle routing problem. *Computers & Operations Research*, v. 44, p. 33–41.
- Johnson, David S. e Mcgeoch, Lyle A. (1997). The traveling salesman problem: A case study in local optimization. Aarts, E.H.L. e Lenstra, J.K., editors, *Local Search in Combinatorial Optimization*, p. 215–310. Wiley.
- Jones, Stephen. Introduction to dynamic parallelism. Presentation at GPU Technology Conference, (2012). URL <https://on-demand.gputechconf.com/gtc/2012/presentations/S0338-GTC2012-CUDA-Programming-Model.pdf>. Accessed on 05/29/2021.
- Juan, Angel A.; Ramalhinho-Lourenço, Helena; Mateo, Manuel; Castellà, Quim e Barrios, Barry B. (2012). Ils-esp: An efficient, simple, and parameter-free algorithm for solving the permutation flow-shop problem. Economics working papers, Department of Economics and Business, Universitat Pompeu Fabra.
- Kare, Anjeneya Swami. (2016). A simple algorithm for replacement paths problem. *Electronic Notes in Discrete Mathematics*, v. 53, p. 307–318.
- Katoh, Naoki; Ibaraki, Toshihide e Mine, Hisashi. (1982). An efficient algorithm for k shortest simple paths. *Networks*, v. 12, n. 4, p. 411–427.
- Knap, Marcin e Czarnul, Paweł. (2019). Performance evaluation of unified memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. *The Journal of Supercomputing*, v. 75, p. 7625–7645. doi: 10.1007/s11227-019-02966-8.
- Kumar, N. e Ghosh, R. K. (1994). Parallel algorithm for finding first k shortest paths. *Computer Science and Informatics: Journal of the Computer Society of India*, v. 24, n. 3, p. 21–28.

- Laganière, Robert e Mitiche, A. (1995). Parallel tabu search for robust image filtering. *Proceedings of IEEE Workshop on Nonlinear Signal and Image Processing (NSIP'95)*, volume 2, p. 603–605, (1995).
- Lai, X. e Lü, Z. (2013). Multistart Iterated Tabu Search for bandwidth coloring problem. *Computers & Operations Research*, v. 40, n. 5, p. 1401 – 1409.
- Lawrence, V. S. e Mark, S. D. (2006). A random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational Research*, v. 174, p. 38–53.
- Le Bouthillier, Alexandre e Crainic, Teodor Gabriel. (2005). A cooperative parallel meta-heuristic for the vehicle routing problem with time windows. *Computers & Operations Research*, v. 32, n. 7, p. 1685–1708.
- Lee, K. G. e Lee, S. Y. (1992)a. Efficient parallelization of simulated annealing using multiple Markov chains: An application to graph partitioning. T.N., Mudge, editor, *Proceedings of the International Conference on Parallel Processing, volume III: Algorithms and Applications*, p. 177–180. CRC Press.
- Lee, K. G. e Lee, S. Y. (1995). Synchronous and asynchronous parallel simulated annealing with multiple Markov chains. Brandenburg, F., editor, *Graph Drawing - Proceedings GD'95, Symposium on Graph Drawing*, p. 396–408. Springer-Verlag, Passau, Germany.
- Lee, S. Y. e Lee, K. G. (1992)b. Asynchronous communication of multiple Markov chains in parallel simulated annealing. T.N., Mudge, editor, *Proceedings of the International Conference on Parallel Processing, volume III: Algorithms and Applications*, p. 169–176. CRC Press, Boca Raton, FL.
- Lee, Soo-Young e Lee, Kyung Geun. (1996). Synchronous and asynchronous parallel simulated annealing with multiple markov chains. *IEEE Transactions on Parallel and Distributed Systems*, v. 7, n. 10, p. 993–1008.
- Lei, De Ming e Xu, Xiao Qiang. 7(2014). A parallel iterated local search for two-agent flow shop scheduling. *Progress in Applied Sciences, Engineering and Technology*, volume 926 of *Advanced Materials Research*, p. 3476–3484. Trans Tech Publications, 7(2014). doi: 10.4028/www.scientific.net/AMR.926-930.3476.
