Willian da Silva Zocolau

Paralelização da busca local da metaheurística A-BRKGA com OpenACC

São José dos Campos, SP

Willian da Silva Zocolau

Paralelização da busca local da metaheurística A-BRKGA com OpenACC

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência de Tecnologia

Bacharelado em Ciência da Computação

Orientador: Prof. Dr. Álvaro Luiz Fazenda

Coorientador: Prof. Dr. Antônio Augusto Chaves

São José dos Campos, SP

Outubro de 2020

Willian da Silva Zocolau

Paralelização da busca local da metaheurística A-BRKGA com OpenACC

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em:

Prof. Dr. Álvaro Luiz Fazenda
Orientador

Prof. Dr. Antônio Augusto Chaves
Coorientador

Prof[a]. Dr[a]. Denise Stringhini
Professora Convidada

Prof. Dr. Rudinei Martins de Oliveira
Prof. Convidado externo
Universidade do Estado de Minas Gerais
(UEMG)

São José dos Campos, SP
Outubro de 2020

Este trabalho é dedicado a Educação que tem aberto muitos caminhos na minha vida. A minha mãe, minha companheira e familiares que, apesar das dificuldades, me acompanharam e estiveram presentes durante todo este ciclo. Aos meus amigos e professores pelas diversas experiências de vida e valiosos ensinamentos.

Agradecimentos

De início, dedico este trabalho aos meus familiares. De modo especial, a meus pais Marcos (in memorian) e Luzia.

A minha companheira Daiane que, apesar da distância, esteve sempre presente e compartilhou grandes valores da vida.

A meu orientador Prof. Dr. Álvaro Luiz Fazenda, pela prontidão, serenidade e sabedoria partilhados durante o desenvolvimento deste trabalho.

A meu coorientador Prof. Dr. Antônio Augusto Chaves, por compartilhar seu conhecimento, experiência e grande contribuição com o trabalho A-BRKGA.

E, por fim, a todos os amigos e colegas que, mesmo indiretamente, participaram da construção dos meus valores e discernimentos.

“To teach is to learn twice.“

(Joseph Joubert)

Resumo

A dificuldade para encontrar soluções ótimas em tempo aceitável tem sido desafiador para a computação. Como estratégia mais eficiente para lidar com esse problema, soluções com foco em algoritmos genéticos têm sido propostas. Baseado nessa técnica de programação, o *Biased Random-Key Genetic Algorithm* com configuração de parâmetros on-line (A-BRKGA) explora o espaço de busca para encontrar soluções interessantes. Dado a dificuldade para determinar tais soluções para instâncias grandes de problema em tempo factível, o objetivo deste trabalho é identificar os principais pontos de paralelização para, em seguida, aplicar técnicas de programação concorrente e paralela. De modo específico, teve-se a finalidade de melhorar o desempenho da busca local com ajuda do padrão OpenACC, no qual pode-se obter Speedup de 12,13. Ao final, apresenta-se os resultados obtidos com as respectivas discussões, além de demonstrar o ganho de tempo no processo de convergência para a solução e verificar estudos comparativos relacionados ao tempo de execução em geral.

Palavras-chaves: programação paralela, OpenACC, programação concorrente e distribuída, A-BRKGA.

Abstract

The difficulty in finding optimal solutions in an acceptable time has been challenging for computing. As a more efficient strategy to deal with this problem, solutions focused on genetic algorithms have been proposed. Based on this programming technique, the textit Biased Random-Key Genetic Algorithm with online parameter settings (A-BRKGA) explores the search space to find interesting solutions. Given the difficulty in determining such solutions for large problem instances in feasible time, the objective of this work is to identify the main points of parallelization and then apply concurrent and parallel programming techniques. Specifically, the purpose was to improve the performance of the local search with the help of the OpenACC standard, in which it is possible to obtain a Speedup of 12.13. At the end, the results obtained with the respective discussions are presented, in addition to demonstrating the time gain in the process of convergence to the solution and verifying comparative studies related to the execution time in general.

Key-words: parallel programming, OpenACC, concurrent and distributed programming, A-BRKGA.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CPU	*Central Processing Unit*, Unidade de Processamento Central
GPU	*Graphics Processing Unit*, Unidade de Processamento Gráfico
CUDA	*Compute Unified Device Architecture*, Arquitetura de Dispositivo de Computação Unificada
AG	Algortimo Genético
RKGA	*Random-key genetic algorithms*, Algoritmo Genético de Chaves Aleatórias
API	*Application Programming Interface*, Interface de Programação de Aplicativos
CUDA	*Compute Unified Device Architecture*
SIMD	*Single Instruction, Multiple Data*
ALU	*Arithmetic Logical Units*, Unidade Lógica e Aritmética
PGProf	*PGI Profiler*
GProf	*GNU Profiler*

Sumário

1 Introdução

1.1 Contextualização e Motivação

A solução de problemas combinatórios requer, normalmente, bastante poder computacional para encontrar seus resultados. O objetivo de um problema combinatório pode ser sintetizado em maximizar a procura pelo espaço de busca de soluções por meio da aplicação de diferentes métodos. Dentre as abordagens conhecidas, deve-se destacar as metaheurísticas. Algoritmos baseados nessa estratégia não oferecem garantias de que a solução ótima será encontrada para o problema em questão. Por outro lado, podem apresentar tendências de que a heurística converge para soluções interessantes.

Logo, através das metaheurísticas, pode-se obter respostas subótimas valiosas para problemas considerados complexos, com custo de processamento menor quando comparado a soluções ótimas (**??**).

Por outro lado, a construção de uma metaheurística requer a organização adequada sobre quais estratégias adotar. Além disso, pode ser relevante considerar informações prévias para ajudar o algoritmo na tomada de decisão.

Dentro desse contexto, tem-se o algoritmo genético de chaves aleatórias viciadas (*biased random-key genetic algorithm* – BRKGA), o qual consiste na aplicação de técnicas de metaheurística evolutiva para encontrar soluções interessantes para determinados problemas (CHAVES; GONÇALVES; LORENA, 2018).

Dado que a aplicação do algoritmo BRKGA pode exigir significativa demanda computacional, tem-se a necessidade de buscar otimizações em tempo de execução. Nesse sentido, o padrão de programação denominado OpenACC, oferece mecanismos que simplificam a portabilidade de códigos seriais para códigos paralelos.

O processo de migração de aplicações seriais para um código paralelo é feito por meio de etapas. A primeira delas é o estudo e identificação dos trechos de código que tomam mais tempo do processamento seguida por paralelização de laços demandantes, otimização de possíveis transferência de dados e, por fim, possíveis aprimoramentos apontados por ferramentas chamadas de *profilers* (OPENACC.ORG, 2015a).

Através da técnica de programação paralela OpenACC, diretivas específicas são passadas para o compilador que, por sua vez, assume a responsabilidade de traduzir o algoritmo para instruções que sejam capazes de orientar unidades de processamento com arquitetura diferentes como GPU e CPU.

Dessa forma, após avaliar o comportamento da aplicação A-BRKGA com a finalidade

de identificar trechos mais demandantes em tempo de execução, teve-se como proposta otimizar a busca local da aplicação A-BRKGA através de diretivas oferecidas pelo padrão de programação paralela OpenACC.

1.2 Definição do problema

Dado o escopo de otimização combinatória, a definição do problema consiste em aplicar a função objetivo em um domínio finito. Apesar de ser delimitado, o domínio é em geral bastante amplo. Em vista disso, a utilização de algoritmos que verificam todas as instância do domínio para chegar a solução se tornam impraticáveis (MIYAZAWA; SOUZA, 2015).

Desse modo, tem-se a necessidade de recorrer a técnicas que apresentem soluções sem garantias de serem ótimas, mas que sejam minimamente interessantes. E, principalmente, sejam capazes de mostrar os resultados em tempo computacionalmente aceitável.

Dentro desse contexto, pode ser aplicado algoritmos baseado em heurísticas como o A-BRKGA cujo objetivo é aplicar conceitos de evolução genética para identificar resultados interessantes dentre as possibilidades existentes (CHAVES; GONÇALVES; LORENA, 2018). Em contrapartida, não há garantias de que a solução ótima será encontrada, mas apenas que alguma solução aproximada será dada com menor tempo (MALAQUIAS, 2006).

Problemas que apresentam domínio de instâncias muito grande são tratados como complexos de serem resolvidos em tempo polinomial. A tarefa de verificar cada combinação de instância exige muito tempo para ser solucionado, até mesmo para algoritmos elegantes que exploram todo o espaço de busca de maneira mais inteligente (CORMEN, 2009).

Ao considerar esse cenário, a computação paralela e concorrente é uma estratégia que pode ser utilizada para otimizar o tempo de execução do algoritmo, especialmente, para instâncias de problemas grandes. No caso do método A-BRKGA, a paralelização das tarefas pode permitir que o algoritmo consiga apresentar solução em menor tempo de execução.

1.3 Justificativas

Dado o tempo proibitivo para encontrar boas soluções para problemas custosos mesmo com a técnica de metaheurística, a paralelização do método permite que resultados sejam obtidos com mais agilidade e, portanto, em tempo de execução factível. No entanto, é necessário pontuar a dificuldade para aplicar essa estratégia, especialmente em códigos legados, os quais podem apresentar dependências em suas instruções que impedem o processamento concorrente.

Dominar o fluxo de operação do método A-BRKGA, principalmente, relativo a busca local pode ser colocado como primeira dificuldade. O uso de ferramentas de compilação e análise foram utilizadas para obter mais detalhes sobre o comportamento e tempo de execução

em geral. Dessa forma, a adaptação pode ser feita com maior precisão.

Posteriormente, tem-se o desafio de conseguir aplicar a técnica de paralelização no método A-BRKGA. Durante esse processo foi necessário investigar e solucionar as dependências de variáveis, identificar as seções mais demandantes da busca local e otimizá-las, além de aprimorar a transferência de dados.

1.4 Objetivos

1.4.1 Geral

O foco deste trabalho é analisar e aplicar soluções, e estratégias para ganho de desempenho no trecho relativo a busca local, utilizada no método BRKGA (*Biased Random Key Genetic Algorithm*) com configuração *on-line* de parâmetros (*A-BRKGA*), por meio de programação para computação paralela com portabilidade para GPU e CPU através do padrão OpenACC.

1.4.2 Específicos

- Revisão bibliográfica sobre metaheurística, BRKGA e A-BRKGA.
- Avaliação de desempenho do método A-BRKGA para instâncias diferentes de problema.
- Criação de código paralelo para GPU e *multicore*, através do OpenACC.
- Apresentar os estudos comparativos entre as estratégias e soluções aplicadas.
- Discutir os resultados obtidos com a paralelização da busca local.

1.5 Organização do documento

Seguido deste primeiro capítulo introdutório, o Capítulo 2 trata sobre a revisão bibliográfica com conceitos básicos sobre otimização combinatória e metaheurísticas, BRKGA e A-BRKGA, programação concorrente e paralela, além de conteúdo sobre modelos de programação para alto-desempenho.

Na sequência, o processo de metodologia é descrito no Capítulo 3. E, por fim, a etapa de desenvolvimento é descrita no Capítulo 4 seguida pelos Capítulos de resultados e discussões 5, e conclusão 6.

2 Revisão Bibliográfica

2.1 Otimização combinatória e metaheurísticas

A otimização combinatória é uma área da ciência da computação que visa estudar problemas de otimização com ajuda de fundamentos matemáticos. O estudo é feito por meio de uma função objetivo e de um conjunto de restrições que se relacionam à variáveis de decisão (incógnitas que precisam ser avaliadas pelo problema em questão, para chegar ou convergir a algum objetivo) (MALAQUIAS, 2006).

O relacionamento entre a função objetivo e o conjunto de restrições é feito através da definição da modelagem, cuja finalidade é descrever sistema real ou algum comportamento desejado. Dessa forma, a estruturação da otimização combinatória permite a exploração e aplicação de hipóteses com a finalidade de maximizar o ganho ou minimizar a perda. A confiabilidade do resultado obtido é dependente da validação do modelo e o quão representativo é no contexto do problema (LISBOA, 2002).

Além disso, algumas experimentações podem levar a apresentação de resultados irrelevantes, ou seja, que não agregam valor como solução. Estes casos são denominados de ótimos locais e estão contidos no espaço de busca assim como soluçoes interessantes e ótimas.

Por sua vez, o conjunto denominado de espaço de busca contém todas as possibilidades possíveis e viáveis para o problema e é limitado pelas funções de restrições de forma a serem associadas como pontos no espaço de busca.

Deve-se considerar que, apesar da existência de muitas classificações para problemas de otimização, existem casos exatos e aproximados. Determinados problemas exigem que métodos baseados em heurísticas sejam aplicados para encontrar uma boa resolução em tempo factível, especialmente em casos que possuem um grande espaço de busca.

A utilização de estratégias heurísticas na resolução de problemas de otimização permite, conforme demonstrado pela literatura especializada, que o método convirja para alguma solução aproximada com menor tempo. Afinal, encontrar a solução ótima para problemas custosos requer, normalmente, um tempo proibitivo (MALAQUIAS, 2006).

2.1.1 Algoritmos Genéticos

Os algoritmos genéticos (AG) são uma técnica baseada em algoritmo de otimização numérica, conceitos de seleção natural e genética evolutiva. A aplicabilidade da estratégia é bastante ampla e pode ser usada em diferentes tipos de problemas.

De acordo com Coley (2014), alguns termos chaves para os AG's são:

- indivíduo: representa uma possível solução para o problema.
- cromossomo: determina as características do indivíduo.
- população: conjunto de indivíduos.

Tipicamente, os AG's precisam apresentar um modo para avaliar a qualidade(função objetivo) dos indivíduos presentes na população que está em análise, além de mecanismos que atuem sobre os indivíduos de modo a tentar refletir o comportamento biológico dos seres vivos como seleção natural, cruzamento e mutação.

Usualmente o método é iniciado com a criação de população aleatória de indivíduos que representam possíveis soluções para o problema. Embora outros mecanismos evolutivos possam ser usados, os mais frequentes são seleção natural, cruzamento e mutação.

Após a aplicação desses três métodos auxiliares sobre a população inicial (com indivíduos pais), considera-se que uma nova população (indivíduos descendentes) foi gerada e, portanto, sendo referente outra geração de indivíduos (COLEY, 2014).

Todas as etapas mencionadas são repetidas até que o críterio de parada definido seja satisfeito. Geralmente, são utilizados o número de gerações ou medida de avaliação de convergência do método para soluções interessantes como críterio de parada.

2.1.2 Método BRKGA

A classe de algoritmos genéticos descrita por Bean (1994) pode ser usada para buscar soluções para problemas de otimização combinatória. A representação do resultado é feita por meio da utilização de vetores de permutação. Dessa forma, são descritos como *random-key genetic algorithms* (RKGA). Estas chaves são geradas dentro do intervalo contínuo [0, 1).

Para encontrar a solução, é aplicado o decodificador. Este processo é responsável por mapear o vetor de chaves aleatórios em uma solução para o problema de otimização e, ao final, retornar o custo da resolução (BEAN, 1994).

O RKGA evolui a procura no espaço de busca por meio da aplicação do princípio *darwinista*. A priorização é feita por meio de medidas de avaliação de cada geração da população. Desta forma, descendentes que apresentam convergência mais interessante para o problema podem ser selecionados (RESENDE, 2013).

Na mesma estratégia de exploração do espaço de busca das soluções, tem-se o *biased random-key genetic algorithm* (BRKGA). A principal diferença para o método anterior está no modo como os vetores considerados pais são selecionados para gerar a próxima geração de descendentes. Deste modo, O RKGA escolhe vetores da população completa, enquanto que no BRKGA duas classificações são definidas: pais pertencentes ao conjunto elite e pais pertencentes ao conjunto não-elite (RESENDE, 2013).

De acordo com Gonçalves e Resende (2011), esta diferença entre os métodos permite que o BRKGA apresente uma melhor convergência para os problemas de otimização combinatória.

2.1.3 Método A-BRKGA

Por sua vez, o *Biased Random Key Genetic Algorithm* com configuração on-line de parâmetros (CHAVES; GONÇALVES; LORENA, 2018), é ajustado para buscar balanceamento entre diversidade e intensidade ao longo da evolução das gerações. A Figura 1 representa o fluxograma do A-BRKGA.

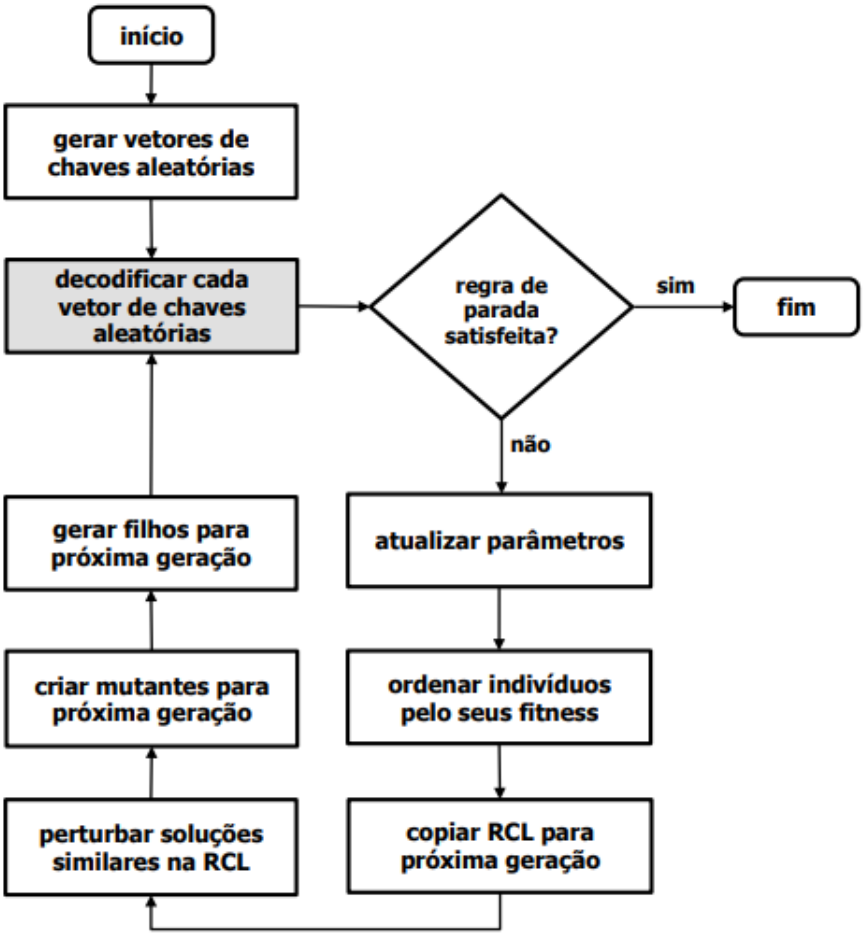

Figura 1: Fluxograma do A-BRKGA

Fonte: (CHAVES; GONÇALVES; LORENA, 2018)

Na Figura 1, o componente do fluxograma realçado representa o decodificador, responsável por realizar o *tradeoff* entre intensificação e diversificação ao longo da evolução das populações.

Entre as principais contribuições apresentadas pelo método A-BRKGA destaca-se a utilização de novas técnicas para determinar quais indivíduos serão considerados como perten-

centes ao grupo elite e a introdução de parâmetros que favoreçam a diversificação nas gerações. Além disso, fornece controle de parâmetros *on-line* com definições determinísticas e capazes de se autoadaptarem. Essas características tornam o método de fácil utilização e com boa robustez para seus usuários (CHAVES; GONÇALVES; LORENA, 2018).

2.1.3.1 Aplicado ao *Traveling Salesman Problem (TSP)*

No *Traveling Salesman Problem* ou Problema do Caixeiro Viajante um vendedor precisa visitar certo número de cidades. O desafio é encontrar a menor rota que passe por cada uma das cidades de maneira única e que também termine no mesmo local onde começou. O problema pode ser modelado como grafo não-direcionado G = (V, E) onde V representa o conjunto de vértices(cidades) e E representa o conjunto de arestas que conectam dois vértices com as respectivas distâncias associadas. (CORMEN, 2009)

De acordo com Cormen (2009) TSP é classificado NP-Hard e, portanto, ainda não é conhecido algoritmo capaz de resolvê-lo em tempo polinomial. Na literatura pode-se encontrar exemplos de instâncias TSP's inclusive com os valores ótimos de resoluções anteriores para fins de comparação.

O método A-BRKGA pode ser aplicado ao TSP de modo a encontrar soluções interessantes em tempo computacional adequado.

2.2 Programação Concorrente e Paralela

Durante os anos de 1986 e 2002, o desempenho dos microprocessadores foi melhorado em média 50% ao ano. Muitos desenvolvedores e usuários esperavam a próxima geração para diminuir o tempo de execução das aplicações. No entanto, a partir de 2002 a diferença de desempenho de uma geração de processadores para a outra passou a ser pequena por esbarrar no limite de superaquecimento da *central processing unit* (CPU) (HERLIHY; SHAVIT, 2006).

Reconhecida a barreira que os processadores monolíticos apresentavam, o caminho para ganho de desempenho passou a ser através da concorrência. No entanto, apesar da mudança de paradigma, as aplicações escritas até então foram desenvolvidas para serem executadas por um único processador e, portanto, não estavam preparadas para extrair todo o desempenho do equipamento. Ou seja, tinha-se a necessidade de construir aplicações que fossem capazes de executar em núcleos diferentes (PACHECO, 2011).

De acordo com Ben-Ari (2006), aplicação concorrente pode ser definida como conjunto de programas sequenciais que podem ser paralelizados. Entende-se como concorrência, processos que podem estar em sobreposição nos diferentes processadores e, portanto, passam a sensação de estarem em execução ao mesmo tempo. Já as aplicações paralelas são, de fato, aquelas em que os processos são executados fisicamente ao mesmo tempo em núcleos diferentes.

Conforme destaca Conte (2017), o modelo de programação paralela permite ganhos significativos em tempo de execução, principalmente, nos casos em que a comparação é feita com implementações seriais. Na Figura 2, pode-se verificar a divisão do problema inicial em outras 4 tarefas menores que podem ser resolvidas de forma paralela. Ou seja, *CPUs* diferentes podem ser alocadas para realizar as operações necessárias sobre cada subproblema. Ao final, espera-se obter o mesmo resultado que a versão serial por meio da junção dos trabalhos feitos em cada subproblema.

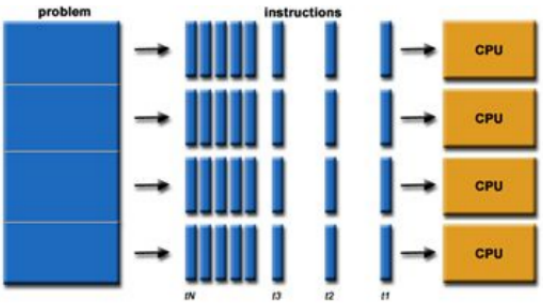

Figura 2: Divisão de tarefas entre CPUs

Fonte: (CONTE, 2017)

2.2.1 Processamento de alto desempenho

Pode-se observar a necessidade de utilização de plataformas computacionais capazes de lidar com grande volume de dados e executar cálculos custosos. Nesse contexto, faz sentido considerar ferramentas e técnicas de programação conhecidas por Processamento de Alto Desempenho (*High Performance Computing* (HPC)).

No entanto, o poder computacional oferecido por equipamentos que se enquadram nessa categoria exige um bom nível de conhecimento do programador para obter ganhos efetivos de desempenho. Além disso, deve-se ainda considerar a existência de diferentes paradigmas de programação antes de decidir qual é o melhor para empregar em determinado problema e plataforma (FAZENDA; STRINGHINI, 2019).

Existe a necessidade, portanto, de conhecer detalhes sobre métricas de desempenho, paralelização em sistemas de memória central, sistemas de memória distribuída e computação com auxílio de coprocessamento numérico por GPU.