- Leskovec, Jure e Krevl, Andrej. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, May(2021).
- Li, Da; Wu, Hancheng e Becchi, Michela. (2015). Nested parallelism on GPU: Exploring parallelization templates for irregular loops and recursive computations. *2015 44th International Conference on Parallel Processing*, p. 979–988. IEEE, (2015). doi: 10.1109/ICPP.2015.107.
- Lin, S-C; Punch, W. F. e Goodman, E. D. (1994). Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Sixth IEEE Symposium on Parallel and Distributed Processing*, p. 28–37. IEEE Computer Society Press.
- Lopes, R.; Morais, V. W. C.; Noronha, T. F. e Souza, V. A. A. (2015). Heuristics and matheuristics for a real-life machine reassignment problem. *International Transactions in Operational Research*, v. 22, n. 1, p. 77–95.

- López, Félix García; Torres, Miguel García; Batista, Belén Melián; Pérez, José A Moreno e Moreno-Vega, J Marcos. (2006). Solving feature subset selection problem by a parallel scatter search. *European Journal of Operational Research*, v. 169, n. 2, p. 477–489.
- López, Félix García; Torres, Miguel García; Batista, Belén Melián; Perez, José A Moreno e Vega, J. (2005). Parallel scatter search. Alba, Enrique, editor, *Parallel Metaheuristics: A New Class of Algorithms*, p. 423–445. Wiley Online Library.
- Lourenço, H. R.; Martin, O. C. e Stützle, T. (2010). Iterated local search: Framework and applications. Glover, F.; Kochenberger, G. e Hillier, F. S., editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research and Management Science*, Capítulo 12, p. 363–398. Springer, 2nd edition edição.
- Lourenço, Helena R.; Martin, Olivier C. e Stützle, Thomas. (2002). Iterated local search. *Handbook of Metaheuristics, volume 57 of International Series in Operations Research and Management Science*, p. 321–353. Kluwer Academic Publishers, (2002).
- Lucena, M. L.; Andrade, C. E.; Resende, M. G. C. e Miyazawa, F. K. (2014). Some extensions of biased random-key genetic algorithms. *XLVI Simpósio Brasileiro de Pesquisa Operacional*, (2014).
- Luebke, David. (2008). CUDA: Scalable parallel programming for high-performance scientific computing. *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, p. 836–838. IEEE, (2008). doi: 10.1109/ISBI.2008.4541126.
- Luong, Thé Van; Loukil, Lakhdar; Melab, Nouredine e Talbi, El-Ghazali. (2010)a. A GPU-based Iterated Tabu Search for Solving the Quadratic 3-dimensional Assignment Problem. *Proceedings of the Workshop on Parallel Optimization in Emerging Computing Environments (POECE) in Conjunction with the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2010)*, p. 1–8, Hammamet, Tunisia. IEEE Computer Society.
- Luong, Thé Van; Melab, Nouredine e Talbi, El-Ghazali. (2010)b. Neighborhood Structures for GPU-based Local Search Algorithms. *Parallel Processing Letters*, v. 20, n. 4, p. 307–324.
- Malek, M.; Guruswamy, M.; Pandya, M. e Owens, H. (1989). Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research*, v. 21, p. 59–84.
- Malve, S. e Uzsoy, R. (2007). A genetic algorithm for minimizing maximum lateness on parallel identical batch processing machines with dynamic job arrivals and incompatible job families. *Computers and Operations Research*, v. 34, p. 3016–3028.
- Martin, Olivier e Otto, Steve. (1996). Combining simulated annealing with local search heuristics. *Annals of Operations Research*, v. 63, n. 1, p. 57–75.
- Martín, Pedro J.; Torres, Roberto e Gavilanes, Antonio. (2009). CUDA solutions for the SSSP problem. Allen, Gabrielle; Nabrzyski, Jarosław; Seidel, Edward; van Albada, Geert Dick; Dongarra, Jack e Sloot, Peter M. A., editors, *International Conference on Computational Science – ICCS 2009*, p. 904–913. Springer, (2009). doi: 10.1007/978-3-642-01970-8_91.