2.2.2 Métricas clássicas de desempenho

Faz-se necessário avaliar o desempenho de aplicações a serem otimizadas para entender as regiões de código mais demandantes. Dessa forma, pode-se obter parâmetros a fim de comparar o retorno obtido com as otimizações. Em síntese, os dois principais objetivos em HPC é a obtenção de desempenho e escalabilidade. Ou seja, capacidade de reduzir o tempo para resolver

o problema de modo a contrapor com a maior alocação de recursos computacionais. Enquanto que a escalabilidade é relacionada com o comportamento do desempenho a medida que os recursos computacionais e também o tamanho do problema crescem (FAZENDA; STRINGHINI, 2019).

Dentre as métricas que podem ser utilizados estão o *Speedup* e Eficiência, conforme destacado pelas equações 2.1 e 2.2.

$$Speedup(P) = \frac{Texec(1 * proc)}{Texec(P * procs)} \tag{2.1}$$

- P = número de processadores
- $1 \leq Speedup \leq P$

Já a eficiência pode ser obtida através da equação:

$$E = \frac{Speedup(P)}{P} \tag{2.2}$$

(AMDAHL, 2007)

2.3 Modelos de programação para alto desempenho

Existem alguns modelos de programação, definidos por um determinado padrão, que oferecem mecanismos apropriados para portar trechos de códigos seriais em paralelos. Por exemplo, o *Open Multi-Processing* OpenMP.org (2018) que apresenta uma *interface* para lidar com sistemas de memória central, disponível para muitas arquiteturas.

Como trata-se de um modelo padronizado, existem diversos compiladores que permitem uma programação aderente ao padrão, como é o caso do *GNU Compiler Collection* (GCC). Para repassar instruções para o compilador, o programador deve fazer uso da diretiva ***#pragma*** combinada com as outras possíveis palavras chaves. Deve-se ainda informar o parâmetro ***fopenmp*** para o GCC, como opção de compilação. Assim, o código será devidamente compilado com as definições das diretivas empregadas no algoritmo (OPENMP.ORG, 2018).

Além do OpenMP, outro modelo de programação padronizado para *HPC* é o *Message Passing Interface* (MPI). No entanto, a premissa de utilização é diferente. Esta ferramenta é focada na troca de dados entre processos, que podem estar distribuídos em *clusters*, nós e processadores.

A utilização do MPI pode ser feita através da inclusão da biblioteca *mpi.h* e definição da lógica de comunicação por meio das próprias instruções (MPI-FORUM.ORG, 2015).

2.3.1 GPU - *Graphics Processing Unit*

Na mesma linha de programação de alto desempenho, tem-se a possibilidade de utilizar *Graphics Processing Units* (GPUs) como um coprocessador aritmético para acelerar o processamento. Trata-se de um outro paradigma de programação e que, portanto, exige que novos padrões sejam utilizados (FAZENDA; STRINGHINI, 2019).

Graças as pequenas unidades aritméticas que possuem em sua estrutura arquitetural, as GPUs são capazes de realizar grandes volumes de cálculo relativamente simples. Por isso, a ideia é aproveitar o desempenho oferecido por cada núcleo, a fim de permitir executar várias *threads* concorrentes.

As GPUs são usadas, normalmente, na forma de uma máquina pertencente a classe *Single Instruction Multiple Data* (SIMD). Em relação as CPUs, as GPUs podem obter maior eficiência quando aplicada em grande quantidade de operações simples conhecidas nas *Arithmetic Logical Units* (ALU).

A linguagem CUDA (*Compute Unified Device Architecture*), desenvolvida pela NVIDIA, é a mais conhecida plataforma de computação paralela para GPUs. Nesta, é oferecido aos programadores um conjunto de instruções que permitem configurar os recursos que serão alocados durante o processamento (LEANDRO; FERREIRA; MATSUMOTO, 2012).

Em CUDA os códigos são definidos por função chamada de *kernel*. Quando chamada, é executada paralelamente por várias *threads* na GPU. Em comparação com uma CPU, as *threads* das placas gráficas são concebidas, normalmente, para operações de menor carga computacional, e, por isso, apresentam baixo custo de alternância de contexto (CHATAIN, 2012). As *threads* podem ainda ser organizadas em blocos de uma, duas ou três dimensões. Já os blocos são organizados em estruturas unidimensionais, dimensionais ou tridimensionais. Na Figura 3, pode-se conferir um exemplo da definição de blocos e *threads* com duas dimensões. Além disso, tem-se a visão sobre a organização das *threads* que são disponibilizadas pela GPU.

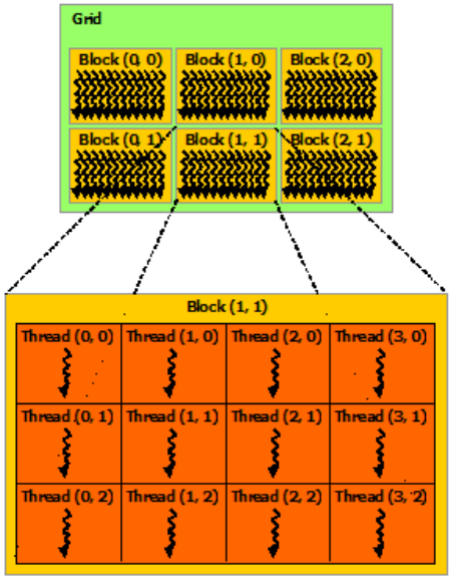

Figura 3: Organização dos blocos e threads

Fonte: (CHATAIN, 2012)

2.3.1.1 OpenACC

Dentro desse paradigma de programação, tem-se o OpenACC (*for open accelerators*) que é relativamente similar ao OpeMP, uma vez que também faz uso de diretivas (através de ***#pragma***) para informar ao compilador que é necessário lidar com o código de maneira diferente.

A proposta do OpenACC é oferecer as facilidades apresentadas pelo OpenMP para utilização de coprocessamento, mas em *hardware* aceleradores. Desta forma, o programador apenas abstrai a lógica de controle sobre os diferentes núcleos (OPENACC.ORG, 2015a).

Criado em 2011, o OpenACC é definido como uma *Application Programming Interface* (API) que estabelece um modelo com diretivas de nível mais alto aos compiladores para tornar códigos portáveis para diversas aceleradoras gráficas. Com base no padrão criado, pode-se portar o mesmo código-fonte para outros dispositivos e obter desempenho similares devido a camada de abstração de Hardware (OPENACC.ORG, 2015b).

Para garantir a portabilidade do OpenACC para diferentes arquiteturas, desde sua concepção definiu-se modelo abstato de computação acelerada com suporte para níveis de paralelismo e especificidades variadas de memória. Desse modo, os desenvolvedores podem explorar a aplicabilidade para arquiteturas diversas.

Pela Figura 4, é apresentado a organização dessa proposta fundamentada pelo OpenACC. Confome destacado pelo padrão, é oferecido suporte para computação e transferência de dados para dispositivos host e device, e de forma mais abrangente, os usuários do modelo podem explorar a aplicabilidade para arquiteturas iguais.

Alguns exemplos de arquiteturas mencionadas por OpenACC são *multicore CPUs, GPUs, DSPs e FPGAs* (OPENACC.ORG, 2015a).

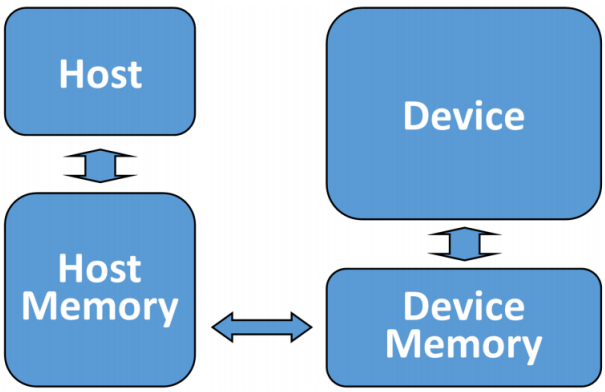

Figura 4: Modelo abstrato de aceleradores

Fonte: (OPENACC.ORG, 2015b)

Quanto a sintaxe de OpenACC podem ser definidas da seguinte forma:

```
#pragma acc <clausulas>
```

A estrutura básica deve ser respeitada. O *pragma* é necessário para informar para o compilador de que existe diretiva diferente das que são encontradas usualmente que deve ser acompanhado da palavras reservadas *acc* <*cláusulas*>. A paralelização simples da estrutura de repetição pode ser feita por meio da utilização da diretiva *loop* como no exemplo abaixo.

```
#pragma acc parallel loop
for (i = 0; i < N; i++) {
  y[i] = 0.0 f;
  x[i] = (float)(i + 1);
}
```

O processo de paralelização do código deve ser feito por meio de abordagem incremental para garantir a corretude. A recomendação é avaliar o desempenho da aplicação através da identificação dos trechos críticos. Em um segundo momento, deve-se aplicar as diretivas OpenACC em laços de repetição mais importantes e otimizar a localidade dos dados para evitar transferências desnecessárias entre o *CPU* (*Host*) e *GPU* (*Device*) (OPENACC.ORG, 2015b).

Desse modo, o chamado *Porting Cycle* é definido pelas etapas:

1. Identificação de regiões críticas.
2. Paralelização graduais de *loops* seguida de acompanhamento dos resultados para manter corretude.
3. Otimizar a transferência de dados entre o *host* e *device*.

De acordo com OpenACC.org (2015b), a análise das regiões críticas pode ser feita através da geração do perfil do algoritmo. Ou seja, com ajuda de ferramentas como, por exemplo,

pgprof e *gprof*. Com base nessa etapa, é esperado que perguntas sejam respondidas a respeito do comportamento da aplicação:

- Qual é o tempo de execução e eficiência do aplicativo?
- Quais as rotinas que gastam mais tempo e que procedimento estão sendo feitos?
- Existem pontos de entrada e saída?
- Qual é o comportamento dos laços de repetição?

Essas são algumas perguntas que ao serem respondidas ajudam a definir os pontos principais de paralelização.

3 Metodologia

Ao longo do capítulo, é apresentado as fases percorridas para a realização do trabalho assim como é descrito com mais detalhes o comportamento da busca local na qual é o alvo principal de otimização.

Além disso, são exibidas as análises iniciais do algoritmo A-BRKGA bem com as respectivas discussões. Tal etapa foi fundamental para identificar os possíveis trechos de otimização.

3.1 Análises iniciais

De acordo com o guia de programação paralela OpenACC.org (2015a), as etapas para aplicar a técnica de programação paralela são: avaliação do desempenho, análise de trecho mais demandantes, otimização de laços de repetição e, posteriormente, melhorar a transferência de dados.

Portanto, o primeiro passo para otimizar uma dada aplicação é entender quais são os laços de repetição e rotinas que correspondem a maior parte do tempo de execução. Essa fase é considerada bastante crítica dado que está diretamente relacionada aos benefícios que a aceleração pode trazer.

Para realizar essa análise do comportamento da aplicação, pode-se recorrer a ferramentas conhecidas como ***profilers*** como, por exemplo, o PGProf (2018) ou *GProf* (`http://www.cbs.dtu.dk/cgi-bin/nph-runsafe?man=gprof`). Também pode ser usado instrumentador simples ***gettimeofday*** da biblioteca <*sys/time.h*> de tempo nos trechos que deseja-se avaliar. Dessa forma, pode-se identificar quais seções demandam bastante tempo para serem executadas, conforme apresentado no capítulo de desenvolvimento.

```
1     %   cumulative   self              self     total
2    time   seconds   seconds    calls  ms/call  ms/call  name
3    79.48    523.25   523.25   199102     2.63     2.67  Decoder(TSol)
4    18.88    647.54   124.28     1588    78.26    78.26  LocalSearch(TSol)
5     1.13    655.00     7.46   198873     0.04     0.04  IC(int, int, double)
6     0.35    657.30     2.30   164217     0.01     0.02  ParametricUniformCrossover(int)
7     0.12    658.08     0.78   198831     0.00     0.00  CalculateFO(TSol)
8     0.03    658.28     0.20   164217     0.00     0.00  std::vector<TVecSol, std::allocator
9     0.02    658.40     0.12    30986     0.00     0.00  CreateInitialSolutions()
10    0.00    658.43     0.03                             A_BRKGA()
```

Figura 5: *Profiler* gerado pelo GProf

Na Figura 5, pode-se verificar cada rotina de chamada do A-BRKGA com ordenação por tempo total de execução. Embora outras seções também sejam executadas durante todo o

fluxo da aplicação, optou-se por não exibi-las pelo simples fato de não serem significantes para as análises devido ao baixo tempo de execução.

Essa mesma estrutura de análise pode ser visualizada e interpretada por meio da árvore de chamadas da Figura 6. A representação foi gerada a partir da ferramenta *gprof2dot* (https://github.com/jrfonseca/gprof2dot)

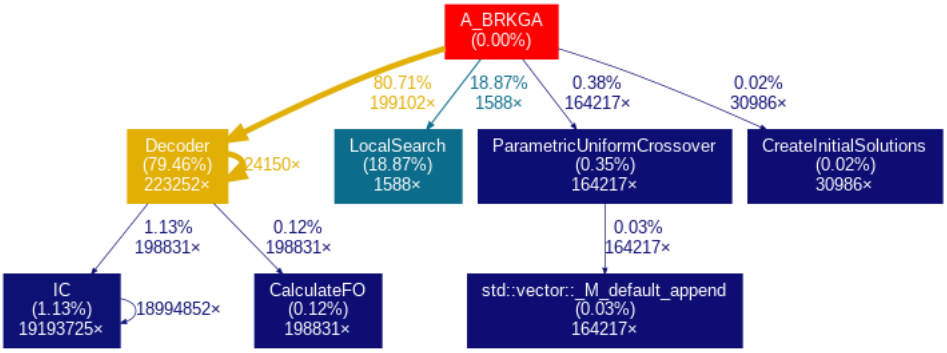

Figura 6: Análise de trechos demandantes com visualizador gprof2dot

Nessas duas representações tem-se bastante clareza sobre as rotinas que correspondem a maior parte do tempo de execução da aplicação, ou seja, *Decoder* com 79,48% e *LocalSearch* com 18,88%.

Além dessas duas representações, também foi utilizado a ferramenta *PGProf* para gerar um outro *profiler*, conforme Figura 7. As rotinas com tempo de execução menor que 5% foram excluídas.

```
1   ======== CPU profiling result (top down):
2  Time(%)      Time  Name
3   100.00%   736.31s  ???
4   100.00%   736.31s    main
5   100.00%   736.29s      A_BRKGA(void)
6   77.22%   568.61s      | Decoder(TSol)
7   21.75%   160.18s      | LocalSearch(TSol)
8
9  ======== Data collected at 100Hz frequency
10 ======== Percentage threshold: 5%
```

Figura 7: *Profiler* gerado pelo PGProf

Com base nessas análises coletadas através das ferramentas citadas, é possível ter bastante clareza dos pontos que a técnica de programação paralela pode ser aplicada. No primeiro momento, tem-se interesse nas rotinas *Decoder* e *LocalSearch*.

3.2 Busca local

A proposta do A-BRKGA é simplificar algumas decisões e, por isso, exige-se que os desenvolvedores preocupem-se com a implementação do *Decoder*, *CalculateFO* e da *Local-*

Search dado que são rotinas específicas para cada tipo de problema alvo. De modo especial é necessário avaliar a busca local.

A combinação de algoritmo evolutivo com a heurística de busca local é reconhecida como uma das metaheurísticas mais populares e que desmonstrou ser interessante devido a capacidade da técnica em explorar o espaço de busca e identificar regiões promissoras. Por outro lado, dado a natureza de intensificação da busca local pode-se encontrar ótimos locais em tais regiões promissoras.

Todavia, conforme mencionado por Chaves, Gonçalves e Lorena (2018), a aplicação da busca local para todos os indivíduos de um algoritmo evolutivo pode levar a tempo de execução inviável devido ao alto custo exigido por essa heurística. Esse cenário foi evitado no A-BRKGA ao aplicar a busca local apenas em regiões consideradas promissoras.

Portanto, apesar de a busca local não ter sido a rotina mais demandante nas análises investigativas realizadas, a literatura evidencia a complexidade requerida por essa heurística. Por esse motivo, o foco deste trabalho foi propor a paralelização da busca local usando OpenACC.

4 Desenvolvimento

Neste capítulo é descrito as iterações realizadas para aplicação da técnica de programação paralela com o OpenACC. Também é apresentado as versões serial e otimizada, além dos desafios e dificuldades encontrados ao longo do caminho.

Conforme citado em Fazenda e Stringhini (2019), além da investigação inicial sobre o tempo de execução da aplicação, é interessante realizar análises de dependências. A técnica de paralelismo pode gerar problemas de concorrência em determinados trechos de código com operações de escrita e leitura, por exemplo.

As atualizações de variáveis compartilhadas durante a execução do código pode gerar incosistências, também conhecidas como condições de corrida (*race conditions*) (FAZENDA; STRINGHINI, 2019).

Por isso, tais operações devem ser avaliadas previamente com objetivo de garantir a corretude do algortimo.

Tem-se ainda que durante o processo de portabilidade do código OpenACC.org (2015a), recomenda-se a divisão em etapas iterativas na abordagem de aceleração da aplicação. A finalidade dessa tática é garantir e facilitar as análises de corretude.

Dentre as estratégias possíveis para a paralelização da aplicação, a apresentada e seguida pelo guia é a de paralelizar laços considerados importantes, otimizar a transferência de dados entre o *host* e *device*, e reavaliar ajustes nos laços de repetição (OPENACC.ORG, 2015a).

4.1 Análise de dependências

A finalidade do primeito trecho de código 4.1.1 é calcular para toda a matriz de distância a função objetivo e guardar o resultado na matriz *AfoOpt* com as respectivas posições *i* e *j* nos vetores *AMi* e *AMj*. O segundo trecho 4.1.2 é o responsável por encontrar a menor função objetivo calculada anteriormente com sua posição.

Pode-se observar nas linhas 19 e 61 dos respectivos trechos de código 4.1.1 e 4.1.2 dependências de dados de ***j*** em relação a ***i***. Na paralelização é interessante que os limites dos laços de repetição estejam explicitamente definidos. Caso contrário, o compilador exibirá a seguinte mensagem: *Inner collapsed loop bounds are not invariant in outer loop.*

Essa questão foi resolvida por meio da definição explícita dos limites de *i* e *j* feita com base na análise do domínio do problema. A partir dessa mudança, o laço mais interno passou a percorrer toda a matriz de distância (indesejável em termos de desempenho), que foi amenizado com a aplicação do seguinte condicional `if (j >= i + 2)`.

```
18 for (int i = 0; i < n - 1; i++) {
19     for (int j = i + 2; j < n; j++) {
           ...
           // Os valores de AfoOpt[i][j], AMi[i] e AMj[j] sao definidos
           ...

51         for (int k = 0; k < n; k++) {
52             temp_fo += dist[s.vec[k%n].sol][s.vec[(k+1)%n].sol];
53         }
54     }
55 }
```

Código 4.1.1: Laços de repetição para definir AfoOpt, AMi e AMj com dependência

```
60 for (int i = 0; i < n - 1; i++) {
61     for (int j = i + 2; j < n; j++) {
62         if (AfoOpt[i][j] < Bestfo) {
63             Bestfo = AfoOpt[i][j];
64             Besti = i;
65             Bestj = j;
66         }
67     }
68 }
```

Código 4.1.2: Laços de repetição para encontrar Bestfo, Besti e Bestj com dependência

4.2 Otimização de laços de repetição

Com a elimicação de dependência feita na seção anterior, foi possível explorar a paralelização dos laços de repetição destacados a pouco. As seguintes diretivas do OpenACC.org (2015a) foram utilizadas:

- *parallel*: informa ao compilador que trata-se de região com código paralelo.
- *loop*: diretiva que sinaliza ao OpenACC a existência de laço de repetição para gerar código portável para GPU e multicore.
- *collapse*: permite converter *n* laços de repetição em apenas 1. Deve ser precedida por *loop*.
- *independent*: indica ao compilador que o laço de repetição é independente e pode ser executado de modo paralelo.
- *private*: determina as variáveis com escopo privado dentro de cada iteração.
- *atomic write*: pode ser usada para garantir atomicidade de leitura/escrita em memória, ou seja, para que não ocorra condições de corrida.

- *reduction*: similar a diretiva *private* no sentido das variáveis serem privadas dentro de cada iteração, mas atua também de modo a reduzir a operação interna do laço para um resultado final.

Após a portabilidade dos laços de repetição, obteve-se como resultado os trechos de código 4.2.1 e 4.2.2. De modo especial, é preciso destacar a solução aplicada na divisão do laços iniciados na linha 60 do trecho 4.1.2.

Conforme descrito, na operação de redução, deseja-se obter valor final de acordo com a instrução declarada. Nesse caso, espera-se que ao final das iterações a variável *Bestfo* contenha o menor valor presente na matriz. Além disso, também é de interesse do método ter conhecimento sobre a posição do menor valor na matriz, de acordo com *i* e *j*.

No entanto, diante desse cenário o processo de redução intrínseco do OpenACC não garante a atomicidade sobre as operações de atribuições das linhas 64 e 65 do trecho 4.1.2. Por esse motivo, optou-se em adotar a estratégia de primeiro encontrar o menor valor de *Bestfo* e, em seguida, buscar a sua posição na matriz por comparação. Assim pode-se evitar condições de corrida dado que não há garantias na ordem de execução no paradigma de programação concorrente.

```
17 #pragma acc parallel loop collapse(2) \
18     private(foOpt, vi, viP, viM, vj, vjM, vjP, temp_fo)
19 {
20     for (int i = 1; i < n - 1; i++) {
21         for (int j = 3; j < n - 1; j++) {
22             if (j >= i + 2) {
                   ...
                   // Os valores de AfoOpt[i][j], AMi[i] e AMj[j] sao definidos
                   ...

53                 #pragma acc loop independent reduction(+: temp_fo)
54                 for (int k = 0; k < n; k++) {
55                     temp_fo +=
56                         temp_dist[stemp_vec[k\%n].sol][stemp_vec[(k+1)\%n].sol];
57                 }
58             }
59         }
60     }
61 }
```

Código 4.2.1: Laços de repetição para encontrar Bestfo, Besti e Bestj sem dependência e otimizado

```
70 #pragma acc parallel loop collapse(2) reduction(min:Bestfo)
71 for (int i = 1; i < n - 1; i++) {
72     for (int j = 3; j < n - 1; j++) {
73         if (j >= i + 2) {
74             if (AfoOpt[i][j] < Bestfo) {
75                 Bestfo = AfoOpt[i][j];
76             }
77         }
78     }
79 }
```

```
80
81 #pragma acc parallel loop collapse(2)
82 for (int i = 1; i < n - 1; i++) {
83     for (int j = 3; j < n - 1; j++) {
84         if (j >= i + 2) {
85             if (AfoOpt[i][j] == Bestfo && (i >= Besti && j >= Bestj)) {
86                 #pragma acc atomic write
87                 Besti = i;
88                 #pragma acc atomic write
89                 Bestj = j;
90             }
91         }
92     }
93 }
```

Código 4.2.2: Laços de repetição para encontrar Bestfo, Besti e Bestj sem dependência e otimizado

4.3 Otimização das transferências de dados

O paradigma de programação paralela exige que o desenvolvedor instrua o compilador sobre como gerenciar o espaço de memória. Muitas arquiteturas de aceleração não compartilham da mesma região de dados. Dado que o padrão OpenACC permite que se use um hardware acelerador com memória independente (como as GPUs), tem-se a necessidade de considerar algumas características do processo que podem influenciar no desempenho, tal como o tempo dispendido em transferências de dados entre a memória da CPU (considerada o equipamento *host*) e a memória externa do hardware acelerador (normalmente uma GPU, considerada como *device*).

O tipo de dado a ser gerenciado também deve ser considerado durante o processo de comunicação. Conforme apresentado em OpenACC-Standard.org (2014), o padrão de programação paralela OpenACC não oferece suporte para estruturas de dados complexas como *vector* da *Standard Template Library* (STL).