- Martins, A. X.; Duhamel, C.; Mahey, P.; Saldanha, R. R. e deSouza, M. C. (2012). Variable neighborhood descent with iterated local search for routing and wavelength assignment. *Computers & Operations Research*, v. 39, n. 9, p. 2133 – 2141.
- Martins, Ernesto de Queirós Vieira; Pascoal, Marta Margarida Braz e dos Santos, José Luís Esteves. May(1997). A new algorithm for ranking loopless paths. Research report,, Universidade de Coimbra. URL www.mat.uc.pt/~marta/Publicacoes/mps.ps.gz.
- Martins, Ernesto de Queirós Vieira; Pascoal, Marta Margarida Braz e dos Santos, José Luis Esteves. (1999). Deviation algorithms for ranking shortest paths. *International Journal of Foundations of Computer Science*, v. 10, n. 03, p. 247–261.
- Michallet, J.; Prins, C.; Amodeo, L.; Yalaoui, F. e Vitry, G. (2014). Multi-start iterated local search for the periodic vehicle routing problem with time windows and time spread constraints on services. *Computers & Operations Research*, v. 41, n. 0, p. 196 – 207.
- Middendorf, Martin; Reischle, Frank e Schneck, Hartmut. (2002). Multi colony ant algorithms. *Journal of Heuristics*, v. 8, n. 3, p. 305–320.
- Milanés, Anolan; Urrutia, Sebastián e Løkketangen, Arne. June(2013). A GPU algorithm for the dtspms. *Eighth Triennial Symposium on Transportation Analysis (TRISTAN VIII)*, San Pedro de Atacama, Chile.
- Morais, V. W. C.; Mateus, G. R. e Noronha, T. F. (2014). Iterated local search heuristics for the vehicle routing problem with cross-docking. *Expert Systems with Applications*, v. 41, n. 16, p. 7495 – 7506. doi: 10.1016/j.eswa.2014.06.010.
- Mühlenbein, Heinz. (1989). Parallel genetic algorithms, population genetics and combinatorial optimization. *Workshop on Parallel Processing: Logic, Organization, and Technology*, p. 398–406. Springer, (1989).
- Mühlenbein, Heinz. (1992). Parallel genetic algorithms in combinatorial optimization. *Computer science and operations research*. Citeseer, (1992).
- Munguía, Lluís-Miquel; Ahmed, Shabbir; Bader, David A.; Nemhauser, George L.; Goel, Vikas e Shao, Yufen. (2017). A parallel local search framework for the fixed-charge multicommodity network flow problem. *Computers & Operations Research*, v. 77, p. 44 – 57. doi: <http://dx.doi.org/10.1016/j.cor.2016.07.016>. URL <http://www.sciencedirect.com/science/article/pii/S0305054816301873>.
- Noronha, T. F. Roteamento e atribuição de comprimentos de onda: Uma abordagem por coloração de partições. Dissertação de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro, (2004).
- Noronha, T. F.; Resende, M. G. C. e Ribeiro, C. C. (2011). A biased random-key genetic algorithm for routing and wavelength assignment. *Journal of Global Optimization*, v. 50, p. 503–518.
- Noronha, T. F. e Ribeiro, C. C. (2006). Routing and wavelength assignment by partition coloring. *European Journal of Operational Research*, v. 171, p. 797–810.

- Nuutila, Esko. 07(1995). Efficient transitive closure computation in large digraphs. *Mathematics and Computing in Engineering Series*, v. 74.
- Nvidia,. CUDA Toolkit Documentation v10.1.105, (2018). URL <https://docs.nvidia.com/cuda/>.
- Orlin, James B e Sharma, Dushyant. (2004). Extended neighborhood: Definition and characterization. *Mathematical programming*, v. 101, n. 3, p. 537–559.