Muitas das estruturas da aplicação A-BRKGA são implementadas por meio da utilização da biblioteca STL. Por esse motivo, durante a etapa de otimização das transferências de dados, teve-se a necessidade de converter os tipos de dados complexos em primitivos.

Por outro lado, deve-se destacar que apesar de ser uma rotina inexistente na versão original do A-BRKGA e que, portanto, acrescenta custo adicional, pode-se aproveitar da alocação contígua de memória da estrutura *vector STL* para ganhar agilidade.

Dentro desse cenário, as seguintes diretivas do OpenACC.org (2015a) foram usadas:

- *pcopy*: caso o dado já esteja presente na memória da acelerador, utiliza essa mesma cópia. Caso contrário, a transferência de dados é feita. Ao final do trecho de aceleração, o dado é removido.

- *copyin*: apenas realiza a cópia para o acelerador.
- *pcreate*: se a região de memória já tiver sido alocada, apenas a utiliza. Senão, realiza a alocação.
- *present*: informa para a região paralela que o dado já está presente na memória do acelerador.
- *delete*: remove os dados referenciados na claúsula e desaloca a memória.

A primeira tentativa de transferir os dados foi a estratégia de realizar cópias logo no começo de cada chamada da Busca Local. No entanto, o resultado obtido foi bastante indesejável devido a quantidade de transferências que eram realizadas e, portanto, com alto custo. Logo, teve-se a inibição dos possíveis ganhos de desempenho com a paralelização dos laços de repetição.

Na Figura 8, pode-se visualizar o comportamento da transferência de dados entre *host* para *device* (*MemCpy - HtoD*) e *device* para *host* (*MemCpy - DtoH*) para esse cenário, o qual apresenta tempos significativos na realização da tarefa. Percebe-se ainda que a execução do algoritmo de busca local na GPU (*Kernel*) somente é executado após as transferências de dados serem executadas. A partir disso, buscou-se melhorar o modo como a cópia era feita.

Pode-se obter melhor granularidade nas transferências de dados por meio da estratégia de copiar os dados e alocar memória para a paralelização antes da chamada do A-BRKGA. Foi possível aplicar essa solução para as variáveis utilizadas na região paralela e que não são atualizadas em nenhuma outra rotina. Nesse caso, são a matriz de distância (*temp_dist*), matriz com valores da função objetivo (*AfoOpt*) e os *arrays* de índices das posições de cada função objetivo (*AMi* e *AMj*). O resultado pode ser visto na Figura 9 onde o tempo dispendido com as transferências de dados entre as execuções da busca local na GPU diminuíram, uma vez que o volume de dados foi reduzido. O resultado obtido corresponde a diretiva inserida na linha 12 do trecho 4.3.1.

As outras variáveis que não se enquadraram nos requisitos descritos a pouco, foram mantidas no processo de transferência para cada busca local, conforme linha 10 do trecho 4.3.3. Ao final do fluxo de execução do A-BRKGA, a memória do *host* e *device* são liberadas conforme a linha 9 do trecho 4.3.2.

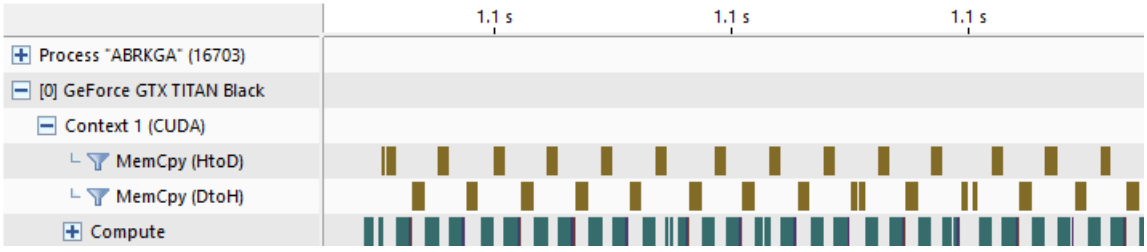

Figura 8: Transferência de dados para 1 rodada da busca local - PGProf

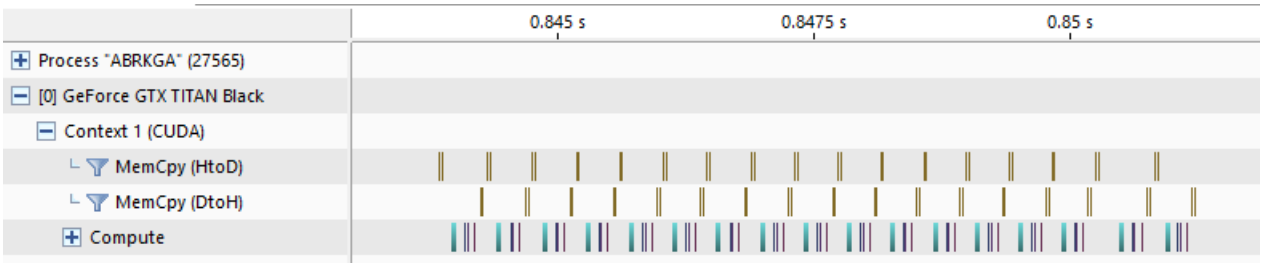

Figura 9: Transferência de dados otimizada para 1 rodada da busca local - PGProf

```
1  void mallocAcc() {
2      AfoOpt = (double **)malloc(n * sizeof(double *));
3      temp_dist = (double **)malloc(n * sizeof(double *));
4      AMi = (int *)malloc(n * sizeof(int));
5      AMj = (int *)malloc(n * sizeof(int));
6
7      for (int i = 0; i < n; i++) {
8          AfoOpt[i] = (double *)malloc(n * sizeof(double));
9          temp_dist[i] = &dist[i][0];
10     }
11
12     #pragma acc enter data copyin(temp_dist[0:n][0:n]) \
13         create(AfoOpt[0:n][0:n], AMi[0:n], AMj[0:n])
14 }
```

Código 4.3.1: Alocação de memória e transferência de dados

```
1  void deallocAcc() {
2      for (int i = 0; i < n; i++) {
3          free(AfoOpt[i]);
4      }
5      free(AfoOpt);
6      free(temp_dist);
7      free(AMi);
8      free(AMj);
9      #pragma acc exit data \
10         delete(temp_dist[0:n][0:n], AfoOpt[0:n][0:n], AMi[0:n], AMj[0:n])
11 }
```

Código 4.3.2: Desalocação de memória

```
1  TSol LocalSearch(TSol s)
2  {
       ...
10     #pragma acc data \
11         pcopy(stemp_vec[0 : n], stemp_fo) \
12         pcreate(Bestfo, Bestj, Besti) \
13         present(temp_dist[0:n][0:n], AfoOpt[0:n][0:n], AMi[0:n], AMj[0:n])
14     {
           ...
       }
       ...
132    return s;
133 }
```

Código 4.3.3: Transferência de dados para cada busca local

4.4 Trechos seriais

Os trecho de código restantes da Busca Local não foram paralelizadas por serem instruções simples, ou seja, são apenas atribuições, somas e verificações condicionais. Para esses casos foi necessário utilizar a diretiva *serial* para sinalizar ao compilador que o trecho em questão deve ser executado serialmente com dados que estão dentro do escopo de memória do dispositivo acelerador. Mais detalhes podem ser encontrados no trecho de código final no Anexo B.

5 Resultados

Neste capítulo são apresentados os resultados obtidos com a versão do A-BRKGA com busca local paralela e os devidos comparativos com a versão serial, além de discussões feitas com análises interpretativas e justificativas. Ao final, destaca-se possíveis pontos de melhoria, bem como a aplicabilidade da técnica de programação paralela para este tipo de problema.

Os resultados foram coletados por meio de 4 experimentos/execuções realizados em cada uma das 3 versões analisadas. Como conjunto de dados de entrada tem-se duas instâncias, zi929.tsp e nrw1379.tsp.

A especificação do hardware utilizado está descrito na Tabelas 1, para o servidor, e 2 para a placa gráfica (GPU) utilizada como coprocessador.

CPU	2x Intel(R) Xeon(R) CPU E5-2660 v4@2.00GHz
Número de núcleos por *socket*	14
Frequência base / Turbo	2.00/3.20 GHz
Memória princial	128GB
Cache	35 MB

Tabela 1: Especificação da CPU

GPU	GeForce GTX TITAN BLACK
Velocidade do clock	980 MHz (boost) / 889 MHz (base)
Cuda Cores	2880
Compute Capability	3.5
Memória	6144MB GDDR5 (384 bits)
Velocidade do clock da memória	3500 MHz (DDR7000)
Largura da Banda	336 GB/s
Barramento	PCI-Express 3.0 x 16

Tabela 2: Especificação da GPU

Todos os testes foram baseados nas instâncias zi929 (http://www.math.uwaterloo.ca/tsp/world/zilog.html) e nrw1379 (http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/). O primeiro representa 929 cidades do país Zimbábue e o segundo 1379 locais do estado Renânia do Norte-Vestfália da Alemanha, cada um com as respectivas coordenadas.

Em relação a compilação os seguintes comandos foram usados:

- Serial: pgc++ -fast -noacc ABRKGA.cpp -O3 -o ABRKGA-SERIAL
- GPU: pgc++ -fast -ta=tesla -acc -Minfo=accel ABRKGA.cpp -O3 -o ABRKGA-GPU
- Multicore: pgc++ -fast -ta=multicore -acc -Minfo=accel ABRKGA.cpp -O3 -o ABRKGA-MULTICORE

Tanto a flag ***fast*** como ***03*** aplicam otimizações durante a compilação. As opções -noacc e -acc instrui o compilador a não ativar e ativar o OpenACC, respectivamente. Já a flag ***ta*** define qual acelerador usar. Para definir o número de cores utilizados na versão multicore é preciso exportar a variável de ambiente ACC_NUM_CORES com a quantidade desejada.

Nos experimentos apresentados as colunas entituladas como *FO* representam o valor da função objetivo para a solução encontrada. Dado a natureza do problema TSP quanto menor o valor de *FO* melhor é o resultado. A título de comparação zi929 tem como valor ótimo *FO* = 95345 e nrw1379 *FO* = 56638.

5.1 Versão serial

Para avaliar ganho de desempenho obtido com as alterações aplicadas, é preciso extrair informações sobre o desempenho da versão serial. Este processo foi realizada por meio de instrumentadores de tempo simples com captura de tempo inicial e final. A diferença entre esses dois pontos é a medida considerada para o tempo de execução da busca local, conforme *profilers* avaliados e apresentados previamente no Capítulo 3.

Nas Tabelas 3 e 4 são apresentados os dados coletados para duas instâncias de problemas com 929 e 1379 pontos, com 4 execuções sucessivas, onde a coluna FO indica o valor da Função Objetivo, Total(ms) representa os tempos de execução total em milissegundos, Busca Local (ms) representa o tempo especificamente despendido para as operações busca local, e a última coluna (Percentagem) indica a fração de tempo que a busca local representa em relação ao tempo total. Pela Tabela 5, os dados de tempo médio são exibidos. Para a instância zi929, tem-se um tempo de execução da busca local correspondente à 21,20% do tempo total. Para nrw1379 esse percentual foi de 23.79%.

Experimento	FO	Total (ms)	Busca local (ms)	Porcentagem
1	96572	728726	154459	21.20%
2	96572	725432	154134	21.25%
3	96572	723720	153232	21.17%
4	96572	726126	153817	21.18%

Tabela 3: Experimentos versão Serial (zi929)

Experimento	FO	Total(ms)	Busca local (ms)	Porcentagem
1	59368	1761512	417747	23.72%
2	59368	1736487	414140	23.85%
3	59368	1742275	414332	23.78%
4	59368	1754857	418104	23.83%

Tabela 4: Experimentos versão Serial (nrw1379)

Instância	Total (ms)	Busca local (ms)	Porcentagem
zi929	726001	153911	21.20%
nrw1379	1748783	416081	23.79%

Tabela 5: Serial - médias para cada instância de problema

5.2 Versão GPU

De modo similar, as mesmas condições descritas na análise da versão serial foram utilizadas para a versão GPU. Tanto para a instância zi929 como para nrw1379, foi possível coletar os tempos de execução conforme Tabelas 6, 7 e 8.