- Orlowski, S.; Pióro, M.; Tomaszewski, A. e Wessäly, R. (2007)a. SNDlib 1.0–Survivable Network Design Library. Relatório técnico, Konrad-Zuse-Zentrum für Informationstechnik Berlin.
- Orlowski, S.; Wessäly, R.; Pióro, M. e Tomaszewski, A. Sndlib 1.0 – survivable network design library. <http://sndlib.zib.de/home.action>, last visited on march 22, 2021, (2007)b.
- Orlowski, S.; Wessäly, R.; Pióro, M. e Tomaszewski, A. (2010). Sndlib 1.0 – survivable network design library. *Networks*, v. 55, n. 3, p. 276–286. doi: <https://doi.org/10.1002/net.20371>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/net.20371>.
- Ortega-Arranz, Hector; Torres, Yuri; Llanos, Diego R. e Gonzalez-Escribano, Arturo. (2013). A new GPU-based approach to the shortest path problem. *2013 International Conference on High Performance Computing & Simulation (HPCS)*, p. 505–511. IEEE, (2013).
- Otte, Michael e Correll, Nikolaus. June(2013). C-Forest: Parallel shortest-path planning with super linear speedup. *IEEE Transactions on Robotics*, v. 29, p. 798–806.
- Pacheco, Peter. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers, Inc.
- PIFRO, Google Project Script. <http://code.google.com/p/pifro/downloads/list>, Visitado last visited on march 22, 2021, (2021).
- Polacek, M.; Benkner, S.; Doerner, K. F. e Hartl, R. F. (2008). A cooperative and adaptive variable neighborhood search for the multi depot vehicle routing problem with time windows. *Business Research*, v. 1, p. 207–218.
- Pollack, Maurice. (1961). Letter to the editor - the k -th best route through a network. *Operations Research*, v. 9, n. 4, p. 578–580. doi: 10.1287/opre.9.4.578. URL <https://doi.org/10.1287/opre.9.4.578>.
- Porto, S. C. S. e Ribeiro, C. C. (1996). Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints. *Journal of Heuristics*, v. 1, n. 2, p. 1–2.
- Porto, Stella C. S. e Ribeiro, Celso C. (1995). A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. *International Journal of High-Speed Computing*, v. 07, n. 01, p. 45–71. doi: 10.1142/S012905339500004X.

- Porto, Stella C.S.; Kitajima, João Paulo F.W. e Ribeiro, Celso C. (2000). Performance evaluation of a parallel tabu search task scheduling algorithm. *Parallel Computing*, v. 26, n. 1, p. 73–90.
- Prasetyo, H.; Qoyyiimah, M. e Fauza, G. oct(2018). Solving CCVRPTW using biased random key genetic algorithm (BRKGA) with multiple parent. *IOP Conference Series: Materials Science and Engineering - 2017 1st International Conference on Engineering and Applied Technology (ICEAT) 29–30 November 2017, Mataram, Indonesia*, volume 403, p. 012045. IOP Publishing, oct(2018). doi: 10.1088/1757-899x/403/1/012045.
- Quinn, Michael J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group. ISBN 0071232656.
- Rahoual, M.; Hadji, R. e Bachelet, V. (2002). Parallel ant system for the set covering problem. Sampels, M. e Caro, M. In Di G. Dorigo, editors, *Ant Algorithms - Proceedings of the Third International Workshop, ANTS 2002*, p. 262–267. Springer-Verlag, Berlin,.
- Randall, Marcus e Lewis, Andrew. (2002). A parallel implementation of ant colony optimization. *Journal of Parallel and Distributed Computing*, v. 62, n. 9, p. 1421–1432. doi: <https://doi.org/10.1006/jpdc.2002.1854>.
- Rego, César. (2001). Node-ejection chains for the vehicle routing problem: Sequential and parallel algorithms. *Parallel Computing*, v. 27, n. 3, p. 201–222. doi: [https://doi.org/10.1016/S0167-8191\(00\)00102-2](https://doi.org/10.1016/S0167-8191(00)00102-2).
- Rego, César e Roucairol, Catherine. (1996). A parallel tabu search algorithm using ejection chains for the vehicle routing problem. *Meta-Heuristics*, p. 661–675. Springer.
- Reis, R.; Ritt, M.; Buriol, L. S. e Resende, M. G. C. (2011). A biased random-key genetic algorithm for ospf and deft routing to minimize network congestion. *International Transactions in Operational Research*, v. 18, p. 401–423.
- Research, NVIDIA. Cub: Main page. <https://nvlabs.github.io/cub/>, (). (Accessed on 11/05/2021).
- Ribeiro, C. C.; Aloise, D. J.; Noronha, T. F.; Rocha, C. e Urrutia, S. (2008). An efficient implementation of a VNS/ILS heuristic for a real-life car sequencing problem. *European Journal of Operational Research*, v. 191, n. 3, p. 596–611.
- Rochat, Yves e Taillard, Éric D. (1995). Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, v. 1, n. 1, p. 147–167.
- Rocki, Kamil e Suda, Reiji. (2012). Large scale parallel iterated local search algorithm for solving traveling salesman problem. *Proceedings of the 2012 Symposium on High Performance Computing, HPC '12*, p. 8:1–8:8, San Diego, CA, USA. Society for Computer Simulation International. ISBN 978-1-61839-788-1. URL <http://dl.acm.org/citation.cfm?id=2338816.2338824>.
- Rosen, Kenneth H. (2010). *Handbook of Discrete and Combinatorial Mathematics, Second Edition*. Chapman and Hall/CRC, 2nd edição. ISBN 158488780X.

- Rossi, Ryan e Ahmed, Nesreen. (2015). The network data repository with interactive graph analytics and visualization. *Twenty-Ninth AAAI Conference on Artificial Intelligence*, (2015). URL <https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9553/9856>.
- Ruiz, Efrain; Albareda-Sambola, Maria; Fernández, Elena e Resende, Mauricio GC. (2015). A biased random-key genetic algorithm for the capacitated minimum spanning tree problem. *Computers & Operations Research*, v. 57, p. 95–108.
- Ruppert, Eric. (2000). Finding the k shortest paths in parallel. *Algorithmica*, v. 28, n. 2, p. 242–254.
- Sakarovitch, M. (1968). The k shortest chains in a graph. *Transportation Research*, v. 2, n. 1, p. 1 – 11. doi: [https://doi.org/10.1016/0041-1647\(68\)90003-8](https://doi.org/10.1016/0041-1647(68)90003-8).
- Sakharnykh, Nikolay. Beyond GPU memory limits with unified memory on Pascal. <https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/>, 12(2016). (Accessed on 05/29/2021).
- Sakharnykh, Nikolay. Maximizing unified memory performance in CUDA | NVIDIA Developer Blog. <https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/>, 11(2017). (Accessed on 05/29/2021).
- Samanlioglu, F.; Ferrell, W. G.Jr. e Kurz, M. E. (2008). A memetic random-key genetic algorithm for a symmetric multi-objective traveling salesman problem. *Computers and Industrial Engineering*, v. 55, p. 439–449.
- Sanvicente-Sánchez, H. e Frausto-Solís, J. (2002). Mpsa: A methodology to parallelize simulated annealing and its application to the traveling salesman problem. Coello Coello, Carlos A.; de Alborno, Alvaro; Sucar, Luis Enrique e Battistutti, Osvaldo Cairó, editors, *MICAI 2002: Advances in Artificial Intelligence*, p. 89–97. Springer-Verlag, Heidelberg.
- Schmid, Rafael; Pisani, Flávia; Borin, Edson e Cáceres, Edson. (2016). An evaluation of segmented sorting strategies on gpus. *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, p. 1123–1130. IEEE, (2016).
- Schrijver, Alexander. (1986). *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA.
- Sevkli, Mehmet e Aydin, M Emin. (2007). Parallel variable neighbourhood search algorithms for job shop scheduling problems. *IMA Journal of Management Mathematics*, v. 18, n. 2, p. 117–133.