Experimento	FO	Total(ms)	Busca local (ms)	Porcentagem
1	97085	622228	55333	8.89%
2	97085	623784	55636	8.92%
3	97085	624114	55571	8.90%
4	97085	626374	55634	8.88%

Tabela 6: Experimentos versão GPU (zi929)

Experimento	FO	Total(ms)	Busca local (ms)	Porcentagem
1	61684	1530058	199539	13.04%
2	61684	1551348	200786	12.94%
3	61684	1559734	200529	12.86%
4	61684	1557966	200022	12.84%

Tabela 7: Experimentos versão GPU (nrw1379)

Instância	Total (ms)	Busca local (ms)	Porcentagem
zi929	624125	55544	8.90%
nrw1379	1549777	200219	12.92%

Tabela 8: GPU - médias para cada instância de problema

Ao comparar os resultados obtidos com a versão serial, chega-se a uma economia de tempo de 101.876 ms no tempo total de execução para zi929 e 199006 ms para nrw1379. A noção dos ganhos de desempenho da nova versão acelerada para GPUs em relação a versão serial pode ser representada por meio do cálculo do Speedup.

Instância	Speedup	Porcentagem melhorada
zi929	1.16	14.03%
nrw1379	1.13	11.38%

Tabela 9: Medidas de desempenho do tempo total

Instância	Speedup	Porcentagem melhorada
zi929	2.77	63.91%
nrw1379	2.08	51.88%

Tabela 10: Medidas de desempenho da busca local

Pode-se notar pela Tabelas 9 e 10, que tanto o tempo de execução total da aplicação, quanto a busca local apresentaram ganho de desempenho. Em especial, a busca local obteve um Speedup de 2,77 e 2,08 para as duas instâncias utilizadas (zi929 e nrw1379).

Por meio da ferramenta de análise PGProf, pode-se obter indicativos sobre otimizações que ainda podem ser aplicadas ou até mesmo reavaliadas de forma a melhorar o desempenho. Na Figura 10 é mostrado as notificações geradas para a versão final da busca local paralelizada neste trabalho para execuções em GPU.

- A primeira notificação diz respeito a possibilidade de sobreposição entre as etapas de computação e transferências de dados, pois isso permitiria que essas operações pudessem ser realizadas de forma assíncrona.
- A segunda notificação informa que existe baixa concorrência entre os *kernels* lançados em GPU.
- A terceira notificação informa que as transferências de dados entre *host* e *device* utilizam uma baixa quantidade de dados, o que não permite usufruir da alta largura de banda para essa comunicação, uma vez que a latência no procedimento será muito significativa.
- A quarta notificação informa ainda que a largura de banda não está sendo utilizada em toda a sua potencialidade.
- A quinta notificação informa que há muito tempo ocioso dos núcleos de processamento da GPU, mostrando ineficiência do processo.

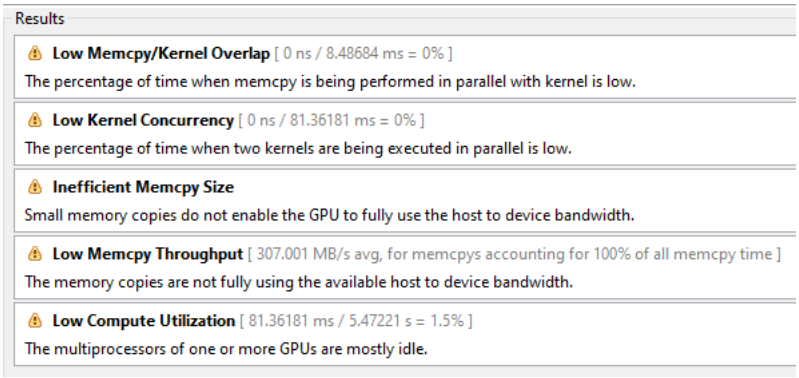

Figura 10: Indicativos de pontos de melhoria da otimização

5.3 Versão multicore

Com a facilidade oferecida pelo OpenACC sobre portabilidade de código para GPU e CPU, o mesmo código foi compilado para arquitetura multicore. Para esse caso, algumas configurações de número de *core* foram utilizadas para extrair as métricas de interesse, conforme as duas seções anteriores.

As Tabelas 11 e 12 mostram a média de 4 execuções para os tempos de computação obtidos com as instâncias zi929 e nrw1379, respectivamente, onde cada linha da Tabela representa a execução com um número crescente de *cores*, que corresponde a mesma quantidade de núcleos utilizados em cada experimento. O limite de 28 *cores* corresponde a quantidade máxima de núcleos disponíveis no servidor utilizado.

Ao observar os resultados sintetizados nas Tabelas 11 e 12, fica evidente o ganho de desempenho obtido com 28 núcleos para ambas instâncias de teste. Tal resultado é surpreendente pelo fato de ter sido consideravelmente melhor que a versão executada em GPU. Como

Cores	Total (ms)	Busca local (ms)	Porcentagem
2	688318	126615	18.39%
4	632239	52277	8.27%
8	596675	28159	4.72%
16	596756	17036	2.85%
28	583854	16666	2.85%

Tabela 11: Multicore - médias para instância zi929

Cores	Total (ms)	Busca local (ms)	Porcentagem
2	1694766	361884	21.35%
4	1581480	207927	13.15%
8	1455203	105751	7.27%
16	1349787	54863	4.06%
28	1370630	34312	2.50%

Tabela 12: Multicore - médias para instância nrw1379

consequência disso, a principal reflexão que surge é em relação a outras otimizações possíveis de serem feitas para aproveitar o poder de processamento oferecido pela arquitetura de GPU's.

Através dos 4 experimentos realizados para as diferentes configurações de *cores*, o resultado sintetizado está representado nas Tabelas 13 e 14 para a instância zi9292, e nas Tabelas 15 e 16 para a instância nrw1370. As Tabelas informam o *speedup*, eficiência no uso dos recursos e a porcentagem de tempo economizado em relação a versão serial, obtido em relação ao tempo total e exclusivamente para o tempo da busca local, respectivamente, e por instância.

Cores	Speedup	Eficiência	Porcentagem melhorada
2	1.05	52.74%	5.19%
4	1.15	28.71%	12.91%
8	1.22	15.21%	17.81%
16	1.22	7.60%	17.80%
28	1.24	4.44%	19.58%

Tabela 13: Medidas de desempenho do tempo total (zi929)

Cores	Speedup	Eficiência	Porcentagem melhorada
2	1.22	60.78%	17.73%
4	2.94	73.60%	66.03%
8	5.47	68.32%	81.70%
16	9.03	56.47%	88.93%
28	9.24	32.98%	89.17%

Tabela 14: Medidas de desempenho da busca local (zi929)

Cores	Speedup	Eficiência	Porcentagem melhorada
2	1.03	51.59%	3.09%
4	1.11	27.64%	9.57%
8	1.20	15.02%	16.79%
16	1.30	8.10%	22.82%
28	1.28	4.56%	21.62%

Tabela 15: Medidas de desempenho do tempo total (nrw1379)

Cores	Speedup	Eficiência	Porcentagem melhorada
2	1.15	57.49%	13.03%
4	2.00	50.03%	50.03%
8	3.93	49.18%	74.58%
16	7.58	47.40%	86.81%
28	12.13	43.31%	91.75%

Tabela 16: Medidas de desempenho da busca local (nrw1379)

Dentre todas as versões, experimentos e instâncias avaliadas o melhor resultado obtido está representado em 15 e 16. Pode-se notar que a busca local foi melhorada em 91,75%, o que correspondeu a diminuir o tempo de execução do A-BRKGA em 21,62% na versão com 28 cores.

5.4 Gráficos comparativos entre versões serial, GPU e Multicore

O Gráfico 11, representa o tempo total de execução da aplicação A-BRKGA para cada uma das versões. Os dados usados são das Tabelas 5, 8, 11 e 12. No caso da versão multicore,

os valores utilizados para a construção do gráfico são referentes a 28 cores.

De maneira similar, o gráfico 12 foi elaborado com uso das informações das mesmas tabelas destacadas logo acima. Como é possível observar o tempo de execução total apresentou queda em todos os casos. Com a paralelização da busca local, a redução foi considerável tanto para GPU como para *multicore*.

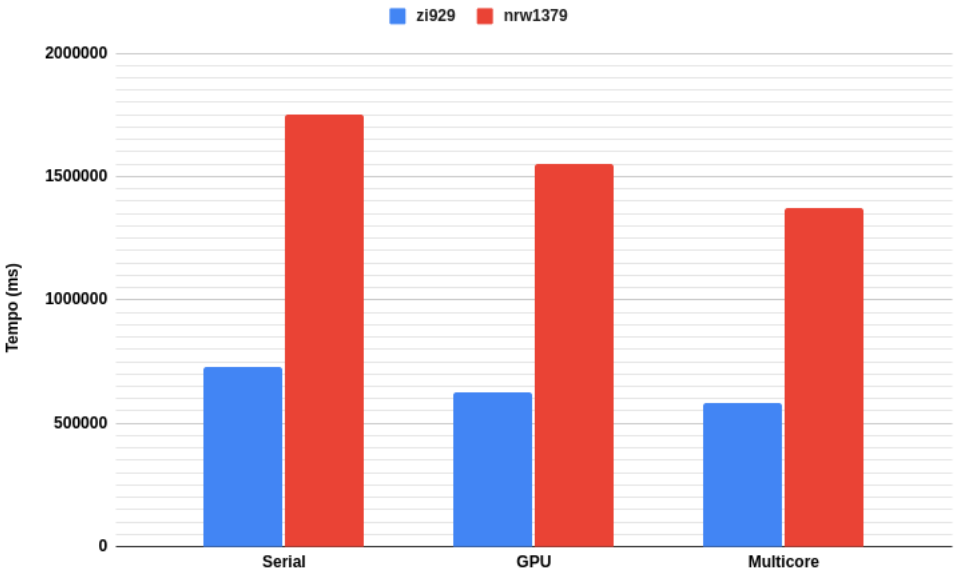

Figura 11: Gráfico do tempo total de execução para serial, GPU e multicore

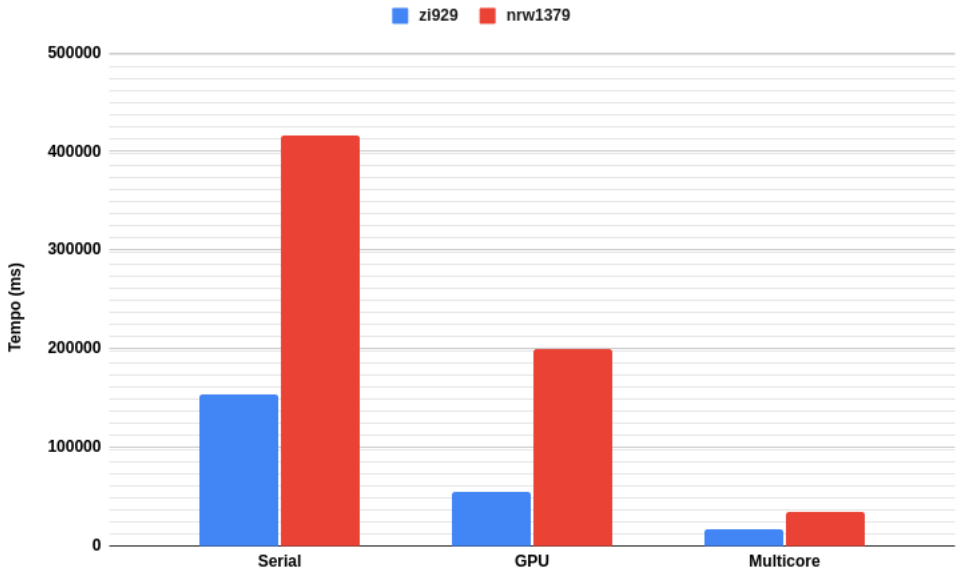

Figura 12: Gráfico do tempo da busca local para serial, GPU e multicore

5.4.1 Métricas clássicas aplicadas ao problema

Nos gráficos 14 e 13 é representado a medida de análise de desempenho *Speedup*, em relação ao tempo total e a busca local, respectivamente. O *Speedup* é a métrica utilizada para avaliar o quanto uma aplicação A é mais rápida que uma aplicação B. Para o contexto desse trabalho, a interpretação que pode ser feita é a seguinte: o quão mais rápida a versão GPU/multicore foi em relação a versão serial.