- Silva, M. M.; Subramanian, A. e Ochi, L. S. (2015). An iterated local search heuristic for the split delivery vehicle routing problem. *Computers & Operations Research*, v. 53, n. 0, p. 234–249.

- Silva, P. H. G.; Brandão, D. N.; Morais, I. S. e deOliveira, S. L. Gonzaga. (2020). A biased random-key genetic algorithm for bandwidth reduction. Gervasi, Osvaldo; Murgante, Beniamino; Misra, Sanjay; Garau, Chiara; Blečić, Ivan; Tanian, David; Aduhan, Bernady O.; Rocha, Ana Maria A.C.; Tarantino, Eufemia; Torre, Carmelo Maria e Karaca, Yeliz, editors, *Computational Science and Its Applications – ICCSA 2020*, p. 312–321. Springer International Publishing, (2020).
- Singh, Avadhesh Pratap e Singh, Dharendra Pratap. (2015). Implementation of k -shortest path algorithm in GPU using CUDA. *Procedia Computer Science*, v. 48, p. 5–13.
- Singh, Dharendra Pratap e Khare, Nilay. (2013). Parallel implementation of the single source shortest path algorithm on cpu-gpu based hybrid system. *International Journal of Computer Science and Information Security*, v. 11, n. 9, p. 74.
- Skorin-Kapov, N. *Heuristic algorithms for virtual topology design and routing and wavelength assignment in WDM networks*. PhD thesis, University of Zagreb, Zagreb, (2006).
- Sörensen, Kenneth. (2015). Metaheuristics: the metaphor exposed. *International Transactions in Operational Research*, v. 22, n. 1, p. 3–18.
- Spears, W. e deJong, K. (1991). On the virtues of parameterized uniform crossover. *Proceedings of the Fourth International Conference on Genetic Algorithms*, p. 230–236, San Mateo.
- Srivastava, Ajitesh; Chen, Ren; Prasanna, Viktor K e Chelms, Charalampos. (2015). A hybrid design for high performance large-scale sorting on FPGA. *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, p. 1–6. IEEE, (2015).
- Steinberger, Markus; Kenzel, Michael; Kainz, Bernhard e Schmalstieg, Dieter. (2012). ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. *2012 Innovative Parallel Computing (InPar)*, p. 1–10. IEEE, (2012). doi: 10.1109/InPar.2012.6339604.
- Stützle, Thomas. (1998). Parallelization strategies for ant colony optimization. *International Conference on Parallel Problem Solving from Nature*, p. 722–731. Springer, (1998).
- Sun, Xian-He e Chen, Yong. (2010). Reevaluating amdahl’s law in the multicore era. *J. Parallel Distrib. Comput.*, v. 70, n. 2, p. 183–188.
- Taillard, E. (1991). Robust taboo search for the quadratic assignment problem. *Parallel Computing*, v. 17, n. 4, p. 443–455. doi: [https://doi.org/10.1016/S0167-8191\(05\)80147-4](https://doi.org/10.1016/S0167-8191(05)80147-4).
- Taillard, Éric. (1993). Parallel iterative search methods for vehicle routing problems. *Networks*, v. 23, n. 8, p. 661–673.
- Taillard, Eric D. (1994). Parallel taboo search techniques for the job shop scheduling problem. *ORSA journal on Computing*, v. 6, n. 2, p. 108–117.
- Talbi, El-Ghazali. (2006). *Parallel combinatorial optimization*, volume 58. John Wiley & Sons.

- Talbi, El-Ghazali. (2009). *Metaheuristics: From Design to Implementation*. Wiley Publishing. ISBN 0470278587, 9780470278581.
- Talbi, El-Ghazali; Mostaghim, Sanaz; Okabe, Tatsuya; Ishibuchi, Hisao; Rudolph, Günter e Coello Coello, Carlos A. (2008). Parallel approaches for multiobjective optimization. Branke, Jürgen; Deb, Kalyanmoy; Miettinen, Kaisa e Słowiński, Roman, editors, *Multi-objective Optimization: Interactive and Evolutionary Approaches*, p. 349–372. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Talbi, El-ghazali; Roux, Olivier; Fonlupt, Cyril e Robillard, Denis. (1999). Parallel ant colonies for combinatorial optimization problems. Rolim, José e etalli., editors, *Proceedings of the Parallel and Distributed Processing: 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, p. 239–247. Springer Berlin Heidelberg, San Juan, Puerto Rico, USA.