Logo, o desempenho apresentado pela busca local na versão de 28 cores foi 12,13 vezes mais rápido que a versão serial para a instância nrw1379, conforme Figura 13. Esse ganho refletiu em uma melhora no desempenho geral do A-BRKGA que correspondeu a 1,28 vezes mais rápido que a verão serial como demonstrado na Figura 14.

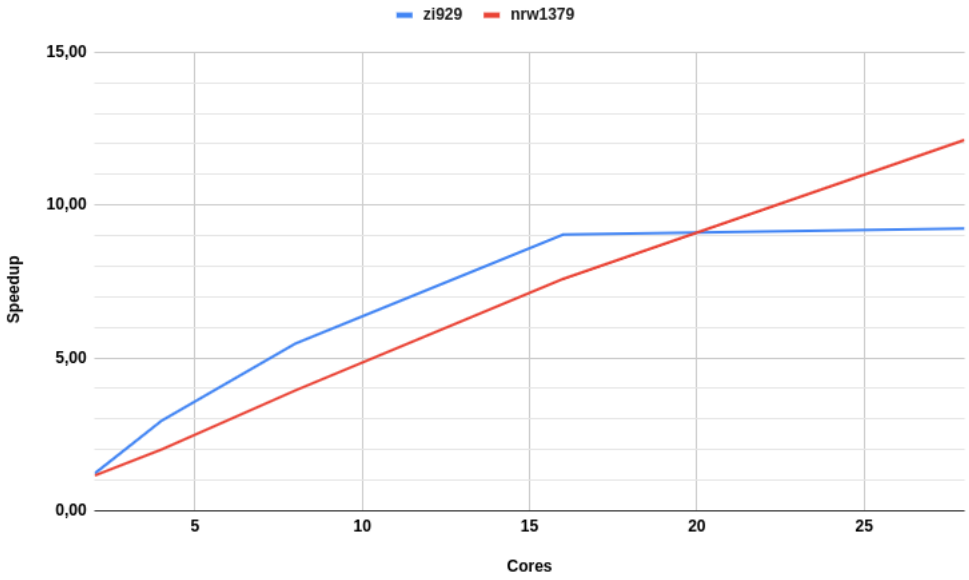

Figura 13: Gráfico do Speedup em relação ao tempo da busca local

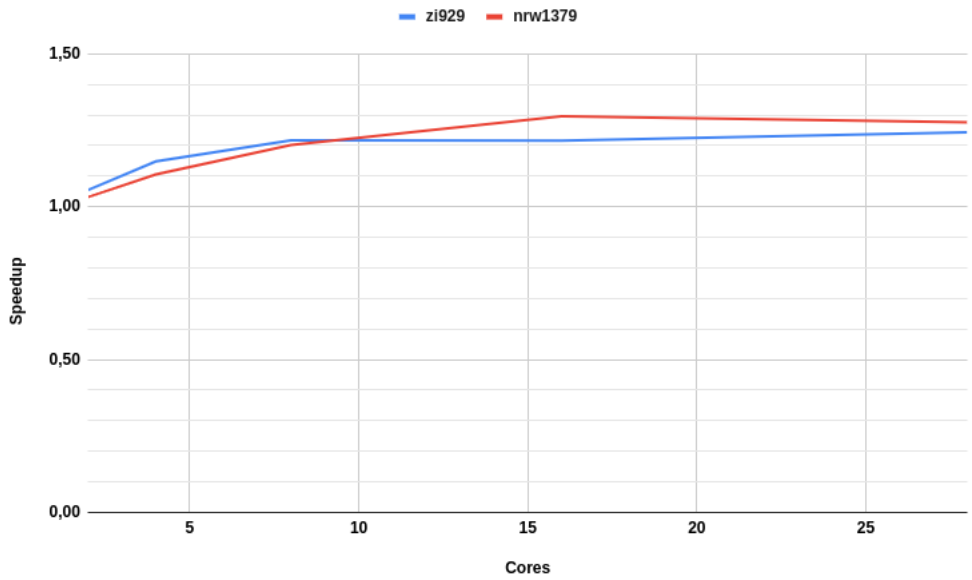

Figura 14: Gráfico do Speedup em relação ao tempo total de execução

Além de avaliar o *Speedup*, é interesse investigar o uso dos processadores que é dado pela medida eficiência. Nos gráficos 15 e 16 os valores de eficiência são mostrados para cada problema analisado nesse trabalho. Como é possível notar, o comportamento de ambos os casos é diminuir conforme o número de *cores* aumenta.

Ou seja, apesar de mais processamento estar sendo alocado entre as variações de 2 para 4, 4 para 8, 8 para 16 e 16 para 28 *cores* (dobro de *cores* alocados com exceção de 16 para 28), o crescimento do *Speedup* não acompanha essa mudança. Isso pode ser justificado por meio da conhecida Lei de Amdhal, a qual estabelece que o problema pode ser dividido em duas categorias: trechos paralelos e seriais. A parte serial é apontada como o limite inferir para o tempo de execução independente do aumento na alocação de recursos computacionais (FAZENDA; STRINGHINI, 2019).

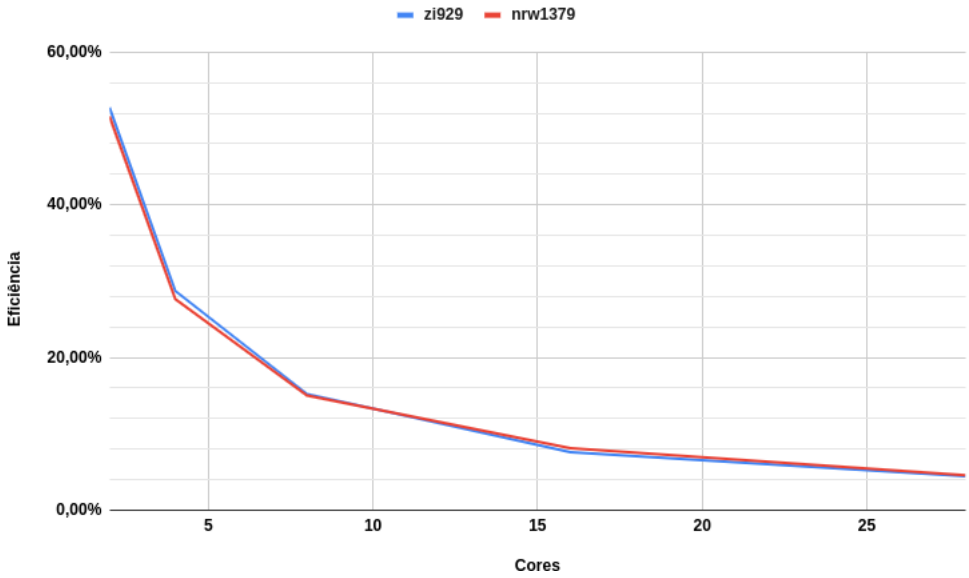

Figura 15: Gráfico da eficiência em relação ao tempo total de execução

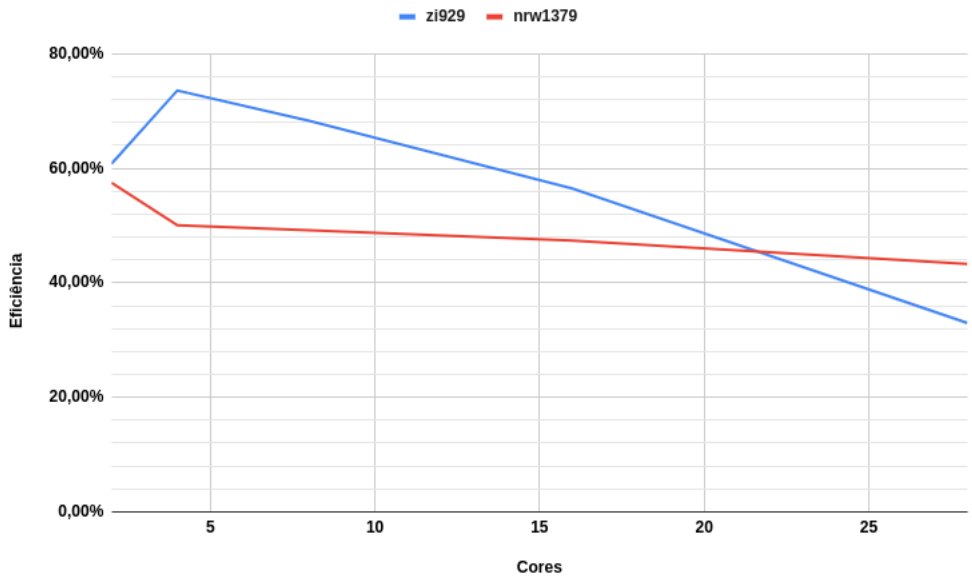

Figura 16: Gráfico da eficiência em relação ao tempo da busca local

6 Conclusão

O padrão de programação paralela OpenACC traz algumas facilidades durante o processo de otimização, visto que permite o desenvolvimento incremental, através da inclusão de diretivas, além de permitir a flexibilidade e portabilidade entre diversos diferentes dispositivos, entretanto, ainda impõe muitos desafios, especialmente relacionados ao uso com GPUs, como ficou evidente no desempenho obtido, aquém dos ganhos com uso de CPUs *multicore*.

Ao longo do trabalho ficou evidente a contribuição da disciplina *Programação Concorrente e Distribuída*, que introduz conceitos importantes sobre esse paradigma de programação. Também deve-se destacar a importância das análises iniciais feitas por meio de *profilers* e de instrumentadores de tempo simples para a obtenção dos resultados apresentados.

Paralelizar um dado algoritmo envolve muito mais do que aplicar as diretivas oferecidas pelo OpenACC. Exige-se, principalmente, conhecimento sobre o comportamento da aplicação e como os dados são atualizados ao londo do ciclo de execução.

A otimização de desempenho feita melhorou a busca local em 63.91% (Speedup de 2,77) na versão GPU, enquanto que para a versão multicore obteve-se 91,75% (Speedup de 12,13). A representatividade desses resultados em relação ao tempo de execução total foi de, respecetivamente, 14.03% (Speedup de 1,16) e 21.62% (Speedup de 1,28).

6.1 Trabalhos futuros

Através do *profiler* PGProf pode-se identificar pontos de melhoria nas otimizações feitas até o momento. A exploração desses indicativos pode trazer resultados mais interessantes, principalmente para GPU. Pelo gráfico 15 pode-se identificar que eficiência da paralelização relativamente baixa, o que pode indicar possível saturação em apenas aumentar o número de cores.

Deve-se considerar também as dificuldades entrentadas com a *Standard Template Library* (STL), utilizada no código-fonte do problema. Em (OPENACC-STANDARD.ORG, 2014) mais detalhes podem ser encontrados sobre suporte de estruturas complexas pelo OpenACC. Uma possível alternativa é avaliar o uso do padrão de programação C++17 com CUDA conforme indicado na publicação Olsen David; Lopez (2020).

Outra ferramenta que pode ser explorada é a biblioteca libcu++, conhecida também por *NVIDIA C++ Standard Library*. Algumas informações iniciais podem ser obtidas em Olsen David; Lopez (2019).

Além disso, como terceira alternativa a se considerar, pode-se citar o Edwards, Trott e

Sunderland (2014) que oferece um modelo de programação para C++ para escrita de aplicações portáveis com foco nas principais plataformas de alto desempenho.

E, por fim, conforme evidenciado pelas análises iniciais, o *Decoder* tomou bastante tempo de execução do A-BRKGA. A proposta de buscar otimizações de desempenho por meio de aceleração nessa trecho pode trazer resultados bem interessantes. O principal desafio é identificar estratégias que façam sentido, pois o *Decoder* é desenvolvido em função do tipo de problema.