- Talukdar, S.; Baerentzen, L.; Gove, A. e Souza, de P. (1998). Asynchronous teams: Cooperation schemes for autonomous agents. *Journal of Heuristics*, v. 4, p. 295–321.
- Talukdar, Sarosh; Murthy, Sesh e Akkiraju, Rama. (2003). Asynchronous teams. *Handbook of Metaheuristics*, p. 537–556. Springer.
- ten Eikelder, Huub M. M.; Aarts, Bas J. M.; Verhoeven, Marco G. A. e Aarts, Emile H. L. (1999). Sequential and parallel local search algorithms for job shop scheduling. Voß, Stefan; Martello, Silvano; Osman, Ibrahim H. e Roucairol, Catherine, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, p. 359–371. Springer US, Boston, MA.
- Tongheng, Guo e Chundi, Mu. (2002). Radio network design using coarse-grained parallel genetic algorithms with different neighbor topology. *Proceedings of the 4th World Congress on Intelligent Control and Automation*, volume 3, p. 1840–1843. IEEE, (2002).
- Toulouse, M.; Thulasiraman, K. e Glover, F. (1999). Multi-level cooperative search: A new paradigm for combinatorial optimization and an application to graph partitioning. Ruiz, D.; Giraud, L.; Frayssé, V.; Duff, I.; Daydé, M.; Berger, P. e Amestoy, P., editors, *5th International Euro-Par Parallel Processing Conference*, p. 533–542. Springer-Verlag.
- Vazirani, Vijay V. (2001). *Approximation Algorithms*. Springer-Verlag, Berlin, Heidelberg. ISBN 3540653678.
- Verhoeven, M. G. A. e Aarts, E. H. L. (1995). Parallel local search. *Journal of Heuristics*, v. 1, n. 1, p. 43–65.
- Verhulst, Eric. (2003). Beyond the von neumann machine. Jantsch, Axel e Tenhunen, Hannu, editors, *Networks on Chip*, p. 217–238. Springer US, Boston, MA. doi: 10.1007/0-306-48727-6_11.
- Wang, Jin e Yalamanchili, Sudhakar. (2014). Characterization and analysis of dynamic parallelism in unstructured GPU applications. *2014 IEEE International Symposium on Workload Characterization (IISWC)*, p. 51–60. IEEE, (2014). doi: 10.1109/IISWC.2014.6983039.

- Wen, Qingsong; Zhou, Qi; Zhao, Chunming e Ma, Xiaoli. (2013). Fixed-point realization of lattice-reduction aided mimo receivers with complex k-best algorithm. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, p. 5031–5035. IEEE, (2013). doi: 10.1109/ICASSP.2013.6638619.
- Widmer, Sven; Wodniok, Dominik; Weber, Nicolas e Goesele, Michael. (2013). Fast dynamic memory allocator for massively parallel architectures. *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, p. 120–126, (2013). doi: 10.1145/2458523.2458535.
- Wolsey, L.A. (1998). *Integer programming*. Wiley.
- Yen, Jin Y. (1971). Finding the k shortest loopless paths in a network. *Management Science*, v. 17, n. 11, p. 712–716.
- yWorks,. yEd Graph Editor, (2016). URL <https://www.yworks.com/products/yed>.
- Zhou, Yi; He, Fazhi e Qiu, Yimin. (2016). Optimization of parallel iterated local search algorithms on graphics processing unit. *The Journal of Supercomputing*, v. 72, n. 6, p. 2394–2416. ISSN 1573-0484. doi: 10.1007/s11227-016-1738-3. URL <http://dx.doi.org/10.1007/s11227-016-1738-3>.