Referências

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, v. 12, n. 3, p. 19–20, 2007. Citado na página 30.

BEAN, J. Genetic algorithms and random keys for sequencing and optimization. *Journal on Computing*, v. 6, p. 154–160, 1994. Citado na página 26.

BEN-ARI, M. *Principles of Concurrent and Distributed Programming (2Nd Edition) (Prentice-Hall International Series in Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 032131283X. Citado na página 28.

CHATAIN, P. L. Hybrid parallel programming - evaluation of openacc. 2012. Citado 2 vezes nas páginas 31 e 32.

CHAVES, A.; GONÇALVES, J.; LORENA, L. Adaptive biased random-key genetic algorithm with local search for the capacitated centered clustering problem. *Computers Industrial Engineering*, v. 124, 07 2018. Citado 5 vezes nas páginas 21, 22, 27, 28 e 37.

COLEY, D. *An Introduction to Genetic Algorithms for Scientist and Engineers*. [S.l.: s.n.], 2014. Citado 2 vezes nas páginas 25 e 26.

CONTE, D. Introdução ao cuda. 2017. Citado na página 29.

CORMEN, T. H. Introduction to algorithms. v. 3rd Edition, p. 1048–1128, 2009. Citado 2 vezes nas páginas 22 e 28.

EDWARDS, H.; TROTT, C.; SUNDERLAND, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, v. 74, 07 2014. Citado na página 58.

FAZENDA, ; STRINGHINI, D. Como programar aplicações de alto desempenho com produtividade. *XXXIX Congresso da Sociedade Brasileira de Computação*, 2019. Citado 5 vezes nas páginas 29, 30, 31, 39 e 54.

GONÇALVES, J.; RESENDE, M. Biased random-key genetic algorithms for combinatorial optimization. *J. Heuristics*, v. 17, p. 487–525, 10 2011. Citado na página 27.

HERLIHY, M.; SHAVIT, N. *The Art of Multiprocessor Programming*. [S.l.]: Morgan Kaufmann, 2006. ISBN B008CYT5TS. Citado na página 28.

LEANDRO, Z.; FERREIRA, A.; MATSUMOTO, M. Arquitetura e programação de gpu nvidia. 2012. Citado na página 31.

LISBOA, E. Pesquisa operacional. v. 1–4, 2002. Citado na página 25.

MALAQUIAS, N. Uso de algoritmos genéticos para a otimização de rotas de distribuição. p. 28–29, 2006. Citado 2 vezes nas páginas 22 e 25.

MIYAZAWA, F. K.; SOUZA, C. C. Introdução à otimização combinatória. *JAI-SBC*, p. 2–3, 2015. Citado na página 22.

MPI-FORUM.ORG. A message-passing interface standard. *MPI Forum Org*, v. 3, 2015. Citado na página 30.

OLSEN DAVID; LOPEZ, G. L. B. *The Cuda C++ Standard Library*. 2019. Disponível em: <https://on-demand.gputechconf.com/supercomputing/2019/pdf/sc1942-the-cuda-c++-standard-library.pdf>. Citado na página 57.

OLSEN DAVID; LOPEZ, G. L. B. *Accelerating Standard C++ with GPUs Using stdpar*. 2020. Disponível em: <https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/>. Citado na página 57.

OPENACC-STANDARD.ORG. Complex data management in openacc. *www.openacc.org*, 2014. Citado 2 vezes nas páginas 42 e 57.

OPENACC.ORG. The openacc: Application programming interface. *OpenACC-Standard.org*, 2015. Citado 7 vezes nas páginas 21, 32, 33, 35, 39, 40 e 42.

OPENACC.ORG. Openacc programming and best practices guide. *OpenACC-Standard.org*, 2015. Citado 2 vezes nas páginas 32 e 33.

OPENMP.ORG. Openmp application programming interface. *OpenMP Architecture Review Board*, v. 5, 2018. Citado na página 30.

PACHECO, P. S. An introduction to parallel programming. n. 1, 2011. Citado na página 28.

PGPROF. Pgi profiler user's guide. *www.pgroup.com*, 2018. Citado na página 35.

RESENDE, M. Introdução aos algoritmos genéticos de chaves aleatórias viciadas. *Simpósio Brasileiro de Pesquisa Operacional*, 2013. Citado na página 26.

Anexos

ANEXO A – Busca Local - Serial

```
TSol LocalSearch(TSol s)
{
    vector<vector<double> > AfoOpt(n, vector<double>(n));
    vector<long int> AMi(n);
    vector<long int> AMj(n);

    int melhorou = 1;
    int vi, viP, viM, vj, vjM, vjP;
    double foOpt = 0, temp_fo = 0;


    while (melhorou) {
        melhorou = 0;
        foOpt = 0;


        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 2; j < n; j++) {
                if (i != 0 && j != n - 1) {
                    vi = s.vec[i].sol;
                    viP = s.vec[i + 1].sol;
                    viM = 0;
                    if (i == 0)
                        viM = s.vec[n - 1].sol;
                    else
                        viM = s.vec[i - 1].sol;

                    vj = s.vec[j].sol;
                    vjM = s.vec[j - 1].sol;
                    vjP = 0;
                    if (j < n - 1)
                        vjP = s.vec[j + 1].sol;
                    else
                        vjP = s.vec[0].sol;

                    foOpt = - dist[viM][vi]
                            - dist[vi][viP]
                            - dist[vjM][vj]
                            - dist[vj][vjP]
                            + dist[viM][vj]
                            + dist[vj][viP]
                            + dist[vjM][vi]
                            + dist[vi][vjP];

                    AfoOpt[i][j] = foOpt;
                    AMi[i] = i;
                    AMj[j] = j;

                    temp_fo = 0;
```

```
                    for (int k = 0; k < n; k++) {
                        temp_fo += dist[s.vec[k % n].sol][s.vec[(k + 1) % n].sol];

                    }
                }
            }
        }

        double Bestfo = AfoOpt[0][0], Besti = 0, Bestj = 0;

        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 2; j < n; j++) {
                if (AfoOpt[i][j] < Bestfo) {
                    Bestfo = AfoOpt[i][j];
                    Besti = i;
                    Bestj = j;
                }
            }
        }

        foOpt = Bestfo;

        if (foOpt < 0) {
            int Mi = AMi[Besti];
            int Mj = AMj[Bestj];

            // first improvement strategy
            TVecSol aux;

            // exchange i and j
            aux = s.vec[Mi];
            s.vec[Mi] = s.vec[Mj];
            s.vec[Mj] = aux;
            s.fo = s.fo + foOpt;

            melhorou = 1;
        }
    }

    // save the best solution found in this run
    if (s.fo < bestSolution.fo) {
        bestSolution = s;
    }

    return s;
}
```

ANEXO B – Busca Local - Otimizada

```
TSol LocalSearch(TSol s)
{
    int melhorou = 1;
    int vi, viP, viM, vj, vjM, vjP, Besti, Bestj;
    double foOpt = 0, Bestfo = 0, temp_fo = 0;

    TVecSol* stemp_vec = &s.vec[0];
    double stemp_fo = s.fo;

    #pragma acc data \
        pcopy(stemp_vec[0 : n], stemp_fo) \
        pcreate(Bestfo, Bestj, Besti) \
        present(temp_dist[0:n][0:n], AfoOpt[0:n][0:n], AMi[0:n], AMj[0:n])
    {
        while (melhorou) {
            melhorou = 0;
            foOpt = 0;

            #pragma acc parallel loop collapse(2) \
                private(foOpt, vi, viP, viM, vj, vjM, vjP, temp_fo)
            {
                for (int i = 1; i < n - 1; i++) {
                    for (int j = 3; j < n - 1; j++) {
                        if (j >= i + 2) {
                            vi = stemp_vec[i].sol;
                            viP = stemp_vec[i + 1].sol;
                            viM = 0;
                            if (i == 0)
                                viM = stemp_vec[n - 1].sol;
                            else
                                viM = stemp_vec[i - 1].sol;

                            vj = stemp_vec[j].sol;
                            vjM = stemp_vec[j - 1].sol;
                            vjP = 0;
                            if (j < n - 1)
                                vjP = stemp_vec[j + 1].sol;
                            else
                                vjP = stemp_vec[0].sol;

                            foOpt = - temp_dist[viM][vi]
                                    - temp_dist[vi][viP]
                                    - temp_dist[vjM][vj]
                                    - temp_dist[vj][vjP]
                                    + temp_dist[viM][vj]
                                    + temp_dist[vj][viP]
                                    + temp_dist[vjM][vi]
                                    + temp_dist[vi][vjP];

```

```
                            AfoOpt[i][j] = foOpt;
                            AMi[i] = i;
                            AMj[j] = j;

                            temp_fo = 0;
                            #pragma acc loop independent reduction(+: temp_fo)
                            for (int k = 0; k < n; k++) {
                              temp_fo +=
                                temp_dist[stemp_vec[k%n].sol][stemp_vec[(k+1)%n].sol
    ];
```

```
                            }
                        }
                    }
                }
            }

            #pragma acc serial
            {
                Bestfo = AfoOpt[0][0];
                Besti = 0;
                Bestj = 0;
            }


            #pragma acc parallel loop collapse(2) reduction(min : Bestfo)
            for (int i = 1; i < n - 1; i++) {
                for (int j = 3; j < n - 1; j++) {
                    if (j >= i + 2) {
                        if (AfoOpt[i][j] < Bestfo) {
                            Bestfo = AfoOpt[i][j];
                        }
                    }
                }
            }


            #pragma acc parallel loop collapse(2)
            for (int i = 1; i < n - 1; i++) {
                for (int j = 3; j < n - 1; j++) {
                    if (j >= i + 2) {
                        if (AfoOpt[i][j] == Bestfo && (i >= Besti && j >= Bestj)) {
```

```
                            #pragma acc atomic write
                            Besti = i;
                            #pragma acc atomic write
                            Bestj = j;
                        }
                    }
                }
            }

            #pragma acc serial
            {
                foOpt = Bestfo;

                if (foOpt < 0) {
```

```
104                     int Mi = AMi[Besti];
105                     int Mj = AMj[Bestj];
106 
107                     // first improvement strategy
108                     int aux_sol = stemp_vec[Mi].sol;
109                     double aux_rk = stemp_vec[Mi].rk;
110 
111                     // exchange i and j
112                     stemp_vec[Mi] = stemp_vec[Mj];
113                     stemp_vec[Mj].sol = aux_sol;
114                     stemp_vec[Mj].rk = aux_rk;
115                     stemp_fo = stemp_fo + foOpt;
116 
117                     melhorou = 1;
118                 }
119             }
120         }
121     }
122 
123     s.fo = stemp_fo;
124     vector<TVecSol> vec_values(stemp_vec, stemp_vec + n);
125     s.vec = vec_values;
126 
127     // save the best solution found in this run
128     if (s.fo < bestSolution.fo) {
129         bestSolution = s;
130     }
131 
132     return s;
133 }
```

ANEXO C – Alocação de memória

```
1  void mallocAcc() {
2      AfoOpt = (double **)malloc(n * sizeof(double *));
3      temp_dist = (double **)malloc(n * sizeof(double *));
4      AMi = (int *)malloc(n * sizeof(int));
5      AMj = (int *)malloc(n * sizeof(int));
6
7      for (int i = 0; i < n; i++) {
8          AfoOpt[i] = (double *)malloc(n * sizeof(double));
9          temp_dist[i] = &dist[i][0];
10     }
11
12     #pragma acc enter data copyin(temp_dist[0:n][0:n]) \
13         create(AfoOpt[0:n][0:n], AMi[0:n], AMj[0:n])
14 }
```

ANEXO D – Desalocação de memória

```
void deallocAcc() {
    for (int i = 0; i < n; i++) {
        free(AfoOpt[i]);
    }
    free(AfoOpt);
    free(temp_dist);
    free(AMi);
    free(AMj);
    #pragma acc exit data \
        delete(temp_dist[0:n][0:n], AfoOpt[0:n][0:n], AMi[0:n], AMj[0:n])
}
```